Enterprise Information Systems, 2013
Vol. 7, No. 2, 227–236, http://dx.doi.org/10.1080/17517575.2012.744854

Taylor & Francis
Taylor & Francis Group

Sisal 3.2: functional language for scientific parallel programming

Victor Kasyanov*

Institute of Informatics Systems, Lavrentiev pr. 6, Novosibirsk, 630090, Russian Federation

(*Received 7 August 2012; final version received 25 October 2012*)

Sisal 3.2 is a new input language of system of functional programming (SFP) which is under development at the Institute of Informatics Systems in Novosibirsk as an interactive visual environment for supporting of scientific parallel programming. This paper contains an overview of Sisal 3.2 and a description of its new features compared with previous versions of the SFP input language such as the multidimensional array support, new abstractions like parametric types and generalised procedures, more flexible user-defined reductions, improved interoperability with other programming languages and specification of several optimising source text annotations.

Keywords: functional programming; dataflow languages; parallel programming; scientific computations

1. Introduction

Using traditional methods, it is very difficult to develop high-quality, portable software for parallel computers. In particular, parallel software for supporting of enterprise information systems cannot be developed on low-cost, sequential computers and then moved to high-performance parallel computers without extensive rewriting and debugging.

Functional programming (Backus 1978) is a programming paradigm which is entirely different from the conventional model: a functional program can be recursively defined as a composition of functions where each function itself can be another composition of functions or a primitive operator (such as arithmetic operators, etc.). The programmer need not be concerned with explicit specification of parallel processes since independent functions are activated by the predecessor functions and data dependencies of the program. This also means that control can be distributed. Furthermore, no central memory system is inherent to the model since data are not 'written in' by an instruction but is 'passed from' one function to the next. However, scientific world is conservative and the Fortran programming language is still quite popular in scientific computations for supercomputers.

The functional language Sisal (Steams and Iterations in a Single Assignment Language) (Gaudiot *et al.* 2001) was created by collaborating teams from the Lawrence Livermore National Laboratory, Colorado State University, the University of Manchester and Digital Equipment Corporation as an alternative to

*Email: kvn@iis.nsk.su

Fortran for parallel programming (Cann 1992). In contrast with other functional languages, Sisal supports data types and operators typical for scientific computations. At present, there are implementations of the Sisal 1.2 (McGraw *et al*. 1985) language for many supercomputers (e.g. SGI, Sequent, Encore Multimax, Cray X-MP, Cray 2, etc). The Sisal 2.0 (Cann *et al*. 1991) and Sisal 90 (Feo *et al*. 1995) language definitions increase the language's utility for scientific programming. They include the language level support for complex values, array and vector operations, higher order functions, rectangular arrays and an explicit interface to other languages like Fortran and C.

A system of functional programming (SFP) (Kasyanov *et al*. 2006) being under development at the Institute of Informatics Systems is intended to be an interactive visual environment on personal computer for supporting of parallel scientific programming. The SFP uses intermediate program presentations based on hierarchical graph models (Kasyanov and Lisitsyn 2000) and provides means for writing and debugging the Sisal-programs regardless target architectures as well as for translating the Sisal-programs into optimised imperative programs, appropriate to the target execution platforms. The Sisal 3.0 (Kasyanov *et al*. 2001) and Sisal 3.1 (Stasenko and Sinyakov 2006) languages that have been designed as the input language of the SFP are based on the Sisal 90 and include language level support for module design, mixed language programming, preprocessing and annotated programming (Kasyanov 1991).

The Sisal 3.2 programming language (Kasyanov and Stasenko 2007) here considered is a new input language of the SFP. It is a successor of Sisal 3.1 (Stasenko and Sinyakov 2006) and Sisal 3.0 (Kasyanov *et al*. 2001) languages and integrates features of Sisal 2.0 (Cann *et al*. 1991) version. This paper contains an overview of Sisal 3.2 and a description of its new features compared with its predecessors such as the multidimensional array support, new abstractions like parametric types and generalised procedures, more flexible user-defined reductions, improved interoperability with other programming languages and specification of several optimising source text annotations.

The rest of the paper is organised as follows. Section 2 describes the general features of Sisal language. Multidimensional arrays that came from Sisal 2.0 are presented in Section 3. New language abstractions such as parametric types and generalised functions and operations are described in Section 4. A new way to specify user-defined reductions introduced in Sisal 90 is considered in Section 5. Section 6 describes how Sisal 3.2 programs can interoperate with other programming languages. Section 7 describes existing optimising annotations in a form of pragma statements.

2. Overview of Sisal 3.2 language

2.1. Program structure

A Sisal 3.2 program consists of one or more separate compilation units called *modules*. Each module consists of *definition* and *declaration* files. A module declaration file corresponds to one module definition and each module definition cannot have more than one module declaration.

A Sisal module contains definition and declaration of procedures (functions and operations), types and contract definitions. A module declaration contains procedure declarations which are defined by the corresponding module definition and are

visible outside it. In addition, the module declaration contains externally visible declarations and definitions of types and contracts.

Any function in any module may be a starting point of program execution. At this outermost level, the function parameters are the values obtained at the operating system level and the function results are produced at that level.

Since Sisal compilers can translate Sisal programs into the C programming language, all Sisal function definitions have corresponding C language equivalent definitions which, in turn, have corresponding declarations in C, which allows software written in another language to have subsidiary parts written in Sisal. Special foreign module declarations (see Section 6) declare the relationship between Sisal and a set of subsidiary code written in other languages. This allows Sisal software to have access to libraries of already written code.

2.2. Types

Data types include the usual scalar types (Boolean, character, integer, real and double), structured types (records and unions, arrays and streams) and functions. Structured types may have values of any type as components; records and unions have heterogeneous components and arrays and streams have homogeneous components. Unions can be recursive like in the following example:

```
type list[T] := union [ empty; item: record [value: T; next: list]]
```

The language supports user-defined types with their custom operations, thus for example allowing a programmer to implement complex number types. This is an example of a definition of a complex number type and its additive operation:

```
type compl := record[ real_part, imag_part: real]
operation + (compl, compl returns compl)
```

A module declaration may specify the name of a record or union type for public use but may prevent export of the components.

In the previous versions of Sisal, a type may be declared to be one of the set of alternatives. This is useful for writing functions with the formal parameter types not given concretely. A function reference will supply known actual parameter types, which are used to complete the compilation of this instance of the function. This facility for *typesets* was used to simplify the production of code that operates on different arithmetic types. However, in Sisal 3.2 version, typesets were replaced by a parametric types described in Section 4 because they provide more flexibility and do not require delayed compilation. Functionality of recursive typesets can be specified via recursive unions.

The function values may be parameters of functions and the results of expression evaluation, so the function types may be declared by specifying the types of all parameters and results. Therefore Sisal does not use a complete type inference system wherein the types of all values are inferred from their contexts. As a result, complete compile-time typing is possible for all Sisal programs. For example,

```
type fun1 = function [real, real returns real, real]
```

is a declaration of the function type of two real values and two real results.

2.3. Functions

A function is declared by listing its name, the names and types of its formal parameters and types of its resulting values. The content of a function is one or more expressions (a *multi-expression*) whose type corresponds to types of its results. The values are available to the expressions via formal parameters, not through globally accessed names.

The higher order function operations are part of Sisal 3.2. Functions can be passed to and returned from functions and be the values of expressions.

2.4. Expressions

Expressions are, of course, the heart of the language. Syntax is designed to be as close as possible to most standard procedural languages like Pascal.

Conventional infix operations combine scalar arithmetic values. Sisal supports some type promotion automatically and provides some predefined type conversion functions.

One can assign the value of any expression to a name and use the name as a shorthand for the expression throughout the scope of the definition. This scoping is done with the **let** construct. For example, multi-expression **let** X := 3.0; A := X * G **in** A * X, X **end let** equals to 3.0 * G * 3.0, 3.0 and expression **let** A := 3 **in let** A := A + 1 **in** A **end let end let** equals to 4.

Sisal has comprehensive facilities for defining and manipulating the array values. An array generator allows the definition of a multidimensional object whose parts form a 'tiling' of the overall structure. Arbitrary subarray selection is provided beyond the rectangular subsets available in some other notations. Many infix operations operate element-by-element on array operands and a useful set of functions on arrays is defined. A subarray update facility allows safe alteration of array values. Many applications are expressible succinctly with these features. Array generation, selection and update may use vector subscripts to refer to arbitrary, nongeometric sections of arrays.

A stream is a sequence of values produced in some order by one expression evaluation and consumed in the same order by one or more other expression evaluations. Producers and consumers are usually **for** expressions but short forms for simple streams are also available. To expose the pipelined parallelism that streams make possible, they should be implemented non-strictly. That is consumer expressions should be started whether or not the producer expression has finished.

Two constructs for selection are provided in Sisal: **if** and **case** expressions. The results of **if** expression are guarded by Boolean expressions, while the **case** expression is guarded by the values of the selecting expression. The arms of a single **if** or **case** expression should agree in arity and type unless the selection is being used for type inquiry.

A single **for** construct has two forms for potentially parallel, as well as sequential, evaluation. In the first form, values are distributed to the bodies of the construct and each body defines values to contribute to the overall result. The second form has dependencies (determined by **old** keyword before loop value name) between the values defined in one body and used in the successor's body. In either form, the values from the bodies are collectable into an array or a stream or reducible to a single value. Array construction in **for** expressions allows permutation of the individual body values. For example, the expression **for** App := 1.0; Sign := 1.0; Den

:= 1.0; i := 1 **while** i < = Cycles **do** Sign := - **old** Sign; i:= **old** i + 1; Den := **old** Den + 2.0; App := **old** App + Sign/Den **returns** value **of** App * 4.0 **end for** computes π value iteratively. The expression **for** i **in** 1..Cycles/2 **do** val := 1.0/(4*i-3):real–1.0/(4*i-1):real **returns** sum **of** val **end for** * 4.0 also computes π value but can do it in parallel since there is no **old** keywords used.

2.5. Errors

Sisal includes a standard error processing semantics for managing erroneous computations. However, a Sisal implementation may elect to stop execution when an error is encountered.

Each Sisal type has a distinguished value, **error**. Any failed expression evaluation results in **error** of the appropriate type. Error values propagate in a well-defined way when they are operands in computations. Error values can be tested and even explicitly assigned to signify other anomalous conditions.

3. Multidimensional arrays

Sisal language was designed to describe scientific computations; so, after analysis of the features of other languages with scientific orientation, such as Fortran and Sisal 2.0, it was decided to introduce[1] multidimensional arrays and arrays with fixed form into the Sisal 3.2 language as well as extended means for their construction.

The array type is described as '*array array form of array element type*'. An array form can be of free '[*list of double dots*]' or fixed '[*list of duplets*]'. A duplet is a construction that looks like '*lower bound .. upper bound*', where *lower* and *upper bounds* are unary expressions of integer type. The lower bound can be omitted and it is assumed to be equal to one by default. The upper bound should be more or equal to the lower bound. The array form can be omitted and is assumed to be '[..]' by default. The array dimension is determined by a dimension of the array form which equals to the number of double dots or duplets in it. Here are some examples:

type Arr1 = **array** [2..5] **of** integer

type Arr2 = **array of** integer

type Arr3 = **array** [.., ..] **of** integer

type Arr4 = **array** [..2, ..3] **of** integer

The array value constructors[2] were extended to cope with arrays of a fixed form: for example, an array of integers [1, 2, 3] can be constructed by **array** [1..3] **of** [1 := 1; 2 := 2; **else** := 3], and two-dim array [[1, 0, 3], [4, 0, 6]] can be constructed by **array** [1..2, 1..3] **of** [1,1 := 1; 1,3 := 3; 2,1 := 4; 2,3 := 6; **else** := 0].

To ease construction of large arrays, Sisal 3.2 supports range-based element specification. For example, the range of elements can be specified by a list of values like in array [1, 2, 3] constructed by **array** [1..3] **of** [1..3 := 1, 2, 3], the values can also be taken from other array like in array [[1, 2, 3], [4, 5, 6]] constructed by **array** [..2, ..3] **of** [1, .. := [1, 2, 3]; 2, .. := [4, 5, 6]], a one-element list fills the whole range like in the array [[1, 1, 1], [1, 1, 1]] constructed by **array** [..2, ..3] **of** [.., .. := 1].

Sisal 3.2 supports specification of ranges with dimensions tied by **dot** operator[3]: **array** [..3, ..3] **of** [.. **dot** .. := 1, 2, 3; **else** := 0] constructs the array [[1, 0, 0], [0, 2, 0], [0, 0, 3]]. The dimension range indices can be named and reused later even in the same range (making it triangular): **array** [..3, ..3] **of** [i **in** .., i .. := 1; **else** := 0] constructs [[1, 1, 1], [0, 1, 1], [0, 0, 1]] and **array** [..3, ..3] **of** [i **in** .. **dot** j **in** .. := i + j; **else** := 0] constructs [[2, 0, 0], [0, 4, 0], [0, 0, 6]]. The array ranges can be also specified by integer arrays of indices: for example, **array** [..2, ..3] **of** [1, [3, 2] := 1, 2; 2, [1, 2] := 4, 5; **else** := 0] constructs array [[0, 1, 2], [4, 5, 0]].

At last, there are some additional examples that clarify error handling in array constructors. Nonspecified array elements equal to error values: for example, **array** [1..3] **of** integer [:= 1, 2], constructs the array [1, 2, error[integer]] and **array** [1..3] **of** [2:= 1; 3 := 2] constructs the array [error[integer], 1, 2].

An ambiguous definition of array elements leads to the erroneous array: for example, **array** [1..3] **of** [1 := 1; 2 := 2; 3 := 3; 1 := 1] constructs the array error[array [1..3] of integer] and **array** [1..3] **of** [2 := 1; 3 := 2] constructs the array [error[integer], 1, 2].

4. New language abstractions

To increase the level of its algorithmic abstractions, Sisal 3.2 was augmented by new concepts of parametric types, contracts and generalised procedures (functions and operations). A parametric type defines a set of types that allows finer control compared to already existing typesets.[4] A contract is another form of abstraction that allows us to bind a set of operations over types listed as contract parameters to contract name. Contracts are used in generalised procedures to specify what kind of operations are their parametric types expected to have.

For example, let us define a parametric type of matrix over any element: **type** matrix[T] := **array** [.., ..] **of** T. For a matrix multiplication operation, which we are going to declare, a matrix element type should support the addition and multiplication operations, so we define a contract with the name 'additive' for all such types:

contract additive[T] **operation** + (T , T **returns** T) **operation** * (T , T **returns** T) **end contract**.

And now we can declare a generalised operation of our matrix multiplication:

operation * **of** additive[T] (matrix[T], matrix[T] **returns** matrix[T]).

5. Reusable user-defined reductions

In Sisal 90, a user-defined reduction was a function definition of a very special form that is used to transform loop values into loop results. Because of its special form, a reduction function cannot be reused outside loops. In Sisal 3.2, user-defined reductions are defined as a combination of several usual functions thus allowing them to be reused.

A general form of reduction invocation in a loop return statement looks as follows:

'**reduction** *name N* (*list of initial values*) **of** (*list of loop values*)'

where the initial values of reduction should be loop constants. The reduction name *N* corresponds to functions with prototypes described below.

The first function (function *I*) computes an initial reduction state in a type *T* which is any type that can hold a reduction internal state: **function** *N* (*types of initial reduction parameters* **returns** *T*).

The following function (function *L*) recomputes the reduction state using loop values of the subsequent loop iteration: **function** *N* (*T, types of loop reduction parameters* **returns** *T*).

The following function (function *J*) determines, how some two reduction states (obtained after a parallel loop execution) can be merged. This function can be omitted if the reduction does not allow such things: **function** *N* (*T, T* **returns** *T*).

The last function (function *R*) computes the reduction results from its internal state: **function** *N* (*T* **returns** *types of return reduction values*).

6. Improved interoperability

The Sisal 3.2 language extends Sisal 2.0 language by adding support of foreign language functions from already written programs and libraries (written in C/C++ or Fortran). A foreign function support is based on the concept of foreign types.

Foreign types in Sisal 3.2 are specified by their string representation in their native programming language. The values of foreign types are constructed via foreign operations and functions written in a foreign programming language and use a special interface to access the values of Sisal 3.2 types if necessary.

Foreign types have no built-in operations, so a Sisal program should define them to make use of them. If a foreign type *T* has a defined operation '**operation** (*T* **returns** *T*)', then this operation is used to create a foreign type value copy.

For a foreign type *T*, one can prohibit a copy operation at all by using 'no **operation** (*T* **returns** *T*)' declaration. If a copy operation is not defined and not prohibited, then a bit-by-bit copy approach is used. If a foreign type *T* has a defined operation '**operation** (*T* **returns** null)', then it is used to free copy of the foreign type *T*; otherwise, no additional actions are performed when a foreign type *T* value is no longer needed. An error value of a foreign type *T* corresponds to it undefined value unless '**operation** (null **returns** *T*)' operation, which returns an error value of the foreign type, is defined.

A declaration of a foreign function looks like '**function** *function name* (*list of formal parameters* **returns** *the type of the return value*)', where at least one formal parameter type or the return value type belongs to a typeset *S*. The typeset *S* contains foreign types, user-defined types based on the types from *S*, arrays with elements of a type from the typeset *S*, and records and non-recursive unions based on types from the typeset *S*. Formal parameters of foreign function can be prefixed by **in, out** and **raw** keywords and the type of the return value can be prefixed by **out** and **raw** keywords.

The keyword **in** can prefix any type *T* from the typeset *S*. The keyword **in** means that a foreign function receives a pointer to a dynamic memory that contains a copy of a value of the type *T*. The keyword **out** can be used together with keyword **in** for the types from a typeset S_2. The typeset S_2 contains foreign types, arrays with elements of type from the typeset S_2, records based on types from the typeset S_2. The keyword **out** means that upon return from a foreign function call a new return value

is formed from the dynamic memory pointed by the pointer passed to the foreign function. If a foreign function has at least one keyword **out** (with or without keyword **in**) then the return statement (and the type of the return value) can be omitted.

The keyword **out** without the keyword **in** can prefix a type T from a typeset S_3. The typeset S_3 contains foreign types, fixed form arrays with elements of a type from the typeset S_3 and records based on types from the typeset S_3. The keyword **out** before a type T means that a foreign function receives a pointer to a dynamic, not initialised memory of size enough to hold a value of the type T and, upon return from the foreign function call, a new return value is formed from the dynamic memory pointed by the pointer passed to the foreign function. For the keyword **out** without the keyword in}, the corresponding argument of a foreign function should be omitted.

The keyword **raw** can prefix a type T from a typeset S_4. The typeset S_4 contains foreign types, records and non-recursive unions based on types from the typeset S_4.

The keyword **raw** for a foreign type can be used only together with the keyword **in** and it means that this foreign type should be considered as a pointer and it should be passed to the foreign function without any additional indirection.[5] The keyword **raw** for a record or union means that it will be passed to a foreign function without any extra information that comes with normal Sisal types (such as the error value information).

7. Optimising annotations

The Sisal 3.2 language supports annotated programming (Kasyanov 1991) and specifies optimising annotations in the form of so-called *pragma statements* being predicate constraints on admissible states of computations.

A pragma statement is a special form of a program comment that starts with dollar sign '$'.

Every expression can be prefixed by a pragma 'assert = Boolean expression', that can be checked for truth after the expression evaluation during program debugging and then can be used in program optimising transformations. The result of the expression can be denoted as the underscore symbol '_' and if the expression is n-ary (where n >1), then its components can be denoted as an array with the name '_': '_[1]', ..., '_[n]'. In addition, the pragma 'assert = Boolean expression' can be placed before returns keyword in procedure declarations and can be used to control results of this procedure after its invocation. As an example of the assert pragma statement usage please consider this factorial function declaration and definition:

```
forward function fact (n: integer
/*$ assert = n > = 1*/ /*$ assert = _ > = n*/
returns integer)
function fact (n: integer returns integer)
if n = 1 then 1
else /*$assert = _ > 0*/ fact(n-1)*n end if
end function
```

Another pragma 'parallel' can be used before a case expression in Sisal (analogous to a switch expression in C language). This pragma can be specified if

it is known that only one test can be true. The pragma of the form 'parallel = Boolean expression' means that only one test is true if the specified Boolean expression is true.

Functions that are used to form a reduction value (see Section 5) can be associated with special pragmas that can be used in parallel loops more efficiently. A function I that computes an initial reduction state can be marked by the pragma 'identity' if it specifies an identity value, relative to functions L and J, that merges the reduction states of a type T: $J(t, L(I(I_1), L_1)) = L(t, L_1) = t$, where t is in T, I_1 is the initial values and L_1 is loop values. If the function I is not marked by the identity pragma then it does not make much sense to define the function J because it would not be able to correctly merge reduction states. The function J marked by the 'associative' pragma is assumed to be associative: $J(J(a, b), c) = J(a, J(b, c))$. The function J marked by the 'commutative' pragma is assumed to be commutative: $J(a, b) = J(b, a)$. If the function J is associative then it may be used in parallel computation of reduction values.[6]

If the function J is associative and commutative then it can be used for a potentially more effective asynchronous parallel computations of the reduction values.

A declaration of a foreign procedure can be prefixed by a pragma 'weight = integer expression', where the value of the integer expression defines an approximate number of cycles of some abstract machine required to execute this procedure. A weight ratio is used for better load balancing of foreign procedures between several processing units.

8. Conclusion

The Sisal 3.2 language is a significant improvement over previous Sisal 3.1 version. Sisal 3.2 became closer to modern programming languages after the introduction of multidimensional arrays, new type and algorithmic abstractions, improved inter-operability with other programming languages and specification of optimising annotations. As a result, both the Sisal 3.2 language and the SFP became more convenient for scientific parallel programming.

Acknowledgements

The author is thankful to all colleagues for taking part in the SFP project. This research was supported in a part by the Russian Foundation for Basic Research (grant RFBR N 07-07-12050).

Notes

1. Sisal 3.1 has multidimensional arrays in a form of simply nested arrays.
2. Since Sisal is a functional language, all its values including arrays are constructed at once and, for example, any array element update conceptually leads to creation of a new array value.
3. The operator **dot** takes two dimension indices $A_{1..m}$ and $B_{1..n}$ that normally a Cartesian product of indices and produces a sequence of indices $(A_1, B_1), (A_2, B_2), \ldots, (A_s, B_s)$, where $s=min\ (n, m)$.
4. A classic typeset cannot define the set of records with two fields of equal type, while this is possible with a parametric type. In addition, type names in parametric type are used in generalised procedures and their contract attachments.

5. The keyword **raw** actually makes sense only when the keywords **in** and **out** are used together because keywords **raw** and **in** do not have any effect without keyword **out** and can be omitted.
6. Therefore, it does not make much sense to define a non-associative function *J*.

References

Backus, J., 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21 (8), 613–641.

Cann, D.C., 1992. Retire Fortran?: a debate rekindled, *Communications of the ACM*, 34 (8), 81–89.

Cann, D.C., *et al*., 1991. *Sisal reference manual: language version 2.0*. Technical Report UCRL-MA-109098. Livermore, CA: Lawrence Livermore National Laboratory.

Feo, J.T., *et al*., 1995. *Sisal 90 user's guide*. *Draft 0.96*. Livermore, CA: Lawrence Livermore National Laboratory.

Gaudiot, J.-L., *et al*., 2001. The Sisal project: real world functional programming. *Lecture Notes in Computer Science*,1808, 45–72.

Kasyanov, V.N., 1991. Transformational approach to program concretization. *Theoretical Computer Science*, 90 (1), 37–46.

Kasyanov, V.N., Biryukova, Yu.V., and Evstigneev, V.A., 2001. Functional language Sisal 3.0, *In*: Kasyanov, V.N., ed. *Supercomputing support and Internet-oriented technologies*. Novosibirsk: IIS. pp. 54–67 (In Russian)

Kasyanov, V.N. and Lisitsyn, I.A., 2000. Hierarchical graph models and visual processing. *In*: *Proceedings of international conference on software: theory and practice*. *16th IFIP World Computer Congress*, Beijing: PHEI, 179–182.

Kasyanov, V.N. and Stasenko, A.P., 2007. Sisal 3.2 programming language. *In*: Kasyanov, V.N., ed. *Tools and techniques of program construction*. Novosibirsk: IIS, 56–134 (in Russian)

Kasyanov, V.N., *et al*., 2006. SFP – an interactive visual environment for supporting of functional programming and supercomputing. *WSEAS Transactions on Computers*, 9, 2063–2070.

McGraw, J.R., *et al*., 1985. *Sisal: streams and iterations in a single assignment language, language reference manual, version 1.2*. Technical Report M-146, Rev. 1. Livermore, CA: Lawrence Livermore National Laboratory.

Stasenko, A.P. and Sinyakov, A.I., 2006. Basic features of Sisal 3.1 programming language. *Preprint of IIS*, 132, 56 (in Russian).