UDC 004.434

THE PROGRAMMING LANGUAGE FOR EMBEDDED REAL-TIME DEVICES WITH REDUCING ERRORS AND WITHOUT REDUCING THE PERFORMANCE OF PROGRAMS

Oleksii Shmalko, Pavlo Rehida, Artem Volokyta, Heorhii Loutskii
Department of Computer Engineering of National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine

Vu Duc Thinh
Faculty of Information Technology of Ho Chi Minh City University of Food Industry
Ho Chi Minh, Vietnam

Background. C or C ++ languages are most commonly used for programming of embedded systems. The main drawbacks are: lack of updates, difficulty in use, limited backward compatibility and potentially a large number of possible programmer errors. Therefore, it is important to provide developers of low-level software of embedded systems, operating systems and system utilities with fast, productive, reliable and stable language on the basis of modern programming theory.

Objective. The aim of the paper is to create a new productive and reliable programming language for embedded systems using the principles and approaches of modern programming theory.

Methods. Analyzing well-known publications devoted to programming languages used for embedded systems allowed identifying their main advantages and disadvantages. Comparing of modern approaches to the implementation of programming languages allowed determining the requirements for the developed language.

Results. A new programming language for embedded devices has been developed. The following compiler modules are described: lexer, parser, semantic analyzer, intermediate code generator. A detailed description of the developed programming language is presented.

Conclusions. In this paper, we propose to use a new programming language for embedded devices. An analysis of existing programming languages and typical developer errors was performed to ensure the reliability of the proposed language.

Keywords: programming languages; embedded systems; real-time systems.

I. Introduction

Nowadays, C [1] is the most widely used language for programming embedded systems. It is known for its complexity and makes it difficult to fix mistakes with high risk. The most common standard is the C99, released in 1999. This means that the language practically stopped its evolution and hasn't received significant improvement over the last 18 years. Language C ++, by contrast, actively developed and published for regular updates every three years. This language is limited by backward compatibility. This fact does not allow fixing most of the bugs that have been inherited from the language C. C ++ is difficult to use. Also, it does not have protection from a large class of errors. The authors do not know the modern language that fully satisfies their needs now. The aim of the paper is to develop a programming language designed for low-level programming: embedded systems, real-time systems, operating systems, system utilities. It should include achievements of the modern theory of programming languages, to be productive, fast and safe to use.

II. Recent research and publications analysis

A. C language

The latest standard (C11) was released in 2011. It provides direct memory access, almost complete control over memory and execution flow has no garbage collector or any runtime library. C language is the de facto standard for writing embedded systems. It has shown itself to be the best in the class of embedded systems. One of the major problems of C is typing. C has a weak static typing. This means that all types are checked at the compilation step. There is an implicit drive type, which often leads to program errors. The system of types is very primitive and does not have useful functions that are characteristic of more modern systems of types.

Idris language

Idris [2] language (2009) is a functional programming language with a powerful system of types [3]. It supports dependent types. Dependent types are a very powerful mechanism for verifying certain assumptions in the self-instruction code and the mathematical proof of the correctness of the program. Unfortunately, Idris requires a garbage collector and a large runtime library. This makes it impossible

to use Idris directly to create software for systems with limited memory. But it's possible to create an Idris-based EDSL (embedded domain-specific language) to generate code for embedded systems. This will allow you to get rid of these language restrictions. Similarly, the Atom [4] and Ivory [5] languages were implemented but based on the Haskell [6] language.

Nim language

The language Nim (2008) [7] is a multiparadigm language that is compiled in Javascript or C. Nim has a simplified system of effects. Nim has a large number of compiler directives [8] that may be of interest during low-level programming. It allows controlling the transfer of function arguments (by value, by reference), specify the hints of the optimizer (loop scan, built-in function calls, notation for non-return functions or no side effects), precisely control the memory scheme (bit fields, field alignment, stack frame generation), turn off the garbage collection if it needed. Unfortunately, while more detailed analysis was revealed serious shortcomings in the implementation of the compiler. Pragma volatile was implemented incorrectly [9], which made it impossible to use normal language for low-level programming.

Rust language

Rust (2010) [10] is a young multiparadigm compiling language [11]. Language focuses on secure and multithreaded programming and has a number of features for this task. The main feature of this language is a powerful type system. Typing at Rust is static and strong, which avoids a wide range of errors caused by implicit type-drives. Rust provides many features that are required for low-level programming: precise memory management, good C language interaction, no garbage collection, and runtime libraries. The Rust Compiler uses LLVM [12] to compile to generate efficient code for a large number of platforms. Rust also has a syntax that is very similar to C language. This is a plus for widespread language, as this syntax is known to many programmers.

Result of Analysis

Language C is a good choice for a commercial project but does not offer any technical benefits. The use of this language is not novel and not interesting in terms of research. Idris has one of the most powerful system of types. Its features could be used to create significantly more reliable operating systems. Unfortunately, Idris requires a garbage collector and a large runtime library. It makes impossible to use Idris for operating systems and embedded devices directly. Developing an EDSL on Idris can be an issue, but it expensive. Nim is a very interesting language that has an optional garbage collector and can work without a library of performance environments. It has a lot of features that can help with writing low-level software. However, the low quality of the implementation of the compiler almost makes it impossible to use language for more or less serious low-level programming. The Rust type system is a good compromise between a powerful Idris type system. It requires some support from the runtime and a system of types in C. Types in Rust help to avoid a certain class of errors. Splitting into safe and dangerous operations allows you to localize the code, which can cause program errors.

III. PROBLEM STATEMENT

Programming for embedded systems imposes the fairly large number of restrictions [13]. There are very few programming languages that can be used for these purposes. That is why it is necessary to determine the main limitations and requirements.

Features of the environment

Consider the features of the environment on the example of microcontrollers ARM Cortex-M [14] series and their implementation from STMicroelectronics [15].

- Low RAM volume. The volumes of RAM comprise from 2 kilobytes to 1 megabyte.
- Low ROM volume. The volumes of the built-in flash memory range from 8 kilobytes to 2 megabytes.
- Limited power. Many built-in systems are powered by batteries, keeping the processor in sleep mode most of the time (literally 99%) [16].
- Programmers are cheap; hardware is expensive. Usually, it is cheaper to use additional time programmers to optimize the code than to choose a more powerful chip [17].
- Predictability. Many built-in systems have soft or hard real-time requirements. It means that the system is obliged to provide a response within a limited time frame. In practice, this means that the use of many planners, garbage collectors, memory allocators is prohibited;
- Safety. The system should be reliable, because otherwise people might die. For this reason, there are many standards, such as MISRA [18].
- Debugging software errors is difficult. Embedded devices do not have the means to protect against improper use of memory.
- Compilation without runtime. Language obliged to compile. Language can support several forms of execution. For example, it can simultaneously support compilation in machine code, JVM virtual machine code, JavaScript, as well as Vimscript, Lisp, and Whitespace.

The main goals and principles of language

The main focus of developed languages is reducing the number of programming errors that do programmers and saving the performance and other critical features for system programming.

Safety. Language should prevent and make it impossible to do a large number of typical mistakes.

Simplicity and clarity. The language developed should be as "transparent" as possible and obvious to the programmer. This reduces the amount of unexpected behavior that results in program errors.

Productivity. The speed of programs written in the language being developed should be compared with similar programs written in C language.

Worst practices should be severe. There is a big difference between "supported language" and "encouraged language" [19]. The C and Rust languages support the constancy of variables, but the constants in Rust are more prevalent.

Analysis of typical program errors. According to the article [20], the most common errors in operating systems can be limited by language. In the article [21] analyzes the error rates in the Linux Kernel. Analysis of the variety of errors described [22], with ways to avoid them at the language level.

Requirements to the programming language (desirable characteristics)

1. Family syntax. In addition to the widely known C-like syntax family, there are many other popular syntaxes that allow you to write a more compact code. In particular, note the syntax of the Python language and the ML-like syntax. In order to facilitate the transition of data programmers and to promote wider perception, the developed language should have a C-like syntax.

2. Compatibility with language C. The language is required to be C-compliant, since it contains many existing libraries, including support libraries from hardware vendors. The language to be developed should support the external interface of the functions of language C

3. Absence of implicit type conversion. The language to be developed should not contain any implicit type transformations, to avoid non-obvious and difficult to debug the program errors. This idea is not new and has already been successfully implemented in languages such as Haskell or Rust. Confirmation of the appropriateness of this idea is also that the MISRA C standard forbids the use of most implicit transformations.

4. Constant Variables. The language to be developed should include the ability to create constant variables and, moreover, all variables must be constant by default. The compiler must issue diagnostic messages if the noncontact variable never changes.

5. System of Effects. A simplified system of effects will mark the individual functions of the effects they have. Example: Function allocates memory; Blocking function; I / O function; A network interacting feature. The compiler can automatically bring out the effects of other functions.

6. Dividing Secure and Dangerous Code. This language feature is useful for ensuring greater reliability of programs. Although the overall idea is similar to the effects system, separating the code into safe and dangerous requires additional support from the compiler.

7 *Assembler inserts.* The language to be developed should include standard mechanisms for the use of assembler inserts.

8 Extended linking support. The language developed should support management the placement of data and application code at specific addresses. Support for specifying section descriptions and their placement in the address space is an additional plus.

9 Generalized programming. Generalized programming is implemented in C ++ using templates, as well as Java (and many others) using generics. The language developed should support generic programming.

10 Traits. It is known that the imitation of implementation greatly complicates the understanding of programs, as well as destroys encapsulation [23]. Traits, on the contrary, are much simpler for general understanding.

11 Conditional compilation. Conditional compilation allows you to enable or disable certain pieces of software code depending on certain conditions that can be checked during compilation. The language under development should have mechanisms to support conditional compilation.

12 Expressions of compilation time. Expressions of compilation time is the ability of the compiler to perform calculations when compiling programs, which allows for even more optimizations. Since they allow you to check more complex conditions.

13 Calculation of the maximum stack size at the compilation stage. One of the common mistakes is the placement of large objects on the stack, which leads to its exhaustion. In the absence of recursion, the compiler can calculate the maximum stack volume that can be used by any function.

14 Optional runtime checks. Buffer overflows are one of the most common programmatic errors that lead to security vulnerabilities and cause many well-known holes in security, including Heartbleed [24] and Blueborne [25]. Additional runtime checks should be enabled by default, but the compiler should be able to disable individual test classes for applications that require maximum performance.

15 Guaranteed tail call optimization. With this optimization, it is possible to implement certain recursive algorithms without increasing the size of the call stack. This can be important for programming embedded systems, since almost always the size of the stack is limited.

IV. LANGUAGE DESCRIPTION

Comments

Multi-line comments can lead to unexpected errors. Nested comments can also cause issues (if they are supported). On the contrary, single-line comments are as simple as possible. EBNF for comments is shown in Fig. 1

```
1  //The language supports only single-line comments
2
3  //In order to continue the comment on the next line
4  //the line should begin with a new comment
```

Fig. 1. An example of comments in the language

```
1  comment = '/', '/', { all characters - '\n'};
2  all characters = ? all unicode characters ?;
```

Fig. 2. EBNF for comments

Identifiers

Identifiers (or names) can be any sequence consisting of Latin letters, numbers, and underscores, and starts with a letter or underscore. The EBNF for identifiers is shown in Fig. 3.

```
1  id = ( alpha | "_"), {alpha | "_" | digit};
2
3  alpha = ? all latin letters ?;
4  digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Fig. 3. EBNF for Identifiers

Type void

The developed language being includes the void type, which has the only possible value - void. This value is zero byte and is fully optimized by the compiler (not present at runtime). This differs from the approach in C, where the type void has no value. This feature will be useful for determining the more powerful properties of the system type. The void type only supports validation checks (== and! =). (All void values are equal to each other).

Logical type

The developed language supports the logical type **bool**. It has two possible values: true and false. It is used in logical operations, as well as in the conditions of operators. The list of operations supported between two values of the logical type is given in Table I.

TABLE I. OPERATIONS SUPPORTED BY THE LOGICAL TYPE

Operation Symbol	Operation	Result Type
<	Less than	bool
>	Greater than	bool
<=	Less than or equal to	bool
>=	Greater than or equal to	bool
==	Equal to	bool
!=	Not equal to	bool
&&	Logical AND	bool
\|\| | Logical OR | bool |

When comparing values, false is less than true.

Numbers

The developed language supports the following numerical types: **u8** 8-bit unsigned integer; **i8** 8-bit signed integer; **u16** 16-bit unsigned integer; **i16** 16-bit signed integer; **u32** 32-bit unsigned integer; **i32** 32-bit signed integer; **u64**. 64-bit unsigned integer; i64 64-bit signed integer.

Floating-point numbers are not supported in this work, as they are rarely used in system software, and in some cases (kernel programming of operating systems), use of floating-point numbers is prohibited. Support for floating point numbers can be implemented in the next version of the language. Numbers can be entered in the usual decimal system, as well as in hexadecimal or binary. The number system is set using the prefix. Prefixes 0x and 0b used to specify hexadecimal and binary numbers respectively; decimal numbers do not require prefixes. Additionally, you can specify the exact type by using postfixes. This avoids ambiguity. EBNF for numerical literals is shown in Fig. 4.

```
number value = hexadecimal number
    | binary number
    | decimal number;

hexadecimal number = '0x', hex digit, {hex digit};
binary number = '0b', binary digit, {binary digit};
decimal number = digit, {digit};

hex digit = digit |
    'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
    'a' | 'b' | 'c' | 'd' | 'e' | 'f';

binary digit = '0' | '1';

number type specifier =
    'u8' | 'u16' | 'u32' | 'u64' |
    'i8' | 'i16' | 'i32' | 'i64';
```

Fig. 4. EBNF for numerical literals

The list of operations supported between numbers of the same type given in Table II, where *number* means the type of arguments.

TABLE II. LIST OF OPERATIONS FOR NUMBERS OF THE SAME TYPE.

Operation Symbol	Operation	Result Type
+	Addition	number
-	Subtraction	number
*	Multiplication	number
/	Integer Division	number
%	The remainder of an integer division	number
&	Bitwise AND	number
|	Bitwise OR	number
	Bitwise XOR	number
>	Greater than	bool
<	Less than	bool
>=	Greater than or equal to	bool
<=	Less than or equal to	bool
==	Equal to	bool

Bitwise shifts are supported for any numbers, provided that the second operand is the number of any unsigned type. The result of the operation is the type of the first operand. The symbol << means a bitwise shift to the left, >> - a bitwise shift to the right. Also, a bitwise NO operation supported by the "!" Symbol. For example, the result of the operation !0x0Fu8 is 0xF0u8. By default, for the addition, subtraction and multiplication operations, the compiler must insert runtime checks for overflows; For a division operation, it is additionally verified that the divisor is not zero.

Pointers

The pointers in the developed language support a functional similar to the functional of pointers in the C language. The language supports a special literal null, which means a zero pointer of any type (analog NULL from C language). Indicators can be created using the address picking operation, or by bringing the values of the numeric type of the corresponding size to the type of the pointer. The developed language supports two types of pointers that are characterized by the constant of the value to which they refer.

* *type* The pointer to the unchanged value of the type *type*. For example, the pointer to the logical type has the type *bool;

* *mut type*. The pointer to the variable type *type*. For example, the pointer to the logical type has the type *mut bool.

Any operations on pointers are considered "dangerous" and must occur in the appropriate blocks.

Links

Links support a limited number of operations: they do not support address arithmetic, can not point to null. Multiple references can not point to the same memory area. Links can be created using the address capture operation, or by bringing the pointer value to a link (which is a "dangerous" operation, since it can form a link indicating null). As with pointers, the developed language supports two types of links that are characterized by the constant of the value to which they refer.

& type The reference to the unchanged value of the *type* type. For example, the reference to the value of the logical type has the type &bool;

& mut type Link to variable *type* type. For example, the reference to the value of the logical type has the type & mut bool.

All operations on links are "safe" and do not require special blocks.

Arrays

The array of a value *type* type that contains N elements is written as [type; N]. Unlike C, arrays are not referred to as a pointer to an element. This can be done using the & arr [0] or the as_ptr () method. To specify an array in the program, it is enough to list all the elements of the array through a comma and surround them in square brackets. Arrays support indexing operation. An example of its use is shown in Fig. 5. If elements are supported by comparison operations, they can also be applied to the array as a whole.

```
let x = [1, 2, 3];
assert!(x[0] == 1);
assert!(x[1] == 2);
assert!(x[2] == 3);

let arr: [[u8; 32]; 32];
arr[13][14] = 10;
```

Fig. 5. An example of a two-dimensional array and indexing

In addition, the arrays support the following methods:

len () Returns the length of the array; is_empty () Returns true if the array length is zero; as_ptr () Returns the pointer to the first element of the array; contains (). Returns true if the array contains an element with a given value; starts_with (). Returns true if the specified argument is a prefix of the array; ends_with (). Returns true if the specified argument is a suffix of the array; swap (). Swap two specified elements of the array; reverse () Expands the order of elements in an array; sort () Sort elements of the array in order; iter () Returns an iterator to elements of an array.

Array slicing

Array slicing elements of the *type* type is written as [type]. The slicing consists of a pointer to the first element of the array and the number that stores the number of elements. The slicing can be obtained using a special operation, an example of which is shown in Fig. 6.

```
let arr: [ u8; 32];
arr[0..3] // cut off an array of 0 to 3 elements inclusive
```

Fig. 6. An example of getting an array cut

The cuts support the following methods: len () Returns the length of the slicing; is_empty () Returns true if the length of the slicing is zero; as_ptr () Returns the pointer to the first element of the slicing; contains (). Returns true if the slicing contains an item with a given value; starts_with (). Returns true if the specified argument is a prefix of the slicing; ends_with (). Returns true if the specified argument is a suffix of the slicing; swap (). Exchange places two specified elements of the slicing; reverse () Expands the order of the elements in the slicing; sort () Sort items in the slicing by order; iter () Returns the iterator to the elements of the slicing.

Strings

The strings in the developed language are represented as an array of unsigned bytes. Supported string literal, which can be any set of symbols surrounded by double quotes. Multi-row literal letters are not allowed. Character shielding is supported with a backslash (\). The table of Screen sequence supported is shown in Table III.

Also, the language allows different interpretations of string literals during compilation using prefixes. An example of such an interpretation is the prefix b, which interprets string letters as an array of unsigned bytes (type u8, unlike type char supporting Unicode character encoding). This interpretation is useful because it does not allow Unicode to be used in programs that do not need it (which is especially useful in systems with limited memory).

TABLE III. TABLE 6: LIST OF SCREEN SEQUENCE

Screen sequence	Hexadecimal value	Symbol
\n	0x0A	New line
\r	0x0D	Carriage return
\t	0x09	Horizontal Tab
\\	0x5C	Backslash
\'	0x27	Single bracket
\"	0x22	Double bracket
\xhh	any	Byte, whose value is hh interpreted as a hexadecimal number

Tuples

The language developed supports types of tuples. The type of the tuple is written as follows: (*type1*, *type2[, ...]*). And it means a tuple of types *type1...typeN*. For example, a tuple of 8-bit unsigned integer, logical value and pointer have a type (u8, bool, * u8). An example of this value can be (42, true, null). You can access the values in the tuple using the field access operations. Field names are natural numbers starting from scratch. An example of such use is shown in Fig. 7.

```
(42, true, null).0 == 42
(42, true, null).1 == true
(42, true, null).2 == null
```

Fig. 7. Example of access to the tuple fields

Structures

Structures allow you to group data of different types, giving them names, thereby they are more organized way than tuples. The general forms of declaring a structure and creating its instances are shown in Fig. 8.

```
// Structure declaration
struct struct_name{
    field1: type1;
    field2: type2;
    ...
    fieldN: typeN;
}

// Easy instance creating
let x = struct_name { value1, value2, ..., valueN };


// Creating an instance using named fields
let y = struct_name {
field1: value1,
field2: value2,
...
fieldN: valueN, };
```

Fig. 8. The type of structure declaration and create its instances

```
struct definition = ['pub'], 'struct', id,
    '', field definition, '';

field definition = ['pub'], id, ':', type, ';';
```

Fig. 9. EBNF for the declaration of structures

Access to the structure fields can be done with the operator ".".

Enumerated types

The language being developed supports enumerated types. The definition of the enumeration creates a separate type that is not automatically assigned to other types. (Unlike C, where enum only defines constants in the global namespace). This feature is similar to the enum class in C ++ [26]. Access to enumerated values can be obtained using the namespace operator. EBNF for definition of the enumerations are shown in Fig. 10.

```
enum definition = ['pub'], 'enum', id,
    '{', enum options, '}';

enum options = [option, [',', enum options], [',']];

option = id, ['=', expression];
```

Fig. 10. EBNF for definition of the enumerations

Functions

The language developed supports user-defined functions. The general form of function definition is shown in Fig. 11.

```
fn function_name(arg1: type1): rettype {
    function;
    body;

    return_value
}
```

Fig. 11. General form of function definition

EBNF for the function definition is shown in Fig. 12.

```
function def = ['pub'], id, '(', args, ')', ':', type,
    compound statement;

args = [argument, [',', args]];

argument = id, ':', type;
```

Fig. 12. EBNF for the function definition

All functions have the type fn(*type1[, ...]*): *rettype* , where *type1 ... typeN* are the types of the function's arguments, and *rettype* is the type of the return value. The function name is a value of this type. This type supports only assignment and call operations, and can also be passed as an argument to the function.

Implementation blocks

Structures and other types can be expanded by methods using implementation blocks. Types may have more than one implementation block, which means that types can be expanded, even if they are in third-party libraries. EBNF for the implementation blocks are shown in Fig. 13.

```
impl block = 'impl', [type, 'for'], type,
    '{', function def, '}';
```

Fig. 13. EBNF for the implementation blocks

It is also worth noting that the extension is allowed even for standard types, which greatly increases flexibility. The implementation blocks allow you to add both instance methods and type methods. If the name of the first parameter of the method is self, then this method is an instance method; otherwise, it is a type method. In the middle of the implementation block, the special name Self means the type for which this block is applied.

The self parameter can have one of the following types:

- Self;
- &Self;
- &mut Self;
- *Self;
- *mut Self.

Other types of the self parameter are not allowed.

Instance methods can be called using the access operator used for the instance of the type. Type methods can be called using the namespace operator (::).

```
let r = Rectangle::square(2);
assert!(r.area() == 4);
assert!(r.perimeter() == 8);

assert!(4u32.pow2() == 16);
```

Fig. 14. An example of using methods

Traits

The trait is a language feature that shows to a compiler, which functionality should provide a certain type. Traits look like Java interfaces. Their peculiarity is that the implementation of the interface should not be inside the class definition. It can be provided separately. An example of determining the trait is shown in Fig. 15.

```
trait Figure {
    fn area(self: &Self): u32;

    fn perimeter(self: &Self ): u32;
}
```

Fig. 15. An example of determining the trait

A trait implementation can be provided with an implementation unit. If at the stage of the definition of the trait is already known, as will usually be implemented a particular method, you can specify its implementation by default. Traits can inherit from it. If trait B is inherited from type A, then the type that wants to implement B is also required to implement type A. EBNF for trait determination is shown in Fig. 16.

```
trait def = 'trait', type, [':', type],
    '{', function def | function decl, '}';

function decl = ['pub'], id, '(', args, ')', ':', type;
```

Fig. 16. EBNF for trait determination

Type inference

The language being developed supports global type inference (“type inference” or “type reconstruction”). In most places where you want to specify a type, you can specify a special placeholder type, indicated by an underscore: "_".

If the compiler will not be able to reconstruct the type of filler, the compilation of the program will be interrupted. Each use of the type-filler instantiates a new type-variable. In other words, this means that all uses of the filler type are independent and can be derived in different types. In order to refer to a particular type of filler, the developed language supports named type fillers. Named type filler is any name that begins with one underline. The same type of fillers of the type will be reconstructed to a common type. By behavior, filler types are very similar to the wildcard types in Haskell [27]. EBNF for filler types is shown in Fig. 17.

```
type placeholder = '_', {alpha | digit};
```

Fig. 17. EBNF for filler types

5.18 Variables

The language supports the definition of variables and constant variables. By default, the variable is constant. To create a non-constant variable, you must add the keyword mut (from the word "mutable"). An example of variables definition is shown in Fig. 18

```
// unchangeable variable
let x: u32 = 5;
x = 6; // error!

// changeable variable
let mut y: u32 = 42;
y = 15;
```

Fig. 18. An example of variables definition

Constants

In addition to unchangeable variables, the language developed supports the compilation time constant. The compiler ensures that their values are calculated during compilation and can not be changed. Example of constant are shown in Fig. 19

```
const x: u32 = 5;
```

Fig. 19. Example of constant

Conditional operator

The syntax of the conditional operator is shown in Fig. 28. "condition" is an expression of a logical type. "expr1" and "expr2" have the same type. Conditional operator is an expression, the resulting type of which is equal to the type expr1 and expr2.

```
if condition { expr1 } else { expr2

if condition1{
    expr1
} else if {
    expr2
} else
    expr3
}


if condition { expr1 }
// similar to
if condition { expr1 } else { void }
```

Fig. 20. The syntax of the conditional operator

As in all programming languages, the expression expr1 will be calculated when the condition is true; otherwise the expression expr2 will be calculated. If the else branch consists of one conditional operator, curly brackets can be omitted, as shown in the second example in Fig. 20. In all other cases, curly brackets around the branches of the conditional expression are mandatory. The else branch can be omitted. In this case, it will return void from this branch. This means that the branch then also has to return void. EBNF for conditional expressions is shown in Fig. 21.

```
conditional expr =
    'if', expression, compound statement,
    ['else', (conditional expr | compound statement)];
```

Fig. 21. EBNF for conditional operator

Cycles

The language being developed supports the while loop, as shown in Fig. 22

```
while condition {
    statements;
}
```

Fig. 22. The general form of the while loop

During execution, the condition is first calculated. If the condition is satisfied, the loop body is executed while the condition is valid; otherwise, the loop body is not executed.

```
let mut i = 0;

let x = while true {
    if i >= 10 {
        break i;
    }

    i = i + 1;
};

assert!(x == 10);
```

Fig. 23. Returning a value from a loop using the break statement

In the body of the loop, you can use the key continue and break statements:

- continue completes the current iteration of the loop and proceeds to check the condition for the next iteration;
- break completes the loop completely. Can return a value from a loop.

In Sections II-IV, the system programming and programming environment for embedded devices was analyzed. The main requirements and limitations for the programming language in this field were identified. It was formed the main priorities in the development of the language, namely: reducing the number of program errors without reducing the performance of compilation programs. A number of scientific publications were analyzed and typical mistakes made by programmers of system software are allowed, and ways of preventing them are suggested. A list was generated and a detailed description of the desired characteristics of the language was provided. Based on the collected information, the language was developed and its description provided, including examples and description of EBNF for the most important parts.

V. Implementation of the System and Testing

In this section a compiler is developed, it can compile programs written in the proposed language.

Compiler Architecture

The developed compiler uses the advantages provided by the LLVM project. LLVM is a collection of modular and re-usable technologies for compiler writing. The LLVM Core library provides a modern optimizer, as well as a code generator for a large number of popular platforms. This library works with an intermediate code, called LLVM IR (LLVM intermediate representation). The developed compiler has a fairly typical architecture, consisting of the following components:

- **Lexer**. Responsible for reading the input file and breaking it into a token stream;
- **Parser**. Gets an input stream of tokens, formed by Lexer, and performs parsing to form an abstract syntax tree;
- **Semantic analyzer**. Accepts an abstract syntax tree and performs a semantic analysis, verifying that the program can be compiled

- **Intermediate code generator**. Gets an abstract syntactic tree, supplemented with reconstructed types, and generates an intermediate-level code, then LLVM Core is used to generate the machine-dependent code.

In fact, the system being developed implements the frontend of the compiler for the language, and LLVM Core takes on the backend duties.

Selection of the implementation language

To implement the compiler, the Haskell language was chosen for the following reasons.

- **Easy to work with (syntax) trees**. Haskell has a very easy syntax for working with algebraic data structures, and trees are very easy to represent in the form of sums and product types;
- **Ease of processing streaming data**. The Haskell language is ideal for programs with linear control flow (and not only). It allows you to perform powerful optimization over the entire program is;
- **Irremovability of data**. Since during the compilation of the program, AST and other program data should remain unchanged, Haskell, being a functional language, is ideally suited for this task;
- **Ease of parallel processing**. Haskell has powerful mechanisms for parallel data processing;
- **Availability of tools for writing compilers**. In Haskell there is an environment for writing compilers. It includes generators of lexers and parsers (Alex and Happy), dozens of libraries for writing parsers, wrappers for the interface of the LLVM software.

Abstract syntax tree

The abstract syntax tree is defined in the Syntax module. This module contains the following types:

- **Module**. It is a top-level object. Each file in the program is presented as a module. Each module consists of an import list and an ad list;
- **Id**. Identifier;
- **QualifiedName**. A qualified name. Allows you to access objects that are in other modules;
- **Import**. One import. Contains the name of the imported module, and may also contain a list of imported objects and their renaming;
- **ImportRename**. Renaming. Allows you to import an object with a different name;
- **Decl**. Top-level ads. Can declare an external function or variable, define local structures, functions, variables, constants, implementation units or traits;
- **Attribute**. An attribute that can be added to any declaration and is a hint for the compiler;
- **FieldDecl**. Declarations of a single field of structure;
- **Scope**. Zone of visibility. Public or private;
- **TypeParam**. Type-parameter. Used for general programming;. Type - parameter. Used for general programming;
- **Param**. Function parameter. Contains an identifier and an associated type,
- **Type**. Encodes the type of value;
- **Expr**. Expression
- **Statement**. Instruction
- **Mutability**. Mutability of the object. Variable or unchangeable
- **Literal**. Literals

Lexer

Generator of lexical analyzers Alex [26] was chosen for the lexer implementation . Alex is a tool for generating lexical analyzers in the Haskell language, based on the description of tokens for recognition, provided in the form of regular expressions. It is similar to the lex (or flex) tool for C / C ++.

The FileInfo type describes the position of a character in a file that can be given (Position), or unknown (PositionUnknown). The specified position contains the line number and columns (fiLine and fiColumn, respectively). The Token type describes all available tokens. Each token has an associated position in the file. The function scanTokens takes input to the contents of the file and returns a list of tokens.

Parser

The generator of syntactic analyzers Happy [27] was selected to implement the parser. Happy is Haskell's parser generation system, similar to the yacc tool for C. Like yacc, it accepts an input file containing an annotated BNF grammar specification and creates a Haskell module that contains a grammar parser. The Parser module implements a parser interface, which consists of one parse function that accepts the list of tokens in the input and returns an abstract syntax tree (type Syntax.Module). Function type: parse :: [Lexer.Token] -> Syntax.Module

Semantic analyzer. Outputting of types

The most important part of the semantic analyzer is a module for outputting types that performs type reconstruction using the Hindley-Milner algorithm (Hindley-Milner). The subsystem of the output of types is implemented in the module Infer. The Hindley Miller type system is a whole family of types of systems that have an algorithm for identifying types with untyped syntax. This is achieved through the unification process, when the program leads to a set of constraints that, when solved, have a unique principal type.

Generator of intermediate code

The intermediate code generator is implemented in two modules:

- Codegen contains the necessary definitions for working with LLVM libraries and code generation;
- Emit contains rules by which AST is converted to LLVM IR.

The bindings for LLVM in Haskell are divided into two packages: llvm-hs-pure this is a pure representation of LLVM IR in Haskell; llvm-hs these are FFI bindings to the LLVM library which are needed to create the C-representation of LLVM IR, as well to optimization and compilation.

CodegenState and BlockState types are used to store the state of the code generator at the moment it passes through a syntax tree. The codegen function is an entry point for the coding module. It takes an abstract syntax tree to the input and returns the generated LLVM IR code.

Testing the system and its components

The lexer and parser modules are covered by unit tests in the LexerSpec and ParserSpec modules, respectively. In addition, the compiler has been tested by compiling several programs of moderate size and beliefs in their performance. Also manually compare the generated code with the expected one. It described all the compiler modules with their interface. The command line interface of the compiler was also documented. The compiler has been tested and no program errors have been detected.

VI. Conclusions.

This work is devoted to language problems for embedded systems, programming of operating systems, as well as system programming. The aim of the paper was to develop a language to reduce the number of programmating errors made by programmers. The subject area was considered, its features, requirements and restrictions were revealed. The languages that are currently used in this area were analyzed. We analyzed the typical mistakes allowed by programmers, and suggested solutions for them. A new language and a compiler for it were developed. The language developed to some extent achieves its goal. Also, it warns a wide class of errors. However, there are many potential approaches and tools that have not been sufficiently considered and implemented.

References

[1] D.M. Ritchie, ‘C Programming Language History’, [Online]. Available: https://www.livinginternet.com/i/iw_unix_c.htm. [Accessed: 27- April- 2018].

[2] idris-lang.org, ‘Idris. A Language with Dependent Types’, [Online]. Available: https://www.idris-lang.org. [Accessed: 27- April- 2018].

[3] hackage.haskell.org, ‘idris: Dependently Typed Functional Programming Language’, [Online]. Available: http://hackage.haskell.org/package/idris-0.1.3. [Accessed: 27- April- 2018].

[4] hackage.haskell.org, ‘atom: An EDSL for embedded hard realtime applications.’, [Online]. Available: https://hackage.haskell.org/package/atom. [Accessed: 27- April- 2018].

[5] ivorylang.org, ‘Ivory Language’, [Online]. Available: https://ivorylang.org/index.html. [Accessed: 27- April- 2018].

[6] haskell.org, ‘Haskel An advanced, purely functional programming language’, [Online]. Available: https://www.haskell.org/. [Accessed: 27- April- 2018].

[7] nim-lang.org, ‘nim Efficient and expressive programming.’, [Online]. Available: https://nim-lang.org/. [Accessed: 27- April- 2018].

[8] nim-lang.org, ‘Nim Manual Pragmas’, [Online]. Available: http://nim-lang.org/docs/manual.html#pragmas. [Accessed: 27- April- 2018].

[9] github.com, ‘Nim Issues’, [Online]. Available: https://github.com/nim-lang/nim/issues. [Accessed: 27- April- 2018].

[10] rust-lang.org, ‘The Rust Programming Language’, [Online]. Available: https://www.rust-lang.org/en-US/. [Accessed: 27- April- 2018].

[11] lambda-the-ultimate.org, ‘Lambda the Ultimate’, [Online]. Available: http://lambda-the-ultimate.org/node/4009. [Accessed: 27- April- 2018].

[12] llvm.org, ‘The LLVM Compiler Infrastructure’, [Online]. Available: http://llvm.org/. [Accessed: 27- April- 2018].

[13] H. Massalin, ‘Synthesis: An Ecient Implementation of Fundamental Operating System Services’, Columbia University, 1992, 158 pages.

[14] arm.com, ‘Cortex-M Series’, [Online]. Available: https://www.arm.com/products/processors/cortex-m. [Accessed: 27- April- 2018].

[15] st.com, ‘STM32 32-bit ARM Cortex MCUs’. [Online]. Available: http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html. [Accessed: 27- April- 2018].

[16] Rasmus Christian Larsen, ‘Advanced Sleep-Mode Techniques for Enhanced Battery Life in Real-Time Environments’, 2011. [Online]. Available: https://www.digikey.com/en/articles/techzone/2011/dec/advanced-sleep-mode-techniques-for-enhanced-battery-life-in-real-time-environments. [Accessed: 27- April- 2018].

[17] J. Atwood, ‘Hardware is Cheap, Programmers are Expensive’, 2008. [Online]. Available: https://blog.codinghorror.com/hardware-is-cheap-programmers-are-expensive/. [Accessed: 27- April- 2018].

[18] misra.org.uk, ‘Motor Industry Software Reliability Association’, [Online]. Available: https://www.misra.org.uk/. [Accessed: 27- April- 2018].

[19] G. Gonzalez, ‘Worst practices should be hard’, [Online]. Available: http://www.haskellforall.com/2016/04/worst-practices-should-be-hard.html. [Accessed: 27- April- 2018].

[20] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors”, ACM SIGOPS Operating Systems Review, vol. 35, pp. 73-88, 2001.

[21] P. J. Guo and D. Engler, “Linux Kernel Developer Responses to Static Analysis Bug Reports”, Proceeding USENIX'09 Proceedings of the 2009 conference on USENIX Annual technical conference, pp. 22-22, June 14-19, 2009.

[22] I. Abal, C. Brabrand and A. Wąsowski, ‘40 Variability Bugs in the Linux Kernel’, IT University of Copenhagen, Copenhagen, 2014.

[23] L. Krubner, ‘Object Oriented Programming is an expensive disaster which must end’, [Online]. Available: http://www.smashcompany.com/technology/object-oriented-programming-is-an-expensive-disaster-which-must-end. [Accessed: 27- April- 2018].

[24] heartbleed.com, ‘The Heartbleed Bug’, [Online]. Available: http://heartbleed.com/. [Accessed: 27- April- 2018].

[25] J. Hildenbrand, ‘Let's talk about Blueborne, the latest Bluetooth vulnerability’, [Online]. Available: https://www.androidcentral.com/lets-talk-about-blueborne-latest-bluetooth-vulnerability. [Accessed: 27- April- 2018].

[26] haskell.org, ‘Alex: A lexical analyser generator for Haskell’, [Online]. Available: https://www.haskell.org/alex/. [Accessed: 27- April- 2018].

[27] haskell.org, ‘Happy. The Parser Generator for Haskell’, [Online]. Available: https://www.haskell.org/happy/. [Accessed: 27- April- 2018].

Шмалько О.О., Регіда П.Г., Волокита А.М., Луцький Г.М., Ву Дик Тхінь.

Мова програмування зі зменшенням помилок без зниження продуктивності для вбудованих пристроїв реального часу

Проблематика. Для програмування вбудованих систем, найчастіше використовуються мови С або С++. До основних недоліків відносять: відсутність оновлень, важкість у використанні, обмежену зворотну сумісність та потенційно велику кількість можливих помилок програмістів. Тому важливо забезпечити розробників низькорівневих програмних засобів для вбудованих систем, операційних систем та системних утиліт швидкою, продуктивною, надійною та стабільною мовою з урахуванням сучасної теорії програмування.

Мета досліджень. Створення нової продуктивної та надійної мови програмування для вбудованих систем із використанням принципів та підходів сучасної теорії програмування.

Методика реалізації. Проведення аналізу відомих публікацій, присвячених мовам програмування, які використовуються для вбудованих систем, дало змогу виявити їх основні недоліки та переваги. Розгляд сучасних підходів до реалізації мов програмування дозволив визначити вимоги до розроблюваної мови.

Результати досліджень. Розроблено нову мову програмування для вбудованих пристроїв. Описані модулі компілятора: лексер, парсер, семантичний аналізатор, генератор проміжного коду. Виконано детальний опис розробленої мови програмування.

Висновки. В даній роботі запропоновано використання нової мови програмування для вбудованих пристроїв. Було проведено аналіз існуючих мов програмування та типових помилок розробників для забезпечення надійності запропонованої мови.

Ключові слова: мови програмування; вбудовані системи; системи реального часу.

Шмалько А.А., Регида П.Г., Волокита А.Н., Луцкий Г.М., Ву Дык Тхинь.

Язык программирования с уменьшением ошибок и без снижения производительности программ для встроенных устройств реального времени

Проблематика. Для программирования встроенных систем чаще всего используются языки С или С ++. К основным недостаткам относят: отсутствие обновлений, сложность в использовании, ограниченную обратную совместимость и потенциально большое количество возможных ошибок программистов. Поэтому важно обеспечить разработчиков низкоуровневых программных средств для встроенных систем, операционных систем и системных утилит быстрым, продуктивным, надежным и стабильным языком с учетом современной теории программирования.

Цель исследований. Создание нового производительного и надежного языка программирования для встроенных систем с использованием принципов и подходов современной теории программирования.

Методика реализации. Проведение анализа известных публикаций, посвященных языкам программирования, которые используются для встроенных систем, позволило выявить их основные недостатки и преимущества. Рассмотрение современных подходов к реализации языков программирования позволило определить требования к разрабатываемому языку.

Результаты исследований. Разработан новый язык программирования для встроенных устройств. Описаны модули компилятора: лексер, парсер, семантический анализатор, генератор промежуточного кода. Выполнено детальное описание разработанного языка программирования.

Выводы. В данной работе предложено использование нового языка программирования для встроенных устройств. Был проведен анализ существующих языков программирования и типичных ошибок разработчиков для обеспечения надежности предложенного языка.

Ключевые слова: языки программирования; встроенные системы; системы реального времени.