Development of a compiler for the high-level programming language Escle into byte-code of virtual machine of the Microsoft .NET platform

Course paper by
Mikhail Barash

Supervised by
Docent Alexander Mezentsev

Irkutsk — 2010
Introduction

Compiler construction is a very important branch of computer science, both from theoretical and practical points of view. Compilers comprise a very significant part of computer science since the time when high-level programming languages became the main tool for software development.

A translator is a tool that, given a program as an input, generates another program which is equivalent to the one in the input. The generated program can be represented either in the native machine code or in an intermediate language (byte-code).

1 Purpose of the paper

The purpose of the present paper to develop an imperative programming language and a compiler for it into the byte-code of Microsoft .NET virtual machine.

In order to achieve this goal, the following tasks shall be addressed:

- specify a high-level programming language;
- construct an attribute translating grammar in the format of CoCo/R Compiler Compiler [13, 14];
- implement semantic analysis module;
- specify an intermediate language and implement the module of intermediate-language code generation;
- implement Microsoft .NET byte-code generation module, given the intermediate-language representation of a program;
- develop a Microsoft Windows application, allowing a user to input source code of a program and to compile the program into the byte-code of Microsoft .NET virtual machine.

2 Escle Programming Language

Escle programming language is an imperative object-oriented programming language. The language has several non-classical constructions, to be introduced and explained within this section.
2.1 Structure of a program

A program in Escle programming language is comprised of the following elements:

- optional program header;
- declarations of variables, invariants, and classes;
- bodies of subroutines;
- body of the entry-point subroutine (with modifier `main`).

In a program in the Escle programming language, the following two kinds of comments are allowed:

- `//` — comment up to the end of the current input line;
- `/*..*/` — block comment, that may span over several lines of code.

Escle programming language has the following special operator

```
fail(<string_expression>);
```

that raises `System.Exception` with the property `Message` having the specified string value.

Since Escle programming language is implemented to be used with Microsoft .NET platform, the user of the language has to have access to all methods of all classes of the *Base Class Library*.

2.2 Data types in Escle

The following types are built-in:

- `int`: integer type (corresponds to `System.Int32` of Microsoft .NET);
- `real`: real type (corresponds to `System.Double` of Microsoft .NET);
- `bool`: Boolean type (corresponds to `System.Boolean` of Microsoft .NET);
- `string`: string type (corresponds to `System.String` of Microsoft .NET);
- `vector`: vector type, to be explained below.
2.3 Expressions in Escle

Arithmetical expressions in Escle programming language are comprised of operands, operators, parentheses and function calls. If an expression does not contain any parentheses, then the order in which the operators are applied to operands is determined by standard priority of operations.

Boolean expressions consist of logical operands (that is, variables, constants, calls to functions returning a Boolean value) and relation operators, parentheses and Boolean operators (and for conjunction, or for disjunction, not for negation, and xor for exclusive disjunction).

A relation expression, in its turn, is comprised of two arithmetical expressions, delimited by relation operators: greater than (>), greater or equal (>=), equal (=,==), not equal (<>), less (<), less or equal (<=). The order, in which Boolean expressions are to be computed, is determined by the standard priority of operations in mathematics: the unary negation operator not has the highest priority, then follows conjunction operator and, and the least priority is given to disjunction operator or.

The following arithmetical and Boolean operators are defined in Escle programming language.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type of arg. 1</th>
<th>Type of arg. 2</th>
<th>Type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, ^, /, mod</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>+, -, *, ^, /</td>
<td>real</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>+, -, *, ^, /</td>
<td>int</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>+, -, *, ^, /</td>
<td>real</td>
<td>int</td>
<td>real</td>
</tr>
<tr>
<td>+</td>
<td>string</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>unary -</td>
<td>int</td>
<td>string</td>
<td>int</td>
</tr>
<tr>
<td>unary -</td>
<td>real</td>
<td>—</td>
<td>real</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=, =, ==, &lt;&gt;, !=</td>
<td>int or real</td>
<td>int or real</td>
<td>bool</td>
</tr>
<tr>
<td>and, or, xor</td>
<td>bool</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>not</td>
<td>—</td>
<td>—</td>
<td>bool</td>
</tr>
</tbody>
</table>

The following additional operators are defined in Escle programming language.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Type of arg.</th>
<th>Type of result</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>any</td>
<td>string</td>
<td>name of the variable</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>int</td>
<td>absolute value</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>real</td>
<td>absolute value</td>
</tr>
<tr>
<td></td>
<td>string</td>
<td>int</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td>vector</td>
<td>real</td>
<td>Euclidean norm of vector</td>
</tr>
</tbody>
</table>

2.4 Declarations of variables

A declaration of a variable can be done in the following three ways:

- in the style of programming language *Pascal* (\texttt{var <names_of_variables> : <type>;})

- in the style of programming language *C* (**type** <names_of_variables>);

- in the style of programming language *C#* with automatic type inference based on the initial value of the variable (**var <name_of_variable> = <expression>;**).

After specifying the type of a variable, one can also specify:

- an initial value of the variable (= \texttt{<expression>}); in case when a variable is declared in the style of programming language *C#*, its initial value has to be specified;

- names of the subroutines that are to be invoked when the value of the variable is read/written (**getter <name_of_subroutine>;** \texttt{setter <name_of_subroutine>;}).

Consider the following examples of variable declarations in Escle programming language.

\begin{verbatim}
/*(1)*/ string a="Hello";
/*(2)*/ string b="This is a string";
/*(3)*/ int c1,c2=0;
/*(4)*/ var d=0; // int
/*(5)*/ var e=2/5; // real
/*(6)*/ int f=1; getter Proc1; setter Proc2;
\end{verbatim}

In the first two lines, two string variables \texttt{a} and \texttt{b} are declared and initialized.
In lines 3–5, integer variables \( c_1, c_2 \) and \( d \) are declared, as well as a real variable \( e \). The type of variables \( d \) and \( e \) is automatically inferred from their initial values.

Line 6 has a declaration of an integer variable \( f \). Every time this variable is used in an expression, the subroutine \( \text{Proc1} \) shall be called; similarly, the subroutine \( \text{Proc2} \) is called every time the variable \( f \) is assigned a new value.

## 2.5 Assignment operator

There are two kinds of assignment operator in Esele programming language:

- "classical" assignment (set <name> = <expression>);
- assignment to a set of variables that satisfy a certain condition (set '"<conditions_on_variables>"' = <expression>).

Let us consider examples of the "classical" assignment operator.

```plaintext
int x;
set x=10+20*System.Math.Sin(45*System.Math.Pi/180);
```

After this code is executed, the variable \( x \) will be assigned a value given in the right-hand side of the assignment symbol\(^1\).

The second kind of assignment operator allows one to make an assignment to a collection of variables, that satisfy certain conditions. Such conditions could be of the following kind:

- the name of a variable matches a certain pattern (mask^2);
- a variable has certain value;
- the value of the variable satisfies a certain range.

Consider the following examples of assignments to a collection of variables.

```plaintext
int h1,h2,h3,h4,h5,h6,x,y,z;
set h2=10;
set h6=100;
set {'h?', var<>10, var<>100}=1; // h1=1; h3=1; h4=1; h5=1;
set {'?'}=2; // x=2; y=2; z=2;
```

\(^1\)Note that if an assignment operator had not been followed by a semicolon, the conditions of the invariants (see Section 2.8) would not have been verified.

^2 Masks are understood in the way this concept is implemented in Microsoft Office Access.
Within specification of conditions (for instance, {’h?’, var<>10, var<>100}), the keyword var refers to a variable that satisfies those conditions.

2.6 Declaration of vectors

Vectors are one-dimensional arrays whose elements can be of any of the built-in types. The indexing of vectors starts with 0.

A declaration of a vector in Escle programming language includes:

- specification of type that the elements of a vector have (vector <type>);
- specification of the name of the vector and its upper bound in square brackets (<name> ’[’ <number_of_elements> ’]’)

In order to access an element of a vector, one has to specify the name of the vector followed by the index of the desired element in the square brackets. The index of the element should be an arithmetical expression.

Consider the following example of a program that prints the element with the maximal value of an integer vector.

**Listing 1. Maximal element of an integer vector**

```plaintext
program MaxInArray;
int i=0;
int max=0;
int maxpos=0;
vector int a[5];
procedure Hello(); main;
for i in [0,5] do
    System.Console.Write("Enter the value of the "+
        System.Convert.ToString(i) + "-th element: ");
end;
set max = a[0];
set maxpos = 0;
for i in (0,5] do // loop starts with i=1
    if (a[i] > max) then
        set max = a[i];
        set maxpos = i;
    end;
end;
```

3Note that the vectors cannot be dynamic.
System.Console.ReadLine();
end;

2.7 Subroutines

Subroutines are “base blocks, of which programs are comprised” [8, p. 81]. In Esce programming language, subroutines can be of the following two kinds:

- **procedures**, that is, subroutines that do not return a value (in terms of Microsoft .NET, such subroutines return a value of type `System.Void`);
- **functions**, that is, subroutines that return a value of a certain type.

Subroutines can be characterized by the following properties [8, p. 81]:

- each subroutine has one entry;
- after a subroutine is executed, the control is passed to the invoking program;
- when a subroutine is being executed, the execution of the invoking program is suspended.

The specification of a subroutine has:

- a **name** of the subroutine;
- a **signature**, that defines the number of arguments (formal parameters), the order they follow each other, the type of every argument, and, if the subroutine is a function, the type of the value it returns;
- a **body** of the subroutine.

The syntax of a specification of a subroutine in Esce programming language is as follows.

---

4Since the Esce programming language is implemented within Microsoft .NET platform, all subroutines of a program are represented as static method of a certain (“main”) class of the assembly.
procedure <name> ( <list_of_params> ): <type_of_returned_value> ;
   // statements
end;

In case when a subroutine is a function, the value that it returns should be given in the operator return.

Within procedures, one can use the operator exit; that interrupts the execution of the subroutine and passes control to the invoking program.

Every formal parameter of a subroutine:

- has type and name, given as <type> <name>;
- can be a pass-by-reference parameter, in this case it should be preceded by the keyword ref (note that by default, all formal parameters of subroutines are passed by value);
- can have an initial value, which follows the name of the parameter with after the equality sign;
- can be optional, and in this case the specification of the parameter should be given within the square brackets [ and ].

Within a procedure one can declare local variables, similarly to the case of variables within a program.

The invoking of a subroutine is done by specifying its name and list of actual parameters within the parentheses. Functions, unlike procedures, can be invoked within expressions.

The following example is a program computing the factorial of a given integer number.

**Listing 2. Factorial of an integer number.**

program Factorial;
var n=1;
var r=1;
invariant (n>0) and (n<20) otherwise {
   fail("Factorial can only be computed for a positive number less than 20");
};
procedure fact(int x):int;
   if arg x==1 then set r=1;
       else set r=arg x * fact(arg x - 1);
end;
return r;
In the present code, the invariant with condition \((n>0)\) and \((n<20)\) allows the program to be automatically aborted if the input number is negative or greater than 20.

Since Escle programming language is implemented within Microsoft .NET platform, the user of the language has to be able to invoke all functional members of all classes of Base Class Library.

The following example demonstrates the use of standard input/output subroutines, as well as the members of the class System.Math:

```csharp
int x=0;
int y=0;
procedure Hello();
    set x=System.Int32.Parse(System.Console.ReadLine());
    set y=System.Int32.Parse(System.Console.ReadLine());
    System.Console.WriteLine(System.Math.Max(x,y));
    System.Console.WriteLine(System.Math.Min(x,y));
end;
// ...
// other subroutines
// ...
// the main subroutine
procedure Main(); main;
    Hello();
end;
```

In this example, the procedure `Main` should be the entry point of a .NET-assembly.

Besides procedures and functions, there is also a special kind of functions, the so-called mathematical functions. Such functions allow a simplified syntax to be used in specifications of piecewise functions.

Consider an example of a piecewise function \(\text{Abs}(x)\), returning the absolute value of its argument:

\[
\text{Abs}(x) = \begin{cases} 
  x, & x \geq 0 \\
  -x, & x < 0 
\end{cases}
\]
This function can be represented as follows in Escle programming language.

\[
\text{function Abs}(x) = \{x, \text{if } x \geq 0; -x \text{ otherwise}\};
\]

### 2.8 Invariants

**Invariants** allow for checking the truth value of Boolean expressions over the variables during the execution of a program.

A declaration of an invariant consists of:

- a Boolean expression, which represents the condition on the variables to be checked (\texttt{invariant <Boolean_expression>});
- statements to be executed should the condition not hold (\texttt{otherwise <statements>}).

The conditions of all invariants are checked after every assignment operator.

The following example demonstrates the machinery of the invariants in action. Let \(x\) and \(y\) be two integer values that represent percentage. A natural condition on these two variables is that the value of each of them is within the range \([0, 100]\) and their sum is exactly 100. These conditions can be implemented as follows in Escle programming language. **Listing 3. Using of invariants.**

```escle
program H100;
int x=0;
int y=100;
invariant (x>=0) and (x<=100) otherwise {
    fail("The value of x is out of range: x=" +System.Convert.ToString(x));
};
invariant (y>=0) and (y<=100) otherwise {
    fail("The value of y is out of range: y=" +System.Convert.ToString(y));
};
invariant x+y=100 otherwise {
    fail("The sum x+y is not equal to 100: x+y=" +System.Convert.ToString(x+y)));
};
procedure Hello(); main;
    set x=5 // SIC! no semicolon, otherwise x+y=105
    set y=95; // the condition (x+y=100) is satisfied
end;
```
Note that there is no semicolon after the assignment operator set x=5: this supresses the checking of invariants, since otherwise the desired condition $x + y = 100$ would be a priori false. Thus, in this case, the conditions of invariants are only checked when assignments to both variables $x$ and $y$ have been made.

On the implementation level, invariants are represented as subroutines (with names hidden from the user of the language), that are invoked every time some variable is assigned a value. The fact that invariants are subroutines allows declaring and using local variables within their otherwise-"bodies". It is also worth noticing that nothing prevents a programmer to change the values of the "controlled variables" within the body of an invariant.

2.9 Conditional statements

A conditional statement allows executing one of the two alternative blocks of statements (branching) or executing a block of statements provided a certain condition holds [8].

Esclle programming language has two kinds of conditional statements:

- **univariant** (if (<Boolean_expression>) then <statements> end);
- **bivariant** (if (<Boolean_expression>) then <statements> else <statements> end).

The following code gives an example of a conditional statement in Esclle programming language.

```
set x=0;
if (x==0) then System.Console.WriteLine("Yes");
else System.Console.WriteLine("No");
end;
```

2.10 Loop statements loop, while, repeat, for and for of

Loop is a control structure used for execution of repeating computations in imperative programming languages. A loop operator consists of header and body, where the header control the number of repetitions, and the body is a block of statements [8].

There are several kinds of loop statements in Esclle programming language:

- **simple iteration loop** (loop <number> {<statements>});
• loop with a pre-condition (while <condition> do {<statements>});

• loop with a post-condition (repeat <statements> until <condition>);

• loop with a counter (for <counter> in [<from>,<to>] do {<statements>});

• loop with an ad hoc counter (for var <name> in [<from>,<to>] do {<statements>}), where the counter can only be accessed from within the loop;

• loop over variables satisfying certain conditions (for <counter> of <type> do {<statements>}, for <conditions> do <statements>).

The header of a simple iteration loop is an arithmetic expression stating the number of times the body of the loop should be repeated.

In the following example, a simple iteration loop is used to initialize a vector.

vector int a[10];
...
var i=0;
loop 11 { set a[i]=0; set i=i+1; };

Loops with pre- and post-conditions have the standard semantics.

A loop with a counter uses a designated variable as an number of repetitions of the loop. In the header of such a loop, the initial and final values of the counter are specified as a range, whose lower- and upper-bound may be inclusive or not: [.,.,..], (.,.,..), (.,.,..] or [.,.,..).

The loop over the variables satisfying certain conditions is similar to the corresponding kind of the assignment statement 2.5. There is a separate syntactic construction for a loop over the variables of a certain type.

int a,b,c=10;
...
for each of int do {
    set each=0;
}

As a result of this code, all integer variables will be initialized with 0.
2.11 Classes

Escl programming language supports the following concepts of object-oriented programming:

- classes with fields and methods;
- instances of classes (objects);
- inheritance.

The following example demonstrates how classes are defined in the Escl programming language.

class Person
  string Name;
  int Age;
  constructor (string aName, int aAge)
    set me.Name = aName;
    set me.Age = aAge;
  end;
  method ShowMe()
    System.Console.WriteLine(
      me.Name + " " +
      System.Int32.ToString(me.Age) );
  end;
end;

class Student
  inherits Person;
  int Group;
  // ...
end;

object Person x;
object Person y;

procedure Hello()
  new x("John", 20);
  x.ShowMe();
  // ...
end;

In this fragment of code, there are two classes defined: Person and Student, and the latter class inherits the former.
In order to refer to the fields and methods of a class within itself, the keyword `me` should be used. Similarly, the keyword `mybase` is used to refer to the members of the base class.

### 2.12 Generic programming

The paradigm of generic programming allows for description of algorithms that can be applied to different types, without changing the description of the algorithm itself [16].

The header of a *generic procedure* specifies, in angle brackets, the list of “types” which formal parameters of the procedure can have. The type of every “generic” argument is also specified in angle brackets and it can only be one of the “types” mentioned in the header of the generic procedure.

Consider the following example of a generic procedure that computes the maximum of its two arguments.

```plaintext
procedure Max<T>(<T> x, <T> y):<T>;
    if (x>=y) then
        return x;
    else
        return y;
end;
end;
```

When a generic procedure is invoked, its name should be followed by a type in angle brackets.

```plaintext
Max<int> (10,20);
```

The compiler of Escle programming language generates a new code for every typed instance of a generic subroutine. During compilation, the list of all possible instances of a generic subroutine is created, and then the code is generated for every procedure in this list.

Thus, for example, the following code is generated for subroutine `Max` with integer arguments.

```plaintext
procedure Max_int(int x, int y):int;
    if (x>=y) then { return x; }
    else { return y; }
end;
```
3 Using CoCo/R Compiler Compiler for generation of scanner and parser

3.1 Attribute grammars

Let us introduce the notion of an \textit{L-attribute grammar}, used by CoCo/R Compiler Compiler.

\textbf{Definition.} [1–3] A \textit{context-free grammar} is a quadruple
\[ G = \langle \mathcal{N}, \mathcal{T}, S, \mathcal{P} \rangle, \]
where:
\begin{itemize}
  \item $\mathcal{N} = \{N_1, \ldots, N_n\}$ is a set of nonterminal symbols;
  \item $\mathcal{T} = \{t_1, \ldots, t_m\}$ is a set of terminal symbols;
  \item $S = \{S\}, S \in \mathcal{N}$ is a start symbol (axiom);
  \item $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a set of rules of the form
        \[ P_j = A \rightarrow \omega, \quad A \in \mathcal{N}, \quad \omega \in (\mathcal{N} \cup \mathcal{T})^*, \quad j = 1, \ldots, k \]
\end{itemize}

\textbf{Definition.} [1–3, 8] An \textit{attribute grammar} is a quadruple
\[ \text{ATG} = (G, A_s, A_i, \mathcal{R}), \]
where:
\begin{itemize}
  \item $G = \langle \mathcal{N}, \mathcal{T}, S, \mathcal{P} \rangle$ is a context-free grammar;
  \item $A_s = \{\alpha^s_1, \ldots, \alpha^s_p\}$ is a finite set of synthesized attributes;
  \item $A_i = \{\alpha^i_1, \ldots, \alpha^i_q\}$ is a finite set of inherited attributes, and $A_s \cap A_i = \emptyset$;
  \item $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_r\}$ is a finite set of semantic rules.
\end{itemize}

Let us denote:
\begin{itemize}
  \item by $\alpha(X)$ the attribute of symbol $X$;
  \item by $\alpha(X) \in A_i(X)$ the fact that attribute $\alpha(X)$ is inherited;
  \item by $\alpha(X) \in A_s(X)$ the fact that attribute $\alpha(X)$ is synthesized.
\end{itemize}
In an attribute grammar, every symbol \( X \in N \cup T \) has a corresponding set \( A_s(X) \) of synthesized attributes and a corresponding set \( A_i(X) \) of inherited attributes.

Consider a rule of the form \( X \rightarrow X_1 X_2 \cdots X_n \). In an attribute grammar, each such rule has a corresponding finite set \( R(P) \) of semantic rules of the form

\[
\begin{align*}
\alpha(X_i) &= f(\beta(X_{j_1}), \gamma(X_{j_2}), \ldots, \chi(X_{j_m})) \\
0 &\leq j_1, j_2, \ldots, j_m \leq n, \\
1 &\leq i \leq n, \text{ if } \alpha(X_i) \in A_i(X_i), \quad i = 0, \text{ if } \alpha(X_i) \in A_s(X_i)
\end{align*}
\]

A semantic rule defines the value of attribute \( \alpha \) of symbol \( X_i \), based on the values of attributes \( \beta, \gamma, \ldots, \chi \) of symbols \( X_{j_1}, X_{j_2}, \ldots, X_{j_m} \), respectively.

**Example.** [3] Consider a grammar of arithmetical expressions

\[
G_1 = \langle N = \{E\}, T = \{i, +\}, S = \{E\}, P = \{P_1, P_2\} \rangle
\]

with the following set of rules.

\[
\begin{align*}
P_1 &= E \rightarrow E + E \\
P_2 &= E \rightarrow i
\end{align*}
\]

For every symbol \( X \) in every rule of the grammar, define a synthesized attribute \( \alpha \), referring to the value of a term in an arithmetic expression; this shall be denoted as follows.

\[
\alpha(E) = E.value
\]

Then the semantic rules of the rules of this grammar shall have the following form.

\[
\begin{align*}
E^0 \rightarrow E^1 + E^2 & \quad | \quad E^0.value = E^1.value + E^2.value \\
E^0 \rightarrow i & \quad | \quad E^0.value = i.value
\end{align*}
\]

Here symbols \( E^0, E^1, E^2 \) represent different instances of nonterminal \( E \); the superscripts are used to differentiate between values of the attributes in the semantic rules. Note that there is no syntactic difference between symbols \( E^0, E^1, \) and \( E^2 \).

In the rules of the grammar \( G_1 \), symbol “+” belongs to the syntax of the language generated by this grammar, while symbol “+” in the semantic rules corresponds to the arithmetic operation of addition.
Example. [3] Consider the following grammar of integer and real variables declaration.

\[ G_2 = \langle N = \{\text{Lang, Declaration, VarType, Ident}\}, T = \{\text{int, real, id, ,}\}, S = \{\text{Lang}\}, \mathcal{P} = \{P_0, P_1, P_2\} \]  

with the following rules in the EBNF-form.

\[
\begin{align*}
P_0 &= \text{Declaration} ::= \text{VarType} \text{ Ident} \\
P_1 &= \text{VarType} ::= \text{int} | \text{real} \\
P_2 &= \text{Ident} ::= \{\text{ident,} \} \text{ ident}
\end{align*}
\]

The corresponding context-free grammar has the following rules.

\[
\begin{align*}
P'_0 &= \text{D} \rightarrow \text{T} \text{I} \\
P'_1 &= \text{T} \rightarrow \text{int} \\
P'_2 &= \text{T} \rightarrow \text{real} \\
P'_3 &= \text{I} \rightarrow \text{I}_1 \text{,i} \\
P'_4 &= \text{I} \rightarrow \text{i}
\end{align*}
\]

Let us define for every symbol \( X \) in the rules of the grammar \( G_2 \) an attribute \( \alpha(X) \), representing the type of the variable. This, similarly to the previous example, shall be denoted as \( \alpha(X) = X\.type \).

The following semantic rules can be defined for this grammar.

\[
\begin{align*}
D & \rightarrow \text{T} \text{I} & \text{I}.\text{type} & := \text{T}.\text{type} \\
T & \rightarrow \text{int} & \text{T}.\text{type} & := 1 \\
T & \rightarrow \text{real} & \text{T}.\text{type} & := 2 \\
I^0 & \rightarrow I^1,i & I^1.\text{type} & := I^0.\text{type}; \text{AddToTable}(i.\text{type}, I.\text{type}) \\
I & \rightarrow i & \text{AddToTable}(i.\text{type}, I.\text{type})
\end{align*}
\]

Definition. [1, 4, 8] An attribute grammar is called \( L\)-attribute, if each inherited attribute \( \alpha \) of a symbol \( X_j \) in the right-hand side part of a rule \( X \rightarrow X_1 X_2 \ldots X_n \) only depends on:

- attributes of symbols \( X_1, X_2, \ldots, X_{j-1} \), which are to the left of \( X_j \);
- inherited attributes of \( X \).

In the case of an \( L\)-attribute grammar, all its attributes can be computed in the parse tree top-down from left to right. Thus, it is possible to compute the attributes simultaneously with the top-down parsing [8, 11].

CoCo/R Compiler Compiler generates a recursive descent parser, and every subroutine corresponding to a nonterminal uses passed-by-value arguments for inherited attributes and passed-by-reference arguments for synthesized attributes [11, 13, 14].

17
3.2 Input format of CoCo/R

Given a description of an L-attribute grammar, CoCo/R Compiler generates a scanner and a parser for this grammar.

The description of a grammar starts with specifying its name after the keyword COMPILER, for instance, COMPILER Escle.

It is followed by a section describing “meta”-characters.

CHARACTERS
letter = "abcdefghijklmnopqrstuvwxyz".
digit = "0123456789".

Then the list of tokens, to be recognized by the scanner, is specified.

TOKENS
ident = letter{letter|digit}.
number = ['+'|'-'] digit {digit}.

The following standard notation is used in description of tokens [3–5, 7–11]:

• an expression in curly braces may repeat 0 or more times (Kleene closure [3]);
• a vertical bar delimits alternatives;
• an expression in square brackets may repeat 0 or 1 time.

After this, the syntax of comments and symbols to be ignored are specified. The ignored symbols are considered as blanks by the generated parser.

COMMENTS FROM "//" TO "\n"
IGNORE "\n" + "\t"

The core of the description of a grammar is the section PRODUCTIONS. In the right-hand side part of a rule, one can use nonterminal symbols, tokens, string constants, as well as special symbols denoting repeating fragments, optional fragments and alternatives. Semantic actions, as well as the actions concerned with the code generation, are specified within the parentheses of the form (. and .).

The following example is a description of an expression.

Expression<out string type>
(. 
   CommandKinds op=CommandKinds.cmd_nop;
string type1="";
);

SimpleExpression<out type>
{
    Rel0p<out op> SimpleExpression<out type1>
    (.
        type=CompateTypes(type, type1, OpKinds.op_rel);
        Code.Add( new Command(op) );
        type=typ_bool;
    .)
};

Here the following attributes are specified in angle brackets (<, >):

- synthesized attribute of nonterminal Expression (<out string type>, representing the type of an expression);
- inherited attributes of nonterminal SimpleExpression (<out type> and <out type1>, representing the types of subexpressions);
- inherited attribute of nonterminal RelOp (<out op>, representing a relation operation).

Further processing of the attribute values (type=CompateTypes(type, type1, OpKinds.op_rel);) allows one to implement the semantic check of correctness of arithmetic and Boolean expressions.

4 Semantic analysis

4.1 Context conditions

Context conditions are those syntactic rules of a programming language, that one usually specifies informally [5].

Below is the list of several context conditions that have to be checked during the analysis of a program written in Escl programming language:

- any identifier used in a program should only be declared once [10];
- when variables are initialized, the types of the variables and of the expression assigned should be compatible [10];
- the types of formal and actual parameters of subroutines should be compatible [10];
in conditional statements if, and loop operators while and repeat, the condition can only be represented by a Boolean expression;

- in the headers of loop operators loop and for, only integer expressions can be used.

Note that names of variables, formal parameters of subroutines and local variables do not have to be all unique; the only restriction is that:

- names of variables are unique;
- names of formal parameters are unique within a subroutine;
- names of local variables are unique within a subroutine.

A complete list of semantic errors recognized by the compiler of Escle programming language is given in the appendix.

4.2 Operational semantics

The operational model of dynamic semantics describes how programs in a given programming language should be executed on a virtual machine. Usually, a virtual machine is defined as an automaton able to perform a finite number of formally defined operations that change its state. The semantics of every construction of a programming language is defined by the changes that take place in the state of the automaton after the execution of this construction [8].

In order to use operational method for a complete description of a semantics of a programming language, one has to have the following two components [8, p. 98]:

- a translator, to transform the constructions of a programming language into commands of the virtual machine;
- an interpreter of commands of the virtual machine.

Since the compiler for Escle programming language is implemented for the Microsoft .NET platform, the virtual machine is the virtual machine of Microsoft .NET, which uses the commands of Common Intermediate Language (CIL).

Below are given the translation functions for arithmetic and Boolean expressions. Denote by $CA$ the translation function for arithmetic expressions, and by $CB$ the translation function for Boolean expressions.
Denote by $\text{CS}$ the translation function for statements of Esce programming language, and by $\text{CE}$ the “generalized” translation function for expressions (arithmetic, Boolean, string, etc.)

\[
\begin{align*}
\text{CS}[[\text{set } x = a]] &= \text{CA}[[a]] : \text{stsfld } x \\
\text{CS}[[\text{set arg } x = a]] &= \text{CA}[[a]] : \text{starg } x \\
\text{CS}[[\text{set local } x = a]] &= \text{CA}[[a]] : \text{stloc } x \\
\text{CS}[[\text{set } x[a_0] = a]] &= \text{1dsfld } x : \text{CA}[[a_0]] : \text{CA}[[a]] : \text{stelem} \\
\text{CS}[[S_1; S_2]] &= \text{CS}[[S_1]] : \text{CS}[[S_2]] \\
\text{CS}[[a]] &= \text{CA}[[a]] : \text{pop} \\
\text{CS}[[\text{if } b \ S_1 \text{ else } S_2]] &= \text{CB}[[b]] : \text{brfalse Label1} : \text{CS}[[S_1]] : \text{br Label12} : \text{Label1} : \text{CS}[[S_2]] : \text{Label12} : ; \\
\text{CS}[[\text{while } b \ \text{do } S_1]] &= \text{Label1} : \text{CB}[[b]] : \text{brfalse Label12} : \text{CS}[[S_1]] : \text{br Label1} : \text{Label12} : ; \\
\text{CS}[[\text{repeat } S_1 \ \text{until } b]] &= \text{Label1} : \text{CS}[[S_1]] : \text{CB}[[b]] : \text{brfalse Label1}
\end{align*}
\]
\[ CS[[\text{loop a do } S_1]] = \text{local } i : \text{ldc.1} : \text{stloc} \ i : \text{Label1. } \ \ \ \ C.A[[a]] : \text{ldc.1} : \text{add} : \text{ldloc} \ i : \text{cgt} : \text{brfalse Label2} : \ \ \ \ CS[[S_1]] : \text{ldc.1} : \text{ldloc} \ i : \text{add} : \text{stloc} : \text{br Label1} : \text{Label2}. \]

\[ CS[[\text{for } i \ \text{in } [a_1, a_2] \ \text{do } S_1]] = \ \ \ \ C.A[[a_1]] : \text{stsfld} \ i : \text{Label1. } \ \ \ \ ldsfld \ i : \text{C.A}[[a_2]] : \text{ldc.1} : \text{add} : \text{clt} : \text{brfalse Label2} : \ \ \ \ CS[[S_1]] : \text{ldsfld} \ i : \text{ldc.1} : \text{add} : \text{stsfld} \ i : \text{br Label1} : \text{Label2}. \]

\[ CS[[\text{proc}(e_1, \ldots, e_n)]] = \text{CE}[[e_1]] : \ldots : \text{CE}[[e_n]] : \text{call} \ proc \]
\[ CS[[\text{func}(e_1, \ldots, e_n)]] = \text{CE}[[e_1]] : \ldots : \text{CE}[[e_n]] : \text{call} \ \ \ \ func : \text{pop} \]

5 Code generation for Microsoft .NET byte-code

5.1 Common Intermediate Language

The compiler for Escle programming language generates the byte-code in Common Intermediate Language (CIL). This can be done in a much simpler way than generation of native machine code.

Common Intermediate Language is the unified byte-code language for Microsoft .NET platform and actively uses stack [6]: all commands in CIL push and pop operands from the stack. The advantage of this approach is that there is no need of register allocation.

Common Intermediate Language is a hardware-independent assembler language, and in order an .NET-assembly can be executed, the CIL-code of this assembly is translated to the native machine code by the JIT-compiler [6].

The compiler for Escle programming language first generates a representation of a program in an intermediate language, which is then further transformed to the CIL-code. This is done by using classes of the standard namespace System.Reflection.Emit as follows.
5.2 An example of Microsoft .NET byte-code generation

Let us outline how a .NET-assembly is generated and show an example of generated code for variable declarations.

The generation of a new assembly starts with creating an instance of the class `AssemblyBuilder`, which is defined in the standard namespace `System.Reflection.Emit`. Then, using methods of the class `AssemblyBuilder`, the necessary modules of the assembly are created (in the implementation described in this paper, only one module is created in an assembly). Within the modules, one creates types (instances of the class `TypeBuilder`), and within the types, one creates constructors (`ConstructorBuilder`), methods (`MethodBuilder`) and fields (`FieldBuilder`).

The generation of the byte-code is done using the members of the class `ILGenerator`, which allows for creation instructions for a certain method.

Consider the fragment of a code in programming language C# that generates a new assembly, one class with a single field (corresponding to a `variable` in Escale programming language) and a static method `Main`, which is the entry point to the assembly.

```csharp
using System;
using System.Threading;
using System.Reflection;
using System.Reflection.Emit;

class CodeGenerator {
    
```
static void Main() {
    AppDomain a = Thread.GetDomain();
    AssemblyName aname = new AssemblyName();
    aname.Name = "nvhello.exe";
    AssemblyBuilder asmb = a.DefineDynamicAssembly(
        aname, AssemblyBuilderAccess.RunAndSave);
    ModuleBuilder mdl = asmb.DefineDynamicModule("nvhello.exe", "nvhello.exe");
    ProgramClass =
        mdl.DefineType("MyClass", TypeAttributes.Class);
    FieldBuilder fieldBuilder =
        ProgramClass.DefineField("x", typeof(Int32),
            FieldAttributes.Static | FieldAttributes.Public);
    MainMethod = ProgramClass.DefineMethod("Hello",
        MethodAttributes.Static | MethodAttributes.Public, typeof(void), null);
    MethodInfo ConsoleWriteLineMethod =
        ((typeof(Console)).GetMethod("WriteLine",
            new Type[]{ typeof(System.String) }));
    ILGenerator il = mainMethod.GetILGenerator();
    il.Emit(OpCodes.Ldstr, "Escle");
    il.Emit(OpCodes.Call, ConsoleWriteLineMethod);
    il.Emit(OpCodes.Ret);
    mdl.CreateGlobalFunctions();
    asmb.SetEntryPoint(mainMethod,
        PEFileKinds.ConsoleApplication);
    asmb.Save("nvhello.exe");
}
}

5.3 Structure of a generated assembly

As a result of its work, the compiler for Escle programming language generates a standard Microsoft .NET assembly in the form of an executable .exe-file. Such an assembly has a single class and the name of this class is the same as the name of the program in Escle programming language, which is given after the keyword `program`. For every global variable in a program in Escle, there is a corresponding static field in the class. The assignment of initial values to global variables is made in the static method `InitializeVars()`, which is invoked in the beginning of the assembly execution. Subroutines of a program in Escle are represented as static methods.
of the class. Local variables and arguments of subroutines are represented accordingly within the appropriate methods of the class.

Invariants are represented as static methods of the class, and the commands invoking this methods are added after every assignment statement.
References


5. S. Sverdlov. Languages of programming and methods of translation, in Russian. (St.-Petersburg, 2007).


Appendix. The list of semantic errors recognized by the compiler for Escle programming language

1. A vector cannot be invoked as a function
2. Index of a vector should be an integer expression, not ...
3. Local variable ... should be of type ...
4. Local variable ... is not declared in subroutine ...
5. Local variable with name ... is redeclared
6. (Implementation-dependent) Local variables cannot be vectors
7. Method ... with arguments ... does not exist
8. Loop counter with name ... cannot be declared, because subroutine ... already has a variable with this name
9. Variable ... cannot be declared without an initial value
10. Incompatible types — ... and ...
11. Integer expression expected, but not ...
12. Boolean expression expected, but not ...
13. String expression expected, but not ...
14. Local variable declaration should start with specifying type
15. Main subroutine of the program not found
16. Argument ... is not declared in subroutine ...
17. Argument ... redeclared
18. (Implementation-dependent) Arguments cannot be vectors
19. Variable ... not declared
20. Variable ... is not a vector
21. Variable ... is a vector, and only elements of vectors can be assigned a value
22. Variable ... is a vector, and for vectors the operations are defined only for elements
23. Variable ... redeclared

24. Subroutine ... is a function and should return a value; statement 'exit' is not allowed within a function

25. Subroutine ... cannot be declared with modifier 'main', since the main procedure has been already defined

26. Procedure cannot return a value

27. Procedure with name ... has been already defined

28. Loop ad hoc counter is already declared as a local variable

29. Type of returned value (...) does not match the declared type (...) 

30. Function should return a value

31. (Implementation-dependent) A function cannot be invoked with modifier 'local' or 'arg'
Appendix. An example of generated byte-code

Consider the program for maximal vector element search, given in Listing 1 of this paper.

```
program MaxInArray;
int i=0;
int max=0;
int maxpos=0;
vector int a[5];
procedure Hello(); main;
  for i in [0,5] do
    System.Console.Write("Enter the value of the "+
      System.Convert.ToString(i) + "-th element: ");
  end;
  set max = a[0];
  set maxpos = 0;
  for i in (0,5] do // loop starts with i=1
    if (a[i] > max) then
      set max = a[i];
      set maxpos = i;
    end;
  end;
end;
```

The compiler for Escle programming language generates first the following intermediate code.

```
cmd_progr          MyProg
cmd_var_int        i
cmd_load_const_int 0
cmd_init_set_var i
cmd_var_int        max
cmd_load_const_int 0
cmd_init_set_var max
cmd_var_int        maxpos
cmd_load_const_int 0
cmd_init_set_var maxpos
```
cmd_array_int a, 6
 cmd_start_proc Hello :
 cmd_load_const_int 0
 cmd_set_var i
 cmd_mark_label label0
 cmd_load_var i
 cmd_load_const_int 5
 cmd_load_const_int 1
 cmd_add
 cmd_lt
 cmd_br_false label1
 cmd_load_const_string Enter the
 cmd_load_var i
 cmd_concat_string
 cmd_load_const_string -th element:
 cmd_concat_string
 cmd_load_var a
 cmd_load_var i
 cmd_call_net System.Console.ReadLine :
 cmd_set_elem_int
 cmd_load_var i
 cmd_load_const_int 1
 cmd_add
 cmd_set_var i
 cmd_br label0
 cmd_mark_label label1
 cmd_load_var a
 cmd_load_const_int 0
 cmd_load_elem_int
 cmd_set_var max
 cmd_load_const_int 0
 cmd_set_var maxpos
 cmd_load_const_int 0
 cmd_set_var i
 cmd_mark_label label2
 cmd_load_var i
 cmd_load_const_int 5
 cmd_load_const_int 1
 cmd_add
 cmd_lt
This intermediate code is then further transformed to the byte-code of Microsoft .NET platform.

```
.class private auto ansi MyProg.Class
    extends [mscorlib]System.Object
{

5The byte-code given here can be seen by using the tool ildasm.exe.
```
.field public static int32[] a  
.field public static int32 i  
.field public static int32 max  
.field public static int32 maxpos  

.method public specialname rtspecialname  
  instance void .ctor() cil managed  
  {  
    .maxstack 2  
    IL_0000: ldarg.0  
    IL_0001: call instance void 
      [mscorlib]System.Object::.ctor()  
    IL_0006: ret  
  } // end of method Class::.ctor  

.method private static void InitializeVars() cil managed  
  {  
    .maxstack 1  
    IL_0000: ldc.i4.0  
    IL_0001: stsfld int32 MyProg.Class::i  
    IL_0006: ldc.i4.0  
    IL_0007: stsfld int32 MyProg.Class::max  
    IL_000c: ldc.i4.0  
    IL_000d: stsfld int32 MyProg.Class::maxpos  
    IL_0012: ldc.i4 0x6  
    IL_0017: newarr [mscorlib]System.Int32  
    IL_001c: stsfld int32[] MyProg.Class::a  
    IL_0021: ret  
  } // end of method Class::InitializeVars  

.method public static void Hello() cil managed  
  {  
    .entrypoint  
    .maxstack 10  
    IL_0000: call void MyProg.Class::InitializeVars()  
    IL_0005: ldc.i4.0  
    IL_0006: stsfld int32 MyProg.Class::i  
    IL_000b: ldstrld int32 MyProg.Class::i  
    IL_0010: ldc.i4.5  
    IL_0011: ldc.i4.1  
    IL_0012: add  
    IL_0013: clt  
    IL_0015: brfalse IL_0063  
  } // end of method Class::Hello
IL_001a: ldstr "Enter the 
IL_001f: ldsfld int32 MyProg.Class::i
IL_0024: call string [mscorlib]System.Convert::ToString (int32)
IL_0029: call string [mscorlib]System.String::Concat (string,string)
IL_002e: ldstr "-th element: 
IL_0033: call string [mscorlib]System.String::Concat (string,string)
IL_0038: call void [mscorlib]System.Console::Write(string)
IL_003d: ldsfld int32[] MyProg.Class::a
IL_0042: ldsfld int32 MyProg.Class::i
IL_0047: call string [mscorlib]System.Console::ReadLine()
IL_004c: call int32 [mscorlib]System.Int32::Parse(string)
IL_0051: stelem.i4
IL_0052: ldsfld int32 MyProg.Class::i
IL_0057: ldc.i4.1
IL_0058: add
IL_0059: stsfld int32 MyProg.Class::i
IL_005e: br IL_000b
IL_0063: ldsfld int32[] MyProg.Class::a
IL_0068: ldc.i4.0
IL_0069: ldelem.i4
IL_006a: stsfld int32 MyProg.Class::max
IL_006f: ldc.i4.0
IL_0070: stsfld int32 MyProg.Class::maxpos
IL_0075: ldc.i4.0
IL_0076: stsfld int32 MyProg.Class::i
IL_007b: ldsfld int32 MyProg.Class::i
IL_0080: ldc.i4.5
IL_0081: ldc.i4.1
IL_0082: add
IL_0083: clt
IL_0085: brfalse IL_00d1
IL_008a: ldsfld int32[] MyProg.Class::a
IL_008f: ldsfld int32 MyProg.Class::i
IL_0094: ldelem.i4
IL_0095: ldsfld int32 MyProg.Class::max
IL_009a: cgt
IL_009c: brfalse IL_00c0
IL_00a1: ldsfld int32[] MyProg.Class::a
IL_00a6: ldsfld int32 MyProg.Class::i
IL_00ab: ldelem.i4
IL_00ac: stsfld int32 MyProg.Class::max
IL_00b1: ldsfld int32 MyProg.Class::i
IL_00b6: stsfld int32 MyProg.Class::maxpos
IL_00bb: br IL_00c0
IL_00c0: ldsfld int32 MyProg.Class::i
IL_00c5: ldc.i4.1
IL_00c6: add
IL_00c7: stsfld int32 MyProg.Class::i
IL_00cc: br IL_007b
IL_00d1: ldstr "Maximum = 
IL_00d6: ldsfld int32 MyProg.Class::max
IL_00db: call string [mscorlib]System.Convert::ToString (int32)
IL_00e0: call string [mscorlib]System.String::Concat (string,string)
IL_00e5: call void [mscorlib]System.Console::WriteLine (string)
IL_00ea: ldstr "Index: 
IL_00ef: ldsfld int32 MyProg.Class::maxpos
IL_00f4: call string [mscorlib]System.Convert::ToString (int32)
IL_00f9: call string [mscorlib]System.String::Concat (string,string)
IL_00fe: call void [mscorlib]System.Console::WriteLine (string)
IL_0103: call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()
IL_0108: pop
IL_0109: ret
} // end of method Class::Hello

} // end of class MyProg.Class
Some commands of CIL

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldnull</td>
<td>none</td>
<td>load constant null</td>
</tr>
<tr>
<td>ldc.ml</td>
<td>none</td>
<td>load integer number -1</td>
</tr>
<tr>
<td>ldc.0 - ldc.8</td>
<td>none</td>
<td>load integers from 0 to 8</td>
</tr>
<tr>
<td>ldc.i4</td>
<td>int32</td>
<td>load an integer number</td>
</tr>
<tr>
<td>ldc.r8</td>
<td>float64</td>
<td>load a real number</td>
</tr>
<tr>
<td>dup</td>
<td>none</td>
<td>duplicate the value on top of the stack</td>
</tr>
<tr>
<td>pop</td>
<td>none</td>
<td>pop out the value from top of the stack</td>
</tr>
<tr>
<td>add</td>
<td>none</td>
<td>addition of two values on top of the stack</td>
</tr>
<tr>
<td>sub</td>
<td>none</td>
<td>subtraction</td>
</tr>
<tr>
<td>mul</td>
<td>none</td>
<td>multiplication</td>
</tr>
<tr>
<td>div</td>
<td>none</td>
<td>division</td>
</tr>
<tr>
<td>rem</td>
<td>none</td>
<td>remainder</td>
</tr>
<tr>
<td>and</td>
<td>none</td>
<td>bitwise “AND”</td>
</tr>
<tr>
<td>or</td>
<td>none</td>
<td>bitwise “OR”</td>
</tr>
<tr>
<td>xor</td>
<td>none</td>
<td>bitwise “XOR”</td>
</tr>
<tr>
<td>and</td>
<td>none</td>
<td>conjunction</td>
</tr>
<tr>
<td>or</td>
<td>none</td>
<td>disjunction</td>
</tr>
<tr>
<td>xor</td>
<td>none</td>
<td>exclusive or</td>
</tr>
<tr>
<td>ldsfld</td>
<td>f</td>
<td>load the value of a static field f</td>
</tr>
<tr>
<td>stsfld</td>
<td>f</td>
<td>save the value on top of the stack into the static field f</td>
</tr>
</tbody>
</table>