Manual for NanoMorpho

Aser Kroma
Janus Bjarki Birgisson

April 19, 2024

Abstract

Table of contents

Contents

1 Introduction

NanoMorpho is a compiler. it is a simple variant of Morpho. Morpho is a multi-paradigm, object-oriented programming language.

Nanomorpho reads program text from an input file and writes "assembly code" in an output file. The compiler is a parser, an intermediate code generator and a final code generator. The parser both factors and produces a intermediate code which is then sent into the final code generator.

2 Setting up and Using

The compiler can be found and cloned from github using the following

```
Git \$ Git clone [insert link later]
```

After cloning the compiler, follow the following steps:
To begin with, cleaning is recommended using:

```
rm -Rf *~ *.class *.masm *.mexe
```

Then compile using:

```
javac *.java
```

Finally, to run a program file, for example 'general.nm' which is in the same directory, use the following:

```
java NanoMorphoParser general.nm > general.masm
java -jar morpho.jar -c general.masm
java -jar morpho.jar general
```

3 The Language

3.1 Elements of the Language

3.1.1 Comments

The compiler allows commenting by using ;;; followed by the comment. An example of a comment would be:

```
;;; this is a comment
```

Both Comments and white spaces are ignored by the compiler. Comments can be anywhere that white space is allowed, that is, between any elements of the language.

3.1.2 Keywords

The keywords of this version of **NanoMorpho** are:
if, else, elseif, while, true, false, null, return, var

3.1.3 Single Character Tokens

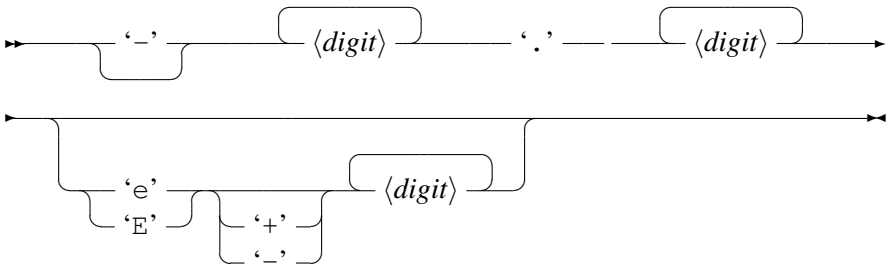
NanoMorpho has the following Single character tokens (special symbols): '(', ')', '{', '}', ',', ';', '='. These single character tokens also act as delimiters.

3.1.4 More Language Elements

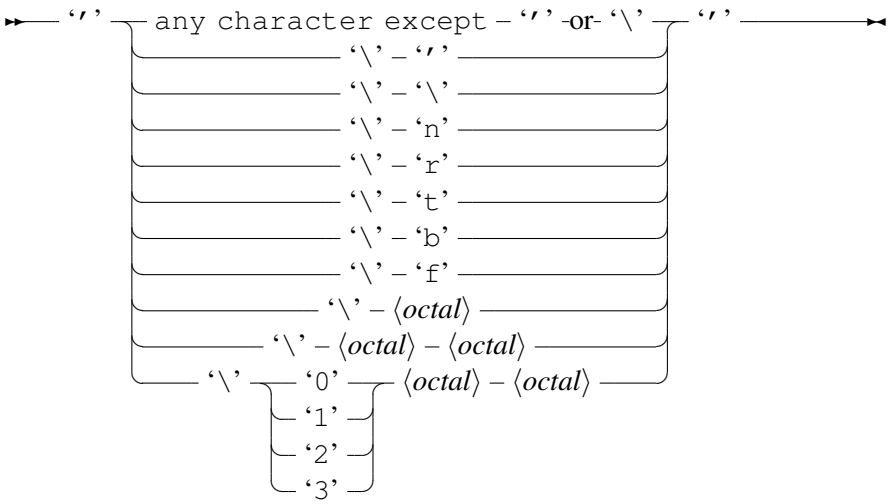
⟨*integer*⟩:

⟨*digit*⟩:

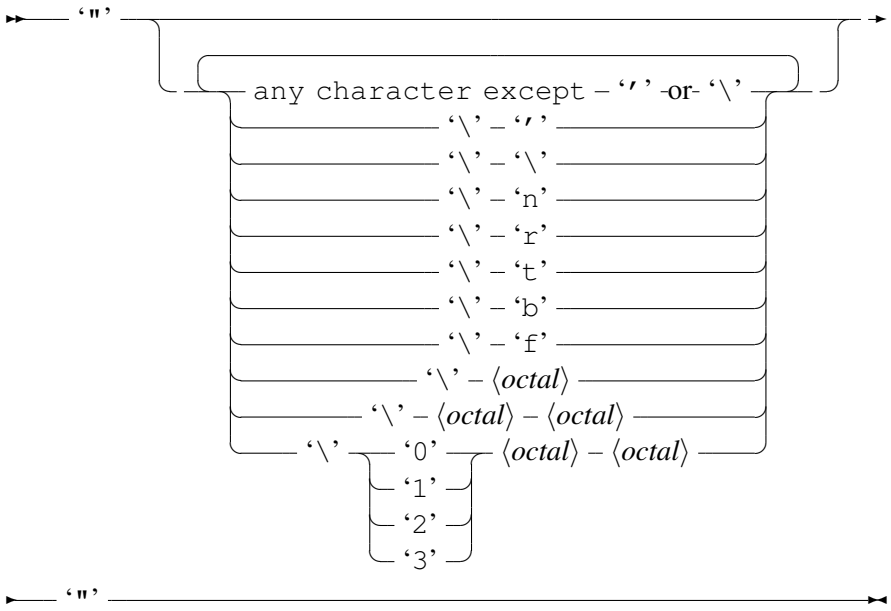
⟨*float*⟩:



⟨*char*⟩:



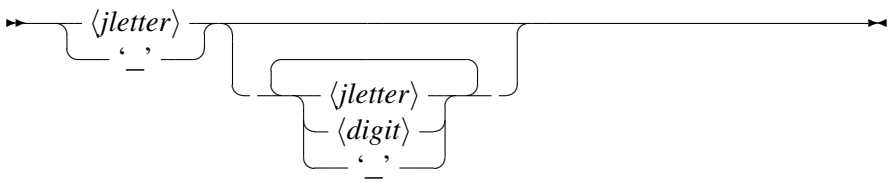
⟨*string*⟩:



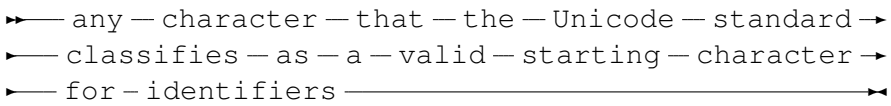
⟨*octal*⟩:

⟨*literal*⟩:

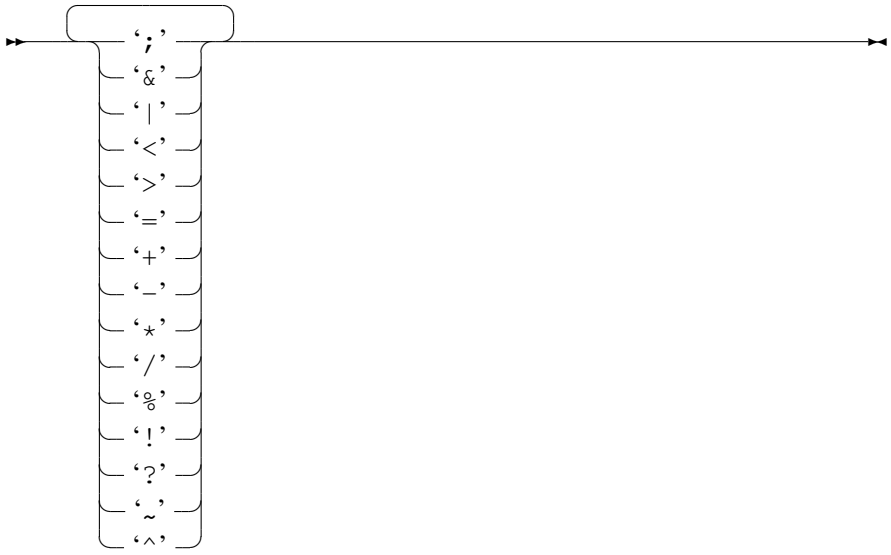
⟨*name*⟩:



⟨*jletter*⟩:



⟨*opname*⟩:



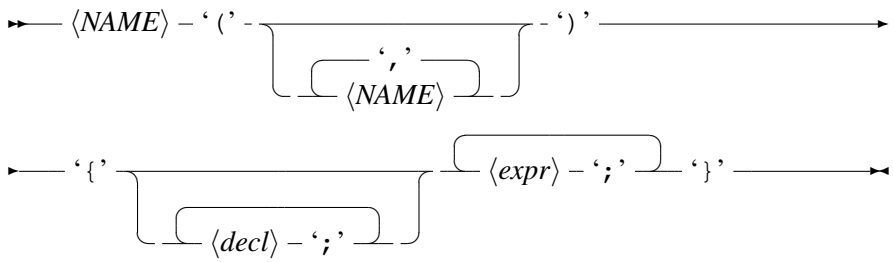
3.2 Grammar

3.2.1 Program

⟨*program*⟩:

3.2.2 Function

⟨*function*⟩:



3.2.3 Bodies

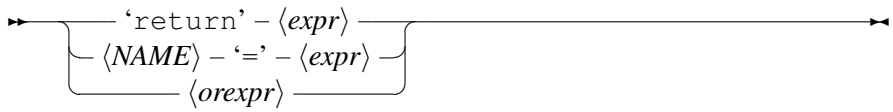
⟨*body*⟩:

3.2.4 Declarations

⟨*decls*⟩:

3.2.5 Expressions

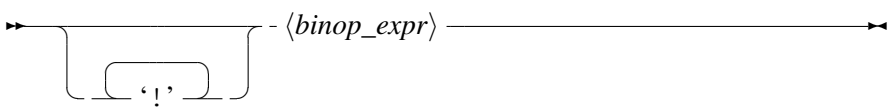
⟨*expr*⟩:



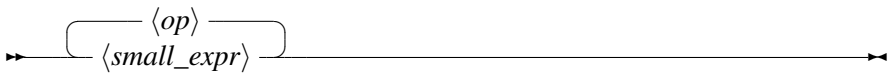
⟨*or_expr*⟩:

⟨*and_expr*⟩:

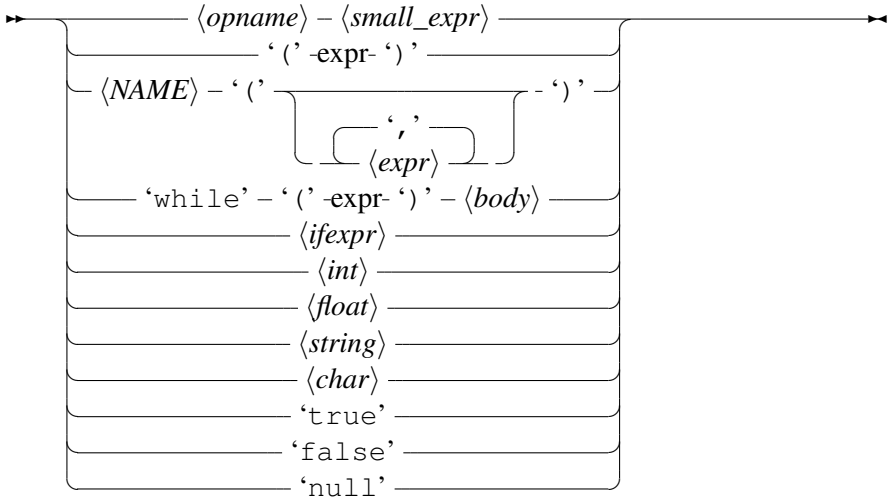
⟨*not_expr*⟩:



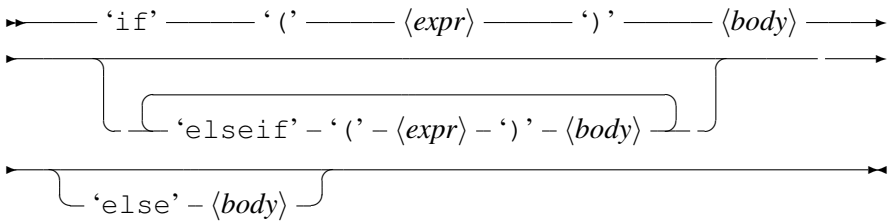
⟨*binop_expr*⟩:



⟨*small_expr*⟩:



⟨*ifexpr*⟩:



4 Semantics

4.1 Values

All values in NanoMorpho are references to Java Objects.

4.2 Variables

NanoMorpho supports local variables and parameters so local variables and parameters store information local to that specific function call. Variables are declared using the keyword var and following var any number of variable names can be declared as long as they are comma separated i.e. (var i, x, y).

example

```
fibo(n)
{
        var i,f1,f2,tmp;  ;;;here i, f1, f2, tmp are all null

        f1 = 1;           ;;;here f1 is assigned the value 1
        f2 = 1;           ;;;here f2 is assigned the value 1
        i = 0;            ;;;here i is assigned the value 0
}
```

4.3 Expression Semantics

4.3.1 The null Expression

The null expression is used to demonstrate that a variable has not been given any value.The literal null is the same as the null literal, the value of the null expression is the Java null reference.

4.3.2 The true Expression

The true expression is used for boolean arithmetic. The value of the true literal is the Java Boolean Object that has the value TRUE.

4.3.3 The false Expression

The false expression is used for boolean arithmetic. The value of the false literal corresponds to the Java Boolean Object that has the value FALSE.

4.3.4 Integer Literals

Integer literal are used to represent any whole number, there are no limits to how big the number is. The Syntax of the integer literals is defined by the integer syntax diagram in 3.1.4. the value of an integer literal is a corresponding Java Integer object, a Java Long object or a Java BigInteger object.

4.3.5 Floating Point Literals

Floating Point Literal represents a floating-point number, which is made up of one or more digits, followed by a decimal point, followed by one or more digits. The syntax diagram of the floating point literals can be found in 3.1.4. The Floating point literal's value is a corresponding Java Double Object.

4.3.6 Character Literals

The Character Literal represents a character enclosed in single quotes, and it supports various escape sequences. It can include any single character, except an unescaped single quote or backslash. Inside the quotes, it allows for:

- Normal characters directly (like a, b, etc.)
- Escape sequences for common whitespace and control characters
- Escaped single quotes and backslashes
- Octal escape sequences, which are specified using up to three octal digits (0 through 7), allowing for the representation of characters by their ASCII values in octal format.

The definition ensures that character literals can include a wide range of characters in a controlled way through direct use, escape sequences, or octal codes. The character literal's value is a corresponding Java Character Object.

4.3.7 String Literals

Is a string enclosed in double quotes. It can include any character except an unescaped double quote or backslash. Inside the string, it supports escape sequences like \b, \t, \n, \f, \r, escaped quotes (\", \'), escaped backslashes (\\), and octal escapes (up to three octal digits). The string literal's value is a corresponding Java String Object.

4.3.8 The return Expression

The return expression returns the value following it. For example, the following function would return 25 when called. It returns the value to the continuation of the current function.

Return expression example

```
test()
```

```
{
        return 25;
}
```

4.3.9 Logical Expressions

NanoMorpho has logical operators &&, || and !, which refer to the and, or and not logical operations.

4.3.9.1 AND expressions

The AND expression is represented by the operator &&, it is used in between a sequence of expressions. The AND operator is left associative, and the extent of it is to check whether all the expressions it is in between are true and returns true if so, otherwise false.

Refer to chapter 3.2.5. for the syntax diagram.

4.3.9.2 OR expressions

The OR expression is represented by the operator ||, it is used in between a sequence of expressions. The OR operator is left associative, and the extent of it is to check whether any of the expressions it is in between are true and return true if so, otherwise it returns false.

Refer to chapter 3.2.5. for the syntax diagram.

4.3.9.3 NOT expressions

The NOT expression is represented by the operator !, refer to chapter 3.2.5. for syntax diagram of not expressions. The NOT operator is left associative, and the extent of it is to check whether the expression after it is false and returns true if so, otherwise returns false.

4.3.10 Call Expressions

To call a function, the name of the function is used and arguments are passed to it depending on how the function is created. A function call can be done to a function with 0 or more parameters. Below is an example of a function that is called from main, the function is being called and passed 3 arguments, two of which are expressions to be evaluated. A function is called, following that the expressions (passed as arguments to the callee) are evaluated as they appear, from

left to right. Then the function is called and passed those evaluated expressions as arguments.

Call expression example

```
func(x,y,z)
{
        var b;
        b = x*y-z;
        b;
}

        main()
        {
                writeln("func(8*2,6-3,4)="++func(8*2,6-3,4));
    }
```

Output: 44

4.3.11 Binary Operators

The binary operations in NanoMorpho utilize a function call to a function which is passed the two arguments which that operation has, for example, '1+2' would return 3, as the operation + has two arguments 1 and 2. Most of the binary operators are left associative and each has a different precedence. refer to the table below for precedence and the associativity of the binary operations. The higher the precedence of the binary operator, the sooner it is applied. For example, the expression '28-25' is evaluated by calling the function '-' with arguments 28 and 25 and then the result would be the integer value 3.

Precedence	First letter	Associativity
7	'*', '/', '%'	Left
6	'+' or '-'	Left
5	'<', '>', '!' or '='	Left
4	'&'	Left
3	'|'	Left
2	':'	Right
1	'?', '~' or '^'	Left

4.3.12 Unary Operators

The unary operators have the highest precedence. Unary operators are followed by one expression (argument) and they are right associative. The unary operation is a function of one argument, which could be for example '-(2)' would be a unary operation where the unary operator '-' is applied to the integer literal 3.

4.3.13 The if Expression

Refer to chapter 3.2.5 to see the if expression syntax diagram and view its pattern. The if expression can occur in 4 different ways, just one if clause, followed by an else clause or followed by one or more elseif clauses followed by 0 or 1 else clause. The if and elseif clauses have one condition, when true that if or elseif's body is executed, otherwise the else's body is executed or nothing in case of a non-existent else clause. see below for an example of each instance of the if expressions.

In general the body of an IF clause is evaluated only if the cond is true, otherwise the elseif's cond is checked and if true the body of that elseif is evaluated, otherwise the else's body is evaluated. In the case of an If clause that does not have an else expression and the cond is false then false is returned. So the value of the return is the value of the cond.

One if clause, without an else

```
_____________________ if expression example _____________________

if(cond)
{
        body;     ;;;if the condition above is true
                     then the body is executed.
}
```

One if clause, with an else

```
_____________________ if expression example _____________________

if(cond)
{
        body;     ;;;if cond is true then this
                     body is executed.
}
else
{
```

```
        body;   ;;;if the cond above is false, then
                    this body is executed.
}
```

One if clause, with 1 or more elseif clauses and an else clause

```
_____________________ if expression example _____________________

if(cond)
{
        body;    ;;;if cond is true then this
                    body is executed.
}
elseif(cond2){
        body;    ;;;if cond is false and cond2 is true,
                    then this body is executed
}
...         ;;;0 or more elseif clauses
else
{
        body;   ;;;if the cond and cond2 from above are false,
                then this body is executed.
}
```

One if clause, with 1 or more elseif clauses without an else clause

```
_____________________ if expression example _____________________

if(cond)
{
        body;    ;;;if cond is true then this
                    body is executed.
}
elseif(cond2)
{
        body;    ;;;if cond is false and cond2 is true,
                    then this body is executed
}
...         ;;;0 or more elseif clauses
```

4.3.14 The while Expression

The condition in the while (s1) is checked and if true the body will be evaluated, otherwise the execution of main continues without executing the body of while, as long as the while's condition is true, the body of the while will keep executing. In the example below, as long as y is bigger than i, i will double in each iteration, and once y is no longer bigger than i, the main function prints i. In the case of a false condition, false is returned, i.e. x = while(false)1;; x will be assigned false.

while expression example

```
main()
{
        var i, y;
        i = 1;
        y = 20;
        while( y>i )
        {
            i = i * 2;
            y = y - 1;
        };
        writeln(i);
}
```

4.4 Functions and Programs

In order to have a functioning, compilable NanoMorpho program, you must one main method without any arguments, which has some execution in it and if needed one or more functions. To execute a function, it must be called from the main function, after that the function called from main can keep calling other functions.

Functions that can be called in the main are either functions defined in the same program or in the basis module (refer to the Morpho Manual for an overview of the basis module functions).

testCode

```
fibo(n)
{
        var i,f1,f2,tmp;
        f1 = 1;
        f2 = 1;
```

```
        i = 0;
        while( i<n )
        {
                tmp = f1+f2;
                f1 = f2;
                f2 = tmp;
                i = i+1;
        };
        f1;
}

main()
{
        writeln(1:2:3:null);
        writeln("fibo(35)="++fibo(35));

}
Output:
[1,2,3]
fibo(35)=14930352
```
