MiniPascal

A Project for the course Code Generation

Paavo Hemmo

24.05.2020

Table of contents

Running the compiler

Required installations

In order to build the project, you need to have **dotnet** installed.

The compiler uses **GCC** to compile C.

The compiler assumes that gcc is installed in the location **/usr/bin/gcc**. You can find out where gcc is installed on your computer by typing the command **which gcc** to the command prompt. If the location is different that **/usr/bin/gcc**, you need to change the code from the file **src/miniPascal/CodeGeneration/Generator.cs** at the **CreateExecutable()** method. Unfortunately I did not have the time to handle this better.

Running

The source code can be found from https://github.com/Pate1337/MiniPascal.

The compiler is implemented using C#. You can run the project in src/miniPascal/ with the command:

dotnet run <filename>

A filename can be an absolute path of the file, which allows you to compile a file from different location, from where the compiler is located. For example:

dotnet run /Users/paavohemmo/projects/code.txt

Or you can compile a file which is in the same location as the compiler with

dotnet run code.txt

dotnet run will build the project, and create an executable file “miniPascal” in src/miniPascal/bin/Debug/netcoreapp3.1/. The executable can be executed with the command:

./miniPascal <filename>

To print an Abstract Syntax Tree, the user can use the argument *-ast.*

Test for the Scanner can be run in src/miniPascalTests/ with the command *dotnet test*.

Output files

Running the compiler will create a **.c**-file and an **.exe**-file to the same location, where the **miniPascal**-file is. The **.c**-file contains the C-code, which the compiler has created.

When building the code, the **.exe**-file is executed automatically after a successful build.

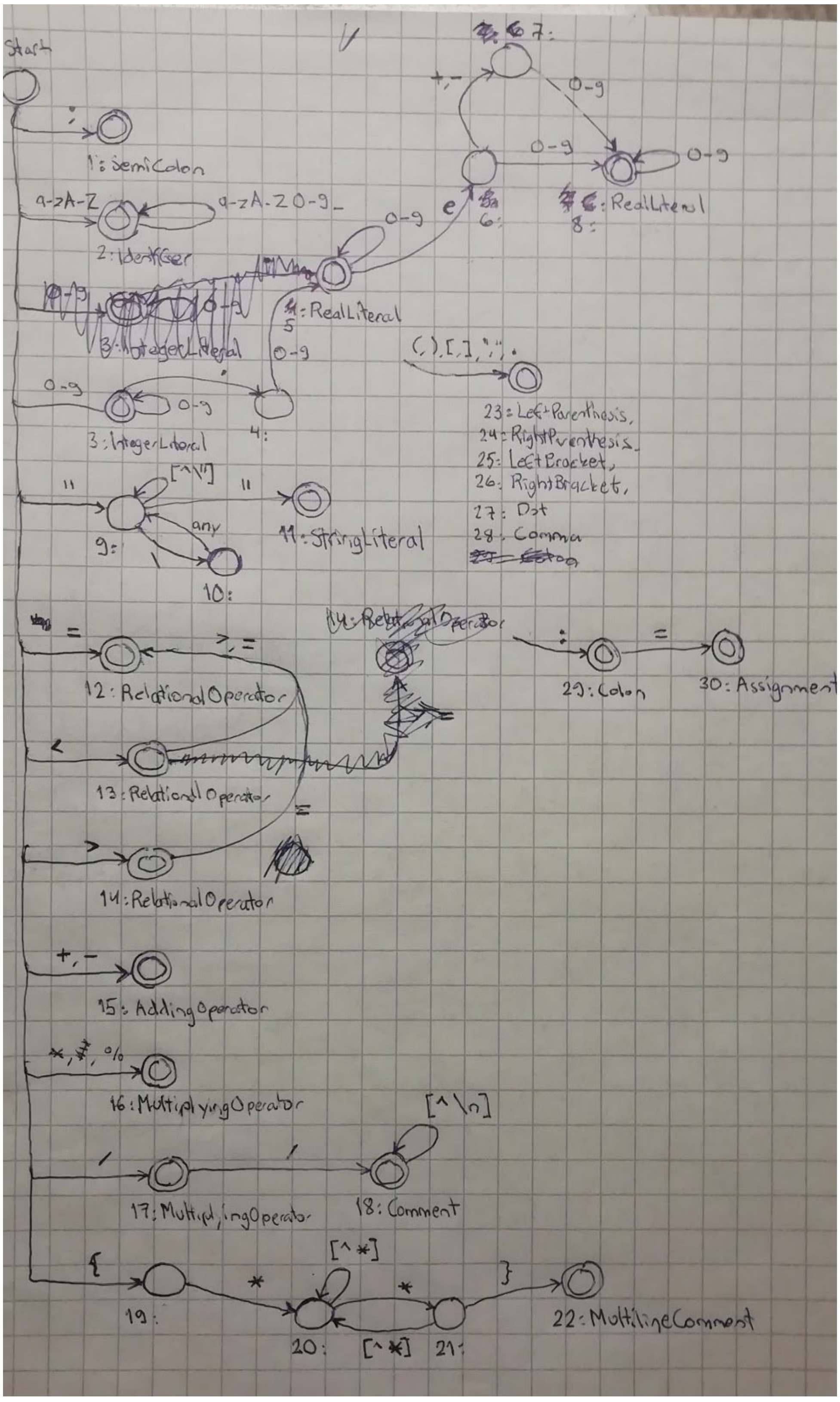
The **.exe**-file will have the same name as the original **miniPascal**-file.

A test file **test.pp** can be compiled with the command *dotnet run test.pp*. The file is located in src/miniPascal/. It contains a few sample programs.

About miniPascal

MiniPascal is a compiled language. The compiler has been written in C#. The target language of the compiler is simplified C.

Token patterns



miniPascal features

This chapter is user instructions for using miniPascal. We will start off by writing a simple Hello World program. I try to cover as many features of the language I possibly can in this short period of time.

Hello World

```
program helloworld;

begin
writeln("Hello world!");
end .
```

The first line of the miniPascal file needs to be program <programname>;.
Each statement requires a semicolon at the end. The mainblock must be wrapped inside the begin - end . followed by a dot. In miniPascal statements do not return a value. And adding a return statement to the main block would cause an error. The writeln -method can be given multiple arguments, and all of them will be printed right after each other.

Case non-sensitivity

MiniPascal is a case non-sensitive language. The following example demonstrates case non-sensitivity:

```
program non_sensitive;

begin
var a : integer;
a := 1;
writeln(A);
end .
```

The program outputs 1. MiniPascal treats every character of the code as lower case letters, and does not care about wether characters are lower or upper-case.

Predefined identifiers

MiniPascal has predefined identifiers. In the example the predefined identifier writeln is used as a variable.

```
program predefined_identifier;

begin
var writeln : integer;
writeln := 2;
writeln(writeln);
end .
```

The program prints 2.

Types

MiniPascal has in total 6 built-in-types: integer, boolean, real, string and the corresponding arrays. Void is also defined as the return value of procedures, but the syntax of miniPascal does not allow the user to declare it. The following example shows how the user can declare variables of different types. The size of integer in miniPascal is the same as int in C, which is normally 4 bytes. On some machines, the size may vary, but it does not break miniPascal. Boolean is the same size as integer. Real is the same as float in C. Strings have dynamic size, each character is 1 byte. I just noticed that booleans were supposed to be true or false in miniPascal. In my implementation 0 means false and any other integer means true.

```
program types;

begin
var a : integer;
var b : boolean;
var c : string;
var d : real;
var e : array[2] of integer;
end .
```

Assignments

Assignment statement uses the := syntax. Boolean is a type in miniPascal, but there is no boolean literal. Booleans need to be assigned the way the following example does it.

```
program assignments;

begin
var a : integer;
a := 1;
var b : boolean;
b := (1 = 1);
var c : string;
c := "Hello";
var d : real;
d := 3.14e-10;
var e : array[2] of integer;
e[0] := 1;
end .
```

MiniPascal only allows assignments between same types. Assigning an integer to a string variable for example would cause an error. Integers, reals, booleans and arrays are not initialized with values. They contain random values after declaration. String arrays are an exception, they are initialized with null characters.

Reals

I did not have the time to implement reals to the backend of the code. They are type checked by the frontend, but because of the lacking backend implementation, the compiler will throw an uncatched error, caused by lacking stack elements. I would have used floats for real types, because in C floats also have the E-notation, which is also a feature in miniPascal.

Arrays

Assignment between two arrays of same type is also possible in miniPascal. The array on the left-hand side will adjust it's size to match the one on the right-hand side, and the values of right-hand side array are copied to the LHS.

```
program arrays;

begin
var a : array[] of integer;
var b : array[2] of integer;
b[0] := 1;
b[1] := 2;
a := b;
writeln(a[0]);
writeln(a[1]);
writeln(b[0]);
writeln(b[1]);
end .
```

The program above prints the values 1, 2, 1, 2. The example also demonstrates how arrays can also be declared without a given size. Arrays in miniPascal are completely dynamic. Arrays can even be declared with a variable as the size. Arrays have a **size**-method, which returns the size of the array. The example below prints 3.

```
program arrays;

begin
var i : integer;
i := 3;
var a : array[i] of integer;
writeln(a.size) // Prints: 3
end .
```

Printing an array is made easy in miniPascal.

```
program arrays;

begin
var i : integer;
i := 3;
var a : array[i] of integer;
```

```
a[0] := 1;
a[1] := 2;
a[2] := 3;
writeln(a); // Prints [1,2,3]
var b : array[i] of string;
b[0] := "first";
b[1] := "second";
b[2] := "third";
writeln(b)  // Prints [“first”,”second”,”third”]
end .
```

Arrays can easily be concatenated with another array of same type using the **+** -operator.

```
program arrays;

begin
var a : array[2] of integer;
a[0] := 1;
a[1] := 2;
var b : array[1] of integer;
b[0] := 3;
a := a + b;
writeln(a)  // Prints [1,2,3]
end .
```

There is no other way to modify the size of arrays. Note that for example the code below does not work in miniPascal.

```
program arrays;

begin
var a : array[2] of integer;
a[0] := 1;
a[1] := 2;
a := a + 3;
writeln(a)  // OPERATION ERROR (line 7, column 7): Can not do operation
end .       // "+" between IntegerArray and Integer.
```

Strings

I am a fan of strings and wanted therefore to make it easy to play with strings in miniPascal. The **+** -operator, where the other operand is string, always results in a string value.

```
program strings;

begin
var a : string;
a := "string";
writeln(a + "string");  // Prints: stringstring
writeln(a + 1);         // Prints: string1
writeln(a + 0.2);       // Prints: string0.2 (Reals not implementedyet!)
var b : boolean;
b := (1 = 1);
writeln(a + b);         // Prints: string1
var arr : array[2] of integer;
arr[0] := 1;
arr[1] := 2;
writeln(a + arr)        // Prints: string[1,2]
end .
```

Strings, like arrays have a size -method. Size to a string returns the amount of character in a string.

```
program strings;

begin
var a : string;
a := "string";
writeln(a.size)     // Prints: 6
end .
```

String arrays

The implementation of string arrays was quite tricky to do without allocating unnecessary memory. A string is an array of characters, which are stored in a sequence in memory. The separator between strings is a null character '\0'. String arrays in memory in miniPascal look like this:

```
[ 'f','i','r','s','t','\0','s','e','c','o','n','d','\0' ]
```

The miniPascal compiler always keeps the size of the array (which is 2 in this case) and the offsets stored in variables. Offsets in the above case are 0 and 6.

Expressions

Another big flaw in miniPascal is, that I did not have the time to implement multiplication operations at all (This is because division without reals would be useless). Therefore only adding operations are supported for now (+, -, or). Parenthesis can be used to create a closed expression, which allows adding operations to be performed before multiplication operations. If multiplication was implemented, the correct arithmetic execution order does work in miniPascal.

Below are different types of expressions.

```
program expressions;

begin
var a : integer;
a := 4 + 2 - 8;
writeln(a);          // Prints: -2

var b : boolean;
b := "str" <> "str";
writeln(b);          // Prints: 0
b := b or 1;
writeln(b);          // Prints: 1
writeln(not b);      // Prints: 0
writeln(not not b); // Prints: 1

var c : array[2] of string;
c[0] := "first";
c[1] := "second";
var d : string;
a := (c[0].size + c[1]).size;
writeln(a)           // Prints: 7
end .
```

If-then-else

MiniPascal has an if-then-else statement for structural programming. The syntax is the following:

```
program structures;

begin
var a : integer;
a := 5;
if a < 6 then
    writeln(5, " is smaller than ", 6)
else
    writeln(5, " is not smaller than ", 6)
end .
```

The else-branch is optional. Note the use of semicolons. Semicolons are only used to separate statements in a begin - end -block. If for example there was a semicolon after the first writeln-method, the compiler would report a syntax error. That is because the compiler would think the statement was only an if-statement without the else-branch, and when the next token is “else”, the compiler reports that statement can not start with the keyword “else”.

While

The only way to loop in miniPascal is the while loop.

```
program structures;

begin
var a : integer;
a := 0;
while a <= 5 do
    begin
    if a > 2 then
        writeln(a); // Prints on separate lines: 3, 4, 5
    a := a + 1
    end
end .
```

Notice how the third and second to last lines do not require a semi colon. The last semicolon before the end of the block is optional in miniPascal.

An infinite loop can be achieved by adding an integer unequal to zero as the boolean expression. Integers as boolean expressions for if and while loops are inferred as booleans.

Read

Read Statements work flawlessly.

Assert

Assert works in frontend. On code generation it is not yet implemented.

Procedures and functions

This example shows the use of functions and procedures. The function prints “Hello” 10 times calling itself recursively. Notice the use of the reference parameter count.

```
program helloprogram;

procedure hello();
begin
    writeln("Hello")
end;

function printHello(var count : integer, times : integer) : integer;
begin
    hello();
    if count = times then return 0
    else
    begin
        count := count + 1;
        return printHello(count, times)
    end
end;

begin
var count : integer;
count := 1;
printHello(count, 10)
end .
```

Errors

The error messages created by the miniPascal compiler all contain a specific message of the error, the line number, column, filename and even the line itself where the error has occurred. A ^ sign is used to point at the correct location of the error in the line, the same way Python does it.

The scanner aka the lexer does not report any errors itself. Instead it creates a Token with an Error type, and then passes it to the parser. The parser will then catch the syntax error.

Syntax errors that occur in Parser are thrown errors. They are catched in the Main()-method, and therefore the execution of the compiler stops immediately after a syntax error.

Semantic errors happening during the type checking are the only errors that are properly reported to the user. There can be many, and all of them are reported. Some error messages should be much more specific than they are now. Very often the compiler only states, that a statement can not start with a symbol, without telling more information of why this is happening.

There are a few errors than are generated in the code generation part aswell. One example of those, is a negative index for an array, that is declared as a parameter (or function return type) of a procedure/function. They are calculated, because a parameter array's integer expression can not contain variables.

The errors happening during the compilation of the C-file and execution of the resulting .exe file are catched by the miniPascal compiler, with the help of error codes. For example when an index of an array element is negative, C will write the line exit(2). The 2 tells the miniPascal compiler that the error was caused by a negative index.

Code generation

The target language of the miniPascal compiler is C. My previous exprerience with C was close to zero, I didn't even take the "Programming in C" course. That made programming the compiler extremely slow. Overall I am pretty satisfied with the code. I could not find any memory leaks, and working with pointers finally started to clear out. Some big refactorings should be done to the C-code in the future, and while some functions return pointers, some just change the value of the pointer. These things should be cleared.

Optimizations

There are no big optimizations towards the generated C-code, and the compiling is done in just one pass. The biggest optimization is the generated variables for C.

Variables

Let’s look at an example of a very simple program in miniPascal.

```
program variables;

begin
var a : integer;
a := 4 + 5;
var b : integer
end .
```

This program generates the following **.c**-file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int main() {
int i0;
int i1=4;
int i2=5;
i1=i1+i2;
i0=i1;
return 0;
}
```

At first glance, it is clear that none of the included libraries are needed to compile this C-code. The including of only the needed libraries should be done in the future.

Now let’s look at the main()-function. The `int i0;` is the line `var a : integer;` translated. Integer variables in C-code start with the letter i, aswell as integer arrays. For booleans (and boolean arrays) the letter b is used. Reals use r (not yet implemented), and strings use s. The numbers start from 0 and keep growing until new variables are declared in C.

On the second line, a new variable `i1` is declared. The declaration is needed, because there are no free integer variables currently available. (`i0` needs to stay in the code, because it indicates the a variable of miniPascal code). `int i1=4;` stores the left operand of the expression `4 + 5`.

Again, the right operand needs to be stored in a variable, and because no free variables exists, new declaration is done in `int i2=5;`.

The next line `i1=i1+i2` is slighty optimized. Instead of declaring a new temporary variable for the result of the sum, the left operand variable is used instead to store the sum.

The line `i0=i1` then does the assignment into the variable a (`a := 4 + 5;`).

The C code has reached it’s last line, what happened to miniPascal line `var b : integer;`? The line is not written in the c-file, because now there are already declared variables available, so there is no need to write anything to C-file now.

If we added one more line `b := 2;` to the miniPascal code, the latest free variable is used. In this case it is the variable `i2`, and C would generate 2 more lines `i2=2;` and `i1=i2;`.

C-functions

Let’s look at another miniPascal program, and see what the generated C-file looks like:

```
program stringPlusInteger;

begin
writeln("hello " + 1);
end .
```

Now the big surprise of how the C-file looks like.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int IntegerSizeAsString(int i){
int t=1;
if(i<0) goto NEG;
if(i==0) goto END;
if(i==1) goto END;
t=(int)(ceil(log10(i)));
goto END;
NEG:;
t=-1*i;
t=(int)(ceil(log10(t))+1);
END:;
return t;
}
void IntegerToStringWithSizeCalc(int i,char** s){
int t=IntegerSizeAsString(i);
*s=realloc(*s,t+1);
sprintf(*s,"%d",i);
}
char* MakeStringVar(char* value){
char* s=malloc(strlen(value)+1);
strcpy(s,value);
return s;
}
char* ConcatStrings(char* s1,char* s2){
size_t l1=strlen(s1);
char* r=malloc(l1+strlen(s2)+1);
strcpy(r,s1);
strcpy(r+l1,s2);
return r;
}
int main() {
char* s0=MakeStringVar("hello ");
int i0=1;
char* s1=malloc(1);
IntegerToStringWithSizeCalc(i0,&s1);
char* s2=ConcatStrings(s0,s1);
printf("%s\n",s2);
return 0;
}
```

All this because of 1 miniPascal line of code? Yes, there is some optimization that needs to be done. But let's look at what is going on here.

At first, we can see that there are now functions other than the main() function aswell. These functions are written to the file, because they are needed in order to first convert the integer 1 into a string and then concatenating the resulting string "1" with the string "hello ".

The optimization is, that only the functions that are needed are written to file. Not every single function, which there are a few.

Final words

I only wish I had started writing the documentation a lot earlier. It is very frustrating not having the time to tell about all the things I have implemented in the language. Hopefully this documentation is enough to pass the course. I really loved this project, and plan to continue to grow this language even futher.