MicroC Compiler

Languages, Compilers and Interpreters

Eleonora Di Gregorio
520655

February 2021

Contents

1 Introduction **3**

2 Language description **3**
2.1 Types and Variable Declarations . . . 3
2.2 Function Declaration . . . 4
2.3 Expressions . . . 5
2.3.1 Literals . . . 5
2.3.2 Arithmetic Operations . . . 5
2.3.3 Relational Operations . . . 5
2.3.4 Logical Operations . . . 6
2.3.5 Accesses . . . 6
2.3.6 Functions calls . . . 6
2.3.7 Assignment . . . 6
2.4 Statements . . . 7
2.5 Lexical conventions . . . 8
2.5.1 Comments . . . 8
2.5.2 Identifiers . . . 8
2.5.3 Reserved Words . . . 8

3 Compiler Design and Implementation **8**
3.1 Scanner . . . 8
3.2 Parser . . . 9
3.3 Semantic analysis . . . 11
3.4 Code generation . . . 12

4 Building and Testing **13**
4.1 Requirement to build the code . . . 13
4.2 Building the code and using the compiler . . . 14

4.3 Testing of implementation 15
4.4 Directory structure 15
4.5 The source code 15

1 Introduction

This report aims to describe the realization of **MicroC compiler**, which is the assignment for "Languages, Compilers and Interpreters" course.
In the following sections, I will provide a language description 2, necessary to understand better the design choices illustrated in section 3, while at the end I will show how to use the language in your local machine 4.

2 Language description

MicroC is a simple subset of C language, it is statically typed and variables are statically scoped. For the sake of completeness, in the following part, we illustrate the main characteristics of the language, more details emerge reading the implementation description and the code.

2.1 Types and Variable Declarations

The primitive types allowed are:

- **int** : to denote an integer value of 32 bit.
- **float** : to denote a floating point value of 32 bit.
- **char** : to denote a character in ASCII code, represented in 8 bit.
- **bool** : it is a type that allows storing a boolean value among the literal "true" and "false".
- **void** : it is a type to indicate that a function does not return any values. We cannot declare a variable of this type.

Here is an example of declarations of variables of previous types.

```
int i; //default value 0
int j = 1;
float x; //default value 0.0
float y = 1.0;
char c; //default value '\x00'
char d = 'd';
bool b;// default value false
bool b = false;
```

Listing 1: Variable declaration

The variables can also be initialized during their declaration, otherwise they will have the default values, which are expressed with comments in [1].

Compound data types are:

- **array**

- **pointer**

Variable of these types example in the following listing[5].

```
int arr[]; //array of length 1
int arr1[5]; //array of length 5 initialized with default values
int arr2[8] = {1,2,3,4} //array of length 4
char arr3[] = {'a','b','c'}; //array of length 3
char str[] = "abc" //array of length 4
int t;
int *p = NULL; //null pointer
int *t ; // default initialization to NULL
int *s = &i;
```

Listing 2: Variable declaration

Arrays of length 0 are actually unusable, since they are not interchangeable with pointers and the language does not allow the use of dynamic allocation, for this reason, if length is not specified, we consider it 1. If the array's actual initialization is different from the declared dimension, the initialization is adapted to reach the declared length.
The language allows **strings** intended as arrays of chars terminated with the null byte '\0'.
If variables are declared outside of a function, they are considered as global, otherwise local. Global variables, following the C rules, can only be initialized with constant values.

2.2 Function Declaration

A function declaration follows the following schema.

```
return_type fun_name( type_param1 name1,...,type_paramn namen ){
//function_body
}
```

Listing 3: Function declaration

Functions can only return primitive types.
In each file, which is a program written in MicroC, there must be the ***main*** function; *main* can only return *void* or *int*. It can also have an *int* parameter that represents an integer taken from standard input.
Four library functions are provided, whose signatures follow.

```
void print(int n);
void printfl(float n);
void printch(char n);
int getint();
}
```

Listing 4: Library functions

The first one takes an integer input as a parameter and prints it on standard output; the second and the third one does the same thing respectively with a float and a char, and the last one takes an integer from standard input.

2.3 Expressions

The expression that can be evaluated by the language are listed briefly in the subsections. They are evaluated from left-to-right, following the traditional C-like style.

2.3.1 Literals

Lexical literals are the representation of fixed unit values that cannot be further reduced.

- Integer literal maps to an *int* datatype .
- Floating literal maps to a *float* datatype.
- Boolean literal maps to a *bool* datatype.
- Character literal maps to a *char* datatype.
- String literal maps to a *char array* datatype.

2.3.2 Arithmetic Operations

+	binary operation addition
-	binary operation subtraction
*	binary operation multiplication
/	binary operation division
%	binary operation modulus
-	unary operation negation

They are applied to operands that are of type *int* or *float*. In case of binary operators, they must be applied on same type arguments, and only the implicit casting from integer constant to float constant is implemented.

2.3.3 Relational Operations

==	binary operation equal
!=	binary operation not equal
<	binary operation less
<=	binary operation less equal
>	binary operation greater
>=	binary operation grater equal

The result of a relational operation is always a *boolean* value, they are applied to operands that are of type *int* or *float*. The two operands must have the same type, except for the case of constant *int* value that can be implicitly casted to *float*.

2.3.4 Logical Operations

&&	binary operation and
||	binary operation or
!	unary operation not

They can be applied only on *boolean* operands.

2.3.5 Accesses

Accesses can be of three types:

- Simple access to primitive type variables.
- Dereferenced access to a pointer using the operator *****.
- Access to an array position.

Examples follows in this listing.

```
1 print(j);
2 print(*s);
3 print(arr2[1]);
```

Listing 5: Variable declaration

2.3.6 Functions calls

Function calls follow the same rule of C. We remark that functions can only return data of primitive types.

2.3.7 Assignment

As in C the assignment allows to change the value associated to a variable and return the value assigned. The value that we are going to assign must be of the same type of the variable to which is assigned. The table shows the different ways to realize an assignment.

a = b	return b
a += b	means a = a + b, returns a + b
a -= b	means a = a - b, returns a - b
a *= b	means a = a * b, returns a * b
a /= b	means a = a / b, returns a / b
a %= b	means a = a % b, returns a % b
++a	means a = a + 1, returns a + 1
a++	means a = a + 1, returns a
- -a	means a = a - 1, returns a - 1
a- -	means a = a - 1, returns a

2.4 Statements

The statements supported by MicroC are:

- **expression** statement, one of the expression described above.
- **if-else** statement, the guard must be a *boolean*, the else branch can be present or not.

```
    if(j == 6){
        j++;
    }else{
        j--;
    }

```

Listing 6: if statement

- **while** statement, the guard must be a *boolean*.

```
    while(j>0){
        print(j);
        j--;
    }

```

Listing 7: while statement

- **do-while** statement, the guard must be a *boolean*.

```
    do{
        print(j);
        j++;
    } while(j<8)

```

Listing 8: do-while statement

- **for** statement, in the guard only assignment and not declaration is admitted.

```
     for (i = 0 ; i < 5 ; i = i + 1) {
        print(i);
    }
    for ( ; i < 5; ) {
        print(i);
        i = i + 1;
    }

```

Listing 9: for statement

- **return** statement.

2.5 Lexical conventions

2.5.1 Comments

MicroC supports single line and multi-line comments.

```
// single line comment
/*
multi line
comment
*/
```

2.5.2 Identifiers

Identifiers start with a letter or an underscore and then can contain letters, underscore and numbers.

2.5.3 Reserved Words

Keywords are: if, return, else, for, while,do, int, float, char, void, NULL, bool.
Other reserved words are: true, false, literals for boolean value.

3 Compiler Design and Implementation

The compiler is written in `OCaml`, using `ocamllex`, `menhir` and the `LLVM API`. The file from which we build the executable is `microc.ml`, where the different stages illustrated below are managed.

3.1 Scanner

The scanner has been realized using `ocamllex`, its implementation is in the file `scanner.ml`. The purpose of the scanner is transforming the input file into a sequence of tokens that are defined in the file `parser.ml`. We use an hashtable to keep all the reserved words 2.5.3 associated with the corresponding tokens.
The scanner is based on four rules:

- **token**. If we met:
  - an *int*, *float*, *char* literal, we generate a token carrying its value.
  - an identifier in the form expressed in 2.5.2, we first check if it is in the hashtable of reserved words, and eventually we return the token, otherwise we return the token ID with the identifier itself.
  - an operator (+, -, *, /, % , ==, !=, <, <=, >, >=,&&, ——, !, =, +=, -=, *=, /=, %=, ++, –), we return the corresponding token.
  - a symbol (‘(’,‘)’, ‘{’, ‘}’, ‘[’, ‘]’, ‘&’, ‘:’, ‘;’, ‘,’), we return the corresponding token.

- the symbol ", the control passes to string rule, that return a buffer containing the string that is stored in STRING token.
- the symbol //, the control passes to line_comment rule.
- the symbol /*,

In case of eof it generates the EOF token; in case of an illegal character that is not listed before, it raises a lexer error.

- **line_comment**. This rule identifies comments on one line, all symbols are ignored until the end of line symbol, when control goes back to the first rule.
- **comment**. This rule identifies comments on more lines, all symbols are ignored until the symbol */ , when control goes back to the first rule. It raises a lexer error if the comment is not closed and the file ends.
- **string**. This rule takes as argument a buffer and the lexbuffer. In the implementation provided the buffer length is 509, because it is the original max length for strings in C language. The rule recognises all the character of the string until the symbol ", when it passes the control back to the first rule. It raises a lexer error if the input file ends while reading the string or if it meets an illegal character.

3.2 Parser

The parser has been realized using `Menhir`, its implementation is in the file `parser.ml`. Parser's purpose is transforming the sequence of tokens, generated by the scanner, into an **abstract syntax tree**, whose implementation is given in the file `ast.ml` .
Given the description of the language, the grammar specification that follows is this one, where tokens with no semantic values are enclosed between quotes, e.g., '(', whereas tokens with semantic values are capitalized, e.g., INT. As usual the operator * means zero or more occurrences, + means one or more occurrences, and ? means at most one.

⟨*program*⟩ ::= ⟨*topdec*⟩* EOF

⟨*topdec*⟩ ::= ⟨*var*⟩⟨*init_var*⟩ '`;`' | ⟨*var*⟩ '`(`' ((⟨*var*⟩ '`,`')* ⟨*var*⟩)? '`)`' ⟨*block*⟩

⟨*var*⟩ ::= ⟨*type_def*⟩ ID | ⟨*type_def*⟩ ID '`[`'`]`' | ⟨*type_def*⟩ ID '`[`' INT '`]`'

⟨*type_def*⟩ ::= '`int`' | '`float`' | '`bool`' | '`char`' | '`void`' | ⟨*type_def*⟩ '`*`'

⟨*int_var*⟩ ::= ϵ | '`=`' ⟨*ex*⟩ | '`=`' '`{`' ((⟨*ex*⟩ '`,`')* ⟨*ex*⟩)? '`}`' | '`=`' STRING

⟨*block*⟩ ::= ‘{’ ⟨*inblock*⟩* ‘}’

⟨*inblock*⟩ ::= ⟨*var*⟩⟨*init_var*⟩ ‘`;`’ | ⟨*stmt*⟩

⟨*stm*⟩ ::= ‘`if`’ ‘`(`’⟨*ex*⟩ ‘`)`’⟨*stmt*⟩ | ‘`if`’ ‘`(`’⟨*ex*⟩‘`)`’⟨*stm*⟩ ‘`else`’ ⟨*stm*⟩
| ‘`for`’ ‘`(`’⟨*acc*⟩‘`=`’⟨*ex*⟩‘`;`’ ⟨*ex*⟩‘`;`’ ⟨*acc*⟩‘`=`’⟨*ex*⟩ ‘`)`’ ⟨*stm*⟩
| ‘`for`’ ‘`(`’ ‘`;`’ ⟨*ex*⟩‘`;`’ ‘`)`’ ⟨*stm*⟩
| ‘`while`’ ‘`(`’⟨*ex*⟩‘`)`’⟨*stm*⟩ | ‘`do`’ ⟨*block*⟩ ‘`while`’ ‘`(`’ ⟨*ex*⟩ ‘`)`’‘`;`’
| ⟨*ex*⟩ ‘`;`’ | ‘`return`’ ⟨*ex*⟩? | ⟨*block*⟩

⟨*acc*⟩ ::= ID | ‘`*`’ ⟨*acc*⟩ | ID ‘`[`’⟨*ex*⟩‘`]`’

⟨*ex*⟩ ::= ⟨*acc*⟩ | ⟨*assop*⟩ | ‘`&`’ ID | NULL | INT | FLOAT | CHAR | TRUE | FALSE | ‘`-`’ ⟨*ex*⟩ | ‘`!`’ ⟨*ex*⟩ | ⟨*ex*⟩ ⟨*binop*⟩ ⟨*ex*⟩ | ⟨*ex*⟩ ‘`<`’ ⟨*ex*⟩ | ⟨*ex*⟩ ‘`<=`’ ⟨*ex*⟩ | ID ‘`(`’ ((⟨*ex*⟩ ‘`,`’)* ⟨*ex*⟩)? ‘`)`’ | ‘`(`’ ⟨*ex*⟩ ‘`)`’

⟨*assop*⟩ ::= ⟨*acc*⟩ ‘`=`’ ⟨*ex*⟩ | ⟨*acc*⟩ ‘`+=`’ ⟨*ex*⟩ | ⟨*acc*⟩ ‘`-=`’ ⟨*ex*⟩ | ⟨*acc*⟩ ‘`*=`’ ⟨*ex*⟩ | ⟨*acc*⟩ ‘`/=`’ ⟨*ex*⟩ | ⟨*acc*⟩ ‘`%=`’⟨*ex*⟩ | ⟨*acc*⟩ ‘`++`’ | ⟨*acc*⟩ ‘`--`’ | ‘`++`’ ⟨*acc*⟩ | ‘`--`’ ⟨*acc*⟩

⟨*binop*⟩ ::= ‘`+`’| ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`==`’ | ‘`!=`’ | ‘`>`’ | ‘`&&`’ | ‘`||`’

Grammar conflicts are resolved by specifying the associativity and the precedence of the operators, precedence rules also manage problems like the shift/reduce conflict of the "else" branch over its absence.
<*var*>, <*init_var*>, <*block*> and <*binop*>, here presented as grammar rules, are managed with Menhir's `%inline` keyword, that causes all references to them to be replaced with their definitions. Their usage helps us in the generation of the nodes of AST, without adding more states to LR automaton.
Menhir allows us to associate code for each grammar production, so for each of them we produce a node of the abstract syntax tree, using the function `make_node`. Each node is an annotated node, it maintains the position of the element in the file in `loc`, the definition of the node itself and the id number. The last value is assigned incrementally using a counter, this value is always incremented even if the theory teaches us to keep variable declarations id, and then use the same for accesses. This value would be useful for future developments that are not related to the implementation of this project.
Since there is an evident correspondence between the productions of the grammar and the nodes of the AST, we only explain the more involving choices and those that bring changes to the provided `ast.ml` file.

- **Initialization of a variable**. The declarations nodes now have an optional list of expression node, that are the values for the initialization. It is implemented as a list, so it would be easier to manage both primitive type variable and arrays.
- **If without else branch**. It is realized using the if node with an empty block as else branch.
- **For**. It is realized using the while node. We create a block node, having as first operation the for initialization instruction, then the while node having as guard the conditional expression and, as body, the for body plus the loop expression.
  In the case of the for having only the conditional expression, we have a direct correspondence with while.
- **Do-While**. We create a block containing an inner block with the body and then the classic implementation of while block.
- **Relational operators**. All relational operators have a direct correspondence with BinaryOp in the AST, except for $<$ and $<=$ that are realized using the available operators and inverting the arguments.
- **Abbreviation for assignment operators**. They are all transformed in classical assignment according to their meaning, except for post decrement and post increment operators that are managed with new unary operation, in order to guarantee their correct returning value.
- **String**. Since strings are arrays of chars, when we initialize a string, we transform the literal string to a null-terminated sequence of expression nodes, each maintaining one character.
- **NULL**. NULL literal is represented using the integer literal -1.

3.3 Semantic analysis

When tha AST is generated by the parser, it is passed as argument of the function `check`, defined in file `semant.ml`. The semantic analysis is performed by visiting the AST and, for each node, a typed new one is created. The result of the semantic checker is a typed AST, as descibed in `tast.ml`, or an error, if one of the rules is violated.
The realization of a typed AST was not necessary, and actually we could have reached the same result in the code generation just using the AST, but it makes the cast operations and null pointer managing simpler in the codegen phase.
We describe the semantic analysis perfomed following the control flow of the semantic checker and mentioning the main aspects. First, a **symbol table** is instantiated, it is implemented as a list of maps. The first map keeps track of definitions in the global environment and than, for each function declaration, we add a new map to simulate the local environment, we remove it when its analysis ends. We also add/remove a new map every time we enter/exit a block in the souce code.
We add to the global environment the declarations of the library functions, the other

function declarations, that can be present only at the top level, and global variables.
For the global variables we check if the type is not *void* and if the initialization, if present, is made by constant expressions of valid types. When analysing an array we also check that the initialization matches the length declaration, otherwise we print a warning and we consider the declared length.
For the functions we check in order: the returned type, that can only be a primitive one, the name of the function, that must be unique, the names of the parameters which can't be repeated and, after adding them to the environment, we perform the semantic check of each statement in the body.
For the **block** analysis we follow the same principles of top level declarations, but in this case the initialization of variables is not forced to be constant. In particular, in each block, we check if the return statement has already been invoked, in that case we print a warning and the code that follows, even if it is analyzed, it won't be passed to the code generation step.
For the **if** statement we check if the guard is a boolean expression and, recursively, the two branches. Same approach is used for the **while** statement. Analysing the **return** statement, we can incur in an error if a void function returns a value.
The inspection of the expressions in the source program is realized through `check_expr` function. Errors pointed out in this scenario are the following ones:

- Access to a non defined variable.
- Dereferencing a non pointer.
- Access an index of a non array variable.
- Inconsistency of types, we ask for a certain type values and it is not of type requested.
- Operators applied on wrong typed operands.
- Function not defined.
- Call of an identifier that doesn't represent a function.
- Invalid number of parameters.
- Invalid type returned by a function.

Controls end with the verification of `main` definition, since MicroC does not support separate compilation, the file must contain a `main` function which returns *void* or *int* and has at most an `int` parameter.

3.4 Code generation

The last phase consists in the transformation of the TAST into an intermediate representation, which will be the LLVM IR. Therefore the code generation is realized using the LLVM API.

The idea is that we generate code while the TAST is visited and, at the end, we return an *llvm module*, which is the top-level container for all other LLVM Intermediate Representation. Since a detailed description of the process would mean a description of the API, only the support data structures and the the main features of the implementation are reported below.
Support data structure are:

- A table mapping a unary operators and the operand type in the LLVM function that implemets them.
- A table mapping a binary operators and the operands type in the LLVM function that implemets them.
- A symbol table that represents the environment and stores *llvalue* associated to functions and variables defined in the microC program. Global variables are defined using `define_global` function and are not present in the table.

When we generate the code for the `main` function, if it has a parameter, it is replaced with a variable having the same name and initialized with a call of the function `getint`, which means that the user must provide a parameter. The library function provided are only declared in the module, while their implementation is provided in the file `rt-support.c`, which must be linked for the building of the executable.

4 Building and Testing

4.1 Requirement to build the code

The code requires:

- OCaml >= 4.10.1
- Menhir >= 20200624
- ppx_deriving >= 4.5
- llvm >= 10.0.0

You can install the required dependencies via `opam`

```
$ opam install menhir ppx_deriving llvm
```

To use the compile you also need to install Clang.

4.2 Building the code and using the compiler

Typing `make` will generate a `microcc.native` executable:

```
$ make
```

To clean-up the folder, run:

```
$ make clean
```

To Parse and print the AST of a source file input.mc, run:

```
$ ./microcc -p input.mc
```

To perform semantic checks and print the TAST of a source file input.mc, run:

```
$ ./microcc -s input.mc
```

To print the generated code of a source file input.mc, run:

```
$ ./microcc -d input.mc
```

To place the output into file output.txt, run:

```
$ ./microcc -d input.mc -o output.txt
```

To optimize the generated code, rum:

```
$ ./microcc -O input.mc
```

To compile, run:

```
$ ./microcc input.mc
```

Once you have obtained your bitcode file a.bc, you have to link it with the object code of the runtime support.

```
$ make rtsupport
$ llvm-link a.bc rt-support.bc -o test.bc
$ llc -filetype=obj test.bc
$ clang test.o
```

Otherwise, to obtain directly an execuable, run:

```
$ make rtsupport
$ make compile input.mc
```

4.3 Testing of implementation

The directory `testsemant` contains some tests used to test semantic analysis. The following command execute all of them.

```
$ make testsemant
```

The directory `testcodegen` contains some tests used to test code generation. Each test case consists of a small MicroC program (the file with extension `*.mc`) together with the result that the program is supposed to produce (the file with extension `*.out`). The following command execute all of them and check differences with expected output.

```
$ make testcodegen
```

The directory `test` contains some tests used during the development, their purpose is reported in the comments.

4.4 Directory structure

Here is a description of content of the repository

```
src/                              <-- The source code lives here
Makefile                          <-- Driver for ‘make‘ (uses OCB)
_tags                             <-- OCamlBuild configuration
test/                             <-- Some programs used to test compiler
testsemnat/                       <-- Some programs to test semantic anlaysis
testcodegen/                      <-- Some programs to test code generation
```

4.5 The source code

The `src/` directory provides:

```
ast.ml              <-- Definition of the abstract syntax tree of MicroC
tast.ml             <-- Definition of the typed abstract syntax tree of MicroC
microcc.ml          <-- The file from which build the executable
parser_engine.ml    <-- Module to interact with the parser
codegen.ml          <-- Module that implements the code generation
util.ml             <-- Utility module
opt_pass.ml         <-- Module that implements some simple optimizations
rt-support.c        <-- The runtime support to be linked to bitcode files produced
parser.mly          <-- Menhir specification of the grammar
scanner.mll         <-- ocamllex specification of the scanner
semant.ml           <-- Module that implements the semantic checker
symbol_table.mli    <-- Interface of a polymorphic symbol table data type
symbol_table.ml     <-- Implementation of a polymorphic symbol table data type
```