MicroC Compiler

Languages, Compilers and Interpreters 2020/2021

Chiara Boni - 600159
University of Pisa

Contents

1 Introduction

The main goal of the *Languages, Compilers and Interpreters'* course was to analyze and study the behaviour of a generic compiler, in order to achieve the realization of an *ad hoc* compiler for a subset language of C: **MicroC**.

Some simplifications have been inserted for the educational purpose of this language, such as:

- It supports only *integers* (int), *characters* (char) and *booleans* (bool) as scalar values, array and pointers.
- There are no structures, unions, doubles, function pointer.
- There is no dynamic allocation of memory nor the use of multi-dimensional arrays.
- There is no shorthand for initialize variable during declaration.
- Functions can only return *void, int, bool, char*.
- Pointer arithmetic is not supported.
- Pointers and arrays are **not** interchangeable.
- It does not allow separate compilation.
- There are only two library functions available:

```
void print(int)
int getint()
```

Once the program is written with the **MicroC** language, it has to be processed by the compiler in order to be executed.

This project aims to present the phases needed to bring the program into its low-level structure.
Firstly, the *Scanner* and *Parser* take the program as input and recognize the **tokens**, i.e. the keywords, to create the *Abstract Syntax Tree* of the overall behavior of the program.
The *Static Analysis* aims to check that a given program obeys the scoping rules and the type system of the language; finally the *Code Generation* compiles the program into LLVM bitcode.

A further contribution included is to extend the previous description of the language with four new constructs:

- *Do-While* loops
- *pre/post* operators, i.e. ++ and –
- the possibility to abbreviate *assignment operators*, i.e. +=, -=, *=, /= and %=
- allowing the *floating point* arithmetic.

2 Scanner and Parser

The *Scanner* is the first component of the front-end of a compiler and it reads the program as input, in order to define the **tokens** that have meaning in the language. To better identify the important components of the language, it is necessary to give a more detailed explanation of the **MicroC** constructs.

2.1 Literals and Identifiers

MicroC allows four primitive types, such as *ingeters, char, booleans* and, with the extension implemented, *floats*.

- Integer literals are sequence of digits

  ```
  let digit = ['0'-'9']+
  ```

- Floating point literals

  ```
  let fdigit = digit+'.'digit+ (['E' 'e'] ['+' '-']? digit+)?
  ```

- Character literals have the form 'c', where c is a character

  ```
  let letter = ['a'-'z' 'A'-'Z']
  let lit_char = letter | ['0'-'9']
  ```

- Boolean literals are true and false

- As for identifiers, they can start with a letter or an underscore and then can contain letters, underscore and numbers.

  ```
  let identifier = ('_' | letter) (letter | digit | '_')*
  ```

2.2 Keywords

By **keywords** we mean the constructs of the language that will then take on meaning in the subsequent phases of the compiler.
They represent the expressiveness of the language as its possibility to implement *loops*, *declarations* of variables and *conditional constructs*.
To simplify the association between keywords and tokens, a hash table has been created.

```
let create_hashtable size init =
    let tbl = Hashtbl.create size in
    List.iter (fun (key, data) -> Hashtbl.add tbl key data) init;
    tbl

let keyword_table =
    create_hashtable 8 [
        ("if",      IF);
        ("else",    ELSE);
        ("return", RETURN);
        ("while",   WHILE);
        ("for",     FOR);
        ("do",      DO);
        ("int",     INT);
        ("float",   FLOAT);
        ("char",    CHAR);
        ("bool",    BOOL);
        ("void",    VOID);
        ("true",    LBOOL(true));
        ("false",   LBOOL(false));
        ("null",    NULL());
    ]
```

2.3 Operators and precedence

Operators mainly represent **arithmetic** and **boolean** operations.
There is no difference between an operator for integer values and the one with float values: the positive outcome of the operation will be proven by semantic checks.
The **operators** are:

```
&, +, -, *, /, %, =, ==, !=, <, <=, >, >=, &&, ||, !
```

The **precedence** of the operators follows the common rules of algebra, with the additional aid of the brackets.

```
(, ), {, }, [, ], &, ;
```

Finally, **MicroC** also allows to create portions of the code that will be ignored.

```
//...      single line comments
/* ... */ multi line comments
```

2.4 Syntax

Once the scanner has identified the keywords, it is the *Parser*'s job to analyze those tokens, report any syntax error in the code and compose the *Abstract Syntax Tree* of the program.
This first intermediate representation is useful to describe how the constructs are used and to show the general data-flow of the program.

```
Program ::= Topdecl* EOF

Topdecl ::= Vardecl ";"  | Fundecl

Vardecl ::= Typ Vardesc

Vardesc ::= ID | "*" Vardesc | "(" Vardesc ")" | Vardesc "[" "]" | Vardesc "[" INT "]"

Fundecl ::= Typ ID "("((Vardecl ",")* Vardecl)? ")" Block

Block ::= "{" (Stmt | Vardecl ";")* "}"

Typ ::= "int" | "char" | "void" | "bool"

Stmt ::= "return" Expr ";" | Expr ";" | Block | "while" "(" Expr ")" Stmt
    |    "for" "(" Expr? ";" Expr? ";" Expr? ")" Stmt
    |    "if" "(" Expr ")" Stmt "else" Stmt  | "if" "(" Expr ")" Stmt

Expr ::= RExpr | LExpr

LExpr ::= ID | "(" LExpr ")" | "*" LExpr | "*" AExpr | LExpr "[" Expr "]"

RExpr ::= AExpr | ID "(" ((Expr ",")* Expr)? ")" | LExpr "=" Expr | "!" Expr
    |  "-" Expr | Expr BinOp Expr

BinOp ::= "+" | "-" | "*" | "%" | "/" | "&&" | "||" | "<" | ">" | "<=" | ">=" | "==" | "!="

AExpr ::= INT | CHAR | BOOL | "NULL" | "(" RExpr ")" | "&" LExpr
```

Figure 1: The initital grammar

It is shown in the Fig. 1 the first proposed grammar for MicroC, where tokens with no semantic values are enclosed between quotes, i.e. "(", and tokens with semantic values are capitalized, i.e. INT.
This model, however, leads to have an **ambiguous grammar**, i.e. a context-free

grammar for which there exists a string that can have more than one leftmost derivation or parse tree.
Bringing it inside the language context, this might lead to **conflicts**:

- A **reduce/reduce** conflict occurs if there are two or more rules that apply to the same sequence of input.
- A **shift-reduce** conflict occurs in a state that requests both a shift action and a reduce action.

Another important aspect to avoid would be the **dangling else problem**, that occurs when the conditional constructs are nested and there is not a clear way to define which *else* statement belongs to which construct.

In order to define a better and not ambiguous grammar, the initial one has been re-written.

```
Program ::= Topdecl* EOF | error

Topdecl ::= Var ";" | Types ID "(" Var* ")" Block

Types ::= "int" | "float" | "bool" | "char" | "void"

Var :: = Types ID | Types "*" ID | Types "(" Var ")"
        | Types ID "[" "]" | Types ID "[" INT "]"

Block ::= "{" Cont* "}"

Cont ::= Stmt | Var ";"

Stmt ::= Selection_stmt | Loop_stmt | Block
        | "return" Expr ";" | Expr ";"

Selection_stmt ::=
  | "if" "(" Expr ")" Stmt
  | "if" "(" Expr ")" Stmt "else" Stmt

Loop_stmt ::=
  | "while" "(" Expr ")" Block
  | "do" "{" Stmt "}" "while" "(" Expr ")"
  | "for" "(" Lexpr "=" Expr ";" Expr ";" Expr ")" Stmt
  | "for" "(" ";" Expr ";" ")" Stmt

Expr ::= Rexpr | Lexpr
```

```
Rexpr ::= Primitive | Call | Assignment
  | Unary | Binary

Primitive ::= INT | FLOAT | CHAR | BOOL | NULL
  | "(" Rexpr ")" | "&" Lexpr

Lexpr :: = ID | "*" Lexpr | ID "[" Expr "]"

Call ::= ID "(" Expr* ")"

Assignment ::= Lexpr "=" Expr | Lexpr "+=" Expr
  | Lexpr "-=" Expr  | Lexpr "*=" Expr
  | Lexpr "/=" Expr | Lexpr "%=" Expr
  | "++" Lexpr | Lexpr "++" | "--" Lexpr | Lexpr "--"

Unary :: = "!" Expr | "-" Expr

Binary ::= Expr Op Expr

Op ::= "+" | "-" | "*" | "/" | "%"
  | "<" | ">" | "<=" | ">="
  | "==" | "!=" | "&&" | "||"
```

3 Semantic Analysis

Once the *Abstract Syntax Tree* has been built, it is necessary to run an additional check upon the program, focusing on the meaning of the constructs.
They might be syntactically correct but, at the same time, they might express forbidden actions and therefore turn out to be semantically wrong or even dangerous.

During this control phase it was necessary to create a *Symbol Table* capable of keeping track of the various scopes and constructs of the program, explained in the following section.
Thereafter, there will be a brief summary of the semantic rules applied for the **MicroC** language.

3.1 Symbol Table

MicroC adopts a *static scope* and it allows the use of a global one for variables and function declarations, united with the possibility to create nested scopes inside the blocks.

To make the management of the constructs more efficient, it was necessary to create an association map between the names and their types.
For this purpose, it is possible to use a *String Map*:

```
module STable = Map.Make(String);;
type 'a t = ('a STable.t) list
```

The symbol table has been modelled as a *list of maps*, due to the fact that in this way it is easier to manage nested environments.
Each cell of this list is a Map and it represents a single scope.

Several additional functions also have been implemented, whose *interface* is as follows:

```
type a' t
val empty_table : 'a t
val begin_block : 'a t -> 'a t
val end_block : 'a t -> 'a t
val add_entry : Ast.identifier -> 'a -> 'a t -> 'a t
val lookup : Ast.identifier -> 'a t -> 'a
```

- *empty_table*: it allows to create a new empty table.
- *begin_block*: it takes the table as input and it returns the table with a new block inserted.
- *end_block*: it returns the table without the first element.
- *add_entry*: it adds a new element inside the table by taking a string, a new element and the table itself as input.
  It raises a **DuplicateEntry** exception if the value is already present in the table.
- *lookup*: it searches for a value inside the table.
  It raises a **NotFoundEntry** exception if the value is not in the table, otherwise it will return it.

3.2 Semantic and typing rules

The MicroC language is a subset language of C, so it is reasonable to think that it implements semantic and typing rules that are similar to the C language itself.
One of the most important difference is that MicroC does not support *separate compilation*, therefore each file must contain the **main** function, which is the entry point of the program.
The signature of this function can only be *int*, when it returns an integer value, or *void* if it does not return any value.

```
int main()
void main()
```

The global scope is firstly filled with the declarations of the two library functions that will be then linked by the compiler, before the execution of the program.

Following the stratification of the *Abstract Syntax Tree*, the semantic checker analyzes each portion of the code, starting for the *top declarations*, which could only contain global variables and function declarations.
For this step is it necessary to check for **void** variables, because they are semantically not supported.
Each function, then, creates its scope which contains the *statements*.

For these constructs it is important to control if the guards of the *If* and *While* statements are *booleans*; meanwhile the *return* statement must match the type of the function, with more particular focus on the fact that functions can not return arrays or pointers.
If the function contains a *Block*, then it is necessary to create an inner scope, which can be filled by local variable declarations or other statements.

The main attention for the semantic checker is upon the *expressions*, due to the fact that the MicroC language does not allow any kind of casting for the operands: only operations between same type of value will be permitted.
In addition, the arithmetic operators expect only integer or float values, meanwhile the logical operators expect only boolean values.
Particular is the case for the array type, because in MicroC arrays and pointers are not interchangeable nor it is possible to assign arrays.

In order to avoid a verbose file for the semantic checker, a file *ErrorMsg* has been created which contains the strings for the errors that this phase could report.

If the program passes every semantic checks, then this phase returns an *Abstract Syntax Tree* that it is semantically correct and ready to be converted into the LLVM IR.

4 Code Generation

Finally, the last phase implemented was the code generation one, which takes the *Abstract Syntax Tree* as input, assumes it as semantically correct, and then it translates each AST node into the LLVM IR.
The construction for the expressions works in a *bottom-up* way, united with the one for the basic blocks and control-flow statements.
In this stage a *Symbol Table* is used as well, in order to keep track of the association between constructs and their respective llvalues.

4.1 Library functions

Since it has been assumed that the MicroC language offers two library functions to its users, they have to be implemented and then linked to the application for the run-time support.
Below there is the C implementation of the functions: the *getint* function is able to read from the *stdin* an integer value and then it returns it; meanwhile the *print* function reports to the *stdout* the integer value passed as a parameter.

```
#include <stdio.h>
#include <stdlib.h>

int getint(){
  int n = 0;
  scanf("%d", &n);
  return n;
}

void print(int n){
  printf("%d\n", n);
}
```

To make these functions visible inside the module, they have been declared as global, as shown below, but they have not been inserted into the *Symbol Table*, since it has already been established that there are no other functions called as them, therefore there is not a duplicate entry.

```
let print_decl =
  let print_t = L.function_type void_type [| int_type |] in
  L.declare_function "print" print_t llmodule

let get_decl =
  let getint_t = L.function_type int_type [| |] in
  L.declare_function "getint" getint_t llmodule
```

4.2 LLVM IR

The purpose of this stage is to translate MicroC code into LLVM IR code, which is a Single-Static Assignment form with the code organized as three-address instructions with the assumption of having infinite registers.
First, it is necessary to define the overall *context* and *module* for the application, which will contain prototypes, global variables and function definitions.

```
let llcontext = L.global_context()
let llmodule  = L.create_module llcontext "microc"
```

Then, aliases for the lltypes were used in order to include all the types that MicroC supports.

```
let int_type      = L.i32_type llcontext
let float_type    = L.float_type llcontext
let bool_type     = L.i1_type llcontext
let char_type     = L.i8_type llcontext
let void_type     = L.void_type llcontext
let array_type    = L.array_type
let pointer_type  = L.pointer_type
```

The overall construction is then left to the three main functions:

- *codegen_topdec*: this function is responsable for the global definitions of functions and variables.
  A *function* definition is made of a set of basic blocks and each of them is made of a sequence of instructions.
  Every parameter must then be added in the new scope of the function and, finally, it is possible to insert the terminator for the block, reporting the end of it.
  For the *variables*, since they are situated now in the top level, they are just added to the module and defined global.
  Finally they are added to the *Symbol Table*.

- *codegen_stmt*: entering more in the detail of the AST, this function creates the LLVM IR for the statements allowed by the language.
  If and *While* statements are similar in the sense that they are formed by the creation of new basic blocks, each representing a new branch, and then by the population of them with the instructions.
  It is also necessary to create a *conditional branch* that will allow the flow of the program to run into a specific branch, depending on the guard value.
  The *Return* statement has the purpose of adding a terminator to the current analyzed block.
  Ultimately, the *Block* statement might contain either *variable* declarations or more *statements*.
  As far for the variables, they are modelled by the use of memory through *load* and *store* instructions.
  In this case, before putting them into the *Symbol Table*, to keep track of their uses later on in the generation, it is necessary to initialize them with default values depending on their types.

- *codegen_expr*: this last function is responsible to generate the LLVM bitcode for the expressions.
  The *Literal* values are translated into their respective llvalues; meanwhile the various *Accesses* need more attention, due to the fact that it is necessary to distinguish whenever it is a pointer, an array or a simple variable.
  As far for the *Unary* and *Binary* operators, we must find the LLVM respective operator for them and build the operation.
  Ultimately, the *Call* expression is generated by searching, into the *Symbol Table*, for its identifier and then it is necessary to generate the code of each one of its parameters.
  A side note is inserted if the function in question is one of the library's one, because in this case it is just searched for into the module.

5 Building

Ultimately, all of the previous phases collide into the generation of the MicroC compiler.
As far for the *Scanner* and *Parser* stages, the main tools used were **Ocamllex**, that produces a lexical analyzer from a set of regular expressions, and **Menhir**, which is a LR(1) parser generator of the Ocaml programming language.
Meanwhile the code generation stage was created by using the **LLVM** Compile Infrastructure.

5.1 Commands

To generate the executable:

```
make
```

This command creats the *microc.native* executable that can be lauched with different options.

```
-p file.mc  - Create and print the AST
-s file.mc  - Perform semantic checks
-d file.mc  - Print the generated code
-o out.txt  - Place the output into a file
-O          - Optimize the generated code
-c file.mc  - Compile (default)
```

To link the external libraries:

```
make ext
```

Ultimately, to compile and run the file:

```
make run f=file.mc
```

To clean the folder from the generated files:

```
make clean
```

5.2 Description of the files

Inside the *MicroC* folder:

/src	The folder for the source files
/test	The folder for the various tests for the program
/doc	The folder for the Ocamldoc files
Makefile	The Makefile for the compilation of the program
README.md	File for the overall documentation

Inside the *src* folder of the project:

microcc.ml	The file from which build the executable
ast.ml	The AST structure
scanner.mll	Ocamllex specification of the scanner
parser.ml	Menhir specification of the grammar
parser_engine.ml	The module that interacts with the parser
semant.ml	Module that implements the semantic checker
codegen.ml	Module that implements the code generation
opt_pass.ml	Module that implements some simple optimizations
symbol_table.mli	Interface of a polymorphic symbol table data type
symbol_table.ml	Implementation of a polymorphic symbol table data type
util.ml	It contains some utily functions
error_msg.ml	The strings representing the semantic errors
rt-support.c	The run-time support to be linked to bitcode files