MicroC report

Niccolò Piazzesi
619619

University of Pisa, Department of Computer Science

Languages, Compilers and Interpreters

1 Introduction

In this report i will describe the general design and implementation of the MicroC compiler. MicroC is a simplified version of the C programming language. The project consisted in developing a functioning compiler for this language, using the LLVM infrastructure and the tools provided by the OCaml programming language. The report is structured in the following way:

- Frontend

  Presentation of the main structure of the scanner and parser, built using Ocamllex and the incremental API of Menhir.

- Symbol Table

  Design and implementation of the symbol table, used to manage namespaces both in semantic analysis and in code generation.

- Semantic analysis

  This section will focus on the module implementing rule and type checking of the language. I will present how well formed MicroC programs are defined and explain how the compiler detects malformed ones.

- Backend

  Description of the code generation phase, which uses the LLVM module and infrastructure to generate executable machine code.

- Conclusions and remarks

Language extensions The eight language extensions proposed (which can be found in [1]) have been all added to MicroC. In each section of the report i will explain how they have been implemented, and the reasoning behind certain choices. For each extension, at least two new unit tests have been added.

Other extensions The rt_support external library was extended with 4 new functions: getfloat, getcharacter, printfloat and printchar. These functions bring the same functionality of print and getint to the other basic types and were also used to test the implementation.

2 Frontend

2.1 Scanner

The scanner is implemented using ocamllex, a tool provided by OCaml itself to generate token recognizers. The module simply follows the general rules required by ocamllex, defining the regular expressions to identify the various tokens that will be used by the parser. The more relevant details are:

- Keywords are stored in an hash table, to simplify detection. When an alphanumeric identifier is recognized, we look it up in the table. If it's present we emit the corresponding keyword token, otherwise we recognize it as a generic identifier, which will be used for function, variable, and structs names.
- White space is distinct from new line characters such as \n and \r. This allows better error reporting by keeping track of the current line that is being read.
- String literals are recognized with a different rule than the one used for single characters. This was needed since in the string rule a buffer is used to build the literal. Having two independent rules can also be beneficial if we handle some specific characters (e.g. newline and white space) differently inside a string.
- Float literals can be written both in standard (0.1 , 1. , .1) and scientific (1e2 , 1E-3 , .1e2) notation.
- Each operator has its own token so, for example, + and ++ will generate two different tokens, PLUS and INCREMENT. This choice was made to avoid a more ambiguous grammar for the parser.

2.2 Parser

2.2.1 Ast changes

Before describing the parser itself, i will explain the additions and changes made to the Ast module to accomodate for the additions introduced:

- Variable declaration now takes an optional expression, representing the possible initial value. This change was also applied to the Dec node.

  ```
  Vardec of typ * identifier * expr option
  ```

- Struct declarations are wrapped in a StructDecl node. A struct is represented by the struct_decl record, where we store the name and the fields.

  ```
  Structdecl of struct_decl
  ```

  ```
  type struct_decl = {
          sname : string;
          fields :  (typ*identifier) list;
  }
  ```

- Do-while loops are represented by the DoWhile node. While the definition is identical to While, I chose to create a different entity because it simplified both parsing and managing the internal scope in later stages.

  ```
  DoWhile of expr*stmt
  ```

- AccField encodes field access of a struct variable.

  ```
  AccField of access * identifier
  ```

- Abbreviation assignments (+=, -=, *=, /=, %=) are expressed with the ShortAssign node. An alternative, simpler solution could have been to convert expression like `a += 1` to `a = a+1`. The problem in this case is that the access expression `a` is duplicated. If this expression contains side effects (think of `a[i++] += 1`), this could lead to unexpected behaviour and hard to detect bugs.

  ```
  ShortAssign of access * binop * expr
  ```

- The operators `++` and `--` are mapped to the four unary operators PreInc, PostInc, PreDec, PostDec, representing pre/post increment/decrement. The reasoning behind adding these as concrete operators is similar to the one used for abbreviation assignments. As we know, the two expressions `++i` and `i++` have a different meaning in c, so we need a way to distinct the different cases.

- The data type typ has three new constructors: TypF, TypS (identifier) and TypNull. TypF corresponds to the float type, TypS(s) represent a struct identified by the identifier s and TypNull is an internal type given to the Null expression (which is represented with the Null node). TypNull is used exclusively inside the compiler to handle NULL correctly, it does not correspond to any valid type in the language itself.

2.2.2 Parser rules definition

The parser itself is built using Menhir. The grammar is defined in parser.mly and it follows the specification provided in [2], with the necessary additions for language extensions. The syntax to define these new constructs is simply copied from c itself. When a token sequence is recognized, an annotated node is constructed using the auxiliary function **node**. The grammar has been slightly modified to avoid shift reduce conflicts. This required the inlining of rules to recognize binary and unary operators. To avoid the dangling else problem, the solution was giving a precedence to an if statement with an empty else. The resulting grammar is described as below:

```
Program ::= Topdecl* EOF

Topdecl ::= Vardecl ";" | Vardecl "=" Expr ";"
| Fundecl | Structdecl ";"

Vardecl ::= Typ Vardesc

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

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

Structdecl = "struct" ID "{" (Vardecl ";")* "}"

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

Typ ::= "int" | "char" | "void" | "bool" | "float" | "struct" ID

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

Expr ::= RExpr | LExpr

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


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

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

```
Shortop ::= "+=" | "-=" | "*=" | "/=" | "&="

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

Handling for In the ast there is no node that defines a for statement, which is instead converted to a while. Each expression is positioned in the correct order to create a semantically equivalent while loop. If an expression is not present, a default statement is created instead. Here i show an example of how the conversion works:

```
for(i = 0; i < n; i++)
{
print(a[i]);
}
```

```
i = 0;
//or empty block
while(i < n)
// or while(true)
{
print(a[i]);
i++;
//or empty block

}
```

2.2.3 Parser engine

The actual parsing logic is defined in parser_engine.ml. It uses the scanner and the parser to convert a source file into a valid ast. In this module, i decided to use Menhir incremental API to provide better signalling of syntax errors. This API uses declarative error reporting. In **errors.messages**, a different message is associated to each error state of the generated LR(1) automata. As an example, let's say that the compiler is parsing the following source code, where we forgot to type a ; after the declaration of a:

```
int a = 2
print(a+2)
```

The compiler will generate the following error message:

```
Syntax error:
Illegal right expression in assignment, did you forget a ; ?
```

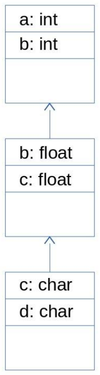
Although some errors might still be obscure, i felt that using the tools provided by the API allowed a significant improvement in the quality of generated syntax errors. The implementation was heavily inspired by the blog post referenced in [3], which provided a solid blueprint to develop my own version.

3 Symbol table

The symbol table was implemented as a hierarchical structure where, for each scope, we maintain an hash table to store the informations associated to an identifier and a reference to the lexically closest parent scope. When a new name is seen, the compiler searches for it in the current innermost scope. If it already exists, an error is raised, otherwise the information associated with this name is stored in the hash table.

```
int a , b;
...
int main(){
        float b,c,d;
        ...
        {
                char c, e;
                ...
        }
...
}
```

⇒



When we want to access a name (e.g `a = 1;`), the compiler scans the hierarchy, going from the innermost scope to the final global scope, and stops if the requested name is found in the hash table. To signal absence of the wanted name, a dummy top node is used as a reference from the global scope. If we reach this dummy node, we return None.

4 Semantic Analysis

The semantic checks are done in semant.ml. The strategy used is very simple: traverse the ast and recursively check the rules defined in [4]. The source code is analysed top to bottom and variables, functions and structs must be declared before being used. A type is given to each expression , raising an error when it is not possible. The following additions and changes were needed to accommodate for extensions:

- Unsized array are allowed as function parameters, so a unifying algorithm (`match_types`) is used. This is needed to pass a sized array to a function that expects an unsized one without violating any previous rule.
- Multi dimensional array are allowed, but in a controlled way. When a function is declared with an array parameter, it must have a complete type. This means that a declaration like `int a[][2][3]` is allowed but not `int a[2][][3]` or `int a[][][]`. This closely follows C rules, where the restriction is needed to generate machine code that correctly allocates memory.

- For similar reasons, a struct can only have recursively defined fields only in the form of pointers. A declaration like

  ```
  struct A
  {
          int a;
          struct A *b;
  };
  ```

  is allowed but not

  ```
  struct A
  {
          int a;
          struct A b;
  };
  ```

- String literals are considered array of characters, but this prevented them from being used when we immediately initialize a variable after declaration. To lift this restriction, the specific rule `string_var_initialization` was introduced. This rule could also be modified and reused for introducing array literals.
- The internal type TypNull is used to check that NULL is only assigned to pointers.
- Global variable initialization is allowed, but only with compile time constants. This means that we can use numerical and string literals, characters, booleans,NULL, and binary or unary expressions that have constant operators. Variable access, function calls, and address are forbidden

  ```
  float foo(){return 1.;}
  int a = 3; //OK
  char c = 'c'; //ok
  float b = foo(); //ERROR
  int main()
  {
  return 0;
  }
  ```

- A shorthand assignment such as `a += 1` is de-sugared to `a = a+1` only for this phase, to reuse already defined rules.

Variable scope, function and structs names are handled with the symbols structure

```
type var_info = position * typ
```

```
type fun_info = position * fun_decl

type struct_info = position * struct_decl

type symbols = {
        fun_symbols : fun_info Symbol_table.t;
        var_symbols : var_info Symbol_table.t;
        struct_symbols : struct_info Symbol_table.t;
}
```

This is carried throughout the ast traversal. When a name is requested or declared, the appropriate symbol table function is used. Since functions and structs are declared only once, a possible different approach could be to store them separately from variables and simply referenced when needed. While this is more efficient , i felt that grouping the tables together was a clearer, less error prone and more "functional" approach.

Runtime support Before the ast check, a preliminary step declares the function prototypes defined in the external rt_support.c module. For each function a "fake" fun_decl structure is created, with the correct type and parameters but with an empty block as function body. These prototypes are defined in the Util module. When a statement such as `print(2)` is read, the compiler will use the informations provided by this step to enforce its rules.

5 Backend

In codegen.ml, the semantically checked ast is finally converted to LLVM IR code. The ast is traversed once again, and for each different node we apply the necessary steps to convert it to machine code. The final bitcode file can then be linked with rt_support.c using the llvm infrastructure. To test correctness, i used clang, which allowed me to directly compile a llvm assembly source file and a c file together into a single executable.

Code generation symbol table Symbol tables are used again, but each name is associated with the required information to translate it into machine code. For variables and functions, we keep the llvalue, a value that represent their lower level implementation. For structs we keep two values: the converted named struct type and the list of field names. The latter one is required for field access since, when a struct is built in llvm, field names are lost and replaced only with their type.

```
type symbols = {
        fun_symbols : L.llvalue Symbol_table.t;
        var_symbols : L.llvalue Symbol_table.t;
        struct_symbols : (L.lltype * string list) Symbol_table.t;
}
```

5.1 Main implementation choices

This part of the compiler proved to be the most challenging to implement, mostly because of these 3 issues:

1. How to translate array parameters in functions
2. Detecting when to stop generating code (e.g code written after an always reached return statement)
3. Handling null

I will now explain the choices that i made to solve these problems.

5.1.1 Array parameters

Array parameters are all converted to pointers. This follows the c implementation (array decay) and it was done to support unsized arrays without any major issues. This choice forced me to handle some part of code generation in a trickier way. When translating expressions such as `a[i]`, the compiler checks beforehand if the lltype of the variable is a pointer or an array and build the correct gep instruction for each case. Another modification was needed to correctly translate function calls. Normal array variables are converted to the llvm array type. This is a different type from pointers inside llvm and, so it would generate an error if e tried to use them as arguments. The compiler, using the gep instruction, pass a pointer to the first element instead. This conversion is also done with string literal arguments.

5.1.2 Stopping code generation

When translating statements, we need to ignore code that we are sure is unreachable. Think for example of a function like

```
int foo(){
        return 1;
        int a;
}
```

The statement `int a;` is never executed. Not only is translating it useless, but it could cause some hard to detect issues inside llvm itself. This problem was solved by making the function **codegen_stmt** return a boolean that is set to false when that code generation should immediately stop.

5.1.3 Null pointer

Null is translated to an undefined integer value because, since null can be assigned or compared to a pointer of any type, we have no information on what type to assign. The actual type is inferred from the context. For example, in `char *c = NULL`, we can assign the char pointer type to null. While this allowed the flexibility provided by NULL, it has the side effects of having to check for null pointers whenever an expression is expected, making the implementation messier and more error prone. A possible solution could be to assign a type to each null occurrence during the static analysis phase.

5.2 Code generation

There are 5 main functions used for code generation:

- codegen_access

  Translates each variable, array index, and pointer access to the associated address in memory. This function applies the steps described in 5.1.1 when dealing with an index access.

- codegen_expr

  Translates each expression into the corresponding intermediate representation and loads values from the address returned by codegen_access.

- codegen_stmt

  Each statement is translated to the associated low level instructions. In particular, it set up the correct blocks, labels and jumps when translating control flow statements (if, while, do while).

- codegen_func

  Creates a llvm function that corresponds to the original declaration. It converts parameters, their types and the function body adding a default terminator (undefined value for non void function) if a return statement is not present.

- codegen_struct

  Creates a new named struct type associated with the defined microC struct. It declares its name and convert its fields to their low level representation.

Global variable initialization Expressions used to initialize global variables are treated in a slightly different way from the normal ones. Since we only allow compile time known constant values, instead of using normal instructions such as **add**, the compiler uses their constant counterparts. This allows the llvm backend to directly assign the final value without building the expression.

In figure 1, we can see an example of how a very simple program is translated into LLVM IR code.

```c
int func(int a){
        if(a < 0) return 1;
        else return 0;
}
int main(){
        int a = 3;
        print(func(a));
}
```

⇒

```llvm
; ModuleID = 'microc_module'
source_filename = "microc_module"

declare void @print(i32)
declare i32 @getint()
declare float @getfloat()
declare i8 @getchar()
declare void @printchar(i8)
declare float @printfloat(float)

define i32 @func(i32 %0) {
entry:
%1 = alloca i32
store i32 %0, i32* %1
%2 = load i32, i32* %1
%3 = icmp slt i32 %2, 0
br i1 %3, label %then, label %else

then:
ret i32 1

else:
ret i32 0

cont:
ret i32 undef
}

define i32 @main() {
entry:
%a = alloca i32
store i32 3, i32* %a
%0 = load i32, i32* %a
%1 = call i32 @func(i32 %0)
call void @print(i32 %1)
ret i32 undef
}
```

Figure 1: Code generation example

6 Conclusions and remarks

The project proved to be quite challenging and complex, but it was also very enjoyable. It made me realize how crucial a good compiler design is in generating fast and reliable code, while providing a chance to consider the work needed to translate high level concepts to machine code. Some aspects such as arrays, managing namespaces and implementing loops were more trickier to handle than i would expect, giving me a way to improve my programming and problem solving skills. Overall, i feel quite satisfied with the final result.

References

[1] `https://github.com/lillo/compiler-course-unipi/blob/main/microc/microc-project.md`.

[2] `https://github.com/lillo/compiler-course-unipi/blob/main/microc/microc-parsing/README.md`.

[3] `https://baturin.org/blog/declarative-parse-error-reporting-with-menhir/`.

[4] `https://github.com/lillo/compiler-course-unipi/blob/main/microc/microc-semantic-analysis/README.md`.