Microc compiler

Giulio Piva

Instructions for compiling and running tests To compile a microc program, you can run the script:

```
chmod +x test_program.sh
test_program.sh <microc_file>
```

To run a suite of tests, you can run the script:

```
chmod +x test_all.sh
test_all.sh <test_folder>
```

1 Frontend

The frontend of the compiler is responsible for parsing the input program and generating the AST. It has been implemented using the Menhir parser generator and the ocamllex lexer generator. ocammllex uses regular expressions to define the patterns for recognizing tokens in the source code. Menhir, on the other hand, processes the stream of tokens produced by the lexer and constructs an abstract syntax tree by means of code snippets associated with each grammar rule. These two components are then combined into a parser engine in the file **parsing.ml**, used to parse the input program.

1.1 Grammar

The following is the grammar of microc:

```
Program::= Topdecl* EOF

Topdecl::= Varlist ";" | Fundecl | Structdecl ";"

Varlist::= Typ Vardecl*

Vardecl::= Vardesc ("=" Expr)?

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

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

Structdecl::= "struct" ID "{" (decl";")* "}"

decl:: Typ Vardesc

Block::= "{" (Stmt | Varlist)* "}"

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
| "*" LExpr | "*" AExpr
| LExpr "[" Expr "]"
| LExpr "." ID

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

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

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

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

As specified before, each time a rule is matched, a node of the AST is created. See the file **parserm.mly** for more information.

1.2 Grammar modifications

In relation to the original grammar provided by the assignment, the following modifications have been implemented to extend the language:

- multiple variable declaration and initialization: multiple variables can be declared and initialized in a single statement like the following: **int a = 1, b = 2, c = 3;**. However, since function parameters and struct fields don't support initialization, two different rules have been created: varlist and decl. The first one is used for variable declarations, while the latter is used for declaring function parameters and struct fields.
- sizeof operator: sizeof is considered as operator and not as a function. As such, its name has been added to the list of reserved keywords.
- bitwise operators: <<, >>, &, | , ˆ have been added to the list of binary operators.
- multi-dimensional arrays.
- short assignment operators: +=, -=, *=, %=.
- struct declaration and access: structs can be declared and accessed using the dot operator. The struct declaration must end with a semicolon.
- do-while loops.
- string literals: implemented as arrays of characters.
- floating point numbers:

1.3 Grammar disambiguation

The raw form of this grammar is ambiguous. To resolve this, Menhir's precedence and associativity rules have been utilized for disambiguation:

- dangling else: give precedence to if statement with an empty else branch
- Binary operators: inline the BinOp rule
- Bitwise operators: give precedence to bitwise operators over logical operators

2 Symbol Table

The symbol table has been implemented as a list of hash tables to resemble a hierarchical scope structure. Therefore, the head of the list represents the local scope, whereas the tail represents the global scope. For convenience and efficiency, three different symbol tables have been created for storing information about variables, functions and structs. Those three tables are then composed into a single object which is carried throughout the sematic analysis and code generation phases. The type signature of a symbol list is the following:

```
type 'a t = (string, 'a) Hashtbl.t list
```

The type signature of the agglomerating object is the following:

```
type environment = {
  fun_symbols : 'a Symbol_table.t;
  var_symbols : 'b Symbol_table.t;
  struct_symbols : 'c Symbol_table.t;
}
```

The types will assumes different values during the semantic analysis and code generation phases, which will explained in the following sections.

3 Semantic Analysis

The aim of the semantic analysis of a microc program is to check the well-typedness of the program. This phase recursively traverses the AST and checks that the statements of the programs satisfy the rules of the type system. The relevant functions are the following:

- **fundecl_type_check**
- **structdecl_type_check**
- **var_type_check**
- **stmt_type_check**
- **stmtordec_type_check**

All of these leverage the following other functions to obtain the type of their arguments and expressions:

- **expr_type**
- **access_type**
- **unaryexp_type**
- **binaryexp_type**

3.1 Semantic analysis Symbol Table

The type signature of the symbol table for this phase is the following:

```
type var_info = (code_pos * fun_decl) Symbol_table.t;
type fun_info = (code_pos * typ) Symbol_table.t;
type struct_info = (code_pos * struct_decl) Symbol_table.t;

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

for each kind of symbol, different information are stored and used during the analysis:

- **var_info**: the type of the variable
- **fun_info**: The return and parameter types of the function
- **struct_info**: The fields of the struct along with their types

3.2 Additional rules

Type Unification In certain cases, types should not be directly compared for equality, but rather checked for compatibility. The helper function **compare_types** is utilized to implement the following type checking rules:

- Variable Initialization and Assignment: Pointer variables can be assigned the value NULL.
- Function Parameter Types: Unsized arrays are permissible as function parameters, allowing declarations in the form of a[] or a[][2][2].

Global variable initialization Global variables can be initialized only with compile-time defined constants. In microc, this means that they cannot be initialied with a function call.

Underflow and Overflow When assigning a value or initializing a variable, the value must be within the range of the type.

String initialization Strings are implemented as arrays of characters, but arrays cannot neither be initialized nor assigned in microc. Therefore, a special case has been added when checking the declaration of an array of characters to allow string declarations. The function which handles string initialization is **init_string**.

Deadcode detection Various types of dead code analyses exist. In this project, I focused on detecting code that follows a return statement. To facilitate this analysis, a boolean flag was incorporated into the return types of the **stmt_type_check** and **stmtordec_type_check** functions. This flag indicates whether the scan should proceed, specifically if a return statement has been encountered. If instructions are found after a return statement, an exception is raised.

Independent-order declarations Functions and structs can be declared in any sequence and utilized prior to their declaration. This necessitates an initial scan of the program to incorporate the signatures into the symbol table. The bodies of these functions and structs are then verified in a subsequent scan.

Runtime support functions During the initial scan, runtime support functions are incorporated into the symbol table. These runtime functions are defined in the **runtime.c** file. For each function, a placeholder AST node with an empty body is created and added to the symbol table. Most of the runtime functions are used to input and output data. Their implementation will be linked to the produced bitcode by Clang.

4 Code Generation

The code generation phase is responsible for generating the LLVM bitcode of the program. As before, the code generation phase recursively traverses the AST and generates the bitcode for each statement, mapping them to the corresponding LLVM operation. The relevant functions are the following:

- **codegen_fundecl**: maps a function declaration to a LLVM function. It generates the body of the function and adds a default terminator if a return statement is not present.
- **codegen_struct**: maps a struct declaration to a "named structure" in LLVM
- **codegen_stmt**: maps a statement to its corresponding LLVM operation. In particular if,while and do-while statements are mapped according to the following templates:
  if statement:

```
                           br <cond>, label %then, label %else
  if(cond) {               then:
    ...                      ; then body
  }                          br label %cont
  else {               →   else:
    ...                      ; else body
  }                          br label %cont
  <remaining code>         cont:
                             ; remaining code
```

while statement:

```
                           br label %test
                           test:
                             ; necessary code to evaluate cond
                             br <cond>, label %body, label %cont
  while(cond) {
    ...                    body:
  }                    →     ; while body
  <remaining code>           br label %test

                           cont:
                             ; remaining code
```

- **codegen_access**: maps each variable, pointer and array access to the associated address in memory.
- **codegen_expr**: maps each expression to the corresponding intermediate representation. It loads values returned by **codegen_access**.

Code generation Symbol Table

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

for var_symbols and fun_symbols, an llvalue is stored, which is their corresponding LLVM implementation. For struct_symbols, a tuple of an lltype and a list of strings. The latter element is required because, once a struct is declared, fields name will be forgotten.

Struct and Function generation

Similar to the semantic analysis, structs are generated in advance to permit their use before declaration. However, in this case, the structs are fully generated, not just their signatures. This is necessary because when accessing a field of a struct object, an address must exist in LLVM. Conversely, for function declarations, generating only the signature is sufficient to enable its invocation.

Array parameters

Array parameters are converted to pointers, mirroring the practice in the C language This mechanism allows the use of unsized arrays as parameters. However, this conversion necessitates additional checks when accessing an array in the code. Consider the following example:

```
int foo(int a[]) {
  int b[2];
  b[0] = a[0];
  ...
}
```

In the produced LLVM bitcode, **a** is of pointer type, while **b** is of array type. Therefore, array access must be handled differently when the code generator encounters a AccIndex expression. Function calls must also accommodate this conversion. Upon encountering a Call statement, the code generator will pass a pointer to the array's first element as the argument.

Global variable initialization

Since global variables can only be initialized with compile-time defined constants, operations on these variables are restricted to constants.

5 Conclusion and future work

All of the extensions of the language were implemented, except for the comma operators and separate compilation. I'd like to give a brief idea of how I would have implemented a basic separate compilation mechanism. First, I would parse the different microc modules and create an AST for each of them. Then, I would create a unique symbol table and scan the ASTs to add all the functions, structs and global variables declarations to it, so we can correctly type-check every single program even if the actual implementation of a function is not present. Finally, I would generate the LLVM bitcode for each module and link them together using the LLVM linker. Overall, I believe that this project was a great opportunity to learn about compilers and LLVM. Even if the language is quite simple, The difficulties encountered during the implementation were quite challenging and interesting to solve.