UNIVERSITÀ DI PISA

μComp language implementation

Gabriele Pappalardo
E-mail: g.pappalardo4@studenti.unipi.it
Department of Computer Science

February 2022

Abstract and Overview

The following paper will illustrate the μComp programming language, created for the ”*Languages, Compilers and Interpreters*” laboratory course. The document explains how the language is implemented.

The first section will introduce the language with its features and will show a little example of a program. The second section will show the language grammar, and it will discuss the design choices of the scanning and parsing phases. The third will describe how the semantic analysis checks are performed, according to the assignment specifications. The fourth and fifth sections will focus onto the code linking and code generation phase using the LLVM compiler infrastructure. Finally, the conclusion section will end the document with an overview of future developments and improvements.

1 Introduction to μComp

As the assignment cites: 'μComp *is a simple component-based imperative language*'. The main features of the language are the following:

- is statically type checked;
- programs are made of *components*, which are linked together to form a whole program;
- a component is stateful, there is only one instance of each component;
- component behaviour specifications are given in terms of *interfaces*, which can be provided or used by the components;
- interfaces specify a set of functions and global variables to be provided by the interface's provider (similar to the Java programming language);
- components are statically linked to each other via their interfaces.

Listing 1 shows a little program, that prints a greeting, using the μComp programming language. The interface `StringPrinter` defines the function `print_str` which takes an array of characters and a size integer. The component `SimplePrinter` provides the interface `StringPrinter` that defines `print_str` function. The `Main` component is the entry program point, since it provides the `App` interface. The `main` can invoke the `print_str` function because the component is using the `StringPrinter` interface. At the end, the `Main.StringPrinter` is linked with `SimplePrinter.StringPrinter`, so the call to `print_str` inside the `Main` component will be qualified with the `SimplePrinter` component. More examples can be found inside the `test/samples` directory of the project.

```
interface StringPrinter {
    def print_str(msg: char[], size: int): void;
}

component SimplePrinter provides StringPrinter {
    def print_str(msg: char[], size: int): void {
        var i: int;
        for (i = 0; i < size; i++) putc(msg[i]);
        return;
    }
}

component Main provides App uses StringPrinter {

    def main(): int {

        var msg: char[6];
        msg[0] = 'H';
        msg[1] = 'e';
        msg[2] = 'l';
        msg[3] = 'l';
        msg[4] = 'o';
        msg[5] = '!';

        print_str(msg, 6);

        return 0;
    }
}

connect Main.StringPrinter ← SimplePrinter.StringPrinter;
```

Listing 1: 'A μComp program that greets an user.'

1.1 Language Extensions

The assignment project requests the implementation of at least two language extensions. I chose the following constructs:

- abbreviation for assignment operators $(+=, -=, *=, /=, \%=)$;
- pre/post increment/decrement operators $(++$ and $--)$;
- variable declaration with initialization;
- floating point arithmetic.

Next sections will explain how these extensions have been implemented inside the compiler. For each new extension, a unit test file has been added to check the correct functioning.

2 Lexing and Parsing phases

This section defines μComp grammar and how the lexing and parsing phase are designed.

2.1 Grammar

Here is the μComp grammar expressed in EBNF notation. The words enclosed by the angle brackets are non-terminal symbol; uppercase words are tokens containing a value. The grammar contains the extensions chosen for the fourth assignment.

$\langle \mathit{CompilationUnit} \rangle ::= \langle \mathit{TopDecl} \rangle^*$ EOF

```
⟨TopDecl⟩ ::= ‘interface’ ID ‘{’ ⟨IMemberDecl⟩+ ‘}’
  |  ‘component’ ID ⟨ProvideClause⟩? ⟨UseClause⟩? ‘{’ ⟨CMemberDecl⟩+ ’}‘ | `connect’
     ⟨Link⟩ ‘;’? | ‘connect’ ‘{’ (⟨Link⟩ ‘;’)* ‘}’

⟨Link⟩ ::= ID ‘.’ ID ‘←’ ID ‘.’ ID

⟨IMemberDecl⟩ ::= ‘var’ ⟨VarSign⟩ ‘;’ | ⟨FunProto⟩ ‘;’

⟨ProvideClause⟩ ::= ‘provides’ (ID ‘,’)* ID

⟨UseClause⟩ ::= ‘uses’ (ID ,)* ID

⟨VarSign⟩ ::= ID ‘:’ ⟨Type⟩

⟨FunProto⟩ ::= ‘def’ ID ‘(’((⟨VarSign⟩ ‘,’)* ⟨VarSign⟩)? ‘)’ (‘:’ ⟨BasicType⟩)?

⟨CMemberDecl⟩ ::= ‘var’ ⟨VarSign⟩ ‘;’ | ‘var’ ⟨VarSign⟩ ‘=’ ⟨Expr⟩ ‘;’ | ⟨FunDecl⟩

⟨Fundecl⟩ ::= ⟨FunProto⟩ ⟨Block⟩

⟨Block⟩ ::= ‘{’ (⟨Stmt⟩ | ‘var’ ⟨VarSign⟩ ‘;’ | ‘var’ ⟨VarSign⟩ ‘=’ ⟨Expr⟩ ‘;’)* ‘}’

⟨Type⟩ ::= ⟨BasicType⟩ | ⟨Type⟩ ‘[’ ‘]’ | ⟨Type⟩ ‘[’ INT ‘]’ | ‘&’ ⟨BasicType⟩

⟨BasicType⟩ ::= ‘int’ | ‘char’ | ‘void’ | ‘bool’ | ‘float’

⟨Stmt⟩ ::= ‘return’ ⟨Expr⟩? ‘;’
  |  ⟨Expr⟩? ‘;’
  |  ⟨Block⟩
  |  ‘while’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Stmt⟩
  |  ‘if’ ‘(’ ⟨Expr ‘)’ <Stmt⟩ ‘else’ ⟨Stmt⟩
  |  ‘if’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Stmt⟩
  |  ‘for’ ‘(’ ⟨Expr⟩? ‘;’ ⟨Expr⟩? ‘;’ ⟨Expr⟩? ‘)’ ⟨Stmt⟩

⟨Expr⟩ ::= INT | CHAR | BOOL | FLOAT
  |  ‘(’ ⟨Expr⟩ ‘)’
  |  ‘&’ ⟨LValue⟩
  |  ⟨LValue⟩ ‘=’ ⟨Expr⟩
  |  ‘!’ ⟨Expr⟩
  |  ID ‘(’ ((⟨Expr⟩ ‘,’)* ⟨Expr⟩)? ‘)’
  |  ⟨LValue⟩
  |  ‘-’ ⟨Expr⟩
  |  ⟨Expr⟩ ⟨BinOp⟩ ⟨Expr⟩
  |  ⟨LValue⟩ ‘++’ | ‘++’ ⟨LValue⟩ | ⟨LValue⟩ ‘--’ | ‘--’ ⟨LValue⟩
  |  ⟨LValue⟩ ‘+=’ ⟨Expr⟩ | ⟨LValue⟩ ‘-=’ ⟨Expr⟩ | ⟨LValue⟩ ‘*=’ ⟨Expr⟩ | ⟨LValue⟩ ‘/=’ ⟨Expr⟩
     | ⟨LValue⟩ ‘%=’ ⟨Expr⟩

⟨LValue⟩ ::= ID | ID ‘[’ Expr ‘]’

⟨BinOp⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘%’ | ‘/’ | ‘&&’ | ‘||’ | ‘<’ | ‘>’ | ‘⩽’ | ‘⩾’ | ‘==’ | ‘≠’
```

The token ID represents strings used for the identifiers. Tokens INT, FLOAT, CHAR and BOOL represent integers, floating point number, characters and boolean literals.

2.2 OCamllex

Ocamllex[1] is a tool that produces a lexical analyser given a file containing regular expressions. This subsection describes how regular expressions are made and how some additional checks, like 32-bit numbers one check, are implemented. The lexer specification can be found inside scanner.mll file.

[1] For further details: https://ocaml.org/manual/lexyacc.html

The *header* section contains utility functions to perform the requested checks by the assignment. Additionally, to the check functions, a dictionary of reserved words (keywords) can be found. Each table key is associated with a token, in which there are nineteen keywords: *true*, *false*, *var*, *def*, *uses*, *int*, *float*, *char*, *bool*, *void*, *for*, *while*, *if*, *else*, *return*, *connect*, *provides*, *uses*, *interface* and *component*.

There are four check functions:

1. `check_identifier`: if the given string does not exceed the length of sixty-four characters generates an identifier token;
2. `check_num_float_32`: checks if a given floating number is a 32-bit one, and it returns a token representing the number;
3. `check_num_int_32`: is similar to the function above, except that is for integer numbers. The function checks if the given number exceed the maximum integer, which is `0×7FFFFFFF`;
4. `check_character`: ensures during the character parsing that there is one and only one character.

All the check functions can raise a `Lexing_error` if the preconditions are not satisfied. Each function accept `lexbuf` buffer as parameter to address the wrong token position to the user.

The *rule* section contains all the regular expression definitions for identifiers, numbers (both integers and floats), hex integer numbers, and blank characters. The rule `next_token` is the scanner main entry, it recognizes all the regular expressions listed before, plus additional operators. The rule also checks for punctuation symbols (parenthesis, commas, brackets, and so on …).

Furthermore, this section, contains two rules for inline and multiline comments, respectively denominated: `single_line_comment` and `multi_line_comment`.

The single characters have a special rule for their handling. The `character` rule takes a character as input, and it checks if it is an escape character. If the character is an escape one, then its hexadecimal representation is tokenized (note: the standard ASCII representation cannot be used for some escape characters since they are not recognized by the OCaml compiler).

The floating point numbers are seen as a list of digits followed by a dot and another list of digits. Listing 2 shows the rule for this numbers' category.

```
let float_number = digit+'.'digit+ (* 3.14, 2.17, 1.1618, ... *)
```

Listing 2: ”Regular expression for floating point numbers.”

2.3 Menhir

The parsing phase uses **Menhir** parser generator[2]. The parser specification is inside `parser.mly` file.

The *header* section defines only two helper functions: the infix operator `(@>)` that creates an `('a, 'b) annotated_node` and `build_compilation_unit` that returns a new compilation unit given interfaces, components and connections.

The *rule* section contains all the productions defined by the language grammar, with the following precedences and associations listed in Listing 3.

```
%nonassoc K_IF
%nonassoc K_ELSE
```

[2] **Menhir** is an LR(1) parser generator for OCaml. http://gallium.inria.fr/ fpottier/menhir/

```
%right O_ASSIGN O_PLUS_ASSIGN O_MINUS_ASSIGN O_TIMES_ASSIGN O_DIV_ASSIGN
    O_MOD_ASSIGN

%left L_OR_OR
%left L_AND_AND
%left C_EQ_EQ C_NOT_EQ

%nonassoc C_LT C_GT C_GT_EQ C_LT_EQ

%left M_PLUS M_MINUS
%left M_TIMES M_DIV M_MOD

%right M_MINUS_MINUS

%nonassoc U_MINUS
%nonassoc U_NOT
```

Listing 3: ”Precedences and associactions.”

The starting symbol is the `compilation_unit` defined in Listings 2.3. The `decls` variable is a list of `Ast.definition` which is a new variant type defined to handle the creation of a compilation unit in a single rule. Each `top_decl` can be an interface, component or connection block definition. The introduction of the new type allows writing the `compilation_unit` in a short and simple form.

```
compilation_unit:
  | decls = top_decl* EOF { build_compilation_unit decls }

top_decl:
  | interface   { Ast.InterfaceDef($1) }
  | component   { Ast.ComponentDef($1) }
  | connections { Ast.ConnectionDef($1) }
```

The precedence rules showed in Listing 3 guarantee that there is no dangling else problem [3].

2.4 Edits on the Abstract Syntax Tree

Not all the new extensions have a strong impact on the existing Abstract Syntax Tree already given. For example, the first language extension listed before in Section 1.1, the abbreviation for assignment operators, does not introduce any change, because during the parsing phase when the token is matched it will be treated just like an assignment. The Listing 4 shows an example rule for this case.

```
  ...
  | lv = l_value O_PLUS_ASSIGN e = expr {
    let lv_exp = Ast.LV(lv) @> $loc in
    let final_exp = (Ast.BinaryOp(Ast.Add, lv_exp, e)) @> $loc in
    (Ast.Assign(lv, final_exp)) @> $loc
  }
  ...
```

Listing 4: ”How the short operator assignment rule is implemented for plus-equal. The rule does not use any new AST construct but it reuses the pre-exisisting AST’s Assign with some pre-coding.”

To implement pre/post increment/decrement operators, the `ast.ml` defines new variant types, respectively:

- `type dop = MinMin | PlusPlus and dop_prec = Pre | Post`, where the `dop_prec` indicates if the operator stands before the increment/decrement operator;

[3] Dangling else: https://en.wikipedia.org/wiki/Dangling_else

- type 'a expr = ... | DoubleOp of dop * dop_prec * 'a lvalue, this new expression has different semantic according to the dop_prec field, which is similar to the C-like programming languages. The ++ and −− can only be applied to numerical type variables, such as integer and floating point numbers.

The parser file contains two new tokens for these two operators: M_PLUS_PLUS and M_MINUS_MINUS. The latter operator raised a problem inside the parser. The expression --42 would be not recognized as valid by the parser, since the lexer would tokenize the token M_MINUS_MINUS and not two M_MINUS, this led to the rule listed in Listings 5 to not be matched. To solve this issue, the rule has been changed like the one shown in Listings 6.

```
| M_MINUS_MINUS l_value {
    (Ast.DoubleOp(Ast.MinusMinus, Ast.Pre, $2)) @> $loc
}
```

Listing 5: ”Previous rule raising problems during the parse phase.”

```
| M_MINUS_MINUS e = expr {
    match (e.Ast.node) with
    | Ast.LV(lv) ->
        (Ast.DoubleOp(Ast.MinMin, Ast.Pre, lv)) @> $loc
    | _ ->
        let u_exp = Ast.UnaryOp(Ast.Neg, e) @> $loc in
        (Ast.UnaryOp(Ast.Neg, u_exp)) @> $loc
}
```

Listing 6: ”New rule wich solves the problem described with preceeding −− operator.”

The implementation of ”*variable declaration with initialization*” required the change of Ast.VarDecl and Ast.LocalDecl, which now accepts an optional expression used as initial value for the variable. The parser rules for this implementation are pretty straightforward as shown in Listings 7. Array initialization is not supported to respect the assignment specification which does not allow array assignments. Furthermore, interface variables cannot be initialized, this is a design choice since an interface guarantees that a component will provide determined members.

```
c_member_decl:
| K_VAR vs = var_sign SEMICOLON { (Ast.VarDecl(vs, None)) @> $loc }
| K_VAR vs = var_sign O_ASSIGN e = expr SEMICOLON { (Ast.VarDecl(vs, Some e))
     @> $loc }
...

block_element:
| K_VAR vs = var_sign SEMICOLON { (Ast.LocalDecl(vs, None)) @> $loc }
| K_VAR vs = var_sign O_ASSIGN e = expr SEMICOLON { (Ast.LocalDecl(vs, Some e
    )) @> $loc }
...
```

Listing 7: ”Rules for variable initialization.”

For the last language extension, the floating point arithmetic, the ast.ml file contains new definitions for a float type and float literals. All the previous maths operators can be used with floating point numbers.

3 Semantic Analysis phase

The section describes how the Symbol Table is implemented and how the Semantic Analysis phase is performed by the compiler.

3.1 Symbol Table

This subsection describes how the symbol table interface is implemented inside the project. The implementation can be found inside symbol_table.ml file.

μComp implements *lexical* (or *stating*) *scoping* as visibility rule for names. The language has a specific rule set:

- interface and components define their own scope containing variables and functions;
- the global scope contains only interface, component and connection declarations which are mutually recursive;
- a function defines its own scope;
- the declarations inside the component are mutually recursive as well;
- a function could contain nested blocks, in which each of them defines their own scope that can hide previous variable definitions (as shown in Figure 1);
- control flow statements with a block as body define a new block scope.

The `Symbol_table` module defines a new type `'a t` for the symbol table. Listing 8 shows how the type is implemented. The implementation emulates a stack of blocks, in which the top block could shadow the previous defined names.

```
type 'a t =
  | EmptySymbolTable
  | SymbolTable of {
        parent: 'a t;
        table: (Ast.identifier, 'a) Hashtbl.t
    }
```

Listing 8: ”Symbol Table type definition.”

The `empty_table` function points to an `EmptySymbolTable` which is a dummy placeholder without any blocks contained in it. To add a new block, the user can use `begin_block`, which pushes a new `SymbolTable` value with the parent pointing to the previous block, and it creates a new hash table with `Ast.identifier` as key and a generic `'a` as value. To pop a block, the function `end_block` is invoked, which returns the parent of the current block that the symbol table is pointing to. If `end_block` takes as parameter a `EmptySymbolTable` an OCaml exception is thrown signalling that the operation cannot be performed.

The `add_entry` function adds, as the name suggests, a new entry inside the symbol table if and only if the current block does not contain a value with the same key. If there already exists a value with a given identifier, the exception `DuplicateEntry` is thrown to the caller. The `lookup` function checks recursively if an identifier is contained inside the symbol table. If the current block does not contain the searched key, then the function will call itself to the parent block. The recursion terminates if the identifier is found or the `EmptySymbolTable` has been reached, raising a `MissingEntry` exception as consequence.

3.2 Semantic checks

The semantic analysis is performed using the symbol table previously described. At the beginning of the file `semantic_analysis.ml` there are two new types definitions for the use of the symbol table: `attr` and `'a sym`. The Listing 9 shows how these two types are defined.
The former type is used as metadata container for the symbols, since there are common fields (like the identifier, location and the type). The second type is a variant one, inspired by the Symbol Table chapter inside ‘*Engineering a Compiler*’ book [TC07], structured in the following way:

- `SComponent` is the symbol representing a Component, this construct has its attributes, a component symbol table which represents the component’s scope containing field and function symbols, and two symbol tables for the used and provided interface symbols;
- `SInterface` represents an interface symbol, and it only has its attributes and an interface symbol table which contains fields and function symbols;

```
def foo(a: int, b: int) {

    var i: int = 0;
    {
        var a: char = 'a';
        {
            var b: bool = true;
        }
        putc(a);
    }
    {
        var b: float = 3.14;
        print_f(b);
    }

    print(a + b);
}
```

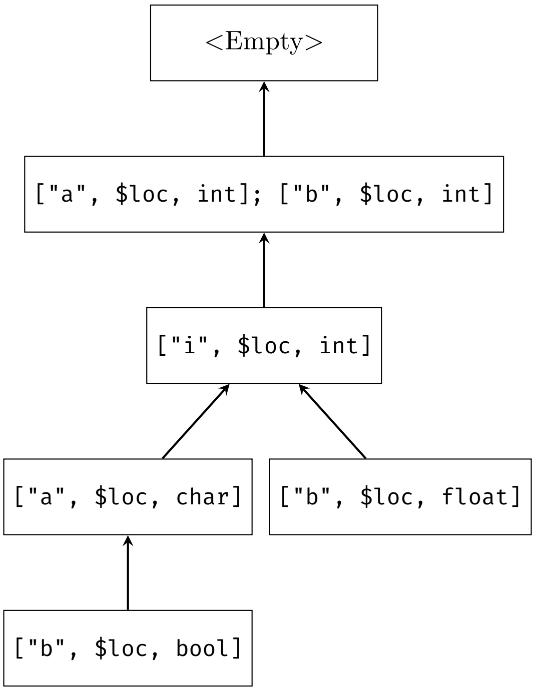


Figure 1: The figure shows how a function scope is represented using the symbol table. Each node represents a level of scoping. A triplet contains the symbol identifier, the location in the code and the type.

- `SFunction` represents a function symbol, and it has a function symbol table for its parameter and local variables symbols;
- `SVar` is the last symbol representing a variable, and it has only the attributes described before.

```
type attr = {id: Ast.identifier; loc: Location.code_pos; typ: Ast.typ}
type 'a sym =
  | SComponent of {cattr: 'a; csym_tbl: 'a sym_table; cprov: 'a sym_table;
    cuses: 'a sym_table}
  | SInterface of {iattr: 'a; isym_tbl: 'a sym_table}
  | SFunction  of {fattr: 'a; fsym_tbl: 'a sym_table}
  | SVar       of {vattr: 'a}
and 'a sym_table = ('a sym) Symbol_table.t
```

Listing 9: ”The symbol type used by the symbol table during the semantic analysis phase.”

The semantic checks happen in two passes, which will be described in the following subsections.

3.2.1 First Pass

The first pass creates a global symbol table which will be used by the second pass during the type checking evaluations. The global symbol table will contain only interface and component symbols. Before the generation of the table, several checks are performed as requested by the assignment specification.

First, the `App` and `Prelude` symbol interfaces and dummy interface declaration node will be created, which will contains all the functions defined in the respective interface signature (see `mcomp_stdlib.ml`). After the loading of standard interfaces, user defined ones will be placed in the global symbol table. Then, the component symbols will be added to the global table ensuring that:

1. the component does not provide the `Prelude` interface, since it is implicitly provided by the runtime support;

2. the *uses* and *provides* component lists are disjointed;
3. a component or an interface with the same name must not exist already;
4. uses and provides symbol table will be created containing only existing interface symbols. This means that if a non-existing interface is referenced or there is a duplicated interface name in both uses and provides list, a semantic exception will be thrown;
5. the component's symbol table will contain symbols for its fields and functions;
6. the component members are not defined twice, and they are type checked according to the specification.

After the interfaces and component symbols are created and inserted into the global symbol table, these checks are strictly verified:

1. each component defines the members declared in the provided interfaces;
2. there are no interfaces used by a component which led to ambiguous names;
3. no component uses the `App` interface;
4. there exists just only one component that provides the `App` interface.

Finally, the global symbol table is generated and verified, thus the second pass can begin.

3.2.2 Second Pass

After the first pass checks, the second pass performs the type checking on the *Abstract Syntax Tree* (from this point abbreviated as AST) made by the parser, producing a new typed AST.

The second pass uses the global symbol table to perform type checking. Each AST's node is re-created but with a type annotation, instead of a location structure.

For each component member, several checks are performed. The fields with inline initialization are type-checked as well. For the functions, a `fun_env` value is created, which holds variables needed by the check (the component symbol, the function symbol table and the function node).

The following list notes important aspects that are checked on the statements:

- the `if`, `while` and `for` guard expression will evaluate to a boolean type;
- an empty `for` guard expression will be interpreted as a constant true boolean;
- the expressions returned by a return statement matches the function return type;
- the `Ast.Block` statement creates a new block inside the current function symbol table;
- any variable declaration will be inserted inside the current function symbol table, and if the current top block contains already a variable with a specific name, a semantic exception will be thrown;

The `Ast.Expr` statements are analysed by the `type_check_expr` function. Each expression is examined very carefully and, as for the statements, I will note important aspects of this evaluation:

- *assignment expressions* ensure that the assignment of an expression to a variable is *safe* (see Section 3.2.3);
- unary operators are applied to correct expressions (`Ast.Not` operator for boolean expression and `Ast.Neg` for numerical expressions);
- double operator ($++$, $--$) applies to numerical variables;
- binary operators are well-formed according to the specification (Section 3.2.3 will dig into this aspect);

- the `Ast.Address` will be typed as a type reference to the given lvalue.

The "*call to a function*" expression guarantees that the program is not invoking a variable, but a function. From the component symbol table, the function symbol is retrieved in order to perform checks for the function arguments (type of the actual types that match the formal types and length of the actual parameter list). Whether the function symbol is not found inside the component symbol table, the function name lookup will shift to component uses symbol table, and for each used interface by the component, the name will be searched. If the lookup succeeds, then the call expression will be qualified with the interface name that declares the function (this will be again qualified to a component name providing the function during the "*Linking phase*" described in Section 4), otherwise a semantic exception is thrown informing that the program is trying to invoke an undefined function.

Once the second pass ends, a typed AST will be passed to the "*Linking phase*" to qualify unqualified interface names to the respective components.

3.2.3 Type checking

As said in the introduction, μComp is a statically type checked language. The language type system is **sound** in respect to the types and operation it defines.

The language defines the following types: `int`, `char`, `float`, `bool`, reference and array types for those. The type system ensures that an array or a reference to a type cannot be returned by a function. Multidimensional arrays are not allowed by the language.

The function `type_check_assign` checks if, given an assignment, the types involving the lvalue and the expression are valid. This check uses OCaml pattern matching combined with `when` statements, and it guarantees that the specification is not violated (see Listing 10).

```
let type_check_assign typ1 typ2 =
  match (typ1, typ2) with
  (* Case: T1 <- T2 <== T1=T2 && Scalar(T1) && Scalar(T2) *)
  | (t1, t2) when (Ast.equal_typ t1 t2) && (Ast.is_scalar_type t1) && (Ast.
    is_scalar_type t2) -> Result.ok true
  (* Case: T1 <- &T2 <== Scalar(T1) && T1=T2 && !Ref(T2) *)
  | (t1, Ast.TRef(t2)) when (Ast.equal_typ t1 t2) && (Ast.is_scalar_type t1)
    && not(Ast.is_ref t2) -> Result.ok true
  (* Case: &T1 <- &T2 <== T1=T2 && Scalar(T1) *)
  | (Ast.TRef(t1), Ast.TRef(t2)) when (Ast.equal_typ t1 t2) && (Ast.
    is_scalar_type t1) -> Result.ok true
  (* Case: &T1 <- T2 <== T1=T2 && Scalar(T1) *)
  | (Ast.TRef(t1), t2) when (Ast.equal_typ t1 t2) && (Ast.is_scalar_type t1)
    -> Result.ok true
  (* Error Cases *)
  | (Ast.TArray(_), Ast.TArray(_)) -> Result.error "Arrays cannot be assigned
    !"
  | (_, Ast.TVoid) -> Result.error "Cannot assign the void type to a variable
    !"
  | (t1, t2) -> Result.error (Printf.sprintf "Assignment not allowed. You
    cannot assign a %s to a %s variable!" (Ast.show_type t2) (Ast.show_type t1
    ))
```

Listing 10: "Type check on assignment expressions"

Binary expressions are type checked as well, so the cases showed in the Table 1 are all valid. Mathematical operations (i.e. sum, comparisons) between integer and floating point numbers are not allowed, but the run-time support (described in Section 5.2) allows converting a float to integer or vice versa.

First Operand Type	Second Operand Type
T1	T1
T1	&T1
&T1	T1
&T1	&T1

Table 1: Table showing allowed type for binary expressions.

4 Linking phase

The *linking phase* has two main goals, tested in the following order: (1) verifies that all the connection statements inside the program are valid and (2) it qualifies all the external names with the component specified in the connections. This phase is implemented inside the file `linker.ml`.

The function `check_connections`, as the name suggests, ensure that given a connection `C1.I1 <- C2.I2`:

- `C1` and `C2` refer to different components;
- `I1` is an interface *used* by the component `C1`;
- `I2` is an interface *provided* by the component `C2`;
- `I1` and `I2` refer to the same interface name;
- there are no overrides with previous established connections.

Moreover, the linker checks that for each component, there exist a connection for all the interface it uses. All the checks use the symbol table defined in Section 3.1.

After the verification phase, the function `qualify_components` qualifies the external interface names with the used component as stated by the connection statements. The qualification uses a symbol table constructed during the previous phase, and when an external name has to be qualified, it performs a lookup operation on the table to find the right component implementing the function. Figure 2 shows how an external name (i.e. the `sort` function) is qualified.

5 Code generation phase

This section explains how a μComp program generates LLVM bitcode.

5.1 LLVM

The **LLVM** Project is a collection of modular and reusable compiler and toolchain technologies.[LA04] The assignment requests to use LLVM in order to produce *LLVM bitcode* (which is a low-level *Intermediate Representation* used by the infrastructure), using the OCaml bindings that provide the needed LLVM API to OCaml programs [4]. The next subsection will describe in details how the API have been used to generate bitcode from the qualified typed AST.

5.1.1 Compilation Strategy

μComp programs are made of several components and interfaces, the latter construct is not needed for this compiler phase, thus we will focus on how to generate bitcode for the components. This phase performs a post-order tree walk on the AST to emit the bitcode.

[4] For further details, see: https://opam.ocaml.org/packages/llvm/

```
interface Sorter {
    def sort(a: int[], size: int): void;
}

component MergeSort provides Sorter {
    def sort(a: int[], size: int): void {
        // Implement merge sort…
    }
}

component Entry provides App uses Sorter {
    def main(): int {
        var arr: int[16];
        // …
        // According to the connection,
        // MergeSort.sort will be invoked.
        sort(arr, 16);
        return 0;
    }
}

connect Entry.Sorter ← MergeSort.Sorter;
```

```
Ast.Call(
    Some "Sorter", "sort", [ … ]
) → Ast.Call(
    Some "MergeSort", "sort", [ … ]
)
```

Figure 2: A simple program used to show the linking phase.

The compilation strategy for components is described as following. First, we create a global LLVM module from the given LLVM context. Second, inside the global module, a builder for *global constructor function* will be generated (details will be described in subsection 5.1.1). Then, from the prelude interface the functions contained in the signature will be declared as external (since these functions are defined inside the run-time support). After that, for each component member, declare it inside a global LLVM module assigning a mangled name. The name mangling process is computed by the `Codegen.name_mangling` function, which takes the component and a member name to produce a unique string formatted in the following way: `__(lowercase component name)_(lowercase member name)`. During the declaration process, if a component provides the `App` interface it means that there is the main function defined in it, so the name mangling procedure is not invoked because that function will be the entry point of our program. All the members declarations will be put inside a global symbol table containing `llvalue` (LLVM values) that will be used later in the process. Finally, the code generation phase defines each component contained by the AST.

For the function definitions, the code generator defines the type for a function, and it creates an entry block where there are contained parameter variables allocation if any. During the generation, a new symbol table containing `llvalue` for the function is created, and parameters name will be put on it. The arguments are allocated and stored inside the stack [5]. Then, the generation for the function body begins using the created builder for it. After the bitcode generation for the function, some helper utilities are executed onto the generated code (these utilities are better explained in the *Additional Notes* subsection).

Component fields are treated like global variables. The inline initialization of these fields is explained in the ”*How global variable initialization is handled*”.

Additional Notes

During the code generation phase, there are several ‘issues’ to be addressed. The next subsections will explain design choices taken to handle these situations.

[5] The parameter type T[], array reference, is mapped into a LLVM pointer type.

Merge of multiple return instructions

Once a μComp function is compiled into LLVM bitcode, the function `unify_blocks` will be called in order to remove eventually multiple return instructions. This function is invoked only for μComp functions returning values different from void, since the LLVM bitcode is in *Single Static Assignment* form, there can only be one return instruction. [6] To address this requirement, the unify function iterates over the basic block to create a list of blocks having return statements. If the created list has length ≤ 1 then no unify operation is performed, otherwise a new basic block called ”`ret.merge`”, containing a φ node whose value will be returned, will be created at the end of the function. For each basic block inside the previous list, the return value is taken, and it will be added to the incoming values of the φ node. Furthermore, the operation removes the extra return statements inside the basic blocks and replaces it with an unconditional jump to the ”`ret.merge`” basic block.

How dead code after the return function is handled

The μComp semantic analysis phase accept functions with multiple return statements at the same scope block level. But, once the return instruction is reached, the code after (called ”*dead code*”) will never be executed. To remove dead code, the function `remove_useless_returns` is called when the μComp function is compiled into LLVM bitcode. The function will iterate the instructions contained in each basic block. If a return statement is encountered during the instruction traversal, a counter will be increased. The next instructions after the return will be added to a removal list. Once the iteration terminate, the extra instructions will be deleted from the basic blocks.

How global variable initialization is handled

One of the extra addition to μComp language was the possibility to declare and initialize variables in one line. Starting from local variables there were not any problems for the implementation, but for the global ones a problem raised up. Since LLVM global variables are initialized with a scalar value, if the μComp global variable initialization contains a *non-trivial expression*[7], then LLVM cannot generate an initial value for that variable.

Thus, I had two ways to handle this issue:

1. *forbid* the initialization of a global variable with non-trivial expressions during the semantic analysis phase; or

2. understand how to generate LLVM bitcode that can solve the problem.

In the end, I choose the second one. In fact, reading the LLVM documentation and seeing how the Clang compiler handles this problem[8], I decided to use the **global constructors** functions, which are functions injected by the compiler before the execution of the *main* function [9]. LLVM APIs define an intrinsic global variable called `llvm.global_ctors` which is a global constant array of pointers to functions. All the functions defined inside the array will be invoked before the execution of the main function. The code generation phase adds the `@_MUCOMP__global_ctors` function to this array, that is used to invoke some helper functions. For this project, only the `@_MUCOMP__global_var_initializer` is invoked, and it is used to initialize global variables so that we can write non-trivial LLVM bitcode.

Listings 11 and 12 show how two global variables will be initialized.

```
component EntryPoint provides App {

  var foo: int = 30;
  var bar: int = 12 + foo;

```

[6] According to LLVM documentation, function returning void as value are not constrained to this requirement.

[7] Here non-trivial expressions means all the one which not contain use of variables or calls to function.

[8] To inspect the behaviour of Clang I used ”**Compiler Explorer**” (https://godbolt.org/), a tool suggested during lectures.

[9] For further details see: https://wiki.osdev.org/Calling_Global_Constructors

```
  def main() : int {
    print(bar);
    return 0;
  }

}
```

Listing 11: ”A program with non-trivial initialization global variables statements.”

```

@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, ↩
    void ()*, i8* } { i32 65535, void ()* @_MUCOMP__global_ctors, i8* null ↩
    }]
@__entrypoint_foo = global i32 0
@__entrypoint_bar = global i32 0

define internal void @_MUCOMP__global_ctors() {
entry:
  call void @_MUCOMP__global_var_initializer()
  ret void
}

define void @_MUCOMP__global_var_initializer() {
entry:
  store i32 30, i32* @__entrypoint_foo, align 4
  %0 = load i32, i32* @__entrypoint_foo, align 4
  %temp.add = add i32 12, %0
  store i32 %temp.add, i32* @__entrypoint_bar, align 4
  ret void
}
```

Listing 12: ”LLVM bitcode to initialize global variables.”

Alloca Instructions

The `alloca` instruction generation is performed by `aux_build_alloca` function, which takes an identifier, a μComp type and the `fun_env`. All the variable allocations are hoisted on top of the function entry. The reason of this operation is that if we allocate a variable (scalar or arrays) in a loop, there will be a number of allocations, at most equal to the number of iterations, of that variable onto the stack, resulting, in the best case scenario, to a stack overflow (on my machine a *segmentation fault* was raised up). Using the hoisting approach, the program semantic is not altered at all, and each allocation is placed next to the previous one. Thus, we can write code like the one shown in Listing 13, which will be compiled into Listing 5.1.1.

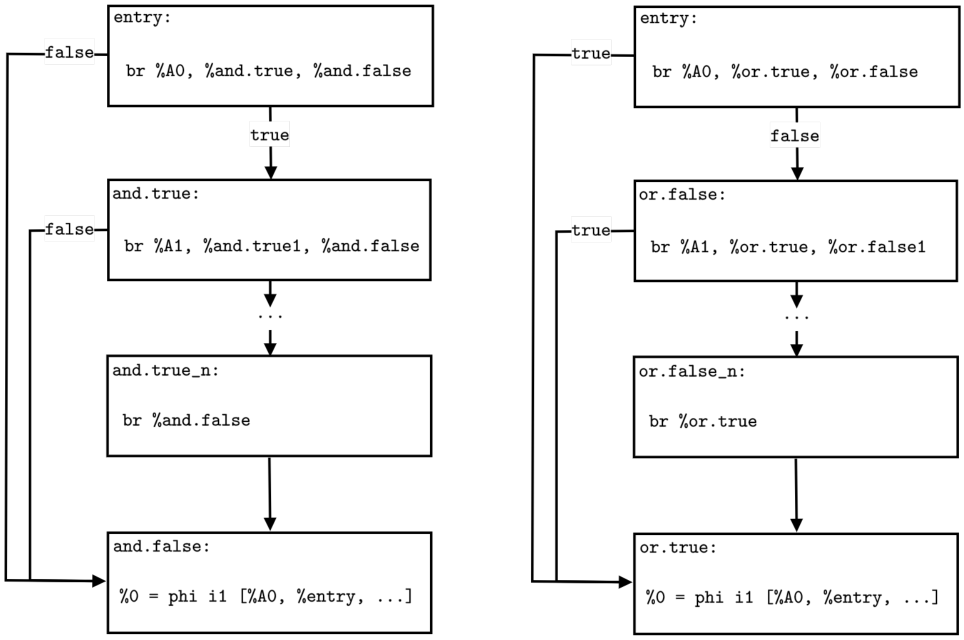
```
component EntryPoint provides App {
  def main() : int {
    var i: int;
    for (i = 0; i < 0x7FFFFFFF; i++) {
      var arr: int[16];
      print(i);
    }
    return 0;
  }
}
```

Listing 13: ”Program that loops to the maximum integer.”

```
define i32 @main() {
entry:
  %i = alloca i32, align 4
  %arr = alloca [16 x i32], i32 16, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                       ; preds = %for.loop, %entry
  %0 = load i32, i32* %i, align 4
  %temp.less = icmp slt i32 %0, 2147483647
  br i1 %temp.less, label %for.loop, label %for.after_loop

for.loop:                                       ; preds = %for.cond
  %1 = load i32, i32* %i, align 4
  call void @__prelude_print(i32 %1)
  %2 = load i32, i32* %i, align 4
  %3 = add i32 %2, 1
  store i32 %3, i32* %i, align 4
  br label %for.cond

for.after_loop:                                 ; preds = %for.cond
  ret i32 0
}
```

Short-circuit evaluation

Like many modern programming languages, μComp uses short-circuit evaluation for the && and || boolean operators. The generation of boolean short circuit expression is accomplished using `eval_bool_and_exp` (for logical 'and' expressions) and `eval_bool_or_exp` (for logical 'or' expressions) functions defined inside `codegen.ml`. Both functions use the φ [10] node to determine the final expression value.


entry:
br %A0, %and.true, %and.false
false
true
and.true:
br %A1, %and.true1, %and.false
false
...
and.true_n:
br %and.false
and.false:
%0 = phi i1 [%A0, %entry, ...]
entry:
br %A0, %or.true, %or.false
true
false
or.false:
br %A1, %or.true, %or.false1
true
...
or.false_n:
br %or.true
or.true:
%0 = phi i1 [%A0, %entry, ...]

(a) Control flow graph for a short-circuit conjunctive boolean expression $A_0 \wedge A_1 \wedge \cdots \wedge A_n$.

(b) Control flow graph for a short-circuit disjunctive boolean expression $A_0 \vee A_1 \vee \cdots \vee A_n$.

Figure 3: Control Flow Graphs for boolean expressions containing logical $\wedge$ and $\vee$.

Figure 3 shows how short-circuit boolean expressions are translated into a *Control Flow Graph* (from now on abbreviated as CFG). Each node is a `llbasicblock` and edges without

[10] https://en.wikipedia.org/wiki/Static_single_assignment_form

label indicates unconditionally branches. For boolean expressions in *conjunctive form*, like the one shown in Figure 3a, the evaluation follows this schema: evaluates the first term, if it is true then jump to the next basic block to evaluate the next term, otherwise, since the term is false, for the *annulment law* in boolean algebra, the formula will evaluate to false, jumping to the last block (`and.false`). For boolean expressions in *disjunctive form*, shown in 3b, the translation schema is similar to the previous one, but if one expression evaluates to true, it will jump directly to the last block (`or.true`). Both second last nodes in CFGs have a direct edge to the last block.

Listing 14 and 15 show how the test `test-if6.mc` is compiled into LLVM bitcode, handling short-circuit boolean expressions. Figure 4 shows a graphical representation of the CFG.

```
def main() : int {
    var i : int;
    i = 42;
    if(true && change(&i,0) || change(&i,1))
        print(i);
    else
        print(51);
    return 0;
}
```

Listing 14: ”Main function from test-if6.mc file.”

```
define i32 @main() {
entry:
  %i = alloca i32, align 4
  store i32 42, i32* %i, align 4
  br i1 true, label %and.true, label %and.false

and.true:        ; preds = %entry
  %call.__application_change = call i1 @__application_change(i32* %i, i32 ↩
    0)
  br label %and.false

and.false:       ; preds = %and.true, %entry
  %0 = phi i1 [ true, %entry ],
              [ %call.__application_change, %and.true ]
  br i1 %0, label %or.true, label %or.false

or.false:       ; preds = %and.false
  %call.__application_change1 = call i1 @__application_change(i32* %i, i32 ↩
    1)
  br label %or.true

or.true:        ; preds = %or.false, %and.false
  %1 = phi i1 [ %0, %and.false ], [ %call.__application_change1, %or.false ↩
    ]
  br i1 %1, label %if.then, label %if.else

if.then:        ; preds = %or.true
  %2 = load i32, i32* %i, align 4
  call void @__prelude_print(i32 %2)
  br label %if.merge

if.else:        ; preds = %or.true
  call void @__prelude_print(i32 51)
  br label %if.merge

if.merge:        ; preds = %if.else, %if.then
  ret i32 0
}
```

Listing 15: "Emitted LLVM code for the main function."

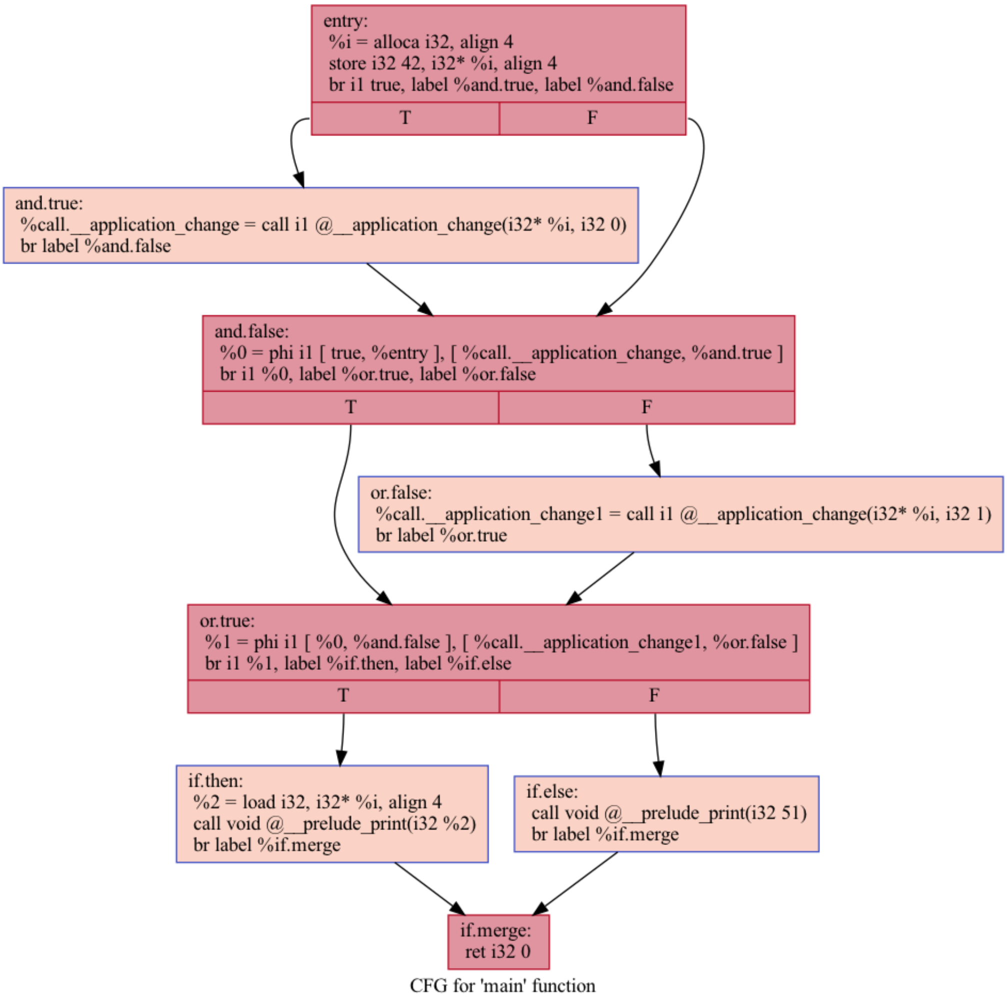

entry:
%i = alloca i32, align 4
store i32 42, i32* %i, align 4
br i1 true, label %and.true, label %and.false
T
F
and.true:
%call.__application_change = call i1 @__application_change(i32* %i, i32 0)
br label %and.false
and.false:
%0 = phi i1 [true, %entry], [%call.__application_change, %and.true]
br i1 %0, label %or.true, label %or.false
T
F
or.false:
%call.__application_change1 = call i1 @__application_change(i32* %i, i32 1)
br label %or.true
or.true:
%1 = phi i1 [%0, %and.false], [%call.__application_change1, %or.false]
br i1 %1, label %if.then, label %if.else
T
F
if.then:
%2 = load i32, i32* %i, align 4
call void @__prelude_print(i32 %2)
br label %if.merge
if.else:
call void @__prelude_print(i32 51)
br label %if.merge
if.merge:
ret i32 0
CFG for 'main' function

Figure 4: CFG for test-if6.mc, generated by **opt** utility provided by LLVM.

5.2 Runtime Support

In addition to the runtime support provided by the professor, I added several functions to generate integer random numbers, print characters onto screen, and so on ...
These functions rely on C Standard Library: `libc`.
The following list contains all the extra function added inside the `Prelude` interface with their respective description.

- `def time(): int`: this function relies on `libc time` function, it just returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
- `def set_rand_seed(seed: int): void`: initialize the `libc` pseudo random number generator using the `seed` parameter as initial seed.
- `def rand(): int`: returns a random number using the `libc rand` function. In macOS, the function relies on `arc4random` to generate random numbers.
- `def ifloat(from: float): int`: converts a floating point number into an integer one.
- `def fint(from: int): float`: converts an integer into a floating point number.
- `def get_float(): float`: reads a floating point number from the `stdin`.

- def get_char(): char: reads a character from the stdin.
- def print_f(n: float): float: prints a floating point number (with a precision of four decimal digits) onto the stdout.
- def putc(c: char): void: prints a character onto the stdout.
- def putendl(): void: simply prints a newline onto the stdout.
- def abort(exit: int): int: it may be used to abort the program execution, returning exit parameter as exit code.

The functions listed before can be found inside rt_support.c file. Each prelude function is name mangled as described in Section 5.1.1. The Listing 16 shows a prelude function to generate a random number.

```
int32_t __prelude_rand() {
  #ifdef __OSX__
    return (int32_t) arc4random();
  #else
    return (int32_t) rand();
  #endif
}
```

Listing 16: ”Prelude’s rand function implementation inside the run-time support file.”

5.3 Optimizations

The optimization phase happens after the LLVM module generation. The optimizer uses the provided optimizations by the professor without any addition.

On my computer, I did not succeed to test the const_propagation optimization because the Opam LLVM package on macOS has several issues.

5.4 Tests and compilation

Each unit test contained inside the test directory has been verified using the provided shell script test_all.sh.

To compile a μComp program, the user requires the following dependencies:

- OCaml ⩾ 4.12.0
- Menhir ⩾ 20210419
- ppx_deriving ⩾ 5.2

All these packages can be installed using Opam package manager. To run the compiler, the user uses the dune [11] build system invoking the command shown in Listing 17.

[11] **Dune** is the default build system for OCaml. For further details, see: https://dune.build

```
# Compile a muComp program
dune exec mcompc -- [options] <input_file>.mc

# Available options to the compiler
  -p Parse and print untyped AST
  -t Type check and print the typed AST
  -l Link and print the linked AST
  -d Compile and print the generated LLVM IR
  -c Compile the source file (default)
  -o Place the output into file (default: a.bc)
  -O Optimize the generated LLVM IR (default: false)
  -verify Verify the generated LLVM module (default: false)
  -help  Display this list of options
  --help  Display this list of options
```

Listing 17: ”Compiler help screen.”

The compiler will output LLVM bitcode to be compiled using the *LLVM system compiler* (`llc`). Once the compilation is done, to make the executable, the user will use `clang` [12] to compile the run-time support file (`rt_support.c`) into an object file. Finally, the user can produce the final file linking the compiled LLVM bitcode and run-time support object using `clang`.

6 Conclusions

We have seen what is μComp programming language and how it is compiled into LLVM bitcode through each previously described phase.

6.1 Future Developments

The compiler can be further improved by using the incremental API by Menhir, which produce better error messages for the parsing phase. The semantic analysis phase can be improved as well (i.e. to detect if all the paths in a function Control Flow Graph lead to a return instruction). The LLVM bitcode can be optimized for the global variable initialization, if an initial expression is trivial we could compute the initial value directly without using the global constructors.

References

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88. URL: `https://llvm.org`.

[TC07] Linda Torczon and Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 012088478X.

[12] **Clang** is a C frontend compiler that uses LLVM. For further details: https://clang.llvm.org