UNIVERSITÀ DI PISA

μcomp-lang

Final project for Languages, Compilers and Interpreters

Alex Colucci

January 2023

1 Introduction

The μcomp-lang compiler project was designed to implement a simple component-based imperative language. The language's main features include the use of components, which are linked together to form complete programs, and the use of interfaces to specify the behavior of components. The project was implemented using the OCaml programming language and various tools and techniques presented during the course.

The project consists of four assignments, which were released incrementally throughout the course, allowing students to start working on the project before the end of classes. The structure of the project includes a lexer, parser, semantic analyzer, code generator, and test suite. Each section of the report will describe the implementation and any challenges or solutions encountered during the development process. Moreover, the assignment asked to implement at least two extensions for the language, we implemented six of them, and we are going to discuss them throughout the document.

The structure of the project is as follows:

- The lexer, implemented with ocamllex, is responsible for converting the source code into a stream of tokens for the parser.
- The parser, implemented with Menhir, is responsible for converting the stream of tokens into an abstract syntax tree (AST) representing the structure of the program.
- The semantic analyzer is responsible for checking the AST for any semantic errors, such as type mismatches or undefined variables.
- The code generator, implemented with the bindings in OCaml for LLVM, is responsible for generating the final executable code from the AST.

- The test suite, is used to ensure that the compiler functions as expected and can handle a variety of inputs and edge cases.

The Listing 1 is an example of code written in µcomp-lang.

```
interface Math {
  def add(a : int, b : int) : int;
}

component MathLib provides Math {
  def add(a: int, b : int) : int {
    return a + b;
  }
}

component Main provides App uses Math {
  var first_number : int;

  def main() : int {
    var second_number : int = 1;
    first_number = 41;

    print(add(first_number, second_number));    // output: 42

    return 0;
  }
}

connect {
  Main.Math <- MathLib.Math;
}
```

Listing 1: Example of a simple math component.

1.1 Grammar

The grammar for µComp-lang is defined in Extended Backus-Naur Form (EBNF) notation. Non-terminal symbols are denoted by angle brackets, while tokens containing a value are represented by uppercase words. The grammar has been enhanced to support the extensions that will be described in Section 3.3, allowing for a more expressive and powerful language.

$\langle$`CompilationUnit`$\rangle$ ::= $\langle$`TopDecl`$\rangle$* EOF

$\langle$`TopDecl`$\rangle$ ::= ``interface'' ID ``{'' $\langle$`IMemberDecl`$\rangle$+ ``}''
 | ``component'' ID $\langle$`ProvideClause`$\rangle$? $\langle$`UseClause`$\rangle$? ``{'' $\langle$`CMemberDecl`$\rangle$+ ``}''
 | ``connect'' $\langle$`Link`$\rangle$ ``;'' | ``connect'' ``{'' ($\langle$`Link`$\rangle$ ``;'')* ``}''

$\langle$`Link`$\rangle$ ::= 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⟩ ‘‘;’’)* ‘‘}’’

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

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

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

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

⟨LValue⟩ ::= ID | ID ‘‘[’’ ⟨Expr⟩ ‘‘]’’

⟨BinOp⟩ ::= ‘‘+’’ | ‘‘-’’ | ‘‘*’’ | ‘‘%’’ | ‘‘/’’ | ‘‘&&’’ | ‘‘||’’ | ‘‘<’’ | ‘‘>’’ | ‘‘<=’’ | ‘‘>=’’ | ‘‘==’’ | ‘‘!=’’

2 Scanner

The lexical analysis of the language is implemented using Ocamllex, which utilizes regular expressions to produce a stream of tokens. The implementation can be found in the file `scanner.mll`. The first section of the scanner contains a header that includes:

- An hash table used to store the keywords of the language: `interface`, `uses`, `provides`, `component`, `connect`, `def`, `var`, `return`, `if`, `else`, `for`, `while`, `do`, `int`, `float`, `char`, `void`, `bool`, `true`, `false`.
- Utility functions that are mainly used to enforce constraints on the tokens.

In case of errors during constraint checks, the `Lexing_error` exception is raised with the position of the error and a helpful error message. The checks applied to enforce constraints include:

- `int32_check_interval`, to check that a given integer is a valid 32-bit signed integer.
- `float32_check_interval`, to check that a given float is a valid 32-bit float number.
- `identifier_check_max_lenght`, to check that the length of a given identifier is not more than 64 characters long.

The second section of the scanner, that is contains the function `next_token`, is the entry point of the scanner and contains the rules defined with regular expressions. The rules are used to create tokens for whitespace, newlines, numbers (e.g. integers in base 10, integers in base 16 and floats), chars, identifiers, binary operators, punctuations and finally single line and multi line comments. The interesting thing to mention are

- floats rules enables to accept the standard float notation (e.g. `4.2`, `4.`, `.2`) but even the scientific notation (e.g. `0.42e2`, `0.42e-1`)
- chars rules start by looking at the first apostrophe, then calls the `character` rule to check if there is a character. Then the `character_parser` is done and there is a check to see if it's closed or not. The parsing takes in account the possibility of special characters, and in that case an escape sequence is applied.
- identifiers, have a maximum length of 64 and are checked against the hash table of keywords to be sure that there is no conflict with identifier name defined by the user-persona of the language
- comments, can be single line (i.e. `// comment`) and multi line (i.e. `/* comment */verb`). The `single_line_comment` rule is a helper to parse the first case, that doesn't emit a token until it finds the first newline. On the other hand, the `multi_line_comment` is a helper that doesn't emit a token until it finds the closing match bracket for multi line comments.

3 Parser

This section will be divided into two parts: the first will discuss the representation of the abstract syntax tree (AST), and the second will cover the implementation of the parser.

3.1 AST

The representation of the AST is contained in the file `ast.ml`. It is defined with a generic type to keep track of the annotation, which separates the structure of the tree from its usage. The main uses of the annotation are as the output of the parser, where it represents the position given by Ocamllex, and as the output of the semantic analysis, where it represents the resulting type.

The file also contains several helper functions, such as `annotate_node` for annotating nodes, `show_typ` for a more readable string representation of types, and `is_primitive`, `is_scalar`, etc. for checking conditions over types and binary operators to simplify later phases.

3.2 Implementation with Menhir

The parser is implemented using Menhir, an LR(1) parser generator. It takes a grammar specification and produces the OCaml code for the parser, which outputs the AST annotated with positions. The

```
%nonassoc NO_ELSE
%nonassoc ELSE

%right ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN DIV_ASSIGN MOD_ASSIGN

%left OR
%left AND

%left EQ NE
%nonassoc LT GT LE GE
%left PLUS MINUS
%left TIMES DIV MOD

%right MINUSMINUS

%nonassoc NEG // unary minus
%nonassoc NOT
```

Listing 2: Precedence rules and associativity of the tokens.

definition can be found in the file `parser.mly`.
Firstly, the needed tokens to define the grammar are declared. Then, the following associativity status and priority levels are defined to allow shift/reduce conflicts to be silently resolved (Listing 2). Regarding the dangling else problem, this has been solved by exploiting the precedence rules and by defining the IF structure as in Listing 3. The defined rules are straightforward, but the starting

```
stmt:
...
  | "if" "(" e=expr ")" s1=stmt "else" s2=stmt
    { Ast.If(e, s1, s2) $$ $loc }
  | "if" "(" e=expr ")" s1=stmt %prec NO_ELSE
    { Ast.If(e, s1, Ast.Skip $$ $loc) $$ $loc }
...
;
```

Listing 3: The IF definition in the parser to solve the problem of the dangling else.

symbol, `compilation_unit` (see Listing 4), deserves special mention. This variant type has one variant constructor, `CompilationUnit`, which takes a record as an argument to keep track of the three main parts of the language: interfaces, components, and connections. The rule matches any top-level declaration, and the helper function builds the convenient variant type.

3.3 Extensions

In general, for the extensions, we opted for a no de-sugaring approach. Using extra nodes in the AST for variable declarations allows for a clear representation of where variables are being declared and what types of variables they are, making the compiler's code easier to understand and less prone to

```
compilation_unit:
  | td=top_decl* EOF
    {
      let rec build_unit inter_list comp_list conn_list = function
        | [] -> (Ast.CompilationUnit({ interfaces = inter_list; components = comp_list;
                  connections = conn_list; }))
        | Ast.Interface(i) :: xs -> build_unit (i :: inter_list) comp_list conn_list xs
        | Ast.Component(c) :: xs -> build_unit inter_list (c :: comp_list) conn_list xs
        | Ast.Connection(c) :: xs -> build_unit inter_list comp_list (c @ conn_list) xs
        in build_unit [] [] [] td
    }
;
```

Listing 4: The compilation unit of the AST.

errors. Additionally, it also allows for more flexibility in the code generation process, as it enables the possibility of generating different code depending on the type of variable being declared, and preserves the original semantics of the code, this way the generated code is more predictable and reliable. Let's analyze the extensions one by one.

Do-while Regarding the do-while loop extension, the idea was to define a new node in the AST to avoid any form of de-sugaring that could lead to a more complex code due to problems in the management of the scopes. The needed token were added, and the new node was defined as shown in Listing 5.

```
and 'a stmt_node =
  ...
  | DoWhile of 'a expr * 'a stmt                                  (* Do-While loop *)
  ...
```

Listing 5: Do-while extension effect on the AST.

Pre/post increment/decrement About the pre-post increment the de-sugaring was possible, but we opted even here to have ad-hoc nodes in the AST. For the pre-increment, it is possible to treat it as an assignment `++i → i = i + 1`. On the other hand, for the post-increment, this would have meant `i++ → x = i; i = i + 1; x` which is not very elegant and especially this would have required the code generation of a temporary variable. Thus, to keep things more straightforward, we opted for the variant types and AST node in Listing 6.

Assignment binary operator For what it concerns the third extension, the abbreviation for the assignment operators (e.g. `x += 2`), no de-sugaring was made. The reason is that by doing so, the de-sugaring process would have translated the operation like this `x += 2 → x = x + 2`. Nevertheless, in the case the left value has an increment/decrement operator, this would have disrupted the semantic by doing the increment/decrement more than once (i.e. `arr[i++] += 42` $\longrightarrow$ `arr[i++] = arr[i++] + 42`). So the problem has been solved by treating the abbreviation for

```
type incdec_typ = Inc | Dec [@@deriving show, ord, eq]
type incdec_order = Pre | Post [@@deriving show, ord, eq]

and 'a expr_node =
  ...
  | IncDec of 'a lvalue * incdec_typ * incdec_order       (* e.g. ++i, i++, --i, i-- *)
[@@deriving show, ord, eq]
```

Listing 6: Pre/post increment/decrement effect on the AST.

the assignment operators as a new node in the AST and by adding the needed tokens. The new node is defined as in Listing 7.

```
and 'a expr_node =
  ...
  | AssignBinOp of 'a lvalue * binop * 'a expr            (* x+=e or a[e]+=e *)
  ...
```

Listing 7: Assignment binary operator effect on the AST.

Variable declaration with initializer For the fourth extension, to accommodate for variable declarations with initialization for local and global variables, we decided to add an optional `'a expr parameter` to the `VarDecl` and `LocalDecl` nodes in the AST (Listing 8). This allowed us to represent variable declarations with initialization, by giving the option of including an expression for the initialization value. This change allowed us to handle variable declarations with initialization differently from those without during the semantic analysis and code generation stages, by giving us the ability to distinguish variables that are initialized and those that are not. Obviously, the changes respect the specification by not allowing array initialization like for the case of the assignment, and no initialization for interface variables.

```
and 'a member_decl_node =
  (* A member of an interface or of a component *)
  ...
  | VarDecl of vdecl * 'a expr option
[@@deriving show, ord, eq]

and 'a stmtordec_node =
  | LocalDecl of vdecl * 'a expr option   (* Local var declaration w/ optional initializer *)
  ...
[@@deriving show, ord, eq]
```

Listing 8: Variable declaration with initializer effect on the AST.

Floating point arithmetic The fifth extension, the floating point arithmetic, has been implemented by adding to the AST, a new type construct `TFloat` for the `typ` variant type, and a new type construct `FLiteral` for the `expr` variant type. Moreover, the tokens for the float numbers have been added to the parser.

Function overloading The last extension that has been implemented, that is the overloading of functions, dint't require any changes to the AST.

4 Semantic analysis

The semantic analysis phase of the compiler aims to verify the well-formedness of the program and add type information to the abstract syntax tree. One of the key data structures used in this phase is the symbol table, which keeps track of the variables and their associated information, such as type and location, in a lexically scoped manner.

4.1 Symbol table

Here a brief overview on the symbol table, an important data structure used in compilers. The goal is to keep track efficiently the variables with the associated information that is needed during the visits. This ensures a way to implement the concept of lexical scope (a.k.a. static scope), that is, informally, when variables refer to portion of source code rather than portion of runtime.

Implementation The symbol table is trivially implemented as a linked list of hash tables. Each hash table represents a scope, and the linked list acts as a stack, where each hash table points to the enclosing parent scope. Whenever a new scope is entered (e.g. `begin_block`), a new hash table is pushed to the head of the list. On the other hand, when a scope is ended (e.g. `end_block`), the head of the list is popped. The variable is always inserted through `add_entry` function, which adds the variable to the hash table in the head of the list. In addition, it ensures that no identical variables can be defined in the same scope, otherwise a `DuplicateEntry` exception is raised. While, the lookup function traverses the list and returns the first entry found. Thus, variables declared in inner scopes can shadow previously declared variables. Finally, the efficiency of the symbol table is guaranteed by the use of hash tables, which have a time complexity of $O(1)$ for insertion and lookup on average. And, the final complexity is $O(n)$ for the lookup, where n is the number of scopes.

Every implementation detail can be found in the file `symbol_table.ml`, where other helpful functions are implemented, such as `fold` and `iter` to respectively fold and iterate over the current scope.

Usage Now we make more clear how the table is used in order to satisfy the following requirements regarding the scoping mechanism of the language, that are:

- The global scope contains all interface and component declarations, which are mutually recursive in nature.
- Each interface and component has its own scope, which contains its own set of declarations that are mutually recursive.
- Functions also have their own scope, separate from the interface and component scopes.

- Blocks can be nested within functions, and a variable declared in an inner block will hide any possible declarations of the same variable in outer blocks.

Before understanding how all the requirements were implemented and how the semantic check was performed, we need to expand on the details of the symbol table that played a key role in the whole process. Foremost, the type `identifier_info` was defined to keep track of the name, type and location of a specific identifier. An identifier is represented by a variant type `symbol` that will maintain extra information about the identifier and based on the tag it can represent a variable, function, interface or component. This information will be the value associated to a key in a given symbol table (Listing 9).

- `VarSymbool`, is just an alias for the `identifier_info` type
- `FunctionSymbol`, contains the `identifier_info` and the symbol table to keep track of its parameters and local variables
- `InterfaceSymbol`, contains the `identifier_info` and the symbol table to keep track of its declarations of functions and variables.
- `ComponentSymbol`, contains the `identifier_info` and three symbol tables to keep track of its uses and provides interfaces, together with the definitions of functions and variables.

```
type identifier_info = Ast.identifier * Ast.typ * Location.code_pos

type symbol =
  | VarSymbol       of identifier_info
  | FunctionSymbol  of identifier_info * symbol Symbol_table.t
  | InterfaceSymbol of identifier_info * symbol Symbol_table.t
  | ComponentSymbol of identifier_info * symbol Symbol_table.t * symbol Symbol_table.t
                     * symbol Symbol_table.t
```

Listing 9: The definitions of the symbols that will act as the value associated to a key inside the symbol table.

Finally, to keep everything tidy during the passing of data around, some helper types were used (Listing 10) to a have a `global_symbol_table` to keep track of the interfaces and components, and an `env` type to keep track of the current scope and the current component being checked.

4.2 Check

Given what has been said in the previous sections, now we have the tools to understand how the check works. The implementation can be found in `semantic_analysis.ml`. The module takes in input the AST annotated with locations, and it produces a semantically checked AST that is annotated with `Ast.typ` types. The analysis is divided into four submodules, the first one is responsible for the management of the Standard library, the second and third one are responsible for the visit of interfaces and components respectively and the last one does the second visit over the AST.

```
type global_symbol_table = {
  components  : symbol Symbol_table.t;  (* Components table *)
  interfaces  : symbol Symbol_table.t;  (* Interfaces table *)
}

type 'a env = {
  current_table : symbol Symbol_table.t;
  component : 'a Ast.component_decl_node;
  component_symbol : symbol;
}
```

Listing 10: The kind of data that is passed around to the functions during the semantic analysis.

4.2.1 Standard Library

The `StandardLibrary` submodule is responsible for managing the standard library, which is defined in `mcomp_stdlib.ml`. This submodule appends the interfaces that represent the standard library to the list of interfaces in the compilation unit (Listing 4). The file contains two lists: `app_signature`, which contains the main signature, and `prelude_signature`, which contains several helpful functions such as `print`. The `App` and `Prelude` interfaces are added to the list of interfaces in the compilation unit at the end of the process.

4.2.2 First Visit

The first visit populates the global symbol table with the information about interfaces and components, which is needed to execute a second visit that can be aware of references to other components.

Interfaces The `InterfaceVisitor` submodule starts by scanning all interfaces and their relative variable and function declarations. It checks that the members are valid (e.g. valid signature, valid return type, etc.), and that there are no duplicate interfaces. If everything is sound, an `InterfaceSymbol` is added to the global symbol table for each interface, containing the information about the interface and a symbol table of the declarations that will be used later on.

Components The `ComponentVisitor` submodule starts by scanning all components and their relative variable and function declarations. The goal is to ensure the following properties:

- The Prelude interface is added to each component, and no component provides it
- Only one component provides the App interface, and no component uses it
- No component with the same name already exists
- Functions and variables have a valid signature and are unique within the component
- There are no collisions in the interfaces used by a component (e.g. the used components provide the same function to the component)
- A component only uses a defined interface once and each use is valid

- A component that provides an interface provides an existing interface at most once, and implements the required functions and variables properly

If everything is sound, a `ComponentSymbol` is added to the global symbol table for each component, containing the information about the component and three symbol tables of the declarations that will be used later on. The first symbol table is for the uses interfaces, the second one is for the provides interfaces and the third one is for the definitions of functions and variables.

4.2.3 Second visit

The **Second Visit** of the semantic analysis is responsible for performing the final necessary checks and type checking on AST produced by the parser. This step allows for the search of available interfaces and components, enabling additional checks that were not possible during the first visit. The output is the AST annotated with types.

One of the main design choices in this phase is the handling of **global variable initialization**. To improve code readability and reduce the likelihood of bugs, we have chosen to only allow the optional initialization of primitive types with constant expressions. From a compiler perspective, this decision also allows for better optimizations and more efficient code generation.

During the semantic analysis, the µComp-lang compiler has been enhanced to support function overloading (extension Section 3.3), which allows multiple functions with the same name to be defined as long as they have different parameter types. However, it's important to note that this feature only applies to functions with primitive parameter types and does not take into account the return type. This has been achieved by using the mangling technique, which takes the name of the function and serializes the types of the parameters, obtaining a unique function name that takes into account the signature of the parameters and the number of parameters. The function is found in the `Utils.manglify_function` and emits a function name with the format `N_{function_name}T{types}` if the function has some parameters, or `N_{function_name}_T_v` if it doesn't. For example, the function `def add(x : int, y : char)` would be mangled to `N_add_T_i_c`. In this way, by working on the mangled function name instead of the original name, function overloading is implemented in a simple and efficient manner.

Then during the analysis of **function definition**, the module checks the various statements (e.g. if, for, while, return, etc.) in the following ways:

- Statements are annotated with a meaningless `Ast.TVoid` type.
- Guard expressions in `if`, `while`, `do-while`, and `for` statements are checked to be of type `Ast.TBool`, and a new scope is defined for each of them.
- For the `for` statement, if a stop condition is missing, it is treated as a constant true boolean.
- The return type of the function is checked to match the one defined in the signature, and if missing, it is treated as a void return.
- Each block defines a new scope, and local variable declarations are checked to be unique within these scopes.

During the analysis of `expr`, the module checks that:

- Every expression is sound with respect to the typing system.
- Access to a variable is checked to be a variable defined in the hierarchy of scopes. First, it checks that it is a variable in the scope of the function. Then it checks that it is a global variable of the component, and in that case it qualifies it with the name of the component. Otherwise, it checks that it is a variable from one of the used interfaces, in that case it qualifies it with the name of the interface. The latter case will be successively qualified with the proper component name during the linking phase (Section 5).

Finally, when analyzing a **function call**, the module performs the required checks to ensure that the types of the arguments match the signature of the function. A similar search is executed for the function as for the variables, that is, first the function is looked up in the definitions' table of the component and in that case it's qualified with the name of the component. Otherwise, it is looked up across all the used interfaces, and in that case it's qualified with the name of the interface.

4.2.4 Typing

The type system in µComp is designed to provide a solid foundation for the language. The analysis proves that the typing system is sound with respect to the semantics of the language. The second pass of the semantic analysis is specifically designed to guarantee that these rules are upheld throughout the program.
The language supports a range of types, including `int, char, float, bool`, and reference types for these. Additionally, single dimension array types for these types are also supported. However, types such as arrays and references cannot be returned by functions.
The `TypeAnalysis` submodule includes functions such as `is_assignment_legal`, `is_binary_op_valid`, and `is_unary_op_valid`, which handle various type-related operations. These include:

- Enabling assignments when the types being assigned are equal, and the types are either two primitives, two primitive references, one primitive and one primitive reference, or vice versa (check Listing 11).
- Enabling only binary operations (e.g. +, −, >, >=) between variables of the same primitive or primitive reference type.
- Forbidding type coercion, although the standard library provides a way to convert between int and floats.
- Forbidding assignments of `void` expressions or arrays.
- Limiting negation only to `int` and `float` expressions.
- Limiting not only to `bool` expressions.
- Restricting increment and decrement operators to be applied only to int and float expressions.

5 Linking

The linking phase plays a crucial role in ensuring the integrity of the program during the compilation process. The main objectives of this phase are to check the validity of the connections between

```
let is_assignment_legal t1 t2 loc =
  match (t1, t2) with
  (* Same primitive type *)
  | t1, t2 when (Ast.equal_typ t1 t2) && (Ast.is_primitive t1) -> ()
  (* Same primitive reference type *)
  | Ast.TRef(t1), Ast.TRef(t2) when (Ast.equal_typ t1 t2) && (Ast.is_primitive t1) -> ()
  (* Left primitive reference type and right primitive type *)
  | Ast.TRef(t1), t2 when (Ast.equal_typ t1 t2) && (Ast.is_primitive t1) -> ()
  (* Right primitive type and left primitive reference type *)
  | t1, Ast.TRef(t2) when (Ast.equal_typ t1 t2) && (Ast.is_primitive t1) -> ()
  (* Forbid array assignment *)
  | Ast.TArray(_), Ast.TArray(_) ->
    let err_msg = Printf.sprintf "Array assignment is not supported yet." in
    let help_msg = Printf.sprintf "Use a loop to assign each element of the array." in
    semantic_error loc err_msg help_msg
  (* Forbid void assignment *)
  | _, Ast.TVoid ->
    let err_msg = Printf.sprintf "Cannot assign a 'void' expression to an lvalue." in
    let help_msg = Printf.sprintf "Use a return statement to return a value." in
    semantic_error loc err_msg help_msg
  | _, _ ->
  let err_msg = Printf.sprintf "Cannot assign '%s' to '%s'." (Ast.show_typ t1) (Ast.show_typ t2) in
  let help_msg = "The types of the left and right hand side of the assignment must be the same." in
  semantic_error loc err_msg help_msg
```

Listing 11: The way assignment is handled during semantic to ensure a sound typing system.

different components and to qualify all external names with the appropriate component. The linking phase is implemented in the file `linker.ml`, and it is divided into two main submodules: Checker and Qualifier.

Checker The Checker module contains the function `visit_links` that is responsible for verifying that all the connections inside the program are valid. During this first visit, a simple Hashtable is built, in order to keep track of the information about components and interfaces. It checks that the connections meet the following criteria:

- The two components that are being connected are distinct.
- The interfaces being used and provided match the correct components.
- The interfaces being used are all satisfied and provided by a connection.
- The interface names match across the connection.
- There are no conflicts with previous established connections.

For example, if we have the following connection statement,

```
connect component1.interface1 <- component2.interface2;
```

the linker will check that:

- component1 and component2 are different components
- interface1 is an interface used by component1
- interface2 is an interface provided by component2
- interface1 and interface2 have the same name
- there are no other connections between component1.interface1 and another component.

Qualifier Once the verification phase is complete, the Qualifier module is responsible for linking the external interface names to the appropriate component. The entrypoint function for this module is `visit_components` that will exploit the hashtable created during the previous step to qualify components. In other words, for variables and function calls that are external to a component, in the AST the inteface name is defined. By using the hashtable, the linker will look up the interface name to get the component that implements it according to the connection rules. And the component will be used in place of the interface name in the AST. Hence, the qualification is applied.

In the example Listing 1, this means that the node for the call to the add method is `Ast.Call(Some "Math", "add", ...)`, but after the qualification it will be `Ast.Call(Some "MathLib", "add", ...)` .

6 Code generation

In this section, we will discuss the code generation phase of our compiler, which utilizes the LLVM OCaml bindings to output optimized machine code. The primary goal of this phase is to take the abstract syntax tree (AST) generated during semantic analysis and translate it into LLVM's intermediate representation (IR).

6.1 Symbol table usage

During this phase, the symbol table holds different information compared to the one used during semantic analysis. The table now contains LLVM's llvalues. The symbol table is passed through the codegen functions, with a helper type that can be either the global context or the function context. The global context holds a reference to the `ll_module`, the global symbol table, the current component name, and the global constructors function used for global variable initialization. On the other hand, the function context holds additional information about the current function, including a reference to the `ll_builder`, the `ll_value` representing the function signature, and the local symbol table that keeps track of local variables. This allows easy access to the symbol table and the LLVM module from any part of the codegen in a clean and efficient way. Listing 12 shows an example of the kind of data that is passed around to the functions during codegen.

6.2 Implementation

The code generation process is implemented in the `codegen.ml` file, which is divided into two submodules: `Declare` and `Codegen`. The first step is to define the code generation environment, as discussed in the previous section. The `Declare` module is called first, and the `Codegen` module is called second. In the following sections, we will delve into the key aspects of the implementation.

```
type global_symbol_table = {
  functions : L.llvalue Symbol_table.t;   (* Functions table *)
  variables : L.llvalue Symbol_table.t;   (* Global variables table *)
}

type global_context = {
  ll_module : L.llmodule;                 (* LLVM module *)
  table : global_symbol_table;            (* Global symbols table *)
  cname : string option;                  (* Name of the component *)
  global_ctors:  L.llvalue * L.llbuilder; (* Global constructors function *)
}

type fun_context = {
  ll_builder : L.llbuilder;               (* LLVM builder *)
  ll_value : L.llvalue;                   (* LLVM function *)
  table : L.llvalue Symbol_table.t;       (* Local variables table *)
  global_ctx : global_context;            (* Global context *)
}
```

Listing 12: The kind of data that is passed around to the functions during codegen.

6.2.1 Global variable initialization

When variable declarations and assignments are combined in one line (see Section 3.3), we need to initialize global variables in the LLVM module. From a design standpoint, this could be achieved by only allowing constant expressions or allowing more complex operations. As discussed in Section 4.2.3, we chose the first approach for several reasons. However, we were also intrigued by the possibility of having a general solution for initialization to facilitate future work. To that end, we used global constructors functions, which, according to LLVM documentation, are functions that are called before the execution of the main function. A global array of pointers, `llvm.global_ctors`, keeps track of the functions to call. The procedure is executed by the function `Codegen.gen_global_ctors_function` that defines an internal function `__llvm__global_ctors`, which contains the code for initializing global variables.

6.2.2 Declare prelude and components

The `Declare` module is responsible for the preliminary declaration of prelude functions from the Standard library, as well as the declaration of the members of each component. This allows for functions to refer to each other and access global variables. The process is made simple by using the `Declare.declare_function` function for the prelude, and the `Declare.component` function for the components' members.

6.2.3 Codegen for components

In the core of the code generation process is the `Codegen` module, where the emission of IR code takes place. The entry point is the `Code.gen_component` function, which generates code only for the functions of each component. The function starts by looking up the previously declared function in the global symbol table, creating a builder at the end of the function and a new scope for variables.

It then allocates the arguments into the stack, and calls the `Code.gen_stmt` function to generate the statements. At the end of the process, an ending instruction may be added based on the result of the previous statement's code generation.

6.2.4 Return management and statements

A crucial aspect of the code generation for components is how we handle the return statement. The µcomp-lang supports returning a primitive value, void, or if the function signature does not explicitly specify a return type, it is treated as void. Additionally, we must also consider the possibility of multiple return statements. To handle this, we have implemented two key functions.

The first is the `add_ending_instruction` (see Listing 13), which takes in a terminator instruction and a builder. It checks if the current block already has a terminator instruction and, if not, adds the terminator. This function is used to avoid the possibility of adding multiple return statements. It is called throughout the code generation whenever terminators such as return or conditional branches are needed.

```ocaml
let add_ending_instruction terminator builder =
  (* If a terminator instruction is already present, do nothing, else add the terminator *)
  let existing_terminator = L.block_terminator (L.insertion_block builder) in
  match existing_terminator with
  | Some(_) -> ()
  | None -> ignore (terminator builder)
```

Listing 13: The helper function to add ending instruction used for return statements.

The other function is the `gen_stmt` (see Listing 14), which is a recursive function that generates the statements. Whenever the function encounters a return statement, it returns false to signal that code generation must terminate. In other cases, it returns true to signal that code generation can continue. The termination logic is handled when the `gen_stmt` function matches an `Ast.Block(stmtordec)`. The function continues to generate code for the list of statements only if the previous statement does not return false. This ensures that any successive statements after a return statement, also known as unreachable code, are skipped and the code generation terminates. In the case where a return statement is missing, when the code generation for the body of the function terminates, the function adds the return statement with a value that is the default value of the return type, as described in Section 6.2.3.

6.2.5 Access of lvalue

The function `gen_lvalue` is responsible for handling the access to a lvalue. It takes in optional parameters such as `address` and `load_value`, with default values of false. The process begins by searching for the variable in the function context table. If it is not found there, it continues to search in the global context table. Depending on the result, it can be an access to a variable or an access to an element of an array. The optional parameter `load_value` acts as a flag to determine whether the value of the variable should be loaded or not. This allows for the avoidance of duplicated code for the different access scenarios. The optional parameter `address`, on the other hand, is used to obtain the address of a variable (e.g. `&x`) or to assign an address to a primitive reference during assignment.

```
let rec gen_stmt fun_ctx stmt =
...
| Ast.Block(stmtordec) ->
      (* Create a new scope for the block *)
      let fun_ctx = {fun_ctx with table = Symbol_table.begin_block fun_ctx.table} in
      let continue_codegen = gen_stmtordec fun_ctx in
      (* Execute the codegen over the list of stmtordec. The List.for_all checks that
         the predicates holds for all the elements. But it uses short-circuiting for
         the "and" evaluation, so at the first false returned by the codegen over a stmt,
         it stops the codegen. *)
      List.for_all continue_codegen stmtordec
...
```

Listing 14: The function to generate statements.

6.2.6 Expressions

The function `gen_expr` handles all expressions defined in the abstract syntax tree (AST). The logic is straightforward, but special attention must be given to the code generation for `Ast.BinaryOp(binop, e1, e2)` and `Ast.AssignBinOp(...)`.

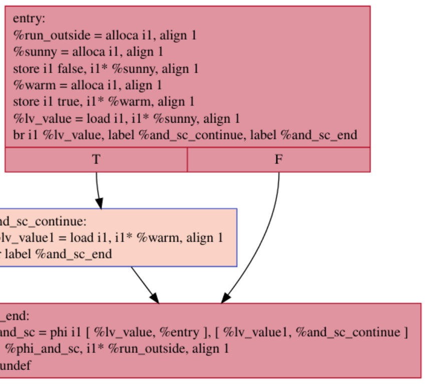
Short-circuiting for bool operators When generating code for the `Ast.BinaryOp(binop, e1, e2)`, two closures are defined to evaluate the left (`e1`) and right (`e2`) expressions. The code generation for numeric operators is handled by the `gen_binary_op` function. However, for boolean operators, a more elaborate procedure is needed to guarantee short-circuiting evaluation. Here, we will demonstrate the `gen_and_short_circuit` function for the `and` operator, as the explanation is similar enough for the `or` operator as well.

Consider an expression $e1$ && $e2$ && $\ldots$ e_n. Short-circuiting allows for more efficient evaluation by stopping as soon as one of the expressions is false. To achieve this, basic blocks called `and_sc_continue` and `and_sc_end` are created, along with the use of conditional jumps and ϕ-nodes. The code to evaluate the first expression is generated, and based on its value, a conditional jump is added to either the `and_sc_continue` block if the first expression is true, or directly to the `and_sc_end` block. The `and_sc_continue` block is used to evaluate the second expression and will unconditionally jump to the final block. The `and_sc_end` block is used to evaluate the final result, which is false if the first expression is false, otherwise it is the result of the second expression. The ϕ-node is used to merge the two possible paths of the code and to return the final result. The code generation for the Or operator is similar. A visual representation of the control flow graph can be seen in Figure 6.2.6 along with the related code example.

Assignment B. Operation When dealing with the `Ast.AssignBinOp(lvalue, binop, expr)`, it is important to ensure that the lvalue is not evaluated multiple times. This is to prevent breaking the semantics in specific edge cases. To accomplish this, we use closures to generate the code for the lvalue and the expr only when necessary. This ensures that the lvalue is only evaluated once and the semantics of the language are preserved.

```
def main() : int {
    var run_outside : bool;
    var sunny : bool = false;
    var warm : bool = true;

    run_outside = sunny && warm;
    return 0;
}
```



```
entry:
%run_outside = alloca i1, align 1
%sunny = alloca i1, align 1
store i1 false, i1* %sunny, align 1
%warm = alloca i1, align 1
store i1 true, i1* %warm, align 1
%lv_value = load i1, i1* %sunny, align 1
br i1 %lv_value, label %and_sc_continue, label %and_sc_end
```

```
and_sc_continue:
%lv_value1 = load i1, i1* %warm, align 1
br label %and_sc_end
```

```
and_sc_end:
%phi_and_sc = phi i1 [ %lv_value, %entry ], [ %lv_value1, %and_sc_continue ]
store i1 %phi_and_sc, i1* %run_outside, align 1
ret i32 undef
```

Figure 1: Example of short-circuiting AND code with its LLVM CFG.

7 Miscellaneous

In this section, we will briefly discuss several topics that do not warrant their own sections.

7.1 Optimization

The optimization phase is executed after the LLVM modules have been generated. We apply several optimizations to improve performance, including:

- Memory to register promotion: This optimization promotes memory accesses to register accesses. This can improve performance by reducing the number of memory operations, which are generally slower than register operations.
- Reassociation: This optimization reorders expressions to make them more amenable to other optimization passes. This can improve performance by making it easier for other passes to optimize the code.
- Dead store elimination: This optimization removes any store to memory operations that are not used by any later instructions.
- Loop unrolling: This optimization unrolls loops to reduce the number of iterations and improve performance by reducing the overhead of the loop control instructions.
- Dead code elimination: This optimization removes any code that has no effect on the program's output.
- Control-flow graph simplification: This optimization simplifies the control-flow graph of the code.
- Tail call elimination: This optimization eliminates tail calls, which are function calls made as the last action of a function.

7.2 Standard Library

The standard library offers a set of functions that can be automatically used in the language and are contained in the hidden Prelude component. These functions include:

- `print(t)` to print any primitive `t` (e.g. int, float, char, bool) on a new line
- `put(t)` to print any primitive `t` (e.g. int, float, char, bool) without an ending new line
- `getint(), getfloat(), getchar(), getbool()` to get one of the primitives from the command line
- `int_to_float()` to convert an integer to a float
- `float_to_int()` to convert a float to an integer

7.3 Test and Compile

To test the compiler, we have provided a test suite in the folder `test/`. The bash script `testall.sh` can be used to run all the tests. To compile an arbitrary µcomp-lang program, we have provided a helper script, `main.sh`. The program can be written in the `main.mc` file and then by running the `main.sh` file, it will be compiled and executed.

7.4 Benchmark

To evaluate the performance of our compiler, we created a small benchmark suite comparing the execution time of a prime number generation program and a factorial program, implemented both in our µcomp-lang and in C. The results, obtained by running the `benchmark.sh` script with the tool hyperfine on a machine with an AMD Ryzen™ 7 PRO 4750U processor with a base clock of 1.7GHz, showed that the performance of µcomp-lang is comparable to that of C. The prime number generation program in particular had statistically significant results. A summary of these results can be found in Table 1.

Program	Language	Mean [s]	Min [s]	Max [s]	Relative
`./primes_mc.out`	ucomp	6.130 ± 0.119	6.009	6.359	1.02 ± 0.02
`./primes_c.out`	c	6.003 ± 0.028	5.962	6.051	1.00
`./factorial_mc.out`	ucomp	0.8 ± 0.5	0.0	2.2	1.13 ± 1.05
`./factorial_c.out`	c	0.7 ± 0.5	0.0	2.2	1.00

Table 1: Benchmark executed with hyperfine, between µcomp-lang and C-lang for primes and factorial.

7.5 VS Code extension

To improve the developer experience, we also created a plugin for the popular code editor VS Code. This plugin provides syntax highlighting for the µcomp-lang language, making it easier to read and write code in the language. The plugin can be easily installed by moving the `vscode-extension` folder, renaming it to `mcomp` and placing it in the `<user home>/.vscode/extensions` directory.

8 Conclusion and Future Works

In this document, we have presented the design and implementation of a compiler for a simple programming language called µcomp-lang. The compiler is built in OCaml, and it takes as input a source code file and produces optimized machine code as output. The project was an intensive one that required a significant effort, but it was also a fun and enlightening experience.

As for future work, one possible direction would be to use the incremental API provided by Menhir to produce more meaningful error messages for the parsing phase. Additionally, a unification algorithm could be implemented to offer a more elegant and efficient way to perform type checking, and to offer different forms of polymorphism. Overall, this project was a great opportunity to learn about compilers and the intricacies of programming languages, and we hope that it will be of use to others looking to gain a similar understanding.