Compilers Project Report

University Of Pisa, Department of Computer Science

Languages, Compilers And Interpreters

Academic Year: **2024 - 2025**

STUDENT:

SALMI ABDERRAHMANE
704608
a.salmi1@studenti.unipi.it

Table of Contents

Introduction

This report is for the project of the module Languages, Compilers and Interpreters, in which we implemented interpreters and a compiler for three educational languages: MiniImp, MiniFun, and MiniTyFun. MiniImp is a simple imperative language, MiniFun extends it with first-class functions, and MiniTyFun adds static type inference.
We also addressed parsing challenges and ambiguity resolution in MiniImp and MiniFun, ensuring correct syntax interpretation.
Additionally, we developed a MiniImp compiler that translates programs into MiniRISC, a simplified assembly language. This process includes CFG generation, register allocation, liveness analysis, and optimization for a target machine with a limited number of registers.
The report concludes with a guide explaining how to use the interpreters and the compiler.

MiniImp: A Simple Imperative Language

Overview

MiniImp is a simple imperative programming language created as an introduction to the principles of language interpreters. It includes basic constructs such as variables, arithmetic operations, and control flow.

Language Syntax

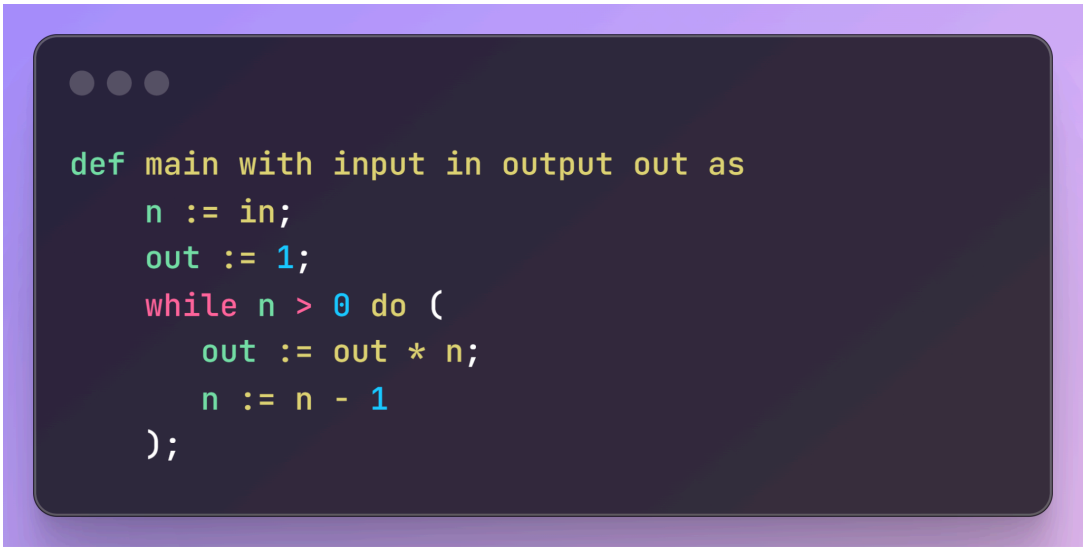
The syntax of MiniImp consists of the following key constructs:

- **Variables**: Identified by strings and can store integer values.
- **Arithmetic Expressions** (`aexp`):
  - `n`: A constant integer value.
  - `Var(x)`: The value of variable `x`.
  - Binary operations: `Plus(a1, a2)`, `Minus(a1, a2)`, `Times(a1, a2)` for addition, subtraction, and multiplication.
- **Boolean Expressions** (`bexp`):
  - `True` and `False`: Boolean constants.
  - Comparison: `Less(a1, a2)` for less-than comparisons.
  - Logical operations: `And(b1, b2)`, `Not(b1)` for logical conjunction and negation.

- **Commands** (`com`):
  - `Skip`: No operation.
  - `Assign(x, a)`: Assigns the result of evaluating the arithmetic expression `a` to the variable `x`.
  - `Seq(c1, c2)`: Sequentially executes commands `c1` and `c2`.
  - `If(b, c1, c2)`: Executes `c1` if `b` evaluates to `True`, otherwise executes `c2`.
  - `While(b, c)`: Repeatedly executes `c` while `b` evaluates to `True`.
  - `DefMain(in_vars, out_vars, c)`: Defines the main program with input variables `in_vars`, output variables `out_vars`, and the command `c` to execute.

Example Program

An example MiniImp program to compute the factorial of a number:



```
def main with input in output out as
    n := in;
    out := 1;
    while n > 0 do (
       out := out * n;
       n := n - 1
    );
```

Implementation of the MiniImp Interpreter

To interpret MiniImp, we developed a functional interpreter using **OCaml**. The approach involves parsing MiniImp programs into an abstract syntax tree (AST), which is then used to execute the commands based on the structure of the AST.

Key Components of the Implementation

Environment Representation:

An **environment** is used to store the mapping between **variable names** and their integer **values** during execution. It is implemented as an **association list**, where each entry is a pair `(variable_name, value)`.

We have implemented 2 helper functions:

- **Lookup**: The `lookup` function searches for a variable in the environment and returns its value, it raises an error if the variable is unbound (not defined).
- **Extend**: The `extend` function updates the environment by adding a new binding (variable and its value).

Evaluation Rules:

We developed 3 evaluation functions:

1. **Arithmetic Expression Evaluation** (`eval_aexp`): Recursively evaluates arithmetic expressions by substituting variable values from the environment and computing operations (i.e. plus, minus, times).
2. **Boolean Expression Evaluation** (`eval_bexp`): Recursively evaluates boolean expressions by interpreting their logical and comparison operators (i.e. and, not, less).
3. **Command Evaluation** (`eval_com`): Evaluates commands, like if statements, loops, and assignments, etc, by evaluating arithmetic and boolean expressions using the previous 2 functions, reading the values from the environment, handling control flow operations, and updating variable values in the environment.

Execution of the Main Program

The `DefMain` command specifies the entry point of the program. Input variables are initialized with provided values, and the output variables are extracted from the final environment after executing the command block.

Testing and Results

The code was tested with several programs to make sure it works as expected:

```
# Program 1
x := 5 ;
y := x + 2 ;
z := y * 3

# Program 2
if x < 10 then
    y := 2 ;
    z := y * 5
else
    y := 3 ;
    z := y * 10

# Program 3
x := 0 ;
while x < 3 do
  x := x + 1

# Program 4
def main with input in output out as
    x := in ;
    out := 0 ;
    while not x < 1 do (
       out := out + x ;
       x := x - 1
    );
```

Here are the results from the previous programs:

```
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab2_semantics> ocamlrun miniimp
MiniImp Main:
MiniImp Test Program 1:
x = 5
y = 7
z = 21

MiniImp Test Program 2:
x = 5
y = 2
z = 10

MiniImp Test Program 3:
x = 3

MiniImp Test Program 4:
in = 2
x = 0
out = 3
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab2_semantics>
```

MiniFun: Adding Functions

Overview

MiniFun extends **MiniImp** by adding support for higher-order functions. In this version, the language includes the ability to define functions, apply them to arguments, and use recursive functions.

Language Syntax

The syntax of MiniFun consists of the following key constructs:

- **Literals**: `IntLit` for integers and `BoolLit` for booleans.
- **Variables**: Represented by the `Var` constructor.
- **Operators**: Binary operations (`Add`, `Sub`, `Mul`, `Less`, `And`) and unary operations (`Not`).
- **Conditionals**: `If-else` expression.
- **Functions**:
  - Anonymous functions (`Fun`) with parameters and bodies.
  - Recursive functions (`Letfun`) with names, parameters, and bodies.
- **Let Bindings**: `Let` for local bindings.
- **Function Applications**: `App` for applying functions to arguments.

Example Program

An example MiniFun program to compute the factorial of a number and applied to 5:

```
letfun fact x =
  if x < 2 then
    1
  else
    x * fact (x - 1)
in fact 5
```

Key Components of the Implementation

Environment Representation:

Exactly the same as MiniImp, we used an **association list** to store the mapping between **variables or functions** and their **values.**

Value Types:

- `IntVal` and `BoolVal` for integer and boolean values.
- `Closure` for non-recursive functions, which includes a parameter, function body, and environment.
- `RecClosure` for recursive functions, which in addition, include the function name.

Evaluation Rules:

We developed one evaluation function that handles all cases:

- **Literals and Variables**
  - For integer and boolean literals it simply returns their values.
  - For variables it returns their values from the environment.
- **Operators**
  - Supports arithmetic operations (`Add`, `Sub`, `Mul`), comparisons (`Less`), and logical operations (`And`, `Not`).
  - Ensures type compatibility during evaluation, for example the `Add` operation can only be applied to integers.
- **Conditionals**
  - Evaluates the condition and selects the appropriate branch based on the boolean value. (then if the condition is true, else otherwise)
- **Functions and Closures**
  - Anonymous functions create normal closures using the fun's parameter, body, and environment.
  - Recursive functions create `RecClosure` objects that refer to themselves in the environment. (because they also save the fun name)
- **Let Bindings**
  - Evaluates the binding expression, updates the environment, and evaluates the body with the new environment.
- **Function Applications**
  - Applies closures by evaluating the argument and extending the closure's environment.
  - Recursive closures extend the environment with the function's name.

Testing and Results

The code was tested with several programs to make sure it works as expected:

```
# Program 1
let x = 10 in x + 5

# Program 2
letfun fact x = if x < 2 then 1 else x * fact (x - 1) in fact 5

# Program 3
let f = (let a = 1 in (fun y => y + a)) in (let a = 2 in f 4)
```

Here are the results from the previous programs:

```
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab2_semantics> ocamlrun minifun
MiniFun Main:
Program 1:      Result: 15
Program 2:      Result: 120
Program 3:      Result: 5
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab2_semantics>
```

MiniTyFun: Adding Types and Type Inference

Overview

MiniTyFun extends **MiniFun** by adding a type system, enabling static typing and type inference for the language. The language allows for explicit type annotations for variables and functions, and it includes a type-checking mechanism that ensures type safety.

Language Syntax

The main changes in the syntax of **MiniTyFun** include **type annotations**:

- Fun expression is modified to allow the user to explicitly define the parameter's type, (`Fun(x, typ_x, t)`),where `x` is the parameter, `typ_x` is the type of the parameter, and `t` is the fun's body.
- `Letfun` expression is also modified to handle explicit types (`Letfun (f, x, typ_f, t1, t2)`), where `f` is the fun name, `x` is the parameter, `typ_f` is the type of the fun, written as (type input -> type output), `t1` is the fun's body and `t2` is the scope.

Our system can handle 3 types:

```
and typ =
    | TInt
    | TBool
    | TFun of typ * typ   (* Function type τ -> τ' *)
```

Key Components of the Implementation

Environment Representation:

Same as the same previous solutions,, we used an **association list** to store the mapping between **variables** and their **types.**

Type System:

The type system introduces several rules for type checking:

- **Variable Binding**: When a variable is bound to a type in the environment, it can be used in expressions and checked for consistency, for example an int variable can't be used with the `Not` operator.
- **Let Expressions**: The environment is extended with the variable and its type, and the body of the expression is checked to make sure it's well-typed.
- **Function Types**: Functions are assigned types of the form `τ → τ'`, where `τ` is the type of the argument and `τ'` is the type of the return value. Functions are checked to ensure that the body matches the return type.

Type Inference:

The type inference process is based on a recursive evaluation of the expressions, using the `type_of` function, ensuring that all function applications and variable references are consistent with their declared types in the environment.

Testing and Results

The code was tested with several programs to make sure it works as expected:

```
# Program 1
let x = 10 in x + 5

# Program 2
letfun fact x:int = if x < 2 then 1 else x * fact (x - 1) in fact 5

# Program 3
let f = (let a = 1 in (fun y:int => y + a)) in (let a = 2 in f 4)

# Program 4
fun is_negative : (int -> bool) = if 1 < 2 then true else false
```

Here are the results:

```
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab3_typing> ocamlrun minityfun
MiniTyFun Main:
Program 1:      Program is well-typed: int      Result: 15
Program 2:      Program is well-typed: int      Result: 120
Program 3:      Program is well-typed: int      Result: 5
Program 4:      Program is well-typed: int -> bool
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab3_typing>
```

Scanning, Parsing, and Resolving Ambiguities in MiniImp

Scanning

We started by implementing the scanner (lexer) using **OCamllex**, which reads the source code and transforms it into a sequence of tokens. The tokens correspond to key components of the MiniImp language, such as variable identifiers, numeric constants, operators, etc. The scanner was designed to ignore whitespace and raise errors on unrecognized characters.

The scanner was designed to recognize:

- **Keywords**: such as `if`, `while`, `then`, `else`, etc.
- **Variables identifiers**: ex: x, y, z.
- **Numeric constants**: ex: 1, 2, 3.
- **Operators**: including arithmetic operators (`+`, `-`, `*`), comparison operators (`<`), and logical operators (`not`, `and`).
- **Punctuation**: such as parentheses `(`, `)`, semicolons `;`, and curly braces `{`, `}`.

Parsing

Next, we defined the parser using **Menhir**. The grammar for MiniImp captures the structure of programs, commands, boolean expressions, and arithmetic expressions. The rules are based on the syntax of MiniImp:

- **Programs**: Begin with the `def main` declaration.
- **Commands**: Include `skip`, assignments, sequential composition (`c1; c2`), conditionals (`if b then c1 else c2`), and loops (`while b do c`).
- **Boolean Expressions**: Include boolean values, logical operators (`and`, `not`), and comparisons (`a < a`).
- **Arithmetic Expressions**: Include variables, numbers, and operators (`+`, `-`, `*`).

Ambiguities

While defining the parser, we encountered several ambiguities due to the nature of **MiniImp**'s syntax. These ambiguities were resolved as follows:

Operator Precedence in Arithmetic Expressions:

Expressions like `a + b * c` could be interpreted incorrectly without precedence rules.
Solution: We explicitly defined the precedence of arithmetic operators, multiplication (`*`) has higher precedence than addition (`+`) and subtraction (`-`).

Parentheses Explicit Precedence:

Parentheses were used to override default precedence and associativity rules when needed. For example, in the expression `(4 + 5) * 2`, the parentheses clearly indicate that addition should occur first.
Solution:

- We added a rule for parentheses in arithmetic expressions (`LPAREN a = aexp RPAREN`).
- We also added a similar rule in commands (`LPAREN c = com RPAREN`)

Sequential Composition:

Without associativity rules, the parser could interpret the sequence of commands ambiguously. For example: `c1; c2; c3` could be parsed as `(c1; c2); c3` or `c1; (c2; c3)`.
Solution: We declared the `;` operator as **left-associative**, ensuring the parser interprets `c1; c2; c3` as `(c1; c2); c3`.

Boolean Expressions:

Boolean expressions like `not a and b` could lead to incorrect interpretations without precedence rules.

Solution: We added precedence and associativity rules to `Not` and `And`

Conditional Expressions:

In nested `if-then-else` statements, the parser could mistakenly associate an `else` branch with the wrong `if`. For example:

```
if true then if false then skip else skip else skip
```

This could be parsed as:

- `if true then (if false then skip else skip) else skip`
- `(if true then if false then skip else skip) else skip`

This introduces a shift-reduce conflict because the parser might encounter the else token and not know whether to finish the then block of the current if or to start a new if-else block.

Solution:

To resolve this ambiguity, we added a precedence declaration for the ELSE keyword

While Loop:

The body of a while loop could be misinterpreted if not clearly delimited, especially in sequences like: `while b do c1; c2`

This could lead to confusion about whether `c2` is part of the loop or not.

Based on the lab pdf, this command should be interpreted as: `(while b do c1); c2`

Solution: To resolve this, we made the DO keyword left-associative. `%left DO`

Note: If we want to have both c1 and c2 in the loop we can use parentheses, ex: `while b do (c1; c2)`

Scanning, Parsing, and Resolving Ambiguities in MiniFun

Scanning

Like in MiniImp, we implemented the scanner (lexer) for MiniFun using OCamllex, which processes the source code and produces a sequence of tokens corresponding to the fundamental components of the language. As with MiniImp, it is designed to ignore whitespace and raise errors for unrecognized characters.

The scanner recognizes:

- **Keywords:** such as `if`, `then`, `else`, `fun`, `let`, `in`, `letfun`, etc.
- **Identifiers:** for variables and function names, e.g., `x`, `y`, `compute`.
- **Constants:**
  - **Integer literals:** e.g., `0`, `42`, `100`.
  - **Boolean literals:** `true`, `false`.
- **Operators:**
  - **Arithmetic operators:** `+`, `-`, `*`.
  - **Comparison operators:** `<`.
  - **Logical operators:** `and`, `not`.
- **Punctuation:** including parentheses `(`, `)`, the semicolon `;`, and the arrow `->`.
- **Assignment and Equality:** `=`.
- **End of File (EOF):** Marks the end of input.

Parsing

For parsing MiniFun, we used Menhir, following the same approach outlined in MiniImp.

The grammar includes:

- **Programs:** Represented as a sequence of expressions followed by `EOF`.
- **Expressions:** The heart of MiniFun, comprising:
  - **Arithmetic Expressions:** Integer literals, variables, and arithmetic operations (`+`, `-`, `*`).
  - **Boolean Expressions:** Boolean literals, logical operations (`and`, `not`), and comparisons (`<`).

    - **Function Applications:** e.g., `f x, g 7`.
    - **Conditionals:** `if-then-else` expressions.
    - **Function Definitions:** Anonymous functions written as `fun x -> t`.
    - **Let Bindings:** Variable assignments using `let x = e1 in e2`.
    - **Recursive Function Definitions:** Recursive functions with `letfun f x = e1 in e2`.
- **Parenthesized Expressions:** Ensure grouping and precedence in expressions.

Ambiguities

The original grammar generated many conflicts, so we had to solve them using different solutions.

```
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab4_parsing\minifun> dune build
File "minifun_parser.mly", line 10, characters 21-25:
Warning: the token SEMI is unused.
Built an LR(0) automaton with 46 states.
Warning: the token SEMI is unused.
Built an LR(0) automaton with 46 states.
The construction mode is pager.
Built an LR(1) automaton with 46 states.
Warning: 11 states have shift/reduce conflicts.
Warning: 154 shift/reduce conflicts were arbitrarily resolved.
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab4_parsing\minifun>
```

Operator Precedence in Arithmetic Expressions:

Like in MiniImp, we needed to define operator precedence to avoid ambiguities in arithmetic expressions.

Solution: We used `%left` to declare `PLUS` and `MINUS` as left-associative with lower precedence, and `TIMES` with higher precedence.

Operator Precedence in Boolean Expressions:

Like in MiniImp too, we needed to resolve ambiguities in boolean expressions.

Solution: We declared `LESS` as left-associative with lower precedence, while `AND` is left-associative with higher precedence. The `NOT` operator was defined as non-associative.

Conditional Expressions:

Like in MiniImp, we needed to resolve ambiguities in nested if-then-else statements.

Solution: We declared `ELSE` as non-associative and gave it precedence to prevent incorrect associations in nested conditionals.

Function Application Ambiguity:

The parser would encounter ambiguity when parsing `FUN IDENT ARROW expr`, as it couldn't decide whether the `FUN` keyword indicated a function definition or if it was the start of a function application. For example the expression `fun x => x 1`, which could be parsed as either `(fun x => x) 1` which results in 1, or `fun x => (x 1)` which gives a higher-order function where x is a function applied to 1.

Solution:
We declared a new non-terminal for function application to distinguish function applications from general expressions. And we moved function definitions into a separate rule so that fun cannot be confused with an application.

Improved Grammar Modularity:
In general, to solve ambiguities, we splitted the grammar into multiple non-terminals, because the original grammar combined all expression types into a single non-terminal, leading to overlaps and ambiguities.
Solution: The grammar was modularized by introducing separate non-terminals for let and letfun expressions (`let_expr`), if expressions (`if_expr`), operation expressions (`op_expr`), function expressions (`fun_expr`), and function applications (`app_expr`).
This modular approach eliminated overlapping rules, reduced conflicts, and made the grammar easier to read and maintain.

Control Flow Graph (CFG) for MiniImp

Introduction

In this section, we describe the implementation of the Control Flow Graph (CFG) for MiniImp. The CFG represents the flow of control in a program by breaking it down into basic blocks and connecting them with edges, which is essential for analyzing and optimizing programs.

CFG Representation

The Control Flow Graph (CFG) in our solution is represented as:

- **Nodes (Basic Blocks):** Each node represents a basic block of code, which has a unique ID and the associated command that it represents.
- **Edges:** The edges represent the flow of control between blocks (nodes), indicating which basic block follows another, either directly (like in simple statements) or based on a conditional branch (like in if or loop).
- **CFG Structure:** Includes a list of nodes, edges, an entry node, and an exit node.

Implementation of Blocks: Minimal vs Maximal

For our solution, we chose to implement **maximal blocks,** which group multiple statements into a single block instead of creating a separate block for each statement.

In our implementation, we grouped consecutive simple statements (like skip and assignments) in sequences into a single block, which gives us a continuous code execution (without jumps or interruptions). For control-flow statements (like if and while) we created separate blocks for conditions and treated their bodies as distinct maximal blocks, and we linked them appropriately using control flow edges.

We used maximal blocks because they are considered the most common approach, and they reduce the number of nodes in the Control Flow Graph (CFG), making it more compact and efficient for future processing, such as code generation.

Generating the CFG for Sequences

For handling sequences of commands (c1; c2):

1. **Group Simple Statements:** Consecutive simple statements (skip, assign) are combined into one maximal block.
2. **Handle Remaining Commands:** If the sequence contains control-flow commands (if, while), they are treated separately and represented as distinct blocks.
3. **Connect Blocks:** The block of grouped simple statements is connected to the first block of the remaining commands using a control flow edge. In other words, we connect the last block of the first command c1 to the first block of the second command c2 using a control flow edge.

This approach ensures that sequences of commands are represented accurately in the CFG, maintaining the flow from one statement to the next.

Control Flow in If and While

Conditionals (if statements) and loops (while statements) introduce branching and require special handling in the CFG.

Conditionals (If-Then-Else):

1. The condition is represented as a **decision** block.
2. Edges connect the decision block to the entry blocks of the **then** and **else** branches.
3. Both branches eventually merge into a **final block**, which is just a skip statement.

Loops (While):

1. We have a **condition block** that checks the loop condition.
2. An edge from the **condition block** to the **loop body** (for true conditions).
3. An edge from the **loop body** back to the **condition block**.

4. An edge from the **condition block** to an **exit block**, which is just a skip statement (for false conditions).

Both statements have a similar principle, which is creating a decision block (representing the condition) and connecting it to the appropriate blocks based on the flow of control.

Note: For the decision blocks, we introduced a new command (**BQuestion**) to represent them in the control flow. For example, if the original code is: "`if x < 7`", the block will contain "`x < 7?`"

Example

To better understand how our solution works, let's see a real example of CFG generation of the following sample MiniImp code, which contains assignments, loop, condition check, and of course it calculates a meaningful output value.

```
def main with input in output out as
  x := in;
  out := 0;
  while not x < 0 do (
    out := out + x;
    x := x - 1
  )
```

```
MiniImp Control Flow Graph (CFG):
Entry: Node 0
Exit: Node 3

Nodes:
Node 0:
x := in
out := 0

Node 2:
not x < 0?

Node 1:
out := out + x
x := x - 1

Node 3:
skip

Edges:
Node 2 -> Node 1
Node 2 -> Node 3
Node 1 -> Node 2
Node 0 -> Node 2
```

Generating MiniRISC and its CFG

Introduction

This section explains the process of generating MiniRISC code and its Control Flow Graph (CFG). It introduces the MiniRISC module, details the translation of MiniImp CFG to MiniRISC CFG, and describes the final step of converting the MiniRISC CFG into MiniRISC code with appropriate control flow.

MiniRISC Module

The MiniRISC module provides a simplified RISC Assembly language designed to represent low-level instructions, which we'll use as a target language and also as an intermediate representation via its control-flow graph (CFG).

MiniRISC Syntax

MiniRISC defines a set of operations and instructions for data manipulation and control flow, like:

- **Arithmetic operations:** e.g., `add r1 r2 => r3, subi r1 5 => r2`
- **Memory operations:** e.g., `loadI 10 => r1, store r1 => r2`
- **Control flow:** e.g., `jump L1, cjump r1 L2 L3`

These instructions will be used for translating MiniImp programs to MiniRISC code.

MiniRISC CFG Representation

The MiniRISC control flow graph (CFG) structure is modeled as:

- **Blocks**: Where each block is identified by a label and contains a list of instructions.
- **Edges:** A list of control flow edges, defining transitions between blocks via their labels.
- **Entry**: The label of the entry block.
- **Exit**: The label of the exit block.

Translation from MiniImp CFG to MiniRISC CFG

The translation from MiniImp CFG to MiniRISC CFG involves mapping (translating) MiniImp commands into their corresponding MiniRISC instructions while maintaining the control flow structure represented by the CFG.

Overview of the Process

The MiniImp CFG consists of nodes representing basic blocks with commands, such as `skip`, `assign`, and `b?`. Each node is connected by edges that define the program's control flow. During the translation, we:

- Translate each MiniImp command in a node into one or more MiniRISC instructions.
- Preserve the control flow by associating the same edges with the new MiniRISC CFG.

Command Translation

- **Skip (`skip`):** Translated to `noop` (no operation).
- **Assignment (`x := aexp`):**
  - Compute the arithmetic expression (`aexp`) using MiniRISC arithmetic and immediate operations (ex: add, subi, etc.).
  - Store the result in the register corresponding to the variable x.
- **Condition (`b?`):**
  - Translate the boolean expression into MiniRISC instructions.
  - Store the result in a dedicated register to support conditional jumps later.

Optimizations

To improve the solution and generate less instructions:

- **Immediate Operations:** When an arithmetic or boolean expression involves a constant (e.g., x + 3), the translation uses immediate instructions like `addi` or `subi` instead of loading the constant into a register first and applying a normal instruction on registers.
- **Target Register:** When the result register is the same as the target register, avoid redundant copy instructions by storing the result directly into the target register.
- **Simplifications in Boolean Expressions:**
  - `x AND true` is simplified to x.
  - `x AND false` directly evaluates to false.

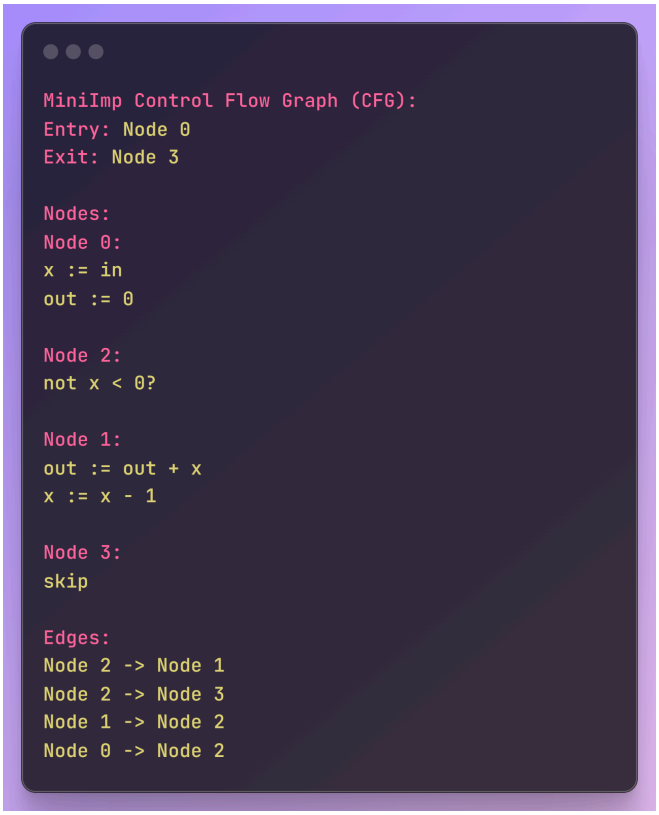
Register Management

In our solution, we tried to efficiently manage registers, including their allocation and access. For example, we specifically handled the **input** and **output** variables registers by reserving two registers for that purpose:

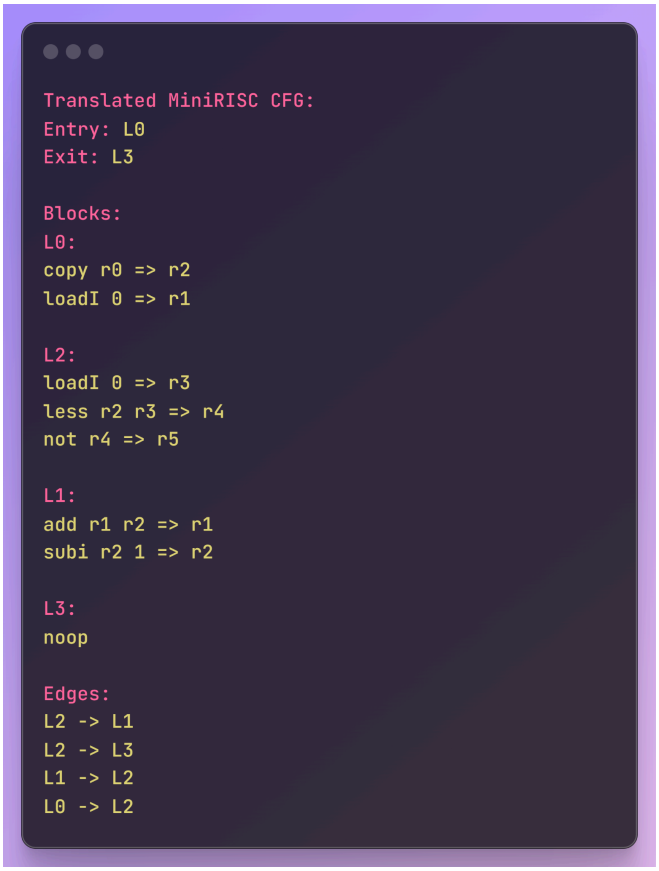
- **r_in (Register 0):** Dedicated to the input variable of the MiniImp program.
- **r_out (Register 1):** Dedicated to the output variable.

Example

Let's keep using the previous example, we already converted the MiniImp code to a CFG, now let's translate it to a MiniRISC CFG.



```
MiniImp Control Flow Graph (CFG):
Entry: Node 0
Exit: Node 3

Nodes:
Node 0:
x := in
out := 0

Node 2:
not x < 0?

Node 1:
out := out + x
x := x - 1

Node 3:
skip

Edges:
Node 2 -> Node 1
Node 2 -> Node 3
Node 1 -> Node 2
Node 0 -> Node 2
```



```
Translated MiniRISC CFG:
Entry: L0
Exit: L3

Blocks:
L0:
copy r0 => r2
loadI 0 => r1

L2:
loadI 0 => r3
less r2 r3 => r4
not r4 => r5

L1:
add r1 r2 => r1
subi r2 1 => r2

L3:
noop

Edges:
L2 -> L1
L2 -> L3
L1 -> L2
L0 -> L2
```

Translation from MiniRISC CFG to MiniRISC

The final step in the translation process is transforming the MiniRISC CFG into MiniRISC code. This process involves converting that control flow between blocks, which is represented as edges in the CFG, into jump instructions (either direct or conditional) within the MiniRISC blocks.

Overview of the Process

The MiniRISC CFG consists of labeled blocks, where each block contains a list of MiniRISC instructions, and edges that define the control flow between these blocks. During translation:

- Each block's original instructions are kept.
- Jump instructions (Jump or CJump) are added based on the outgoing edges from each block.

Translation Process

1. **Processing Blocks:** Each block in the MiniRISC CFG is processed individually. The original list of instructions is preserved, and jump instructions are appended at the end.
2. **Handling Outgoing Edges:**
    a. If a block has a **single outgoing edge**, an unconditional Jump instruction is added to jump to the next block's label.

b. If a block has **two outgoing edges**, it represents a condition. A CJump (conditional jump) is added.
    i. The **condition register** is determined based on the last instruction in the block, which computes the condition value (ex: less, and, or not).
    ii. The CJump instruction specifies the condition register and the two labels corresponding to the true and false branches.
c. If a block has **no outgoing edges** (e.g., the exit block), no jump is needed.

3. **Combining Instructions:** The original instructions in the block are combined with the generated jump instructions (added at the end) to form the final translated code.
4. **Preserving Entry and Exit:** The entry and exit labels from the CFG are kept to define the program's start and end points.

Example

Let's keep using the previous example, now we want to translate the MiniRISC CFG to MiniRISC code.

```
Translated MiniRISC CFG:
Entry: L0
Exit: L3

Blocks:
L0:
copy r0 => r2
loadI 0 => r1

L2:
loadI 0 => r3
less r2 r3 => r4
not r4 => r5

L1:
add r1 r2 => r1
subi r2 1 => r2

L3:
noop

Edges:
L2 -> L1
L2 -> L3
L1 -> L2
L0 -> L2
```

```
Translated MiniRISC Program:
Entry: L0
Exit: L3

Blocks:
L0:
copy r0 => r2
loadI 0 => r1
jump L2

L2:
loadI 0 => r3
less r2 r3 => r4
not r4 => r5
cjump r5 L1 L3

L1:
add r1 r2 => r1
subi r2 1 => r2
jump L2

L3:
noop
```

Dataflow Analysis - Defined Variables

Introduction

In this section, we describe the implementation of the Defined Variables Analysis for the MiniRISC CFG. The goal of this analysis is to make sure that no register is used before being initialized in the program.

Analysis Overview

The analysis works by computing the **in and out sets** for each block in the CFG:

- **in_set**: The set of registers defined at the **entry** of the block.
- **out_set**: The set of registers defined at the **exit** of the block.

The analysis propagates these sets through the CFG until a **fixpoint is reached** (i.e., no further changes occur). It then checks that no register is used before being defined.

Implementation Choices

Maximal Blocks

In the previous lab we chose maximal blocks as our main approach for the CFG, which reduces the number of blocks compared to minimal blocks approach, but it makes the analysis more challenging, since it requires careful handling of the in and out sets for each instruction within each block.

Fixpoint

In our implementation, we used the least fixpoint approach for the defined variables analysis.

- We start with the `in_set` of the entry block containing `r_in` (since it is always initialized).
- The `in_set` of all other blocks is initialized to the **empty set**.
- We iteratively compute the in and out sets for each block, adding registers as they are proven to be defined.

This approach ensures that we only include registers that are definitely defined.

Use-Before-Def Check

For each block, we check that no register is used before being defined. This is done by:

- Iterating through the instructions in the block.
- For each instruction, checking if all used registers are in the in_set.
- Updating the out_set with any newly defined registers.

Data Structures

Analysis State

Each block has an analysis_state record with an `in_set` and an `out_set`.

Hashtable

A hashtable maps block `labels` to their `analysis_state`. Which allows efficient lookup and update operations of the in and out sets.

Sets

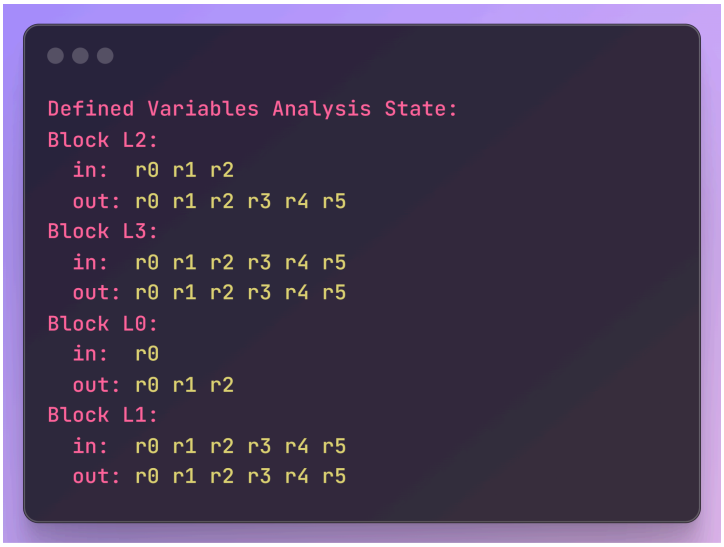
We use OCaml's Set module to represent sets of registers, which provides us with useful operations like union, intersection, and membership checks.

Algorithm

1. **Initialization:**
    a. The entry block starts with `r_in` defined.
    b. All other blocks start with empty in and out sets.
2. **Fixpoint Computation:**
    a. For each block, compute the `in_set` as the intersection of all predecessors' out sets.
    b. Compute the `out_set` by processing the instructions in the block.
    c. Repeat until no further changes occur (fixpoint reached).
3. **Use-Before-Def Check:**
    a. For each block, check that no register is used before being defined.

Example

Using the same previous example as always, we want to perform the defined variables analysis.



```
Defined Variables Analysis State:
Block L2:
  in:  r0 r1 r2
  out: r0 r1 r2 r3 r4 r5
Block L3:
  in:  r0 r1 r2 r3 r4 r5
  out: r0 r1 r2 r3 r4 r5
Block L0:
  in:  r0
  out: r0 r1 r2
Block L1:
  in:  r0 r1 r2 r3 r4 r5
  out: r0 r1 r2 r3 r4 r5
```

But just to give a **counterexample**, let's take the following MiniImp program, which contains a variable "y" which is used before being declared, the compiler should detect this as an error.

```
def main with input in output out as
  x := in;
  out := y;
  while not x < 0 do (
    out := out + x;
    x := x - 1
  )
```

```
L3:
noop

Edges:
L2 -> L1
L2 -> L3
L1 -> L2
L0 -> L2
Fatal error: exception Failure("Register r3 used before definition in block L0")
PS E:\Quick Access\Documents\School\M1 UNIPI CS\S1\Languages, Compilers And Interpreters\Lab\lab7_dataflow\miniimp>
```

Dataflow Analysis - Liveness

Introduction

In this section, we describe the implementation of the Liveness Analysis for the MiniRISC CFG. The goal of this analysis is to determine which registers are live at each point in the program. A register is live if its current value may be used in the future before being overwritten.
This analysis allows us to optimize register usage and avoid unnecessary allocations.

Implementation Choices

Maximal Blocks

Since we're using **maximal blocks** in our project, this makes liveness analysis more complex because registers can be used and defined multiple times within a single block. To handle this correctly:

- We process each block's instructions in **reverse order**, to make sure that we account for later uses before definitions within the block.
- We track registers **used** and registers **defined** in the block separately.
- For each instruction, we update the **in and out sets** dynamically as we process the block.

Other Implementation Choices

The other implementation choices are the same as the defined variables analysis, since we used the same data structures, fixpoint approach, etc.

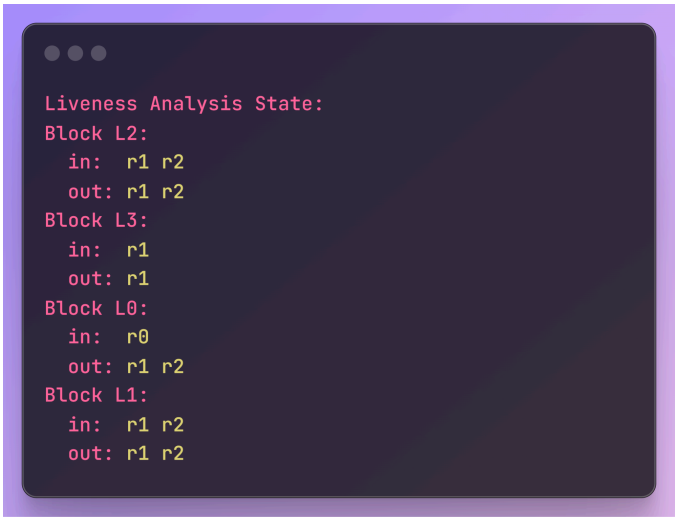
Algorithm

1. **Initialization:**
    a. The exit (final) block starts with the output register as live.
    b. All other blocks start with empty in and out sets.
2. **Fixpoint Computation:**
    a. For each block, compute the **out set** as the union of the **in sets** of its successors.
    b. Compute the **in set** by processing the instructions in the block backward, and applying the formula: **{r used in L} ∪ (lvout(L) \ {r defined in L}).**
    c. Repeat until no further changes occur (fixpoint reached).

Example

Using our previous example, we want to perform the liveness analysis.

PS: This, and all the previous screenshots, are real logs from my VS Code terminal, I just presented them like this instead of direct screenshots from the terminal for report design and easy readability purposes.



Register Allocation for a Target Architecture

Introduction

This section describes the process of translating a MiniRISC CFG into a MiniRISC program for a target architecture where the number of available registers (n) is limited and provided as a parameter (n ≥ 4). The goal is to reduce register usage by spilling registers to memory while ensuring correctness and efficiency in execution.

Implementation Choices

Register Allocation Strategy

We followed a **priority-based allocation strategy** where registers are classified into two categories:

- **Kept registers:** Registers that stay in physical registers.
- **Spilled registers:** Registers that are mapped to memory and need explicit load/store instructions.

To choose which registers to keep, we employed a heuristic based on registers usage frequency:

- We count the number of times each register is used in the MiniRISC CFG.
- The most frequently used registers are **prioritized** for keeping in registers.
- At least two registers (**rA**, **rB**) are reserved for temporary values to handle spills.

Memory Addressing for Spilled Registers

We maintain a mapping (spill table) from spilled registers to memory addresses, in other words, for each register that is spilled, we have an associated memory address that is used in the load and store instructions. The temporary registers (rA or rB) are used to handle the data.

Register Allocation Algorithm

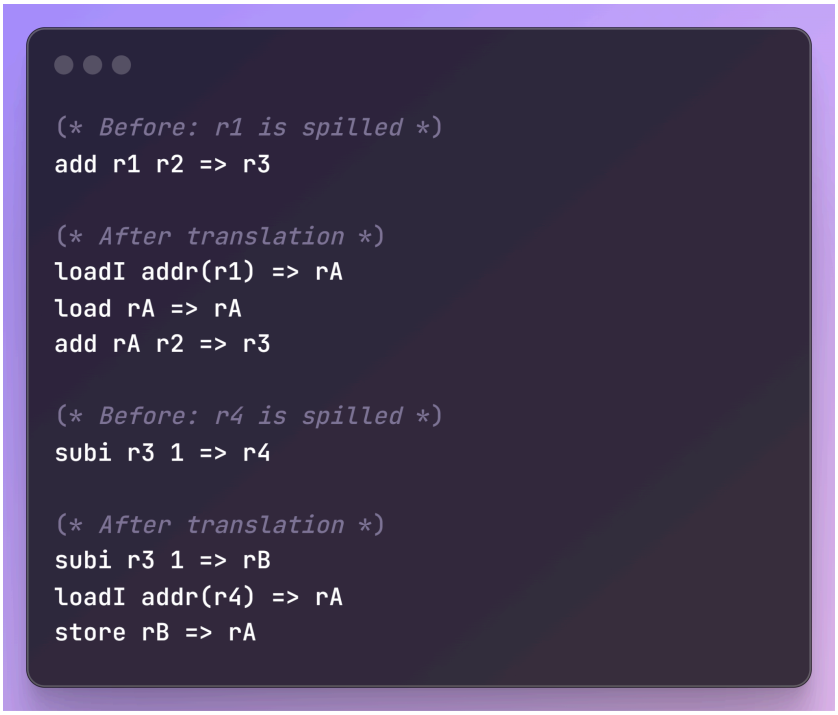
- **Step 1: Compute Register Usage Frequency**
    - We iterate over all blocks and their instructions in the MiniRISC CFG.
    - We collect the usage frequency of each register and store it in a frequency table.
- **Step 2: Select Registers to Keep**
    - We sort registers by frequency (most used first).
    - We select n - 2 registers to keep, ensuring that the special (reserved) registers are always kept.
    - The remaining registers are spilled, meaning they are assigned memory addresses and added to the spill table.

- **Step 3: Translate Instructions**
    - For each instruction, we adjust it based on whether its registers are in the kept set or the spill table:

Handling Spilled Registers (Translation)

- If an operation needs to read a spilled register, we load its address from the spill table into a temporary register (rA or rB), then we read its value from the memory using the load instruction.
- If an operation writes to a spilled register, we load its address from the spill table into a temporary register (rA or rB), then we store the result value (stored in a different register) in memory using the store instruction.

Examples of Translated Instructions:



```
(* Before: r1 is spilled *)
add r1 r2 => r3

(* After translation *)
loadI addr(r1) => rA
load rA => rA
add rA r2 => r3

(* Before: r4 is spilled *)
subi r3 1 => r4

(* After translation *)
subi r3 1 => rB
loadI addr(r4) => rA
store rB => rA
```

Code Overview

Our solution is structured as follows:

- **Register Usage Analysis (count_register_usage)**
    - Counts how frequently each register is used.
    - Helps us decide which registers to keep in registers and which to spill.
- **Register Allocation (allocate_registers)**
    - Selects the n - 2 most frequently used registers to keep.
    - Assigns memory addresses to spilled registers.

    - Creates a spill table to track memory mappings.
- **Instruction Translation (translate_instruction)**
    - Replaces register operations with load/store instructions if a register is spilled.
    - Uses temporary registers (rA, rB) to hold values during computations.
    - If an instruction doesn't contain spilled registers, it keeps it as it is.
- **Applying Allocation to the CFG (apply_register_allocation)**
    - Processes all blocks in the MiniRISC CFG.
    - Replaces instructions with their translated versions.

Register Optimization via Liveness Analysis

Introduction

In this phase of the project, we implemented an optimization procedure that reduces the number of registers used in MiniRISC CFG by exploiting liveness analysis and merging registers that are not live at the same time (their live ranges do not overlap).

Computing Live Ranges

The first step in the optimization process is to compute the live ranges for each register, which we can get from the liveness analysis that we already implemented.
For each register, the live range is defined as the set of control-flow edges where the register is live. A register is live in the edge (**block 1**⟶ **block 2**) if it is live when entering block 2.
We implemented the `compute_live_ranges` function, which takes a MiniRISC CFG and its liveness analysis as input, and returns a mapping from registers to their live ranges. This is achieved through the following approach:

1. Create a hashtable to store live ranges for each register.
2. For each edge (`l_from, l_to`) in the CFG:
    a. Retrieve the `in_set` of the successor block (`l_to`).
    b. For each register in the `in_set`, add the edge `l_from->l_to` to its live range.
3. Return the computed live ranges.

Merging Registers

The next step is to merge registers whose live ranges do not overlap. This reduces the number of registers required by reusing registers for multiple purposes without causing conflicts.

We implemented the `merge_registers` function, which takes the live ranges as input and returns a mapping from original registers to merged registers. This is achieved through the following approach:

1. Create a hashtable (`reg_mapping`) to store the mapping from original registers to merged registers.
2. Maintain a list of groups, where each group is a tuple `(merged_reg, group_range)` representing a merged register and its combined live range.
3. Check for overlapping live ranges:
    a. For each register, try to find an existing group whose live range does not overlap with the current register's live range. If a suitable group is found, we add the current register to that group and update the group's live range.
    b. If no suitable group is found, we create a new group for the current register and assign it a new merged register.
    c. **Note:** Special registers (r_in, r_out) are not merged, because they have specific roles in the program.

Optimizing the CFG

The final step is to apply the register merging optimization to the MiniRISC CFG. This involves updating all instructions to use the merged registers, if there are any of course.
We implemented the `optimize_registers` function, which takes a MiniRISC CFG and its liveness analysis state as input and produces an optimized CFG. This is achieved through the following approach:

1. Use the `compute_live_ranges` function to compute live ranges for all registers.
2. Use the `merge_registers` function to compute the register merging mapping.
3. For each instruction in the CFG, replace original registers with their merged registers using the `reg_mapping`.
4. The updated CFG is returned with merged registers.

How to Use the Program

Introduction

In this section, we explain how to run the different parts of the project.

Prerequisites

Before using the program, ensure the following:

- **OCaml and Dune:** The program is written in OCaml and uses Dune for building. Make sure OCaml and Dune are installed on your system.
- **MiniImp Program:** Prepare a MiniImp program file (e.g., program.miniimp) that you want to compile.
- **Setting Up the Environment:** If you are using VS Code or a terminal, you need to set up the OCaml environment using **opam env**. This step ensures that the necessary tools and libraries are available for building and running the program.

```
// For PowerShell (ex: VS Code Terminal), run the following command in your terminal:
(& opam env) -split '\r?\n' | ForEach-Object { Invoke-Expression $_ }
// For Command Prompt (ex: CMD), run the following command in your terminal:
for /f "tokens=*" %i in ('opam env') do @%i
```

MiniImp Interpreter

Our MiniImp interpreter takes a MiniImp program as input, plus an integer value as input to the program, and compiles the program and outputs the result.

1. Navigate to the MiniImp interpreter directory “miniimp_interpreter” inside the project.
2. Build the program using dune. This will compile the OCaml code and generate an executable.
3. Run the program by typing the run command.

```
cd miniimp_interpreter
dune build
dune exec ./main.exe <program.miniimp>
dune exec ./main.exe ./test/program.miniimp
```

MiniFun Interpreter

Our MiniFun interpreter takes a MiniFun program as input, and compiles it and outputs the result.

1. Navigate to the MiniFun interpreter directory “minifun_interpreter” inside the project.
2. Build the program using dune. This will compile the OCaml code and generate an executable.
3. Run the program by typing the run command.

```
cd minifun_interpreter
dune build
dune exec ./main.exe <program.minifun>
dune exec ./main.exe ./test/prgm.minifun
```

MiniFun Interpreter

For MiniTyFun, we didn’t implement a parser, since we already implemented a parser for MiniFun.

1. Navigate to the MiniTyFun interpreter directory “minityfun_interpreter” inside the project.
2. Build the program using ocamlc. This will compile the OCaml code and generate an executable.
3. Run the program by typing the run command using ocamlrun.

```
cd minityfun_interpreter
ocamlc -o minityfun minityfun.ml
ocamlrun minityfun
```

MiniImp Compiler

Our MiniImp compiler takes a **MiniImp** program as input and produces an optimized **MiniRISC program** as output. The compiler allows us to configure several options, including the number of registers available in the target machine, whether to check for undefined variables, and whether to apply optimizations.

1. Navigate to the MiniImp compiler directory “compiler_compiler” inside the project.

2. Build the program using dune. This will compile the OCaml code and generate an executable.
3. Run the program by typing the run command and specifying the arguments correctly:
    a. **<num_registers>:** The number of registers available in the target machine. Must be an integer greater than or equal to 4.
    b. **<check_undefined_vars>:** A boolean flag (true or false) to enable or disable the undefined variable check. If enabled, the compiler will check for undefined variables and fail if any are found.
    c. **<enable_optimization>:** A boolean flag (true or false) to enable or disable optimization. If enabled, the compiler will perform register merging and other optimizations.
    d. **<program.miniimp>:** The path to the MiniImp program file you want to compile.
    e. **<output_path>:** The path where the compiled MiniRISC code will be written.

```
cd miniimp_compiler
dune build
dune exec ./main.exe <num_registers> <check_undefined_vars> <enable_optimization>
<program.miniimp> <output_path>
dune exec ./main.exe 5 true true ./test/merge.miniimp ./compiled/merge.minirisc
```

For more details, please check out the **project repository**.