Languages, Compilers and Interpreters

MiniImp, MiniFun, and MiniTyFun

Francesco Scarfato

February 1, 2026

Contents

Chapter 1

Introduction

This project implements four components: a compiler for MiniImp (a simple imperative language), an interpreter for MiniImp, an interpreter for MiniFun (a functional language), and a type checker for MiniTyFun (a statically-typed functional language).

1.1 Project overview

The project is implemented in OCaml using the following tools:

- **Dune**: Build system for OCaml projects, managing compilation, dependencies, and testing
- **Menhir**: LR(1) parser generator that transforms grammar specifications into OCaml parser code
- **OCamllex**: Lexical analyzer generator that tokenizes source code
- **OCaml Standard Library**: Provides functional data structures (Map, Set) used throughout the implementation

The entire codebase follows a pure functional style, avoiding mutable state. This design choice makes the code easier to reason about, test, and verify for correctness.

1.1.1 Code Structure

The implementation is organized into modular components under the `lib/` directory:

- **miniImp/**: MiniImp language implementation
  - `MiniImpSyntax.ml`: Abstract syntax tree definitions for the imperative language
  - `MiniImpLexer.mll`: Lexical analyzer (tokenizer) specification
  - `MiniImpParser.mly`: Grammar specification for parsing
  - `MiniImpEval.ml`: Interpreter implementation using operational semantics
  - `MiniImpCFG.ml`: Control Flow Graph construction for the compiler
- **miniFun/**: MiniFun language implementation
  - `MiniFunSyntax.ml`: AST definitions for the functional language
  - `MiniFunLexer.mll`: Lexical analyzer for functional syntax
  - `MiniFunParser.mly`: Grammar specification with operator precedence
  - `MiniFunEval.ml`: Environment-based interpreter with closures
  - `MiniTyFunSyntax.ml`: Extended AST with type annotations
  - `MiniTyFunTypeCheck.ml`: Static type checker implementation
- **miniRISC/**: Target assembly language and compiler backend
  - `MiniRISCSyntax.ml`: RISC instruction set definitions

  - `MiniRISCUtils.ml`: Register analysis utilities (used/defined registers)
  - `MiniRISCCFG.ml`: Control flow graph data structures and utilities
  - `MiniRISCTranslation.ml`: MiniImp to MiniRISC translation phase
  - `MiniRISCDataflow.ml`: Dataflow analyses (definite variables, live variables)
  - `MiniRISCAllocation.ml`: Register allocation with coalescing and spilling
  - `MiniRISCLinearize.ml`: CFG to linear instruction stream conversion

The `bin/` directory contains executable entry points (`MiniImpCompiler.ml`, `MiniImpInterpreter.ml`, `MiniFunInterpreter.`
that orchestrate these modules into complete tools.

1.1.2 MiniImp: An Imperative Language

MiniImp is a minimal imperative language with variables, arithmetic expressions, boolean expressions, conditionals (`if-then-else`), and loops (`while`). Programs follow a simple structure:

```
def input_var output output_var as
  commands
```

Every MiniImp program takes one input parameter and produces one output value. For example, this factorial program demonstrates assignments, loops, and arithmetic operations:

```
def n output result as
  result = 1;
  while (n > 0) do
    result = result * n;
    n = n - 1
```

For MiniImp, both a complete compiler and an interpreter were built. The **compiler** translates MiniImp to MiniRISC assembly through multiple phases:

- lexing/parsing (OCamllex/Menhir),
- Control Flow Graph construction with maximal basic blocks,
- translation to register-based assembly with unlimited virtual registers,
- dataflow analysis (definite variables and live variables),
- register coalescing (optional optimization using instruction-level liveness),
- register allocation (frequency-based heuristics),
- spilling (load/store generation for memory),
- final code generation.

The compiler performs algebraic simplifications during translation.

The **interpreter** provides a implementation that directly evaluates the AST using operational semantics, maintaining an environment mapping variables to values.

1.1.3 MiniFun: A Functional Language

MiniFun is a functional language based on the lambda calculus. It features anonymous functions (`fun`), function application, local bindings (`let`), recursive functions (`letfun`), conditionals, and arithmetic/boolean operations. Functions are first-class values and support closures (capturing their lexical environment). The language uses currying: multi-argument functions are represented as nested single-argument functions.

```
letfun factorial n =
  if n < 2 then 1
  else n * factorial (n - 1)
in factorial 5
```

MiniFun is dynamically typed—type errors are detected at runtime during evaluation. An environment-based interpreter was built for MiniFun. It includes:

- lexing/parsing to build the AST,

- a recursive evaluator using environments to map variables to values,
- closure implementation (functions capture their defining environment for lexical scoping),
- recursion handling through a special closure variant that binds the function name in its own body.

1.1.4 MiniTyFun: A Statically-Typed Functional Language

MiniTyFun extends MiniFun with static type checking. Function parameters and recursive functions require explicit type annotations. The type system includes three types: `Int`, `Bool`, and `Closure(`τ_1`, `τ_2`)` for functions from type τ_1 to τ_2. Programs are type-checked before execution, and well-typed programs are guaranteed not to have runtime type errors.

```
letfun factorial (n : Int) =
  if n < 2 then 1
  else n * factorial (n - 1)
in factorial 5
```

A static type checker was built for MiniTyFun. It includes:

- syntax extensions to handle type annotations on function parameters and recursive functions,
- a recursive type checker that verifies program correctness using typing rules,
- a type environment (separate from value environment) that maps variables to their types,

1.2 Building and Running the Project

To build all components of the project:

```
dune build
```

This compiles all three language implementations (MiniImp compiler, MiniFun interpreter, and MiniTyFun type checker) along with their dependencies. The compiled executables are placed in the `_build/default/bin/` directory.

The MiniImp compiler translates `.minimp` source files to `.risc` assembly files:

```
dune exec MiniImpCompiler -- [OPTIONS] <num_registers> <input.minimp> <output.risc>
```

Required Arguments:

- `num_registers`: Number of physical registers available (minimum 4)
- `input.minimp`: Path to the MiniImp source file
- `output.risc`: Path for the generated MiniRISC assembly output

Optional Flags:

- `-O`: Enable optimizations (register coalescing)
- `-s`: Enable safety checking (detect uninitialized variables)
- `-v`: Verbose output (show intermediate compilation stages)

Example:

```
dune exec MiniImpCompiler -- -O -s 8 examples/factorial.minimp output.risc
```

This compiles `factorial.minimp` with 8 physical registers, optimizations enabled, and safety checking on.

1.2.1 Running the MiniImp Interpreter

The MiniImp interpreter directly evaluates `.minimp` source files without compilation:

```
dune exec MiniImpInterpreter <input.minimp>
```

Required Arguments:

- `input.minimp`: Path to the MiniImp source file

Example:

```
dune exec MiniImpInterpreter examples/simple.minimp
```

The interpreter parses the program and evaluates it directly using the operational semantics, without generating assembly code. This is useful for testing program correctness and comparing against compiled output.

1.2.2 Running the MiniFun Interpreter

The MiniFun interpreter evaluates `.minifun` source files:

```
dune exec MiniFunInterpreter <input.minifun>
```

Required Arguments:

- `input.minifun`: Path to the MiniFun source file

Example:

```
dune exec MiniFunInterpreter examples/factorial.minifun
```

The interpreter parses the program, evaluates it using environment-based semantics, and prints the result.

Chapter 2

MiniTyFun Type System

MiniTyFun extends MiniFun with static type checking. The key difference from MiniFun is the addition of explicit type annotations on function parameters and recursive function definitions.

2.1 Syntax Extensions

The syntax of MiniTyFun extends MiniFun in two ways:

Listing 2.1: MiniTyFun Syntax Extensions

```
(* MiniFun: no types *)
Fun of string * term

(* MiniTyFun: parameter has explicit type *)
Fun of string * typ * term
  (* fun (x : Int) -> x + 1 *)

(* MiniFun: recursive function *)
LetFun of string * string * term * term

(* MiniTyFun: recursive function with type annotation *)
LetFun of string * string * typ * term * term
  (* letfun f (x : Int -> Int) = ... in ... *)
```

The type language includes three constructors:

- `Int`: Type of integers
- `Bool`: Type of booleans
- `Closure(`τ_1`, `τ_2`)`: Type of functions from τ_1 to τ_2

The type system provides a strong guarantee:

If $\emptyset \vdash e : \tau$, then either:

- e evaluates to a value of type τ, or
- e diverges (infinite loop), or
- e fails with a non-type error (e.g., unbound variable)

But e will **never** have a runtime type error (adding Int to Bool, calling a non-function, etc.).

Chapter 3

Parser Design and Ambiguity Resolution

Parsing transforms a flat stream of tokens into a structured Abstract Syntax Tree (AST). The main challenge is handling ambiguity—multiple possible parse trees for the same input. We use Menhir (an LR(1) parser generator) with explicit precedence and associativity declarations.

3.1 MiniImp Parser: Ambiguities and Solutions

3.1.1 The Dangling Semicolon Problem

Consider this code:

```
if x > 0 then y := 1 else y := 2; z := 3
```

This could be parsed two ways:

1. `(if x > 0 then y := 1 else y := 2); z := 3` — semicolon sequences the if with z assignment
2. `if x > 0 then y := 1 else (y := 2; z := 3)` — semicolon is inside the else branch

Solution: The grammar rule for `command_list` makes sequencing explicit:

```
command_list:
  | c1 = command; SEMICOLON; c2 = command_list  { Seq(c1, c2) }
  | c = command; SEMICOLON                      { c }
  | c = command                                 { c }
```

And critically, `if` and `while` accept a single `command`, not a `command_list`:

```
command:
  | IF; cond = b_expr; THEN; p1 = command;
    ELSE; p2 = command                          { If(cond, p1, p2) }
```

This means `if...then...else` captures exactly *one* command per branch. To include a sequence, you must use parentheses:

```
if x > 0 then (y := 1; z := 2) else y := 3
```

3.1.2 Operator Precedence

Consider `a + b * c`:

1. `a + (b * c)` — multiplication first (correct)
2. `(a + b) * c` — left-to-right (wrong)

Our Solution: Precedence declarations:

```
%left PLUS MINUS
%left TIMES
```

Lower in the file = higher precedence. So `TIMES` binds tighter than `PLUS/MINUS`.

The `%left` means left-associative, so `a - b + c` becomes `(a - b) + c`.

The alternative is encoding precedence in the grammar itself with multiple levels of nonterminals (expr → term → factor → atom). That works but makes the grammar confusing. Precedence declarations are clearer and match standard mathematical conventions directly.

3.1.3 Negative Number Ambiguity

Consider `x - 5`. Is the minus:

1. A binary operator (subtraction)
2. Part of a negative literal (-5)

Solution: The grammar has a special rule for integer literals:

```
int:
  | p = INT                 { p }
  | MINUS; p = INT          { -p }
```

And `MINUS` is also a binary operator in `a_expr`:

```
a_expr:
  | p1 = a_expr; MINUS; p2 = a_expr    { Minus(p1, p2) }
```

The parser resolves this by context: `MINUS INT` creates a negative literal, while `expr MINUS expr` creates a subtraction node.

3.2 MiniFun Parser: Ambiguities and Solutions

3.2.1 Let Body Precedence

Consider:

```
let x = 1 in x + 2 * 3
```

Where does the `let` body end? Is it:

1. `let x = 1 in (x + 2 * 3)` — body is entire expression (correct)
2. `(let x = 1 in x + 2) * 3` — body ends early (wrong)

The same issue appears with `fun` (where does the arrow's body end?) and `if...else` (where does the else branch end?).

Solution: Precedence declarations give `IN`, `ARROW`, and `ELSE` the *lowest* precedence:

```
%left IN ELSE ARROW
%left AND NOT
%left LESS
%left PLUS MINUS
%left TIMES
```

This means they bind weakest—their bodies extend as far right as possible.

So `let x = 1 in x + 2 * 3` parses as `let x = 1 in (x + (2 * 3))` because `TIMES` and `PLUS` bind tighter than `IN`.

3.2.2 Function Application Precedence

Consider:

```
f g + h
```

Is this:

1. `(f g) + h` — apply f to g, then add h (correct)
2. `f (g + h)` — compute g+h, then apply f (wrong)

Solution: We use a three-level grammar:

```
expr:        (* operators, if, let, fun *)
  | ...
  | t = fun_app { t }

fun_app:     (* function application *)
  | t1 = fun_app; t2 = atomic   { FunApp(t1, t2) }
  | t = atomic                  { t }

atomic:      (* literals, variables, parentheses *)
  | n = INT                     { IntLit n }
  | ...
```

Function application is in its own middle layer. Since `expr` references `fun_app`, which references `atomic`, applications bind tighter than any operator.

3.2.3 Application Associativity

Consider:

```
f g h
```

In curried function application, is this:

1. `(f g) h` — left-associative (correct)
2. `f (g h)` — right-associative (wrong)

Solution: The `fun_app` rule is left-recursive:

```
fun_app:
  | t1 = fun_app; t2 = atomic   { FunApp(t1, t2) }
```

Left recursion in a left-to-right parser (like LR(1)) produces left-associative trees.

We choose left-associative application because it's standard in functional languages (ML, Haskell, OCaml). It matches programmer expectations.

3.2.4 The Dangling Else (Same as MiniImp)

Consider:

```
if c1 then if c2 then t1 else t2
```

Which `if` does the `else` belong to?

Solution: The `%left ELSE` declaration means "in a conflict, prefer shifting (closing the inner if)." So this parses as:

```
if c1 then (if c2 then t1 else t2)
```

The `else` matches the nearest `if`.

Why this behavior? It's the standard convention in most programming languages (C, Java, Python, etc.). Programmers expect `else` to match the nearest `if`.

Chapter 4

Control Flow Graph Construction

The Control Flow Graph (CFG) is the foundation for all analysis and optimization. It represents program structure as a graph where nodes are basic blocks (sequences of straight-line code) and edges are possible control flow transitions.

4.1 Minimal or maximal blocks

Given a MiniImp AST with nested `Seq` nodes, we need to build a CFG. Two extreme approaches:

Minimal Blocks: One command per block.

- Pro: Simple, explicit, predictable
- Con: Many tiny blocks, less efficient for analysis, multiple jump instructions

Maximal Blocks: Longest possible sequences without branches.

- Pro: Fewer blocks, more efficient
- Con: More complex

4.2 Building Maximal Blocks Directly

We rejected a naive "one block per statement, then merge" approach. Instead, we build optimal blocks directly using a two-step process:

Step 1: Flatten Sequences

The parser produces deeply nested `Seq` trees:

`Seq(s1, Seq(s2, Seq(s3, Seq(s4, ...))))`

This is natural for recursive parsing but not efficient for iteration. We flatten it to a list:

`[s1; s2; s3; s4; ...]`

The flattening code is simple:

```
let rec flatten_cmd cmd =
  match cmd with
  | Seq (c1, c2) -> flatten_cmd c1 @ flatten_cmd c2
  | _ -> [cmd]
```

This gives us a linear list of statements to process.

Step 2: Accumulate Into Blocks

Now we process the flat list, accumulating straight-line statements into the current block. When we hit a branch point (if, while), we:

1. Close the current block
2. Create blocks for the branch paths

3. Recursively process sub-commands
4. Create a join block where paths merge

Every CFG has dedicated entry and exit blocks, even if they only contain `Skip`. This provides clear starting/ending points for dataflow analysis and ensures uniformity across all CFGs.

Chapter 5

Translation from MiniImp to MiniRISC

The translation phase converts high-level MiniImp (with variables and expressions) to low-level MiniRISC (with registers and explicit instructions). This is the ”compiler” in the traditional sense.

We use a strict separation:

1. **Expressions** always compute into a FRESH temporary register
2. **Assignments** explicitly copy the result to the target variable

This creates extra `copy` instructions, but it’s simple and avoids a critical bug.

5.1 Variable Mapping Strategy

We maintain a map: `variable_name → register_name`

- `"x"` → `r5`
- `"counter"` → `r12`

Special Cases:

- Input variable **always** maps to `r_in`
- Output variable **always** maps to `r_out`

In the first phase, we generate code with *unlimited* virtual registers (r0, r1, r2, ... r42, etc.). The second phase (register allocation) maps these virtual registers to a fixed number of physical registers.

5.2 Algebraic Simplification During Translation

We perform optimizations while we can still see the AST structure:

Pattern	**Optimization**	**Why**
`x + 0`	No code generated	Identity operation
`0 + x`	`copy x => r`	Identity operation
`x * 0`	`loadi 0 => r`	Constant result
`x * 1`	`copy x => r`	Identity operation
`1 * x`	`copy x => r`	Identity operation

In this way, it’s easier to recognize `x * 0` in the AST than after generating:

```
loadi 0 => r1
mult r2 r1 => r3
```

5.3 Translation of Control Flow

5.3.1 Conditionals

An `if` statement translates to:

1. Evaluate condition into a register
2. Generate `branch` instruction with true/false labels
3. Translate both branches
4. Both branches jump to a join label

Example:

```
if (x > 0) then y := 1 else y := 2
```

Translates to:

```
loadi 0 => r1
less r1 r_x => r_cond
branch r_cond => L_then, L_else

L_then:
  loadi 1 => r_temp
  copy r_temp => r_y
  jump => L_join

L_else:
  loadi 2 => r_temp
  copy r_temp => r_y
  jump => L_join

L_join:
  ...
```

5.3.2 Loops

A `while` loop translates to:

1. Header label (loop entry point)
2. Evaluate condition
3. Branch: true → body, false → exit
4. Translate body
5. Jump back to header
6. Exit label (after loop)

Example:

```
while (x > 0) do x := x - 1
```

Translates to:

```
L_header:
  loadi 0 => r1
  less r1 r_x => r_cond
  branch r_cond => L_body, L_exit

L_body:
  loadi 1 => r2
  sub r_x r2 => r_temp
  copy r_temp => r_x
  jump => L_header
```

```
L_exit:
  ...
```

Notice the body jumps back to the header, creating the loop.

Chapter 6

Dataflow Analysis

Dataflow analysis is how compilers understand program behavior without executing it. We analyze the control flow graph to compute properties like "which variables are definitely defined?" and "which variables might be used later?"

The dataflow analyses follow the same fixpoint algorithm pattern:

1. **Initialize** all blocks with a starting state (TOP or BOTTOM)
2. **Iterate:** For each block, compute new IN/OUT based on neighbors
3. **Check:** If anything changed, repeat step 2
4. **Done:** When nothing changes, we've reached a "fixpoint"

This always terminates because we work with finite sets (registers), updates are monotonic (sets only grow or shrink, never oscillate), and there's a maximum size we can't exceed.

6.1 Definite Variables Analysis (Forward, Must)

The define variable analysis detects uninitialized register usage—a safety check that prevents using variables before they're defined.

6.1.1 Algorithm Details

Direction: Forward (follow execution order from entry to exit)

Join Operation: Intersection (∩) — a variable is definitely defined only if defined on *all* incoming paths.

Transfer Function:

$$\text{OUT}[B] = \text{IN}[B] \cup \text{Defined}[B]$$

Join at Block Entry:

$$\text{IN}[B] = \bigcap_{\text{pred}} \text{OUT}[\text{pred}]$$

Initialization: TOP (all registers assumed defined everywhere, except entry where only `r_in` is defined)

This is a *must* analysis (greatest fixpoint). Starting optimistically (everything defined) and shrinking down ensures we only keep properties that are truly guaranteed.

6.1.2 Implementation

The analysis state is represented using immutable maps:

We use immutable maps throughout:

```
type analysis_result = {
  in_sets : RegisterSet.t BlockMap.t;
  out_sets : RegisterSet.t BlockMap.t;
}
```

Each iteration returns a new `analysis_result`.

The core iteration loop implements the fixpoint algorithm. Here's how it works:

Listing 6.1: Definite Variables Fixpoint Iteration

```
let rec iterate iteration out_state =
  let new_out_state, new_in_state, changed =
    BlockMap.fold
      (fun id block (acc_out_state, acc_in_state, acc_changed) ->
        (* Get predecessors of the current block *)
        let preds = get_predecessors cfg id in

        (* Compute IN[B] = intersection of OUT[pred] *)
        let in_value =
          match preds with
          | [] ->
              (* Entry block: only r_in is defined *)
              if id = cfg.entry then
                RegisterSet.singleton input_register
              else all_regs  (* Unreachable *)
          | p :: ps ->
              (* Intersect OUT sets of all predecessors *)
              List.fold_left
                (fun acc p -> RegisterSet.inter acc (get_out p))
                (get_out p) ps
        in

        (* Compute OUT[B] = IN[B] union Defined[B] *)
        let defined_here = compute_defined block in
        let out_value = RegisterSet.union in_value defined_here in

        (* Check if anything changed *)
        let old_out = BlockMap.find id out_state in
        let has_changed =
          not (RegisterSet.equal out_value old_out) in

        (* Return updated state *)
        (BlockMap.add id out_value acc_out_state,
         BlockMap.add id in_value acc_in_state,
         acc_changed || has_changed)
      )
      cfg.blocks
      (out_state, BlockMap.empty, false)
  in

  (* Continue iterating if anything changed *)
  if changed then iterate (iteration + 1) new_out_state
  else (iteration, new_out_state, new_in_state)
```

Key implementation details:

1. Initialization: All blocks start with TOP (all registers defined), except the entry block which has only `r_in`.

2. Meet Operation: The intersection (`RegisterSet.inter`) ensures we only keep registers defined on *all* paths.

3. Transfer Function: Union of incoming definitions with locally defined registers (`RegisterSet.union`).

4. Termination: The loop continues until no OUT set changes (`RegisterSet.equal`).

5. Pure Functional Style: Each iteration creates new maps rather than mutating existing ones. This makes the code easier to reason about and test.

Safety Checking: The analysis result is used to detect uninitialized variables:

Listing 6.2: Safety Check Using Definite Variables

```
let check_safety cfg =
```

```ocaml
  let analysis_result = analyze cfg in

  let check_block id block =
    (* Get definitely-defined registers at block entry *)
    let initial_defs = BlockMap.find id analysis_result.in_sets in

    (* Check each instruction *)
    List.fold_left
      (fun (current_defs, errors) cmd ->
        let used = get_used_registers cmd in
        let defs = get_defined_registers cmd in

        (* Check if all used registers are defined *)
        let new_errors = List.fold_left
          (fun errs r ->
            if not (RegisterSet.mem r current_defs) then
              let err = Printf.sprintf
                "Block %d: Register %s used before definition"
                id (string_of_register r)
              in err :: errs
            else errs
          ) errors used
        in

        (* Update definitions for next instruction *)
        let updated_defs = List.fold_right
          RegisterSet.add defs current_defs in
        (updated_defs, new_errors)
      )
      (initial_defs, []) block.commands
  in

  (* Check all blocks and collect errors *)
  let all_errors = BlockMap.fold
    (fun id block acc -> check_block id block @ acc)
    cfg.blocks []
  in
  all_errors = []  (* Safe if no errors *)
```

This implementation threads the set of defined registers through each instruction, checking that every use is preceded by a definition.

6.2 Live Variables Analysis (Backward, May)

The live variable analysis determine which registers might be used later. It is essential for register coalescing (merge registers with disjoint live ranges)

6.2.1 Algorithm Details

Direction: Backward (propagate from exit to entry, opposite of execution order)

Join Operation: Union ($\cup$) — a variable is live if it *might* be used on *any* path.

Transfer Function:

$$\text{IN}[B] = (\text{OUT}[B] - \text{Def}[B]) \cup \text{Use}[B]$$

Where:

- `Use[B]` = registers read before being written in block
- `Def[B]` = registers written in block

Join at Block Exit:

$$\text{OUT}[B] = \bigcup_{\text{succ}} \text{IN}[\text{succ}]$$

Initialization: BOTTOM (empty set everywhere)

This is a *may* analysis (least fixpoint). Starting pessimistically (nothing live) and growing ensures we find all potentially live variables.

6.2.2 Data Structures

The live variables analysis uses multiple interconnected data structures to track liveness at different granularities:

1. Block-Level Liveness Maps:

```
block_in  : RegisterSet.t BlockMap.t   (* Live at block entry *)
block_out : RegisterSet.t BlockMap.t   (* Live at block exit *)
```

These maps associate each block ID with a set of registers. The `block_out` map is computed through fixpoint iteration (propagating backward from successors). The `block_in` map is derived from `block_out` using the transfer function.

2. Instruction-Level Liveness Map:

```
instr_after : RegisterSet.t InstrPointMap.t
```

This map tracks liveness after each instruction, keyed by instruction points:

```
type instr_point =
  | AfterInstr of int                (* After instruction i in block *)
  | Entry                             (* Before first instruction *)

type instr_point_key = block_id * instr_point
```

3. Control Flow Graph:

```
type risc_cfg = {
  blocks : risc_cfg_block BlockMap.t;   (* All basic blocks *)
  entry : block_id;                      (* Entry block ID *)
  exit : block_id;                       (* Exit block ID *)
}
```

The CFG provides successor/predecessor relationships needed for fixpoint iteration.

Once the fixpoint converges, `block_out` values seed the instruction-level backward walk. For each block, we start with its converged OUT set and propagate backward through instructions. The `instr_after` map records liveness after each instruction within blocks. This gives precise lifetime information for sequential code.

6.2.3 Implementation

The live variables analysis uses a backward (reverse execution order) fixpoint algorithm. The analysis state includes both block-level and instruction-level liveness:

Listing 6.3: Live Variables Analysis State

```
type analysis_result = {
  block_in : RegisterSet.t BlockMap.t;    (* Live at block entry *)
  block_out : RegisterSet.t BlockMap.t;   (* Live at block exit *)
  instr_after : RegisterSet.t InstrPointMap.t;
    (* Live after each instruction *)
}
```

First, we compute local information for each block—which registers are used before being defined (upward exposed) and which are defined (killed):

Listing 6.4: Computing Upward Exposed Uses

```
let compute_local_info block =
  let upward_exposed, killed =
    List.fold_left
      (fun (exposed, killed) cmd ->
        let uses = get_used_registers cmd in
        let defs = get_defined_registers cmd in

        (* Add to exposed if used before killed *)
        let new_exposed = List.fold_left
          (fun acc u ->
            if RegisterSet.mem u killed then acc
            else RegisterSet.add u acc)
          exposed uses
        in

        (* Update killed set *)
        let new_killed =
          List.fold_right RegisterSet.add defs killed in
        (new_exposed, new_killed)
      )
      (RegisterSet.empty, RegisterSet.empty)
      block.commands
  in
  (upward_exposed, killed)
```

The key insight: a register is "upward exposed" if it's used in the block before being defined. These are the registers that must be live at block entry.

The block-level backward analysis then propagates liveness information:

Listing 6.5: Block-Level Backward Fixpoint

```
let rec iterate iteration in_state =
  let new_in_state, new_out_state, changed =
    BlockMap.fold
      (fun id block (acc_in, acc_out, acc_changed) ->
        (* Get successors of current block *)
        let succs = get_successors cfg id in

        (* OUT[B] = union of IN[succ] for all successors *)
        let out_value = match succs with
          | [] ->
              (* Exit block: only r_out is live *)
              if id = cfg.exit then
                RegisterSet.singleton output_register
              else RegisterSet.empty
          | _ ->
              List.fold_left
                (fun acc (succ_id, _) ->
                  let succ_in = get_in succ_id acc_in in
                  RegisterSet.union acc succ_in)
                RegisterSet.empty succs
        in

        (* IN[B] = Use[B] union (OUT[B] - Def[B]) *)
        let (upward_exposed, killed) =
          compute_local_info block in
        let in_value =
          RegisterSet.union upward_exposed
            (RegisterSet.diff out_value killed) in

        (* Check for changes *)
        let old_in = BlockMap.find id in_state in
        let has_changed =
          not (RegisterSet.equal in_value old_in) in

        (BlockMap.add id in_value acc_in,
         BlockMap.add id out_value acc_out,
         acc_changed || has_changed)
      )
      cfg.blocks
      (in_state, BlockMap.empty, false)
  in

  if changed then iterate (iteration + 1) new_in_state
  else (iteration, new_in_state, new_out_state)
```

Key differences from definite variables:

1. Direction: Backward—we start from exit blocks and work toward entry.

2. Meet Operation: Union (`RegisterSet.union`) instead of intersection. A register is live if it's live on *any* successor path.

3. Initialization: BOTTOM (empty sets), growing upward as we discover live variables.

4. Transfer Function: IN[B] = Use[B] ∪ (OUT[B] - Def[B])—add upward exposed uses and propagate live variables that aren't killed.

6.2.4 Instruction-Level Liveness

The live variables analysis computes liveness at two granularities. Standard block-level analysis tells us what's live at block boundaries (IN[block] and OUT[block]), but for register coalescing, we need finer granularity: live *after each instruction.*

Algorithm:

Given block-level OUT set, propagate backward through instructions:

Listing 6.6: Instruction-Level Liveness Propagation

```
let compute_instruction_level_liveness cfg block_out_sets =
  BlockMap.fold (fun block_id block acc_map ->
    (* Get all instructions (commands + terminator) *)
    let all_instrs = match block.terminator with
      | None -> block.commands
      | Some term -> block.commands @ [term]
    in

    (* Start with what's live at block exit *)
    let block_out = BlockMap.find block_id block_out_sets in

    (* Walk backward through instructions *)
    let rec backward_walk instr_idx current_live acc =
      if instr_idx < 0 then
        (* Reached start - record entry point *)
        InstrPointMap.add (block_id, Entry) current_live acc
      else
        let instr = List.nth all_instrs instr_idx in

        (* Record: what's live AFTER this instruction *)
        let point = (block_id, AfterInstr instr_idx) in
        let updated_acc =
          InstrPointMap.add point current_live acc in

        (* Compute what's live BEFORE this instruction *)
        let defs = get_defined_registers instr in
        let uses = get_used_registers instr in
        let live_before =
          (* Remove killed registers, add used registers *)
          let after_kill = List.fold_right
            RegisterSet.remove defs current_live in
          List.fold_right RegisterSet.add uses after_kill
        in

        (* Continue to previous instruction *)
        backward_walk (instr_idx - 1) live_before updated_acc
    in

    (* Start from last instruction *)
    backward_walk (List.length all_instrs - 1) block_out acc_map
  ) cfg.blocks InstrPointMap.empty
```

This is a single backward pass through the block. Once we know OUT[block] from the block-level analysis, we can compute precise liveness for every instruction point.

Chapter 7

Optimization and Register Allocation

Register allocation is the heart of the compiler. We must map unlimited virtual registers to a fixed number of physical registers, inserting memory loads/stores when we run out.

7.1 Target Architecture

With n total physical registers:

Registers	Count	Purpose
`r_in`	1	Input (never spilled)
`r_out`	1	Output (never spilled)
`r_a, r_b`	2	Swap registers for spilling
`r0..r`$_{n-5}$	n-4	General-purpose
Total	n	

Available for variables: n - 4 registers (excluding I/O and swap)

Minimum: n = 4 (must have at least I/O + swap)

7.2 Register Coalescing (Optional)

The goal is to merge virtual registers that never interfere.

7.2.1 Live Range Construction

The first step is converting instruction-level liveness into per-register live ranges:

Listing 7.1: Building Live Ranges from Liveness

```
(* Type: register -> set of instruction points *)
type live_range = InstrPointSet.t RangeMap.t

let compute_live_ranges_from_liveness liveness_result =
  (* Invert the map: from (point -> live_set)
     to (register -> point_set) *)
  InstrPointMap.fold
    (fun point live_set acc ->
      RegisterSet.fold
        (fun reg range_map ->
          let current_points =
            try RangeMap.find reg range_map
            with Not_found -> InstrPointSet.empty
          in
          (* Add this point to the register's live range *)
          RangeMap.add reg
            (InstrPointSet.add point current_points)
            range_map
        )
        live_set acc
    )
    liveness_result.instr_after RangeMap.empty
```

This creates a map where each register is associated with the set of instruction points where it's live.

7.2.2 Interference Check

Two registers interfere if their live ranges overlap at any instruction point:

Listing 7.2: Checking Range Interference

```
let ranges_interfere range1 range2 =
  not (InstrPointSet.is_empty (InstrPointSet.inter range1 range2))
```

Example:

- r1 live at: $\{$(block 0, after instr 1), (block 0, after instr 2)$\}$
- r2 live at: $\{$(block 0, after instr 2), (block 0, after instr 3)$\}$
- Intersection: $\{$(block 0, after instr 2)$\} \neq \emptyset$
- Result: They interfere (cannot merge)

7.2.3 Merging Algorithm

The coalescing algorithm processes registers one by one, attempting to merge each into an existing group:

Listing 7.3: Register Merging

```
let merge_registers cfg =
  let live_ranges = compute_live_ranges_from_liveness liveness in

  (* For each register, try to merge into existing group *)
  let _, renaming, merged_count =
    List.fold_left
      (fun (groups, renaming, count) reg ->
        let reg_range = RangeMap.find reg live_ranges in

        (* Find first group with non-overlapping range *)
        let valid_group =
          RangeMap.fold
            (fun rep_reg group_range acc ->
              match acc with
              | Some _ -> acc   (* Already found *)
              | None ->
                  if not (ranges_interfere reg_range group_range)
                  then Some rep_reg
                  else None
            )
            groups None
        in

        match valid_group with
        | Some rep ->
            (* Merge into existing group *)
            let new_range =
              union_ranges reg_range (RangeMap.find rep groups) in
            let new_groups = RangeMap.add rep new_range groups in
            let new_renaming = RegisterMap.add reg rep renaming in
            (new_groups, new_renaming, count + 1)
        | None ->
            (* Start new group *)
            let new_groups = RangeMap.add reg reg_range groups in
            let new_renaming = RegisterMap.add reg reg renaming in
            (new_groups, new_renaming, count)
      )
      (RangeMap.empty, RegisterMap.empty, 0)
      all_regs
  in
  renaming
```

Note: r_in and r_out are never merged. This may miss some optimization opportunities (e.g., eliminating `copy r_out, r5`).

7.3 Allocation & Spilling

The goal is to map remaining virtual registers to physical slots or memory.

The allocator uses a frequency-based heuristic: count how many times each register appears (used or defined). The hottest registers get physical registers; the rest are spilled to memory.

7.3.1 Instruction Rewriting

When a register is spilled to memory, every instruction using it must be rewritten to load/store values. We use two swap registers (r_a and r_b) to handle this:

- **r_a:** Used for loading source operands and storing destination addresses
- **r_b:** Used for holding computed results before storing

For each instruction involving spilled register `r`:

Before Use (Load):

```
loadi 0x1000 => r_a    (* Load memory address *)
load r_a => r_a        (* Load value from memory *)
```

After Definition (Store, if live):

```
loadi 0x1000 => r_a    (* Load memory address *)
store r_b => r_a       (* Store value to memory *)
```

Listing 7.4: Spilling Code Rewriting

```
(* Helper: Load from memory if needed *)
let load_if_needed alloc_map reg temp_reg =
  match RegisterMap.find_opt reg alloc_map with
  | Some (InRegister r) -> ([], r)  (* Already in register *)
  | Some (InMemory addr) ->
      (* Generate load sequence *)
      ([LoadI (addr, temp_reg); Load (temp_reg, temp_reg)], temp_reg)
  | None -> ([], reg)

(* Helper: Store to memory if needed *)
let store_if_needed alloc_map reg val_reg addr_temp_reg =
  match RegisterMap.find_opt reg alloc_map with
  | Some (InMemory addr) ->
      [LoadI (addr, addr_temp_reg); Store (val_reg, addr_temp_reg)]
  | _ -> []

(* Rewrite binary register operation *)
let rewrite_binop op r1 r2 rd alloc_map =
  (* Load sources *)
  let l1, t1 = load_if_needed alloc_map r1 r_a in
  let l2, t2 = load_if_needed alloc_map r2 r_b in

  (* Determine destination *)
  let op_dest, stores =
    match RegisterMap.find_opt rd alloc_map with
    | Some (InRegister phys_reg) ->
        (* Destination in register: use directly *)
        (phys_reg, [])
    | Some (InMemory _) ->
        (* Destination spilled: compute to r_b, then store *)
      (r_b, store_if_needed alloc_map rd r_b r_a)
    | None -> (rd, [])
  in

  (* Final instruction sequence *)
  l1 @ l2 @ [BinRegOp (op, t1, t2, op_dest)] @ stores
```

7.3.2 Allocation Decision

After computing frequencies, allocate physical registers to the hottest variables:

Listing 7.5: Frequency-Based Allocation

```
let allocate_locations cfg n_target =
  let available_slots = n_target - 4 in  (* Exclude I/O + swap *)
  let freqs = get_frequencies cfg in

  (* Sort by frequency (highest first) *)
  let sorted =
    RegisterMap.bindings freqs
    |> List.filter (fun (reg, _) ->
        reg <> r_in && reg <> r_out && reg <> r_a && reg <> r_b)
    |> List.sort (fun (_, c1) (_, c2) -> compare c2 c1)
    |> List.map fst
  in

  (* Top (n-4) get physical registers, rest go to memory *)
  let alloc_map, next_mem_addr =
    List.fold_left
      (fun (map, next_addr) (idx, reg) ->
        if idx < available_slots then
          (* Fits in physical register *)
          (RegisterMap.add reg (InRegister reg) map, next_addr)
        else
          (* Spilled to memory at offset address *)
          let mem_addr = 0x1000 + next_addr in
          (RegisterMap.add reg (InMemory mem_addr) map, next_addr + 1)
      )
      (RegisterMap.empty, 0)
      (List.mapi (fun i r -> (i, r)) sorted)
  in

  (* Add I/O registers (always in physical registers) *)
  alloc_map
  |> RegisterMap.add r_in (InRegister r_in)
  |> RegisterMap.add r_out (InRegister r_out)
```

Memory Base Address: Spilled variables use addresses `0x1000`, `0x1001`, `0x1002`, ... This distinguishes them visually from small integer constants.