Project – Part 3 Documentation

Taimor sabek 212102362
Ariel Itskovich 214501348

04/02/2025

Compiler Structure

This compiler translates files from the C-- programming language into Riski. It uses a grammar-based approach with backpatching, and it's built on an LR bottom-up parser (Bison and Flex). Each time the parser reduces a rule, the generated commands are added to a global buffer, which later feeds into the linker to produce an execution file.

Data Structures

1. **Symbol Table**
    - A global map from a symbol's name to a `Symbol` structure.
    - Each `Symbol` stores:
        - A vector of types for all its declarations.
        - A vector of memory offsets for those declarations.
        - The depth of the deepest block where the symbol appears.
    - When declaring a variable:
        - Check if a variable of the same name already exists in the same depth. If yes, report an error.
        - Otherwise, insert the new variable with its memory offset.
    - On exiting a block, remove any variables declared within that block from the table.
2. **Function Table**
    - A global map from a function's name to a `Function` structure.
    - Each `Function` stores:
        - The address where it's defined.
        - Its return type.
        - A vector of argument types.
        - A list of call sites in the `.rsk` file.
        - A flag indicating whether it's implemented.
    - On declaring or defining a function:
        - Check if it already exists.
        - Verify that its definition (return type, arguments, name, etc.) matches any existing declaration.
3. **YYSTYPE**
    - Holds attributes for semantic actions during parsing. It includes:
        - The variable's name (or numeric value, for a literal).
        - The type.

        - The offset.
        - The falselist, truelist, and nextlist (for control flow).
        - The address (quad) in the `.rsk` file.
        - A register number (starting from 3, 0-2 are saved).
        - Vectors for function argument types, argument offsets, and argument register numbers.

Backpatching

- **If Statement**:
    - The “true” list is backpatched to the `then` part.
    - The “false” list merges with the `next` list to handle what follows.
- **If-Else Statement**:
    - Similar to an `if` statement, but the “false” list is backpatched to the `else` part.
- **While Loop**:
    - The “true” list is backpatched to the loop body.
    - The “false” list goes to the instruction after the loop.
- **Exiting a Block**:
    - When reducing a block (BLK), the block’s “next” list is backpatched to the correct instruction in the global buffer.
- **Function Call**:
    - After parsing is complete, we know each function’s start address, so we backpatch all the call sites stored in the function table.

Control Flow

- **Caller Role**:

    - Save all used registers on the stack.
    - Push the function arguments in order.
    - Reserve space for the return value.
    - Update `l2` (stack pointer) and set `l1 = l2`.
    - Save the program counter in `I0`, then call the function with `JLINK`.

- **Callee Role**:
  - Allocate new memory only upwards from `l2 (stack pointer)`.
- **Callee Role (When Returning)**:
  - Place the return value at `[l1 - 4]`.
- **Caller Role (When Returning)**:
  - Restore `l2` by setting `l2 = l1`.
  - Move the return value from `[l1 - 4]` into a register.
  - Restore the old `l1`.
  - Pop all previously saved registers from the stack.

Allocating Registers

- **Reserved Registers**:
  As recommended in the assignment:
  - `l0`: Return address
  - `l1`: Frame pointer
  - `l2`: Stack pointer
- **General Registers**:
  - Start at register 3.
  - On function calls, the caller saves all current registers at the newly allocated stack frame.
  - Each `STOR` instruction checks the type size and offset to know which registers to use.
- Memory management:
  Treated the memory as 1, meaning for a given offset only F or I memory are allocated (the holes in one memory are exactly the blocks of the other one).
- Converted the stack/frame pointers to float as needed when using STOR/LOADF.

Compiler Modules

- **Lex file**: Used for lexical analysis with Flex.
- **Part3_helpers.hpp**: Header containing compiler data structures and definitions.
- **Part3_helpers.cpp**: Implementations of the data structures (as given by course staff).
- **Part3.ypp**: used for semantic analysis with Bison:
  - The `main` function for the compiler.

- Allocation of output buffer.
- Creation of the `.rsk` output file.
- Printing of all generated code to the buffer.
- Syntax and semantic analysis of the source code (.cmm file).