Compiler Project

Ofek Dahan – 313598385, Ori Gross - 318829330

General Explanation of the Compiler Implementation

Our compiler translates a custom intermediate language (C--) into a sequence of assembly-like instructions for a virtual machine (RX-VM). It uses Flex for lexical analysis, Bison for parsing, and C++ for code generation and management. The compiler supports features like function calls, variable declarations, and conditional statements. During code generation, backpatching is employed to handle unresolved jump labels and ensure the correctness of the control flow.

Data Structures Used for Compiler State Management

1. **Symbol Table:**
   A map-based global structure that tracks all symbols (variables) encountered during compilation. It stores information such as the type (int/float), scope depth, and memory address of each symbol.
2. **Function Table:**
   Tracks function declarations and definitions globally. For each function, the table holds details like the return type, parameter types, the implementation address, and call locations. This is essential for resolving function calls and performing semantic checks.
3. **Type Stack:**
   Manages type consistency during parsing, especially for expressions involving mixed types (e.g., int + float).
4. **Jump Lists:**
   Dynamic lists used for managing forward references in control flow statements (e.g., if, while) and resolving jump targets during backpatching.

Backpatching Process

Backpatching resolves addresses for jump instructions generated during parsing. The key steps include:

1. **Function Calls:** After parsing, the addresses of called functions are backpatched with their actual implementation locations.
2. **Control Structures:**
    a. For if statements, the true branch is patched to the correct target.
    b. For if-else statements, both the true and false branches are resolved.
    c. For while loops, backpatching connects the condition evaluation and loop body.
3. **Boolean Expressions:** Logical operators (AND, OR) are backpatched to ensure correct evaluation sequences.

Structure of Activation Records

Each function invocation creates an activation record that includes:

- **Parameters:** Stored in the stack at specific offsets.
- **Local Variables:** Allocated in the stack with offsets relative to the frame pointer.
- **Return Address:** Stored in a dedicated register (I0).

Allocation of Special Registers

1. **Frame Pointer (I1/F1):** Points to the start of the current stack frame.
2. **Stack Pointer (I2/F2):** Points to the end of the current stack frame.
3. **Return Address (I0/F0):** Used for returning control after a function call.

Temporary calculations use registers (I3/F3 to I1024/F1024), with stack space allocated for values that must persist across operations.

Modules Description

1. **Lexical Analyzer (part3.lex):**
   Performs tokenization, identifying keywords, operators, and symbols in the input.
2. **Parser (part3.ypp):**
   Implements the context-free grammar and contains logic for syntax and semantic checks. Generates intermediate code and applies backpatching.
3. **Helper Functions (part3_helpers.hpp):**
   Defines utility functions, type management, and data structures such as symbol and function tables.

4. **Helper Functions (part3_helpers.cpp):**
   Defines the buffer class functions.

External Libraries Used

- **C++ STL:**
  - std::map: For managing symbol and function tables.
  - std::vector: For dynamic lists used in backpatching.
  - std::stack: For managing scope and type stacks.

This modular design ensures a clear separation of concerns and facilitates efficient parsing, semantic analysis, and code generation.