Logo, company name

Description automatically generated



**COMSATS University Islamabad (CUI)**

**Lab terminal**

**Submitted to:**

**Submitted By: ANIS MAJID**

**Reg No: FA21-BCS-024**

**Course: COMPILER CONSTRUCTION**

**Date: 3 JANUARY 2025**

**Question No 03) Explain optimization in your Mini compiler?**

**1. Single-Pass Compilation and On-the-Fly Code Generation:**

* **What it is:** The compiler performs lexing, parsing, and code generation in a single pass over the source code. It doesn't create an intermediate representation (like an Abstract Syntax Tree or AST) that would require memory allocation, complex manipulation, and traversal. Instead, as tokens are recognized, assembly code is generated immediately.
* **"Optimization" Aspect:**
  + **Reduced Memory Footprint:** Avoiding an AST or intermediate code representation significantly reduces the memory needed by the compiler. This is crucial for self-hosting, where the compiler needs to fit within its own address space and reduce its own compile times.
  + **Simplified Implementation:** No need for complex AST data structures or code for traversing them. This drastically simplifies the implementation, resulting in a smaller codebase and faster initial development.
  + **Reduced Compilation Time:** The single pass reduces the number of steps required to generate assembly.

**2. Stack-Based Intermediate Representation:**

* **What it is:** The compiler uses the CPU's stack for storing intermediate values during expression evaluation, particularly when handling binary operators. Instead of registers to keep these values, they are pushed onto the stack before each operation and popped afterward.
* **"Optimization" Aspect:**
  + **Reduced Register Usage:** By using the stack instead of registers, the compiler can use eax as the primary register for expression evaluation and avoid needing more register allocation.
  + **Simplified Code Generation:** It eliminates the complexity of tracking register usage and allocating/freeing registers, allowing for a simpler code generation logic.
  + **Less Memory Footprint:** The stack memory is already allocated at runtime, so no memory allocation is required for temporary results.

**3. Typeless Approach:**

* **What it is:** The mini-c language is typeless; all data is treated as a 4-byte signed integer (32 bits). This applies to variables, parameters, return values, and memory addresses. The compiler does not track or enforce any types.
* **"Optimization" Aspect:**
  + **Simpler Symbol Table:** The symbol table does not need to store type information, making it smaller and faster to manage.
  + **Simplified Parsing:** Parsing and expression handling becomes straightforward as no type checking or conversions are necessary. It eliminates the need of a type checker.
  + **Simplified Code Generation:** The compiler directly emits instructions to treat all values as 32-bit integers.
  + **Reduced compiler Complexity:** It reduces the overall complexity of the compiler, as the type checker is completely eliminated.

**4. Concatenation of Consecutive String Literals:**

* **What it is:** The compiler automatically concatenates consecutive string literals into a single string constant in the read-only data section (.rodata). This behavior is handled in the factor function.
* **"Optimization" Aspect:**
  + **Reduced Code Size:** By concatenating strings, the compiler avoids the creation of many small string constants, reducing data segment size and the need for multiple string labels in the final executable.
  + **Reduced Code Complexity:** Instead of outputting multiple strings, it will output just one.

**5. Minimalist Approach to Error Handling:**

* **What it is:** The compiler performs minimal error checking. When an error is detected, it outputs a message and continues to parse. No error recovery is attempted.
* **"Optimization" Aspect:**
  + **Reduced Code Size:** Reduced code complexity due to lack of complex error recovery algorithms.
  + **Simplified Implementation:** A basic error detection is easier to implement as complex error checking logic is omitted.

**6. Linear Symbol Table Lookups**

* **What it is:** The symbol table lookups are performed by linear scanning array.
* **"Optimization" Aspect:**
  + **Simpler Implementation:** While not the most performant for large symbol tables, using linear lookup simplifies the code and it is faster for small tables.

**Why These Aren't "True" Optimizations:**

* **Limited Scope:** These are not optimizations that are typical to modern production compilers. They are architectural choices that impact the compiler's structure and complexity rather than optimizing the generated code (like register allocation or common subexpression elimination).
* **Trade-offs:** These choices introduce trade-offs, often sacrificing performance in favor of size. The typeless approach is particularly limiting.
* **No Global View:** The mini-c compiler is not doing any interprocedural optimizations or analysis.
* **Targeted for Self-Compilation:** They are optimized for the specific task of building a small, self-hosting compiler and are designed to keep the codebase minimal.