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**

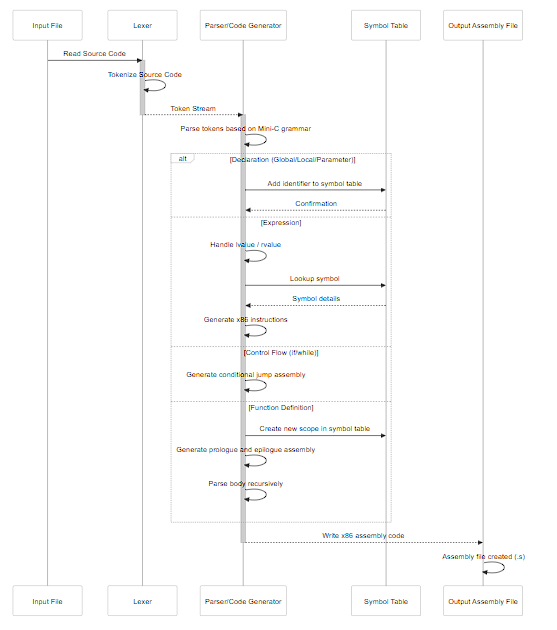
**Date: 3 JANUARY 2025**

**Explain 2 code functions of your mini compiler**

1. **void expr(int level):** This function is the core of the expression parser and code generator. It recursively parses expressions based on operator precedence using the level parameter.
   * It handles different types of expressions based on operator precedence, from basic operands (like variables and literals), function calls, array indexing to binary operators (+, -, \*, comparisons, logical operators) and assignments.
   * It generates x86 assembly instructions to compute the result of the expression, leaving the final result in the eax register. The intermediate results are pushed and poped from the stack.
   * It also handles lvalue checks. If an expression can be used on the left-hand side of an assignment, it sets the lvalue flag for other functions.
2. **void decl(int kind):** This function parses and generates code for variable and function declarations.
   * It handles three kinds of declarations: decl\_module (global variables and functions), decl\_local (local variables), and decl\_param (function parameters).
   * For variables, it reserves space and optionally initializes the variable with the given value. Globals are allocated in the .data section of the assembly code. Locals are allocated on the stack.
   * For functions, it creates a new scope and defines parameters. It generates the function's prologue, epilogue, and body with the help of function(char\* ident). It adds the function and variables into the appropriate symbol table.

**Question No 01)**

**SEQUENCE DIAGRAM**

****

**explain optimizations in your mini compiler**

The mini-compiler's primary goal is *simplicity* not *performance*, so there are few explicit optimizations. However, here are some notable aspects that contribute to its small size and effectiveness:

1. **Single-Pass Compilation:** The compiler does both parsing and code generation in a single pass. It avoid the cost and complexity of having an Abstract Syntax Tree (AST), saving memory and implementation effort.
2. **Stack-Based Intermediate Representation:** It uses the stack to hold temporary expression results. Instead of generating a intermediate language, it directly outputs x86 assembly using eax as the accumulator and stack for temporary storage.
3. **Typeless:** All values are treated as 4-byte integers. This eliminates a complex type system and type checking, simplifying the code and also allows pointer arithmetic to be simple.
4. **Concatenated String Literals:** The parser automatically concatenates consecutive string literals into a single string in the read-only data section of the assembly code. This minimizes the number of labels used and reduces the size of the final executable.

**explain input/ output of your mini compiler**

* **Input:**
  + The input to the mini-c compiler is a text file (.c) containing source code written in the simplified mini-c language.
* **Output:**
  + The primary output of the compiler is a text file (.s) containing x86 assembly language code.
  + This assembly code is then meant to be assembled and linked using another tool (like gcc) to create a working executable.

**outline symbol table**

The symbol table is implemented using arrays:

* **globals (char\*\* globals):**
  + Stores the names (identifiers) of global variables and functions as strings.
  + Indexed from 0 to global\_no - 1.
* **is\_fn (bool\* is\_fn):**
  + A boolean array parallel to globals, indicating if the corresponding entry in globals is a function or variable.
* **locals (char\*\* locals):**
  + Stores the names of local variables as strings.
  + Indexed from 0 to local\_no - 1 (local variable counter).
  + The parameter is stored as a local variable.
* **offsets (int\* offsets):**
  + Stores the stack offset of each local variable.
  + Corresponds to the locals array.
  + The base pointer ebp is used as the base address for calculating local stack offsets.
* **Lookup:** Symbol lookups are performed using linear search within these arrays using sym\_lookup function.

**EXPALIN THE FUNCTION WHICH PERFORM SYMMENTIC ANALYSIS IN THE MINI COMPILER.**

**Functions Involved in Semantic Analysis:**

* **sym\_lookup(char\*\* table, int table\_size, char\\* look):** Used to look up an identifier in a given symbol table. Used for both globals and locals.
* **new\_global(char\* ident):** Adds a global variable or function identifier to the global symbol table (globals).
* **new\_fn(char\* ident):** Adds a function identifier to the global symbol table (globals) and sets the is\_fn flag.
* **new\_local(char\* ident):** Adds a local variable identifier to the local symbol table (locals) and sets its stack offset.
* **new\_param(char\* ident):** Adds a parameter identifier to the local symbol table (locals). Also computes the stack offset for parameter based on the cdecl convention.
* **new\_scope():** Resets the local symbol table and parameters to start a new function scope.
* **needs\_lvalue(char\* msg):** Checks if the lvalue flag is set, prints an error if not, and resets the flag.
* **decl(int kind):** Handles variable and function declarations and interactions with the symbol table.
* **object():** The call operation (expr ("," expr)\*)? from object calls expr recursivelly which performs semantic checks for expressions.