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 02)**

**1. void expr(int level)**

This function is at the heart of the compiler's expression parsing and code generation process. It's responsible for handling expressions with different operator precedences.

**Purpose:**

* To parse a mini-c expression according to operator precedence rules.
* To generate corresponding x86 assembly code to evaluate the expression, leaving the final result in the eax register.
* To support the lvalue concept in mini-c.
* To handle short circuit evaluation of logical operators.

**How it Works (Detailed Breakdown):**

1. **Recursive Descent Parsing:** The function implements a recursive descent parsing strategy. The level parameter controls the level of precedence being handled, starting with the lowest (level=0 for assignments) and going to the highest (level=5 for unary operators and basic operands).
2. **Base Case:** When level is 5, the function calls unary() which in turn handles basic operands (literals, identifiers), function calls and array indexing. It does not recurse any further. This step obtains the value of single operands and puts them in eax.
3. **Recursive Step:** For levels below 5, expr(level) does the following:
   * First, it calls expr(level+1) to evaluate the next higher precedence expressions.
   * Then it checks if the current operator from its level are present.
     + If yes it will process the operator:
       - Pushes the result of the lower level evaluation into the stack.
       - Advances to the next token.
       - Call expr(level+1) to evaluate the right hand side expression.
       - Generates the assembly code for the operator, using the value on eax and the value previously stored in stack.
   * This loop continues until the current token is not an operator of current level.
4. **Operator Handling (Assembly Generation):** The expr function generates assembly instructions based on the detected operators:
   * **Arithmetic:** +, -, \*: Uses add, sub, imul instructions.
   * **Comparison:** ==, !=, <, >=: Uses cmp with conditional set instructions sete, setne, setl, setge.
   * **Logical:** ||, &&: Employs short-circuiting using conditional jumps (jz, jnz) to avoid evaluating unnecessary parts of the expressions. The evaluation ends as soon as the final result is known.
   * **Ternary:** ?:: Uses conditional jump instructions to skip the false branch expression.
   * **Assignment:** =, Generates the mov instruction to store the result on the specified address.
5. **Lvalue Detection:** Before parsing assignments, the lvalue flag is set by factor.
6. **Stack Management:** The function carefully uses the stack to store the intermediate results of subexpressions before applying binary operators, in order to not to overwrite the eax register.
7. **Short Circuit Evaluation:** The function implement short-circuit evaluation for logical operators || and &&. It uses conditional jumps to bypass the evaluation of the right-hand side operand.
8. **Ternary Evaluation:** The function implements the ternary operator ? : by jumping to either the true or false expression depending on the expression before the question mark.

**Example:**

For the mini-c expression a + b \* c - d, the expr function would be called recursively as follows (roughly):

1. expr(0) // Lowest precedence (assignment, not found in this expression)
2. expr(1) // Ternary operator, not found
3. expr(2) // Logical operators, not found
4. expr(3) // Comparison operators, not found
5. expr(4): "+"  
   6. expr(5) // Calls unary -> gets a  
   7. expr(4): "*"  
   8. expr(5) // Calls unary -> gets b  
   9. expr(5) // Calls unary -> gets c  
   10. Computes b*c using imul.  
   11. Computes a + (b \* c) using add  
   12. expr(4): "-"  
   13. expr(5) // Calls unary -> gets d  
   14. Compute (a + b \* c) - d using sub

**Key Parts of the Code:**

* if (level == 5): Base case, calling unary().
* Recursive calls: expr(level + 1)
* Operator checks: see("+"), see("-"), etc.

**2. void decl(int kind)**

This function parses and generates code for declarations. It handles global, local, and parameter declarations.

**Purpose:**

\* To parse mini-c variable and function declarations.

\* To add declared identifiers to the correct symbol table (`globals` or `locals`).

\* To allocate storage for declared variables.

\* To generate the prologue of function for allocated local variables.

\* To generate optional initial values for variables.

\* To generate assembly code for globals declaration.

**How It Works (Detailed Breakdown):**

1. **Declaration Type:** The kind parameter determines the type of declaration:
   * decl\_module (1): For global variables and functions at the top level of the program.
   * decl\_local (2): For local variables declared inside a function.
   * decl\_param (3): For function parameters.
2. **Type Specifier Parsing:** The function first expects the keywords int, char or bool, advances the cursor, and then parses pointer declarator \*. These keywords are mostly ignored as mini-c is typeless.
3. **Identifier Parsing:** It then extracts the identifier (variable or function name) from the token buffer.
4. **Function Declaration Handling:**
   * If it detects a function declaration (, it creates a new scope if the current kind is decl\_module.
   * It then parses function parameters by recursively calling decl with kind set to decl\_param.
   * The function is added into the symbol table.
   * If it detects a function definition {, it calls the function function to generate function prolog, body and epilogue.
5. **Variable Declaration Handling**
   * If the declaration is local, it calls new\_local to add the variable to local symbol table, and to compute the stack offset.
   * If the declaration is global, it calls new\_global to add the variable to the global symbol table.
6. **Initialization (Optional):** If it sees an = sign (for both globals and locals), it parses the initialization expression using expr(0) and generates the mov instruction to initialize the variables. Global variables are initialized in .data section of the assembly file and local variables are initialized at run time with assembly instructions.
7. **Global Variable Handling:** If the declaration is global, the function outputs a definition in the .data section with the correct .quad directive for the variable and optional initialization.
8. **Semicolon Termination:** For local variables and global variable declarations it will consume the closing semicolon using match(";").
9. **Error Checking:** The function checks for common errors such as invalid initialization of function or parameters.

**Example:**

For the mini-c code:

```c

int global\_var = 10;

int function(int param) {

int local\_var = 20;

return local\_var;

}

```

content\_copydownload

The decl() function would handle it as:

1. decl(decl\_module): Processes int global\_var = 10;
   * Adds global\_var to the global symbol table.
   * Generates assembly code to declare in the data section .quad 10.
2. decl(decl\_module): Processes int function(int param) { ... }
   * Adds function to the global symbol table.
   * Resets local scope.
   * decl(decl\_param): Processes parameter int param, and add to local symbol table, with stack offset computation.
   * Calls function to process function body.
     + decl(decl\_local): process int local\_var = 20;, adding to local symbol table and assigning stack offset.
     + Generates assembly code for the local variable initialization, and return.
     + Generates function prologue/epilogue.

\*\*Key Parts of the Code:\*\*

* switch(kind): To check what type of declaration we have.
* Function declaration processing using try\_match("(")
* new\_global, new\_local, new\_param: adding identifiers to symbol table
* Assembly generation in different .section, using fprintf
* Recursive calls to decl to parse function parameters.
* Calls the function to generate function code.
* expr(0) is used to parse variable initialization.