**Give two core functions of your mini compiler ?**

Answer:

**Core Function 1: Parsing Program Structure - parser.mly (procedure and statement Rules)**

* **Purpose:** The parser.mly file defines the grammar rules for the Mini-C language, transforming the sequence of tokens into an Abstract Syntax Tree (AST). This phase is where the structure of the code is parsed and represented in an abstract and structured form.
  1. **procedure Rule Funtion (Code Snippet from parser.mly):**

**procedure:**

TYPE ID LPAREN parameters RPAREN block { mkProc $1 $2 $4 $6 }

| TYPE ID UNIT block { mkProc $1 $2 [] $4 }

;

* + **Explanation:** This code shows the procedure rule which matches two different syntax patterns. The first rule matches a function definition that has formal parameters inside parenthesis, and the second rule matches a function definition that has no formal parameters, when the parenthesis are empty. In both cases, it creates an AST node to represent a procedure by calling mkProc (a constructor that is defined at the top of the parser.mly file in OCaml), using values obtained from parsing and the values of $1, $2, $4, and $6 obtained from matching grammar symbols.
  1. **statement Rule Function (Code Snippet from parser.mly):**

**statement**:

RETURN exp SEMI { Ast.Return($2) }

| ID LPAREN expressions RPAREN SEMI { mkCall $1 $3 }

| ID UNIT SEMI { mkCall $1 [] }

| ID GETS exp SEMI { mkAssn $1 $3 }

| PRINT exp SEMI { Ast.Print($2) }

| block { $1 }

| WHILE LPAREN exp RPAREN statement { mkWhile $3 $5 }

| IF LPAREN exp RPAREN statement ELSE statement { mkIfS $3 $5 $7}

;

* 1. **Explanation:** The code above is the statement rule which acts as the main dispatcher for all types of statements, using pattern matching. For each different type of statement, it calls a corresponding AST constructor function to produce the AST representation of the given statement. For example, RETURN exp SEMI would call Ast.Return, ID GETS exp SEMI calls mkAssn and so on. Each of these constructors is typically defined at the beginning of the parser.mly file, or in a corresponding file that is used by the parser.mly file, using OCaml code.

**Core Function 2: Generating Assembly Code - codegen.ml (translateProcedure function)**

* **Purpose:** The codegen.ml file handles the translation of the AST into x86-64 assembly instructions. The translateProcedure function orchestrates this process for each procedure (function) in the program.
* **translateProcedure Function (Code Snippet from codegen.ml):**
* let translateProcedure (Q.Procedure {entry; formals; code}) =
* let (numLocals, env) = makeEnv formals code in
* let prologue = calleePrologue entry numLocals in
* let instructions = List.map (translateInstruction env entry) code in
* let instructions = concatCodestreams instructions in
* let epilogue = calleeEpilogue entry
* in
* Debug.dumpCGEnv entry (offsetsOf env);

(entry, concatCodestreams [prologue; instructions; epilogue])

* + **Explanation:** The translateProcedure function takes an OCaml record, containing the name (entry point), formal parameters and the list of instructions (code) as input. The makeEnv formals code function produces a mapping between variables and stack offsets, for use in generating memory access. This is called a "symbol table". The calleePrologue entry numLocals function creates the assembly instructions for the function's prologue (setup). The code of the function is converted into a list of assembly instructions by mapping translateInstruction, using the symbol table env and the entry point of the function to correctly produce the assembly code for each instruction in the function. The instructions are concatenated into a single code stream. Lastly, the calleeEpilogue function creates the assembly code to correctly tear down the stack frame when a function returns to the calling function. Lastly, debugging information is printed to assist with the debugging process.