Extending a Core ML like language with Exception

Final Project Report
CS1023 - Software Development Fundamentals

Anant Maheshwary
CS24BTECH11006

May 2025

1 MiniML

An implementation of an eager, statically typed functional language with a compiler and an abstract machine.

The language has the following constructs:

- Integers with arithmetic operations `+`, `-`, and `*`. (There is no division because the language has no exceptions.)
- Booleans with conditional statements and comparison of integers `=` and `<`.
- Recursive functions and function application. The expression

  ```
  fun f (x :  t) :  s is e
  ```

  denotes a function of type $t \to s$ which maps x to e. In e the function refers to itself as f.
- Toplevel definitions

  ```
  let x = e
  ```

  There are no local definitions.

2 Project Aim and Extensions

The aim of the project is to extend the MiniML language and add the following features:

1. **Division**
   Syntax: `e1 / e2`
2. **Define two new exceptions:**
   (a) `DivisionByZero` exception
   (b) `GenericException` which takes an integer as an argument

3. **Support for raising exceptions**
   Syntax: `raise GenericException i` or `raise DivisionByZero`
4. **Adding try-with blocks for handling multiple exceptions.**
   Syntax:

```
try { expression } with {
  | Exception1 -> expression1
  | Exception2 -> expression2
}
```

In this project, `GenericException` is thrown in case of type errors, for example, arithmetic operations on invalid types or calling functions with arguments of invalid type, etc.

3 Design and Implementation

3.1 Working of the Original MiniML Language

Compilation Workflow:

1. **Lexical Analysis (`lexer.mll`):**
   - **Role:** The `lexer.mll` file defines the rules for the lexer (also known as a scanner or tokenizer).
   - **Process:** It takes the raw input program (a string of characters) as its input.
   - **Output:** It breaks down the input into a stream of tokens. Each token represents a basic building block of the language, such as keywords (`let`, `fun`, `if`), identifiers (variable names), literals (integers, booleans), operators (`+`, `-`, `*`, `/`), punctuation (`(`, `)`, {, }), and special symbols (`->`, `=`).
   - **Example:** For the input `let x = 5 + 2;;`, the lexer would produce a sequence of tokens like: `LET`, `VAR "x"`, `EQUAL`, `INT 5`, `PLUS`, `INT 2`, `SEMISEMI`.
2. **Syntactic Analysis (`parser.mly`):**
   - **Role:** The `parser.mly` file defines the grammar of the miniML language.
   - **Process:** It takes the stream of tokens produced by the lexer as its input.
   - **Output:** It attempts to structure these tokens according to the grammar rules. If the token sequence is syntactically correct, the parser produces an Abstract Syntax Tree (AST). The AST is a hierarchical representation of the program's structure, making it easier for the compiler to understand the meaning of the code. If the token sequence violates the grammar, the parser reports a syntax error.
   - **Example:** For the token sequence from the previous step, the parser would build an AST that represents a `Def` (definition) where the variable `"x"` is bound to a `Plus` expression with operands `Int 5` and `Int 2`.
3. **Abstract Syntax Tree Definition (`syntax.ml`):**
   - **Role:** The `syntax.ml` file defines the data structures that represent the abstract syntax of the miniML language.
   - **Content:** It typically includes OCaml type definitions for:

- `name`: Representing variable names (usually strings).
- `ty`: Representing the types in the language (e.g., `TInt`, `TBool`, `TArrow` for function types, `TExp` for exceptions).
- `expr`: Representing the different kinds of expressions in the language (e.g., `Var`, `Int`, `Bool`, `Plus`, `Minus`, `If`, `Fun`, `Apply`, `Let`, `TryWith`, `Raise`). Each constructor in the `expr` type corresponds to a syntactic construct in the language and holds the necessary sub-expressions and information.
- `command`: Representing top-level commands that can be executed (e.g., evaluating an expression, defining a variable).

- **Importance:** This file acts as the blueprint for the AST that the parser creates and the compiler consumes.

4. **Compilation to Bytecode (`compile.ml`):**

   - **Role:** The `compile.ml` file defines the compiler, which translates the AST into a sequence of instructions that can be executed by the miniML virtual machine.
   - **Process:** It takes the AST (produced by the parser based on the `syntax.ml` definitions) as input.
   - **Output:** It generates a list of low-level instructions (bytecode) that are specific to your miniML virtual machine. These instructions represent operations like pushing values onto a stack, performing arithmetic, comparing values, branching, creating closures, and handling exceptions.
   - **Example:** A `Plus (Int 5, Int 2)` AST node might be compiled into a sequence of instructions that push the integer 5, push the integer 2, and then perform an integer addition.

Execution Workflow:

- **Virtual Machine Execution (`machine.ml`):**
  - **Role:** The `machine.ml` file defines the miniML virtual machine, which is responsible for executing the bytecode generated by the compiler.
  - **Components:** It typically includes:
    - **State:** Data structures to represent the machine's state during execution, such as a stack (for storing intermediate values), an environment (mapping variables to their values), and potentially a program counter (to track the current instruction).
    - **Instruction Set:** Definitions for the bytecode instructions that the compiler generates.
    - **Execution Cycle:** A loop that fetches, decodes, and executes the bytecode instructions one by one, updating the machine's state.
  - **Process:** It takes the list of bytecode instructions as input.
  - **Output:** It executes the program according to the instructions, potentially producing a final value or triggering side effects (if your language supports them). It also handles runtime errors, such as division by zero or unbound variables (if not caught during type checking or if your machine handles them). For exception handling, it would manage the stack and environment to find appropriate exception handlers when a `Raise` instruction is executed.

3.2 Extension to this Project

Below are the main additions made to each file for extending MiniML with exception handling and related features:

1. **Lexer (`lexer.mll`):** Added new tokens: `DIVIDE`, `PIPE`, `LBRACE`, `RBRACE`, `TRY`, `WITH`, `DIVISIONBYZERO`, `GENERICEXCEPTION`, `RAISE`, and `TEXP`.

2. **Parser (`parser.mly`):** Defined new parsing rules:

   - **For division:**

     ```
     | e1 = expr DIVIDE e2 = expr
         { Divide(e1, e2) }
     ```

   - **For raising exceptions:**

     ```
     | RAISE e = exception_type
         { Raise(e) }
     ```

   - **For handling exceptions (try-with blocks):**

     ```
     | TRY LBRACE try_e = expr RBRACE WITH LBRACE handlers = exception_handlers
        RBRACE
         { TryWith(try_e, handlers) }
     ```

     Here, `exception_handlers` is a non-empty list of tuples of an exception type and the expression to be evaluated if the exception is caught.

   - **Exception types:**

     ```
     | DIVISIONBYZERO
         { DivisionByZero }
     | GENERICEXCEPTION e_int = INT
         { GenericException(e_int) }
     | GENERICEXCEPTION MINUS e_int = INT
         { GenericException(-e_int) }
     ```

3. **Syntax (`syntax.ml`):** Defined the corresponding constructors:

   ```
   | Raise of exception_type         (* To raise an exception *)
   | TryWith of expr * (exception_type * expr) list
   | Exception of exception_type     (* Exception *)
   ```

   Added new type:

   ```
   ty = ... | TExp
   ```

4. **Changes to `machine.ml` and `compile.ml`:**
   Added a new machine value:
   `MExp`
   Added new instructions:

   - `IDiv`
     Performs integer division. Pushes the result of the division onto the stack, or an `MExp` value if the denominator is zero.
   - `IExp`
     Pushes an `MExp` (exception) value onto the stack.

- `IRaise`
  Clears the environments, stack, and frames of the machine, then pushes an `MExp` onto the stack to indicate an exception has been raised.
- `IHandle`
  Checks if the expression on the top of the stack is an exception; if so, adds the corresponding list of instructions to the frames for exception handling.