CSCI-3110 Honors Programming Languages
Project Report - MiniOO Interpreter

Muhammad Musa Khan - mk8737

How to run

1. External dependencies required: menhir. Install using ‘opam install menhir’.
2. Unzip the project files from the folder.
3. A makefile is included in the project directory. Run ‘make all’ to create the executable file.
4. Run ‘./interpreter < examples/input1.oo’ to feed a MiniOO source code file to the interpreter.
5. Alternatively, run ‘./interpreter’ to type MiniOO code inside the terminal. End by pressing Ctrl+D twice.
6. Run ‘make delete’ to delete the compiled files and the executable file.

Project Structure

1. `examples/`:
   - This folder contains example `.oo` input files (e.g., `input1.oo`, `input2.oo`, etc.) that demonstrate various features of the MiniOO interpreter. These files are used for testing and showcasing the interpreter's functionality. Each input file begins with a miniOO comment stating the feature covered.
2. `makefile`:
   - Automates the build process for the MiniOO interpreter. It includes rules for compiling the lexer, parser, and other modules, as well as for linking the final executable.
3. `main.ml`:
   - The entry point for the MiniOO interpreter. It integrates all components, including the lexer, parser, static semantics, and operational semantics, to process input programs, run static checks, and execute them.
4. `ast.ml`:
   - Defines the Abstract Syntax Tree (AST) data structure for the MiniOO language. It includes types for program elements like blocks, commands, expressions, and boolean expressions, forming the core representation of the language.
5. `lexer.mll`:
   - Implements the lexical analyzer for MiniOO, written using `ocamllex`. It tokenizes the input source code into meaningful tokens, such as keywords, operators, identifiers, and literals, which are passed to the parser.
6. `parser.mly`:
   - Implements the syntax analyzer (parser) for MiniOO, written using `menhir`. It defines the grammar of the language and translates sequences of tokens into an AST.

7. `operationalSemantics.ml`:
    - Implements the dynamic behavior of MiniOO programs. It defines functions to execute commands, manage the program's state (stack, heap, and address), and evaluate expressions and boolean conditions during runtime.
8. `operationalTypes.ml`:
    - Contains type definitions for the interpreter's runtime components, such as values, environments, stack frames, heap, and program state. These types are used by the operational semantics.
9. `prettyPrint.ml`:
    - Contains functions for visually representing the AST and runtime state of MiniOO programs in a human-readable format. It enhances debugging and program visualization.
10. `staticSemantics.ml`:
    - Implements static checks for MiniOO programs to ensure correctness before execution. It verifies properties like variable declarations, preventing runtime errors.

Features with Examples

My MiniOO Interpreter supports all the features from the MiniOO specification with the exception that fields must be integers, not variables themselves. Moreover, I have expanded the boolean operators to include >, <=, >=, and arithmetic operators to include +, *, /. Semi-colon after the last command is required unlike in the specification. Parallel Execution randomly executes one of the two branches. The interpreter also supports nested multi-line comments: comments must begin with (* and end with *). Below are the 7 example programs included in the examples/ directory with additional comments.

`input1.oo`: Variable Declaration and Assignment

```
(* Variable Declaration and Assignment *)
var x;
x = 10;
if (x > 5) then {print(x);} else {print(0);};
```

- The same code can also be written as below, demonstrating concatenation C;C

```
var x; {x = 10; {if (x > 5) then {print(x);} else {print(0);};;}}
```

`input2.oo`: Procedure and Function Call

- {} around function body is optional

```
(* Procedure and Function Call *)
var p;
p = proc y: print(y + 10); print(0);;
p(5);
```

`input3.oo`: Conditional Statements

- () around condition is optional
- Else branch is optional
- {} around each branch is optional, but semicolons must be the right number. 1 for the branch, and 1 for the entire if statement.

```
(* Conditional Statements - () around condition is optional, so are {}
around branches,
but semicolons must balance out *)
var x;
x = 10;
if (x > 5) then {print(x);} else {print(0);};
x = 2;
if x < 5 then print(x); else print(1);;
x = 10;
if x == 10 then print(2);;
```

`input4.oo`: While Loop

- Brackets {} around ‘while loop’ body are optional. Appropriate use of semi-colons is required. Indentation does not matter.
- Parenthesis () around the condition is optional.

```
(* While Loop and Arithmetic Operators *)
var x;
x = (10 * 2) / 4 + 1 - 2;
while (x > 0) {
  print(x);
  x = x - 1;
};
```

`input5.oo`: Field Access and Malloc

- This should give a runtime error, demonstrating that malloc command is necessary to allocate memory.

```
(* Field Access and Malloc *)
var x;
(* malloc(x); *) (* de-comment this line to remove runtime error *)
x.F = 100;
print(x.F);
```

`input6.oo`: Parallel and Atomic Execution

- Parallel randomly executes one of the branches: C1 OR C2

```
(* Parallel Execution *)
var x;
var y;
{
  x = 1; print(x); print(x);
 |||
  y = 2; atom(print(y); print(y););
};
```

`input7.oo`: Factorial

```
(* Factorial *)
var factorial;
var result;
result = 1;

factorial = proc n: {
    if (n == 0) then {
        print(result);
    }
    else {
        result = result * n;
        factorial(n - 1);
```

```
    };
};

factorial(5);
```

Other Information

Field and Variable naming:

- A valid variable name must start with a lowercase letter and can be followed by letters, digits, or underscores. A valid field name must start with an uppercase letter and can also be followed by letters, digits, or underscores.

THE END