CSC254 Assignment 3: Interpretation

Team: Yuesong Huang, Wentao Jiang

E-mail: yhu116@u.rochecter.edu, wjiang20@u.rochester.edu

Introduction:

- We implemented two versions of complete interpreters:
  i. **Base** (with function: `interpret (ast:ast_sl) (full_input:string) : string`): `intepreter.ml`,
  ii. **Static Analysis** (with function: `dfs (ast:ast_sl): status * memory * string list`): `interpreter_static_analysis.ml`.
- Both versions will generate equivalent C code (with function: `c_gen (ast:ast_sl) filename : unit`)
  - if it's interpreted successfully or passes static analysis, the generated equivalent C code will automaticlly save to `[filename].c`
  - for the `[filename].ecl` program you used.

How to Build the Project:

- Normal compile will give you the compiled version of the interpreter named `ecl`:

```
make ecl
```

- Compile with *Extra Credit* will give you the compiled version of the interpreter named `ecl_sa`:

```
make ecl_sa
```

How to Run the Project:

- **With static analysis** and **C code generation** (*Extra Credits*):
  - Run the `prog.ecl` program with stdin input end by `Ctrl^D`:

    ```
    make run_sa
    ```

  - Run the `prog.ecl` program and take the input for input file:

    ```
    make run_sa_in
    ```

  - Run with stdin input end by `Ctrl^D` but prefix ECL programs (sum-and-ave, primes, gcd, sqrt):

```
make run_sa_test1
...
make run_sa_test4
```

- Run the ECL program provided in *Trivia Assignment 2* that read 2 integers:

```
make run_sa_test5
```

- Run the ECL program provided in *Trivia Assignment 3* that read an integer n, then read n real numbers:

```
make run_sa_test6
```

- Custom ECL program power program that read 2 integers k and n and return k^n (take 2 integers as input):

```
make run_sa_test7
```

- Custom ECL program min-max that read an integer n, then read n real numbers and output the min. and max.:

```
make run_sa_test8
```

- Run tests with input file:
  Note: the input files need to be manually change according to ECL program.

```
make run_sa_test1_in
...
make run_sa_test8_in
```

- Run tests with input file then compile the generated C code according the `[filename].ecl` you used:

```
make run_sa_c_in
make run_sa_test1_c_in
...
make run_sa_test8_c_in
```

- **Base** without static analysis:
  - Run the `prog.ecl` program with stdin input end by `Ctrl^D`:

```
make run
```

- Run the `prog.ecl` program and take the input for input file:

```
make run_in
```

- Run with `stdin` end by `ctrl'D` but prefix ECL programs:

```
make run_testl
...
make run_test8
```

- Run tests with input file:
  Note: the input files need to be manually change according to EGL program.

```
- make run_testl_in
...
- make run_test8_in
```

- Run tests with input file then compile the generated C code according the `[filename].eel` you used:

```
make run_c_in
make run_testl_c_in
...
make run_test8_c_in
```

How to Clean the Project:

- Clean all binary files and C source files:

```
make clean
```

Extra Credit Implementation:

1. **Static Analysis:**
   - This involves introducing a separate pre-pass over the Abstract Syntax Tree (AST) to enforce static semantics.
   - With this approach, we are able to identify:
     a. Use of an undeclared variable,
     b. Redeclaration of a variable in the same scope,
     c. Non-integer provided to float,
     d. Non-real provided to trunc,

e. Type clash in binary expression, comparison, or assignment,
f. check statement not inside a loop.

- To carry out this process, we first perform a Depth-First Search (DFS) on the AST.
- If the finally status indicates `Done`, it means there are no static issues.
- After static analysis and the interpretation, if the status is `Done`, the system proceeds to generate the equivalent C code.

- **Remark1:**
  - The DFS function is utilized for semantic analysis.
  - If you wish to test the semantic analysis independently or test the interpreter, you can enter the interactive mode.
    In this mode, calling `print_string (interpret (ast:ast_sl) (full_input:string))` in both versions will interpret the AST without performing static analysis. On the other hand, invoking `dfs (ast:ast_sl)` in `interpreter_statie_analysis.ml` will only execute the static analysis, and return a tuple of `status * memory * string list`.
  - Additionally, please note that if you wish to enter the interactive mode, you must first load using `#load "str.cma";;` then use `#use "interpret.ml";;` or `#use "interpreter_statie_analysis.ml";;` to test the interpreter.

2. **Generating Equivalent C Code:**
   As mentioned above. After Generating equivalent C code, you can use `gee [filename].e -o [exeutable_name]` then `./[exeutable_name]` to compile and run it.

Feature:

1. Pure Functional Programming Features. The behavior is exactly the same as the given excutable file.
2. We have implemented structured static analysis.
3. Extra Credit: static analysis.
4. Extra Credit: generate equivalent C code source file for the ECL program.

Program Experience:

1. We have learned a lot about the OCaml language. We have learned how to use the OCaml language to implement a interpreter and by static analysis. Also we have learned how to use the OCaml language to generate equivalent C code for the ECL program.
2. We have noticed that if we have a nested match with multistatement in one case, then we have to then we have to use `()` to wrap the statement in match cases. Otherwise, the compiler will complain about the syntax error. This have produced a lot of complaints from us.

3. Programming in OCaml has been a distinctive experience. One of the immediate differences we noticed was its use of `<>` for inequality instead of the `!=` commonly found in other languages, requiring a structural comparison approach. OCaml is notably rigorous when it comes to syntax, which at times can pose challenges to those more accustomed to more permissive languages. Nevertheless, this strictness often translates into robust and error-free code.