Report

Introduction

The is a programming project for course *CSCI-GA.3110-001 Honors Programming Languages* at NYU. In this project, an interpreter for the dynamic object-oriented mini-language *Miniool* is designed and implemented in *OCaml*. The syntax and semantics strictly follow the MiniOO-SOS. One specification needed to be clarified is that the variables all start with an upper case, and the fields start with a lower case.

How to Execute

To run code in *miniool* such as *var X; X = 1*, use

```
> make
> ./miniool
  var X; X = 1
  ...
```

Whenever you want to exit the interpreter, use *Ctrl-D*.

You can remove your extra junk files in */bin* as below.

```
> make delete
```

Features and Examples

We will introduce the features in *miniool*. If you want to run the examples, use

```
> make example{index}
```

1. Check the static Scoping Rules

The static scoping rules define that "*every variable use must match a declaration of that variable according to the static scoping rules*". In our interpreter, we throw an error when there is an undeclared variable and reports a warning for the declared but unused variables. For example, the variable *Y* is not declared in *var X; Y = 1*. Thus, the interpreter throws an error.

```
> make example1
```

```
# Check the static Scoping Rules
var X; Y = 1
Checking static scoping rules...
Error: Variable Y is not declared.
```

2. Pretty Print AST and Configurations

Our interpreter can pretty print the abstract syntax tree, and output the change to the configuration (i.e. control, stack and heap) after each step of the program execution. Here is a simple example.

```
> make example2
# Pretty Print AST and Configurations
var X; X = 1
Checking static scoping rules...
All static scoping checking passed.

Printing AST...
... (omit the printed AST)

Executing program...
Control:
var X; {malloc(X); X.f = 5}
Stack:
Heap:
>>======>
Control:
block({malloc(X); X.f = 5})
Stack:
decl<[X -> l0]>
Heap:
[<l0, val> -> null]
>>======>
Control:
block(X.f = 5)
Stack:
decl<[X -> l0]>
Heap:
[<l1, f> -> null]
[<l0, val> -> l1]
>>======>
Stack:
Heap:
```

```
[<l1, f> -> 5]
[<l0, val> -> l1]
```

3. Recursive Procedure

The interpreter implements the recursive procedures. You can try it using *make example3*.

4. Object Creation

malloc(x) would allocate the variable x on the heap, and all fields (which ever appeared in the program text) are initialized as null. For instance,

```
> make example4
# Object Creation
var X; {malloc(X); X.f = 5}
Checking static scoping rules...
All static scoping checking passed.

Printing AST...
... (omit the printed AST)

Executing program...
Control:
var X; {malloc(X); X.f = 5}
Stack:
Heap:
>>======>
Control:
block({malloc(X); X.f = 5})
Stack:
decl<[X -> l0]>
Heap:
[<l0, val> -> null]
>>======>
Control:
block(X.f = 5)
Stack:
decl<[X -> l0]>
Heap:
[<l1, f> -> null]
[<l0, val> -> l1]
>>======>
Stack:
Heap:
```

```
[<l1, f> -> 5]
[<l0, val> -> l1]
```

5. Parallelism and Atomicity

In this interpreter, the Random module is used for the parallelism feature. You can try parallelism and atomicity using *make example5* and *make example6* respectively.

Contents

```
miniool                     - the interpreter

src                         - source files for miniool
 ├─── lexer.mll               - the lexer
 ├─── parser.mly              - the parser
 ├─── abstractSyntax.ml       - an OCaml type for the abstract syntax tree
 ├─── staticScoping.ml        - check the static scoping rules
 ├─── printAST.ml             - print the AST
 ├─── semanticDomain.ml       - an Ocaml type for the semantic domains
 ├─── printSemantics.ml       - pretty print the configurations (including
controls, stacks and heaps)
 ├─── operationalSemantics.ml - the transition relation according to the
operational semantics
 └─── miniool.ml              - the main program

bin                         - compiled files for miniool

makefile                    - the makefile
```