This assignment tests your understanding of the content of Modules 8 and 9. You will implement an object-oriented, concurrent language. You may use either OCaml or Smalltalk, at your preference. If you really want to, you can also do both, but you will only receive the higher of the two grades, no bonus points. This assignment has two parts: the object-oriented component, and the concurrent component, but they are two facets of the same language. Each part contains a complete description for both OCaml and Smalltalk. If you do this assignment in OCaml, you only need to read 1.1 and 2.1. If you do this assignment in Smalltalk, you only need to read 1.2 and 2.2. Of course, you may want to read both to decide which you’d prefer.

Submit all code via UWaterloo submit. If you are submitting OCaml code, submit to the a6ml assignment, like so:

submit cs442 a6ml .

If you are submitting Smalltalk code, submit to the a6st assignment, like so:

submit cs442 a6st .

This assignment is due on Friday, April 23rd, by 12PM NOON, NOT MIDNIGHT, Eastern time.

This assignment is worth 15% of your final grade, while previous assignments were worth 12%.

**Feather Pie**

In this assignment, you will implement a language called Feather Pie, or Fpie for short, which is an amalgam of language components from several sources:

- Numeric and boolean expressions from the Simple Imperative Language,
- if-then-else expressions (rather than statements) from functional languages,
- object orientation from Featherweight Java (which was one of the referenced sources of our Smalltalk OO semantics),
- concurrency from $\pi$-calculus, and
- Erlang-style process spawning.

The full semantics for the non-concurrent portion are at the end of this document. The semantics for both the non-concurrent and concurrent portions of the language are described informally here.

Like many of the languages we’ve looked at, the file syntax for Fpie is a list of declarations. As Fpie is object-oriented, those declarations are class declarations. Example Fpie files are provided. Read them to accustom yourself to the syntax.
In order to eliminate the extra baggage of a store and heap, you will be implementing immutable objects in a functional language. Since objects are immutable, object values are simply expressions. Execution proceeds as in \(\lambda\)-calculus with AOE steps. You may want to read the Featherweight Java paper, as you are now equipped to do so, but you don’t need to. Method calls are achieved through substitution; the major difference between Featherweight Java method calls and \(\lambda\)-calculus applications is simply method lookup.

An object value in Feather Pie (and Featherweight Java) is written like so:

```plaintext
new C(f1, f2, ..., fn)
```

Those familiar with Java may recognize this as Java’s object-creation syntax, but in Featherweight Java, this isn’t actually a call to any constructor. Instead, \(f1\) through \(fn\) are the values for each field of the class \(C\). If \(C\) has superclasses, then those fields come first; for instance, if the class \(Square\) defines a field \(width\), and the class \(ColoredSquare\) is a subclass of \(Square\) which defines the field \(color\), then \(f1\) is \(width\) and \(f2\) is \(color\). Method calls look similar to Java method calls. For instance, this example calls the “\(rotate\)” method on the object \(x\), with the argument 90:

```plaintext
x.rotate(90)
```

Since method calls use substitution, \(x\) must be substituted by an object literal for this to actually work, so the expression you actually evaluate will look something like this:

```plaintext
(new Cube(20, 0, 15)).rotate(90)
```

The receiver is named \(this\) (not \(self\) as in Smalltalk), so in the above method call, as well as the substitution for \(rotate\)’s argument, \(this\) should be substituted with \(new\ Cube(20, 0, 15)\).

Numeric expressions, boolean expressions, and if-then-else evaluate in the obvious way. Like in SIL, boolean expressions are only allowed in the condition of an \(if\) (there are no \(while\) loops), so there cannot be boolean values in normal expressions. To fit slightly better with Featherweight Java, if-then-else has a slightly different syntax than we’ve previously used: \(if\ (<\text{cond}>) \ <\text{exp}> \ else \ <\text{exp}>\).

Objects, numbers, and variables (which are channel names in \(\pi\)-calculus) are terminal values. \(\pi\)-calculus expressions themselves are also terminal values for the purpose of the object-oriented component of Fpie. Note that an object is only a terminal value if all of its fields have themselves been evaluated. So, for instance, this is not a terminal value:

```plaintext
new Cube(10+10, 0, 5+10)
```

Feather Pie is both an object-oriented language and a concurrent language, and includes \(\pi\)-calculus-like concurrency and messages. Because \(\pi\)-calculus is full of weird overlines and angle brackets, the syntax of \(\pi\)-calculus expressions has been simplified. Receiving and sending messages are written like so:

```plaintext
receive chr(msg) .
send chs(msg) . 0
```

The above example receives a message on the channel \(chr\) and then sends it along on channel \(chs\), and then evaluates to the number 0. Like in \(\pi\)-calculus, message receipt is implemented by substitution, so if the channel \(chr\) is sent the message 42, the next step will be:

```plaintext
send chs(42) . 0
```

Messages may be any value, so \(send\) allows any expression for \(msg\). The message to be sent should be reduced to a terminal value before \(send\) is allowed to execute.

Rather than \(\pi\)-calculus style concurrency and replication, Feather Pie uses a \(spawn\) expression, like Erlang. Execution starts with a single process, and the \(spawn\) expression creates new processes. The following expression spawns a new process, and then proceeds to the expression \(b\):

```plaintext
spawn Cl(ch) . b
```

\(Cl\) is the name of a class, and \(ch\) is a variable into which to place the name of a fresh channel. \(ch\) will be substituted with the channel name in \(b\), similar to \(receive\). In full, to spawn a process, follow these steps:

- Let \(chan\) refer to a fresh channel name.
- Let \(bc\) refer to \(b[chan/ch]\). That is, \(b\), with the variable named by \(spawn\) replaced with the variable \(chan\).
- Update the spawning process to `bc`.
- Create a new process with the expression `(new Cl(chan)).main()`. This expression calls the `main` method on a new instance of the class `Cl`, in which the first field is `chan`.

To make Feather Pie programs a bit more interesting, one other expression is added: `print x` evaluates to `x`, but additionally prints `x` and a newline to the screen. `x` must be a number to be printed.

\( \pi \)-calculus and Featherweight Java can have some unusual interactions. For instance, consider this expression:

\[(\text{new Cube}(20, 0, 15)).\text{rotate}(\text{receive ch}(x) . x)\]

Methods work by substitution, so this expression with only one receive may transform into an expression with more than one, or even zero, identical receives. Furthermore, because \( \pi \)-calculus is defined only to match message sends and receives at the outermost expression, the following expression cannot make progress, even if there are matching sends:

\[(\text{receive ch1}(x) . x) + (\text{receive ch2}(y) . y)\]

This makes programming in Feather Pie a bit strange, but it’s not your problem to deal with; just follow the semantics as defined, even if it would be quite a bit more convenient to be able to nest messages in this way.

One could write an implementation of Feather Pie with true parallelism, but you’re asked to implement something simpler. A program is a list of processes. In a step of execution, you should fully evaluate the non-concurrent part of each process. From that point, there are some number of \( \pi \)-calculus steps that might be possible: any number of receive-send pairs with the same channel, and any number of spawns. A random one of these steps is chosen and taken. Thus, two different executions of a Feather Pie program will undergo steps in a different order, and can undergo different steps entirely.

By convention, the starting point of a Feather Pie program is the expression `(new Main()).main()`, but you will write an evaluator which takes any expression as a starting point.
Feather Pie in OCaml

(If you are implementing your solution in Smalltalk, you may skip this section)

You are provided with a file fpie.ml, which implements a parser and some utilities for Fpie. It defines the following types:

(* A "randomizer" is a function that returns a number in the range 0...n-1 *)

```ocaml
type randomizer = int -> int
```

(* Expressions in Feather Pie *)

```ocaml
type expr =
  | (* SIL *)
  | Num of int
  | AddExpr of (expr * expr)
  | MulExpr of (expr * expr)
  | NegExpr of expr
  (* IfThenElse has this form:
     if <expr> <op> <expr> then <expr> else <expr>
     The condition is spread across two expressions and an operator,
     which must be one of "<", ">", or ". This is to remove booleans
     from the language while still having boolean expressions. *)
  | IfThenElse of (expr * string * expr * expr * expr)
  (* Featherweight Java *)
  | Var of string
  | FieldRead of expr * string (* Target, field name *)
  | MethodCall of expr * string * expr list (* Target, method name, arguments *)
  | ObjLiteral of string * expr list (* Class name and field values *)
  (* Pi calculus *)
  | Receive of string * string * expr (* Channel, variable to receive, following expr *)
  | Send of string * expr * expr (* Channel, value to send, following expr *)
  (* Erlang-ish *)
  | Spawn of string * string * expr (* Class name, variable to receive the channel name into,
    following expr *)
  (* Extras *)
  | Print of expr
```

(* A method consists of a name, a list of parameters, and a body expression *)

```ocaml
type methodDecl = string * string list * expr
```

(* The only declarations in Feather Pie are class declarations: Class name, superclass name, field list,
and method list *)

```ocaml
 type classDecl = string * string * string list * methodDecl list
```

(* A Feather Pie program is a list of class declarations *)

```ocaml
 type program = classDecl list
```

(* A substitution is a "from" and "to": string and expression *)

```ocaml
type substitution = string * expr
```

The most important type is `expr`, a variant type of all of the (many) expressions in Fpie. Note that because of the structure of `if`, you cannot explicitly take a step to replace the boolean subexpression with `true` or `false`; instead, if the left and right of the comparison expression are both values, you should perform the comparison and step to the then or else branch in one step.

An actual Fpie program is a list of class declarations. A class declaration is a tuple: class name, superclass name, field list, and method list. The fields are just string names. A method declaration is a triple: method name, parameter list (as strings), and the body.

Substitutions in Fpie are from strings (variable names) to full expressions.

In addition, `fpie.ml` defines the following parsing and utility functions (as well as many others):

(* Parse a single Feather Pie expression *)

```ocaml
parseFpieExpr : string -> expr option
```

(* Parse a Feather Pie program *)

```ocaml
parseFpie : string -> classDecl list option
```
(* Perform a substitution over a single name *)
subName : string -> substitution list -> expr

(* Substitute names in this expression *)
subExpr : expr -> substitution list -> expr

(* Substitute names in this expression list *)
subExprList : expr list -> substitution list -> expr list

(* Freshen a variable name *)
freshenName : string -> string

(* Get a fresh channel name *)
freshChannel : unit -> string

(* Get the class list that corresponds to a given class name, in this 
* declaration list *)
classList : program -> string -> classDecl list

(* Print an expression *)
printExpr : expr -> unit

The `parseFpieExpr` function parses an Fpie expression, and the `parseFpie` function parses a full Fpie program. Both return option types, where `None` indicates that the program did not parse.

The `sub*` functions perform substitution using a substitution list, over names, expressions, and expression lists.

The `freshenName` function creates a fresh variable name, based on the name given as an argument. The `freshChannel` function creates a fresh channel name. Its argument type is `unit`, so it’s called as `freshChannel ()`.

In class-based object orientation, it is often necessary to get a class and all of its superclasses, up to the root. For instance, to know all of the fields of a given object, you need to know the fields defined in its entire class hierarchy. Similarly, to look up a method, you may need to look in superclasses. The `classList` function, given an Fpie program and a class name, generates the list of class declarations for a class and all of its superclasses, from the top down to the specific class. For instance, if `Square` is a subclass of `Rectangle`, and `Rectangle` is a subclass of `Object`, then `classList program "Square"` will return the list `[Object; Rectangle; Square]`. It isn’t necessary to explicitly define `Object` (it has no fields or methods), so the list is expected to simply stop there. If `Object` isn’t defined, then the above will return `[Rectangle; Square]`.

The `printExpr` function prints an expression (without a newline).
Feather Pie in Smalltalk

(If you are implementing your solution in OCaml, you may skip this section)

You are provided with a file `fpie.st`, which implements a parser and some utilities for Fpie. It defines the following classes and class extensions:

```smalltalk
Object subclass: FpieExpr [  
  isNum.  
  isAddExpr.  
  isMulExpr.  
  isNegExpr.  
  isIf.  
  isVar.  
  isFieldRead.  
  isMethodCall.  
  isObjLiteral.  
  isReceive.  
  isSend.  
  isSpawn.  
  isPrint.  
  isValue.  
  printString.  
  " Implemented in all subclasses: "  
  dup.  
  substitute: slist.  
  displayString.  ]

Object subclass: FpieSubstitutionList [  
  FpieSubstitutionList class >> withFrom: sfrom to: sto next: snext.  ]

Object subclass: FpieClassDecl [  
  FpieClassDecl class >> withName: sname superklass: ssuperklass fields: sfields methods: smethods.  
  name.  
  superklass.  
  fields.  
  methods.  ]

Object subclass: FpieMethodDecl [  
  FpieMethodDecl class >> withName: sname params: sparams body: sbody.  
  name.  
  params.  
  body.  ]

String extend [  
  fpieClassList: program.  ]

FpieExpr subclass: FpieNum [  
  FpieNum class >> withValue: aNumber.  
  value.  ]

FpieExpr subclass: FpieAddExpr [  
  FpieAddExpr class >> withLeft: sleft right: sright.  
  left.  
  right.  ]

FpieExpr subclass: FpieMulExpr [  
  FpieMulExpr class >> withLeft: sleft right: sright.  
  left.  
  right.  ]
```

CS442: A6
FpieExpr subclass: FpieNegExpr [  
FpieNegExpr class >> withSubexpr: ssubexpr.  
subexpr.  
]

FpieExpr subclass: FpieIf [  
FpieIf class >> withLeft: sleft op: sop right: sright thenExpr: sthenExpr elseExpr: selseExpr.  
left.  
op.  
right.  
thenExpr.  
elseExpr.  
]

FpieExpr subclass: FpieVar [  
FpieVar class >> withName: sname.  
name.  
]

FpieExpr subclass: FpieFieldRead [  
FpieFieldRead class >> withTarget: starget fieldName: sfieldName .  
target.  
fieldName.  
]

FpieExpr subclass: FpieMethodCall [  
FpieMethodCall class >> withTarget: starget methodName: smethodName args: sargs.  
target.  
methodName.  
args.  
]

FpieExpr subclass: FpieObjLiteral [  
FpieObjLiteral class >> withKlass: sklass fields: sfields.  
klass.  
fIELDS.  
]

FpieExpr subclass: FpieReceive [  
FpieReceive class >> withChannel: schannel varName: svarName body: sbody.  
channel.  
varName.  
body.  
]

FpieExpr subclass: FpieSend [  
FpieSend class >> withChannel: schannel message: smessage body: sbody.  
channel.  
mESSAGE.  
body.  
]

FpieExpr subclass: FpieSpawn [  
FpieSpawn class >> withKlass: sklass channel: schannel body: sbody.  
klass.  
channel.  
body.  
]

FpieExpr subclass: FpiePrint [  
FpiePrint class >> withSubexpr: ssubexpr.  
subexpr.  
]

Object subclass: FpieParser [  
FpieParser class >> new: text.  
FpieParser class >> parse: text.  
FpieParser class >> parseFile: text.  
parseFile.  
parse.  
]
All Fpie expressions are of subclasses of \texttt{FpieExpr}. Each of its \texttt{is*} methods returns \texttt{true} if the particular expression is of the named subclass. In addition, the \texttt{isValue} method returns \texttt{true} for all expressions which are terminal values. Each expression can be duplicated with the \texttt{dup} method, and can have substitutions applied with the \texttt{substitute:} method, which takes a substitution list as an argument.

Otherwise, the expression classes are simply containers; their public methods simply return the subexpression or substring. The word “class” is often replaced with “klass”, to avoid any conflicts with Smalltalk; this is commonly done when implementing an object-oriented language in an object-oriented language. Most values contained in expressions are also expressions, with the following exceptions:

- \texttt{FpieNum\textgreater\textgreater value} is a number.
- \texttt{FpieIf\textgreater\textgreater op} is a string, one of ‘<’, ‘=’, or ‘>’.
- \texttt{FpieVar\textgreater\textgreater name} is a string.
- \texttt{FpieFieldRead\textgreater\textgreater fieldName} and \texttt{FpieMethodCall\textgreater\textgreater methodName} are strings.
- \texttt{FpieMethodCall\textgreater\textgreater args} and \texttt{FpieObjLiteral\textgreater\textgreater fields} are arrays of expressions.
- \texttt{FpieObjLiteral\textgreater\textgreater klass} and \texttt{FpieSpawn\textgreater\textgreater klass} are strings, the name of the class to which the object belongs or which is being spawned.
- \texttt{FpieReceive\textgreater\textgreater channel}, \texttt{FpieSend\textgreater\textgreater channel}, and \texttt{FpieSend\textgreater\textgreater channel} are strings, channel names.
- \texttt{FpieReceive\textgreater\textgreater varName} is a string, the name of the variable to substitute with the value received.

Note that because of the structure of \texttt{if}, you cannot explicitly take a step to replace the boolean subexpression with true or false; instead, if the left and right of the comparison expression are both values, you should perform the comparison and step to the then or else branch in one step.

An actual Fpie program is an array of class declarations. A class declaration is an object of the class \texttt{FpieClassDecl}. \texttt{FpieClassDecl\textgreater\textgreater name} is the name of the class, a string. \texttt{FpieClassDecl\textgreater\textgreater superklass} is the name of the superclass, a string (not an \texttt{FpieClassDecl}). \texttt{FpieClassDecl\textgreater\textgreater fields} is the fields defined in the class, an array of strings. \texttt{FpieClassDecl\textgreater\textgreater methods} is the methods defined in the class, an array of method declarations.

A method declaration is an object of the class \texttt{FpieMethodDecl}. \texttt{FpieMethodDecl\textgreater\textgreater name} is the name of the method, a string. \texttt{FpieMethodDecl\textgreater\textgreater params} is the names of the parameters, an array of strings. \texttt{FpieMethodDecl\textgreater\textgreater body} is the body of the method, an expression.

Substitutions in Fpie are from strings (variable names) to full expressions. The \texttt{FpieSubstitutionList} class represents substitution lists. It forms a list, so you can build substitution lists iteratively like so:

\begin{verbatim}
sl := FpieSubstitutionList withFrom: 'foo' to: (FpieNum withValue: 42) next: nil.
sl := FpieSubstitutionList withFrom: 'this' to: target next: sl.
\end{verbatim}

The \texttt{FpieParser} class is an Fpie parser. It can be used in two ways: either by creating an object of the \texttt{FpieParser} class and then calling \texttt{parse} or \texttt{parseFile} on it, or by calling \texttt{parse:} or \texttt{parseFile:} directly on the class. The \texttt{parse} and \texttt{parse:} methods parse expressions. The \texttt{parseFile} and \texttt{parseFile:} methods parse full Fpie programs.

In class-based object orientation, it is often necessary to get a class and all of its superclasses, up to the root. For instance, to know all of the fields of a given object, you need to know the fields defined in its entire class hierarchy. Similarly, to look up a method, you may need to look in superclasses. The \texttt{String} class is thus extended with a \texttt{fpieClassList:} method, which generates the list of class declarations for a class and all of its superclasses, from the top down to the specific class. The class to look up is the string itself, so \texttt{fpieClassList:} is used like ‘\texttt{Rectangle}’ \texttt{fpieClassList: program}. For instance, if \texttt{Square} is a subclass of \texttt{Rectangle}, and \texttt{Rectangle} is a subclass of \texttt{Object}, then ‘\texttt{Square}’ \texttt{classList: program} will return the list \texttt{OrderedCollection (Object Rectangle Square ).} It isn’t necessary to explicitly define \texttt{Object} (it has no fields or methods), so the list is expected to simply stop there. If \texttt{Object} isn’t defined, then the above will return \texttt{OrderedCollection (Rectangle Square ).}
1 Object Orientation

Your first task is to implement an evaluator for the non-concurrent portion of Feather Pie. It should follow precisely the semantics defined at the end of this document. Bear in mind that with the exception of the argument to `send`, Feather Pie never reduces anything inside of a concurrent expression until the appropriate concurrent action has been taken.

You are not required to perform any error checking for this entire assignment. A program which attempts, for example, to call a method with the incorrect number of arguments, or to add two values which are not numbers, may fail in any way, or simply get stuck, at your preference.

1.1 OO in OCaml

(If you are implementing your solution in Smalltalk, you may skip this section)

Create a file, `a6.ml`, which implements at least the following function:

\[
\text{eval : expr -> program -> expr}
\]

Given an expression and a program (i.e., the list of class declarations), `eval` fully evaluates it, returning the resulting expression. That is, it performs individual evaluation steps until no further steps can be taken (excluding concurrent steps). Of course, if the expression results in an infinite reduction, `eval` should similarly evaluate infinitely.

Because you’ve written parts of this several times by now, you are provided with a template, `a6template.ml`, as a suggested starting point, with the simple numeric parts, if-then-else, and print filled in. With it, you can focus instead on object orientation. You are not required to use it, but might find it useful.

An example frontend is provided, `frontendoo.ml`. To use it, run it with a non-concurrent Fpie program as an argument. For example:

```bash
$ ocamlfind ocamlc -package core -linkpkg -thread fpie.ml a6.ml frontendoo.ml \
-0 frontendoo
$ ./frontendoo facobj.fpie
120
```

Hints

Most of this is just matching lists: given a class and a list of field values, find the value of a given field; given a class and a method name, find the method. You will want to use `classList` as a starting point to generate full field lists and method lists. In short, explicitly implement the `fields` and `method` functions defined in the semantics.

You’re only required to write a full evaluator (`eval`), not a function to take individual steps of reduction. Even if you use a step function (as the template does), it is thus not necessary that that step function actually match one-to-one with the defined steps of reduction; it can take multiple steps if that’s easier to define. Or, the `eval` function can fully evaluate “in one step”.

CS442: A6
1.2 OO in Smalltalk

(If you are implementing your solution in OCaml, you may skip this section)

Create a file, a6.st, which implements at least the following class:

```
Object subclass: Fpie |
    Fpie class >> withExpr: sexpr program: sprogram.
    eval.
|
```

An object of the Fpie class is a Feather Pie evaluator. It is created with an expression and a program (i.e., array of class declarations). Fpie>>eval fully evaluates the expression, returning the resulting expression. That is, it performs individual evaluation steps until no further steps can be taken (excluding concurrent steps). Of course, if the expression results in an infinite reduction, eval should similarly evaluate infinitely. eval only needs to work once, and you are allowed to mutate the expression; of course, if you mutate the expression, you should be careful about multiple references to the same expression, as in previous evaluators.

Because you’ve written parts of this several times by now, you are provided with two templates, a6template1.st a6template2.st, as suggested starting points, with the simple numeric parts, if-then-else, and print filled in. With one of them, you can focus instead on object orientation. You are not required to use a template, but might find it useful. a6template1.st works by extending each FpieExpr subclass with a step: method, while a6template2.st puts the step: method in Fpie directly. Either style works, so you may use whichever template you prefer.

An example frontend is provided, frontendoo.st. To use it, run it with a non-concurrent Fpie program as an argument after -a. For example:

```
$ gst fpie.st a6.st frontendoo.st -a facobj.fpie
120
```

Hints

Most of this is just matching lists: given a class and a list of field values, find the value of a given field; given a class and a method name, find the method. You will want to use fpieClassList: as a starting point to generate full field lists and method lists. In short, explicitly implement the fields and method functions defined in the semantics.

You’re only required to write a full evaluator (eval), not a function to take individual steps of reduction. Even if you use a step method (as the templates do), it is thus not necessary that that step method actually match one-to-one with the defined steps of reduction; it can take multiple steps if that’s easier to define. Or, the eval method can fully evaluate “in one step”.

2 Concurrency

A concurrent Feather Pie program is a list (or array) of Feather Pie expressions, each of which corresponds to a single process. To run a full Feather Pie program is to repeatedly evaluate the non-concurrent part of each process, then perform a concurrent step, until no more concurrent steps can be performed.

\(\pi\)-calculus defines its steps in terms of structural congruence, and reordering the processes to match one of the cases. While it’s possible to implement Feather Pie concurrency in this way, it will be easier to simply search for concurrent steps that can be taken, and perform one.

There are two forms of concurrent steps: a send-receive pair, and a spawn. You will need to perform a nested loop over your processes to find send-receive pairs (remember that a given send might match two receives, or vice-versa!).

To simulate the non-determinism of concurrency, your program runner will take a randomizer as an argument. In your implementation, use this randomizer, and do not use any random number generator built into the language. During testing, we will swap it out with decidedly non-random generators, so it’s important that you don’t have an additional source of randomness. You should give equal probability to each concurrent step which can be taken.

2.1 Concurrency in OCaml

(If you are implementing your solution in Smalltalk, you may skip this section)

Extend a6.ml, adding the following function:

```
run : expr -> program -> randomizer -> expr list
```

The run function runs a Feather Pie program until no further concurrent steps can be taken (possibly forever), and returns the final state of each process. The exact behavior and return is non-deterministic, but constrained by the concurrent semantics of Feather Pie.

The randomizer function is given as an argument. Given \(n\) as an argument, it returns a number between 0 and \(n - 1\). So, for instance, if \(n\) is the length of an array, it returns a valid index to that array.

Run a Feather Pie program by repeatedly:

- fully reducing (with eval) every current process,
- enumerating all possible concurrent steps,
- if there are no concurrent steps, halting, or
- choosing a concurrent step with the randomizer and enacting it.

Because of the randomness, while the behavior of any given process is predictable for any step, the behavior of the whole program is not.

An example frontend is provided, frontend.ml. To use it, run it with an Fpie program as an argument. For example:

```
$ ocamlfind ocamlc -package core -linkpkg -thread fpie.ml a6.ml frontend.ml \   -o frontend
$ ./frontend facproc.fpie
```

It expects every process to terminate by resolving to a number, and will warn about any processes that evaluate to anything other than a number. Of course, if a process is not supposed to terminate in this way, then the warning is spurious.
Hints

On this rare occasion, I recommend imperative code in OCaml. The best way to store the current processes is an array; simply mutate the array when you evaluate a process. The best way to find all the possible steps is to iterate over that array. When you find a receive, you should iterate in a nested loop to find matching sends (or vice-versa, equivalently). It is possible to do all of this without mutation, but the alchemy needed to keep all your lists as expected, pull out specific processes, execute a randomly selected one or pair of them, etc, is far easier done with mutation, even in a functional language.

You will probably want a variant type to distinguish the two kinds of concurrent steps that can be taken (message sending and process spawning).

If you do use a mutable array, you may want to know about a few useful functions:

- `Array.of_list` converts a list to an array.
- `Array.to_list` converts an array to a list.
- `Array.append` concatenates two arrays, as `List.append` does for lists.
2.2 Concurrency in Smalltalk

(If you are implementing your solution in OCaml, you may skip this section)

Extend a6.st, adding the following method to Fpie:

```ocaml
Object subclass: Fpie [  
...  
run: rand.  
]
```

The run method runs a Feather Pie program until no further concurrent steps can be taken (possibly forever), and returns the final state of each process, as an array of expressions. The execution should start with a single process, defined by the expression given when the Fpie was constructed. The exact behavior and return is non-deterministic, but constrained by the concurrent semantics of Feather Pie.

A randomizer object is given as an argument. To generate a random number, use rand between: n and: m. It returns a number between n and m. So, for instance, if n is the length of an array, rand between: 1 and: n returns a valid index into that array.

Run a Feather Pie program by repeatedly:

- fully reducing (with eval) every current process,
- enumerating all possible concurrent steps,
- if there are no concurrent steps, halting, or
- choosing a concurrent step with the randomizer and enacting it.

Because of the randomness, while the behavior of any given process is predictable for any step, the behavior of the whole program is not.

An example frontend is provided, frontend.st. To use it, run it with an Fpie program as an argument after -a. For example:

```
$ gst fpie.st a6.st frontend.st -a facproc.fpie
120
```

It expects every process to terminate by resolving to a number, and will warn about any processes that evaluate to anything other than a number. Of course, if a process is not supposed to terminate in this way, then the warning is spurious.

Hints

Store the current processes is an array, and mutate the array when you evaluate a process. To find all the possible steps, simply iterate over that array. When you find a receive, you should iterate in a nested loop to find matching sends (or vice-versa, equivalently).

You will probably want a new class with subclasses to distinguish the two kinds of concurrent steps that can be taken (message sending and process spawning). You can even implement the actual concurrent step in that class instead of run:, to make run: a bit less expansive.

No helper function is provided to create fresh names. However, fresh names are fairly easy to create: since we always create processes but never destroy them, and only need fresh names when we create processes, the current number of processes is a fresh number. So, you can create a fresh name simply by converting the number of processes to a string, and boxing that in a FpieVar. To avoid confusion, you should probably suffix that fresh number to some obvious name prefix, such as ‘ch_’, thus creating channel names such as ‘ch_1’, ‘ch_2’. Make sure that the concatenation does not result in a valid variable name, so that you don’t accidentally conflict with existing variables; the parser does not accept ‘_’ in variable names, so including an underscore is sufficient to assure your names are unique.
Examples

Several example programs are provided on the course web site. Some are non-concurrent, some are concurrent. The non-concurrent examples should work with either `eval` or `run`, while the concurrent examples will only work with `run`. Most have configurable parameters, but because Fpie has no way of taking arguments, the parameters can only be configured by editing the example programs.

`facobj.fpie` is a simple, non-concurrent factorial calculator. It is preset to compute $5!$.

```
$ ./fpie facobj.fpie
120
```

`facproc.fpie` is similar to `facobj.fpie`, but implemented as a sequence of communicating processes.

```
$ ./fpie facproc.fpie
120
```

`fibobj.fpie` is a $O(2^n)$, non-concurrent Fibonacci sequence generator. It is preset to compute the 10th Fibonacci number.

```
$ ./fpie fibobj.fpie
55
```

`fibproc.fpie` is similar to `fibobj.fpie`, but implemented as a tree of communicating processes.

```
$ ./fpie fibproc.fpie
55
```

`fibref.fpie` is also a Fibonacci sequence generator, but uses references simulated by processes to implement the standard $O(n)$ dynamic programming solution to Fibonacci sequence generation. It is preset to compute the 42nd Fibonacci number.

```
$ ./fpie fibref.fpie
267914296
```

`factors.fpie` concurrently, non-deterministically finds the factors of a number. Note that this is simply the list of factors, *not* the prime factorization. It is preset to compute the factors of 12.

```
$ ./fpie factors.fpie
2
3
4
6
```

It is non-deterministic, so multiple runs may not yield the factors in the same order; they should always yield the same factors, however:

```
$ ./fpie factors.fpie
3
4
2
6
```
Because of the way that processes are generated, a random selection of steps is more likely to generate the factors in order than out of order, so you may need to run it several times to reveal the non-determinism.

pingpong.fpie creates a number of pairs of pinging and ponging processes, each of which outputs a countdown as they ping. It is preset to create three processes, each of which pings four times (from 3 down to 0). The pong processes repeat forever, so if you use the sample frontend, it should warn that they may not have terminated. Because each pair acts independently, there are many possible outputs. This is one:

$ ./fpie pingpong.fpie
3
2
1
0
3
3
2
2
1
1
0
0

WARNING: Possibly non-terminated process(es):
(send ch_3(0) . ((new Ponger(ch_3)).go(ch_3)))
(send ch_5(0) . ((new Ponger(ch_5)).go(ch_5)))
(send ch_8(0) . ((new Ponger(ch_8)).go(ch_8)))

compete.fpie is similar to pingpong.fpie, but many ping processes compete for a single channel to communicate with a single pong process, and thus can interfere with one another as they accidentally receive each other’s pings, thinking they’re pongs. Because of this competition, there are many possible outputs. This is one:

$ ./fpie compete.fpie
3
2
0
1
0

WARNING: Possibly non-terminated process(es):
(receive ch_1(ct) . ((new Ponger(ch_1)).pong((print ct),ch_1)))
Semantics

Let the metavariables $E$, $V$, $N$, $C$, $x$, $f$, and $m$ range over expressions, terminal values (including π-calculus expressions), numbers, class names, variable names, field names, and method names, respectively. Assume that $\sigma$, a mapping of class names to class declarations, is provided externally; since it never changes, it will not be made part of the reduction relation. Assume that the function $\textit{fields}$ maps a class to a list of fields in that class and all superclasses, in the order that they should appear in an object literal. Assume that $\textit{method}$ is defined as in Module 8. Then the following is a formal semantics for the non-concurrent subset of Feather Pie:

\[
\begin{align*}
\text{IntOpLeft} & \quad \frac{\text{op} \in \{+, \ast\}}{E_1 \to E'_1} \\
\quad & \quad \frac{E_1 \text{ op } E_2 \to E'_1 \text{ op } E_2}{E_1 \to E'_1}
\end{align*}
\]

\[
\begin{align*}
\text{IntOpRight} & \quad \frac{\text{op} \in \{+, \ast\}}{E \to E'} \\
\quad & \quad \frac{V \text{ op } E \to V \text{ op } E'}{V \to V'}
\end{align*}
\]

\[
\begin{align*}
\text{Add} & \quad \frac{N_1 + N_2 = N_3}{N_1 + N_2 \to N_3}
\end{align*}
\]

\[
\begin{align*}
\text{Mul} & \quad \frac{N_1 \ast N_2 = N_3}{N_1 \ast N_2 \to N_3}
\end{align*}
\]

\[
\begin{align*}
\text{NegStep} & \quad \frac{E \to E'}{-E \to -E'}
\end{align*}
\]

\[
\begin{align*}
\text{Neg} & \quad \frac{N' = -N}{-N \to N'}
\end{align*}
\]

\[
\begin{align*}
\text{BoolOpLeft} & \quad \frac{\text{op} \in \{>, <, =\}}{E_1, \to E'_1} \\
\quad & \quad \frac{E_1 \text{ op } M_2 \to E'_1 \text{ op } E_2}{E_1 \to E'_1}
\end{align*}
\]

\[
\begin{align*}
\text{BoolOpRight} & \quad \frac{\text{op} \in \{>, <, =\}}{E \to E'} \\
\quad & \quad \frac{V \text{ op } E \to V \text{ op } E'}{V \to V'}
\end{align*}
\]

\[
\begin{align*}
\text{GtTrue} & \quad \frac{N_1 > N_2}{N_1 > N_2 \to \text{true}}
\end{align*}
\]

\[
\begin{align*}
\text{GtFalse} & \quad \frac{N_1 \leq N_2}{N_1 > N_2 \to \text{false}}
\end{align*}
\]

\[
\begin{align*}
\text{LtTrue} & \quad \frac{N_1 < N_2}{N_1 < N_2 \to \text{true}}
\end{align*}
\]

\[
\begin{align*}
\text{LtFalse} & \quad \frac{N_1 \geq N_2}{N_1 < N_2 \to \text{false}}
\end{align*}
\]

\[
\begin{align*}
\text{EqTrue} & \quad \frac{N_1 = N_2}{N_1 = N_2 \to \text{true}}
\end{align*}
\]

\[
\begin{align*}
\text{EqFalse} & \quad \frac{N_1 \neq N_2}{N_1 = N_2 \to \text{false}}
\end{align*}
\]

\[
\begin{align*}
\text{IfCond} & \quad \frac{E_1 \to E'_1}{\text{if } (E_1) E_2 \text{ else } E_3 \to \text{if } (E'_1) E_2 \text{ else } E_3}
\end{align*}
\]

\[
\begin{align*}
\text{IfTrue} & \quad \frac{\text{if } (\text{true}) E_1 \text{ else } E_2 \to E_1}{\text{if } (\text{true}) E_1 \text{ else } E_2 \to E_1}
\end{align*}
\]

\[
\begin{align*}
\text{IfFalse} & \quad \frac{\text{if } (\text{false}) E_1 \text{ else } E_2 \to E_2}{\text{if } (\text{false}) E_1 \text{ else } E_2 \to E_2}
\end{align*}
\]

\[
\begin{align*}
\text{PrintStep} & \quad \frac{E \to E'}{\text{print } E \to \text{print } E'}
\end{align*}
\]

\[
\begin{align*}
\text{Print} & \quad \frac{\text{print } N \to N'}{(\text{additionally, } N \text{ and a newline are printed to the screen})}
\end{align*}
\]

\[
\begin{align*}
\text{FieldReadTarget} & \quad \frac{E \to E'}{E.f \to E'.f}
\end{align*}
\]
**FieldRead** \(\text{fields}(\sigma, C) = f_1; f_2; \ldots; f_i; \ldots f_n; (\text{new } C(V_1, V_2, \ldots, V_n)), f_i \rightarrow V_i\)

**MethodTarget** \(E_0 \rightarrow E'_0\) 
\(E_0.m(E_1, E_2, \ldots, E_n) \rightarrow E'_0.m(E_1, E_2, \ldots, E_n)\)

**MethodArg** \(E_1 \rightarrow E'_1\) 
\(V_0.m(V_1, V_2, \ldots, V_n, E_1, E_2, \ldots, E_m) \rightarrow V_0.m(V_1, V_2, \ldots, V_n, E'_1, E_2, \ldots, E_m)\)

**Call** \(V_0 = \text{new } C(V_{1.1}, V_{1.2}, \ldots, V_{1.n})\) 
\(\text{method}(\sigma, C, m) = m(x_1, x_2, \ldots x_k)\{E\}\) 
\(V_0.m(V_{2.1}, V_{2.2}, \ldots, V_{2.k}) \rightarrow E[V_{2.1}/x_1][V_{2.2}/x_2] \cdots [V_{2.k}/x_k][V_0/\text{this}]\)

**ObjLiteralArg** \(E_1 \rightarrow E'_1\) 
\(\text{new } C(V_1, V_2, \ldots, V_n, E_1, E_2, \ldots, E_m) \rightarrow \text{new } C(V_1, V_2, \ldots V_n, E'_1, E_2, \ldots, E_m)\)

**SendStep** \(E_1 \rightarrow E'_1\) 
\(\text{send } x(E_1).E_2 \rightarrow \text{send } x(E'_1).E_2\)

**Concurrence.** The concurrent semantics of Feather Pie are the concurrent semantics of the subset of π-calculus that is mirrored in Feather Pie, with the exception of the unique spawn expression, the semantics of which follow:

\[
\text{Spawn} \quad x_2 \text{ is a fresh name} \quad \text{spawn } C(x_1).E | Q \rightarrow E[x_2/x_1] | (\text{new } C(x_2)).\text{main()} | Q
\]

**Rights**

Copyright © 2020, 2021 University of Waterloo.
This assignment is intended for CS442 at University of Waterloo.
Any other use requires permission from the above named copyright holder(s).