Comp Sci 321: Programming Languages — Homework 5

1 Languages

Start with the interpreter that uses deferred substitution for this assignment (see the lecture notes).

Here is the language that your interpreter should accept:

```
FAE ::= {+ <FAE> <FAE>}
      | {- FAE FAE}
      | <num>
      | <id>
      | {if0 <FAE> <FAE> <FAE>}
      | {fun {<id>} <FAE>}
      | {<FAE> <FAE>} ;; application expressions
```

Your parser, however, should accept the following, extended language:

```
FWAE ::= {+ <FWAE> <FWAE>}
       | {- <FWAE> <FWAE>}
       | <num>
       | {with {<id> <FWAE>} <FWAE>}
       | <id>
       | {if0 <FWAE> <FWAE> <FWAE>}
       | {fun {<id>*} <FWAE>}
       | {<FWAE> <FWAE>*} ;; application expressions
```

Please use the following datatype definitions for these languages:

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?) (rhs FAE?)]
  [sub (lhs FAE?) (rhs FAE?)]
  [id (name symbol?)]
  [if0 (test FAE?) (then FAE?) (else FAE?)]
  [fun (param symbol?) (body FAE?)]
  [app (fun FAE?) (arg FAE?)])
```

```
(define-type FWAE
  [W-num (n number?)]
  [W-add (lhs FWAE?)
         (rhs FWAE?)]
  [W-sub (lhs FWAE?)
         (rhs FWAE?)]
  [W-with (name symbol?)
          (named-expr FWAE?)
          (body FWAE?)]
  [W-id (name symbol?)]
  [W-if0 (tst FWAE?)
         (thn FWAE?)
         (els FWAE?)]
  [W-fun (params (listof symbol?))
         (body FWAE?)]
  [W-app (fun-expr  FWAE?)
         (arg-exprs (listof FWAE?))])
```

2 Compilation

The `FWAE` language has two forms that are not directly available in the `FAE` language: `with`, and multiple-argument functions. Both of these, however, can be translated into the simpler forms offered by `FAE`.

Implement a `compile` function that translates between the `FWAE` data structures produced by your parser to the `FAE` data structures accepted by your interpreter.

```
; compile : FWAE? -> FAE?
```

Compile `with` according to this rule:

```
{with {x E1} E2} ⇒ {{fun {x} E2} E1}
```

Compile multiple-argument functions according to these rules:

```
{E1 E2 E3 E4*} ⇒ {{E1 E2} E3 E4*}
{fun {ID1 ID2 ID3*} E} ⇒ {fun {ID1} {fun {ID2 ID3*} E}}
```

These are example uses of the multiple-argument function rules:

```
{fun {a b} {+ a b}} ⇒ {fun {a} {fun {b} {+ a b}}}
{f x y z} ⇒ {{f x y} z} ⇒ {{{f x} y} z}
{fun {a b c} {- a {+ b c}}}
    ⇒ {fun {a} {fun {b c} {- a {+ b c}}}}
    ⇒ {fun {a} {fun {b} {fun {c} {- a {+ b c}}}}}
```

Nullary functions (i.e., functions with no arguments) and application expressions that supply no arguments are both errors. Your compiler must raise errors in both cases, with messages containing the strings `"nullary function"` and `"nullary application"`, respectively.

Also note that these rules are intended to be schematic, not precise descriptions of how to implement your compiler (but the examples form the basis of test cases for your compiler).

3 Conditionals, revisited

Add `if0` to your interpreter with the same syntax as homework 4 but, unlike homework 4, you must be careful that the test position can be any value at all, not just a number. If the test position is a closure, it is treated the same as any other non-`0` value (i.e., it is not an error; it takes the false branch).

4 Errors

There are three different kinds of errors that can occur (at run-time) in this language and for each error in the input program, your interpreter must signal an error that includes one of the following phrases:

- `"free identifier"`
- `"expected function"`
- `"expected number"`

Note that each operation in the language that evaluates its sub-expressions immediately must evaluate them from left to right and must evaluate all of them before checking any error conditions. For example:

```
(test/exn (interp-expr (compile (parse `{+ {fun {x} x}
                                           {1 2}})))
          "expected function")
```

is a valid test case; the application error must be signaled first even though the add expression also gets incorrect inputs.

In addition, your compiler must detect the following two kinds of errors, described above:

- `"nullary function"`
- `"nullary application"`

5 Programming in **FWAE**

Implement the following two functions in the `FWAE` language. You may not add any extensions to `FWAE`, e.g., you may not build in recursion into the language.

- `factorial`: which takes a positive integer as argument (you can assume it is given that, and don't need to do any checking) and return its factorial, following the standard definition.
- `prime?`: which takes a positive integer as argument (again, no need for checking) and returns `0` if the number is prime, and `1` if not. For the purposes of this function, you may count `1` as prime if you want (or not, it's up to you; we won't test it). **Hint**: No need to be clever and/or efficient. Plain old trial division is fine.

Hint: you are likely to need helper functions along the way. Helper functions are totally cool. Just be sure that their definitions are in scope when you define `factorial` and `prime?`.

Each function should be implemented as an S-expression that can be parsed by your parser. Each such S-expression should be bound by a PLAI-level definition; see below for details.

6 Handin instructions

The final program you hand in should use the precise `define-type` definitions for `FAE` and `FWAE` provided above.

Provide a definition of `interp-expr : FAE -> number or 'function`, as above. Note that `interp-expr` is a wrapper function around your main, recursive interpreter. It translates whatever you decide to use for your representation of values into either a number, or the symbol `'function`. For example, these test cases should pass:

```
(test (interp-expr (num 10)) 10)
(test (interp-expr (fun 'x (id 'x))) 'function)
```

Provide a definition of `compile : FWAE -> FAE`, as above.
Provide a definition of `parse : s-expression -> FWAE`, as above.
Bind your definitions of `factorial` and `prime?` to PLAI-level definitions of the same name, e.g.:

```
(define factorial `{with {any-helpers-you-may-need ...} {fun {n} ...}})
(define prime?    `{with {any-helpers-you-may-need ...} {fun {n} ...}})
```

Note that these should each be complete `FWAE` programs, with all necessary helper functions in scope; if you wish, use quasiquote and unquote (see the parsing lecture notes for more on quasiquote and unquote) to avoid duplicating code.

Have the 8 rules from the Provost's website (see homework 1 for more details).
Submit your code via Canvas.
Your submission must include your test cases; submissions without test cases will get a grade of 0.