CS251 Programming Languages **Handout # 19**
Prof. Lyn Turbak **February 23, 2004**
Wellesley College

Problem Set 4
Due: 6pm Saturday, February 28

Scheduling:
This assignment is due at 6pm on Sat. Feb. 28. This assignment will be followed by Exam 1, a take-home exam that will be distributed on Fri. Feb 27 and will be due at midnight on Sun. Mar 7.

Overview:
The purpose of this assignment is to give you practice with reasoning about the BINDEX language and writing OCAML programs that manipulate BINDEX programs.

Reading:

- Handout #18: Intex: An Introduction to Program Manipulation
- Handout #20: Bindex: A Simple Language with Names

Working Together:
Reminder: if you worked with a partner on a previous problem set and want to work with a partner on this assignment, you must choose a different partner.

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 6pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your pencil-and-paper solution to Problem 1;
3. your final versions of `Sigmex.ml`, `SigmexEnvInterp.ml`, and `SigmexSubstInterp.ml` for Problem 2;
4. your final version of `Simp.ml` for Problem 3.

Each team should also submit a single softcopy (consisting of your final `ps4` directory) to the drop directory `~cs251/drop/p3/`*username*, where *username* is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs251
cp -R ps4 ~cs251/drop/ps4/username/
```

Problem 0: Studying BINDEX

All of the problems on this problem set involve the BINDEX language discussed in class or extensions to this language. Before attempting the problems, you should study the code for the implementation of the BINDEX language, which can be found in `~/cs251/bindex` after you perform `cvs update -d`. There are three files to study: `Bindex.ml`, `BindexEnvInterp.ml`, and `BindexSubstInterp.ml`.

To use any of the functions defined within files in the `bindex` directory, you should first execute the following directives in OCAML:

```
#cd "/students/username/cs251/bindex"
#use "load-bindex.ml"
```

Having done this, you can now experiment with any functions in the BINDEX interpreter. For example:

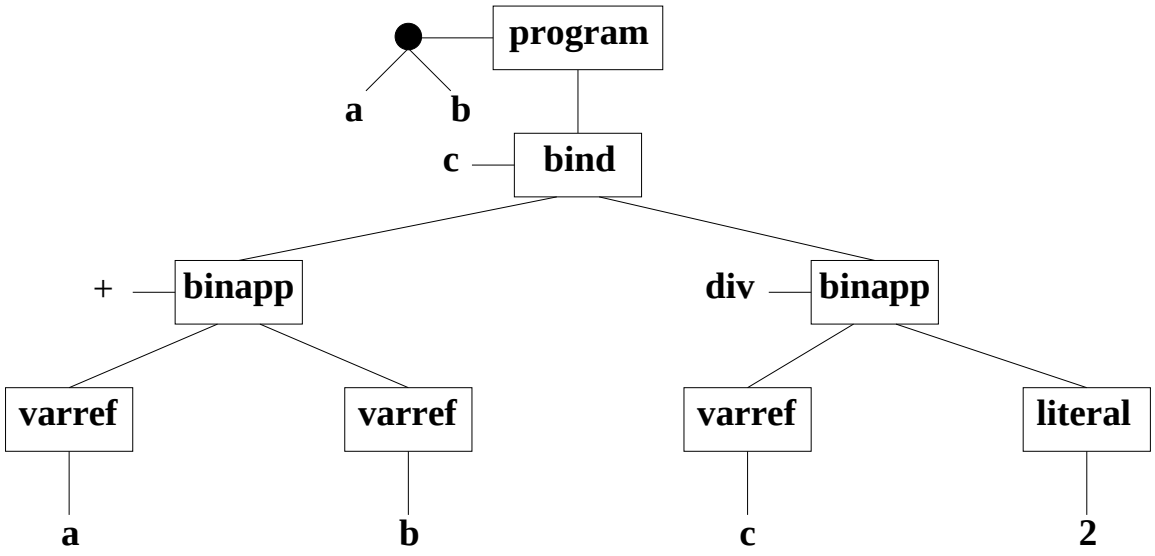
```
# open Bindex;;

# setToList (freeVarsExp (stringToExp "(bind c (+ a b) (* c d))"));;
- : Bindex.StrSet.elt list = ["a"; "b"; "d"]

# subst1 "a" (stringToExp "(+ b c)") (stringToExp "(bind a (+ a a) (* a a))");;
- : Bindex.exp =
Bind ("a.1",
 BinApp (Add, BinApp (Add, Var "b", Var "c"), BinApp (Add, Var "b", Var "c")),
 BinApp (Mul, Var "a.1", Var "a.1"))

# StringUtils.print (expToString (subst1 "a" (stringToExp "(+ b c)")
                                          (stringToExp "(bind a (+ a a) (* a a))")));;
(bind a.2 (+ (+ b c) (+ b c)) (* a.2 a.2))- : unit = ()

# BindexEnvInterp.run (Pgm(["a";"b"], BinApp(Add, Var "a", Var "b"))) [3;7];;
- : int = 10

# BindexEnvInterp.runString "(bindex (a) (bind b (* a a) (+ a b)))" [5];;
- : int = 30

# BindexEnvInterp.runFile "avg.bdx" [3;7];;
(* Assume that the file avg.bdx contains an averaging program *)
- : int = 5

# BindexEnvInterp.repl();;


bindex> (+ 1 2)
3

bindex> (bind a (+ 1 2) (+ a (* a a)))
12

bindex> (#args (a 3) (b 4) (c 5))


bindex> (+ a (* b c))
23

bindex> (#quit)

done
```

Problem 1 [25]: Abstract Syntax Trees

Consider the following BINDEX averaging program:

```
(bindex (a b)
  (bind c (+ a b)
    (div c 2)))
```

Here is the abstract syntax tree (AST) for this program:

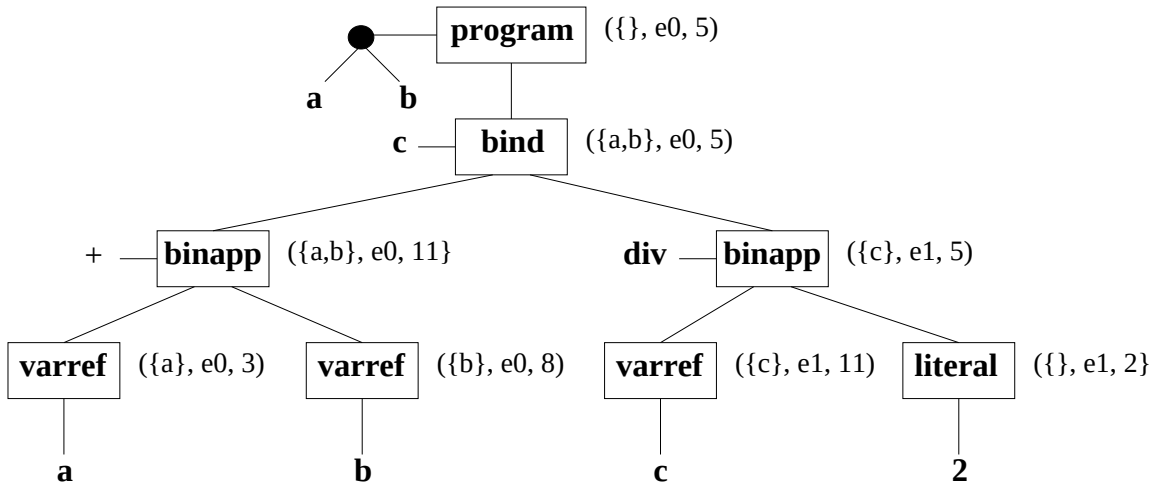


Note that the multiple parameters of the program are shown branching off a single solid node that stands for the sequence of parameters.

Suppose we annotate each node of the abstract syntax tree with a triple (*FV*, *Env*, *Val*) of the following pieces of information:

1. *FV*: the free variables of the program or expression rooted at the node.
2. *Env*: The environment in which the node would be evaluated if the program were run on the actual parameters `a` = 3 and `b` = 8. (Write environments as sets of bindings of the form *key* = *value*.)
3. *Val*: The value that would result from evaluating the node in the environment *Env*.

The following picture shows the AST for the averaging program annotated with this information. The name *e0* abbreviates the environment $\{a = 3, b = 8\}$ and *e1* abbreviates the environment $\{a = 3, b = 8, c = 11\}$.



In this problem, you are to draw a similar annotated AST for the following BINDEX program:

```
(bindex (a b c)
  (* (bind d (* a c)
       (bind e (- d b)
         (div (* b d) (+ e a))))
     (bind e (bind b (* 12 a)
               (- b c))
       (div e b))))
```

You should annotate each node of the abstract syntax tree with the same three pieces of information used in the average example above. In this case, assume that the program is run on the actual parameters `a` = 2, `b` = 3, and `c` = 5.

Note: for this problem, you will need to use a large sheet of paper and/or to write very small. It is strongly recommended that you write the solution using pencil (not pen, so you can erase) and paper. Don't waste your time attempting to format it on a computer with a drawing program.

Problem 2 [35]: sigma

The SIGMEX *Language*

In this problem, we consider the SIGMEX language, a dialect of BINDEX that has been extended with the following summation construct:

> **(sigma** *var lo hi body*)
> Assume that *var* is a variable name, *lo* and *hi* are expressions denoting integers that are not in the scope of *var*, and *body* is an expression that is in the scope of *var*. Return the sum of *body* evaluated at all values of the index variable *var* ranging from *lo* up to *hi*, inclusive. This sum would be expressed in traditional mathematical summation notation as:
>
> $$\sum_{var=lo}^{hi} body.$$
>
> If the value of *lo* is greater than that of *hi*, the sum is 0.

Here are some examples of `sigma` in action:

```
(sigma k 1 6 k)
21 (* 1 + 2 + 3 + 4 + 5 + 6 *)

(sigma k 3 5 (* k k))
50 (*  3^2 + 4^2 +5^2 *)

(sigma k 3 2 (* k k))
0

(sigma i 2 5
  (sigma j i 4
    (* i j)))
55 (* (2*2) + (2*3) + (2*4) + (3*3) + (3*4) + (4*4) *)

(sigma i 1 3
  (sigma j (* i i) 4
    (sigma k (- j i) (+ j i)
      (* i (* j k)))))
250

(sigma i (sigma k 1 3 (* k k)) (sigma j 1 5 j) i)
29
```

Your Task

Your goal is to implement SIGMEX by extending the BINDEX implementation so that it appropriately handles the `sigma` construct. In `~/cs251/ps4`, you have been given copies of the BINDEX implementation files named `Sigmex.ml`, `SigmexEnvInterp.ml`, and `SigmexSubstInterp.ml`. As described below, you need to extend certain functions in these files to correctly handle `sigma`.

In `Sigma.ml`, the `exp` data type has already been extended for you to represent the `sigma` construct:

```
and exp =
    Lit of int (* integer literal with value *)
  | Var of var (* variable reference *)
  | BinApp of binop * exp * exp (* primitive application with rator, rands *)
  | Bind of var * exp * exp (* bind name to value of defn in body *)
  | Sigma of var * exp * exp * exp (* name * lo * hi * body *)
```

You should extend the following functions to appropriately handle `Sigma`:

a. Extend the definition of `freeVarsExp` in `Sigmex.ml` to correctly determine the free variables of a `sigma` expression.

b. Extend the definition of `subst` in `Sigmex.ml` to correctly perform substitutions into `sigma` expressions.

c. Extend the definition of `sexpToExp` in `Sigmex.ml` to correctly parse `sigma` expressions.

d. Extend the definition of `expToSexp` in `Sigmex.ml` to correctly unparse `sigma` expressions.

e. Extend the definition of `eval` in `SigmexEnvInterp.ml` to correctly evaluate `sigma` expressions using the environment model.

f. Extend the definition of `eval` in `SigmexSubstInterp.ml` to correctly evaluate `sigma` expressions using the substitution model.

Notes:

- As always, begin the problem set with the following Linux commands:

  ```
  cd ~/cs251
  cvs update -d
  ```

- You can load the Sigmex implementation via the following OCAML commands

  ```
  #cd "/students/username/cs251/ps4"
  #use "load-sigmex.ml"
  ```

- Implementing parts (e) and (f) requires you to perform a summation as the `sigma`-bound variable ranges over the integer values from the lower bound to the upper bound. There are many ways to implement this summation, but a particularly elegant approach is to use some of the higher-order list functions from the `List` and/or `ListUtils` modules. (`ListUtils` is what you used on PS2.)

Problem 3 [40]: Program Simplification

Avoiding Magic Constants

It is good programming style to avoid "magic constants" in code by explicitly calculating certain constants from others. For instance, consider the following two BINDEX programs for converting years to seconds:

```
; Program 1
(program (years)
  (* 31536000 years))

; Program 2
(program (years)
  (bind seconds-per-minute 60
    (bind minutes-per-hour 60
      (bind hours-per-day 24
        (bind days-per-year 365 ; ignore leap years
          (bind seconds-per-year (* seconds-per-minute
                                    (* minutes-per-hour
                                       (* hours-per-day
                                          days-per-year)))
            (* seconds-per-year years)))))))
```

The first program uses the magic constant 31536000, which is the number of seconds in a year.[1] The second program shows how this constant is calculated from simpler constants. By showing the process by which `seconds-per-year` is calculated, the second program is a more robust and well-documented software artifact. Calculated constants also have the advantage that they are easier to modify. Although the numbers in the above program aren't going to change, there are many so-called "constants" built into a program that change over its lifetime. For instance, the size of word of computer memory, the price of a first-class stamp, and the rate for a certain tax bracket are all numbers that could be hard-wired into programs but which might need to be updated in future version of the software.

However, magic constants can have performance advantages. In the above programs, the program with the magic constant performs one multiplication, while the other program performs four multiplications. If performance is critical, the programmer might avoid the clearer style and instead opt for magic constants.

Program Simplification

Is there a way to get the best of both approaches? Yes! We can write our program in the clearer style, and then automatically transform it to the more efficient style via a process known as **program simplification**. In program simplification, we rewrite a program into another one that has the same meaning by performing computation steps that would otherwise be performed when running the program. Any steps we can perform during simplification are steps that are avoided later; in most cases, this improves the run-time performance of the program.

For instance, we can use program simplification to systematically derive the first program above from the second. We begin via a step known as **constant propagation**, in which we substitute the four constants at the top of the second program into their references to yield:

[1] It is worth noting that this number is approximately $\pi \times 10^7$. So a century is approximately $\pi \times 10^9$ seconds, which means that π seconds is approximately one nano-century!

```
(program (years)
  (bind seconds-per-minute 60
    (bind minutes-per-hour 60
      (bind hours-per-day 24
        (bind days-per-year 365 ; ignore leap years
          (bind seconds-per-year (* 60 (* 60 (* 24 365)))
            (* seconds-per-year years)))))))
```

Next, we eliminate the now-unnecessary first four bindings via a step known as **dead code removal**:

```
(program (years)
  (bind seconds-per-year (* 60 (* 60 (* 24 365)))
    (* seconds-per-year years)))
```

We can now perform the three multiplications involving manifest integers in a step known as **constant folding**:

```
(program (years)
  (bind seconds-per-year 31536000
    (* seconds-per-year years)))
```

Finally, another round of constant propagation and dead code removal yields the first program:

```
(program (years)
  (* 31536000 years))
```

It is not possible to eliminate bindings whose definition ultimately depends on the program parameters. Nevertheless, it is often possible to partially simplify such definitions. For example, consider:

```
(program (a)
  (bind b (* 3 4)
    (bind c (+ a (- 15 b))
      (bind d (div c b)
        (* d c)))))
```

The simplification techniques described above can simplify this program to:

```
(program (a)
  (bind c (+ a 3)
    (bind d (div c 12)
      (* d c)))))
```

Your Task

In this problem, your task is to write a function `simplify` that performs simplification on a BINDEX program by using the constant propagation, constant folding, and dead-code elimination steps illustrated above. Given a BINDEX program, `simplify` should return another BINDEX program that has the same meaning as the original program, but which also satisfies the following properties:

1. The program should not contain any `bind` expressions in which a variable is bound to an integer literal.

2. The program should not contain any binary applications in which an arithmetic operator is applied to two integer literals. There are two exceptions to this property: the program may contain binary applications of the form `(div n 0)` or `(mod n 0)`, since these cannot be simplified by the constant folding process.

It is possible to write separate functions that perform the constant propagation, constant folding, and dead-code elimination steps, but it is tricky to get them to work together to perform all simplifications. It turns out that it is much more straightforward to perform all three kinds of simplification at the same time in a single walk over the expression tree.

By analogy with `BindexEnvInterp.run` and `BindexEnvInterp.eval`, simplification can be performed by a pair of functions `simplify` and `simp`:

> **val** `simpPgm: Bindex.pgm -> Bindex.pgm`
> Returns the simplified version of the given BINDEX program.
>
> **val** `simp: Bindex.exp -> int Env.env -> Bindex.exp`
> Given a BINDEX expression *exp* and a simplification environment *env*, returns the simplified version of *exp*. The simplification environment contains name/value bindings for names whose integer values are known.

Your goal is to implement simplification by fleshing out these two function definitions in the file `Simp.ml`.

The correspondence between `run`/`eval` and `simpPgm`/`simp` is not coincidental. Indeed, `simp` is effectively a version of `eval` that evaluates as much of an expression as it can based on the "partial" environment information it is given. Because bindings for some names may be missing in the environment, `simp` cannot always evaluate every expression to the integer it denotes and in some cases must instead return a residual expression that will determine the value when the program is executed. Because of this, `simp` must always return an expression rather than an integer; even in the case where it can determine the value of an expression, that value must be expressed as an integer literal node, not an integer.

Notes

- Perform `#use "load-simp.ml"` to load the simplifier.

- Divisions and remainders whose second operands are zero must be left in the program. Such programs will encounter divide-by-zero errors when they are later executed. For example,

```
(bindex (a)
  (bind b (* 3 4)
    (bind c (div b (- 12 b))
      (* c b))))
```

  should be transformed to:

```
(bindex (a)
  (bind c (div 12 0)
    (* c 12)))
```

- In some cases it would be possible to perform more aggressive simplification if you took advantage of algebraic properties like the associativity and commutativity of addition and multiplication. To simplify this problem, *you should not use any algebraic properties of the arithmetic operators.* For example, you should not transform `(+ 1 (+ a 2))` into `(+ 3 a)`, but should leave it as is. You should not even perform "obvious" simplifications like `(+ 0 a)` $\Rightarrow$ `a`, `(* 1 a)` $\Rightarrow$ `a`, and `(* 0 a)` $\Rightarrow$ `0`. Although the first two of these simplification are valid, the last is unsafe in the sense that it can change the meaning of a program. For instance, `(* 0 (div a b))` cannot be simplified to `0`, because it does not preserve the meaning of the program in the case where `b` is `0` (in which case evaluating the expression should give an error).

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 4
Due 6pm Saturday, February 28

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading your problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [25]		
Problem 2 [35]		
Problem 3 [40]		
Total		