Homework 2

Due 15 October

Handout 3
CS242: Autumn 2003
8 October

Reading

1. From Chapter 4, read Sections 4.1–4.2 and 4.4–4.5. You may skim section 4.3 on denotational semantics if you like.
2. Read Chapter 5 on the Algol family of programming languages and ML.

Problems

1. .... Parsing and Precedence

Draw parse tress for the following expressions, assuming the grammar and precedence described in Example 4.2:

(a) $1 - 1 * 1$.

(b) $1 - 1 + 1$.

(c) $1 - 1 + 1 - 1 + 1$, if we give $+$ higher precedence than $-$.

2. .... Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda x.\lambda y.xy)(\lambda x.xy)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

3. .... Symbolic Evaluation

The Algol-like program fragment

```
function f(x)
   return x+4
end;
function g(y)
   return 3-y
end;
f(g(1));
```

can be written as the following lambda expression:

$$\left(\underbrace{(\lambda f.\lambda g.f\ (g\ 1))}_{\text{main}}\ \underbrace{(\lambda x.x+4)}_{f} \right)\ \underbrace{(\lambda y.3-y)}_{g}$$

Reduce the expression to a normal form in two different ways, as described below.

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *left* as possible.

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the *right* as possible.

4. .... Lazy Evaluation and Parallelism

In a "lazy" language, a function call `f(e)` is evaluated by passing the *unevaluated* argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of `f`. For example, consider the function g defined by

```
fun g(x, y) = if x = 0
              then 1
              else if x + y = 0
                   then 2
                   else 3;
```

In a lazy language, the call $g(3, 4+2)$ is evaluated by passing some representation of the expressions 3 and $4+2$ to g. The test $x = 0$ is evaluated using the argument 3. If it were `true`, the function would return 1 without ever computing $4+2$. Since the test is `false`, the function must evaluate $x + y$, which now causes the actual parameter $4+2$ to be evaluated. Some examples of lazy functional languages are Miranda, Haskell and Lazy ML; these languages do not have assignment or other imperative features with side effects.

If we are working in a pure functional language without side-effects, then for any function call $f(e_1, e_2)$, we can evaluate e_1 before e_2 or e_2 before e_1. Since neither can have side-effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expression. Therefore, something can go wrong if we evaluate both expressions and one of them does not terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call $f(e_1, e_2)$ we can evaluate both e_1 and e_2 in parallel. In fact, we could even start evaluating the body of `f` in parallel as well.

(a) Assume we evaluate $g(e_1, e_2)$ by starting to evaluate g, e_1, and e_2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

(b) Now, suppose the value of e_1 is zero and evaluation of e_2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression $g(e_1, e_2)$ will terminate in error. What will happen with lazy evaluation? Parallel evaluation?

(c) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate $g(e_1, e_2)$ by starting g, e_1, and e_2 in parallel, if the value of e_1 is zero and evaluation of e_2 terminates in an error?

(d) Suppose, now, that the language contains side-effects. What if e_1 is z, and e_2 contains an assignment to z. Can you still evaluate the arguments of $g(e_1, e_2)$ in parallel? How? Or why not?

5. .... Single-Assignment Languages

A number of so-called *single-assignment languages* have been developed over the years, many designed for parallel scientific computing. Single assignment conditions are also used in program optimization and in hardware description languages. Single-assignment conditions arise in hardware since only one assignment to each variable may occur per clock cycle.

One example single-assignment language is *SISAL,* which stands for Streams and Iteration in a Single Assignment Language. Another is *SAC,* or Single-Assignment C. Programs in single-assignment languages must satisfy the following condition:

> *Single-Assignment Condition:* During any run of the program, each variable may be assigned a value only once, within the scope of the variable.

The following program fragment satisfies this condition

```
if (y>3) then x = 42+29/3 else x = 13.39;
```

since only one branch of the if-then-else will be executed on any run of the program. The program `x=2; loop_forever; x=3` also satisfies the condition since no execution will complete both assignments.

Single-assignment languages often have specialized loop constructs, since otherwise it would be impossible to execute an assignment inside a loop body that gets executed more than once. Here is one form, from SISAL:

```
for ⟨range⟩
   ⟨body⟩
returns ⟨returns clause⟩
end for
```

An example illustrating this form is the following loop, which computes the dot (or inner) product of two vectors:

```
for i in 1, size
   elt_prod := x[i] * y[i]
returns value of sum elt_prod
end for
```

This loop is parallelizable since different products x[i]∗y[i] can be computed in parallel. A typical SISAL program has a sequential outer loop containing a set of parallel loops.

Suppose you have the job of building a parallelizing compiler for a single-assignment language. Assume that the programs you compile satisfy the single-assignment condition and do not contain any explicit process fork or other parallelizing instructions. Your implementation must find parts of programs that can be safely executed in parallel, producing the same output values as if the program was executed sequentially on a single processor.

Assume for simplicity that every variable is assigned a value before the value of the variable is used an in expression. Also assume that there is no potential source of side effects in the language other than assignment.

(a) Explain how you might execute parts of the sample program

```
x = 5;
y = f(g(x),h(x));
if y==5 then z=g(x) else z=h(x);
```

in parallel. More specifically, assume that your implementation will schedule the following processes in some way:

process 1 – set x to 5
process 2 – call `g(x)`
process 3 – call `h(x)`
process 4 – call `f(g(x),h(x))` and set `y` to this value
process 5 – test `y==5`
process 6 – call `g(x)` and then set `z=g(x)`
process 7 – call `h(x)` and then set `z=h(x)`

For each process, list the processes that this process must wait for and list the processes that can be executed in parallel with it. For simplicity, assume that a call cannot be executed until the parameters have been evaluated and assume that processes 6 and 7 are *not* divided into smaller processes that execute the calls but do not assign to z. Assume that parameter passing in the example code is by value.

(b) If you further divide process 6 into two processes, one that calls `g(x)` and one that assigns to `z`, and similarly divide process 7 into two processes, can you execute the calls `g(x)` and `h(x)` in parallel? Could your compiler correctly eliminate these calls from processes 6 and 7? Explain briefly.

(c) Would the parallel execution of processes you describe in parts (a) and (b), if any, be correct if the program does not satisfy the single-assignment condition? Explain briefly.

(d) Is the single-assignment condition decidable? Specifically, given an program written in a subset of C, for concreteness, is it possible for a compiler to decide whether this program satisfies the single-assignment condition? Explain why or why not. If not, can you think of a decidable condition that implies the single-assignment condition and allows many useful single-assignment programs to be recognized?

(e) Suppose a single-assignment language has no side-effecting operations other than assignment. Does this language pass the declarative language test? Explain why or why not?

6. .... Algol 60 Pass-By-Name

The following Algol 60 code declares a procedure `P` with one pass-by-name integer parameter. Explain how the procedure call `P(A[i])` changes the values of `i` and `A` by substituting the actual parameters for the formal parameters, according to the Algol 60 copy rule. What integer values are printed by tprogram? using pass-by-name parameter passing?

The line`integer x` does not declare local variables – this is just Algol 60 syntax declaring the type of the procedure parameter.

```
begin
   integer i;
   integer array A[1:2];

   procedure P(x);
      integer x;
      begin
         i := x;
         x := i
      end

   i := 1;
   A[1] := 2; A[2] := 3;
   P (A[i]);
   print (i, A[1], A[2])
end
```

7. .... Currying

This problem asks you to show that the ML types $'a \to ('b \to 'c)$ and $('a * 'b) \to 'c$ are essentially equivalent.

(a) Define higher-order ML functions

$$\texttt{Curry} : (('a * 'b) \to 'c) \to ('a \to ('b \to 'c))$$

and

$$\texttt{UnCurry} : ('a \to ('b \to 'c)) \to (('a * 'b) \to 'c)$$

(b) For all functions $\texttt{f} : ('a * 'b) \to 'c$ and $\texttt{g} : 'a \to ('b \to 'c)$, the following two equalities should hold (if you wrote the right functions):

$$\texttt{UnCurry}(\texttt{Curry}(\texttt{f})) = \texttt{f}$$

$$\texttt{Curry(UnCurry(g))} = \texttt{g}.$$

Explain why each is true, for the functions you have written. Your answer can be 3 or 4 sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than number of words.) Be sure to consider termination behavior as well.

8. .... ML implementation of lambda reduction

This exercise asks you to implement β-reduction in ML and test your reduction function. The next few paragraphs explain the datatype you should use and give you some example terms. The ML function declarations with parts missing are on the CS242 web site at http://www.stanford.edu/class/cs242/handouts/lambda-eval.txt To submit your program on the Leland systems, create a directory for your project called hw2 and run /usr/class/cs242/bin/submit from that directory. Make sure that your hw2 directory only contains files for this programming assignment.

Lambda terms with addition have the BNF

$$t \quad ::= v \,|\, c \,|\, t+t \,|\, t\,t \,|\, \lambda v.\, t$$

In words, a lambda terms is either a variable, a constant (symbol with a fixed meaning), a sum of two terms, an application of one term to another, or a lambda abstraction. It is assumed that there are infinitely many variables, $v_1, v_2, v_3, \ldots$ so we can write a term with as many variables as we need, and there are enough "unused" variables that we can always α-convert when we need to. Since the only operation built into this version of lambda calculus is addition, we might as well assume that the constants are numbers $0, 1, 2, \ldots$. Some examples of terms produced by this grammar are $\lambda y.\, y + x$ and $\lambda y.\, \lambda x.\, y (y\, x)$

We can write a very similar ML datatype for lambda terms with addition, representing the variables $v_1, v_2, v_3, \ldots$ in the form $\texttt{Var}(1), \texttt{Var}(2), \ldots$ and the numbers $0, 1, 2, \ldots$ in the form $\texttt{Const}(0), \texttt{Const}(1), \texttt{Const}(2), \ldots$, as follows

```
datatype term =  Var of int| Const of int | Plus of term * term
              |  App of term * term | Lambda of int* term
```

Here are some example terms that can be used to make larger terms more readable.

```
val x = Var(1);
val y = Var(2);
val one = Const(1);
val two = Const(2);
```

Using the convenient declarations `val x_ = 1; val y_ = 2;` that let us write the numbers 1 and 2 using symbols that remind us of variables x and y, we can write terms $\lambda y.\, y + x$ and $\lambda y.\, \lambda x.\, y (y\, x)$, along with the application of the second term to the first, as

```
val twice = Lambda(y_, Lambda(x_, App(y, App(y,x))));
val addx = Lambda(y_, Plus(y,x));
val test = App(twice,addx);
```

You might notice that `test` is the lambda term used in lecture to illustrate the need for renaming bound variables when performing β-reduction.

(a) Write a function that substitutes a term for a variable, without renaming bound variables. Your function should have the form below, with the missing parts indicated by `(*-- missing part --*)` filled in. The function `subst{t,x,s}` substitutes `t` for all free occurrences of `x` in `s` and is defined by cases on the form of `s`:

```
fun subst(t,x,Var(n)) = if x=n then t else Var(n)
 |  subst(t,x,Const(n)) = Const(n)
 |  subst(t,x,Lambda(n,t1)) = if x=n then (*-- missing part --*)
                                     else (*-- missing part --*)
 |  subst(t,x,App(t1,t2)) =  (*-- missing part --*)
 |  subst(t,x,Plus(t1,t2)) = (*-- missing part --*)
```

(b) Write a function `ev` that evaluates a terms using `subst`. This function should not rename bound variables. Use the following form which breaks the problem into separate cases. In each case, evaluate all the parts that you can, but just return an expression like `ev (App(Const(3),Const(4)))` as is since there is no way to reduce this further. However, when an expression adds two numbers, you should return their sum.

```
fun ev(Var(n)) = Var(n)
 |  ev(Const(n)) = Const(n)
 |  ev(Lambda(n,t1)) = Lambda(n,ev(t1))
 |  ev(App(t1,t2)) =
        (case ev(t1) of
              Var(n) => App(Var(n),ev(t2))          |
            Const(n) => App(Const(n),ev(t2))        |
         Lambda(n,t) => (*-- missing part --*)      |
           App(t,t1) => App(App(t,t1),ev(t2))       |
          Plus(t,t1) => (*-- missing part --*)
 |  ev(Plus(t1,t2)) =
         case ev(t1) of
              Var(n) => Plus(Var(n),ev(t2))          |
            Const(n) => (case ev(t2) of
                           Var(m) => Plus(Const(n),Var(m))         |
                         Const(m) => (*-- missing part --*)        |
                      Lambda(m,t) =>  Plus(Const(n),Lambda(n,t))   |
                        App(t3,t4) => (*-- missing part --*)       |
                       Plus(t3,t4) => App(Const(n),Plus(t3,t4)) )  |
         Lambda(n,t) => Plus(Lambda(n,t),ev(t2))  |
           App(t,t1) => Plus(App(t,t1),ev(t2))    |
          Plus(t,t1) => (*-- missing part --*)    ;
```

Note that the case given for `ev(Lambda(n,t1))` evaluates the body of a lambda expression.

(c) Write a function `alpha` that renames bound variables so that no free variable in `alpha(term)` is the same as any bound variable. One way to do this is define `alpha(term) = alph( ... ,term, ...)` where `alph` has additional arguments besides the term itself. For example, you could try something like `alpha(term) = alph(free_variables(term),term)` where `free_variables` is a function that returns a list of free variables in a term. The idea here might be to use the list of free variables so that you can rename bound variables to names that are not in the list. There might also be some way to use the fact that variables are numbered to simplify this. You can be creative, but use your creativity to write a short, easy to read function, not some complicated mess. There is no need for assignment or reference cells, so do not use them.

(d) Define `eval(term) = ev(alpha(term))` and compare `ev(test)` with `eval(test)`.

(e) (*Extra Credit*) Consider replacing `ev(Lambda(n,t1)) = Lambda(n,ev(t1))` with `ev(Lambda(n,t1)) = Lambda(n,t1)`. How is the resulting new definition of `eval` better? Worse? (*Hint:* Think about the definition of factorial in Homework 1.)