Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
http://courses.softlab.ntua.gr/pl2/

Γλώσσες Προγραμματισμού II

Αν δεν αναφέρεται διαφορετικά, οι ασκήσεις πρέπει να παραδίδονται στους διδάσκοντες σε ηλεκτρονική μορφή μέσω του συνεργατικού συστήματος ηλεκτρονικής μάθησης `moodle.softlab.ntua.gr`. Η προθεσμία παράδοσης θα τηρείται αυστηρά. Έχετε δικαίωμα να καθυστερήσετε το πολύ μία άσκηση.

Άσκηση 4 Υλοποίηση συναρτησιακών γλωσσών προγραμματισμού

Προθεσμία παράδοσης: 16/1/2012

Στις επόμενες δύο σελίδες υπάρχουν δύο φωτοτυπημένες ασκήσεις από το βιβλίο του John C. Mitchell, "*Concepts in Programming Languages*," Cambridge University Press.

Για να λύσετε την άσκηση 4.12 θα χρειαστείτε τον παρακάτω ορισμό του *Declarative Language Test*:

> Within the scope of specific declarations of $x_1, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, \ldots, x_n$ have the same value.

το οποίο περνάνε οι αγνά συναρτησιακές γλώσσες προγραμματισμού.

Για την άσκηση 7.17, εκτός από τις ερωτήσεις που αναφέρονται στη φωτοτυπημένη σελίδα απαντήστε και στην εξής επιπλέον ερώτηση:

(d) Όπως αναφέρεται και στην εκφώνηση, "τυπικά", η συνάρτηση `fact″` είναι tail recursive. Πόσος χώρος χρειάζεται για τη αποτίμηση της $\texttt{fact}_C(n)$ ως συνάρτηση του n; Εξηγήστε πού δεσμεύεται αυτός ο χώρος κατά την εκτέλεση των αναδρομικών κλήσεων της `fact″` και πότε και με ποιο τρόπο ο χώρος αυτός αποδεσμεύεται. Με βάση την απάντησή σας σε αυτή την ερώτηση, τι γνώμη έχετε για τον συνήθη ορισμό της αναδρομής ουράς που ουσιαστικά λέει ότι μια συνάρτηση που είναι tail recursive αποτιμάται σε σταθερό μέγεθος μνήμης (στοίβας);

4.12 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 as only one assignment to each variable may occur per clock cycle.

One example of a 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;
```

because 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 because no execution will complete both assignments.

Single-assignment languages often have specialized loop constructs, as 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 because 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 were 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-effect operations other than assignment. Does this language pass the declarative language test? Explain why or why not.

7.17 Tail Recursion and Order of Operations

This asks about the order of operations when converting a function to tail recursive form and about passing functions from one scope to another. Here are three versions of our favorite recursive function, *factorial*:

Normal recursive version

$\textbf{fact}_A\ (n) =$ (if n=0 then 1 else n*fact_A(n-1))

Tail-recursive version

$\textbf{fact}'(n,a) =$ (if n=0 then a else fact'(n-1,(n*a)))

$\textbf{fact}_B(n) =$ fact'(n,1)

Another tail-recursive version

$\textbf{fact}''(n,rest) =$ (if n=0 then rest(1)
else fact''(n-1,(fn(r)⇒ rest(n*r))))

$\textbf{fact}_C(n) =$ fact''(n,fn(answer)⇒answer)

Here is the evaluation of $\textbf{fact}_A$ (3) and $\textbf{fact}_B$ (3):

$\textbf{fact}_A(3) =$ (if 3=0 then 1 else 3*fact_A(2))
= (if false then 1 else 3*fact_A(2))
= (3*fact(2))
= (3*(if 2=0 then 1 else 2*fact_A(1)))
= (3*(2*fact(1)))
= (3*(2*(if 1=0 then 1 else 1*fact_A(0))))
= (3*(2*(1*fact(0))))
= (3*(2*(1*(if 0=0 then 1 else 0*fact_A(-1)))))
= (3*(2*(1*(1))))

$\textbf{fact}_B(3) =$ fact'(3,1)
= (if 3=0 then 1 else fact'(2,(3*1)))
= (fact'(2,(3*1)))
= (if 2=0 then (3*1) else fact'(1,(2*(3*1))))
= (fact'(1,(2*(3*1))))
= (if 1=0 then (2*(3*1)) else fact'(0,(1*(2*(3*1)))))
= (fact'(0,(1*(2*(3*1)))))
= (if 0=0 then (1*(2*(3*1))) else fact'(-1,(0*(1*(2*(3*1))))))
= (1*(2*(3*1)))

The multiplications are not carried out in these symbolic calculations so that we can compare the order in which the numbers are multiplied. The evaluations show that $\textbf{fact}_A$ (3) multiplies by 1 first and by 3 last; $\textbf{fact}_B$ (3) multiplies by 3 first and 1 last. The order of multiplications has changed.

(a) Show that $\textbf{fact}_C$ (3) multiplies the numbers in the correct order by a symbolic calculation similar to those above that does not carry out the multiplications.

(b) We can see that **fact''** is tail recursive in the sense that there is nothing left to do after the recursive call. If these functions were entered into a compiler that supported tail recursion elimination, would $\textbf{fact}_C$ (n) be closer in efficiency to $\textbf{fact}_A$ (n) or $\textbf{fact}_B$ (n)? First, ignore any overhead associated with creating functions, then discuss the cost of passing functions on the recursive calls.

(c) Do you think it is important to preserve the order of operations when translating an arbitrary function into tail recursive form? Why or why not?