Midterm #1 Solution

CSE 428 Fall 1998
7 October

1. For each of the following grammars, (20 pts)

   - state whether or not it is ambiguous
   - state any operator precedences which are enforced
   - state any operator associativities which are enforced

   Note that even if a grammar is ambiguous, it can still enforce operator precedences and associativities.

   (a)

$$\begin{aligned} E &::= E\text{“}*\text{”}E \mid F \\ F &::= F\text{“}+\text{”}T \mid F\text{“}-\text{”}T \mid T \\ T &::= N \mid Id \mid \text{“(”}E\text{“)”} \end{aligned}$$

   Ambiguous (due to E“$*$”E);
   Precedence: $* < +, -$
   $+$ **and** $-$ **are left-associative**

   (b)

$$\begin{aligned} E &::= E\text{“}*\text{”}F \mid F \\ F &::= F\text{“}+\text{”}G \mid G \\ G &::= T\text{“}-\text{”}G \mid T \\ T &::= N \mid Id \mid \text{“(”}E\text{“)”} \end{aligned}$$

   Unambiguous ;
   Precedence: $* < + < -$
   $*$ **and** $+$ **are left-associative,** $-$ **is right-associative**

2. Recall the general form of let expressions: (20 pts)

```
let x1 = e1;
    x2 = e3;
    ...
    xn = en
in e
endlet
```

The original operational semantics for expressions evaluated the e_i *sequentially*, incrementally adding new bindings to the environment. We also saw (in an assignment) how to give an alternative semantics in which the e_i are evaluated in *parallel.*

Give a precise description of when a let expression (such as the one above) will yield the same value using either the sequential or parallel semantics for let in an arbitrary environment ρ. I.e., what syntactic restrictions must be placed on let expressions to ensure this behavior?

Answer: for all e_j, $(1 \leq j \leq n)$, e_i cannot contain any x_i for all i, $1 \leq i < j$.

3. Recall the typechecking rule for recursive function declarations: (20 pts)

$$\frac{\Gamma[\mathtt{f} : \tau \rightarrow \tau', \mathtt{x} : \tau] \vdash e : \tau'}{\Gamma \vdash \mathtt{f(x)\ =\ e} \Rightarrow \Gamma[\mathtt{f} : \tau \rightarrow \tau']}$$

Since we also added function calls to the language of expressions, we also need to add typechecking rules for function calls:

$$\frac{\Gamma(f) = \tau' \rightarrow \tau \quad \Gamma \vdash e : \tau'}{\Gamma \vdash f(e) : \tau}$$

Let $\Gamma_0 = [f\text{:integer} \rightarrow \text{bool},\ g\text{:integer} \rightarrow \text{integer},\ x\text{:bool},\ y\text{:integer}]$. Using the rule above for typechecking function calls (and all the original rules for typechecking expressions), type each of the following expressions with respect to Γ_0. If the expression is not well-typed, then write **no type**; otherwise, give the type of the expression.

(a)
```
let x = g(y+1) + y
in  f(x)
endlet
```

bool

(b)
```
let f = f(y);
    y = f or x
in  f(y)
endlet
```

no type

(c)
```
let z = let x = 5 in f(x);
    y = z and x
in  y
endlet
```

bool

(d)
```
let g = let f = 5 in g(f);
    y = f(g)
in  f(y)
endlet
```

no type

4. Consider the following program: (20 pts)

```
program main
  x,y : integer;

  procedure lear()
  x : integer;
  begin
    x := y + 1;
    y := x + y;
    write(x,y);
  end lear;

  procedure gonerill()
  y : integer;
  begin
     y := x + 1;
     lear();
     write(x,y);
  end gonerill;

begin main
  x := 1;
  y := 1;
  gonerill();
  write(x,y);
end main;
```

What is output by this program under

(a) static scoping: **2 3   1 2   1 3**

(b) dynamic scoping **3 5   1 5   1 1**

5. Consider the following program: (20 pts)

```
program main
  x,y : integer;

  procedure regan(a,b :integer)
  begin
    a := b + x;
    b := a + x;
    write(a,b);
  end regan;

begin main
  x := 1;
  y := 2;
  regan(x,y);
  write(x,y);
end main;
```

What is output by this program if **all** parameters are passed using the following.

(a) call-by-value **3 4   1 2**

(b) call-by-reference **3 6   3 6**