p5q10
ACN

~~and our states will be various subsets of these. See page 64 of Appel for this very example worked through!~~

~~For behaviour of LR parser eating x+x+x$ it has to go something like~~

~~x   reduce T->x~~
~~+   shift~~
~~x   reduce T->x~~
~~+   shift~~
~~x   reduce T->x~~
~~reduce again using E->T~~
~~reduce again using E->T+E~~
~~reduce again using E->T+E~~
~~reduce using S->E$~~
~~accept~~

2000

2 Foundations of Functional Programming

Give a brief account of how **four** of the following features of general programming systems can be modelled in terms of a form of un-typed functional programming where none of the mentioned facilities are provided as built-in features:

1. Tuples (it will be sufficient to consider just the case of pairs);
2. Boolean quantities and an *if/then/else* construct;
3. Lists (both empty and non-empty);
4. Recursive function definitions;
5. The numbers 0,1,2,..., with the associated operations of a zero test, addition and multiplication.

(4 marks for each part attempted)
Select the cases you describe so that in one of the cases your modelling could still be carried out in a polymorphically typed functiona language and in one case it could not, and explain briefly which is which and why.
(4 marks)

2.1 Marking notes

If I use ML-like syntax

```
fun p1 a b = a;  // projection functions
fun p2 a b = b;

fun pair a b f = f a b;
fun left p = p p1;
fun right p = p p2;

This is OK typed.

val true = p1;
val false = p2;
fun if a b c = a b c;

OK typed.


val nil = pair true junk;
fun cons a b = pair false (pair a b);
fun null x = left x;
fun car x = left (right x);  // valid if not nil
fun cdr x = right (right x); // ditto

Getting a type-valid thing for the "junk" in nil is a bit of a
pain and it seems hard to get a finite type out in all.


The Y operator is standard in the notes, and has big trouble with
finite types if you define it in terms of raw lambda.

Church numerals are standard in the notes. Types are tolerably behaved.
```

3 Foundations of Functional Programming

(a) Give as simple a set of rules as you can for transforming lambda calculus to a form where there are no bound variables mentioned, but where there are many instances of the three standard combinator constants **S**, **K** and **I**;
(6 marks)

(b) Describe tree-rewrites suitable for reducing expressions written in terms of combinators;
(6 marks)

(c) Explain how you might deal with the issue of keeping track of the values of bound variables if you were to interpret lambda calculus directly.
(8 marks)

3.1 Marking notes

```
[x]x = x
[x]y = Ky
[x]fg = S([x]f)([x]g)


Ix -> x    Kxy -> x   Sfgx -> (fx)(gx)
```