Types & Programming Languages
Assessed Exercise

The assessed exercise has three parts, involving a mixture of theoretical and practical work. The deadline for handing in the exercise is Friday 1st December.

1. **Mainly practical**

   This part of the exercise involves extending the implementation of the BFL typechecker to include product types, reference types, the unit type, and sum types.

   Your tasks are:

   (a) Add product types (you only need to deal with binary products of the form $T \times U$, not general products $T_1 \times \cdots \times T_n$) with pairs (e, f) and the projection operators `fst` and `snd`.

   (b) Add the unit type `unit` and its value `()`.

   (c) Add reference types with reference creation (the `ref` operator), assignment, and dereferencing (the `!` operator). Also add the sequencing operator `;` .

   (d) Add sum types (you only need to deal with binary sums of the form $T + U$, not general sums $T_1 + \cdots + T_n$ or variant types) with the inl and inr constructors and case expressions. You will find that it is not possible to calculate the type of inl(e) or inr(e) from the type of e (why not?); deal with this by introducing syntax so that expressions of the form inl(e) and inr(e) have an explicit type declaration attached to them.

   For each of these extensions you will need to modify the grammar, extend the representation of types, extend the typechecker, and add suitable error classes.

   The main point of this exercise is implementing the typechecker, not modifying the grammar, so it's OK if you have to introduce minor variations to the syntax in order to get the grammar to work. However, you should try to stick as closely as possible to the syntax discussed in lectures. You should also allow sequences of expressions separated (or terminated) by `;`, so that you can write programs like this:

```
let val x = ref 2
in  x := (!x) + 1;
    x
end
```

2. **Theoretical and practical**

   In this part of the exercise you will introduce a new construct which makes it easier to use product types in BFL. The idea is to be able to write

```
split x as (y,z)
in    code
end
```

   where `code` is arbitrary. If `x` has type $T \times U$ then at runtime `y` should be set to the first component of `x` and `z` should be set to the second component of `x`, when executing `code`.

   (a) Define suitable typing rules and reduction rules for the `split` construct.

   (b) Extend your typechecker from (1) so that `split` is included.

3. **Theoretical**

   Consider a language consisting of the simply-typed λ-calculus extended with sum types. Write out proofs of Type Preservation and Type Safety for this language, to the same level of detail as the proofs in lectures. Also prove any lemmas that you need (for example, a Substitution Lemma).

Your submission will consist of the complete source code for your modified implementation, and a written report. The report should contain, for each of the extensions specified above:

- A description of the syntax, with examples.
- A specification of the typing rules which you have implemented.
- An explanation of the new code in the `Checker` class.
- An explanation of any new error messages which you have defined.
- Examples of both correct and incorrect programs, showing the types which are calculated or the error messages which are produced. Your examples should show the new features being used in isolation and in combination with each other and with existing features of BFL. For example you should consider functions which accept or return pairs, functions whose parameter or result type is a sum, functions which accept or return references, pairs of functions, pairs of references, references to functions, references to pairs, etc.

The report should also contain the reduction rules from question (2), and the proofs from question (3).

Marking

Each of the three extensions from question (1), and the extension from question (2), will be marked according to:

- The code which you have added to the implementation.
- Your report and how well you have described your implementation; the range of examples of correct and incorrect code; the quality of the error messages you produce.
- Performance on other test cases which I will use.

The reduction rules from question (2) and the proofs from question (3) will be marked for correctness and clarity.

The number of marks for each task in each category will be as follows.

	Products	Unit	References	Sums	Split
Code	5	5	5	5	5
Report	5	5	5	5	5
Tests	5	5	5	5	5

There will be an additional 25 marks for the proofs.

The overall mark will be out of 100.

The marks for the report will include writing style, clarity of explanation, and the understanding you show.

The deadline is the end of week 10 (Friday 1st December).