CMSC 22610	**Implementation**	**Handout 2**
Winter 2005	**of**	**January 4, 2005**
	Computer Languages	

Project overview

1 Introduction

The project for this course is the implementation of a simple ML-like language, called MLR, that supports record polymorphism. The project will be broken down into four parts: the lexer, which converts a stream of input characters into *tokens*; the parser, which analyses the syntactic structure of the token stream and produces a *parse tree*; the type checker, which checks the parse tree for type correctness; and a simple code generator for interpretation. Each part of the project builds on the previous parts.

1.1 Project schedule

The following is a tentative schedule for the project assignments.

Assignment date	**Description**	**Due date**
Jan. 6	Lexer	Jan. 21
Jan. 20	Parser	Jan. 31
Jan. 27	Typechecker	Feb. 18
Feb. 17	Code generation and runtime	Mar. 11

2 MLR

The syntax and semantics of MLR are similar to Standard ML, but with many simplifications and one addition. MLR does not have type inference, tuples, datatypes, exceptions, or modules. It is a strongly typed higher-order language. As does SML, MLR has labeled records, but with the major difference that a limited form of record subtyping (sometimes called record polymorphism) is supported.

2.1 Types and values

MLR supports three primitive types of values: booleans, integers, and strings. In addition, MLR has lists, labeled records, and first-class function values.

2.2 Declarations

An MLR program is a sequence of top-level declarations, which are either type definitions or function definitions. The last declaration in an MLR program should be a function named `main` that should take a list of strings as its argument (the command-line arguments) and return an integer result (0 for okay, non-zero for an error).

2.3 Functions

Functions in MLR are first-class: they may be nested, taken as arguments, and returned as results. Function definitions may be curried; for example,

```
fun add (x : int) (y : in) : int = x+y in
val inc : int -> int = inc 1 in
  inc (add 1 2)
```

evaluates to `4`.

2.4 Expressions

MLR is an *expression* language, which means that all computation is done by expressions (there are no statements). Expressions include let bindings, conditionals, function application, and various operations. Of particular interest are expressions used to compute records. For example, the expression

```
{ x = 1, y = 2 }
```

constructs a record with two fields (`x` and `y`). Record fields can be selected using the “dot” notation and modified or extended using the **`with`** construct. For example,

```
r with { x = 1 }
```

evaluates to a new record that adds an `x` field to the record bound to `r`. If `r` already has an `x` field, then the field is replaced, otherwise the new record has one more field than `r`.

3 An example

Here is a simple example of an MLR program.

```
type salary = { salary : int }
fun salary (x : salary) : int = x.salary
fun isWealthy (x : salary) : bool = (100000 <= salary x)
fun main (args : string list) : int =
  let r = { name = "Susan", age = 21, salary = 34000 }
  in IO.println (String.btoa isWealthy r)
```

In this example, `IO` and `String` are variables bound to globally defined records, which serve the role of *modules*.

4 MLR syntax

The following is the collected syntax of MLR. We assume the following kinds of terminal symbols: *identifiers*, which are used for types (tid), labels (lid), and variables (vid), integer literals (num), and string literals (str).

Prog
::= *TopDecl*$^+$

TopDecl
::= **`type`** tid **`=`** *Type*
| *Function*

Type
::= *AtomicType* **`->`** *Type*
| *AtomicType*

AtomicType
::= **`bool`**
| **`int`**
| **`string`**
| tid
| *AtomicType* **`list`**
| **`{`** *RowType*^{opt} **`}`**
| **`(`** *Type* **`)`**

RowType
::= lid **`:`** *Type* (**`,`** lid **`:`** *Type*)*

Function
::= **`fun`** vid *Param*$^+$ **`:`** *Type* **`=`** *Exp*

Param
::= **`(`** vid **`:`** *Type* **`)`**

Exp

::= **let** vid **:** *Type* = *Exp* **in** *Exp*
| *Function* **in** *Exp*
| **if** *Exp* **then** *Exp* **else** *Exp*
| *Exp* **with {** *RowExp* **}**
| *Exp* **andalso** *Exp*
| *Exp* **orelse** *Exp*
| *Exp* **==** *Exp*
| *Exp* **<=** *Exp*
| *Exp* **<** *Exp*
| *Exp* **::** *Exp*
| *Exp* **+** *Exp*
| *Exp* **-** *Exp*
| *Exp* ***** *Exp*
| *Exp* **/** *Exp*
| *Exp* **%** *Exp*
| *Exp* *Exp*
| **-** *Exp*
| **not** *Exp*
| **isnull** *Exp*
| **hd** *Exp*
| **tl** *Exp*
| *Exp* **.** lid
| **true**
| **false**
| num
| str
| vid
| **(** *Exp* (**;** *Exp*)* **)**
| **[** (*Exp* (**,** *Exp*)*)^{opt} **]**
| **{** *RowExp*^{opt} **}**

RowExp

::= lid **=** *Exp* (**,** lid **=** *Exp*)*