CMSC 22610	**Implementation**	**Handout 2**
Winter 2007	**of**	**January 4, 2007**
	Computer Languages	

Project overview

1 Introduction

The project for this course is the implementation of a subset of the Standard ML language, called *MinML*. 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. 4	Lexer	Jan. 19
Jan. 18	Parser	Jan. 29
Jan. 30	Typechecker	Feb. 19
Feb. 15	Code generation and runtime	Mar. 7

2 MinML

MinML is a subset of Standard ML, with many features removed or simplified. Here is a list of the major differences:

- MinML does not have a module system.
- MinML does not have **`abstype`**, **`exception`**, or **`withtype`** declarations.
- MinML does not have record types.
- MinML does not have the equational syntax for function definitions.
- Patterns in MimML are single-level.
- There are no anonymous functions in MinML.
- Equality in MinML is only defined on non-functional monotypes.

- Except for equality, MinML does not support overloading.
- MinML has a much smaller basis library.
- MinML does not have references or arrays.
- Identifiers in MinML are alphanumeric (no symbolic identifiers).

To simplify the language, we have removed modules and exceptions and simplified many other features of the language.

2.1 Types and values

MinML supports three primitive types of values: booleans, integers, and strings. There is a built-in list-type constructor, as well as the usual function and tuple-type constructors.

2.2 Declarations

A MinML program is a sequence of top-level declarations, which are either type, datatype, value, or function definitions, followed by an expression.

3 MinML syntax

The following is the collected syntax of Objective ML. We assume the following kinds of terminal symbols: *identifiers*, which are used for types (tyid), data constructors (conid), and variables (vid); *type variables* (tyvar); integer literals (num); and string literals (str).

Prog
  ::= (*TopDecl* **;**)* *Exp*

TopDecl
  ::= **type** *TypeParams*^{opt} tyid **=** *Type*
  | **datatype** *TypeParams*^{opt} tyid **=** *ConsDecl* (**|** *ConsDecl*)*
  | *ValueDecl*

TypeParams
  ::= *tyvar*
  | **(** *tyvar* (**,** *tyvar*)* **)**

Type
  ::= *TupleType* **->** *Type*
  | *TupleType*

TupleType
  ::= *AtomicType* (***** *AtomicType*)*

AtomicType
::= tyid
| tyvar
| *AtomicType tyid*
| `(` *Type* (`,` *Type*)* `)` tyid
| `(` *Type* `)`

ConsDecl
::= conid (**`of`** *Type*)^{opt}

ValueDecl
::= **`val`** *TuplePat* `=` *Exp*
| **`fun`** *FunDef* (**`and`** *FunDef*)*

FunDef
::= vid *TuplePat* `=` *Exp*

Exp
::= **`let`** *ValueDecl*$^{+}$ **`in`** *Exp* (`;` *Exp*)* **`end`**
| **`if`** *Exp* **`then`** *Exp* **`else`** *Exp*
| **`case`** *Exp* **`of`** *Match* (`|` *Match*)*
| *Exp* **`andalso`** *Exp*
| *Exp* **`orelse`** *Exp*
| *Exp* `=` *Exp*
| *Exp* `<=` *Exp*
| *Exp* `<` *Exp*
| *Exp* `::` *Exp*
| *Exp* `@` *Exp*
| *Exp* `+` *Exp*
| *Exp* `-` *Exp*
| *Exp* `*` *Exp*
| *Exp* **`div`** *Exp*
| *Exp* **`mod`** *Exp*
| *Exp Exp*
| `~` *Exp*
| *Const*
| vid
| `(` *Exp* (`,` *Exp*)* `)`
| `(` *Exp* (`;` *Exp*)* `)`

Match
::= *Pat* `=>` *Exp*

Pat
::= *Const*
| conid *TuplePat*
| *TuplePat*

TuplePat
  ::= *AtomicPat*
  | **(** *AtomicPat* (**,** *AtomicPat*)* **)**

AtomicPat
  ::= vid
  | _

Const
  ::= num
  | str
  | conid

The associativity and precedence of operators is as in Standard ML.