CMSC 22610 **Winter 2011**

Implementation of Computer Languages I

Project 2 **January 20, 2011**

MinML parser

Due: February 7, 2011

1 Introduction

Your second assignment is to implement a parser for MinML. You may either use ML-Yacc or ML-Antlr to generate your parser from an specification. ML-Yacc is described in Chapter 3 of Appel's book and there is a link to the manual on the course web page. ML-Antler is a new tool that supports LL(k) parsing. Its manual is also available from the course web page. The actions of this parser will construct a *parse tree* representation for an MinML program. We have provided an ML-Lex based scanner and will also provide the definition of the parse-tree representation. You may also choose to use your own scanner from Project 1.

2 The MinML grammar

The concrete syntax of MinML is specified by the grammar given in Figures 1 and 2.

There are four classes of identifiers: *vid* for value identifiers (variables), *cid* for data constructor identifiers, *tyv* for type variables, and *tyc* for type constructors (including type constants). The identifier classes *vid* and *tyv* consist of alphanumeric identifiers starting with a lower-case letter, while *cid* and *tyc* range over alphanumeric identifier starting with an upper-case letter.

As written, this grammar is ambiguous. To make this grammar unambiguous, the precedence of operators must be specified. The precedence of the binary operators are (from weakest to strongest):

```
        ||
        &&
== <> < <= > >=
         @
        + -
       * / %
```

All binary operators are left associative except "@" (string concatenation) which is right associative. The next highest precedence is function application, which associates to the left. Here are some examples:

```
a + b * c + d  ≡  (a + (b * c)) + d
f a @ b @ "    ≡   (f a) @ (b @ "")
hd l x y       ≡       ((hd l) x) y
```

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

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

TypeParams
  ::= *tyv*
  | **`(`** *tyv* (**`,`** *tyv*)* **`)`**

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

AtomicType
  ::= tyv
  | tyc (**`(`** *Type* (**`,`** *Type*)* **`)`**)^{opt}
  | **`(`** *Type* **`)`**

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

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

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

Figure 1: The concrete syntax of MinML (A)

Const
    ::= num
    | str
    | cid

Pat
    ::= *Const*
    | cid *TuplePat*
    | *TuplePat*

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

AtomicPat
    ::= vid
    | _

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

Exp
    ::= vid
    | *Const*
    | *Exp* **`||`** *Exp*
    | *Exp* **`&&`** *Exp*
    | *Exp* **`==`** *Exp*
    | *Exp* **`<>`** *Exp*
    | *Exp* **`<`** *Exp*
    | *Exp* **`<=`** *Exp*
    | *Exp* **`>`** *Exp*
    | *Exp* **`>=`** *Exp*
    | *Exp* **`@`** *Exp*
    | *Exp* **`+`** *Exp*
    | *Exp* **`-`** *Exp*
    | *Exp* **`*`** *Exp*
    | *Exp* **`/`** *Exp*
    | *Exp* **`%`** *Exp*
    | **`~`** *Exp*
    | *Exp* *Exp*
    | **`(`** *Exp* (**`,`** *Exp*)* **`)`**
    | **`(`** *Exp* (**`;`** *Exp*)* **`)`**
    | **`if`** *Exp* **`then`** *Exp* **`else`** *Exp*
    | **`let`** *ValueDecl*$^+$ **`in`** *Exp* (**`;`** *Exp*)* **`end`**
    | **`case`** *Exp* **`of`** *Match* (**`|`** *Match*)* **`end`**
    | **`fn`** vid **`=>`** *Exp*

Figure 2: The concrete syntax of MinML (B)

3 Requirements

Your implementation should consist of the following five files:

`parser.cm` — a CM sources file for compiling your project.

`parser.sml` — An SML source file containing the definition a structure `Parser`, that defines a function

```
    val parseFile : string -> Syntax.program
```

where `Syntax.program` is the type of program parse trees. This function should open the named source file, parse it, and return the resulting syntax tree.

`symbol.sml` — An provided SML file containing a module `Symbol` that defines the symbol type used for identifiers in syntax trees.

`syntax.sml` — An provided SML file containing a module `Syntax` that defines the syntax tree representation of MinML programs. This representation is to be produced by the parser.

`minml.grm` — An ml-yacc parser specification file for parsing MinML programs. The actions of this parser should construct syntax trees (*i.e.*, values of type Syntax.program).

`minml.lex` — An ml-lex specification file for lexing MinML. This file can be found in the directory `solutions/project1/lexer3` in your project repository.

We have set up a directory project2 for each student in their phoenixforge svn repository, and will seed the project with the files mentioned above. You should use this repository to hold the source for your project. We will collect the projects at 11pm on Tuesday, Februay 8th from the SVN repositories, so make sure that you have committed your final version before then.