CMSC 22610	**Implementation of**	**Project 2**
Winter 2015	**Computer Languages I**	**January 22, 2015**

Flang parser

Due: February 6, 2015

1 Introduction

The second project is to implement a parser for **Flang**, which will convert a stream of tokens into an *parse tree*. The grammars for most programming languages are of sufficient complexity that such components of a compiler are best written using a *parser generator*; *i.e.*, an external tool that takes the specification of a grammar and produces code for a corresponding parser. (Parser generators can also analyze the grammar sepecification and identify potential ambiguities.)

For this project, we will use ML-Antlr, which is an LL(k) based parser generator. ML-Antlr is documented as part of the ML-LPT Manual, which is linked to on the course web site. The project seed code includes a ML-ULex based scanner (ML-ULex is also part of ML-LPT), but you may also adapt your Project 1 lexer to work with ML-Antlr. The seed code also includes a module defining the parse-tree representation and a skeleton of the ML-Antlr specification.

2 Requirements

We will seed a directory, called `proj2`, in your `phoenixforge` repository. This directory will contain the following files:

`basis-names.sml` — defines `Atom.atom` values for the binary operators.

`error.sml` — An SML source file that supports error reporting.

`flang.grm` — This file contains a skeleton of an ML-Antlr parser specification file for parsing **Flang** programs.

`flang.lex` — An ML-ULex lexer specification for scanning **Flang**. You may choose to use this scanner for your project, or, for extra credit, you may choose to use the lexer you wrote for Part 1 of the project (doing so will require some restructuring of the Project 1 code as described in Section 5).

`flang-hand-lexer.sml` — Defines a wrapper around a hand-written scanner (if you choose to use your Project 1 code).

`flang-hand-scanner-sig.sml` — Defines a signature `FLANG_HAND_SCANNER` that your Project 1 lexer must match (if you choose to use your Project 1 code).

`parser.sml` — An SML source file containing the definition a structure `Parser`, that combines the lexer and parser into a single function for parsing files.

`parse-tree.sml` — An SML file containing the module `ParseTree` that defines the parse-tree representation of MinML programs.

`print-parse-tree.sml` — An SML file containing the module `PrintParseTree` for printing parse trees using an S-expression syntax.

`sources.cm` — A CM sources file for compiling your project. You will need to modify this file if you choose to use your code from Project 1.

In addition, you should add any other files that you need to your repository.

Your task is to complete the `flang.grm` file according to the definition of **Flang** as described in Section 3.

We will collect the projects at **10pm** on **Friday February 6th** from the SVN repositories, so make sure that you have committed your final version before then.

3 Flang Grammar

Literal symbols, such as keywords and punctuation, are written in a **`bold fixed-width font`**, other terminal symbols are written in roman font, and non-terminal symbols are written in *italic font*. We use the following terminal symbols in the grammar:

Terminal	Lexical class	Description
TyId	upper-case identifier	type constructor and data types
TyVar	lower-case identifier	type variable
ConId	upper-case identifier	data constructor
ValId	lower-case identifier	value identifier

The concrete syntax of **Flang** is specified by the following context free grammar:

Program
  ::= (*Definition* **`;`**)* *Exp*

Definition
  ::= **`type`** TyId *TypeParams*^{opt} **`=`** *Type*
  | **`data`** TyId *TypeParams*^{opt} **`with`** *ConDef*$^+$
  | *ValBind*

ConDef
  ::= **`con`** ConId (**`of`** *Type*)^{opt}

TypeParams
  ::= **`[`** TyVar (**`,`** TyVar)* **`]`**

```
Type
    ::=   TypeParams Type
     |    Type -> Type
     |    Type (* Type)^+
     |    TyId TypeArgs^opt
     |    TyVar
     |    ( Type )

TypeArgs
    ::=   [ Type (, Type)^* ]

ValBind
    ::=   fun ValId FunParam^+ -> Type = Exp
     |    let SimplePat (: Type)^opt = Exp
     |    Exp

FunParam
    ::=   TypeParams
     |    ( ValId : Type )

Exp
    ::=   if Exp then Exp else Exp
     |    Exp == Exp
     |    Exp < Exp
     |    Exp <= Exp
     |    Exp :: Exp
     |    Exp @ Exp
     |    Exp + Exp
     |    Exp - Exp
     |    Exp * Exp
     |    Exp / Exp
     |    Exp % Exp
     |    ApplyExp

ApplyExp
    ::=   AtomicExp
     |    ApplyExp AtomicExp
     |    ApplyExp TypeArgs

AtomicExp
    ::=   ValId
     |    ConId
     |    Int
     |    String
     |    ( Exp (, Exp)^* )
     |    { Scope }
     |    case Exp of MatchCase^+ end

Scope
    ::=   (ValBind ;)^* Exp

MatchCase
    ::=   { Pat => Scope }
```

Pat
$::=$ *SimplePat*
$|$ ConId *SimplePat*^{opt}
$|$ *SimplePat* `::` *SimplePat*
$|$ `(` *SimplePat* (`,` *SimplePat*)* `)`

SimplePat
$::=$ ValId
$|$ `_`

The grammar as written has some ambiguities, which are resolved by specifying the precedence and associativity of operators.

For types, the `->` constructor associates to the right and has lower precedence than the tuple-type constructor (`*`). Type abstraction has the lowest precedence.

Conditional expressions have the lowest precedence, followed by infix binary expressions. For binary expressions, all operators are left associative, except the infix list cons operator `::`, which is right associative. Binary operations are groups into five precedence levels from lowest to highest as follows:

- relational operators: `==`, `<`, and `<=`
- list operator: `::`
- string operator: `@`
- addition operators: `+` and `-`
- multiplication operators: `*`, `/`, and `%`

To understand how to apply the precedence of productions to resolve ambiguity, consider two productions for *Exp*. such that the first ends with an *Exp* and the second starts with an *Exp*:

Exp
$::=$ `if` *Exp* `then` *Exp* `else` *Exp*
$|$ *Exp* `+` *Exp*

Suppose that we must parse the sequence:

$\cdots$ `if` $\cdots$ `then` $\cdots$ `else` *Exp* `+` $\cdots$

where *Exp* stands for a token sequence that has already been determined to be an *Exp* (if necessary, by applying precedence and associativity resolution). The higher precedence of the *Exp* `+` *Exp* production dictates that the sequence should be parsed as:

$\cdots$ `if` $\cdots$ `then` $\cdots$ `else` (*Exp* `+` $\cdots$)   correct

and not as:

($\cdots$ `if` $\cdots$ `then` $\cdots$ `else` *Exp*) `+` $\cdots$ incorrect

The latter parse requires explicit parentheses.

The associativity of keywords and operators resolves ambiguity among productions of the same precedence. Suppose we must parse the sequence:

$$\cdots \; \mathit{Exp}_1 \texttt{ :: } \mathit{Exp}_2 \texttt{ :: } \mathit{Exp}_3 \; \cdots$$

where Exp_1, Exp_2, and Exp_3 stand for token sequences that has already been determined to be *Exp*s. The right associativity of the `::` operator dictates that the sequence should be parsed as:

$$\cdots \; \mathit{Exp}_1 \texttt{ :: } (\; \mathit{Exp}_2 \texttt{ :: } \mathit{Exp}_3 \;) \; \cdots \qquad \text{correct}$$

and not as:

$$\cdots \; (\; \mathit{Exp}_1 \texttt{ :: } \mathit{Exp}_2 \;) \texttt{ :: } \mathit{Exp}_3 \; \cdots \qquad \text{incorrect}$$

The latter parse requires explicit parentheses.

Here are some more examples:

```
           if b1 then x                  if b1 then x
           else if b2 then y        ≡    else (if b2 then y
           else z + w                    else (z + w))
               a + b * c + d        ≡    (a + (b * c)) + d
   "i␣=␣" @ intToString i @ "\n"    ≡    ("i␣=␣" @ (intToString i)) @ "\n"
         [a][b] a -> b -> a * b     ≡    [a] ([b] (a -> (b -> (a * b))))
    fst [Integer] [Bool] 1 False    ≡    (((fst [Integer]) [Bool]) 1) False
        [a] a -> [b] b -> a * b     ≡    [a] (a -> ([b] (b -> (a * b))))
```

4 Errors

ML-Antlr utilizes a parsing algorithm that integrates automatic error repair. Hence, your parser specification need not explicitly support error reporting. ML-Antlr does support declarations for improving error recovery, which you are welcome to include in your specification. The automatic error repair mechanisms require that semantic actions be free of significant side effects, because error repair may require executing a production's semantic action multiple times. All of the functions in the `ParseTree` structure are pure; thus, they may be freely used in semantic actions.

In order to support error reporting in the type-checker (to be implemented in Project 3), the abstract parse tree must be annotated with position information. Therefore, each node in the parse tree is constructed with a *source span* that pairs the left and right source positions of the node. For example, you might have the following rule for matching a constant as an *AtomicExp* in your grammar:

```
AtomicExp : NUMBER
        => (PT.ExpMark{span = NUMBER_SPAN, tree = PT.IntExp NUMBER});
```

Source positions and spans of terminals are provided by the scanner. Consult the ML-LPT manual for information for more information about source positions and accessing position information in semantic actions.

5 Extra Credit: Integrating a Hand-Written Scanner

For extra credit, you may choose to adapt your hand-written scanner from Project 1 for use in Project 2. To do so, you will need to add support for the `%` operator that was omitted from the

Project 1 description. You will also need extend your implementation to include position information for tokens. You should copy your Project 1 code into your Project 2 directory and rename your `FLangLex` structure to `FLangHandScanner`. Remember to add your source code to the **svn** repository and to the `sources.cm` file! You will also need to modify it to match the following signature:

```
signature FLANG_HAND_SCANNER =
  sig
    val lexer : {
                getPos : 'strm -> Error.pos,
                forward : Error.pos * int -> Error.pos,
                reportErrorAt : Error.pos * string -> unit
              } -> (char, 'strm) StringCvt.reader
                -> (FLangTokens.token * Error.span, 'strm) StringCvt.reader
  end
```

Note that `lexer` is now a function that takes a character reader and returns a `(token * span)` reader. To support position information and error reporting, the `FLangHandScanner.lexer` function takes an initial record argument with the following components:

- a `getPos : 'strm -> Error.pos` function for querying the current position of the input character stream,
- a `forward : Error.pos * int -> Error.pos` function for computing the position n characters forward from a given position, and
- a `reportErrorAt : Error.pos * string -> unit` function for reporting an error at a given position.

Figure 1 sketches how a hand-written scanner should use `getPos` to get the left position of a token and `forward` to compute the right position of a token.

The project seed code includes a file `flang-hand-lexer.sml` that wraps your scanner (*i.e.*, the `FLangHandScanner` module) with an interface that is compatible with the ML-Antlr generated parser. If you choose to use your Project 1 scanner, then you will need to include this file in your `sources.cm` file (and omit the `flang.lex` file).

Document History

February 5, 2015 Fixed typo in rule for *Scope*.

February 5, 2015 Fixed typo in syntax of tuple expressions.

February 5, 2015 Fixed small typos in grammar (*TyParams* should be *TypeParams* and *Expr* should be *Exp*).

January 31, 2015 Fixed typo in grammar rule for *TypeParams*.

January 25, 2015 Fixed type of `lexer` in `FLANG_HAND_SCANNER` signature and example code. Also added another example of precedence rules and fixed a typo.

January 23, 2015 Original version

```
fun lexer {
        getPos : 'strm -> Error.pos,
        forward : Error.pos * int -> Error.pos,
        reportErrorAt : Error.pos * string -> unit
      }
      (getc : (char, 'strm) StringCvt.reader) :
      (FlangTokens.token * Error.span, 'strm) StringCvt.reader = let
      ...
      fun scan strm0 = let
            val pos0 = getPos strm0
            in
              case getc strm0
               of NONE => NONE
                | SOME(#"+", strm1) =>
                    SOME((T.PLUS, (pos0, forward (pos0, 1))), strm1)
                | ...
                | SOME(c, strm1) => if Char.isUpper c then ...
                    else if Char.isLower c then ...
                    else if Char.isDigit c then ...
                    else (
                      reportErrorAt (pos0, concat[
                          "bad␣character␣'", Char.toString c, "'"
                        ]);
                      scan strm1)
              (* end case *)
            end
      ...
      in
        scan
      end
```

Figure 1: Skeleton hand-written scanner with position information