1 Introduction

Your first assignment is to implement a scanner and a parser for LangF, which will convert an input stream of characters into a parse tree. You will use the ML-Ulex scanner generator and ML-Antlr parser generator (collectively the ML Language Processing Tools) for this assignment. These tools documented in the ML-LPT Manual, which is linked to on the course web site.

2 Requirements

We will seed a directory, called proj1, in your phoenixforge repository. This directory will contain five sub-directories:

bin — holds the lfc.sh script for running the compiler.

common — common utility code that will be used across the project

driver — main program for connecting phases and running the compiler

parse-tree — the representation of the parse tree

parser — the scanner and parser

as well as a Makefile. You should read and try understand the supplied code, since it will provide more familiarity with SML.

Your task is to complete the langf.lex and langf.grm files that are located in the parser directory so as to implement the grammar of LangF described below. You should not have to make changes to any of the other files.

We recommend that you proceed by first running ML-Antlr on the langf.grm file (it is sufficient to run make to do this). Once you have done that step, you can write and test the lexer (the lexer depends on the token datatype that is generated by ML-Antlr). The scanner can be tested by supplying the "--test-scanner" flag to the lfc.sh command, which will cause the token stream to be printed to a file. Once you have a working scanner, we suggest that you write the parser without actions and get ML-Antlr to accept your grammar. The last step will then be to add actions to the ML-Antlr specification that build the parse tree.
3 The LangF scanner

You will use ML-ULex to implement the scanner for LangF. Your scanner should handle the lexical properties of the language as described in Section 5.1.

4 The LangF parser

ML-Antrlr utilizes a parsing algorithm that integrates automatic error repair. Hence, your parser specification need not explicitly support error reporting. ML-Antrlr 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 2), the parse tree must be annotated with source-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, consider the following rules for atomic expressions:

```
AtomicExp
    : NUMBER
    => (PT.MarkExp{span = NUMBER_SPAN, tree = PT.IntExp NUMBER})
    | "(" Exp ")" 
    => (PT.MarkExp{span = FULL_SPAN, tree = Exp})
```

here, FULL_SPAN span specifies the span that covers the full width of the parse (i.e., the span starting with the position of the left parenthesis and ending with the right parenthesis). Source positions and spans of terminals are automatically 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 The collected syntax of LangF

5.1 Lexical issues

There are five classes of tokens in LangF:

1. lower-case identifiers: a, b, toString, y23, etc.
2. upper-case identifiers: X, Foo, SOME_VAL, etc.
3. numbers: 0, 42, etc.
4. strings: "hello world", "some\ntext", etc.
5. various delimiters and operators: (,), =, <=, +, etc.

Tokens can be separated by whitespace and/or comments.
Identifiers in **LangF** can be any string of letters, digits, underscores (ASCII code 95), or single quote characters (ASCII code 39), beginning with a letter. Identifiers are case-sensitive; we use *LC-identifier* to refer to those that begin with a lower-case letter (e.g., **foo**) and *UC-identifier* to refer to those that begin with an upper-case letter (e.g., **Foo**). The following LC-identifiers are reserved as keywords:

```
case  data  else  end  fun
if  let  of  then  type
```

Numbers in **LangF** are represented as sequences of decimal digits. Negative numbers are using the unary negation operator (**−**); the minus sign is not part of the literal token.

String literals are delimited by matching double quotes and can contain spaces (ASCII code 32) and any unescaped graphical character except “\"” (ASCII code 92) or “\”” (ASCII code 34). In addition, the following escape sequences are permitted:

- `\n` — newline (ASCII code 10)
- `\r` — carriage return (ASCII code 13)
- `\t` — horizontal tab (ASCII code 8)
- `\\` — backslash
- `\"` — quotation mark

A character in a string literal may also be specified by its numerical value using the escape sequence ‘`\ddd`’, where `ddd` is a sequence of three decimal digits. Strings in **LangF** may contain any 8-bit value other than zero; thus ‘`\000`’ is not a legal escape code.

The delimiters and operators of **LangF** are the following:

```
(  ) [  ] {  } := || && == != <= < :: ^ + - * / % != = , ; : | -> => _
```

Comments in **LangF** are either **end-of-line**, which start with “//” and extend until the next newline character (or end-of file), or **block comments**, which start with “/*” and are terminated with a matching “*/”. Note that block comments may nest (as in SML) and that comments cannot start inside a string literal.

Whitespace is any non-empty sequence of spaces (ASCII code 32), horizontal or vertical tabs, form feeds, newlines, or carriage returns. Any other non-printable character is treated as an error.

### 5.2 Grammar

The collected syntax of **LangF** given below using an extended-BNF format. 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:
<table>
<thead>
<tr>
<th>Terminal</th>
<th>Lexical class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tyvar</td>
<td>LC-identifier</td>
<td>type-variable name</td>
</tr>
<tr>
<td>tyid</td>
<td>UC-identifier</td>
<td>type name</td>
</tr>
<tr>
<td>conid</td>
<td>UC-identifier</td>
<td>data-constructor name</td>
</tr>
<tr>
<td>varid</td>
<td>LC-identifier</td>
<td>variable name</td>
</tr>
<tr>
<td>num</td>
<td>number</td>
<td>integer literal</td>
</tr>
<tr>
<td>str</td>
<td>string</td>
<td>string literal</td>
</tr>
</tbody>
</table>

The collected syntax of **LangF** is presented in Figures 1 (Declarations and Types) and 2 (Expressions and Patterns). We use parentheses for grouping, when necessary, and superscript “∗∗” for zero-or-more items, superscript “∗” for one-or-more items, and superscript “opt” for zero-or-one items.

### 5.3 Syntactic conventions

The syntax of expressions in Figure 2 is ambiguous, but we resolve the ambiguities by specifying the precedence and associativity of operators.\(^1\) Expression forms are grouped into ten precedence levels from lowest to highest as follows:

- **if-then-else** expression
- assignment: `:=`
- conditional or-else operator: `||`
- conditional and-also operator: `&&`
- relational operators: `==`, `!=`, `<`, and `<=`
- list-cons operator: `::`
- string-concatenation operator: `^`
- addition operators: `+` and `−`
- multiplication operators: `∗`, `/`, and `%`
- unary operators: `−`, `!`

And all binary operators **except** list-cons (“`::`”) are left associative. For example, “`a+b−c`” parses as “`(a+b)−c`,” while “`a::b::c`” parses as “`a::(b::c)`.”

### 6 Submission

We will collect the projects at **11pm on October 20, 2020** from the SVN repositories, so make sure that you have committed your final version before then.

---

\(^1\)You will also need to do some factoring of the grammar to make it satisfy the LL(k) requirements of ML-Antlr.
Program
::= Definition ( ; Definition)*

Definition
::= type tyid TypeParamsopt = Type
    | data tyid TypeParamsopt = ConDef ( | ConDef)*
    | ValBind

ConDef
::= conid (of Type)opt

TypeParams
::= [ tyvar ( , tyvar)* ]

Type
::= TypeParams Type
    | Type -> Type
    | Type ( * Type)+
    | tyid TypeArgsopt
    | tyvar
    | ( Type )

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

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

FunParam
::= TypeParams
    | ( varid : Type )

Figure 1: LangF Grammar: Declarations and Types
Exp ::= if Exp then Exp else Exp
     | OpExp := OpExp
     | OpExp

        | OpExp && OpExp
        | OpExp == OpExp
        | OpExp != OpExp
        | OpExp < OpExp
        | OpExp <= OpExp
        | OpExp : OpExp
        | OpExp ^ OpExp
        | OpExp + OpExp
        | OpExp - OpExp
        | OpExp * OpExp
        | OpExp / OpExp
        | OpExp % OpExp
        | - ApplyExp
        | ! ApplyExp
        | ApplyExp

ApplyExp ::= AtomicExp
            | ApplyExp AtomicExp
            | ApplyExp TypeArgs

AtomicExp ::= varid
            | conid
            | Int
            | String
            | ( (Exp (Exp)* opt )
            | { Scope }
            | case Exp of MatchCase* end

Scope ::= (ValBind ;)* Exp

MatchCase ::= { Pat => Scope }

Pat ::= SimplePat
      | conid SimplePat\opt
      | SimplePat : SimplePat
      | ( SimplePat ( SimplePat\opt )

SimplePat ::= varid
            | _

Figure 2: LangF Grammar: Expressions and Patterns
Important note: You are expected to submit code that compiles and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will not receive any points for correctness.

7 Document history

October 18, 2020  Added integer inequality (\(!=\)) to the OpExp grammar rules.

October 18, 2020  Fixed inconsistent use of valid (should be varid).

October 13, 2020  Fixed syntax of tuple expressions to include the shorthand for the Unit constructor (as described in the Project Overview).

October 13, 2020  Fixed syntax of programs to require a semicolon between definitions.

October 7, 2020  Original version.