CMSC 22610 | **Implementation of** | **Project 2**
Winter 2009 | **Computer Languages - I** | **January 22, 2009**

LangF Parser

Due: February 6, 2009

1 Introduction

The second project is to implement a parser for LangF, which will convert a stream of tokens into an *abstract 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*, 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.) You may use either ML-Yacc or ML-Antlr to generate your parser; both tools target Standard ML. ML-Yacc generates parsers that use an LALR(1) parsing algorithm, while ML-Antlr generates parsers that use an LL(k) parsing algorithm. There are links to the manuals for both tools in the Resources section of the course web site; ML-Yacc is also described in Chapter 3 of *Modern Compiler Implementation in ML*. The project seed code will provide an ML-ULex based scanner (but you may also adapt your hand-written scanner from Project 1), modules implementing the parse-tree representation, and skeleton grammar specifications for both ML-Yacc and ML-Antlr.

2 LangF Grammar

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

If one ignores the parenthetical annotations, then the grammar is ambiguous in the *Type* and *Exp* nonterminals. In order to make the grammar unambiguous, the parenthetical annotations specify the precedence and associativity of the *Type* and *Exp* productions. The *Type* and *Exp* productions are given in order of increasing precedence; higher precedence productions bind more tightly than lower precedence productions. L (resp. R) indicates left (resp. right) association of a keyword or operator.

To understand how to apply the precedence of productions to resolve ambiguity, we take *Exp* as an example. Consider two productions, such that the first ends with an *Exp* and the second starts with an *Exp*. For example, consider:

$$\mathit{Exp} \rightarrow \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \qquad \text{and} \qquad \mathit{Exp} \rightarrow \mathit{Exp}\ \texttt{+}\ \mathit{Exp}$$

Suppose that we must parse the sequence:

$$\cdots\ \cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ \mathit{Exp}\ \texttt{+}\ \cdots\ \cdots$$

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 *Exp* associate to the right; that is, the sequence should be parsed as:

$$\cdots\ \cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ (\ \mathit{Exp}\ \texttt{+}\ \cdots\ )\ \cdots \qquad \text{correct}$$

and not as:

$$\cdots\ (\ \cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ \mathit{Exp}\ )\ \texttt{+}\ \cdots\ \cdots \qquad \text{incorrect}$$

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:

$$\cdots\ \cdots\ \mathit{Exp}_1\ \texttt{\^{}}\ \mathit{Exp}_2\ \texttt{\^{}}\ \mathit{Exp}_3\ \cdots\ \cdots$$

```
Prog
    ::=  Exp
     |   (Decl)* ; Exp

Decl
    ::=  type tyconid TypeParams = Type
     |   datatype DataDecl (and DataDecl)*
     |   val SimplePat (: Type)? = Exp
     |   fun FunDecl (and FunDecl)*

TypeParams
    ::=
     |   [ (tyvarid (, tyvarid)*)? ]

Type
    ::=  [ tyvarid ] -> Type                                    (lowest precedence)
     |   Type -> Type                                                         (R)
     |   tyconid TypeArgs
     |   tyvarid
     |   ( Type )                                              (highest precedence)

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

DataDecl
    ::=  tyconid TypeParams = DaConDecl (| DaConDecl)*

DaConDecl
    ::=  daconid DaConArgTys

DaConArgTys
    ::=
     |   { (Type (, Type)*)? }

SimplePat
    ::=  varid
     |   _

FunDecl
    ::=  varid Param+ : Type = Exp

Param
    ::=  ( varid : Type )
     |   [ tyvarid ]
```

Figure 1: The concrete syntax of LangF (A)

```
Exp
    ::=   fn Param+ => Exp                                          (lowest precedence)
     |    if Exp then Exp else Exp
     |    Exp orelse Exp                                                          (L)
     |    Exp andalso Exp                                                         (L)
     |    Exp : Type
     |    Exp op Exp          op ∈ {==, <>, <, <=, >, >=}                         (L)
     |    Exp ^ Exp                                                               (R)
     |    Exp op Exp          op ∈ {+, -}                                         (L)
     |    Exp op Exp          op ∈ {*, /, %}                                      (L)
     |    ~ (AtomicExp | daconid)
     |    daconid TypeArgs DaConArgs
     |    ApplyExp                                                 (highest precedence)

DaConArgs
    ::=
     |    { (Exp (, Exp)*)? }

ApplyExp
    ::=   ApplyExp ApplyArg
     |    AtomicExp

ApplyArg
    ::=   (AtomicExp | daconid)
     |    [ Type ]

AtomicExp
    ::=   varid
     |    integer
     |    string
     |    ( Exp (; Exp)+ )
     |    let Decl+ in Exp (; Exp)* end
     |    case Exp of MatchRule (| MatchRule)* end
     |    ( Exp )

MatchRule
    ::=   Pat => Exp

Pat
    ::=   daconid TypeArgs DaConPats
     |    SimplePat

DaConPats
    ::=
     |    { (SimplePat (, SimplePat)*)? }
```

Figure 2: The concrete syntax of LangF (B)

where Exp_1, Exp_2, and Exp_3 stand for token sequences that has already been determined to be *Exp*s (if necessary, by applying precedence and associativity resolution). The right associativity of the **^** operator dictates that the sequence should be parsed as:

$$\cdots\ \cdots\ Exp_1 \;\mathbf{\hat{}}\; (\ Exp_2 \;\mathbf{\hat{}}\; Exp_3\ )\ \cdots\ \cdots \qquad \text{correct}$$

and not as:

$$\cdots\ \cdots\ (\ Exp_1 \;\mathbf{\hat{}}\; Exp_2\ ) \;\mathbf{\hat{}}\; Exp_3\ \cdots\ \cdots \qquad \text{incorrect}$$

The latter parse requires explicit parentheses.

Here are some examples:

```
     b1 orelse b2 : Bool : Bool     ≡    b1 orelse ((b2 : Bool) : Bool)
 b1 andalso b2 : Bool orelse b3     ≡    (b1 andalso (b2 : Bool)) orelse b3
                  a + b * c + d     ≡    (a + (b * c)) + d
    "i = " ^ intToString i ^ "\n"   ≡    "i = " ^ ((intToString i) ^ "\n")
          ['a] -> 'a -> 'a -> 'a    ≡    ['a] -> ('a -> ('a -> 'a))
     fst [Integer] [Bool] 1 False   ≡    (((fst [Integer]) [Bool]) 1) False
```

3 Requirements

You should complete either the ML-Yacc (`langfc-src/parser/langf-yacc.grm`) or the ML-Antlr (`langfc-src/parser/langf-antlr.grm`) grammar specification. In addition to writing a grammar specification for LangF, your grammar specification should include semantic actions that construct an abstract parse tree representation of the input LangF program. The **`structure`** `ParseTree : PARSE_TREE` module is provided in the seed code; the `PARSE_TREE` signature implementation is at `langfc-src/parse-tree/parse-tree.sig` and the `ParseTree` structure implementation is at `langfc-src/parse-tree/parse-tree.sml`. Your parser should return a value of type `ParseTree.Prog.t`.

The project seed code includes a compiler control (`-Cparser=yacc` / `-Cparser=antlr`) that selects between the ML-Yacc parser and the ML-Antlr parser. After deciding between ML-Yacc and ML-Antlr, you should change the default setting to match your chosen parser. This default is specified by the `parserCtl` value in the `langfc-src/parser/wrapped-parser.sml` file (lines 36 – 42).

3.1 Errors

Both ML-Yacc and ML-Antlr utilize parsing algorithms that integrate automatic error repair. Hence, your parser specification need not explicitly support error reporting. (ML-Yacc does support declarations for improving error recovery, which you are welcome to include in your specification.) However, 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 object in the parse tree is constructed with a *source span* (`Source.Span.t`), which pairs the left and right source positions (`Source.Pos.t`) of the object. The `Source: SOURCE` module is provided in the seed code; the `SOURCE` signature implementation is at `langfc-src/common/source.sig` and the `Source` structure implementation is at `langfc-src/common/source.sml`. Source positions and spans of terminals are provided by the scanner. Consult the ML-Yacc and ML-Antlr manuals for information about how to access position information in semantic actions.

4 GForge and Submission

Sources for Project 2 have been (or will shortly be) committed to your repository in the `project2` sub-directory. You will need to *update* your local copy, by running the command:

```
svn update
```

from the *`cnetid`*`-proj` directory.

We will collect projects from the SVN repositories at 10pm on Friday, February 6; make sure that you have committed your final version before then. To do so, run the command:

```
svn commit
```

from the *`cnetid`*`-proj` directory.

5 Hints

- There is no "better choice" between ML-Yacc and ML-Antlr. Both tools and underlying parsing algorithms have features that will make some portions of the LangF grammar more natural to specify and will make other portions more difficult to specify. The reference solutions are of nearly identical length and complexity.
- To complete the assignment, you should only need to make changes to the *`cnetid`*`-proj/project2/langfc-src/parser/wrapped-parser.sml` file and *either* the *`cnetid`*`-proj/project2/langfc-src/parser/langf-antlr.grm` file *or* the *`cnetid`*`-proj/project2/langfc-src/parser/langf-yacc.grm` file.
- Executing the compiler (from the *`cnetid`*`-proj/project2` directory) with the command

  ```
  ./bin/langfc -Ckeep-parse=true file.lgf
  ```

  will produce a *`file`*`.parse.pt` file that contains the abstract parse tree returned by the parser. Use this control and its output to check that your parser is working as expected. The `tests/parser` directory includes a number of tests (of increasing complexity); for each `test`*`NN`*`.lgf` file, there is either a `test`*`NN`*`.out` file containing the parse tree to be returned by the parser or, if the test has syntax errors, a `test`*`NN`*`.err` file containing sample error messages.
- Because ML-Yacc and ML-Antlr provide automatic error repair, their error messages (and resulting parses) are dependent upon the grammar specification. Hence, you are likely to produce error messages slightly different from those found in the `text`*`NN`*`.err` files (and parses slightly different from those found in the `text`*`NN`*`.out`).

6 Extra: Integrating a Hand-Written Scanner

If you would like to adapt your hand-written scanner from Project 1, then you will need extend your implementation to include position and span information for tokens. You should copy your implementation from `cnetid-proj/project1/langfc-src/scanner/langf-scanner.sml` into `cnetid-proj/project2/langfc-src/scanner/langf-hand-scanner.sml`, which implements the `LangFHandScanner : LANGF_HAND_SCANNER` module. The `LANGF_HAND_SCANNER` signature is as follows:

```
signature LANGF_HAND_SCANNER =
sig
   val scan : {getPos: 'strm -> 'pos,
               forwardPos: 'pos * int -> 'pos,
               reportErrorAt: 'pos * string -> unit} ->
              (char, 'strm) StringCvt.reader ->
              (Tokens.token * ('pos * 'pos), 'strm) StringCvt.reader
end
```

Note that `scan` is now a function that takes a character reader and returns a `Tokens.token * ('pos * 'pos)` reader. To support position information and error reporting, the `LangFHandScanner.scan` function takes an initial argument with

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

Figure 3 sketches how a hand-written scanner should use `getPos` to get the left position of a token and `forwardPos` to compute the right position of a token. The `tests/scanner` directory includes the tests from Project 1, updated with position information in the output and error files.

The project seed code includes a compiler control (`-Cscanner=ulex` / `-Cscanner=hand`) that selects between the ML-ULex scanner and the hand-written scanner. Use this control to select the hand-written scanner for an invocation of the compiler; alternatively, you can change the default setting. This default is specified by the `scannerCtl` value in the `langfc-src/scanner/wrapped-scanner.sml` file (lines 36 – 42).

The `testNN.out` and `testNN.err` files in the `tests/scanner` directory have been updated with position and span information for tokens and error messages.

Document History

February 4, 2009
Figure 2: Removed *AtomicExp* → *daconid* production and revised *ApplyArg* → *AtomicExp* and *Exp* → **~** *AtomicExp* productions to *ApplyArg* → (*AtomicExp* | *daconid*) and *Exp* → **~** (*AtomicExp* | *daconid*), respectively. This resolves an unintentional ambiguity in the grammar.

January 29, 2009 Changed due date to February 9, 2009.

January 27, 2009
Section 2: are given in order of decreasing precedence ⇒ are given in order of increasing precedence

January 22, 2009 Original version

```
structure T = Tokens
fun scan {getPos: 'strm -> 'pos,
          forwardPos: 'pos * int -> 'pos,
          reportErrorAt: 'pos * string -> unit}
         (charRdr: (char, 'strm) StringCvt.reader) :
         (Tokens.token * ('pos * 'pos), 'strm) StringCvt.reader =
  let
     ...
     fun scan strm0 =
        let
           val pos0 = getPos strm0
        in
           case charRdr strm0 of
              NONE => NONE
            | SOME (#"+", strm1) =>
                 SOME ((T.PLUS, (pos0, forwardPos (pos0, 1))), strm1)
            | ...
        end
  in
     scan
  end
```

Figure 3: Skeleton hand-written scanner with position information