Compiler Construction

CSCI-742 **Project 2**
Term 20145 **February 13, 2015**

LangF Parser

Due: March 6, 2015

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 and ML-Antlr generates parsers that use an LL(k) parsing algorithm. There are links to the manuals for both tools in the Resources page of the course web site (`http://www.cs.rit.edu/~mtf/teaching/20135/cc/resources.html`); 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 Extended-BNF grammar given in Figures 1, 2, and 3.

If one ignores the parenthetical annotations, then the grammar is ambiguous in the *Type*, *Exp*, and *ComplexExp* nonterminals. In order to make the grammar unambiguous, the parenthetical annotations specify the precedence and associativity of the *Type*, *Exp*, and *ComplexExp* productions. The *Type*, *Exp*, and *ComplexExp* 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 *ComplexExp* as an example. Consider two productions, such that the first ends with a *ComplexExp* and the second starts with a *ComplexExp*. For example, consider:

$$\mathit{ComplexExp} \rightarrow \mathit{ComplexExp}\ \texttt{andalso}\ \mathit{ComplexExp} \qquad \text{and} \qquad \mathit{ComplexExp} \rightarrow \mathit{ComplexExp}\ \texttt{<>}\ \mathit{ComplexExp}$$

Suppose that we must parse the sequence:

$$\cdots\ \cdots\ \texttt{andalso}\ \mathit{ComplexExp}\ \texttt{<>}\ \cdots\ \cdots$$

where *ComplexExp* stands for a token sequence that has already been determined to be a *ComplexExp* (if necessary, by applying precedence and associativity resolution). The higher precedence of the *ComplexExp* **<>** *ComplexExp* production dictates that *ComplexExp* associate to the right; that is, the sequence should be parsed as:

$$\cdots\ \cdots\ \texttt{andalso}\ (\ \mathit{ComplexExp}\ \texttt{<>}\ \cdots\ )\ \cdots \qquad \text{correct}$$

and not as:

$$\cdots\ (\ \cdots\ \texttt{andalso}\ \mathit{ComplexExp}\ )\ \texttt{<>}\ \cdots\ \cdots \qquad \text{incorrect}$$

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:

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

where $\mathit{ComplexExp}_1$, $\mathit{ComplexExp}_2$, and $\mathit{ComplexExp}_3$ stand for token sequences that has already been determined to be *ComplexExp*s (if necessary, by applying precedence and associativity resolution). The right associativity of the **^** operator dictates that the sequence should be parsed as:

$$\cdots\ \cdots\ \mathit{ComplexExp}_1\ \texttt{\^{}}\ (\ \mathit{ComplexExp}_2\ \texttt{\^{}}\ \mathit{ComplexExp}_3\ )\ \cdots\ \cdots \qquad \text{correct}$$

and not as:

$$\cdots\ \cdots\ (\ \mathit{ComplexExp}_1\ \texttt{\^{}}\ \mathit{ComplexExp}_2\ )\ \texttt{\^{}}\ \mathit{ComplexExp}_3\ \cdots\ \cdots \qquad \text{incorrect}$$

The latter parse requires explicit parentheses.

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

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

TypeParams
  ::= (empty)
  | `[` (*tyvarid* (`,` *tyvarid*)*)$^?$ `]`

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

TypeArgs
  ::= (empty)
  | `[` (*Type* (`,` *Type*)*)$^?$ `]`

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

DaConDecl
  ::= *daconid* *DaConArgTys*

DaConArgTys
  ::= (empty)
  | `{` (*Type* (`,` *Type*)*)$^?$ `}`

SimplePat
  ::= *varid*
  | `_`

FunDecl
  ::= *varid* *Param*$^+$ `:` *Type* `=` *Exp*

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

Figure 1: The concrete syntax of LangF (A)

Exp			
$::=$	$\mathtt{fn}\ \mathit{Param}^{+}\ \mathtt{=>}\ \mathit{Exp}$		(lowest precedence)
$\mid$	$\mathtt{if}\ \mathit{Exp}\ \mathtt{then}\ \mathit{Exp}\ \mathtt{else}\ \mathit{Exp}$		
$\mid$	$\mathit{Exp}\ \mathtt{:}\ \mathit{Type}$		
$\mid$	$\mathit{ComplexExp}$		(highest precedence)
ComplexExp			
$::=$	$\mathit{ComplexExp}\ \mathtt{orelse}\ \mathit{ComplexExp}$		(L) (lowest precedence)
$\mid$	$\mathit{ComplexExp}\ \mathtt{andalso}\ \mathit{ComplexExp}$		(L)
$\mid$	$\mathit{ComplexExp}\ \mathtt{!}\ \mathit{ComplexExp}\ \mathtt{:=}\ \mathit{ComplexExp}$		(R)
$\mid$	$\mathit{ComplexExp}\ \mathtt{!}\ \mathit{ComplexExp}$		(L)
$\mid$	$\mathit{ComplexExp}\ \mathit{op}\ \mathit{ComplexExp}$	$\mathit{op} \in \{\mathtt{==}, \mathtt{<>}, \mathtt{<}, \mathtt{<=}, \mathtt{>}, \mathtt{>=}\}$	(L)
$\mid$	$\mathit{ComplexExp}\ \mathtt{\hat{}}\ \mathit{ComplexExp}$		(R)
$\mid$	$\mathit{ComplexExp}\ \mathit{op}\ \mathit{ComplexExp}$	$\mathit{op} \in \{\mathtt{+}, \mathtt{-}\}$	(L)
$\mid$	$\mathit{ComplexExp}\ \mathit{op}\ \mathit{ComplexExp}$	$\mathit{op} \in \{\mathtt{*}, \mathtt{/}, \mathtt{\%}\}$	(L)
$\mid$	$\mathit{SimpleExp}$		(highest precedence)
SimpleExp			
$::=$	$\mathit{op}\ (\mathit{AtomicExp} \mid \mathit{daconid})$	$\mathit{op} \in \{\mathtt{\sim}, \mathtt{\#}\}$	
$\mid$	$\mathit{daconid}\ \mathit{TypeArgs}\ \mathit{DaConArgs}$		
$\mid$	$\mathit{ApplyExp}$		
DaConArgs			
$::=$			(empty)
$\mid$	$\mathtt{\{}\ (\mathit{Exp}\ (\mathtt{,}\ \mathit{Exp})^{*})^{?}\ \mathtt{\}}$		
ApplyExp			
$::=$	$\mathit{ApplyExp}\ \mathit{ApplyArg}$		
$\mid$	$\mathit{AtomicExp}$		
ApplyArg			
$::=$	$(\mathit{AtomicExp} \mid \mathit{daconid})$		
$\mid$	$\mathtt{[}\ \mathit{Type}\ \mathtt{]}$		
AtomicExp			
$::=$	varid		
$\mid$	$\mathit{integer}$		
$\mid$	string		
$\mid$	$\mathtt{(}\ \mathit{Exp}\ (\mathtt{;}\ \mathit{Exp})^{+}\ \mathtt{)}$		
$\mid$	$\mathtt{let}\ \mathit{Decl}^{+}\ \mathtt{in}\ \mathit{Exp}\ (\mathtt{;}\ \mathit{Exp})^{*}\ \mathtt{end}$		
$\mid$	$\mathtt{case}\ \mathit{Exp}\ \mathtt{of}\ \mathit{MatchRule}\ (\mathtt{\mid}\ \mathit{MatchRule})^{*}\ \mathtt{end}$		
$\mid$	$\mathtt{(}\ \mathit{Exp}\ \mathtt{)}$		

Figure 2: The concrete syntax of LangF (B)

MatchRule
  ::= *Pat* **=>** *Exp*

Pat
  ::= *daconid TypeArgs DaConPats*
  | *SimplePat*

DaConPats
  ::= (empty)
  | **{** (*SimplePat* (**,** *SimplePat*)*)$^?$ **}**

Figure 3: The concrete syntax of LangF (C)

Here are some examples:

```
           b1 orelse b2 : Bool : Bool    ≡    ((b1 orelse b2) : Bool) : Bool
         b1 andalso i == j orelse b2    ≡    (b1 andalso (i == j)) orelse b2
                       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
                           a ! 0 ! 0    ≡    (a ! 0) ! 0
                  a ! 0 ! 0 := 1 + 2    ≡    (a ! 0) ! 0 := (1 + 2)
  a ! i ! j := b ! j ! i := c ! k    ≡    (a ! i) ! j := ((b ! j) ! i := (c ! k)
```

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 43 – 49).

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 FusionForge 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 *ritid*`cc` directory.

We will collect projects from the SVN repositories at 11:59pm on Fri, March 6, 2015; make sure that you have committed your final version before that time. To do so, run the command:

```
svn commit
```

from the *ritid*`cc` directory.

5 Hints

- Start early!
- 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 `ritidcc/project2/langfc-src/parser/wrapped-parser.sml` file and *either* the `ritidcc/project2/langfc-src/parser/langf-antlr.grm` file *or* the `ritidcc/project2/langfc-src/parser/langf-yacc.grm` file.
- Executing the compiler (from the `ritidcc/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 `testNNN.lgf` file, there is either a `testNNN.parser.soln.pt` file containing the parse tree to be returned by the parser or, if the test has syntax errors, `testNNN.parser.soln.yacc.err` and `testNNN.parser.soln.antlr.err` files 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 `textNNN.yacc.err` and `textNNN.antlr.err` files.
- A reference solution is available on the CS Department file system at:

  ```
  /usr/local/pub/mtf/cc/project/project2-soln/langfc
  ```

  Use the reference solution to confirm your understanding of the project and to develop additional tests.
- Ask questions and use the reference solution! It is rather difficult to give a prose description of the disambiguation rules for an ambiguous grammar. In particular, the interaction of array update and array index is difficult to describe in text.
- The most difficult non-terminals to handle are *ComplexExp*, *Type*, *Exp*, and *SimpleExp*. The most difficult productions to handle are *ComplexExp* → *ComplexExp* `!` *ComplexExp* `:=` *ComplexExp* and *ComplexExp* → *ComplexExp* `!` *ComplexExp*.
- Although the semantic actions of productions can be arbitrary Standard ML code, in practice, they are simple applications of constructors from **`structure`** `ParseTree`.

6 Extra Credit[1]: 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 *ritid*`cc/project1/langfc-src/scanner/langf-hand-scanner.sml` to *ritid*`cc/project2/langfc-src/scanner/langf-hand-scanner.sml`. The revised `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 4 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 `test`*NNN*`.out` and `test`*NNN*`.err` files from Project 1 in the `tests/scanner` directory have been updated with position and span information for tokens and error messages.

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 43 – 49).

[1] +4% to Programming Projects category

```
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 scanTok strm0 =
        let
           val pos0 = getPos strm0
           fun posN n = forwardPos (pos0, n)
        in
           case charRdr strm0 of
              NONE => NONE
            | SOME (#"+", strm1) => SOME ((T.PLUS, (pos0, posN 1)), strm1)
            | SOME (#"-", strm1) =>
                 (case charRdr strm1 of
                     SOME (#">", strm2) => SOME ((T.MINUS_ARROW, (pos0, posN 2)), strm2)
                   | _ => SOME ((T.MINUS, (pos0, posN 1)), strm1))
            | ...
        end
  in
     scanTok
  end
```

Figure 4: Skeleton hand-written scanner with position information

Document History

February 13, 2015

Original version