School of Computer Science, University of Nottingham

G52MAL Machines and their Languages, Spring 2012

Thorsten Altenkirch

Exercises, Set 6

Tuesday 29th May 2012

Deadline: Tuesday, 5 June 2012, by email to txa@cs.nott.ac.uk

Consider the following grammar giving the *context-free syntax* for a Trivial Expression Language (TXL):

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathbf{let}\ \underline{\mathit{Identifier}}\ \mathbf{=}\ \mathit{Exp}\ \mathbf{in}\ \mathit{Exp} \\
 & | & \mathit{ArithExp} \\
\mathit{ArithExp} & \rightarrow & \mathit{ArithExp}\ \mathbf{+}\ \mathit{ArithExp} \\
 & | & \mathit{ArithExp}\ \mathbf{-}\ \mathit{ArithExp} \\
 & | & \mathit{ArithExp}\ \mathbf{*}\ \mathit{ArithExp} \\
 & | & \mathit{ArithExp}\ \mathbf{/}\ \mathit{ArithExp} \\
 & | & \mathit{PrimExp} \\
\mathit{PrimExp} & \rightarrow & \underline{\mathit{IntegerLiteral}} \\
 & | & \underline{\mathit{Identifier}} \\
 & | & \mathbf{(}\mathit{Exp}\mathbf{)}
\end{array}
$$

Exp is the start symbol. Non-terminals are typeset in italic font, like *this*. Terminals are typeset in bold, like **this**. Terminals whose lexeme (the concrete character sequence) is different from what is shown in the grammar are typeset in italics and underlined, such as $\underline{\mathit{Identifier}}$ and $\underline{\mathit{IntegerLiteral}}$; their precise spelling is handled during lexical analysis (see Exercises, Set 3).

Consider also the following grammar giving the *abstract syntax* of TXL:

$$
\begin{array}{lcl}
\mathit{AST} & \rightarrow & \mathbf{LitInt}\ \underline{\mathit{IntegerLiteral}} \\
 & | & \mathbf{Var}\ \underline{\mathit{Identifier}} \\
 & | & \mathbf{BinOpApp}\ \mathit{AST}\ \mathit{BinOp}\ \mathit{AST} \\
 & | & \mathbf{Let}\ \underline{\mathit{Identifier}}\ \mathit{AST}\ \mathit{AST} \\
\mathit{BinOp} & \rightarrow & \mathbf{+} \mid \mathbf{-} \mid \mathbf{*} \mid \mathbf{/}
\end{array}
$$

This grammar captures the tree structure of TXL programs as concisely as possible. Once a program has been successfully parsed, its structure has been determined — thus there is no need for parentheses or any of the other non-terminals that are present in the TXL language purely to define a program's structure. Finally, note that each production has been assigned a single terminal denoting the label the corresponding node would have were it to be drawn as a tree.

1. **(30 marks)**

   (a) Impart precedence on the TXL CFG by giving ***** and **/** higher precedence than **+** and **-**.

   (b) Disambiguate both the *original* and *modified* CFGs by imparting left associativity on the binary operators.

   (c) Recall the transformation used to eliminate left-recursion from a grammar. Apply the transformation to the four CFGs (the original and your answers to (a) and (b)).

   Note: *When answering these (and later) questions, only give the productions for non-terminals that you have added or modified. There is no need to repeat productions for unmodified non-terminals.*

2. **(15 marks)**

   Study the parse functions in the provided source code for the TXL parser.

   (a) Which of your four non-left-recursive grammars (your answers to Question 1c) does the parser correspond to?

   (b) What associativity and precedence rules will be reflected by an Abstract Syntax Tree constructed by this parser? (Consider **`let`** expressions as well as the binary operators.)

   (c) Explain how the parse functions correspond to the productions of the grammar.

3. **(35 marks)**

   (a) Extend the *original* TXL CFG with productions for the following language extensions:

      i. **`if-then-else`** expressions;

      ii. character literals;

      iii. the following seven binary operators:

         `<=`, **`<`**, **`==`**, **`>`**, **`>=`**, **`<>`**, **`^`**

      Note: *Do not concern yourself that some expressions can now have a Boolean type (whereas before they could only be integers). This is dealt with during the Semantic Analysis phase of compilation, which follows after parsing.*

   (b) Disambiguate your extended grammar by imparting associativity and precedence according to the following table:

Operator	Precedence	Associativity
`^`	highest	right
`*` `/`	high	left
`+` `-`	medium	left
`<` `<=` `==` `>=` `>` `<>`	low	non
`if-then-else` `let-in`	lowest	all

   Note that **`if-then-else`** and **`let-in`** expressions should be allowed to associate in any direction, but this is unambiguous because the keywords make the structure explicit.

   (c) Transform your disambiguated grammar into a form that makes it suitable for recursive-descent parsing by eliminating left recursion.

   (d) Extend the TXL *abstract syntax* to allow for the new language constructs.

4. **Bonus Exercise (20 marks)**

   Using your extended TXL grammars as a guide (your answers to questions 3c and 3d), extend the provided TXL parser to handle the new language constructs.

   You will need to:

   (a) Extend the representation of Abstract Syntax Trees (the *AST* data type).

   (b) Extend the pretty printer accordingly.

   (c) Modify the parser, adding new parsing functions where necessary, making sure to respect operator precedence and associativity as defined in Question 3b.

   Test your solution thoroughly! For example,

```
main "if 7 > 3 then 'x' else 1 + 2 ^ 3 ^ 4 - 5"
```

   should print an Abstract Syntax Tree similar to the following:

```
If
  BinOpApp
    7
    More
    3
  'x'
  BinOpApp
    BinOpApp
      1
      Plus
      BinOpApp
        2
        Power
        BinOpApp
          3
          Power
          4
    Minus
    5
```