CSC488H1S - Assignment 2: Design Document.

Group 2
g0dalaln - Natasha Dalal
g0getter - Tony (Hao) Cheng
g0faizan - Faizan Rashid
g0alimuh - Showzeb Ali
g3ksingh - Karandeep Basi

February 6, 2014

When designing the grammar, I started by simply following the language specification, correcting obvious problems as I went. The first real variation from the given language specification was splitting up grammar rules that could lead to multiple parse trees. For instance, $statement \rightarrow statement\ statement$. In order to disambiguate cases like these, I split up rules with two recursive references into another rule. Therefore with the example of $statement$, I created $statementList \rightarrow statement \mid statement\ statementList$. Then, wherever there were references to $statement$ in the reference language rules, I replaced them with $statementList$. A similar procedure was followed for $declaration$ (leading to $declarionList$), $variablenames$ (leading to $variableNamesList$), $output$ and $input$ (leading to $outputList$ and $inputList$), $arguments$ (leading to $argumentsList$), and $paramaters$ (leading to $parametersList$).

Another deviation from the reference grammar involved replacing all references to variablenames, parameternames, functionnames, procedurenames, etc... with the single $IDENT$ in order to prevent reduce-reduce conflicts.

The part of the grammar I found most challenging was the expressions. Specifically since expressions needed to be defined in a way to allow for operator precedence and associatively. The way that I handled this was by starting at a base form that I called $baseExpression$. This base form defined some of the most basic forms of expressions (those not including operators): Terminals, function calls, expressions in parantheses etc.... From there, I built up at every step starting with the operator of highest precedence, including in its rules a rule to generate the previous type and then a rule using the previous type with the current operator. For example: the unary minus has highest precedence and so I did $UnaryExpression \rightarrow BaseExpression \mid MINUS\ BaseExpression$. In this way, I built up all the way to the or operator which had the lowest precedence. $OrExpression$ was therefore inclusive of all other generatable expressions, as well as expressions separated by an or. Since and was the operator with precedence directly above or, $orExpression$ was defined as $andExpression \mid orExpression\ OR\ andExpression$. The use of $orExpression$ as the fist non-terminal in the second rule for $orExpression$ was to allow for associativity and will be explained below. I then used $OrExpressions$ as my $finalExpression$ which is used wherever the reference language makes reference to expression. This $finalExpression$ was also used in $baseExpressions$ within parentheses.

An important note is that in order to maintain associativity from Left to Right for multiplication, division, addition, subtraction, and, or (and also not and unary minus – though the direction of associativity was not relevant there) My rules for those have a recursive part as the left most non-terminal on the RHS of the rule instead of the operator expression above it in the precedence hierarchy. This ensures the left to right associativity. For example: $andExpression \rightarrow notExpression \mid andExpression\ AND\ notExpression$ instead of $notExpression\ AND\ notExpression$. This would allow for something like a AND b AND c which would be parsed as something like ((a AND b) AND c). Such a recursive rule was not used for the equality type of operators $(=, <, >, <=, >=, not =)$ as those rules were not meant to be associative.