1 Complete Parse Expressions: A Notation for Parse Trees

A parse tree can be unambiguously denoted using a symbolic form that I refer to as a *complete parse-expression*. Let G denote an arbitrary context-free grammar G. Let A denote a nonterminal symbol in G and let α denote a string of symbols (terminals and/or nonterminals) such that:

$$A \Rightarrow^{+} \alpha$$

Let α' denote a string that is derived from α by subscripting all nonterminals appearing in α. Under these assumptions we say that the term $A[\![\alpha']\!]$ is a complete parse expression with respect to the grammar G.

1.1 An Example

Consider the following grammar containing the terminal symbols $\{c, d\}$ and the nonterminal symbols $\{A\_list, B, D\}$:

$$\boxed{\begin{array}{lcl} A\_list & ::= & A\_list\ B \mid c \\ B & ::= & B\ D \mid D \\ D & ::= & d \end{array}}$$

Among others, this grammar contains the following complete parse expressions:

$A\_list[\![c]\!]$
$A\_list[\![A\_list_1\ B_1]\!]$
$A\_list[\![A\_list_1\ B_1\ B_1]\!]$
$A\_list[\![A\_list_1\ B_1\ B_2]\!]$
$A\_list[\![cB_1\ B_2]\!]$
$A\_list[\![cB_1\ d]\!]$

$B[\![D_1]\!]$
$B[\![B_1\ D_1]\!]$
$B[\![B_1\ D_1\ D_2]\!]$
$B[\![B_1\ D_1\ D_1]\!]$
$B[\![d]\!]$

The following expressions are **NOT** complete parse expressions with respect to G:

Expression	Comment
$A\_list[\![A\_list_1]\!]$	The length of the derivation must be at least 1.
$A\_list[\![A\_list_1\ B]\!]$	The nonterminal B must be given a subscript.
$A\_list[\![D_1]\!]$	The derivation $A \stackrel{+}{\Rightarrow} D$ is not possible in G.

1.2 An Example

In class conditions were given when the leading nonterminal of a complete parse expression can be dropped. Consider the following grammar containing the terminal symbols $\{c, d\}$ and the nonterminal symbols $\{A\_list, B, D\}$:

$$\boxed{\begin{array}{lcl} A\_list & ::= & A\_list\ B \mid B \\ B & ::= & c \mid D \\ D & ::= & d \end{array}}$$

Among others, this grammar contains the following parse expressions:

Parse-expression	Derived from
$[\![B_1]\!]$	A
$[\![A\_list_1\ B_1]\!]$	A
$[\![A\_list_1\ B_1\ B_1]\!]$	A
$[\![A\_list_1\ B_1\ B_2]\!]$	A
$[\![c\ B_1\ B_2]\!]$	A
$[\![c\ B_1\ d]\!]$	A
$[\![D_1]\!]$	B
$[\![c]\!]$	B
$[\![d]\!]$	D

2 Matching

Parse expressions are useful because we will use them in equations to define the semantics of programming language constructs. In this framework, a program can be executed by using these equations to simplify a program until it can be simplified no further. This simplification is based on equational reasoning and requires matching as one of its computational steps. Therefore, to understand how a program can be simplified we must first understand matching in this context.

In the context of parse-expressions, a *match expression* is of the form:

$$[\![\alpha']\!] \ll [\![\beta]\!]$$

In this case, α denotes a string that may contain terminals and subscripted nonterminals and β denotes a string that may only contain terminal symbols. It is also assumed that $[\![\alpha']\!]$ and $[\![\beta]\!]$ are well-formed parse expressions.

In the context of matching, subscripted nonterminals occurring in parse expressions are treated as **variables** quantified over the set of all strings they can derive with respect to the given grammar. A match is said to **succeed** if the variables on the left-hand side of the match expression (i.e., the variables to the left of $\ll$) can be instantiated in such a manner so that the left and right sides of the match expression become syntactically equal.

The algorithm for solving a match expression of the form $[\![\alpha']\!] \ll [\![\beta]\!]$ can be summarize as follows:

1. Check that $[\![\alpha']\!]$ and $[\![\beta]\!]$ are well-formed.
2. Find values for variables (if possible) so that when the variables in α' are replaced with their values, the resulting parse-expression is $[\![\beta]\!]$.

3. Make sure that a value assigned to a variable can actually be derived from that variable. That is if A_1 is a variable and v_1 is a value, then $A \stackrel{+}{\Rightarrow} v_1$ must be possible within the grammar.

2.1 Example

In this example, we use a standard BNF grammar fragment describing a subset of mathematical expressions. We assume that *num* and *ident* are terminal symbols and have the usual meaning.

E	::=	E + T | T
T	::=	T * F | F
F	::=	num | ident | (E)

Match Expression	Result of Match
$[\![E_1 + T_1]\!] \ll [\![1+2]\!]$	true – $E_1 \stackrel{*}{\Rightarrow} 1$ and $T_1 \stackrel{*}{\Rightarrow} 2$
$[\![E_1 + T_1]\!] \ll [\![1+1]\!]$	true – $E_1 \stackrel{*}{\Rightarrow} 1$ and $T_1 \stackrel{*}{\Rightarrow} 1$
$[\![T_1 * F_1]\!] \ll [\![1 * x]\!]$	true – $T_1 \stackrel{*}{\Rightarrow} 1$ and $F_1 \stackrel{*}{\Rightarrow} x$
$[\![T_1 * F_1]\!] \ll [\![x * x]\!]$	true – $T_1 \stackrel{*}{\Rightarrow} x$ and $F_1 \stackrel{*}{\Rightarrow} x$
$[\![E_1 + E_2]\!] \ll [\![x+2]\!]$	false – $[\![E_1 + E_2]\!]$ is not well-formed
$[\![T_1 + T_2]\!] \ll [\![1+y]\!]$	true – $T_1 \stackrel{*}{\Rightarrow} 1$ and $T_2 \stackrel{*}{\Rightarrow} y$
$[\![T_1 + T_1]\!] \ll [\![1+2]\!]$	false – T_1 cannot be instantiated to be both 1 and 2 at the same time.
$[\![T_1 * T_2]\!] \ll [\![x * 2]\!]$	false – $[\![T_1 * T_2]\!]$ is not well-formed
$[\![F_1 * F_2]\!] \ll [\![1 * y]\!]$	true – $F_1 \stackrel{*}{\Rightarrow} 1$ and $F_2 \stackrel{*}{\Rightarrow} y$
$[\![F_1 * F_1]\!] \ll [\![1 * 2]\!]$	false – F_1 cannot be instantiated to be both 1 and 2 at the same time.
$[\![E_1 + T_1 * F_1]\!] \ll [\![1 + 2 * 3]\!]$	true – $E_1 \stackrel{*}{\Rightarrow} 1$ and $T_1 \stackrel{*}{\Rightarrow} 2$ and $F_1 \stackrel{*}{\Rightarrow} 3$
$[\![T_1 + T_1 * F_1]\!] \ll [\![2 + 2 * 3]\!]$	true – $T_1 \stackrel{*}{\Rightarrow} 2$ and $F_1 \stackrel{*}{\Rightarrow} 3$
$[\![T_1 + T_1 * F_1]\!] \ll [\![1 + 2 * 3]\!]$	false – T_1 cannot be instantiated to be both 1 and 2 at the same time.
$[\![E_1 + F_1 * F_2]\!] \ll [\![1 + 2 * 3]\!]$	true – $E_1 \stackrel{*}{\Rightarrow} 1$ and $F_1 \stackrel{*}{\Rightarrow} 2$ and $F_2 \stackrel{*}{\Rightarrow} 3$
$[\![F_1 + F_2 * F_3]\!] \ll [\![1 + 2 * 3]\!]$	true – $F_1 \stackrel{*}{\Rightarrow} 1$ and $F_2 \stackrel{*}{\Rightarrow} 2$ and $F_3 \stackrel{*}{\Rightarrow} 3$
$[\![F_1 + F_2 * T_3]\!] \ll [\![1 + 2 * 3]\!]$	false – $[\![F_1 + F_2 * T_3]\!]$ is not well-formed.