CSCI 4220
– Assignment 1 –

1 Overview

The goals of this assignment are as follows:

- To develop a framework that will allow us to formally specify how to *construct* a variety of *basic expressions*, in what I will refer to as *parenthesized normal form*, that can occur in programming languages.
- To gain familiarity with the syntax of SML expressions.
- To get experience using Bascinet.

2 Ten Normal Forms

Goal: To get you to play around with rewriting SML expressions into their parenthesized normal forms.

Transformation to be applied: *Multifile_check_pnf_with_metrics.tlp*

Recall that in class we discussed how to rewrite expressions into their parenthesized normal form. For this assignment you need to create ten distinct expressions and their corresponding parenthesized normal forms. When doing this, you should use the following syntax:

Abstract Description of Syntax

any-old-expression -> *normal-form-of-expression* ;

Example

1 + 2 + 3 -> ((1 + 2) + 3) ;

More specifically, you are to write a "program" consisting of ten examples of expressions rewritten to their (correct) normal form. Each rewrite must be terminated by a semi-colon. In addition:

- Your program should be stored as a file having a dot-tgt extension.
- Your program should contain your name as a comment. In this context (as in SML), a comment begins with a (* and ends with a *).

3 The Challenge

```
file[:]input_that_passes.tgt [:]
correct;
correct;
correct;
correct;
correct;
correct;
correct;
correct;
correct;
correct;

========================================
Operators Used:
========================================
        if-then-else = 8
        orelse = 8
        andalso = 8
        not = 4
        < = 8
        <= = 6
        > = 6
        >= = 6
        = = 6
        <> = 8
        + = 10
        - = 10
        * = 10
        div = 8
        mod = 8

========================================
Metrics:
========================================
        sum of constructs = 114
        mean = 7
        variance = 2
        standard deviation = 1
        correct = 10
        incorrect = 0

PASSED!
```

Figure 1: Example output for a program that passes.

Grading for this assignment is *all-or-nothing*. In order for you to receive a grade, you program must have certain properties. Conceptually, you will need to write reasonably complex expressions and make use of all of the operators and constructs we have gone over in class. Technically, your program should satisfy the properties shown in the table below.

total number of constructs	$>$	100
mean	$>$	4
variance	$<$	3
standard deviation	$<$	3
correct	$=$	10
incorrect	$=$	0

If your program has these properties, then you will see an output similar to that shown in Figure 1.

A Primitive Values and Operations

Basic expressions are constructed by composing primitive values and operations. There are rules specifying which compositions are syntactically legal. It is assumed that you know the rules for how to construct mathematical expressions.

A.1 Primitive Values

Equality Types: $''\tau$		
int	$=$	$\{min, \ldots, \sim 2, \sim 1, 0, 1, 2, \ldots, max\}$
bool	$=$	$\{false, true\}$

A.2 Logical Operations

Logical		
orelse	:	$bool * bool \to bool$
andalso	:	$bool * bool \to bool$
not	:	$bool \to bool$

Remark: *andalso* and *orelse* are simply a short-hand (i.e., a derived form) for *if-then-else* expressions.

A.3 Relational Operations

Relational		
$<$	:	$int * int \to bool$
$<=$	:	$int * int \to bool$
$>$	:	$int * int \to bool$
$>=$	:	$int * int \to bool$

A.4 Equality Operations

We will use the symbol $''\tau$ to denote types for which equality operations are defined. Equality operations are defined for integers and booleans (but are not defined for reals as well as some other types which we will discuss later in the semester).

Equality		
$=$	:	$''\tau * ''\tau \to bool$
$<>$	:	$''\tau * ''\tau \to bool$

A.5 Arithmetic Operations

Arithmetic		
$+$	:	$int * int \to int$
$-$	:	$int * int \to int$
$*$	:	$int * int \to int$
div	:	$int * int \to int$
mod	:	$int * int \to int$

A.6 Precedence and Associativity

The precedence table shown below provides the basis for inserting/omitting parenthesis from basic expressions.

Operator	**Precedence**	**Associativity**
not	highest	left
*, div, mod	7	left
+, -	6	left
=, <>, <, >, <=, >=	4	left
andalso	*-1*	*left*
orelse	*-2*	*left*

Recall that *andalso* and *orelse* are simply a short-hand (i.e., a derived form) for *if-then-else* expressions. What this means is that, technically speaking, they do not show up in the precedence-associativity table. However, it is convenient to add them.

B Entities Being Targeted by Rules

A fundamental question when writing rules is to define the entities to which rules should be applied. There are two natural choices for representing expressions.

1. An expression is a *term* – this representation assumes the parsing problem is solved and that you understand the structure of terms.

2. An expression is a *string* – this representation assumes the parsing problem is not solved and that you do not understand the structure of terms.

Conceptually speaking, the parsing problem is what lets us correctly decompose an expression into its parts. In other words, the parsing problem lets us incrementally unravel the compositional steps that were used to create the expression.

Unraveling an expression that is represented as a term (a two-dimensional object) is easy and parenthesis are not needed. However, when viewing an expression as a string (a one-dimensional object) we need the help of parenthesis to help us with the incremental unraveling process. For this reason, our first mission is to define how to construct expressions that are appropriately parenthesized.

C Parenthesized Normal Form Construction

We want to have rules for constructing fully parenthesized expressions (but not overly parenthesized expressions)– it is only notational conventions together with precedence and associativity rules that let us omit parenthesis. We will use the phrase *parenthesized normal form* to denote expressions that are properly parenthesized, and we will use the symbol E to denote this set of expressions.

$$\frac{v \in Int}{v \in E}(\textit{E-Int}) \qquad \frac{v \in Bool}{v \in E}(\textit{E-Bool})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 + e_2) \in E}(\textit{E-Plus}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 - e_2) \in E}(\textit{E-Minus})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 * e_2) \in E}(\textit{E-Mult}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 \text{ div } e_2) \in E}(\textit{E-Div}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 \text{ mod } e_2) \in E}(\textit{E-Mod})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 < e_2) \in E}(\textit{E-Lt}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 <= e_2) \in E}(\textit{E-LtEq})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 > e_2) \in E}(\textit{E-Gt}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 >= e_2) \in E}(\textit{E-GtEq})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 = e_2) \in E}(\textit{E-Eq}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 <> e_2) \in E}(\textit{E-Neq})$$

$$\frac{e_1 \in E \qquad e_2 \in E}{(e_1 \text{ andalso } e_2) \in E}(\textit{E-Andalso}) \qquad \frac{e_1 \in E \qquad e_2 \in E}{(e_1 \text{ orelse } e_2) \in E}(\textit{E-OrElse})$$

$$\frac{e_1 \in E \qquad e_2 \in E \qquad e_3 \in E}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \in E}(\textit{E-If}) \qquad \frac{e \in E}{(not\ e) \in E}(\textit{E-Not})$$

C.1 Example I

$$\cfrac{\cfrac{\cfrac{1 \in Int}{1 \in E}(\textit{E-Int}) \qquad \cfrac{2 \in Int}{2 \in E}(\textit{E-Int})}{(1+2) \in E}(\textit{E-Plus}) \qquad \cfrac{3 \in Int}{3 \in E}(\textit{E-Int})}{((1+2)*3) \in E}(\textit{E-Mult})$$

C.2 Example II

$$\cfrac{\cfrac{\cfrac{2 \in Int}{2 \in E}(\textit{E-Int}) \qquad \cfrac{3 \in Int}{3 \in E}(\textit{E-Int})}{(2<3) \in E}(\textit{E-Lt}) \qquad \cfrac{\cfrac{4 \in Int}{4 \in E}(\textit{E-Int}) \qquad \cfrac{5 \in Int}{5 \in E}(\textit{E-Int})}{(4*5) \in E}(\textit{E-Mult}) \qquad \cfrac{21 \in Int}{21 \in E}(\textit{E-Int})}{(if\ (2<3)\ then\ (4*5)\ else\ 21) \in E}(\textit{E-Cond})$$