COMP 317: Semantics of Programming Languages

Problem Sheet 1: Solutions

1.
```
<DecimalDigit>    ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<DecimalNumeral>  ::=  <DecimalDigit> | <DecimalDigit><DecimalNumeral>
```

To get rid of leading zeroes,

```
<NLZ>    ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <NLZ><DecimalNumeral>
<DecimalNumeral>  ::= 0 |  <NLZ>
```

Note that it still follows from this definition that every numeral is either a digit or a numeral followed by a digit (challenge: write down a rigorous argument to show this is indeed the case).

2. We would need to change the inductive definition of `[[_]]`, because the BNF grammar tells us that a numeral is either 0, 1, or of the form *DN* for some digit *D* and numeral *N*. The digit *D* tells us whether or not to add `2 ^ n` (writing "^" for exponentiation), where `n` is the length of the numeral *N*. We can therefore define the denotation function by

```
[[ DN ]]  =  2 ^ length(N) + [[ N ]] .
```

As a bonus problem: define the `length` function (hint: define it inductively).

3.
```
<Alpha>    ::=  a | b | ... | z | A | ... | Z
<AlphaNum> ::=  <Alpha> | <DecimalDigit> | <AlphaNum><Alpha> |  <AlphaNum><DecimalDigit>
<Var>      ::=  <Alpha><AlphaNum>
```

We'll use the syntactic category <**Var**> for both variable and method names in the following question.

4.
```
<Scope>  ::=  public | private | protected |
<Type>  ::=  String | int
<ReturnType>  ::=  <ReturnType> | void
<ParamDecs>  ::=  <Type> <Var> | <Type> <Var>, <ParamDecs>
<Params>  ::=  <ParamDecs> |
<MethodSig>  ::=  <Scope> <ReturnType> <Var>(<Params>);
```

Note that <**Scope**> and <**Params**> each have an empty option, to allow for the default (absent) scope modifier, and the empty list of parameters.

5. We define it inductively (also considering the two possibilities for the least significant digit), for any numeral *N*:

```
increment(0)  =  1
increment(1)  =  10
increment(N0)  =  N1
increment(N1)  =  increment(N)0
```

For example, increment(11) = increment(1)0 = 100. Note that this follows the algorithm you learnt at school for adding (decimal) numerals column-by-column (the last equation is "carry 1"). Addition is defined by induction on both arguments:

```
 0 + N  = N
 1 + N  =  increment(N)
M0 + 0  =  M0
M0 + 1  =  M1
```

```
M0 + N0 =  (M + N)0
M0 + N1 =  (M + N)1
M1 + 0  =  M1
M1 + 1  =  increment(M)0
M1 + N0 =  (M + N)1
M1 + N1 =  increment(M + N)0
```

Again, this follows the column-by-column algorithm.

6. See any past exam paper.

7. Simplify the following as far as possible:
   1. [[`2 * ('x + 13)`]](init) = [[`2`]](init) * [[`'x + 13`]](init) = [[`2`]] * ([[`'x`]](init) + [[`13`]](init)) = 2 * (init(`'x`) + 13) = 2 * (0 +13) = 26 .
   2. [[`5 * (1 + 'x + 'y)`]](*S*) = ... (similarly) ... 5 * (1 + *S*(`'x`) + *S*(`'y`))
   3. [[`'x < 'x + 1`]](*S*) = ... *S*(`'x`) < *S*(`'x`) + 1 = true.

8. Show that for any expressions *E1*, *E2* and *E3*, and for any Store *S*, the following are true:
   1. [[`E1 + (E2 + E3)`]](*S*) = [[`(E1 + E2) + E3`]](*S*). Both reduce to [[*E1*]](*S*) + [[*E2*]](*S*) + [[*E3*]](*S*).
   2. [[`E1 * (E2 + E3)`]](*S*) = [[`E1 * E2 + E1 * E3`]](*S*). Both reduce to [[*E1*]](*S*) * [[*E2*]](*S*) + [[*E1*]](*S*) * [[*E3*]](*S*).

Grant Malcolm

Last modified: Fri Feb 11 13:49:12 GMT 2011