BNF, Grammars, the Simple AE Language

- **Context free grammars using BNF**

Getting back to the theme of the course: we want to investigate programming languages, and we want to do that *using* a programming language.

The first thing when we design a language is to specify the language. For this we use **BNF** (**B**ackus-**N**aur **F**orm). For example, here is the definition of a simple arithmetic language:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

Explain the different parts. Specifically, this is a mixture of low-level (concrete) syntax definition with parsing.

We use this to **derive expressions** in some language. **We start with** <AE>, which should be one of these:
* a number <num>
* <AE>, the text "+", and another <AE>
* the same but with "-"

It should be clear that the "+" and the "-" are things we expect to find in the input -- because they are not wrapped in <>s. Each of them is a **terminal**.
<num> is also a **terminal**: when we reach it in the derivation, **we're done**. (Small letters)
<AE> is a **non-terminal**: when we reach it, we have to continue with one of the options.

Example:

```
<AE> ::= <num>              ; (1)
       | <AE> + <AE>        ; (2)
       | <AE> - <AE>        ; (3)
```

We can derive the expression "1-2+3" in the following manner (proving "1-2+3" to be a valid <AE> expression):

```
<AE>                      ;          ==>
  <AE> + <AE>             ; (2)      ==>
  <AE> + <num>            ; (1)      ==>
  <AE> - <AE> + <num>     ; (3)      ==>
  <AE> - <AE> + 3         ; (num)    ==>
  <num> - <AE> + 3        ; (1)      ==>
  <num> - <num> + 3       ; (1)      ==>
  1 - <num> + 3           ; (num)    ==>
  1 - 2 + 3               ; (num)
```

- **Using Racket to implement a language.**

We could specify what <num> is (turning it into a <NUM> non-terminal):

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

But we don't -- why?
Because in Racket **we have numbers as primitives** and we want to use Racket to implement our languages. This makes life a lot easier, and we get free stuff like floats, rationals, etc.

Back to our example:

```
<AE>                      ;          ==>
<AE> + <AE>               ; (2)      ==>
<AE> + <num>              ; (1)      ==>
<AE> - <AE> + <num>       ; (3)      ==>
<AE> - <AE> + 3           ; (num)    ==>
<num> - <AE> + 3          ; (1)      ==>
<num> - <num> + 3         ; (1)      ==>
1 - <num> + 3             ; (num)    ==>
1 - 2 + 3                 ; (num)
```

This would be one way of doing this. Instead, we can **visualize the derivation using a tree**, with the rules used at every node. (Leave this on -- later show that this removes some confusion but not all.)

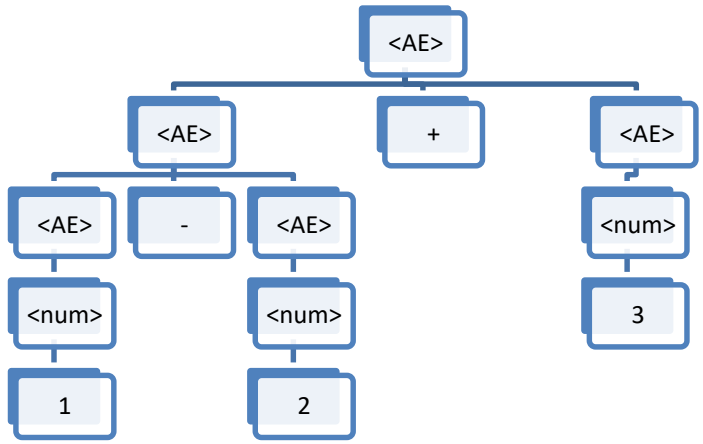

Figure 1 (Leaves are read left to right)

- **Ambiguity**

These specifications suffer from being ambiguous: **an expression can be derived in multiple ways (i.e., have multiple derivation trees).** Even the little syntax for a number is ambiguous -- a number like "123" can be derived in two ways that result in trees that look different. This **ambiguity** is not a "real" problem now, but it will become one very soon. We want to get rid of this ambiguity, so that there is a **single** (= **deterministic**) way to derive all expressions.

There is a standard way to resolve that -- we add another **non-terminal** to the definition, and make it so that each rule can continue to exactly one of its alternatives.

For example: this is what we can do with numbers:

```
<NUM>   ::= <DIGIT> | <DIGIT> <NUM>

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

Similar solutions can be applied to the **<AE>** BNF -- we either restrict the way derivations can happen or we come up with new non-terminals to force a deterministic derivation trees.

- **Restricting derivations**

As an example of restricting derivations, we look at the current (ambiguous) grammar:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

Example 1 – **Left association:**
Instead of allowing an <AE> on both sides of the operation, we force the left one to be a number:

```
<AE> ::= <num>
       | <num> + <AE>
       | <num> - <AE>
```

Now there is a **single** way to derive any expression, and it is always associating operations to the right: an expression like "1+2+3" can only be derived as "1+(2+3)".

Example 2 – **Right association:**
To change this to left-association, we would use this:

```
<AE> ::= <num>
       | <AE> + <num>
       | <AE> - <num>
```

Example 3 – **Semantically required precedence:**
But what if we want to force precedence? Say that our AE syntax has addition and multiplication (now (1+2)*3 is no longer semantically the same as 1+(2*3)):

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> * <AE>
```

We can do that same thing as above and add new non-terminals -- say one for "factors":

```
<AE>  ::= <num>
        | <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
```

Now we must parse any AE expression as additions of multiplications (or numbers). First, note that if <AE> goes to <fac> and that goes to <num>, then there is no need for an <AE> to go to a <num>, so this is the same syntax:

```
<AE>  ::= <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
```

To further loose ambiguity we do the following

```
<AE>  ::= <FAC> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <num> * <FAC>
```

Multiplying additions:
If we want to still be able to **multiply additions**, we can force them to appear in parentheses:

```
<AE>  ::= <AE> + <AE>
        | <FAC>

<FAC> ::= <num>
        | <FAC> * <FAC>
        | ( <AE> )
```

Loosing ambiguity:
Note that AE is still ambiguous about additions (e.g., 1+2+3). This can be fixed by forcing the left hand side of an addition to be a factor:

```
<AE>  ::= <FAC> + <AE>
        | <FAC>
```

```
<FAC> ::= <num>
        | <FAC> * <FAC>
        | ( <AE> )
```

We still have an ambiguity for multiplications (e.g., 1*2*3), so we do the same thing and add another non-terminal for "atoms":

```
<AE>   ::= <FAC> + <AE>
         | <FAC>

<FAC>  ::= <ATOM> * <FAC>
         | <ATOM>

<ATOM> ::= <num>
         | ( <AE> )
```

- **Loosing Ambiguity - the simpler solution**

You can try to derive several expressions to be convinced that derivation is always deterministic now.

But as you can see, this is exactly the cosmetics that we want to avoid -- it will lead us to things that might be interesting, but unrelated to the principles behind programming languages. It will also become **much** worse when we have a real language rather such a tiny one.

Is there a good solution? -- It is right in front of us: do what Racket does -- always use fully parenthesized expressions:

```
<AE> ::= <num>
       | ( <AE> + <AE> )
       | ( <AE> - <AE> )
```

To prevent confusing Racket code with code in our language(s), we also change the parentheses to curly ones:

```
<AE> ::= <num>
       | { <AE> + <AE> }
       | { <AE> - <AE> }
```

- **Using Prefix notation:**

But in Racket ***everything*** has a value -- including those `+'s and `-'s, which makes this extremely convenient with

future operations that might have either more or less arguments than 2 as well as treating these arithmetic operators as plain functions. In our toy language we will not do this initially (that is, `+' and `-' are **second order operators**: they cannot be used as values). However, since we will get to it later, we'll adopt the Racket solution and use a fully-parenthesized prefix notation:

```
<AE> ::= <num>
       | { + <AE> <AE> }
       | { - <AE> <AE> }
```

(Remember that in a sense, Racket code is written in a form of already-parsed syntax...)
