# Grammar for arithmetic expressions

Reading material: These notes and an implementation (see course web page).

The best way to prepare [to be a programmer] is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and fished out listings of their operating system. (Bill Gates)

First learn computer science and all the theory. Next develop a programming style. Then forget all that and just hack. (George Carrette)

Look at this website:

Grammar is boring because the set of Sentences (8) is finite.

---

**Recursive grammar**

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence ::= Sentence and Sentence</td>
<td>Recursive definition of Sentence</td>
</tr>
<tr>
<td>Sentence ::= Noun Verb Noun</td>
<td>A Sentence is a Noun followed by a Verb followed by a Noun</td>
</tr>
<tr>
<td>Noun ::= boys</td>
<td>A Noun can be a boys</td>
</tr>
<tr>
<td>Noun ::= girls</td>
<td>A Noun can be a girls</td>
</tr>
<tr>
<td>Verb ::= like</td>
<td>A Verb can be a like</td>
</tr>
<tr>
<td>Verb ::= see</td>
<td>A Verb can be a see</td>
</tr>
</tbody>
</table>

Example of Sentence: boys see girls girls like boys and boys see girls girls like boys and boys see girls girls like boys and boys see girls and boys see girls and boys see girls boys like girls

---

**Notations used in grammars**

Notation used to make grammars easier to write:

- `{ ... }` stands for zero or more occurrences of ...
- `<b | c>` stands for either a b or a c.
- An Expression is a Term followed by either + or – followed by a Term
- An Expression is a Term + Term or a Term – Term
- An Noun phrase is zero or more Adjectives followed by a Noun

---

**Grammar**

A grammar: set of rules for generating sentences of a language.

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence ::= Noun Verb Noun</td>
<td>A Sentence is a Noun followed by a Verb followed by a Noun</td>
</tr>
<tr>
<td>Noun ::= boys</td>
<td>A Noun can be a boys</td>
</tr>
<tr>
<td>Noun ::= girls</td>
<td>A Noun can be a girls</td>
</tr>
<tr>
<td>Verb ::= like</td>
<td>A Verb can be a like</td>
</tr>
<tr>
<td>Verb ::= see</td>
<td>A Verb can be a see</td>
</tr>
</tbody>
</table>

Example of Sentence: boys see girls girls like boys

---

**Syntax trees**

Expression ::= E $ – $ marks the end of the Expression

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ::= T { &lt;+ I -&gt; T }</td>
<td>An E is a T followed by any number of things of the form</td>
</tr>
<tr>
<td>T ::= F { &lt;*&gt;I/&gt; F }</td>
<td>Here are four Es: T T + T T + T + T</td>
</tr>
<tr>
<td>F ::= Integer</td>
<td></td>
</tr>
<tr>
<td>F ::= –F</td>
<td></td>
</tr>
<tr>
<td>F ::= ( E )</td>
<td></td>
</tr>
</tbody>
</table>

Expression ::= E $ F ::= Integer

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ::= T { &lt;+ I -&gt; T }</td>
<td></td>
</tr>
<tr>
<td>T ::= F { &lt;*&gt;I/&gt; F }</td>
<td></td>
</tr>
<tr>
<td>F ::= Integer</td>
<td></td>
</tr>
</tbody>
</table>

Expression E E T T T F F F 2 $ 2 + 3 $
Trees

The node labeled E is the parent of the nodes labeled T and +. These nodes are the children of node E. Nodes labeled T and + are siblings.

Grammar gives precedence to * over +

Expression ::= E $  
F ::= Integer  
E ::= T { <+ | –> T }  
F ::= – F  
T ::= F {<* | /> F }  
F ::= ( E )

Grammar gives precedence to * over +

Expression ::= E $  
F ::= Integer  
E ::= T { <+ | –> T }  
F ::= – F  
T ::= F {<* | /> F }  
F ::= ( E )

Writing a parser for the language

Expression ::= E $  
F ::= Integer  
E ::= T { <+ | –> T }  
F ::= – F  
T ::= F {<* | /> F }  
F ::= ( E )

Parser for a language is a program that reads in a string of characters and tells whether it is a sentence of the language or not. In addition, it might construct a syntax tree for the sentence.

We will write a parser for the language of expressions that appears above.

Writing a parser for the language

Expression ::= E $  
F ::= Integer  
E ::= T { <+ | –> T }  
F ::= – F  
T ::= F {<* | /> F }  
F ::= ( E )

Scan.getToken() is always you the token being processed. Scan.scan() deletes the token being processed, making the next one in the input the one being processed, and returns the new one.

Scanner is the part of the program that reads in characters and produces tokens from them, deleting all whitespace.

22 + 35 * – 46 / 2 $

a1 a21 a6
Token Token Token

22 + 35 * – 46 / 2 $

22 + 35 * – 46 / 2 $

22 + 35 * – 46 / 2 $

22 + 35 * – 46 / 2 $

22 + 35 * – 46 / 2 $
Writing a parser for the language

Expression ::= E $  
F ::= Integer  
E ::= T { <+ | –> T }  
F ::= – F  
T ::= F { <* | /> F }  
F ::= ( E )

For E, T, F, write a method with this spec (we show only E):

/** Token Scan.getToken() is first token of a sentence for E. Parse it, giving error mess. if there are mistakes. After the parse, Scan.getToken should be the symbol following the parsed E. */

public static void parseE()

2 + ( 3 + 4 * 5 ) + 6

after call parse(), situation is this:

2 + ( 3 + 4 * 5 ) + 6

We now use the blackboard. You should look at the final program, which is on the course website. Download it and play with it. Parts of it will be discussed in recitation this week.