Basic Concepts

The Congo Parser Generator is, at its core, a classic *recursive descent parser generator*. Now, I have to assume that many readers do not really understand what that means. For now, let me just make the following key point: when you use a tool like this to generate your parser, you are really implementing two separate *machines* -- the *lexer* (a.k.a. *scanner* or *tokenizer*) and the *parser* itself.

Of the two, the lexer is the lower level machine. Its job is to break the input into *tokens*, chunks of text input that are effectively the smallest pieces of text that have some actual meaning in the *language*.

Concretely, consider the following line of Java code:

```
    int foo = bar + baz.bat;
```

At the *pre-lexical* stage, that is just a sequence of characters:

```
    .
    .
    .
    .
    i
    n
    t
    .
    f
    o
    o
    .
    =
    .
    b
    a
    r
    .
    =
    .
    b
```

```
a
z
.
b
a
t
;

```

The lexer's job is to group those characters into the following:

- `int` (a Java *keyword*)
- `foo` (a Java *identifier*)
- `=` (a Java *operator*)
- `bar` (another *identifier*)
- `+` (another *operator*)
- `baz` (*identifier*)
- `.` (The dot operator)
- `bat` (*identifier*)
- `;` (A Java *delimiter*)

That is what the *lexical* stage is about, turning a stream of characters into a stream of *tokens*. As you surely see, some of the input characters are really just ignorable, most typically *whitespace* -- for example, the space between the *keyword* `int` and the *identifier* `foo` after it, or the spaces immediately before and after the `+` operator. Those are not even strictly necessary. There is no *semantic* difference between `foo + bar` and `foo+bar`. The extra spaces would only be for human readability. By the same token, the line-feed character that terminates the line is simply ignored by a Java compiler, which would be just as happy (*sorry for the anthropomorphism!*) if all the Java code was on a single line. However, code that is not broken into lines would obviously be very onerous for a human being to work with!

While the lexer is concerned with breaking a stream of characters into *tokens*, the parser works at the *syntactic* level. It takes that stream of *tokens* that come from the lexer and generates a *tree* (an inverted tree data structure) that could be visually represented as follows:

```
<FieldDeclaration>
  <PrimitiveType>
    int
  <VariableDeclarator>
    <Identifier>
      foo
    <Operator>
      =
```

```
        <AdditiveExpression>
          <Name>
            <Identifier>
              bar
            <Operator>
              +
          <Name>
            <Identifier>
              baz
            <Operator>
              .
            <Identifier>
              bat
      <Delimiter>
        ;
```

So that is what the parser builds. Or something like that... Again, the problem is partitioned into the *lexical* and *syntactic* side. The lexer takes the stream of characters and turns that into a stream of tokens. The parser takes that stream of tokens and produces a tree-like data structure as we see above.

Now, it should be clear that this lexer/parser is invariably part of a larger system, very likely a compiler or interpreter. A compiler can't do anything with that raw sequence of characters. Actually, it still can't do much with a stream of tokens either, though we're getting a bit warmer... However, the above tree-like data structure really is something that a program can do something with! This is because it is arranged in a way that reflects the *logic and structure*, a.k.a. *semantics*, of the language we are working with.

Revision #4
Created 22 February 2023 11:43:33 by Jonathan Revusky
Updated 3 March 2023 15:51:56 by Jonathan Revusky