10.1.2 An expression parser

Description

In this section, we show how to write a parser for our simple expression grammar. The parser reads a string of characters and checks whether or not the string is a valid expression as described by the grammar.

We assume that the input string to the parser is in the String `inn`. We define a method `nextChar` that assigns the next character in `inn` to the variable `ch`, but skipping all blank characters. The variable `pos` keeps track of the current position in `inn`. When the end of `inn` is reached, the null-character is assigned to `ch`.

```
inn: ref String
pos: var integer
ch: var integer
nextChar:
   -- read the next character from inn and assign it to ch
   -- skip white space like blanks and end-of-lines
   -- return ch = 0, if at the end of inn
   pos := pos + 1
   if (pos <= inn.length) :then
      ch :=  inn.get[pos]
      if (ch = ' ') :then
         restart(nextChar)
    else
      ch := 0
```

The parser is based on a technique of so-called *recursive* methods – we return to recursion later.

For each nonterminal symbol of the grammar we define a method that checks if the input is in accordance with the nonterminal.

For the nonterminal <Exp> , we define the method `exp`:

```
   exp:
      -- parse an expression generated from <Exp>
      term
      if (ch = '+') :then
         nextChar
         restart(exp)
```

The `exp`-method starts by calling the `term`-method, which we assume parses an expression as generated by <Term>.

If the next character is a `'+'`, `nextChar` is called to assign the next char into `ch` and then execution of `exp` Is restarted.

Given that `term` actually parses a string described by <Term>, `exp` will parse an sequence:

```
<term> + <term> + ... + <term>
```

We may now show the `term`-method, which is similar to `exp`:

```
term:
    primary
    if (ch = '*') :then
        nextChar
        restart(term)
```

The `term`-method assumes that the method `primary` parses a string described by <Primary> and will then parse a sequence:

```
<Primary> * <Primary> * ... * <Primary>
```

Next we look at the `<Primary>` rule, which has the form:

```
<Primary> ::= "(" <Exp> ")" | <Number>
```

This give us the following `primary`-method:

```
primary:
   -- parse an expression generated from <Primary>
   if (ch = '(') :then
      nextChar
      exp
      if (ch = ')') :then
         nextChar
      :else
         syntaxError(1)
   :else
        number
```

If the next character (`ch`) is a left-bracket (`"("`), it calls `nextChar` and then `exp`. When `exp` returns, the next character must be a right-bracket (`")"`), otherwise we have an error in the string – and a syntax error is reported.

If the next character is not a left-bracket, the `number`-method is called and again we assume that `number` parses a string as described by `<Number>`.

The call of the `exp`-method is an example of a recursive call. We return to recursion in the section below.

Next we show the `number`-method and the `digit`-method:

```
number:
   -- parse an expression generated from <Number>
   digit
   moreDigits: do
      if (ascii.isDigit(ch)) :then
         digit
         restart(moreDigits)
digit:
   -- parse an expression generated from <Digit>
   if (ascii.isDigit(ch)) :then
      nextChar
   :else
      syntaxError(3)
```

The `number`-method expects at least one digit, which is reflected in the `digit`-method that repoerts a syntax error if `ch` is not a digit.

Finally we may show how to start parsing:

```
parse:
   -- parse the expression assigned to inn below
   inn :=  "10 + 11 * (1 + 9)"
   console.print("Parse: " + inn + "\n")
   nextChar
   exp
   if (ch <> ascii.null) :then
          syntaxError(3)
```

The `parse`-method calls `exp` and when `exp` returns, `ch` must be the `ascii.null` character.

Recursion

Recursion is a technique for defining an entity in terms of a simpler version of itself. In order for this to work, there must be a terminating condition that can be defined without applying recursion, i.e. a recursive definition must be to a part of the entity that is simpler than the original entity.

The expression grammar is and example of a recursive definition. A nonterminal like <Exp> is defined using <Exp> on the rigt side of the rule, the same is the case for <Term>. The rule for <Primary> is an example of an indirect recursion, since <Primary> is defined by means of <Exp> which in turn is defined by means of <Primary>.

The terminating conditions here is that an <Exp> can be a <Term>, a <Term> can be a <Primary>, which can be a <Number>.

The methods of the parser are example recursive methods and they follow the same scheme as the grammar except that only the indirect recursion of `exp` via `primary` is explicitly encoded.

The snapshot below shows the situation of the the invocation of parse of the string `"11 * (1 + 9) + 12"` at the point where the substring `"11 * ("` has been parsed and `primary` has made a recursive invocation of `exp` which has invoked `term`. The red arrow (`-->`) shows the point of execution in `exp` and the black arrow (`-->`) shows the point of invocation of `exp` in `primary`.

```
  primary:
     -- parse an expression generated from <Primary>
     if (ch = '(') :then
        nextChar
-->     exp
        if (ch = ')') :then
           nextChar
        :else
           syntaxError(1)
     :else
        number
  exp:
     -- parse an expression generated from <Exp>
--> term
    if (ch = '+') :then
       nextChar
       restart(exp)
```