The SyntX Parser Library

The SyntX parser library is a set of C++ classes that enables users to create a parser for an LL(k) grammar by giving the production rules with an EBNF-like notation as C++ expressions. There is thus no need for a precompiler, the parser can be an integral part of a C++ project.

Recursive descent parsing

The SyntX library is based upon the theory of recursive descent parsers (RDPs). In order to gain a deep and confident understanding of the library, one has to be familiar with the rudiments of recursive descent parsing.

This section provides a brief introduction with a very simple example. It also introduces some of the types and notations that are used in the source code of the library as well.

The Extended Backus-Naur Form

EBNF[1] is a widespread notation that can be used to express context-free grammars. It is the extension of the Backus-Naur Form (BNF) and actually several EBNF styles exist. This document uses the one defined by W3C[2].

The "Hello World" grammar that is used to introduce parsing describes a mathematical expression containg numbers, the four basic operations and an arbitrary depth of parentheses. The grammar below is even simpler than that, it allows only addition, so the way precedence can be handled is not shown. It allows an infinite depth of grouping using parentheses however. The ease with which recursive structures can be handled by EBNF grammars and RDPs is a very important feature and is often exploited.

```
addition = addend ('+' addend)*
addend = [0-9] | expression
expression = '(' addition ')'
```

The production rule named `addition` is the *start rule*. The text that we're trying to analyze should conform to this rule as a whole. The details can be investigated by following the references to other rules.

[1]Extended Backus-Naur Form – en.wikipedia.org/wiki/Ebnf
[2]World Wide Web Consortium – www.w3.org

The `addition` rule states that an addition is made up of an addend that can be followed by an arbitrary number of addends each preceded by a '+' operator. The `*` operator in EBNF allows zero or more occurence of a rule (or a sequence of rules). This means that an addition may consist of a single addend as well, which is OK: a number can be thought of as a mathematical expression.

If we wanted to have at least two addends seperated by a '+' operator, than EBNF's operator `+` should be used, which allows one or more occurences.

An addend in our simple grammar can be either a single digit (given here by the shorthand used to describe character sequences in EBNF) or an expression. The `|` operator stands for *alternation*.

The interesting part lays in the `expression` rule. An expression is an addition enclosed by parantheses. This is how the grammar becomes recursive and how an expression of infinite complexity can be described by a handful of production rules.

Let's see some examples! The simplest possible expression is a single digit:

```
8
```

The start rule is where the matching starts, so we assume that the our text is an `addition`. An addition starts with an `addend`. An `addend` can be a digit so the text indeed starts with an addend. Thus the first part of the `addition` rule matched, we may move on to the rest of the text. Next we're looking for a '+' character which we don't find. Luckily this part of the rule is optional, so we get to the end of the rule having matched the entire text. This is a successful match.

Let's go for something more complex next:

```
2 + 3 + (4 + 8)
```

The `addition` rule will find two digits with a '+' character in between. It could already stop there as we have a sum which complies to the rules, but we haven't reached the end of the text and actually the rule allows more than two addends. So the parsing continues, we find a second '+' character and then an opening parenthesis. This is OK, as the `addend` rule doesn't only match digits, `expressions` are also allowed.

So next we're trying to match the `expression` rule which contains an `addition` between parentheses. That's exactly what we have here so, again, we have a successful match. Please note that the `addition` inside the `expression` can again turn into an `expression` and then into an `addition` – this is exactly how the arbitrary depth of the expression is analyzed.

Functions corresponding to each production rule

In a recursive descent parser there is a function corresponding to every production rule. These functions receive the context of parsing (i.e. the text to be parsed and the current position) and return a Boolean value which is true if the function could match a certain substring of the text and could move the position further towards the end of the text. If the function returns false, the position is not altered.

Following is an example function that matches one character of the text if it can be found in the string serving as a character set taken as an argument. The structure of this function is characteristic of the rule methods in the SyntX library.

```
 1 bool character(match_range &context, std::string const &characters) {
 2   match_range local = context;
 3
 4   if (local.first == local.second) return false;
 5
 6   for (auto c: characters) {
 7     if (*local.first == c) {
 8       ++local.first;
 9       context.first = local.first;
10       return true;
11     }
12   }
13
14   return false;
15 }
```

The match_range type is an std::pair that holds two std::string::iterators. It describes the context of the parsing: the first element points to the current position (the character that should be analyzed next), the second to the end of the text.

```
using match_range = std::pair< std::string::const_iterator,
                               std::string::const_iterator >;
```

The function has two arguments, the context and the character set that contains the characters that are accepted. The context is taken as a non-constant reference because the function sets the current position after the the last character it could match.

In line 2 a copy of the context is created. This copy will reflect the analysis performed by the function. A complex parser function can delve deep into a string before finding out that it doesn't match it after all. It might also call a series of other parsing functions on the way that also alter this value, so it is absolutely necessary to keep a copy of the original position and only change that value when there is a successful match.

Every parser function that operates at the character level should always check whether the end of the text has been reached. This can be seen in line 4.

As long as a function finds characters it can consume, it advances the current position, i.e. the first element in the local copy of the context (line 8).

If a function matches a certain substring of the text and can not advance further, two things need to be done: the context taken by reference has to be changed to reflect the advancement in the analysis of the text and it has to return true (lines 9-10). Otherwise it has to merely return false (line 14), the context is left unchanged.

Recursive descent

The example above showed a simple a parsing function working at the character level. It doesn't call any other function, instead it decides on its own whether the text at the given position matches it or not. This is because that rule describes a so called *terminal symbol*, one that corresponds to a symbol actually appearing in the text, in this case, a letter.

Other rules might define symbols that contain other symbols and define a structure that these symbols should have in order to comply to the rule. The composite symbols are called *non-terminal symbols*.

In RDPs non-terminal symbols can be parsed using functions that call other parsing functions just as any rule can be referenced in the definition of an EBNF rule.

Let's see an example for such a function: the `expression` rule in the simple grammar seen earlier.

```
bool expression(match_range &context) {
  match_range local = context;

  if (
    character(local, "(") && addition(local) && character(local, ")")
  ) {
    context.first = local.first;
    return true;
  }

  return false;
}
```

It is interesting to note that as this function doesn't operate at the character level (every character consumed by the rule is processed by one of the functions called by it), it doesn't need to check for the end of the text – it has to be done by the low-level functions.

The sequence of the sub-rules is realized by the `&&` operator of the C++ language. Short-circuit evaluation is exploited here: if the first rule doesn't match and thus returns `false` then the second is not called and the entire expression will get a `false` value.

Furthermore the order of evaluation is fixed too and goes from the left to the right. So `addition` will receive an updated `local` – the value that was altered by the first call to `character`.

So when all three functions return `true`, the body of the `if` statement is evaluated and `context` receives a value updated by all three functions and now pointing to the next position of the text to be parsed.

Alternatives can be realized using the `||` operator where short-circuit evaluation comes handy again as the second function gets called only if the first failed to match (in which case the first doesn't alter the context which is also important).

Please note that composite logical expression should not be constructed as they can lead to mispositioned iterators. Let's investigate the following code fragment:

```
if (
  (rule1(local) && rule2(local)) || rule3(local)
) {
  ...
}
```

Let's assume that `rule1` matches but `rule2` doesn't, so `rule3` gets a chance. The problem is that `rule1` moves the position that `local` points to. Unfortunately it doesn't get corrected before `local` is fed to `rule3` so `rule3` will try to match from a different position then the one where `rule1` started from and this is not what we intended to do.

If the above expression is realized in two seperate functions, one containing the sequence (AND logic) and the other containing the option (OR logic) then this situation doesn't occur as the functions will only alter the context if they match. This is done automatically in the SyntX framework but it is something the programmer has to pay attention to if the parser is handwritten.

Actions during parsing

All the example functions shown up to here did was to tell whether their input complied to their requirements or not. If the task of the parser is merely to determine if a text conforms to a specific grammar then this is sufficient. Otherwise the functions should perform operations to yield a result of the parsing. This can be in the form of an AST (abstract syntax tree) or pratically any data that can be generated based on the input.

When the parsing functions are handwritten the actions to be performed when a match is found can be put inside the parsing function resulting in a code where parsing and data processing is intertwined. For simple problems this is a perfect solution and is easy to handle.

When problems become more complex this method becomes tiresome or even impracticable: there are cases where data processing can not be performed at the time of parsing as additional knowledge is needed that can only be extracted later, possibly after the parsing of the entire input is finished.

In the SyntX framework an `std::function` (which can wrap a plain function, a method, a function object or a lambda) can be assigned to any rule – these are called *action*s. The function receives the matched range as a `std::string` and can do whatever is needed when the given rule is successfully matched.

There is just one problem with this approach: if a complex data is defined by a sequence of rules such as in this case

```
complex = rule1 rule2 rule3
```

then we can not be sure that the entire expression will match until `rule3` returns `true` but we need to extract the results on a rule-basis, otherwise we get the matched range of the three rules packed in one string needing yet another extraction.

A solution to this problem can be to store the results of the three rules in temporary variables and build the complex data only when all three matched successfully.

In SyntX actions are added to rules using `operator[]` and they can be assigned to a group of rules as well:

```
complex = ( rule1[action1] rule2[action2] rule3[action3] )[action4]
```

Please note that this is not exactly the correct syntax – it is only shown for demonstration purposes here.

The SyntX framework

In the next few sections the framework's structure and the basics of its operation are explaing briefly. For the details please refer to the Doxygen documentation, which can be generated from the source code with the help of the Makefile provided with the project (`make docs`).

The `base_rule` class

The base class of every rule is `base_rule`. It defines two data types used throughout the framework, contains the semantic action assigned to a rule and performs the basic administrative tasks concering the matching process. The exact instructions regarding the matching have to go in a virtual function defined as pure virtual in this class (`test`).

One of the data types define in `base_rule` has already been mentioned on page 3, it is the `match_range` which is used for two purposes: it defines the limits between which the parsing is done and also the range matched by a given rule.

The other type is `semantic_action` discussed earlier on page 5:

```
using semantic_action = std::function<void(std::string const &)>;
```

One action can be assigned to a rule and it's stored in the `the_semantic_action` data member of the class.

The `match` method handles the tasks associated with matching.

```
bool base_rule::match(match_range &context, match_range &
    the_match_range) {
  match_range a_range;

  if (test(context, a_range)) {
    the_match_range.first = a_range.first;
    the_match_range.second = a_range.second;

    if (the_semantic_action) {
      std::string the_matched_substring(the_match_range.first,
          the_match_range.second);
      the_semantic_action(the_matched_substring);
    }

    return true;
  }
  else return false;
}
```

The `match` method calls the virtual `test` to find out whether the text conforms to the rule at the current position. If it doesn't, the method simply returns `false`, while in the case of a successful match, the matched range is stored in the local variable `a_range`. This value is delegated to `the_match_range`, which is a reference of a variable taken as an argument.

If a semantic action has been assigned to the rule, it is called and the result of the matching process is given to it as a `std::string`.

The `base_rule` class has a virtual funtion called `clone` the purpose of which is to make the entire class hierarchy clonable. More on this can be found on page 6.

Example of a rule subclassed from `base_rule`

Realizing EBNF operators

The `rule` class

Contents