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`. The problem is that 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 8.

Example of a rule subclassed from `base_rule`

Let's look at a simple example of how an actual rule can be created using the `base_rule` class. It is the realization of the previously discussed character rule (page 3) in the SyntX framework.

As the `test` function to be overridden has a fixed argument list, the set of allowed characters is given in the constructor and stored as a data member. Only two functions need to be written: `clone` – which is basically a oneliner that dynamically creates a copy of the rule and `test`.

The code of `test` resembles closely to the `character` function on page 3.

```
bool character::test(base_rule::match_range &context, match_range &
    the_match_range) {
  if (context.first == context.second) return false;

  base_rule::match_range local_context = context;

  for (auto allowed_character: allowed_characters) {
    if (*local_context.first == allowed_character) {
      ++local_context.first;

      the_match_range.first = context.first;
      the_match_range.second = local_context.first;
      context = local_context;
      return true;
    }
  }

  return false;
}
```

The same operations are performed as seen earlier. First the function checks whether the current parsing position is at the end of the input or not. If the end has been reached there is nothing to do but return `false`. This operation has to be done in every rule that operates at the character level.

If there is input to process a local copy of the context is created. This is not necessary in this case, this code rather shows the generic way of writing a rule.

Then it iterates over the set of allowed characters and if any of them matches the current reading position then the context is adjusted, the match range is set and `true` is returned.

The goal of this example was to show how easy it is to complement the framework with a new rule. In fact the need to create new rules should arise very rarely as the framework contains a great number of simple classes with which almost every task can be solved.

Realizing EBNF operators

Next we look at the realization of EBNF operators (sequence, alternation, etc.) as subclasses of `base_rule`. Naturally, these classes are never instantiated manually – it would make the grammars unreadable. There are operators that do this so that the grammars defined in SyntX will look very much like their EBNF counterparts.

Our example is alternation which will try to match one of the two rules given to it. It will first try the first one and then the second if the first fails. This means that the first rule will have a precedence over the second and this should be kept in mind when constructing the grammar[3].

The `alternation` class stores the `shared_ptrs` of two rules and overrides `clone` and `test`. Let's analyze the latter:

[3]This can be important when the input matched by one of the rules is the starting substring of the one matched by the other. In such a case, if the shorter rule comes first, it will match and the second will never be run. There could be a special alternation rule which chooses the longer of the two matches, but SyntX doesn't provide such a rule – it is easy to write though.

```
 1  bool alternation::test(base_rule::match_range &context, base_rule::
        match_range &the_match_range) {
 2    base_rule::match_range first_range, second_range;
 3    base_rule::match_range local_context = context;
 4
 5    if (first->match(local_context, first_range)) {
 6      the_match_range.first = first_range.first;
 7      the_match_range.second = first_range.second;
 8      context = local_context;
 9
10      return true;
11    }
12
13    local_context = context;
14
15    if (second->match(local_context, second_range)) {
16      the_match_range.first = second_range.first;
17      the_match_range.second = second_range.second;
18      context = local_context;
19
20      return true;
21    }
22
23    return false;
24  }
```

The function first tries to match the first rule. If it succeeds, `the_match_range` and `context` is adjusted and `true` is returned.

If the first rule didn't match, then the local copy of the context is set to the original value and the second rule is given a chance. Theoretically the `local_context` should not be altered by the first rule if it doesn't match, so line 13 is unnecessary – it is there to make sure that no mistake is made even if the rule does not conform entirely to the expectations.

The `rule` class

The operators that make the grammars EBNF-like also make the framework a bit complicated. Let's first consider a single rule that contains only built-in rules such as the one in the following example (which doesn't follow the correct syntax of the framework to ease the understanding):

```
binary_digit = (character("0") | character("1")) character("b")
```

In order to avoid memory leaks and to make grammars easier to write and read, the rules are not created dinamically. This means that unnamed, temporary local variables are created on the right side of the expression above (as e.g. `character("0")` is the constructor of the class `character`).

This means that `operator |`, which creates an `alternation`, receives two temporary objects as arguments. As it needs to accept any kind of `base_rules`, it can not take them as value, so the only choice is to take them as constant reference. This OK, as long as it doesn't want to store these values, as the reference of temporaries should not be stored.

The problem is that every operator will indeed want to store the rules that are given to them – as we have seen that earlier on page 7. This is because the evaluation of these expressions happens long after their construction, so actually some kind of a syntax tree is built automatically, which is traversed during the parsing process. Thus, the `test` method of the temporary rules is called long after they get deleted.

The solution to this problem is to create copies of these rules. As it has been shown, not much is stored in these rules (a few pointers and maybe short strings), so copying them is a relatively cheap operation.

The copies have to be made inside the operator functions that take the constant reference of the temporaries. The problem here is that all they know of these rules is that they are subclasses of `base_rule`. Virtual constructors would come handy in this situation if they existed. Instead we need the `clone` function, a typical solution used in heterogeneous containers in several languages. This is why `base_rule` defines the `clone` function as pure virtual – it makes every subclass override it, so that the framework can rely on its existence in every rule object.

And indeed, what e.g. `alternation` does in its constructor is that it instantly makes copies of the two rules it received using their `clone` function:

```
alternation::alternation(
  base_rule const &first,
  base_rule const &second
) :
  first(first.clone()),
  second(second.clone()) {}
```

And all that `operator|` does is create an instance of `alternation` and return it in a `rule`:

```
rule operator |(base_rule const &first, base_rule const &second) {
  return rule(
    std::shared_ptr<base_rule>(new alternation(first, second))
  );
}
```

We will soon get to why this is needed and what the class `rule` is used for.

If there were no rules that refer to other composite rules then we could stop here and there would be no need for the `rule` class. We could build expressions of arbitrary complexity and get a `shared_ptr<base_rule>` as a result, which contains dynamic copies of other rules that also contain dynamic copies of rules in a tree structure that reflect the structure of the grammar.

Luckily, thanks to the smart pointers provided by C++11's STL, there is no need to free this tree manually, though it would not be a very complicated task.

As a grammar grows, it becomes inevitable to introduce intermediate rules to avoid the need to write one very complex rule that could overflow with repetitions of the same expression. These rules are also called the non-terminals of a grammar, while the built-in rules of the framework can be thought of as the terminal symbols.

Let's consider the following grammar (our introductory grammar repeated):

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

As you can see, each of the rules here contain references of other rules. None of them consists of merely terminals. Actually this is because this is a recursive grammar describing a structure that can be arbitrarily deep but the statement that a large proportion of the rules in every grammar contains references to other rules is true nevertheless.

The problem arises already in line 1: the alternation operator will try take the copy of addend, a rule that has not been defined yet! Of course it exists, a previous line has to contain the following declaration:

```
rule addend;
```

but it is a completely empty rule until it is defined in line 2. This means that the copy that is stored by the instance of `alternation` and eventually the rule `addition` is not the final value of the `addend` rule!

The same is true for the rules `addend` and `expression`.

The Fundamental Theorem of Software Engineering[4] attributed to David J. Wheeler[5] says, that "All problems in computer science can be solved by another level of indirection." A statement that might have been meant partly as a joke, it is absolutely true for the above problem.

What we need, is a representation of a composite rule that doesn't change when it is filled with contents so that a copy of its empty state is exactly identical to its final value after its definition has been processed. It might sound unrealizable at first, but actually all we need is a pointer to a pointer (i.e. another level of indirection).

The class `rule` is a subclass of `base_rule` that stores a `shared_ptr` to a `shared_ptr` of a `base_rule`.

When a `rule` is constructed, it dynamically creates a default `shared_ptr` (a `nullptr`) and stores it in its pointer-to-a-pointer data member called `the_rule`. The address of the dynamic pointer will never change in the lifespan of the variable, only the contents of the inner pointer will do: it will be overwritten with the address of the cloned rule assigned to the rule.

So we have realized a class that does not change when it is filled with contents and a copy of which is valid no matter when it is created – before or after the construction of the contents. All it takes is a pointer to a pointer.

Now the grammar on page 9 doesn't cause any problems: the composite rules referred to in the definition of a rule shall be stored as `rule`s. But how can an operator decide which rule is a terminal and which is a non-terminal? Well it doesn't have to: every operator will create `rule`s and it simply solves the problem. This might seem wasteful as constructing a rule implies the dynamic allocaton of a `shared_ptr`. Considering the fact though that every rule is copied many times and that the copy constructor of a `rule` is trivial, as the only data member it has is a `shared_ptr`, this method is feasible.

Thus class `rule` is basically a wrapper class introducing the needed extra level of indirection. It follows that its methods basically delegate the tasks to the rule that's address is stored. E.g. the `test` method looks like this:

```
bool rule::test(match_range &context, match_range &the_match_range) {
  if (!(*the_rule)) return false;
  return (*the_rule)->match(context, the_match_range);
}
```

Undefined rules return `false`, other rules return what the rule inside returns.

[4]en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering
[5]en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

One might ask why the name of the `rule` class is so short and one that doesn't reflect its purpose. The reason for this is that this is the class that is instantiated by the user of the framework who doesn't need to understand the mechanisms behind the scene. For someone who merely creates grammars using the framework these variables represent rules of a grammar and nothing more, so their type name should exactly be `rule`. This way the code is short, easy to understand and reflects how the writer of code understands it, which is more important than what the creator of the framework sees.

This is why the base class of the rule hierarchy was called `base_rule` and not simply `rule`.

The operators of the framework

The header that contains the definition of class `rule` also hosts a number of global operators, the ones that represent the EBNF operators in the framework. Although these are global operator by nature, they create `rule` objects so they really belong to the class even if the language has no way to express this relationship. This is why they were placed here.

Unfortunately EBNF operators can not all be realized in C++ as only the C++ operators are available for overloading. The ones closest to EBNF were chosen.

The rule of thumb that one should only overload an operator if it operates "as ints do" does not apply here, in my opinion, as this is an entirely different context where operators have a different meaning, one that is known to everyone familiar with the field. Thus, I used operators quite freely and did not hesitate when I assigned an operator a function that didn't even resemble the operation that it performs on ints.

Another problem is that there is no operator in EBNF for concatenation. If the name of two rules are written after eachother then it means that the first has to follow the second. This is something that can not be done in C++, there has to be an operator between every two operands. This is an example for an operator that was defined randomly. In this case the shift left operator was used (<<) as it is already used for a similar purpose in STL. For the sake of consistency, the operator <<= was used as the assignment operator.

The following table summarizes the operators of the SyntX framework.

Operator	**Meaning**
|	alternation
!	option
+	repetition
*	repetition of zero or more time
-*rule*	consume whitespaces before *rule*
~*rule*	consume whitespaces except new line before *rule*
<<	concatenation
<<=	assignment to a rule

The built-in rules (terminal symbols)

There are a great number of built-in rules defining the terminal symbols in the framework. Most ordinary parsing tasks can be solved using these rules making it unnecessary to write custom rules (although this is something that can easily be done and actually extension of the framework is highly recommended should the need arise).

The following table summarizes the built-in rules (terminal symbols) in the framework.

Built-in rule	**Consumes**
`character`	one character if it is in a given set
`eol`	the end-of-line character
`epsilon`	nothing (and always matches)
`identifier`	a string that can be an identifier
`integer`	an integer (signed or unsigned)
`keyword`	the given keyword
`range`	one character in the given range
`real`	a real number
`string`	a string literal with a given delimiter and escape character
`substring`	a given word even as a substring

The `epsilon` rule which does not consume anything but always returns `true` might seem useless. As the consumption of white spaces can only by realized with prefix operators (`~` and `-`), there always has to be a rule after them. When the white spaces at the end of the input are to be consumed, `epsilon` comes into picture. This can be necessary if one wants to test whether the match was a full match, i.e. the entire input matched the grammar and not just a part of it.

An interesting result of the C++ operator overloading is that the postfix EBNF operators `*`, `+` and `?` become prefix operators and `!` has to be used instead of `?`.

An example

Finally the introductory grammar on page 1 is given here in SyntX to show how the operators and built-in rules can be used. Please refer to the Doxygen documentation and the source code for more examples.

```
rule addition, addend, expression;

addition    <<= addend << *(character("+") << addend);
addend      <<= range('0', '9') | expression;
expression  <<= character("(") << addition << character(")");

std::string input = "2+(3+4)";
match_range context(input.begin(), input.end());
match_range result;

if (expression.match(context, result))
  std::cout << "Matched:␣" << std::string(result.first, result.second);
```

Gergely Nagy
Budapest, June 19, 2014

Contents