# 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 in the form of 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 fundamentals 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). Actually several EBNF styles exist – this document uses the one defined by W3C[2].

The "Hello World" grammar that is generally 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 match (or a sequence of matches). 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 lies 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

One way to realize a recursive descent parser in a programming language is to create 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 `string::const_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 cannot 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 shows a simple 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.

```
1  bool expression(match_range &context) {
2    match_range local = context;
3
4    if  (
5      character(local, "(") && addition(local) && character(local, ")")
6    ) {
7      context.first = local.first;
8      return true;
9    }
10
11   return false;
12 }
```

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 evaluate to `false`.

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 expressions should not be constructed as they can lead to mispositioned iterators. Let's investigate the following code fragment:

```
1  if (
2    (rule1(local) && rule2(local)) || rule3(local)
3  ) {
4    ...
5  }
```

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 practically 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 cannot 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 a `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 cannot 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]
```

Actions 1-3 should place the results of the matches in temporary variables and action 4 should construct the complex data.

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 subsections the framework's structure and the basics of its operation are explained 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 concerning 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 defined 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 4:

```
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[3].

```
1  bool base_rule::match(match_range &context, match_range &
       the_match_range) {
2    match_range a_range;
3
4    if (test(context, a_range)) {
5      the_match_range = a_range;
6
7      if (the_semantic_action) {
8        std::string the_matched_substring(the_match_range.first,
             the_match_range.second);
9        the_semantic_action(the_matched_substring);
10     }
11
12     return true;
13   }
14   else return false;
15 }
```

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 the `character` function on page 3.

[3]This is not the actual code of the function – the parts concerning the automatic AST generation are discussed on page 13 and error message handling is discussed on page 20. This is true for the coming examples as well.

```
1  bool character::test(base_rule::match_range &context, match_range &
       the_match_range) {
2    if (context.first == context.second) return false;
3
4    base_rule::match_range local_context = context;
5
6    for (auto allowed_character: allowed_characters) {
7      if (*local_context.first == allowed_character) {
8        ++local_context.first;
9
10       the_match_range.first = context.first;
11       the_match_range.second = local_context.first;
12       context = local_context;
13       return true;
14     }
15   }
16
17   return false;
18 }
```

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, in fact this code shows the generic and recommended way of writing a rule.

Then it iterates over the set of allowed characters and if any of them matches the one at 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 extend 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[4].

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

[4]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_range;
7      context = local_context;
8
9      return true;
10   }
11
12   local_context = context;
13
14   if (second->match(local_context, second_range)) {
15     the_match_range = second_range;
16     context = local_context;
17
18     return true;
19   }
20
21   return false;
22 }
```

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 12 is unnecessary – it is there to make sure that no mistake is made even if the first 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 cannot take them by value, so the only choice is to take them as constant reference. This is 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`. The *virtual constructor idiom* comes handy in this situation, thus we need a `clone` function, a typical solution used e.g. 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 receives 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 returns 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 purpose of class `rule` is.

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[5].

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):

```
1  addition = addend ('+' addend)*
2  addend = [0-9] | expression
3  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 it 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.

[5]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.

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[6] attributed to David J. Wheeler[7] 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, is the solution for our problem here.

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 its value will: it will be overwritten with the address of the cloned rule assigned to the rule object.

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 rules. But how can an operator decide which rule is terminal and which is non-terminal? Well it doesn't have to: every operator will create rule objects and it simply solves the problem. This might seem wasteful as constructing a rule implies the dynamic allocation 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)) throw undefined_rule();
  return (*the_rule)->match(context, the_match_range);
}
```

Undefined rules throw an exception (undefined_rule – defined as a public inner class of rule), other rules return what the rule inside returns.

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

[6] en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering
[7] en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

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 also 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 operators are not members of `rule`, they create `rule` objects and they really belong to the class. They reside in the same namespace and are found by *Koenig-lookup*[8].

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 performs the same operation as on ints 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, operators were used quite freely and, quite often, an operator was assigned 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 one another then it means that the first has to be followed by the second. This is something that cannot 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 |

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

[8] http://en.wikipedia.org/wiki/Argument-dependent_name_lookup

## The built-in rules (terminal symbols)

There is a great number of built-in rules defining 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 the 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 example

Having duscussed the most important features of the framework, a fully functional example is considered in this subsection. 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.

```
1  rule addition, addend, expression;
2
3  addition    <<= addend << *(character("+") << addend);
4  addend      <<= range('0', '9') | expression;
5  expression  <<= character("(") << addition << character(")");
6
7  std::string input = "2+(3+4)";
8  base_rule::match_range context(input.cbegin(), input.cend());
9  base_rule::match_range result;
10
11 if (addition.match(context, result))
12   std::cout << "Matched:␣" << std::string(result.first, result.second);
13 else
14   std::cout << "Didn't␣match";
```

It should be noted that simply relying on the return value of `match` can be misleading. If a grammar is a series of recurring elements at the top level (e.g. a C code is basically a series of functions) then it might return true for a text that contain errors.

Let us assume that a very simplistic approach to the grammar of the C language contains the following starting rule:

```
program <<= *-function;
```

If there is a syntax error in the third function of a code, the `function` rule will not match that function. This will stop the parsing, but as the first two functions matched, the repetition operator will return true and so will the `program` rule as it has no way to know that it is the starting rule of the grammar.

This problem can be solved by improving the test condition of matching:

```
1  if (addition.match(context, result) && context.first == context.second)
2    std::cout << "Matched:␣" << std::string(result.first, result.second);
3  else
4    std::cout << "Didn't␣match";
```

The first line contains an extra condition: the two fields of the `context` has to be equal. The second field always points to the end of the text while the first starts from the beginning and is increased continuously as the text is parsed. We have a full match when the first member reaches the last: the text is parsed to the last character.

Another way to check for a full match would be to compare the bounds of the parsed text with `result` – they have to be equal.

## Automatic AST generation

The processing of texts based on a simple grammar can be done on the fly, i.e. appropriate semantic actions (defined in e.g. lambdas) can be added to the rules. Examples showing how this can be done are shipped with the SyntX framework.

When a grammar becomes complex this task becomes very exhausting especially when we consider the fact that a subrule (a rule that is a part of another rule) matching some part of the text doesn't mean that the rule that contains the subrule will match as well. So an action that is performed when a certain rule matches might have to be undone when it turns out that parsing went the wrong way. This problem was discussed in the subsection on actions starting on page 4.

Another frequent situation is that a text has to be parsed several times to gather all the information needed to fully understand and process the information represented by the text. These cases cannot be handled with semantic actions – only if what the actions do is build a representation of the text that can later be traversed the needed times.

This is such a common task that a parsing framework should directly support it. The data structure that is typically built is called *Abstract Syntax Tree (AST)*, which is basically a tree representation of the text's structure.

SyntX can build such a tree automatically. Naturally the framework does not know the internal build-up of the text, so it rather works with the grammatical structure. The tree consists of leaf nodes containing subtrings matched by rules like `character`, `identifier` or `keyword` defining terminal symbols and nodes that have children matched by the rules that are given using operators (`<<`, `|`, `*`, etc.).

For example if our rule is the following:

```
addition <<= integer << +( character("+") << integer );
```

and our input is:

```
2 + 3 + 4
```

then the tree generated is the following:

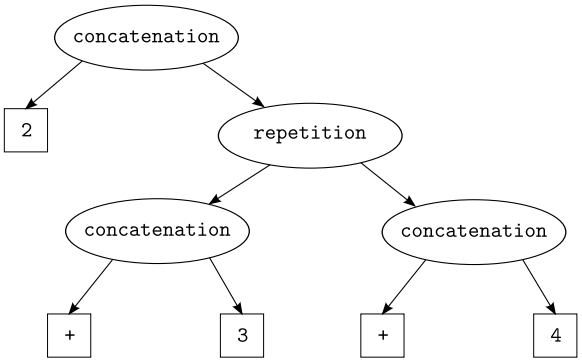


*Figure 1:* The Abstract Syntax Tree of the expression: $2 + 3 + 4$

Naturally, the node representing a `repetition` may have an unlimited number of children while `concatenation` always has two.

The data structure representing a node of the AST can be found inside the `base_rule` class:

```
1  struct node {
2    enum class type {value, alternation, concatenation, option, repetition,
         repetition_or_epsilon, named_rule};
3
4    type the_type;
5    std::string the_value;
6    std::vector<std::shared_ptr<node>> children;
7
8    node(std::string a_value): the_type(type::value),the_value(a_value) {}
9
10   node(type a_type): the_type(a_type) {}
11 };
```

The type of the node can determined using `the_type` member which has one of the values of the `type` enumeration. The nodes could be represented by a class hierarchy with one class assigned to each operator, one to the named rules and one that would serve as a leaf node. This idea was rejected as the typical use case of tree traversal will branch depending on the node type so in case of a class hierarchy a complex `if` structure would appear with `dynamic_casts` which has a larger time cost then switching based on an enumeration. In order to make use of polymorphism in case of a class hierarchy, the visitor pattern[9] could possibly be used but that would complicate an already complex, recursive tree traversal algorithm.

The subtrings matched by the character-level rules are stored in `the_value` member.

[9] en.wikipedia.org/wiki/Visitor_pattern

A node might have no children at all, or it might have an unlimited number of offsprings. They are stored in the `children` container. Nodes are created dynamically and stored using `shared_ptrs`.

There is a type called `named_rule` to support a feature that enables users to add extra nodes to an AST corresponding to user defined non-terminals. More about this can be found later.

If the AST is needed, the `base_rule::set_build_ast(true);` call has to be issued before parsing (line 2) and the reference of the root of the AST (line 13) to be built has to be given to the `match` method call of the highest level rule (line 15) as seen below:

```
1  int main() {
2    base_rule::set_build_ast(true);
3
4    rule addition, addend, expression;
5
6    addition <<= -addend << *(-character("+") << -addend);
7    addend <<= -range('0', '9') | -expression;
8    expression <<= -character("(") << -addition << -character(")");
9
10   std::string input = "2␣+␣3␣+␣4\n";
11   base_rule::match_range context(input.cbegin(), input.cend());
12   base_rule::match_range result;
13   std::shared_ptr<base_rule::node> root;
14
15   if (addition.match(context, result, root)) {
16     std::cout << "Matched:␣" << std::string(result.first, result.second);
17     parse_tree(root);
18   } else {
19     std::cout << "Didn't␣match";
20   }
21 }
```

This example also shows how the whitespaces can be consumed using the operators - or ∼.

The resulting AST can be traversed by writing a simple recursive function such as the one in the following example. Line 17 in the code above shows how it should be called.

```
1  void parse_tree(std::shared_ptr<base_rule::node> const &node, size_t depth=0) {
2    if (node) {
3      for (size_t i = 0; i < depth; ++i) std::cout << "␣";
4
5      switch (node->the_type) {
6        case base_rule::node::type::value:
7          std::cout << node->the_value << std::endl;
8        break;
9
10       case base_rule::node::type::alternation:
11         std::cout << "alternation" << std::endl;
12       break;
13
14       case base_rule::node::type::concatenation:
15         std::cout << "concatenation" << std::endl;
16       break;
17
18       case base_rule::node::type::option:
19         std::cout << "option" << std::endl;
20       break;
21
22       case base_rule::node::type::repetition:
23         std::cout << "repetition" << std::endl;
24       break;
25
26       case base_rule::node::type::repetition_or_epsilon:
27         std::cout << "repetition_or_epsilon" << std::endl;
28       break;
29
30       case base_rule::node::type::named_rule:
31         std::cout << "named_rule:␣" << node->the_value << std::endl;
32       break;
33     }
34
35     for (auto &a_node: node->children) parse_tree(a_node, depth + 1);
36   }
37 }
```

The named_rule type is special and is discussed later.

Earlier we have discussed the code of match and test methods. Those listings were not identical to the actual code of these methods found in the framework as the parts concering the building of the AST were omitted.

Actually only a little needs to be added to those methods. Both of them receive an extra argument: the reference of a pointer to the subtree's root that has to be built by the rule. This pointer is overwritten with the address of the newly created node and the tree is created by the successive recursive function calls as the parser digs into the text. The default value of this argument is the reference of a static node – this is needed as a reference has to be initialized, but it would be cumbersome for the user to give a root reference even if an AST is not needed[10].

The method match simply delegates the given pointer to the test function.

The test method of character-level rules and high-level rules are different so an example for both are discussed below.

First, let's dwelve into the code of character class's test method. This is a class that operates at the character-level so it has to produce a leaf node in the AST, one that has no children and contains the result of a match (a string containing a single character in this case).

The only difference to the code seen earlier apart from the extra argument is that it creates a new node in line 14 and saves its address in the ast_root variable. This is only

[10]The match method could have been overloaded, but the code of the two function would have been almost identical, so that was considered as bad style.

done if an AST is needed, otherwise time and memory is not wasted on it.

The `get_build_ast()` call reads the value of the static `build_ast` variable.

The value of the node is set by creating a `std::string` object using the `const_iterators` of `the_match_range`. No children are set which leaves the `children` container empty showing that this node is a leaf.

```
1  bool character::test(base_rule::match_range &context, match_range &the_match_range,
       std::shared_ptr<base_rule::node> &ast_root) {
2    if (context.first == context.second) return false;
3
4    base_rule::match_range local_context = context;
5
6    for (auto allowed_character: allowed_characters) {
7      if (*local_context.first == allowed_character) {
8        ++local_context.first;
9
10       the_match_range.first = context.first;
11       the_match_range.second = local_context.first;
12       context = local_context;
13
14       if (get_build_ast()) ast_root = std::make_shared<base_rule::node>(std::string(
             the_match_range.first, the_match_range.second));
15
16       return true;
17     }
18   }
19
20   return false;
21 }
```

The other type of `test` method can be found in the high-level rules that are instantiated using operators. The code below belongs to the `concatenation` rule.

```
1  bool concatenation::test(base_rule::match_range &context, base_rule::match_range &
       the_match_range, std::shared_ptr<base_rule::node> &ast_root) {
2    base_rule::match_range first_range, second_range;
3    base_rule::match_range local_context = context;
4    std::shared_ptr<base_rule::node> first_child, second_child;
5
6    if (first->match(local_context, first_range, first_child) && second->match(
         local_context, second_range, second_child)) {
7      the_match_range.first = first_range.first;
8      the_match_range.second = second_range.second;
9      context = local_context;
10
11     if (get_build_ast()) {
12       ast_root = std::make_shared<base_rule::node>(base_rule::node::type::
             concatenation);
13       ast_root->children.push_back(first_child);
14       ast_root->children.push_back(second_child);
15     }
16
17     return true;
18   }
19   else return false;
20 }
```

This rule creates a node that always has two children: one for each of the two rules following eachother. The rule calls two rules that have to match for this one to be successful. Those rules can be very complex and create a large subtree each.

The roots of the two subtrees are created locally in line 4 and fed to the two subrules in line 6. If the match is successful and an AST is needed then a new node is created

in line 12. This one is not constructed with the string containing the match range as it could be a complex match of several rules and would be of little use to the user. This node will tell the traversing algorithm the type of operator that matched at the current position instead. It is `type::concatenation` in this case.

The children of this node are the two nodes (or subtrees) that are created by the `match` calls of the inner rules of `concatenation` in line 6. They are added to the node in lines 13 and 14.

Finally the code of `repetition::test` is also listed here to show how an unlimited number of children can be handled.

```
1  bool repetition::test(base_rule::match_range &context, base_rule::match_range &
       the_match_range, std::shared_ptr<base_rule::node> &ast_root) {
2    base_rule::match_range range;
3    base_rule::match_range local_context = context;
4    std::shared_ptr<base_rule::node> child;
5
6    if (repeated_rule->match(local_context, range, child)) {
7      the_match_range = range;
8
9      if (get_build_ast()) {
10       ast_root=std::make_shared<base_rule::node>(base_rule::node::type::repetition);
11       ast_root->children.push_back(child);
12     }
13
14     while (repeated_rule->match(local_context, range, child)) {
15       the_match_range.second = range.second;
16
17       if (get_build_ast()) {
18         ast_root->children.push_back(child);
19       }
20     }
21
22     context = local_context;
23
24     return true;
25   }
26
27   return false;
28 }
```

It is important to note that the node is created only once (when the first child is added). When the `while` loop is entered in line 14, the node already exists, so a child can be added to it in line 18. This situation is handled similarly in `repetition_or_epsilon::test`, the code of which is not listed here. The reader is encouraged to have a look at it though.

There is one additional feature of SyntX's AST-building facilities that's worth mentioning. At the beginning of this subsection it was stated that ASTs represent the grammatical structure of the text and not the internal logic of the text. Fortunately this is not entirely true: the users of the framework can add extra nodes to the tree representing their rules. In order to do that, a rule created by the user has to be supplied with a name at construction.

So let's change our little example as follows.

```
1  int main() {
2    base_rule::set_build_ast(true);
3
4    rule addition, addend("addend"), expression;
5
6    addition <<= -addend << *(-character("+") << -addend);
7    addend <<= -range('0', '9') | -expression;
8    expression <<= -character("(") << -addition << -character(")");
9
10   std::string input = "2 + 3 + 4\n";
11   base_rule::match_range context(input.cbegin(), input.cend());
12   base_rule::match_range result;
13   std::shared_ptr<base_rule::node> root;
14
15   if (addition.match(context, result, root)) {
16     std::cout << "Matched: " << std::string(result.first, result.second);
17     parse_tree(root);
18   } else {
19     std::cout << "Didn't match";
20   }
21 }
```

The only difference to the previously seen version is that the rule `addend` was given a name in line 4.

The AST generated will now look like this:

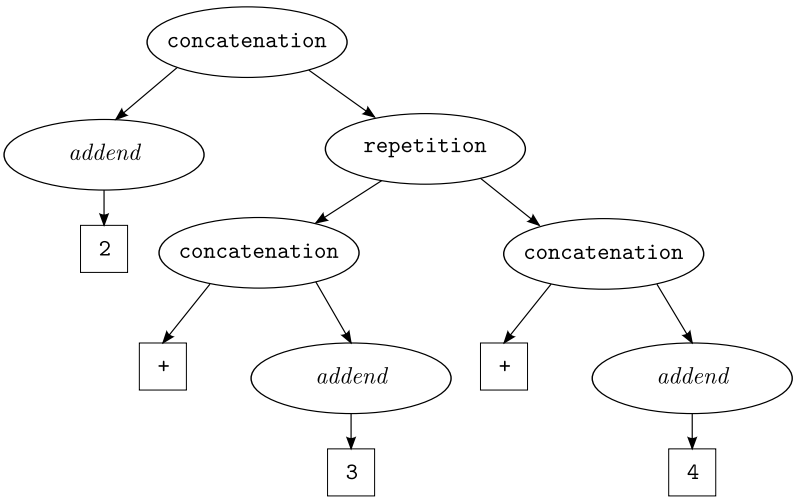


***Figure 2:*** The Abstract Syntax Tree of an expression with a named rule (addend)

This might not seem very useful in this rather simple example but it becomes very handy when a huge AST based on a complex grammar has to be processed.

All it took to achieve this feature was to give the class `rule` a new member (`rule_name`), a new constructor that actually sets this name and the `test` function had to be altered a bit.

```
1  bool rule::test(match_range &context, match_range &the_match_range, std::
       shared_ptr<base_rule::node> &ast_root) {
2    if (!(*the_rule)) throw undefined_rule();
3    if (!get_build_ast() || rule_name == "")
4      return (*the_rule)->match(context, the_match_range, ast_root);
5    else {
6      std::shared_ptr<base_rule::node> child;
7      if ((*the_rule)->match(context, the_match_range, child)) {
8        if (child) {
9          ast_root = std::make_shared<base_rule::node>(base_rule::node::type::
               named_rule);
10         ast_root->the_value = rule_name;
11         ast_root->children.push_back(child);
12       }
13       return true;
14     }
15     else return false;
16   }
17 }
```

If the rule doesn't have a name (i.e. the `rule_name` member is an empty string) or the AST was not requested by the user then the `test` method merely delegates the context, the match range and root of the AST to the inner `base_rule` that is wrapped by it. This can be seen in lines 3 and 4.

On the other hand, if the rule has a name and we are building an AST, an extra node has to be inserted. In order to do this, a child node is created and fed to the inner rule (lines 6 and 7) and then a new node is created in line 9. It is of type `named_rule` and its name is stored in `the_value` member of the node (line 10).

The child that is created by the inner rule is set as the child of the named rule node in line 11.

All this happens only if a child is indeed created by the inner rule. If the inner rule is an `option` or a `repetition_or_epsilon` then it may match successfully an empty string. In this case these rules will add null pointer to the tree to show that there is where the tree ends in the current direction. This case can be easily checked by converting the child pointer to a boolean value and checking whether it is true – this happens in line 8.

If this check was omitted, a `named_rule` node would be inserted in the tree without children which could mislead the processing algorithm or simply make it necessary to add an extra conditional statement testing whether a named rule is really there. This would be very cumbersome.

## Error message handling

When the text being parsed is fully conforming to the grammar then it is very easy and comfortable to work with using the library as described above. Probems arise when there are syntactic errors as all the `match` method tells us about the success of the matching is a boolean value. Without further help the entire text needs to be parsed by the user of the framework in case of errors which can be a very hard task and thus parsing libraries are expected to be of a little more help.

Finding the position of the syntactic error is not so easy though. The problem is that there are always rules that fail to match during parsing even when the text is perfect. This is because alternations can be defined. If a grammar accepts "apple" or "pear" at a point, then one of these fruits will always fail. So simply reporting a failing rule is not a correct solution to this problem.

Another difficulty is rooted in the nature of the parsing algorithm: it is recursive so it may be at a position very deep down the call stack when it finds a misspelling. Reporting an error from down there by passing the message up the stack is very cumbersome.

Yet another aspect of the problem is to determine the type of rules that should report. If a high-level rule (a so called *non-terminal*) rule reported an error, it could position the place of the flaw long before it actually occurs. Just think of a concatenation of several symbols failing at the last one. As the entire concatenation fails in this case, it would report the position of the error for its first element.

It is very hard to find the ultimate solution to this problem. Even the best compilers give ambiguous error messages sometimes. Heuristic methods are used and they are constantly improved.

The SyntX library takes a very simplistic but generally fairly correct approach. The `base_rule` class has a static member called `failure_log` that contains tuples consisting of two members: an iterator pointing to a position where the parsing failed and a string that says what was expected at that point.

Terminal rules can report errors in this container. Whenever a rule that operates at the character level fails, it adds an entry to it telling its current position and describing what it was looking for. The `insert_failure_entry` method is used to do this which is a virtual method in `base_rule`. It is not pure virtual as every rule inherits it, but only the terminals will act, the rest leave it empty.

This method is called by `base_rule::match` as it is that method that knows whether the matching of a rule was successful or not and also the current position of the parsing. The latter has to be passed to `insert_failure_entry`. Thus the `match` method amended with error message handling looks like this:

```
1  bool base_rule::match(match_range &context, match_range &the_match_range, std::
       shared_ptr<node> &ast_root) {
2    match_range a_range;
3
4    if (test(context, a_range, ast_root)) {
5      the_match_range = a_range;
6
7      if (the_semantic_action) {
8        std::string the_matched_substring(the_match_range.first, the_match_range.
             second);
9        the_semantic_action(the_matched_substring);
10     }
11
12     return true;
13   }
14   else {
15     insert_failure_entry(context.first);
16     return false;
17   }
18 }
```

The only difference to the previously seen version of the method is in line 15, where `insert_failure_entry` is called with the first field of the `context` pair containing the current parsing position.

Next, let us investigate an example of the logging method:

```
1  void character::insert_failure_entry(std::string::const_iterator const &position
       ) const {
2    std::stringstream stream;
3    stream << "a␣character␣from␣the␣set:␣{" << allowed_characters << "}";
4    failure_log.insert(std::make_tuple(position, stream.str()));
5  }
```

It simply passes the received position to the failure log together with a string that describes what the rule is looking for. This string can be a part of the following sentence: "The parser was looking for ...".

The `failure_log` will contain elements after almost every parsing run regardless of its success. As mentioned above, every alternation will result in failures that appear in the log. Here comes the heuristics: the log entry that is most likely to be the root cause of a failure has to be found when the parsing fails.

The SyntX library chooses the entry that is farthest from the beginning of the text. In most cases the processed text is mostly correct, so it is very likely that the parsing route that reaches deepest into it is the interpretation that the author intended.

There are two ways to get hold of the information regarding errors in SyntX: `base_rule::get_failure_cause` returns the element of `failure_log` with the largest string iterator and lets the user create an error message using the information in the entry, `base_rule::get_error_message` returns a string that contains the text in the `failure_log` entry plus the part of the text that contained the error, the position of which is shown with a ^ symbol:

```
An error occured here (line 11):
      +apple;
       ^
The parser was expecting the substring: ++
```

*Gergely Nagy*
*Budapest, October 27, 2014*

# Contents



*Please consider the environment before printing this document.*