Programcode Interpretation via Regular Expressions

Authors: Dávid Magyar
Sándor Szénási

Fundamentals

Programcode

Regular Expressions

Let’s start with the fundamentals;
we must understand what “programcode” and “regular expression” actually is.

Fundamentals

Programcode

Regular Expressions

- **ordinary text file** (.txt)
  - **sequence of characters**

```
int x = function();
```

Programcode is just ordinary text, which is a sequence of characters.

Fundamentals

Programcode

Regular Expressions

- ordinary text file (.txt)
  - sequence of characters
- **encloses <u>two information</u>:**
  - syntactical information
  - semantical information

```
int x = function();
```

It holds two kind of information for us: syntactical and semantical.
These two are not mutually exclusive, but two different understanding of what the text means.

Fundamentals

Programcode

Regular Expressions

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - **<u>syntactical</u> information**
  - semantical information

Syntactical information is basically grammar itself. Based on what we expect, we can categorize given parts of the text and associate them with a role. Such text, which does not match our expected grammar cannot be interpreted.

It is the actual meaning and intention behind the written words, its logical structure.
This is what a programmer thinks, while writing the lines.
What is syntactically correct, might not be semantically.

Programcode

Regular Expressions

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - **<u>semantical</u> information**

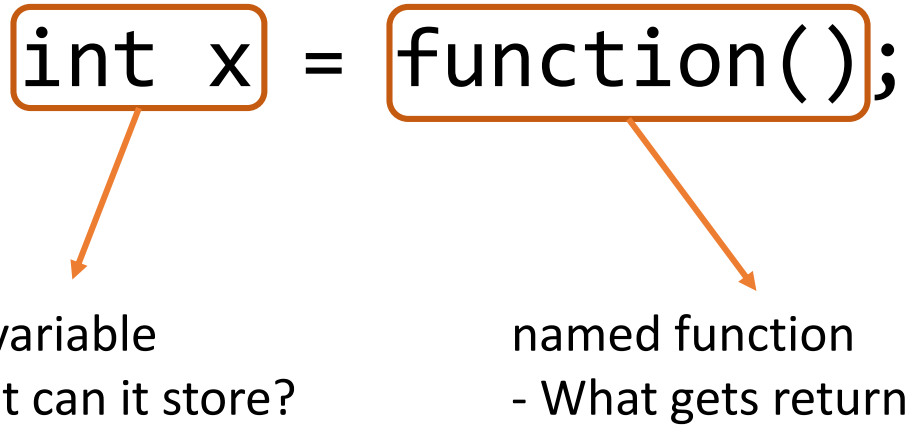
```
int x = function();
```

local variable
- What can it store?

named function
- What gets returned?

Let’s see the example on the right side; the statement is syntactically correct, we invoke a function and store its returned value in a newly declared variable “ex”.
But semantically it might be wrong; can we declare such variable in this scope? Can it store the value returned by the function? Does the function even have a return value?
Without more context we don’t know these semantics.

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - **<u>semantical</u> information**

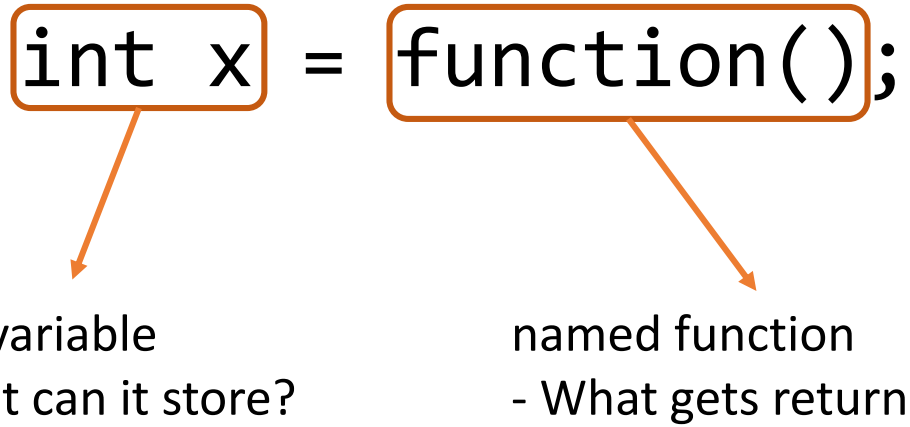


This interpreter and presented algorithm does not aim to decipher these deeper interpretation of the code, but is limited to obtain the syntactical information from the code text.

Programcode

Regular Expressions

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - **<u>semantical</u> information**



The presented algorithm aims to interpret syntactical information.

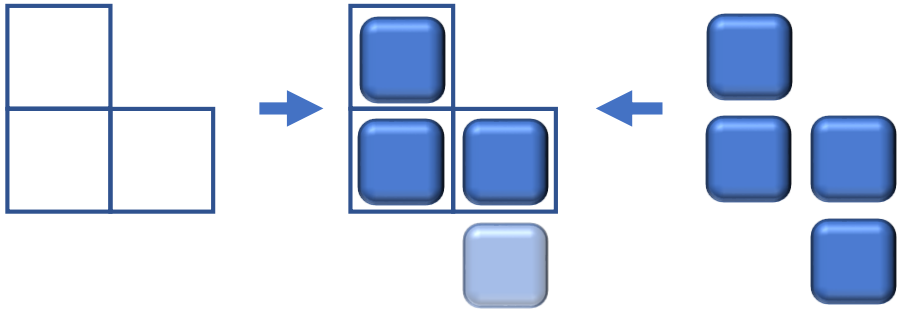
Let’s see what we will use to accomplish this: regular expressions. What are they? Regular expressions are essentially pattern, which can match an arrangement of elements.

Programcode

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

Regular Expressions

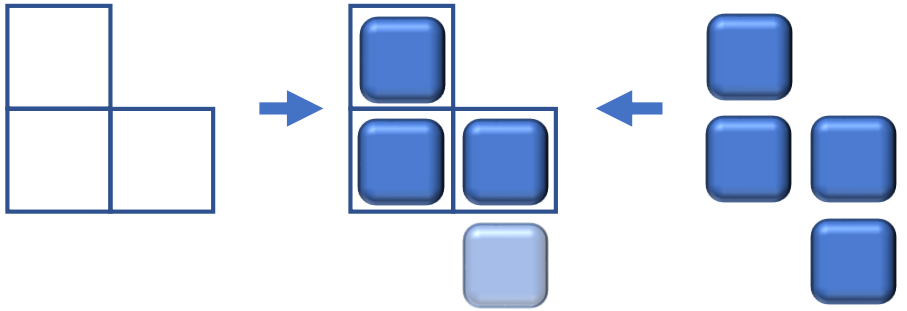
- **<u>patterns</u>, which can match**



As an example, we have a pattern here represented by the squares in L shape on the left.
It represents, that it matches elements, which are in such shape.
On the right, we have the input elements.
In the middle we see how this pattern can have a match on the input elements. It does not match all of the input, but only a part of it.

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

- **<u>patterns</u>, which can match**

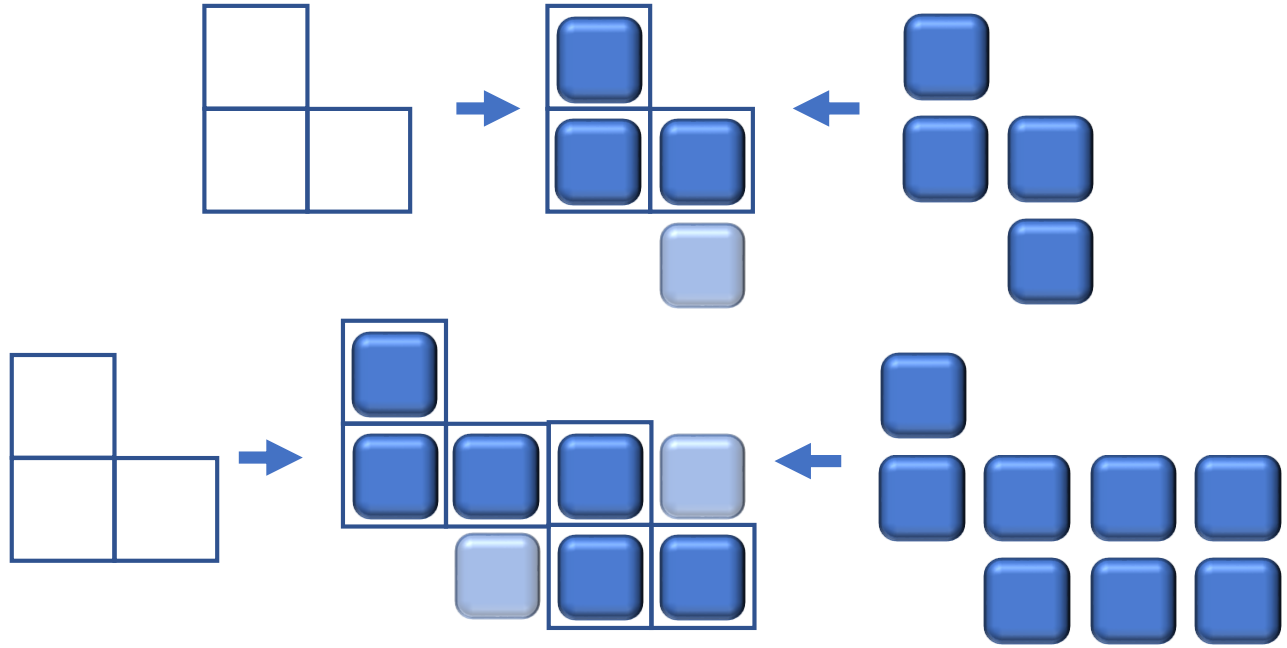


Fundamentals

If the input is longer or larger, the pattern might match multiple times and in multiple possible ways; usually we consider the left-most matches, so, the example pattern starts on the top-left corner and matches there first, then continues towards right and then towards down to find further matches.

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

- **<u>patterns</u>, which can match**

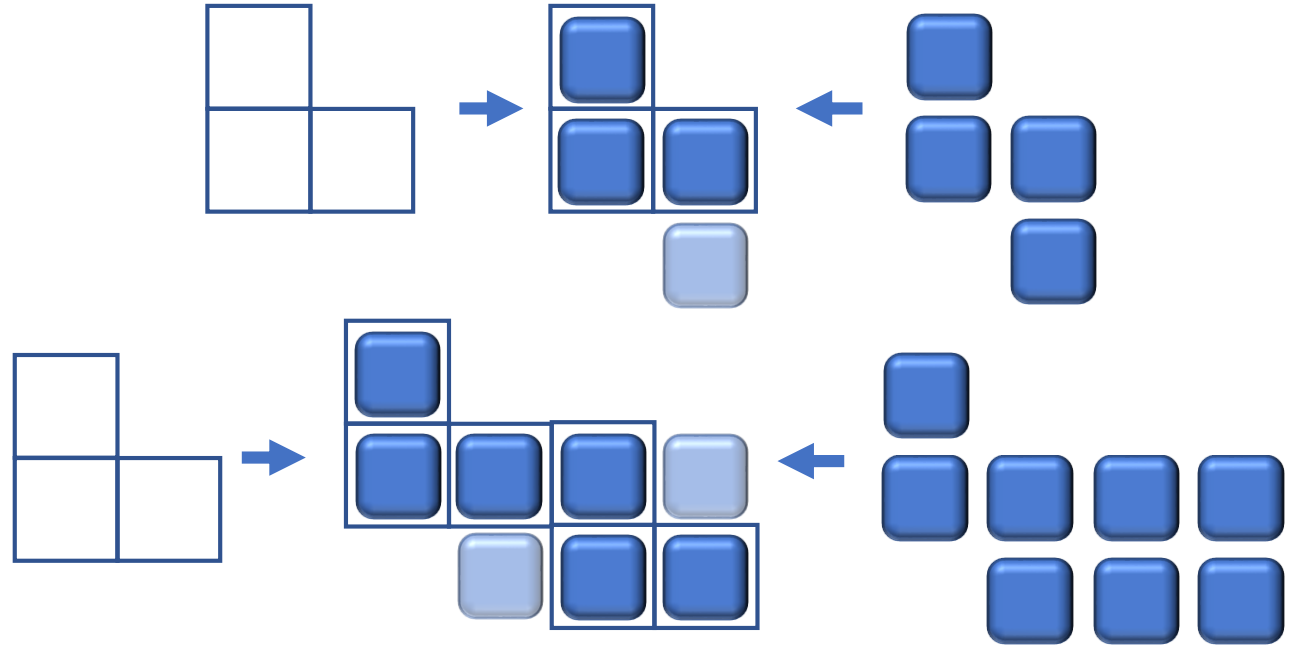


Fundamentals

By definition, regular expressions are a bit more complex than that
- the pattern can be given though a system of „production rules”
- we won’t dig any deeper into its theory here, but look at its actual usage
- which somewhat differs from theoretical “by-definition” regular expressions anyways

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

- **<u>patterns</u>, which can match**



Fundamentals

Regular expressions most often are used primarily for text, so often, that some programmers might think it is applicable only for text.

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

- patterns, which can match
- while programming we usually **apply it only <u>to text</u>**

Fundamentals

Programcode

Regular Expressions

Here is an example about a typical “a+” pattern can match all sequences of “a” characters in an input sequence. It finds 2 occurrences, as shown.

- sequence of characters

- encloses two information:
  - syntactical information
  - semantical information

- while programming we usually **apply it only <u>to text</u>**

Fundamentals

This “a+” is a production rule, which theoretically can generate all sequences it matches; being “a”, then “aa”, “aaa” and so on.
The “a” part of it represents an actual “a” character, while the “+” means one or more of it.

- sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

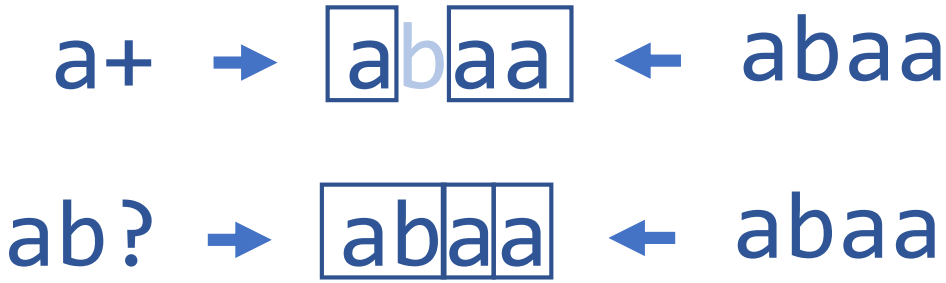
- while programming we usually **apply it only <u>to text</u>**

Fundamentals

Similarly, “ab?” is another production system, which describes an “a” character optionally followed by a “b” character, so “a” and “ab” can be matched.
It finds 3 occurrences, as shown.

- sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

- while programming we usually **apply it only <u>to text</u>**



Fundamentals

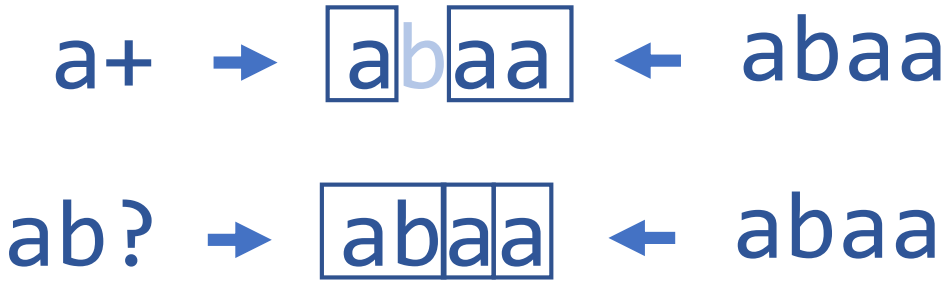
Programcode

- sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

Regular Expressions

We saw on the previous slides, that we could apply a pattern to two-dimensional input too, so we can think with regular expressions in a larger context as well.

- while programming we usually **apply it only <u>to text</u>**



Fundamentals

Program code

- ordinary text file (.txt)
  - sequence of characters
- encloses two information:
  - syntactical information
  - semantical information

Regular Expressions

- patterns, which can match
- while programming we usually apply it only to text

So, these two term being defined, I use such regular expressions in my algorithm to obtain the syntactical information of the input program code.

Process

```
class MyClass {
    static int getAnswer() {
      int result = 42;
      return result;
    }
}
```

Source Code

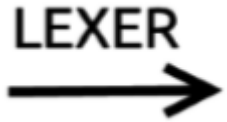
Let’s see the algorithm itself.

The code is colored intentionally – we programmers typically see it this way, but the color is added by the developer environment while editing, it is not part of the code itself.

Process

```
class MyClass {
    static int getAnswer() {
      int result = 42;
      return result;
    }
}
```

Source Code



Tokens

- “class” keyword
- “MyClass” name
- “{” block begin
- “static” keyword
- “int” keyword
- “getAnswer” name
- “(” expr. begin
- “) ” expr. end
- “{” block begin
- “int” keyword
- “result” name
- “=” equal symbol
- “42” number
- “;” statement end
- “}” block end

First, a component typically called “lexer” slices the input sequence into so called “tokens”.

These are recognized words associated with a role, like a keyword or a name.
Such lexer is usually not too difficult to implement, so it’s the easier part of interpreters.

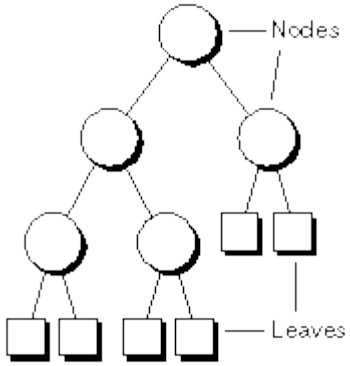
Process

```
class MyClass {
```

Tokens

- “class” keyword
- “MyClass” name
- “{” block begin
- “static” keyword
- “int” keyword
- “result” name
- “=” equal symbol
- “42” number
- “;” statement end
- “return” keyword
- “result” name
- “;” statement end
- “}” block end
- “}” block end

The more difficult part is to write the “parser” part, which uses the token sequence and builds an AST – an abstract syntax tree – of it.
The AST is tree structure of data, which obviously holds even more information about the code than tokens.
Since it’s hierarchical, it describes which part belongs to which larger concept described by the code.



@Class: MyClass
- @Function: getAnswer
  - @Statement: int result = 42;
  - @Statement: return result;

PARSER

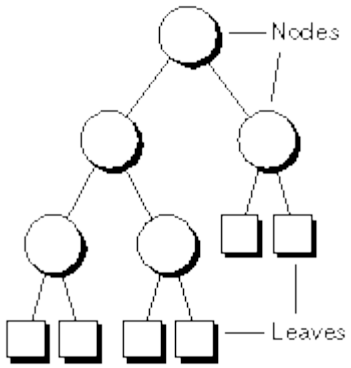
AST – Abstract Syntax Tree

Process

Tokens

- “class” keyword
- “MyClass” name
- “{” block begin
- “int” keyword
- “result” name
- “=” equal symbol
- “42” number
- “;” statement end
- “return” keyword
- “result” name
- “;” statement end
- “}” block end
- “}” block end

PARSER



@Class: MyClass

@Function: getAnswer

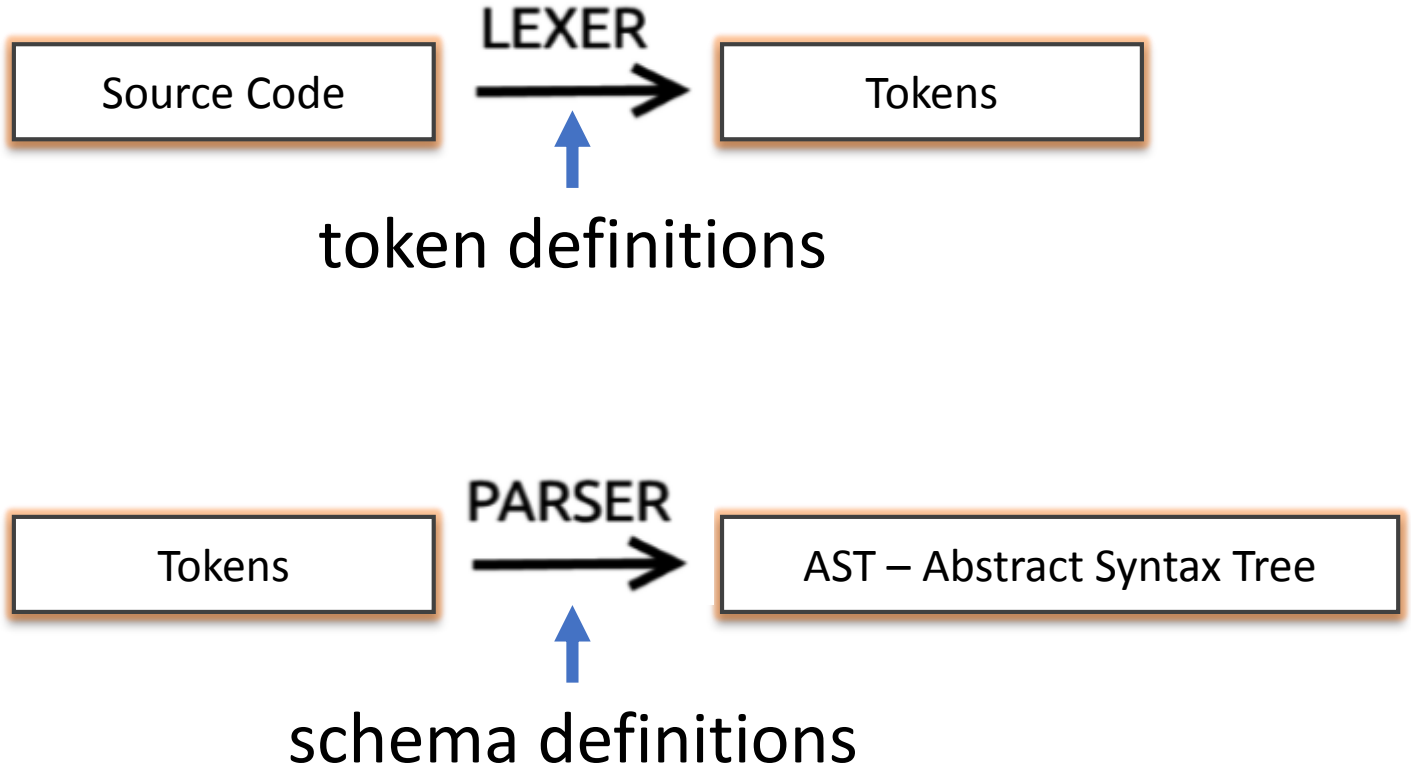
@Statement: int result = 42;

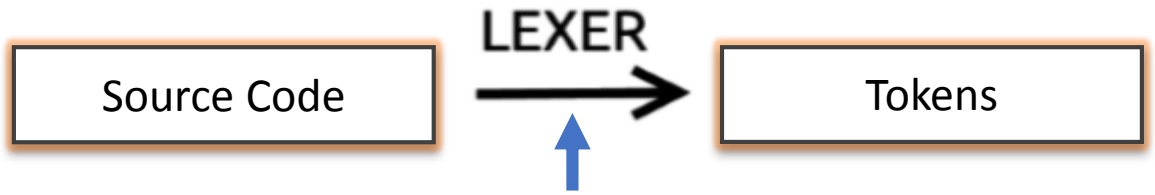
@Statement: return result;

AST – Abstract Syntax Tree

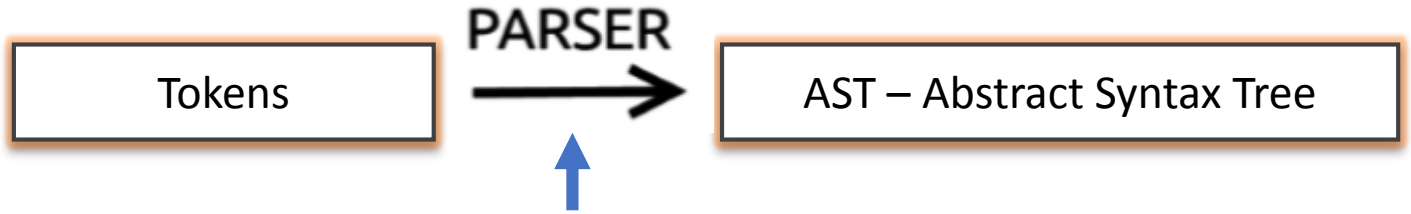
For example, if we observe a “return” word, the lexer provides the information, that it is a “keyword”, but nothing more.
The parser though the AST tells us however, that it’s part of a return statement, which is part of a function definition, which is part of a given class, and we know even know the names of these containing elements.
To compile the program code to an executable processor code, the compiler needs all this information.

Definitions





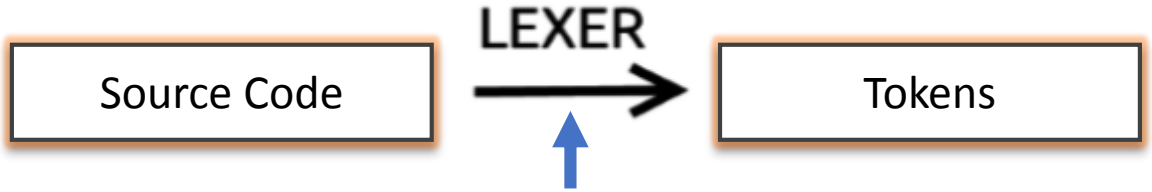
token definitions



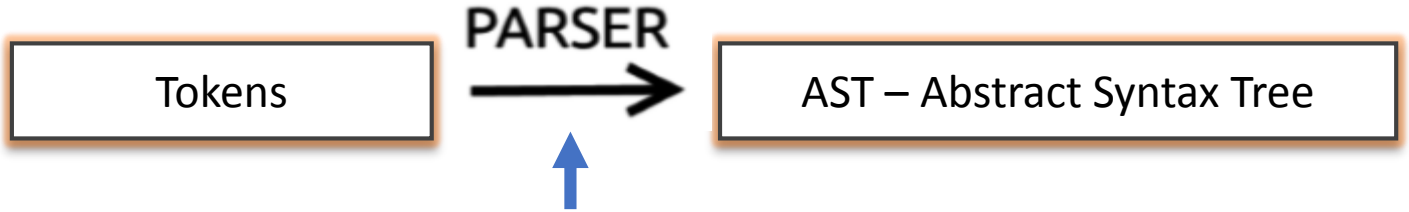
schema definitions

So, reviewing these parts;

- the lexer gets the source code as its input and produces a sequence of tokens from it
- then the parser gets this token sequence and builds an abstract syntax tree of it



token definitions



schema definitions

In this algorithm the lexer works based on a system of token definitions
and the parser works based on a system of schema definitions.
These systems are designed to be similar and easy to understand.
They describe what tokens the lexer looks for, in what order,
and what concepts those tokens describe in what various patterns.
Such pattern, which describe a concept in the code is called - in this algorithm - a schema.

Definitions

All definitions have:

- a **name**:
  - What does it identify?

- a **matcher**:
  - How does it find it?

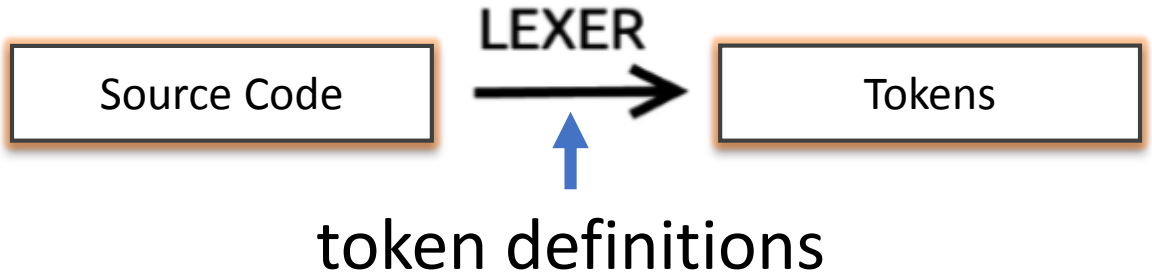
All definitions have:

- a **name**:
  - What does it identify?

- a **matcher**:
  - How does it find it?

Both token and schema definitions have a name and a matcher.
The name is just an identifier, so the matches can be labeled with it.
The matcher encloses the pattern itself, being capable of making matches on the input sequence.
(The input sequence being text characters for token definitions and token* sequence for schema definitions -- *more on this later)

Definitions

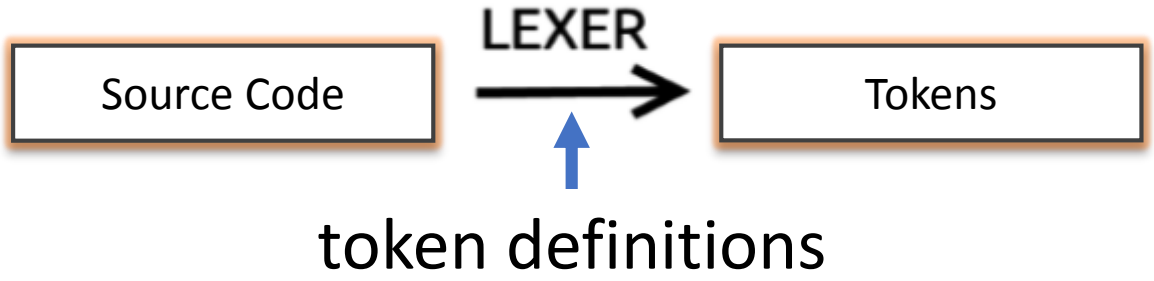


name matcher

E.g.:

```
[ID]: regex [\w]+
[NUMBER]: regex ([\d]*\.)?[\d]+
[CLASS]: word class
[EQUAL]: infix =
```

Definitions



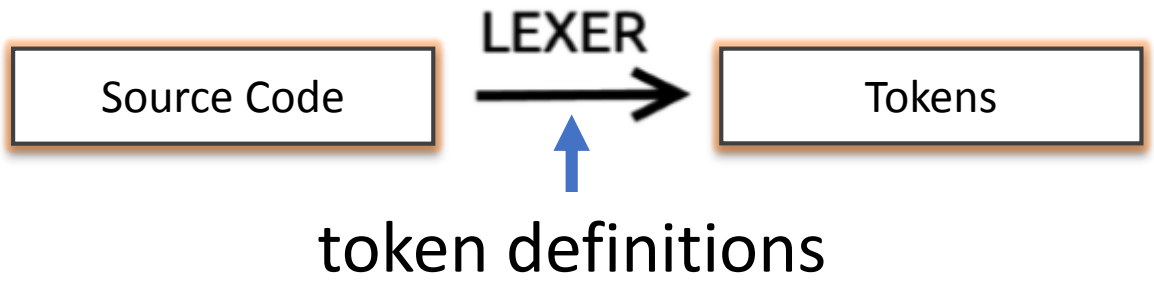
name matcher

E.g.:

```
[ID]: regex [\w]+
[NUMBER]: regex ([\d]*\.)?[\d]+
[CLASS]: word class
[EQUAL]: infix =
```

Let's see how token definitions looks like.
In blocky brackets we must give our token definition a unique name among other token definitions.
Following a colon, the matcher is described.

Definitions

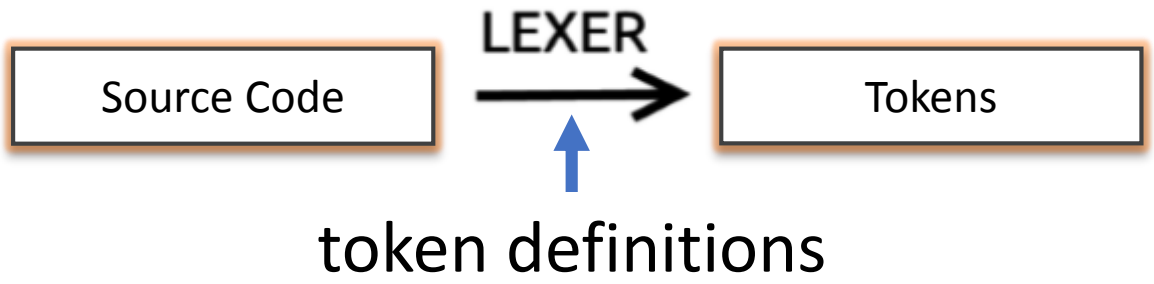


These example definitions do match certain sequence of characters; for example, those on the right side.

E.g.:

```
[ID]: regex [\w]+                    → “MyClass”
[NUMBER]: regex ([\d]*\.)?[\d]+      → “1.618”
[CLASS]: word class                  → “class”
[EQUAL]: infix =                     → “=”
```

Definitions



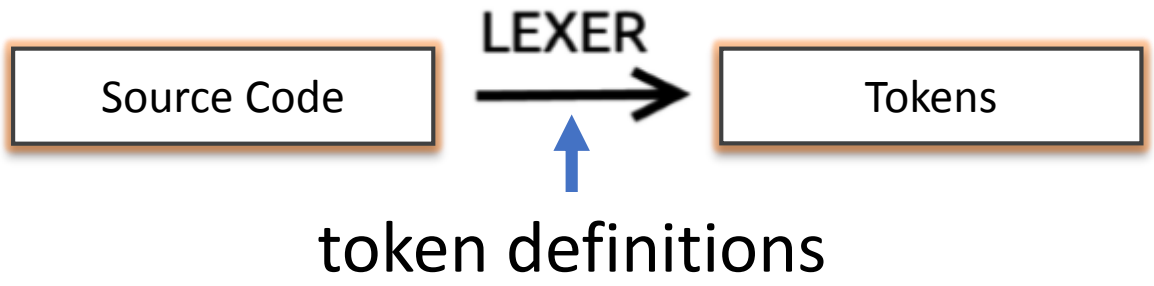
An obvious problem is to decide what happens if two token definitions can match the same word.
Which will recognize a "`class`" word in an input sequence?

E.g.:

```
[ID]: regex [\w]+
[NUMBER]: regex ([\d]*\.)?[\d]+
[CLASS]: word class
[EQUAL]: infix =
```

→ "class"
→ "1.618"
→ "class"
→ "="

Definitions

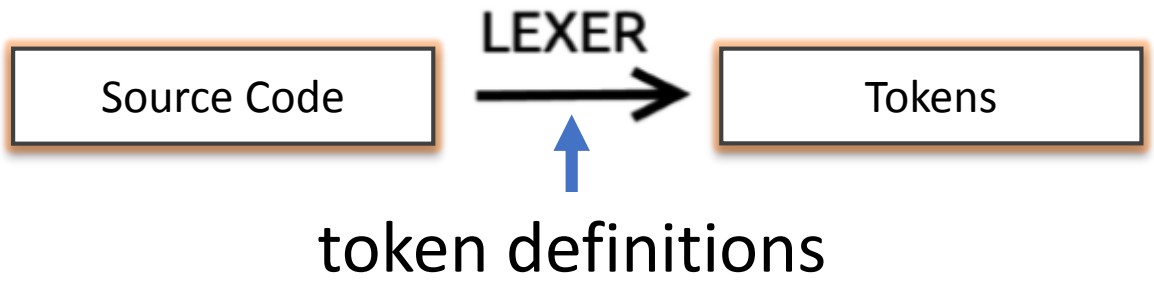


This can be solved by adding a hard section separator between the two definitions. This guarantees, that the definition defined upper will match it in all cases.

E.g.:

```
[NUMBER]: regex ([\d]*\.)?[\d]+     → “1.618”
[CLASS]: word class                 → “class”
[EQUAL]: infix =                    → “=”
=====
[ID]: regex [\w]+                   → “class”
```

Definitions

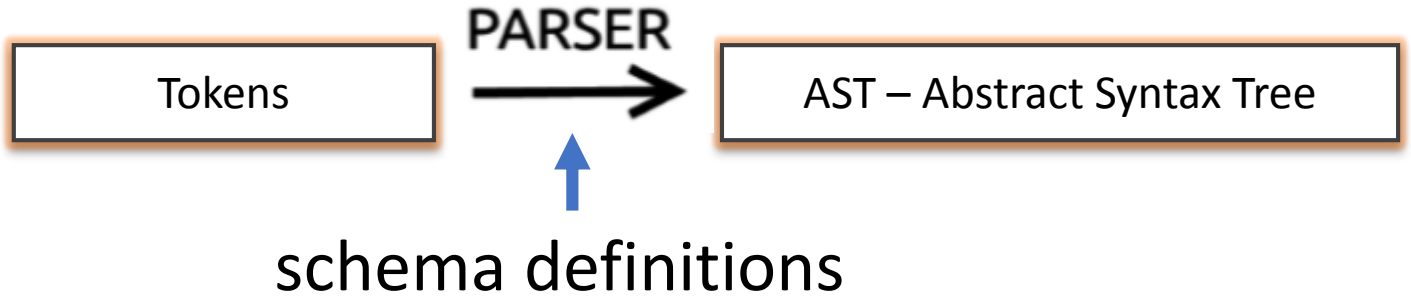


Unless the separator is added, we could still expect, that the definition defined upper is the one to match it – and will usually do it actually – but the algorithm takes no guarantees for that.

E.g.:

```
[NUMBER]: regex ([\d]*\.)?[\d]+     → “1.618”
[CLASS]: word class                 → “class”
[EQUAL]: infix =                    → “=”
=====
[ID]: regex [\w]+                   → “class”
```

Definitions

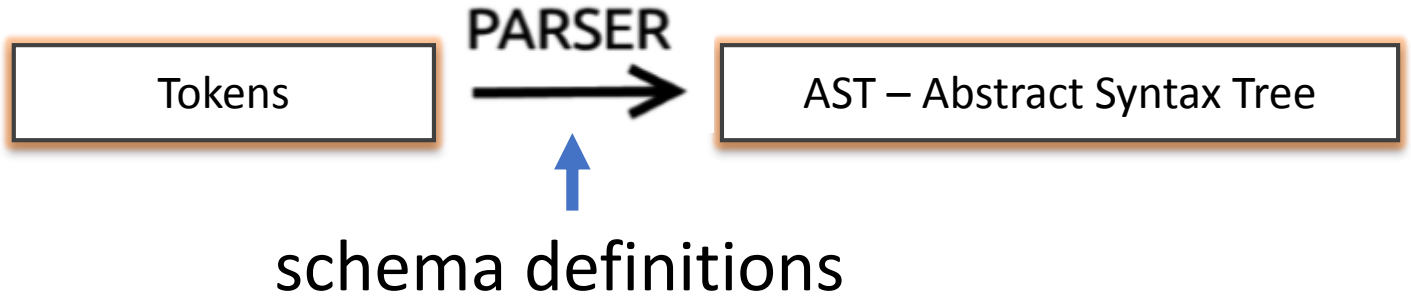


Schema definitions can be defined similarly,
but with “at” (@) symbol before their names and without brackets.
This difference will be important in a moment.

E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions



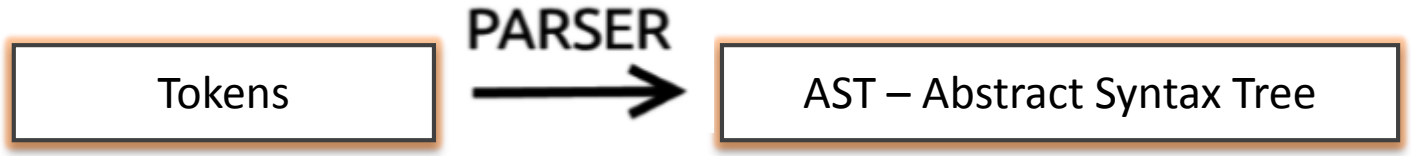
`myFunction()`

Let's see an example.

E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions



The input character sequence gets supplied into the lexer, which creates 3 tokens from it.
(Let's assume the token definitions are made somewhere else and so this is the result.)

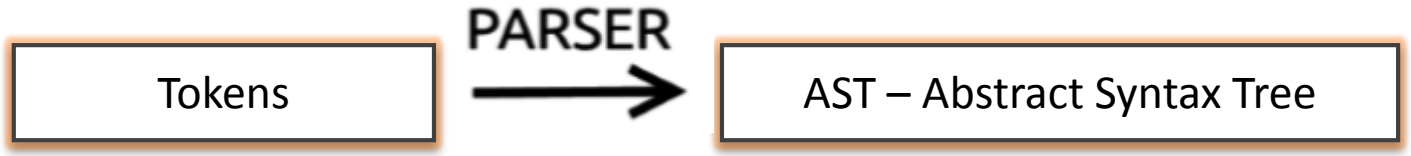
`myFunction()` → LEXER →

- `"myFunction"` ID
- `"("` LParen
- `")"` RParen

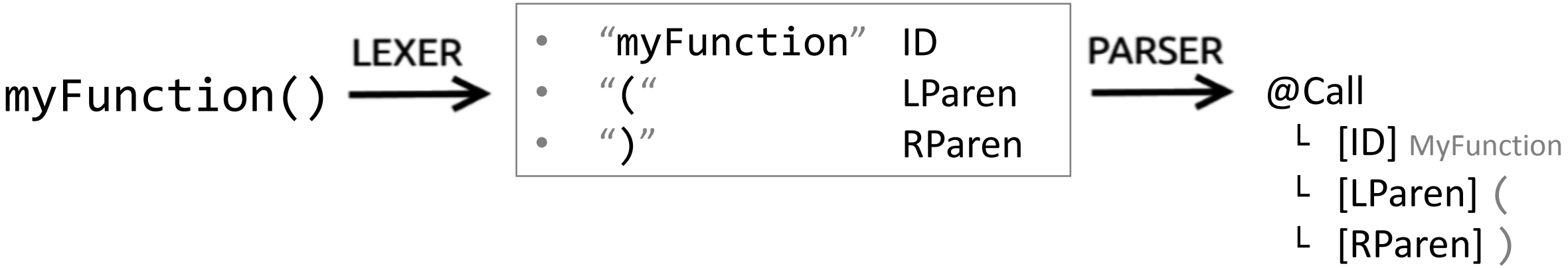
E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions



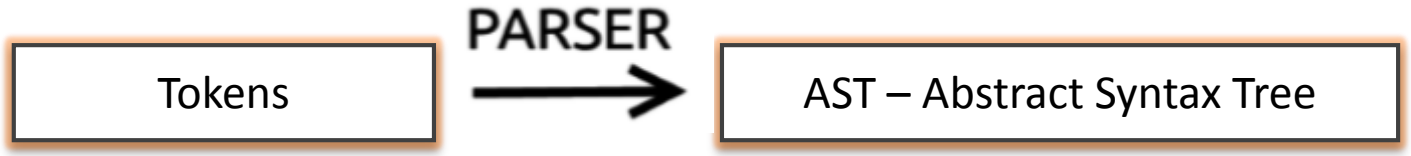
The parser then will match one “Call” schema node, containing the three tokens, because they were matched by the matcher of the “Call” schema definition.



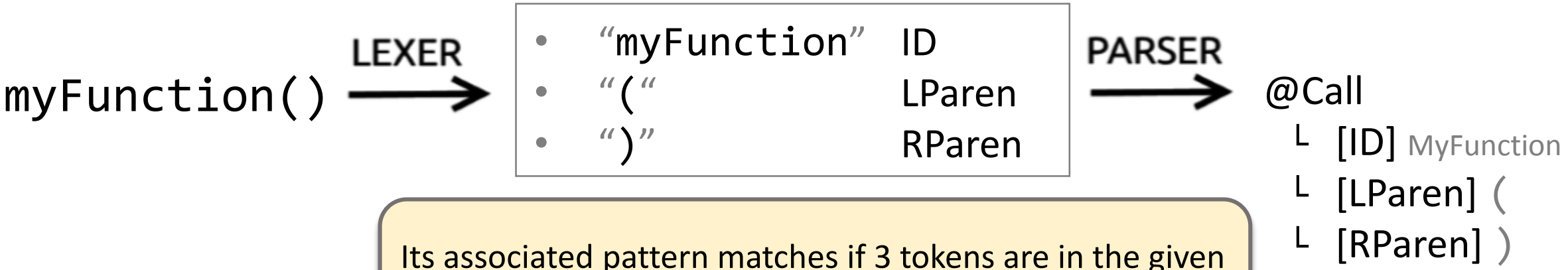
E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions



The parser then will match one “Call” schema node, containing the three tokens, because they were matched by the matcher of the “Call” schema definition.



E.g.:

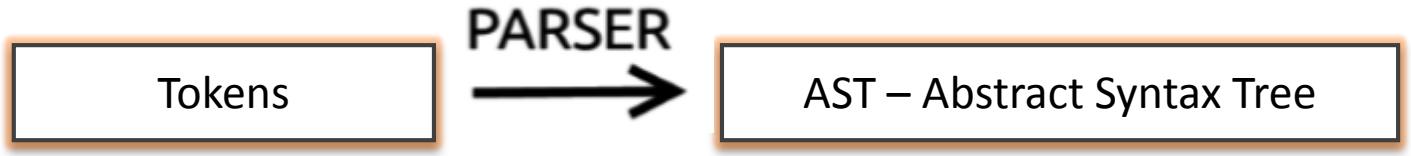
```
@Call: [I
@Value: [
```

Its associated pattern matches if 3 tokens are in the given order, being recognized of token definitions with names:

- “ID”,
- “LParen”,
- “RParen”

in this order.

Definitions



schema definitions

myFuncti

Its associated pattern matches if 3 tokens are in the given order, being recognized of token definitions with names:

- “ID”,
- “LParen”,
- “RParen”

in this order.

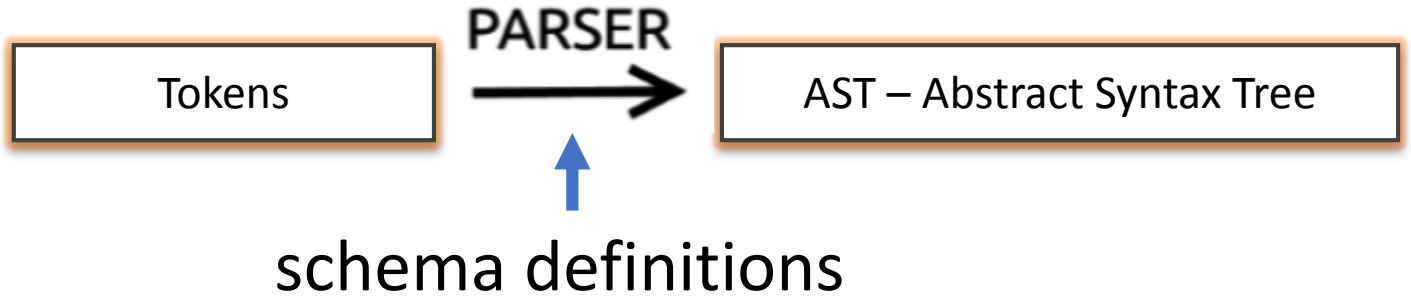
PARSER

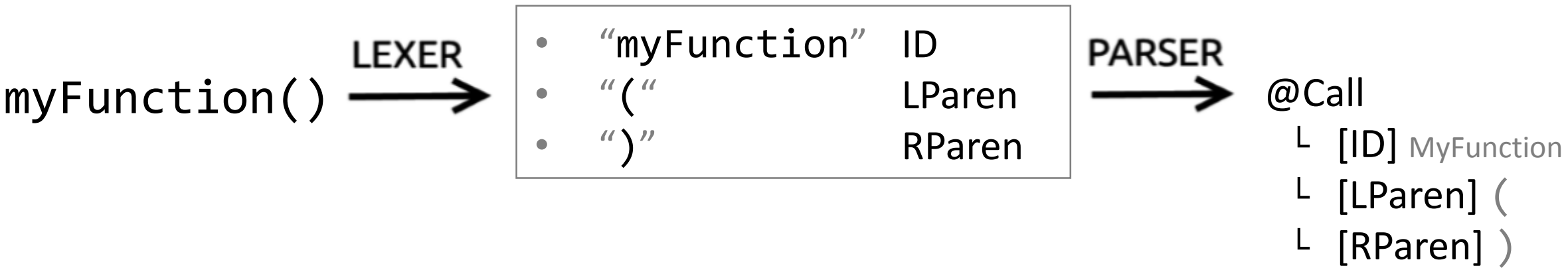
@Call
- [ID] MyFunction
- [LParen] (
- [RParen])

E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions

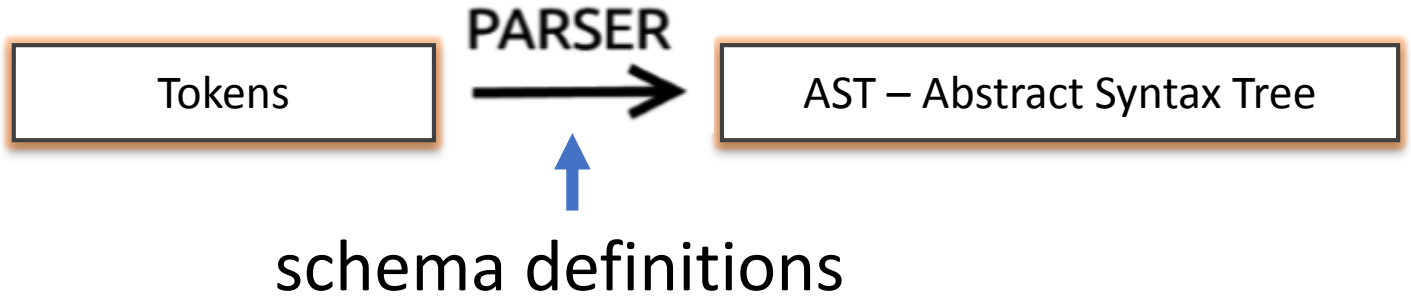




E.g.:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions

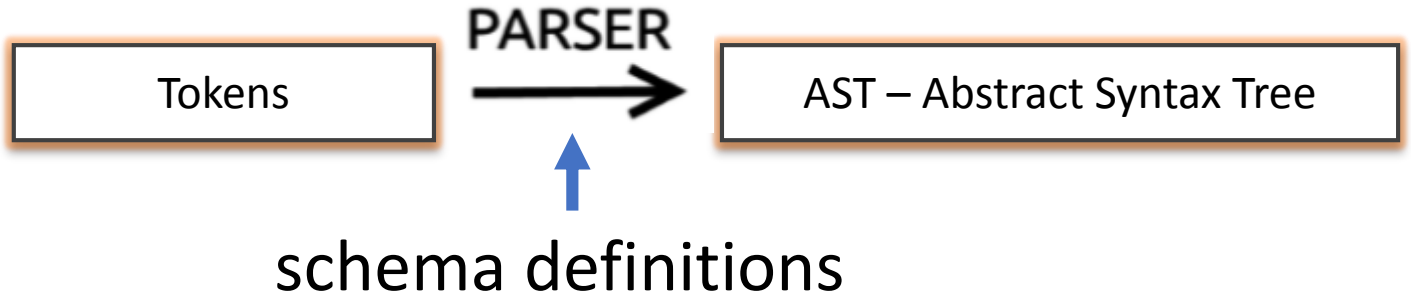


The second example shown some additional features, which schema patterns are capable of. First, notice the | symbol there. It means “OR”, so the “Value” schema definition matcher either a lonely “Number” token, or the other side of the OR symbol.

Példák:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions

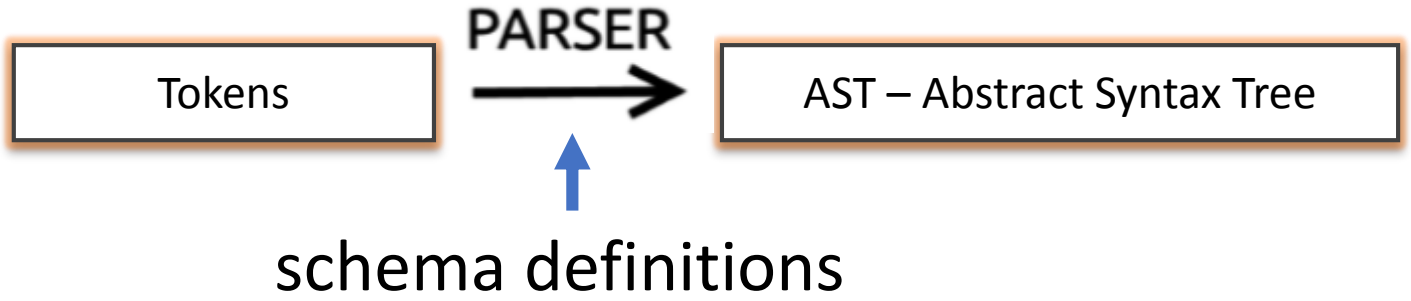


Now let's examine the right-hand side of this "OR".
It references the schema definition itself, recursively, which is completely valid.
And so it matches three elements following each other, being an already recognized "Value" schema, an "OP" token and another already recognized "Value" schema.

Példák:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions

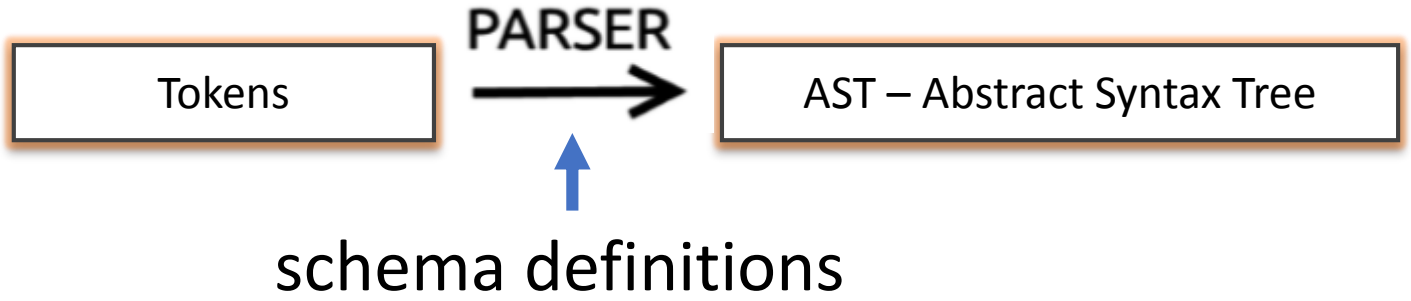


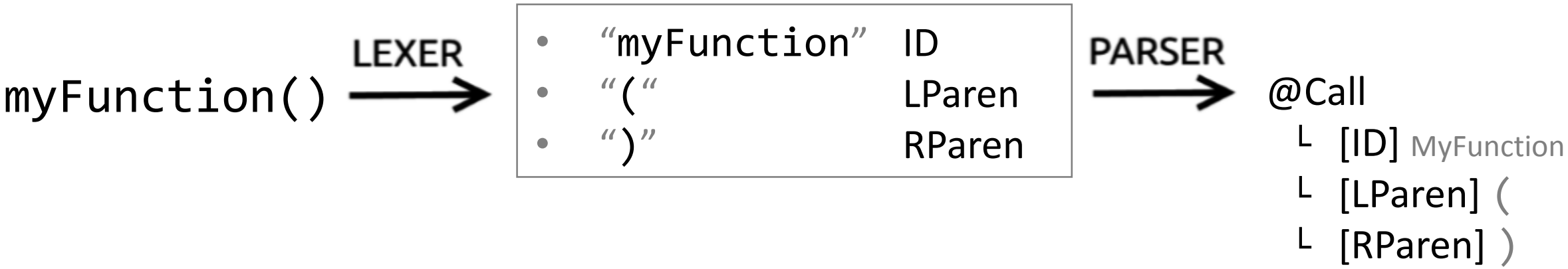
Also notice, that the pattern does not work with tokens only, but expects schemas too.
This is, because the interpretation is carries out in loops
and sequentially stepping though all hard sections of the given definitions.
What a previous schema definition recognized,
a later one can use in its pattern and so they build from each other.

Példák:

```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions

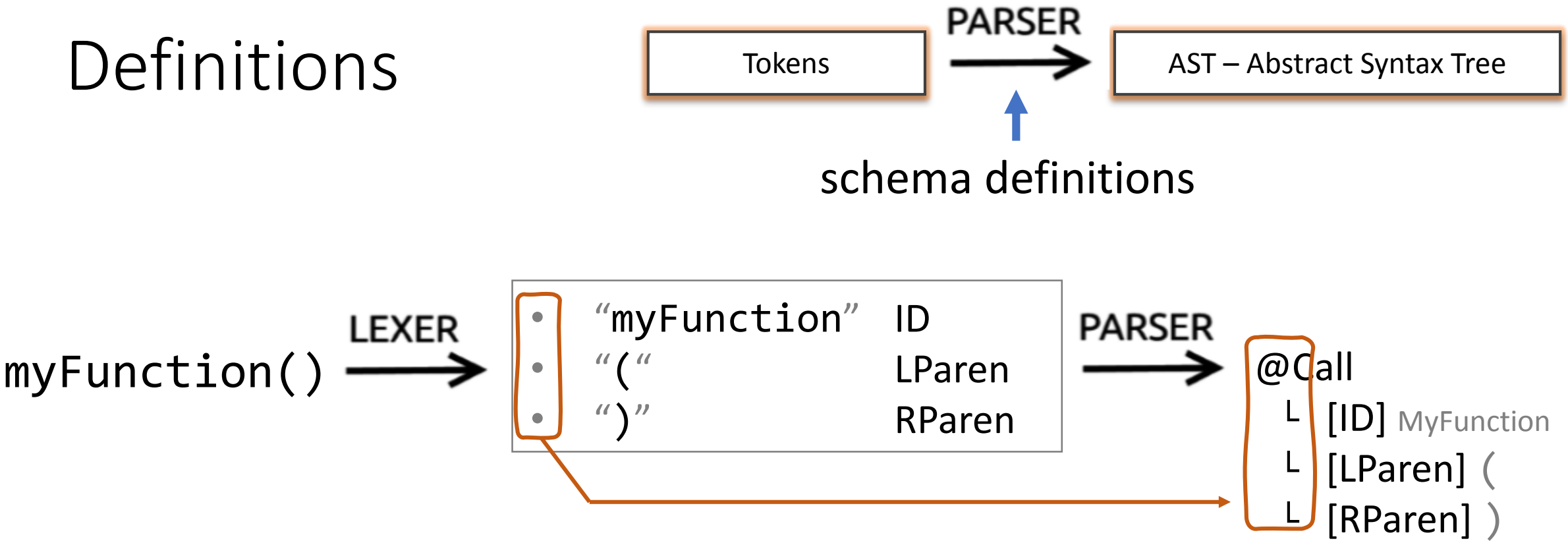




Példák:

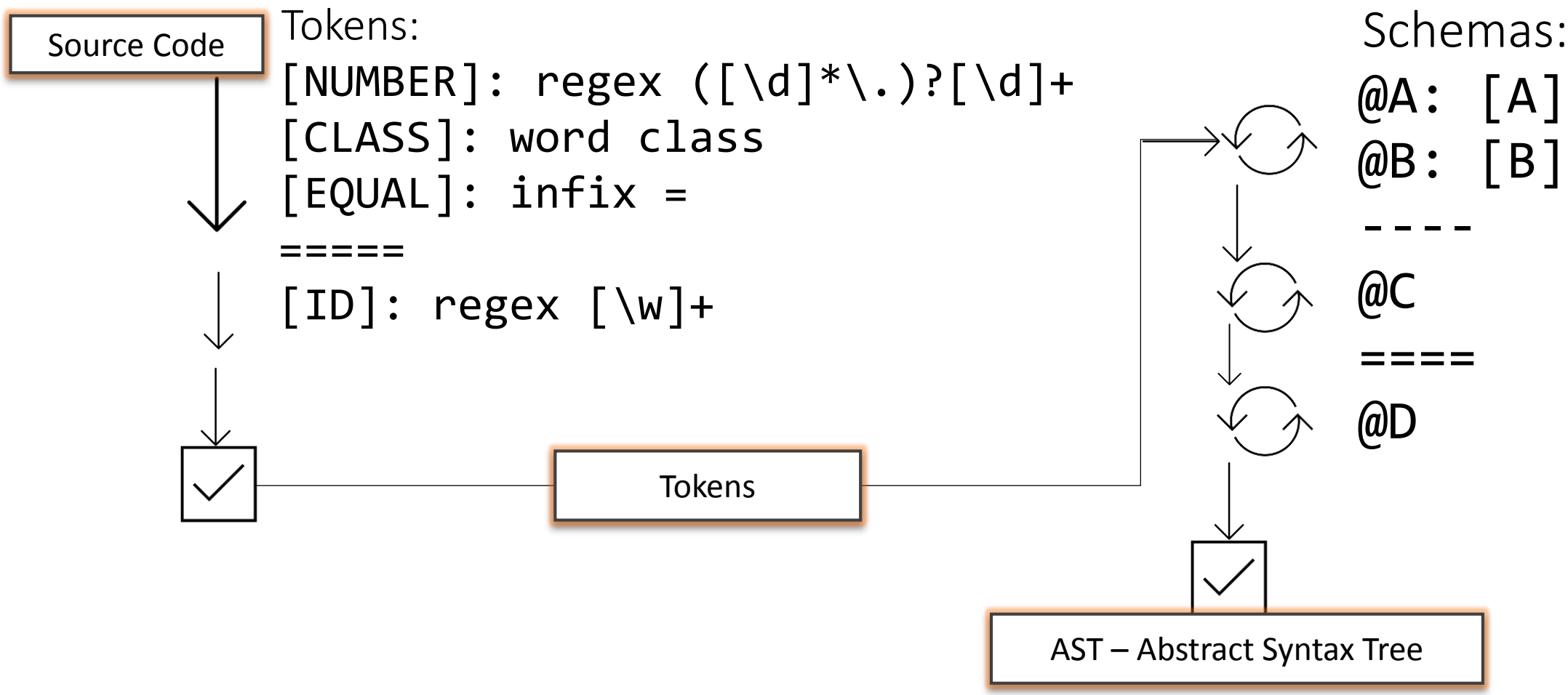
```
@Call: [ID] [LParen] [RParen]
@Value: [Number] | @Value [OP] @Value
```

Definitions



An important detail should not be missed here:
while the patterns matched are sequential, the match forms a new schema node, which is hierarchical in nature.
This means, that when the "Call" schema definition matches, it creates a new schema match node and adds the matched elements as its children.
Since all schema definitions create such schema nodes, one or more trees are created in the end, being the resulting AST.

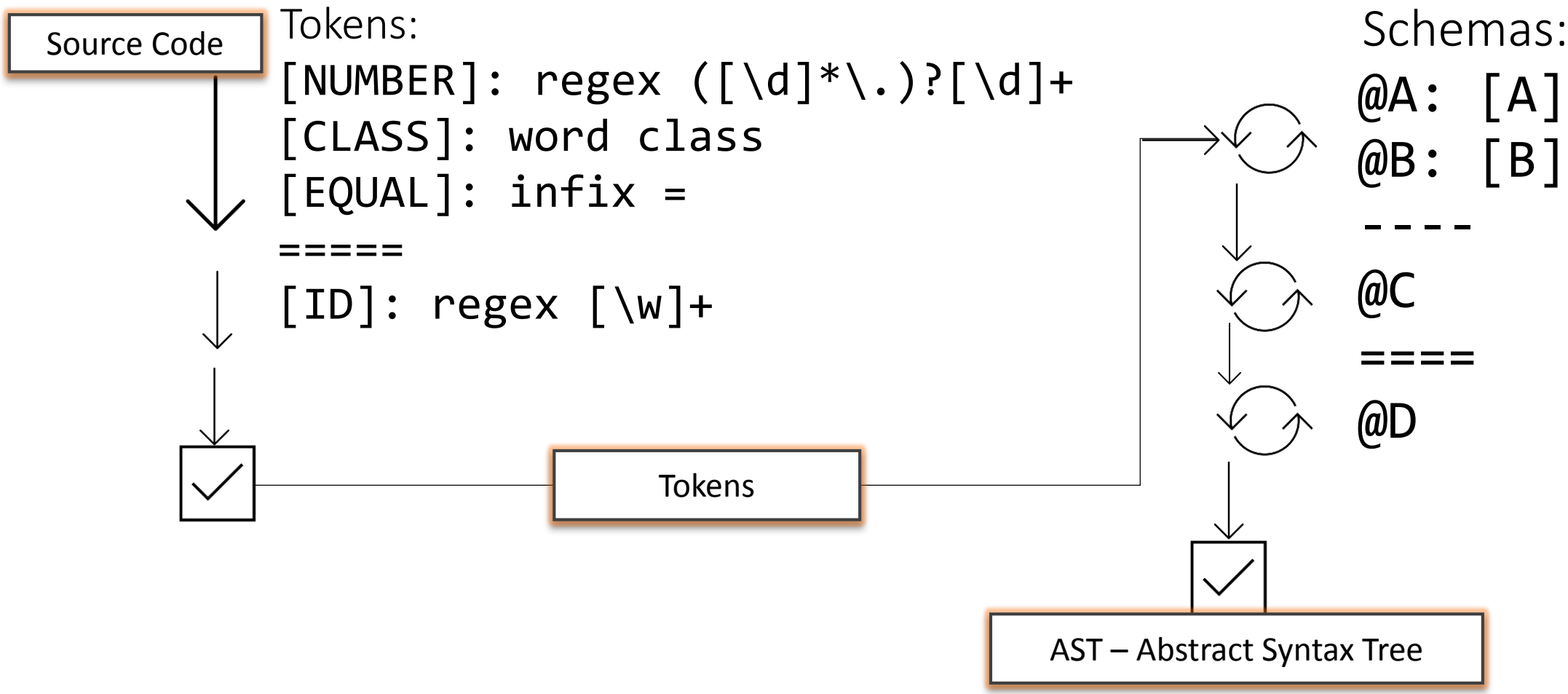
Process


Source Code
Tokens:
[NUMBER]: regex ([\d]*\.)?[\d]+
[CLASS]: word class
[EQUAL]: infix =
=====
[ID]: regex [\w]+
Tokens
Schemas:
@A: [A]
@B: [B]

@C
====
@D
AST – Abstract Syntax Tree

As an overview of the whole system;

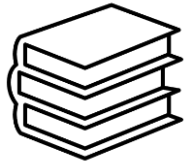
- the source code enters the interpreter
- based on the system of given token definitions the lexer creates a sequence of tokens from the source code
- this sequence then enters the parser
- the parser creates hierarchical schema nodes of it based on the given system of schema definitions
- the tree or trees present in the end are returned as the resulting AST



Results and Improvements

Other parsers are complex, this one is simple and flexible.

This was the original goal too.



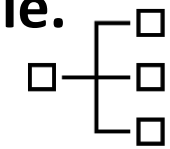
The workings of the algorithm is closer to how we conceptualize.

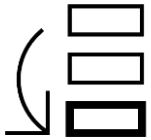
Declarative, can be learned and used fast.

Provides many ways to parallelize processing of one given file.

Coarse-grained: the recognized blocks can be interpreted parallel.

Fine-grained: each matcher can run parallel.





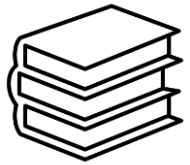
Already interpreted parts of the file can be reused in later interpretation.

Only the change-affected part of the AST must be reinterpreted.

Results and Improvements

Other parsers are complex, this one is simple and flexible.

This was the original goal too.



The workings of the algorithm is closer to how we conceptualize.

Declarative, can be learned and used fast.

I have checked out multiple interpreters, but as being my hobby, I did not want to allocate much time to learn their own ecosystem - also having the fear, that once I learned them and recognized the given interpreter does not serve my needs well, I wasted all the effort, since another interpreter can be set up and configured differently. This inspired my goal to create a very simple one, which supports all basic needs of interpreters. It should be understandable, must be relatively simple, but should be able to accomplish all usual features of interpreters. A primary decision is flexibility over performance; although performance remains a concern, maintainability is considered much more important.
These ideas are summarized in the first two 2-liners.

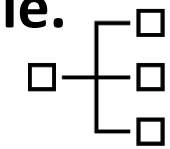
Results and Improvements

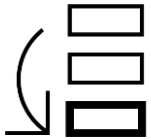
The two on the bottom on the other hand presents some serious optimizations available in the algorithm, coming from its relatively unique concepts. While they are not the main interest of the project, one might focus on them and pursue the concepts of this algorithm to accomplish such optimizations, if found promising, so I felt adding these thoughts a necessity.

Provides many ways to parallelize processing of one given file.

Coarse-grained: the recognized blocks can be interpreted parallel.

Fine-grained: each matcher can run parallel.





Already interpreted parts of the file can be reused in later interpretation.

Only the change-affected part of the AST must be reinterpreted.

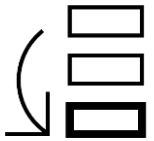
Results and Improvements

Many compilers spare computation time by remembering what, when, and with what results did it compile previously, and if some parts are unchanged, then reuses the result awhole or partially of the previous compilation.
This is very useful, especially in case of large projects with thousands of files – but if a file changes, it still must be recompiled.
With the interpreter itself being hierarchical and recognizing blocks, it could be implemented with not that much of an effort to recognize unchanged parts and reuse their previous compilation AST tree part. With even the previous compilation of files being reused, the compilation effort could really be minimized. This however depends on the given programming language. That more context-free the language is, the best results can be achieved – although that might involve more semantics than syntax, so this approach could still work.

Provides many ways to parallelize processing of one given file.

Coarse-grained: the recognized blocks can be interpreted parallel.

Fine-grained: each matcher can run parallel.



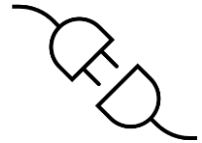
Already interpreted parts of the file can be reused in later interpretation.

Only the change-affected part of the AST must be reinterpreted.

Future Plans

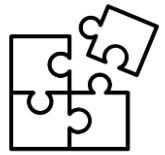
Future plans, for now, are these.

Add/Improve Usability



- Integrate into Visual Studio Code as a highligter and interpreter engine.
- Provide a CLI interface.

Add convenience features.

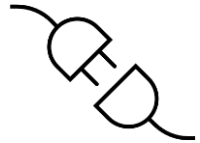


Add simple solutions for everyday problems, like:

- matching string with escaped quotes
- more control over schema node insertion

Future Plans

Add/Improve Usability

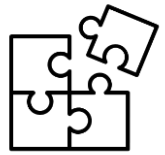


- Integrate into Visual Studio Code as a highligter and interpreter engine.
- Provide a CLI interface.

Integration into VSC, so it obtains a containing IDE, which uses it for actual work.
Adding a CLI is obviously a base thing, so that application can be used standalone and called from command line tools and other applications.

Adding convenience features is mainly about making the algorithm support real-world scenarios better. Handling escape characters is not a trivial question, but can be simplified by magnitudes with some easy implementation, so it should be added.
The plans are to implement that with configurable range matchers.
Also, more control over schema node insertion could make the system less strict and more flexible on the level of definitions, so the maintainer developers can really focus on readability and semantically logical structure instead of being stuck with adding definitions only where the hierarchy in the AST needs it.

Add convenience features.

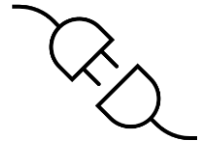


Add simple solutions for everyday problems, like:

- matching string with escaped quotes
- more control over schema node insertion

Future Plans

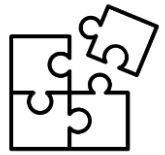
Add/Improve Usability



- nd interpreter engine.
-

Thank you for your attention! :)

Add convenience features.



Add simple solutions for everyday problems, like:

- matching string with escaped quotes
- more control over schema node insertion

Thank you for your attention! :)

Thank you for your attention! :)

Thank you for your attention! :)