B5 - Advanced Functional Programming

B-FUN-500

Parsing Expression Grammar

A gentle introduction

1.0

{EPITECH.}

This paper is a gentle presentation of the PEG algorithm and formalism.
PEG stands for **Parsing Expression Grammar**.
It's a formalism and an algorithm that allow you to easily write parsers in any programming language.

BRIEF TOUR OF LANGUAGE THEORY

Writing a parser is a well known problem.
The story begins in 1968 during the writing of the ALGOL compiler, when **John Backus** and **Paul Naur** need a formalism to express the language without ambiguities, as well as an algorithm to simplify the task of programmers.

They create the **BNF (Backus & Naur Form)** to specify the language.
We won't discuss the entire story here, but all you need to know is that there are a lot of algorithms to write parsers (LL, LR, PEG, ...).

Check this out on the Internet if you do not feel confident about it.

But parsing algorithms and writing parsers is just the beginning.
The main objective is to translate a stream of characters (the language) into a **programming data structure** that could be dealt with by the rest of the program.

Therefore, writing a compiler (or an interpreter) is more challenging than just parsing some text.
In parsing lectures, there is talk of building an **AST (abstract syntax tree)** and the many ways to handle it.
A good method for constructing an AST during parsing will be presented here.

We've chosen **PEG** for you for many reasons:

- it's easy to understand;
- it's easy to implement;
- it doesn't need tools (but some exists, like pyrser);
- it's usable in any language.

The academic paper explaining PEG is here.
You **must** read it.

Here, we'll gently explain how one could understand this formalism and write a parser with it.

What is a formalism?

Don't panic. Using specific symbols to describe a grammar is very common.
All you need is a few hints as to how to read it.

The PEG formalism is just a specific language to describe what to read.
It's so precisely defined that you could have specific tools that read it and generate the parser's code in a specific programming language.
You could also read it as a specification and implement the parser by hand.

The benefits of this kind of formalism are that:

- we use a common language between programmers, without ambiguities;
- we use language to describe our problem in an very expressive way without wanderings in detail.

HOW DO YOU READ IT?

If you read the academic paper you can see that PEG uses a set of symbols and identifiers.
The aim of a PEG is to describe what to read in a logical way.
It’s **a sequence of rule definitions**.
Rules calling other rules in a certain order.
You must read it in a natural order, from top to down, left to right.

Because parsing is reading a stream of characters in a specific order, a set of symbols is used to order the reading sequence of rules.
The first rule defined is the main rule, which recursively calls the others.

There are 2 kinds of rules:

- **terminal rules** that effectively read characters, and use the following symbols:
  - simple (’) or double quotes (") for defining literal strings,
  - brackets for defining range of characters to read (e.g. `[a-zA-Z_]`),
  - dots (`.`) to read any character;
- **non-terminal rules**, defined using other rules, using the left arrow (`<-`) to introduce the rule definition.

Some symbols are used to describe the reading order:

- parenthesis (`()`) for grouping rules that must be read in sequence
- a set of operators (`?`, `*`, `+`) for repeating a sequence

The most important thing to understand is the order of the sequence and the prioritized choice (denoted by slash ‘`/`’).

Indeed, when you read:

```
A <- B C / D
```

You must understand:

```
To read an ‘A‘ rule you must read a ‘B‘ rule followed by a ‘C‘ rule.
Otherwise, forget all past reading and just read a ‘D‘ rule.
```

You may want to come back to this sentence every now and then.

How do you use it?

Parsing Expression Grammars describe a **top-down recursive parser** and it is relatively easy to translate them into any programming language.
All you need to do is to follow some guidelines during your translation.
The process is so simple that some tools do it for you. However, you need to understand what is going on behind the scenes.

You just need to follow these 7 rules of thumb:

1. Non-terminal rules are just **boolean functions**.
   We call them rule-functions.
2. Terminal rules are also **boolean functions** but effectively read from an "infinite" stream of characters (the input).
3. The "infinite" stream of characters is indexed by an **internal cursor**.
4. Terminal rules have fundamental behaviors to **read from the stream of characters**.
5. Ordering symbols have fundamental behaviors to **sequence and call rule-functions**.
6. When a rule-function returns `true`, the **stream's internal cursor has moved** due to the calls of all the functions that describe this rule.
   A valid sequence of rules was found.
7. When a rule-function returns `false`, the **stream's internal cursor hasn't moved**.
   No valid sequence of rules was found.

+ A basic example

```
IniFile <- Sections* #eof

Sections <- Spacing '[' Identifier ']' KeyValue+

KeyValue <- Spacing Identifier Spaces '=' Spaces Value Spaces EndOfLine

Value <- [^ \t\r\n]*

Identifier <- [a-zA-Z_][a-zA-Z0-9_]*

Spaces <- [ \t]*

EndOfLine <- '\r'? '\n'

Spacing <- (Spaces EndOfLine)? / ';' [^\n]* EndOfLine
```

This grammar specifies how to parse an INI file.

It is assumed that:

- a leading ';' denotes a comment.
- leading and trailing whitespace around keys and values should be ignored
- Windows or Unix files can both be read (there may therefore be a carriage return (`\r`) character before newlines (`\n`))
- **#eof** is a special rule that denote the end of file. So, it's just a function that check **internal cursor**.

+ A STEP BY STEP TRANSLATION IN C.

You could find next to this document an example in C in the tarball (c_peg.tgz).

RULE OF THUMB #1:

We have 8 rules, so we need 8 boolean functions. We could name them exactly like the rules, or prefix them with `read_`.

RULE OF THUMB #2:

We need a set of basic functions to read the grammar's terminal rules: `[`, `]`, `=`, `;`, `\r`, `\n`.
So we define a primitive rule:

```
int read_char(parse_ctx_t *p, char c);
//Depending on our parsing context 'p', we try to read the character provided by c.
```

RULE OF THUMB #3:

We define a struct parser_s to store the parsing **internal cursor**.

RULE OF THUMB #4:

All terminals rule are handled by these functions:

- parser_eof
- parser_getchar
- parser_peekchar
- parser_move_cursor
- parser_readchar
- parser_readrange
- parser_readtext

RULE OF THUMB #5:

Ordering and sequence of rules are handled by these macros:

- Optional
- Rep0N
- Rep1N
- Alt

RULE OF THUMB #6:

The internal cursor is handled by these macros:

- HANDLE_CACHE
- YES_IN_CACHE

RULE OF THUMB #7:

To restore the **internal cursor**, we save it at the beginning of function.

```
char *tmp = p->cursor;
```

And we restore it before leaving when failed.

```
p->cursor = tmp;
return false;
```

+ CONSTRUCTING AN AST:

Check in the 'test.c' file, how the following functions are used:

- parser_begin_capture
- parser_end_capture
- parser_get_capture

How do you write it?

You completely understand PEG when you start writing a specification with it.
However, just like you learned how to program, you need to learn some basic patterns that are commonly used.

+ List, Separator and priority.

Writing a grammar about a specific language is like searching logical sequences hidden behind the character stream.
A common algorithm, is the **list** of **item** separated by the rule **sep**.

```
list <- item (sep item)*
```

Another common pattern, is to use the **top-down** nature of the parsing to handle priority.

```
lowprio <- hiprio (sep hiprio)*
hiprio <- term (sep term)*
```

+ Left and Right Recursivity

Due to is **top-down** nature, you must avoid **left recursivity** of rules and write grammar at least using **right recursivity** or repeaters *,+,?.
Indeed **left recursivity** could produce **infinity loop** if you never move the **internal cursor**.

You could find here how to handle left recursion in a packrat parser.

+ Lookahead

See the following macros to see how to handle predicate.

- LookAhead
- NegLookAhead
- Complement

+ Handling errors

If you store the deepest value of your **internal cursor**, you could easily detect **syntax error**.