ParseLang

The language that expresses itself

Programmers love languages. They love to use them, have both interesting and fruitless discussions about them, and above all, they love to design them. All that love comes at a cost, however. This is why there are about 24 Javascript frameworks and about 9000 programming languages, most of which you've never heard of. Many of these languages originate from people who want to develop in a specific style that their preferred language does not support (e.g. lambdas), or who want to express their world view in their code.

Take for example the simple `System.out.println("Hello world!");`. Some developer like to type it that way, while some people prefer `printf("Hello, World!");`, `print("Hello, World!")` or even `Dear compiler, would you be so kind as to print the string "Hello, world!" for me in the console?`. In pretty much all cases, the programming language that you use decides which of these you have to use to get that famous greeting printed in your console.

Not with ParseLang.

Some languages are more flexible than others: some allow different ways to express the same semantics, some have weak typing and some don't even care whether you use semicolons. ParseLang is so flexible, it allows all of this and much more. In fact, it allows you to ***change the grammar of the language itself***. In it, you program by extending the language with new parse rules and also using the language to describe the semantics of that new rule. This way, it allows *any semantics* to be coupled with *any* textual expression.

While it allows you to extend the language to a beautiful, simple whole, it also allows you to go completelty overboard. Take the following two examples of beauty and madness that demonstrate what ParseLang may support after extending the language yourself:

```
(after some lines of code to allow the following syntax...)
```

```
131 | ConcatFun < SimpleExpression = 'repeat' WhiteSpace '('
Expression a WhiteSpace ',' WhiteSpace Expression b
WhiteSpace ')' {
133 |     String result = '';
134 |     for (int i = 0; i < b; i++) {
135 |         result = result + a;
136 |      }
137 |      return result;
138 | }
139 | repeat('hello', 3)

> hellohellohello
```

(This prototype doesn't support some elements required for this yet, but this does give an indication of what's possible with a future version)

```
(after some lines of code to allow the following syntax...)

131 | Foo < SimpleExpression = ('much' WhiteSpace)* muches
'bigger than ' Expression myVariable {
133 |     int y = (λx.x+1)(myVariable);
134 |     do <y += 1000> length(muches) times;
135 |     rename y to u;
136 |     give a compliment;
137 |     u;
138 | }
139 | much much much much bigger than 6

> Your hair looks nice!
> 4007
```

(This prototype doesn't support some elements required for this yet, but this does give an indication of what's possible with a future version)

Don't be scared of the complexity of these examples: you can make the language precisely as complex as you want, because **you** are the one extending the language (beyond its *very* basic set of instructions). As you can see, you can go completely overboard with how you want to specify your semantics, and ParseLang will run it.

F.A.Q.

1. **With whom did you make this language?**
   I (Pim, 23) made this language on my own.
2. **Can ParseLang really bind any semantics to any syntax?**
   Well, up to a reasonable degree. The first iteration of ParseLang only recognises context-free languages. In language theory, this is an infinite set of languages that contains a lot of programming languages, but not every. Python, for example, uses indentation to indicate its scopes, which makes it not a context-free language. Most parser generators can not handle this either, and require you to use a preprocessor for indentation. I have plans to improve ParseLang that allow a greater set of language features, including Python's indentation.
3. ***Did you really write a parser from scratch? Why not use a parser generator?***
   I did, and it was not easy. However, parser generators do not support changing the grammar during the parsing process, hence I had to write my own. This is also why the parser is currently limited in functionality (e.g. absence of left recursion support): it is a one man project that includes writing a freaking parser from scratch.
4. **Why doesn't ParseLang use a tokenizer? Tokenizers are good practice in parsers.**
   The goal of ParseLang is to give the user absolute freedom over how their content is being parsed. This freedom means that every single character has to be a terminal, as to allow parse manipulation of single characters.
5. **Programming in this language is hard! I keep getting parse errors!**
   ParseLang is, and never will, be an easy language to program in, since you keep changing the language itself. Furthermore, if you make a change in your ParseLang program that changes parse rules such that the parser fails later in the program, the mistake is very difficult to identify. I recommend frequent backups, and sticking to the same format once you've designed a solid language base.
6. **Programming in this language is hard! I keep getting (other) errors!**
   The base language uses a Java interpreter that makes some assumptions about the semantics of certain nonterminals. It assumes, for example, that every Number has the semantics of an integer. By extending the language, you can change this behaviour and make it return a String, causing exceptions underneath. Usually, these exceptions explain pretty well what is going wrong.
7. **This is just what macros are, this isn't special at all.**
   *Macros allow redefinition in some languages **within the bounds of the tokenizer**, and often using only the base language to define its semantics. Parselang offers absolute flexibility over what you define without a tokenizer, and allows you to use previously extended language.*

Language

In this section, we'll go over each language element and how it works.

Expressions

An expression in ParseLang is anything that has a semantic value. This includes:

1. An integer (e.g. `2` or `-100`)
2. A floating point number (e.g. `2.3` or `-1.0`)
3. A string (e.g. `'hello!'`)
4. A summation (e.g. `3 + 100.3`)
5. A subtraction (e.g. `3 - 100.3`)
6. A division (e.g. `3 / 100.3`)
7. A multiplication (e.g. `3 * 100.3`)
8. A list literal (e.g. `[‘h’, [7.8, 3]]`)
8. A list indexation (e.g. `[‘h’, [7.8, 3]][0]`)
9. A built-in function (e.g. `~concat(['foo', 'bar'])`)

Statements

A statement in ParseLang can have two forms: if it is part of a sequence of statements that are executed in order, it is an **expression** terminated by `;`. This is the case if you are writing more complex declarations, or just like the feeling of sequential programming. This is a valid sequence of statements:

```
5 + 3;
'hello!';
```

And this is not:

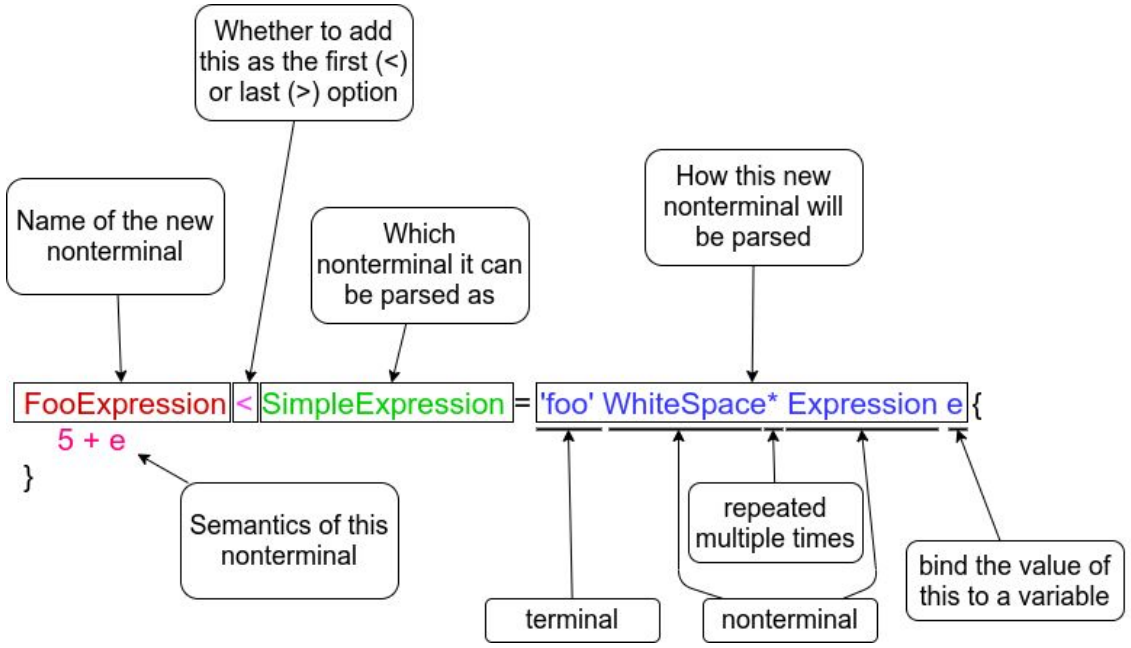
```
5 + 3
'hello!'
```

If it is not part of a sequence of statements, the semicolon can be omitted. This is a valid single statement:

```
5 + 3
```

Declaration

A declaration is the core of ParseLang. A declaration extends the ParseLang grammar with an extra rule, and indicates its semantics. The structure of a declaration looks like this:



This example adds two new parse rules to the parser:

1. SimpleExpression = FooExpression
2. FooExpression = ‘foo’ WhiteSpace* Expression

Note that ParseLang uses a packrat parser. That means that an order exists between parse rules, and that it will always attempt parse rules in that order. Rule 1 will be added as *first* candidate for the nonterminal SimpleExpression because of the usage of the `<` symbol. Rule 2 will by default be added as the *last* candidate for FooExpression.

It starts with indicating which nonterminal you would like to add a new rule for (marked in red). This can be a new nonterminal or one that already exists. Then, you specify when this nonterminal will be used. In the example, we add a simple rule to the front of the list of rules to parse a *SimpleExpression* as, such that it can also be parsed as a single FooExpression.

Then, the right hand side of the parse rule follows. This may be a combination of terminals (strings with single quotation marks) and nonterminals. Moreover, these tokens can be grouped with brackets `(` and `)`, and may be appended with a

kleene star `*` to denote it should be greedily parsed as many times as possible. Lastly, nonterminals or grouped tokens may be assigned a parameter name. This way, the (semantic) value of whatever the content of this token is can be accessed in the definition of this rule's semantics. If this parameter name is appended with a single quote `'` it is regarded as **lazy**: it is not evaluated upon entering this declaration's semantics immediately, but only when it is used inside. This way, it can be executed multiple times.

Lastly, the semantics of this rule are shown in pink. This example has the semantics of 5 added to the value of whatever expression is parsed as the Expression token. Note that this declaration content may use any language element previously defined in a Declaration, and may even recursively use its own rule `FooExpression = 'foo' WhiteSpace* Expression`.

Types

ParseLang recognises the following types:

1. Integers
2. Floating point numbers
3. Lists
4. Strings
5. Maps/dictionaries (in progress)

And that's it. If you want combined types, you will have to define their syntax and semantics through ParseLang.

Example: Greatest common divisor

The greatest common divisor of two integers x and y is the largest integer z such that z divides both x and y.

```
GCD < SimpleExpression = 'gcd'        WhiteSpace*
                         '('          WhiteSpace*
                         Expression a WhiteSpace*
                         ','          WhiteSpace*
                         Expression b WhiteSpace*
                         ')'          {
    ~if(b==0, a, gcd(b, a % b))
}
```

```
gcd(88, 99)
```

Example: "I want double quotes for strings"

ParseLang uses single quotes (') for Strings. This is different from the usual double quotes of most programming languages. This makes for a good challenge for starters of ParseLang: extending the language to allow double quotes as well.

Default behaviour:

```
'hello!'

> hello
```

```
"hello!"

> Exception in thread "main"
parselang.parser.exceptions.ParseErrorException: No
alternative at index (1:1) at "
```

Since ParseLang doesn't recognise text within double quotes as a string literal, it fails and specifies it doesn't know what to do at the first double quote.

The tricky problem with ParseLang using single quotes by default, is that the double quote character is a valid character inside of a string. Therefore, we cannot simply declare the double-quote-string the same way as the single-quote-string. For example, 'foo"bar' is a valid string in the base language. It starts with a single quote, has some safe to use characters and ends with a single quote. If we just substitute the single quotation marks with double quotation marks, ParseLang will think " is the opening quotation, foo"bar" is content of the string and closing double quotes are missing.

```
NewStringLiteral < SimpleExpression = '"' SafeChar* a '"' {
~concat(a)
```

```
}

"hey"

> Exception in thread "main"
parselang.parser.exceptions.ParseErrorException: No
alternative at index (5:6) at EOF
```

The closing quotation " would be parsed as content of the string, and the parser would fail since it does not find a closing quote (i.e. it does not expect the end of the program, as the Exception specifies). To resolve this issue, we specify a new literal that we want strings to be parsed into: NewStringLiteral. It is defined exactly the same as the default StringLiteral, except that it uses double quotes at the start and end and does not permit the double quote character inside the string. This way, a double quote will always be parsed as the *end* of a string.

In the rule `NewStringLiteral < SimpleExpression = '"' NewSafeChar* a '"'`, we specify that a string is a sequence of characters. Furthermore, we bind whatever this sequence is to the variable `a`. If we would just return `a`, we would return a list of single-character strings, which is not what we want. Instead, we use the built-in function `~concat` to concatenate a list of strings into a single string.

```
NewSafeSpecial < Nothing =  '}' {'}'}
NewSafeSpecial < Nothing =  '{' {'{'}
NewSafeSpecial < Nothing =  '(' {'('}
NewSafeSpecial < Nothing =  ')' {')'}
NewSafeSpecial < Nothing =  ';' {';'}
NewSafeSpecial < Nothing =  '+' {'+'}
NewSafeSpecial < Nothing =  '*' {'*'}
NewSafeSpecial < Nothing =  '/' {'/'}
NewSafeSpecial < Nothing =  '-' {'-'}
NewSafeSpecial < Nothing =  '!' {'!'}
NewSafeChar < Nothing = UpperOrLowerCase a {a}
NewSafeChar < Nothing = Number a {a}
NewSafeChar < Nothing = NewSafeSpecial a {a}
NewStringLiteral < SimpleExpression = '"' NewSafeChar* a '"'
{
    ~concat(a)
}
```

```
"3afoo"

>3afoo
```

Example: Redefining the number 5, because you are evil

Challenges

While ParseLang is an incredibly powerful language, it poses some serious challenges. First of all, the language does currently not permit left-recursive expressions. This means that declarations such as `PowExpression < SimpleExpression = Expression a '*' Expression b` are currently not supported. The parser becomes stuck in a never-ending loop of attempting to parse an Expression as a PowExpression starting with an Expression as a PowExpression, et cetera. However, left-recursion **is possible** with so-called packrat parsers such as ParseLang's according to research papers. If this project would be continued, future versions will support it.

What the future brings

- Allowing left recursion
- Adding more built-in functions
- Adding `function` as a type to allow even more flexibility
- Allowing references to parameters to be used within lazily evaluated expressions (in essence, allowing for-loops in which you use parameters).