



ParseLang

The language that expresses itself

Contents:

- Introduction
- [F.A.Q.](#)
- [Language](#)
 - [Expressions](#)
 - [Statements](#)
 - [Declarations](#)
 - [Types](#)
- [Example: Concatenating two strings](#)
- [Example: Greatest common divisor](#)
- [Example: "I want double quotes for strings"](#)
- [Example: Introduction: For-loop](#)
- [Example: Introduction: Madness](#)
- [Challenges](#)
- [What the future brings](#)
- [Appendix: Formal base language \(sorted\)](#)

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 while it is parsing**. 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 completely 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...)
```

```
FooExp < SimpleExpression = 'excited ' NumberLiteral b {  
    String result = 'hello';  
    for (int i = 0; i < b; i = i + 1) {  
        result = result + '!';  
    };  
    result;  
}
```

```
excited 20
```

```
> hello!!!!!!!!!!!!!!!!!!!!!!
```

(Want to try this? Try calling `runDemo("examples/forloop.plang");` from Main.java, then clicking Run.)

```
Evil < Number = '9' {  
    8  
}
```

```
1997
```

```
> 1887
```

(Yes, we are literally redefining the literal “9” as having the semantics of the number 8. Try changing it to a two-digit number for funzies! Try calling `runDemo("examples/madness.plang");` from Main.java, then clicking Run.)

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, student, 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. Can I also declare normal functions?

Absolutely! Just add a declaration with a constant name of your function, an opening bracket, some comma-separated expression argument and a closing bracket. Then you can specify its semantics between `{` and `}`.

4. 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.

5. 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.

6. Programming in this language is hard! I keep getting parse errors!

ParseLang is, and probably 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 running the program after every small increment, and sticking to the same format once you've designed a solid language base. If you are stuck, you can set the verbosity of the parser to ``1``: this prints the parsing process to the console and shows you what it attempts to parse (when it fails).

7. 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, the exception messages explain pretty well what is going wrong. If not, then whoopsie, that's the effect of a prototype
`_(ツ)_/`

8. **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. Furthermore, macros cannot be recursive. ParseLang offers absolute flexibility over what you define without a tokenizer, allows you to use previously extended language and use your language extension recursively.*

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'])`)
11. Data storage read (e.g. `~data['variables'][3]`)
12. Data storage write (e.g. `~data['variables'][3] = 5`). Dictionaries/maps as values are automatically created when you use unrecognised keys.)

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:

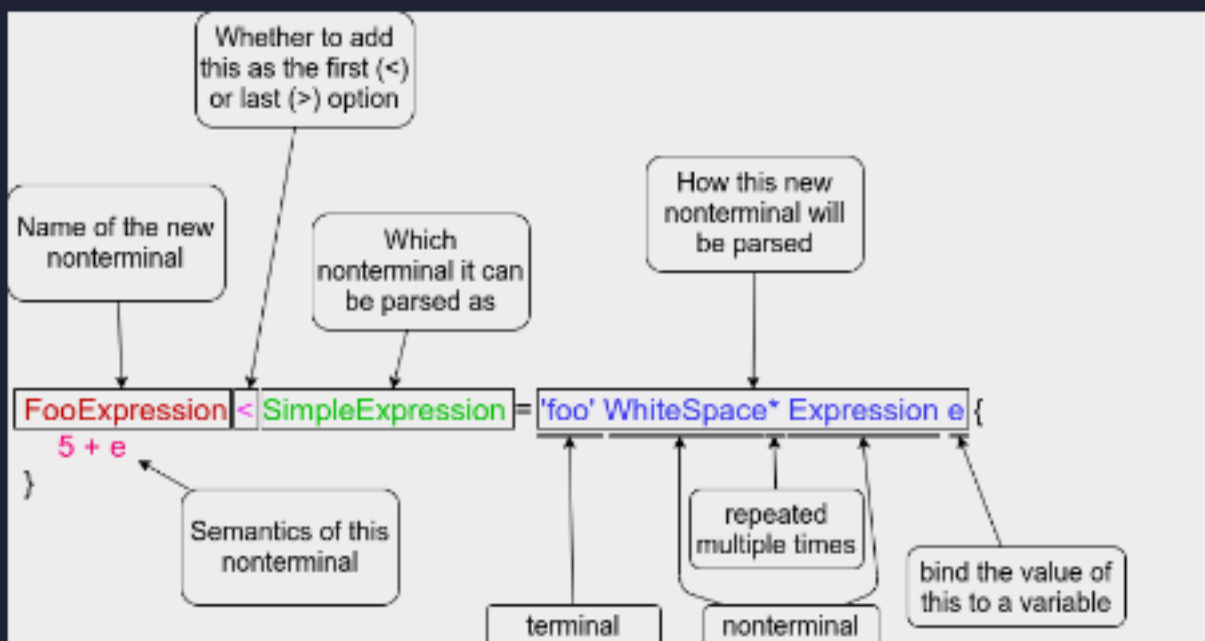
```
5 + 3
'hello!'
```

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

```
5 + 3
```

Declarations

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. Think of it as the *call by reference* instead of the *call by value* for parsed values (rather than function parameters).

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 (only accessible through the `~data` built-in)

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

Example: Concatenating two strings

(examples/concat.plang)

This program adds a new language functionality that allows you to concatenate two expressions in natural language instead of the built-in `~concat` function. It takes two expressions, places them into a list and uses `~concat` internally to join them together as strings.

```
ConcatExpression < SimpleExpression = 'concat '
                                         Expression a
                                         'with '
                                         Expression b {
    ~concat([a,b])
}

concat 3 with 5

> 35
```

Example: Greatest common divisor

(examples/gcd.plang)

The greatest common divisor of two integers x and y is the largest integer z such that z divides both x and y. A common way to calculate this is using the [Euclidean algorithm](#):

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

~concat(['gcd of 88 and 99 is ', gcd(88, 99)])

> gcd of 88 and 99 is 11
```

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: Introduction: For-loop

(examples/forloop.plang)

ParseLang offers no out-of-the box functionality for iteration. However, it does allow for a way to introduce iteration by adding a for-loop construction. In this for-loop, recursion is used internally to execute code multiple times.

Let us first examine what we want to work towards:

```
FooExp < SimpleExpression = 'excited ' NumberLiteral b {
    String result = 'hello';
    for (int i = 0; i < b; i = i + 1) {
        result = result + '!';
    };
    result;
```

```
}  
  
excited 20  
  
> > No alternative at index (2:4) at 'S'
```

Without prefixing anything, this will result in a parse error: ParseLang doesn't know the concept of variable declaration or that of for-loops. Let us start by adding a way to declare a String. The following code block introduces java-style notation for declaring strings. The value of these variables are stored in `~data['variables']`. Note that the input parameters are separate letters that we need to concatenate.

```
StringDecl < DelimitedSentence = 'String ' UpperOrLowerCase first  
UpperOrLowerCaseOrNumber* other WhiteSpace* '=' WhiteSpace* Expression value {  
  ~data['variables'][~concat([first, ~concat(other)])] = value;  
}
```

Next, we need to make sure the value of a variable can be accessed later through its name. We add another declaration that indicates what the semantics are of a variable name:

```
VariableName > SimpleExpression = UpperOrLowerCase first UpperOrLowerCaseOrNumber*  
other {  
  ~data['variables'][~concat([first, ~concat(other)])];  
}
```

Since we also want to update an existing variable ($i = i + 1$), we need to add a declaration that reads an assignment to a variable and performs it:

```
xVariableAssignment < DelimitedSentence = UpperOrLowerCase first  
UpperOrLowerCaseOrNumber* other WhiteSpace* '=' WhiteSpace* Expression value {  
  ~data['variables'][~concat([first, ~concat(other)])] = value;  
}
```

Lastly, we add the for-loop declaration. It takes many arguments, such as the name of the iterator, its initialisation, the condition to be evaluated, how the iterator is updated and the content of the for loop. Since we want to execute the condition and updater multiple times, we assign them to lazy parameters. In the semantics, we first write an initial value of the iterator to memory. Then, we check whether the condition is true. If so, we execute the for-loop's contents, update the iterator and recursively call a for-loop that initialises the iterator to the updated value. This way, iteration is modeled through recursion. If the condition is false, it returns a bogus value.

```
Forloop < SimpleExpression = 'for'                               WhiteSpace*
```

```

'('      WhiteSpace*
'int'    WhiteSpace*
UpperOrLowerCase firsta UpperOrLowerCaseOrNumber* othera WhiteSpace*
'='      WhiteSpace*
Expression iteratorInit WhiteSpace*
','      WhiteSpace*
Expression condition' WhiteSpace*
','      WhiteSpace*
UpperOrLowerCase firstb UpperOrLowerCaseOrNumber* otherb WhiteSpace*
'='      WhiteSpace*
Expression updater' WhiteSpace*
')'      WhiteSpace*
'{'      WhiteSpace*
Sentence sentence' WhiteSpace*
'}'      WhiteSpace* {
~data['variables'][~concat([firsta, ~concat(othera)])] = iteratorInit;
~if(condition,
  [sentence,
    ~data['variables'][~concat([firstb, ~concat(otherb)])] = updater,
    for (int i = ~data['variables'][~concat([firstb, ~concat(otherb)])]; condition; i = updater) {
      sentence;
    }],0);
}

```

The entire program can be seen in the file `examples/forloop.plang`.

Example: Introduction: Madness

(`examples/madness.plang`)

In ParseLang, you can add not only rules for expressions, but also for other language elements, such as numeric characters. By default, the character 9 has, surprise surprise, the semantic value of 9. However, we can add a parse rule to the front of Number that parses it with the semantic value of 8. This results in the following behaviour.

```

Evil < Number = '9' {
  8
}

1997

> 1887
Evil < Number = '9' {
  8
}

```

1997

> 1887

Can you guess what the output is if you define the semantics of 9 to be 10?

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 [Warth et al.](#) Future versions will support it.

What the future brings

- Support for left-recursion in declarations
- Adding more built-in functions (e.g. for list manipulation)
- Adding `function` as a type to allow even more flexibility.
- Better feedback from errors

Appendix: Formal base language (sorted)

AdditiveExpression = MultiplicativeExpression
 (PlusOrMinus MultiplicativeExpression)*
 WhiteSpace*

An addition or subtraction of expressions.

BooleanLiteral = "true" |
 "false"

A boolean literal

BracketToken = "("
 (WhiteSpace* Token)*
 ")"

A grouped collection of tokens.

Comparator = "==" |
 "!=" |
 "<=" |
 ">=" |
 "<" |
 ">" |

A symbol used in comparisons.

ComparitiveExpression = AdditiveExpression
 (Comparator AdditiveExpression)*
 WhiteSpace*

A comparison of expressions

Data = "~data"
 ("[Expression "]")*
 WhiteSpace*
 OptionalAssignment

Access of data and optionally reassigning it

Declaration = NonTerminal
 WhiteSpace*
 GTorLT
 WhiteSpace*
 NonTerminal
 WhiteSpace*
 "="
 (WhiteSpace Token)*
 WhiteSpace*
 "{"
 WhiteSpace*
 DeclarationContent
 WhiteSpace*
 "}"

A declaration of a new parse rule, along with its semantics.

DeclarationContent = Sentence
 WhiteSpace*
 (Sentence WhiteSpace*)* |
 DelimitedSentence
 WhiteSpace*

Either a single statement without delimiter or one or multiple statements with delimiter.

DelimitedSentence = Expression
Sentence without delimiter included.

Expression = ComparativeExpression
Any expression.

GTorLT = ">" |
 "<"

A character used to signify whether to add a parse rule to the front or the back of the packrat queue.

HighLevel = (WhiteSpace* Declaration)*
 WhiteSpace*
 OptionalExpression
 WhiteSpace*

An entire program in ParseLang.

ListLiteral = "["
 WhiteSpace*
 "]" |
 "["
 WhiteSpace*
 Expression
 WhiteSpace*
 (", " WhiteSpace* Expression)*
 "]"

A list of values of various types.

LowerCase = [a-z]
Any single lowercase character

MultiplicativeExpression = SingleExpression
 (TimesDivisionOrModulo MultiplicativeExpression)*
 WhiteSpace*

A multiplication, division or modulo of expressions.

NonTerminal = UpperCase
 UpperOrLowerCaseOrNumber*

A name of a nonterminal.

NonZeroNumber = [1-9]
Any non-zero numeric character

Number = "0" |
NonZeroNumber

Any numeric character

NumberLiteral = "0" |
OptionalMinus
NonZeroNumber
Number*
OptionalDecimalPlaces

A number literal

OptionalAssignment = "="
WhiteSpace*
Expression

Optional part of Data.

OptionalDecimalPlaces = "."
Number* |
""

The decimal part of a floating point literal

OptionalExpression = Expression |
""

Presence or absence of an expression.

OptionalMinus = "-" |
""

Either a minus symbol or not.

PlusOrMinus = "+" |
"-"

Operators with addition level operator precedence

PotentialLazy = "" |
""

A marker of whether a parameter is lazy.

PotentialStar = "*" |
""

A marker of whether the parser should greedily repeat a token.

PotentialVariable = Variable |
""

A variable binding.. or not.

UpperCase = [A-Z]
Any single uppercase character

UpperOrLowerCase = UpperCase |
LowerCase

Any single alphabetical character

UpperOrLowerCaseOrNumber = UpperOrLowerCase |
Number

Any single alphanumerical character

SafeChar = UpperOrLowerCaseOrNumber |
SafeSpecial |
WhiteSpace

Any single character that is allowed within a string literal

SafeSpecial =
" , " |
" { " |
" } " |
" (" |
") " |
" . " |
" + " |
" * " |
" / " |
" _ " |
" ! " |
" ' " |
" = " |
" "" "

Any single special character that is allowed within a string literal

Sentence = DelimitedSentence
WhiteSpace*
" , "

A sentence that ends with a delimiter.

SimpleExpression =
" (" |
WhiteSpace* |
Expression |
WhiteSpace* |
") " |
NumberLiteral |
StringLiteral |
BooleanLiteral |
ParameterName WhiteSpace* |
" ~concat " |
WhiteSpace* |
" (" |
WhiteSpace* |
Expression |
WhiteSpace* |
") " |
WhiteSpace* |
" ~if " |
WhiteSpace*

```

"("
WhiteSpace*
Expression
WhiteSpace*
" "
,
WhiteSpace*
Expression
WhiteSpace*
" "
,
WhiteSpace*
Expression
WhiteSpace*
")"
WhiteSpace*
"~map"
Data
ListLiteral

```

Any expression without top-level operators. A *ParameterName* is dynamically added whenever a parameter is bound in a declaration. *~concat* concatenates a list of strings into one string. *~if* evaluates the second expression if the first expression is true, otherwise the third. *~map* yields an empty map.

```

SingleExpression =      SimpleExpression
                        ("[" WhiteSpace* Expression WhiteSpace* "]" ) *

```

A simple expression with indexation.

```

StringLiteral =        ""
                        SafeChar*
                        ""

```

A string literal.

```

TimesDivisionOrModulo = "*" |
                        "/" |
                        "%"

```

Operators with multiplication level operator precedence

```

Token =               StringLiteral* |
                        BracketToken
                        PotentialStar
                        WhiteSpace*
                        PotentialVariable |
                        NonTerminal
                        PotentialStar
                        WhiteSpace*
                        PotentialVariable

```

A token that may be used in a declaration with potential bindings.

```

Variable =            LowerCase
                        UpperOrLowerCase*
                        PotentialLazy

```

A variable binding of a token.

```
WhiteSpace = "\t" | "\n" | "\r" | " "
```

Any whitespace character