プログラミング言語を作ろう！

今井　朝貴 Twitter:@Tomoki_Imai,Email:tomo832@gmail.com

君はどれくらい、プログラミング言語を知ってるかな？
　１つ？2つ？もっとたくさん？
　世の中には星の数ほど、プログラミング言語があるけれど、
　その多くを僕らは知らない。
　みんなが知ってるような言語もあるけれど、
　それらはほんの一部分。
　このテキストでは、僕らは自分のプログラミング言語を作る。
　誰も知らない、まだ出会ったことがない、プログラミング言語を。

何が必要？
　君はこう思うんじゃないかな……？
　　「なんか難しそう。機械語とか使うんじゃないの？」
　それに対して僕はこう言う。
　　「プログラミング言語を作るのは少し、難しい。
　　　でも、機械語は使わないし、C も今回は使わない。」

　プログラミング言語の中には機械語に変換される言語もあるけれど、
　今回はコンパイルしない言語を作成しよう。
　あと、「プログラミング言語」を書くプログラミング言語として、C#を使おう。

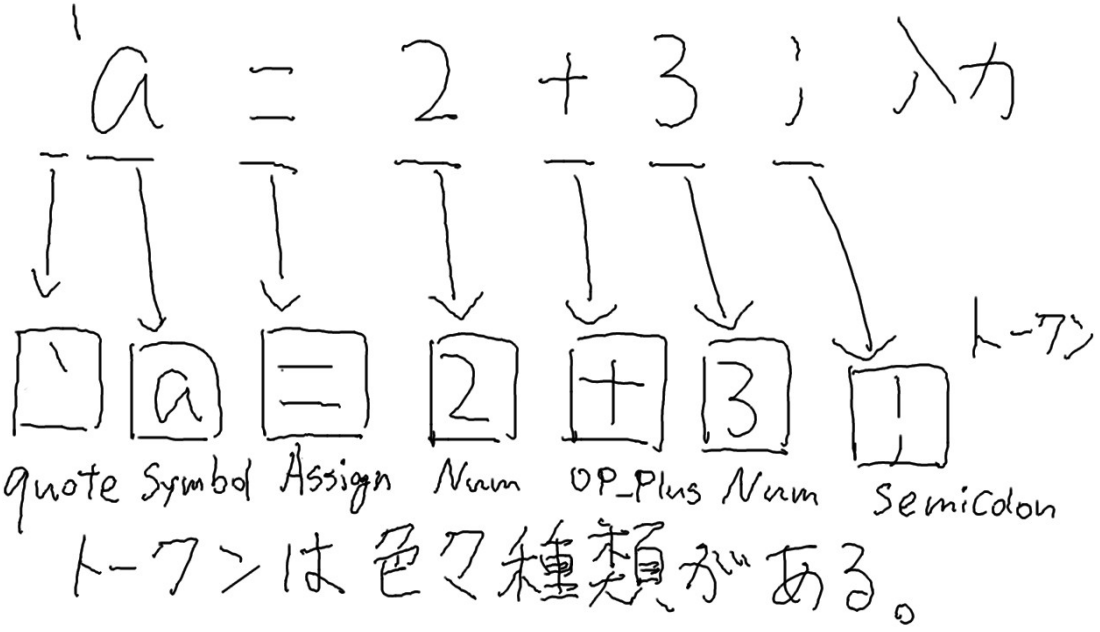
　なんでかって？
　僕は、C#をつかったことが無かったから。
　僕にとって、C#を使った最初の大きなプログラムが、
　このプログラミング言語なんだ。

　それに、「C#講座」というのが最近、始まったからね。
　僕はできるだけ、君と、スタート地点を揃えたかったし。
　ただ、このテキストはC#のテキストではないから、
　文法とかは、別のテキストを見ながら、やるといいよ。

そうそう。このテキストをプログラミング初心者が読むときのために、
プログラミングが上達する、とっておきの方法を書いておこう。
　それは、サンプルを写経すること。
　今回はCalc がそれにあたる。
　コピペですませても、それはあまり意味がないんだ。
　だから、ぜひサンプルを書き写してほしいな。
　変数名とかはもちろん変えていい。むしろ、それはいいことだと思う。

さて、早速、プログラミング言語の説明に移ろう。
　まずはプログラミング言語を作る、部品を２つ説明するよ。
　　・ Lexical Analyzer(字句解析器)
　　・ Parser　　(パーサ)　　の２つ。

　Lexical Analyzer(以下 Lexer）は、文字列を
　トークンと呼ばれる、ものの並びに変換するプログラムなんだ。
　トークンっていうのは意味のあるコードの最小単位のこと。
　例えば “ 'a = 2 + 3; “という文字列を読み込むと
　',a , = , 2 , + , 3 , ; ,
　というトークン列に変換する。



トークンには色々種類がある。
　分け方もたくさんある。分け方は言語に依存した部分なんだ。
　上の例だと、Symbol トークン、Number トークンとかがあるね。
　これは自分の好きなように名前を付けたりすればいいよ。

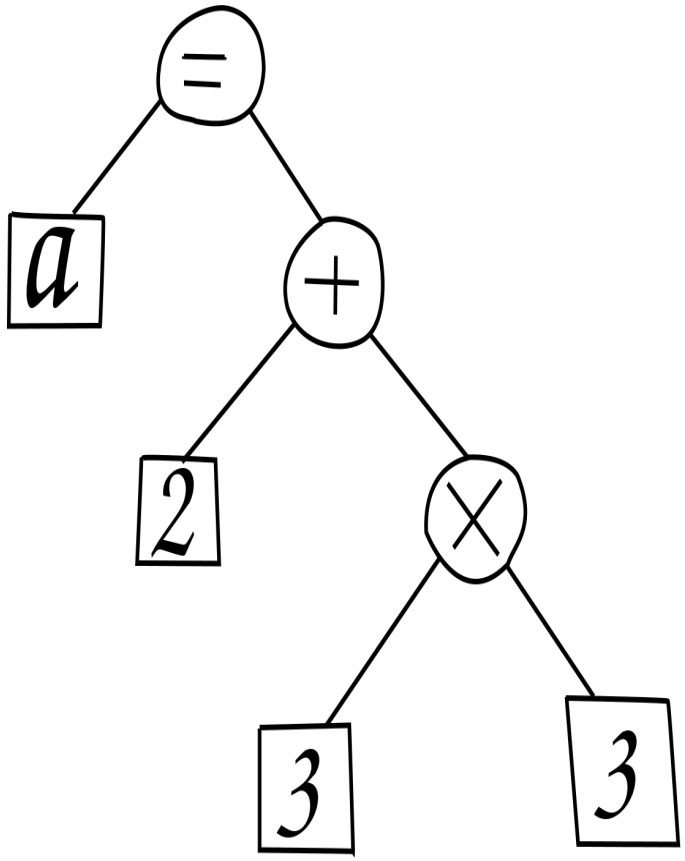
ちなみに上の例は僕の作っていた言語の前の仕様だったんだ。
　なかなかひどいことになっていたから、一回書きなおした。
　プログラミングをするときはいつもそうだけど、一回はすべて書きなおすとうまくいく。
　何がダメだったか、それを前提にはじめから書き直すといいものができる。

Parser について。

Parser はトークンの列を読み込んで構文木を構成する。

構文木(SyntaxTree)というのは、

構文解析の結果を木として表現したものでだいたいこんなもの。



プログラミング言語の多くは構文木を作る。

作るとすっごく楽だからね。

このテキストで作る、プログラミング言語も構文木を作ることにしよう。

たとえば a = 2 + 3*3 だったら

←こんな構文木を作ってみたり。

構文木はノードと呼ばれる、節目、葉で構成されてるんだ。

この例でいうと、「=」と「+」と「x」が節目ノードで「a」と「2」と「3」が葉ノード。

パーサーの実装についてだけど、これはすっごい色々ある。

大きく分けてトップダウン、ボトムアップとか、それこそパーサを研究している人たちもいるくらい。

その中から今回は再帰下降という手法を使うことにしてる。

それがどんなのかはソースをかきながら学ぼう。

なんで再帰下降パーサかっていうと、楽だからね。人が書けるくらいには。

ただ、あまり強力じゃない。

複雑な文法になっていくと、再帰下降ではパースできなくなることもある。

本当のところ、パーサは手書きされることはあまりないみたいなんだ。

パーサジェネレータ(Yacc とか)を使うのが一般的で、たとえば Ruby なんかは Yacc を使ってる。

でも、あえて今回は手書きで、再帰下降パーサを作ろう。

なんでかっていうと、勉強になるから。それに尽きると思う。

手書きした経験があると、パーサジェネレータのありがたみもわかるからね。

さて、なんだかんだと理論を述べるより、実装してみたほうが、楽しいし、
理解しやすいと僕は思うし、君もそうだと思う。
だから、早速作っていこう。
でも、いきなりプログラミング言語を作るのは大変だから、簡単な電卓をつくろう。
電卓から、プログラミング言語に行くのは簡単だからね。
変数や、関数を入れていったら……それはプログラミング言語になってるんだ。

僕らの作る、電卓はこんなものだ。

- 四則演算ができる。(足し算引き算、掛け算割り算の順序を知ってる。)
- （）を使える。（）の中は優先的に演算される。

次のページから、早速コードを書いていこう。

と、ソースコードを載せようと思っていたのだけど、
すべてを載せるにはここは狭すぎるね。
https://github.com/Tomoki/Calc からソースコードを持ってこよう。
基本的にこのテキストはソースを見ながら読むことを想定してるからね。
見るだけならダウンロードする必要すらないかもしれないね。
電卓については著作権を放棄しているから、好きにしていいよ。

参考までに、僕の作ったプログラミング言語"Umi"は
https://github.com/Tomoki/Umi にある。
これは今回作る電卓にかなり似た内部構造をしている。
変数とか、関数を実装する時には参考になると思う。

余談だけど、プログラミング言語を作る、一番のメリット、というか楽しみは
自分の言語に名前を付けられることだと思う。
名前は、どんな名前でもいい。ネタに走ってもいいし、何かの頭文字をとってもいい。
僕の場合は、ある女の子の名前を付けた。
大好きな、女の子のね。

Lexer の実装をしよう。

僕らの電卓が持ってるトークンについてまず考えてみよう。
　数字を表すトークン、+-*/のトークン、開き括弧,閉じ括弧トークン。
とりあえずこれくらいかな？
というわけで、まず TokenType.cs というファイルを作って、
　トークンの種類を列挙型で並べておこう。

```
public enum TokenType
{
		Plus,
		Minus,
		Asterisk,
		Slash,
		OpenParen,
		CloseParen,
		Number,
		EOF,
}
```

　こんな感じにね。
　EOFっていうのが増えてるね。これは文字列が終わったことを表すトークンで便利だから入れておこう。
　LexerはソースコードをTokenTypeの列に変換していくわけだね。
　でも一気に変換はせずに、一個づつ、変換して、Parserに渡して……を繰り返すことにしよう。
　言語の文法によっては2つ先読みしなくちゃいけなかったりするのだけど、
　今回は1個で大丈夫。

Lexerの本体のほうは、細かいところはソースを見てもらうとして、
本質だけ載せ……ようと思ったのだけど、ちょっと長いから次のページにしよう。
代わりにIO/LexReader.csについてすこし話しておこう。
　LexReaderは文字列を与えると、それをArrayListにして内部に保存する。
　外部からはLexReader.Peek(int n)として、ArrayListの中を見ることが出来て、
　LexReader.Read()でArrayListの最初を削除できる。
　なんでそんな周りくどいことをするかというと、わかりやすさもあるし、楽という理由もある。
　入力を一元管理することでエラーチェックを容易にしたり、一貫性を保ってるんだ。

```
public int Peek (int n)
{
	if (n >= code.Count) {
		return -1;
	}  else {
		return (int)code [n];
	}
}
```

Lexerの本質の部分。
詳しくはLexer.csを見て欲しい。コメントとかもちゃんと書いてあるから。

```
private void LexVisibles ()
{
	int c = reader.Peek ();
	switch (c) {
	case '+':
		reader.Read ();
		tokenType = TokenType.Plus;
		break;
	case '-':
		reader.Read ();
		tokenType = TokenType.Minus;
		break;
	case '*':
		reader.Read ();
		tokenType = TokenType.Asterisk;
		break;
	case '/':
		reader.Read ();
		tokenType = TokenType.Slash;
		break;
	case '(':
		reader.Read ();
		tokenType = TokenType.OpenParen;
		break;
	case ')':
		reader.Read ();
		tokenType = TokenType.CloseParen;
		break;
	default:
		if (char.IsDigit ((char)c)) {
			LexNumber ();
		}  else {
			throw new Exception (string.Format ("unknown char:{0}", (char)c));
		}
		break;
	}
}
```

switch-case文によって分岐してトークンをセットしている。
Lexerの仕事はこれだけ。

Parserの実装について。

再帰下降というのを理解する一番の方法は、まずはサンプルを書き写して、
実際に自分で紙の上でパースしてみること。
再帰下降パーサありきのやり方だけど、一番いい方法だと思う。
Parser.csを開いてみよう。色んなメソッドがあるけれど、とりあえず一つ見てみる。

```
private BaseNode ParseAdditiveExpression ()
{
	BaseNode node = ParseMultiplicativeExpression ();
	while ((tokenType == TokenType.Plus) ||
		 (tokenType == TokenType.Minus)) {
		TokenType op_type = tokenType;
		GetToken ();
		BaseNode right = ParseMultiplicativeExpression ();
		if (op_type == TokenType.Plus) {
			node = new PlusNode (node, right);
		}  else {
			node = new MinusNode (node, right);
		}
	}
	return node;
}
```

これは足し算、引き算を評価する部分。
この中でParseMultiplicativeExpressionが呼ばれているね。
これは掛け算、割り算を評価するメソッドで、ParseAdditiveExpressionと似た構造を持ってる。

```
private BaseNode ParseMultiplicativeExpression ()
{
	BaseNode node = ParseUnaryExpression ();
	while ((tokenType == TokenType.Asterisk) ||
		 (tokenType == TokenType.Slash)) {
		TokenType op_type = tokenType;
		GetToken ();
		BaseNode right = ParseUnaryExpression ();
		if (op_type == TokenType.Asterisk) {
			node = new MultiplyNode (right, node);
		}  else {
			node = new DivideNode (right, node);
		}
	}
	return node;
}
```

ParseMulticativeExpressionのなかではParseUnaryExpressionが呼ばれていて、
それは単項演算子をパースするメソッドなんだけど、細かいことはソースコードを見て欲しい。
重要なのは、これらの順番で
　足し算引き算のメソッド→掛け算割り算のメソッド
　　→単項演算子のメソッド→数値もしくはかっこのメソッド
　というふうに呼ばれていく。
　これが再帰下降と呼ばれる所以で演算順序を制御するポイントになっている。
　これが演算順序とどう関係があるかというと、
　たとえば1+2*(-4+5)という文字列をよんだときには、普通は
　　-4を[-4]に→([-4]+5)をする→2*[([-4]+5)]をする→1+[2*[([-4]+5)]]をするという順序で計算する。
　　どういうことかというと、
　　数値もしくはかっこの中→単項演算子→掛け算割り算→足し算引き算の順番で計算をしているんだ。
　　これはパーサのメソッドとちょうど逆向きになっている。
　　構文木をまた書こうと思ったのだけど、ちょっと面倒。
　　でも、Calcには構文木を表示する機能があるからそれを使うと、

```
Calc > 1+2*(-4+5)
[PlusNode]
┣┫-[NumberNode: 1]
┣┫-[MultiplyNode]
┣┫┣┫-[PlusNode]
┣┫┣┫┣┫-[NegativeNode]
┣┫┣┫┣┫┣┫-[NumberNode: 4]
┣┫┣┫┣┫-[NumberNode: 5]
┣┫┣┫-[NumberNode: 2]
 => [NumberNode: 3]
```

←こうなった。
すこしわかりづらい印象を受けるね。
でもまぁちゃんとわかる。
評価が遅いものが上にきてるのがわかればそれで十分。
今回構文木は下のほうから評価されていくから、下に評価が早いものがきてるんだ。

最後に=>で表されているのが計算結果。
構文木を作った後に、これを計算するのは一番上のノードのevalを呼び出す。
ノードの実装については次のページで。

ノードの実装について。

ノードは必ずevalメソッドを持ってる。

evalは下のノードのevalを呼んで……というふうに再帰的に最後までevalしていく。

PlusNodeのコードを見てみよう。

```
public class PlusNode:BaseNode
{
	private BaseNode Left, Right;

	public PlusNode (BaseNode left, BaseNode right)
	{
		Left = left;
		Right = right;
	}
	public override BaseNode eval ()
	{
		NumberNode EvaledLeft = (NumberNode)Left.eval ();
		NumberNode EvaledRight = (NumberNode)Right.eval ();
		return new NumberNode (EvaledLeft.Value + EvaledRight.Value);
	}
	//構文木表示用
	public override void print (int depth)
	{
		Console.WriteLine ("[PlusNode]");
		Console.Write ("|" + (new String ('|', depth)) + "|-");
		Left.print (depth + 1);
		Console.Write ("|" + (new String ('|', depth)) + "|-");
		Right.print (depth + 1);
	}
}
```

evalが呼ばれると、コンストラクタに渡されたLeftとRightをevalして、
それをもとに新しいNumberNodeを作る。

ほとんどのノードが同じように下のノードを評価していく。

電卓の場合、例外はNumberNode。

```
//再帰的なevalの最終地点
public override BaseNode eval ()
{
	return this;
}
```

自分を返して、ここでevalは終わり。

電卓からプログラミング言語へ。

ここまでで電卓はできた……ことにしよう。
実際はこのテキストを見ただけじゃ、まず出来ないと思うから、ソースを見て、それを写そう。
本当はプログラミング言語のことまで書けたら良かったのだけど。
　変数の実装とかはまず自分でチャレンジしてみるといいと思う。
　ギブアップしちゃったら、Umiのソースコードを参考にしてくれたら嬉しいな。

自分で最初から作るのが難しかったりしたら他の言語処理系をハックするのもいいと思う。
何にせよ、君がプログラミング言語をブラックボックスとして扱うのではなくて、
　もうすこしフレンドリィなものとして見れるようになったなら、このテキストは意味があったんだと思う。

参考になりそうな文献

http://i.loveruby.net/ja/rhg/book/ Ruby Hacking Guide.Rubyの処理系についてのテキスト。
http://kmaebashi.com/programmer/devlang/index.html yacc/lexでプログラミング言語を作るテキスト。
http://www.geocities.jp/takt0_h/cyan/index.html cyanというプログラミング言語。
C#で書かれているが、すこし構造が違う。