Text-Based Math Calculator Pt3: Parsing

Picking up from where we left off...

- Last time, we created a complete and working lexer to break up our user's inputs from raw string data into sequential tokens for use.
- So we have code make these tokens but what do we exactly do with them to turn those tokens into a math result?
- ***The answer***: **<u>Parsing.</u>**

```
<postal-address> ::= <name-part> <street-address> <zip-part>

     <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL> | <personal-part> <name-part>

 <personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

      <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
    <opt-apt-num> ::= "Apt" <apt-num> | ""
```

Named after John Backus (1924 - 2007) and Peter Naur (1928 - 2016).

← **Note**: This example is MISSING definition of the tokens!

Intro to Parsing Theory

- Parsing is the application of grammar onto a set of tokens.
- Special math notation called *Formalized Grammars* used to well… *formalize* a grammatical behavior for some kind of input.
- There are several common ones: **[Extended] Backus-Naur Form or [E]BNF** for short.
- Parsing Expression Grammar [***PEG*** for short].
- My style: ***Combo of PEG & EBNF***

Backus-Naur Form [written in itself!]

```
 <syntax>         ::= <rule> | <rule> <syntax>
 <rule>           ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::="
<opt-whitespace> <expression> <line-end>
 <opt-whitespace> ::= " " <opt-whitespace> | ""
 <expression>     ::= <list> | <list> <opt-whitespace> "|" <opt-whitespace>
<expression>
 <line-end>       ::= <opt-whitespace> <EOL> | <line-end> <line-end>
 <list>           ::= <term> | <term> <opt-whitespace> <list>
 <term>           ::= <literal> | "<" <rule-name> ">"
 <literal>        ::= '"' <text1> '"' | "'" <text2> "'"
 <text1>          ::= "" | <character1> <text1>
 <text2>          ::= "" | <character2> <text2>
 <character>      ::= <letter> | <digit> | <symbol>
 <letter>         ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
"K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
| "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
"l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y"
| "z"
 <digit>          ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
 <symbol>         ::= "|" | " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" |
"+" | "," | "-" | "." | "/" | ":" | ";" | ">" | "=" | "<" | "?" | "@" | "[" | "\"
| "]" | "^" | "_" | "`" | "{" | "}" | "~"
 <character1>     ::= <character> | "'"
 <character2>     ::= <character> | '"'
 <rule-name>      ::= <letter> | <rule-name> <rule-char>
 <rule-char>      ::= <letter> | <digit> | "-"
```

Extended Backus-Naur Form

- Much simpler form of Backus-Naur Form
- Introduces additional notation to simplify grammar definitions.
- , Commas – sequence of rules [can be omitted as its implied].
- [] square brackets – optional rules/token sets.
- {} curly brackets – repetition of rules/token sets.

Postal Address but in EBNF

```
address          ::= street_address "," city_state_zip .
street_address   ::= street_number street_name street_type .
street_number    ::= digit+ .
street_name      ::= word [ "-" street_name ] .
street_type      ::= word .
city_state_zip   ::= city state ZIP_code .
city             ::= word [ "-" city ] .
state            ::= uppercase_letter uppercase_letter .
ZIP_code         ::= digit{5} [ "-" digit{4} ] .
word             ::= letter+ .
letter           ::= uppercase_letter / lowercase_letter .
uppercase_letter ::= "A" / "B" / "C" / ... / "Z" .
lowercase_letter ::= "a" / "b" / "c" / ... / "z" .
digit            ::= "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" .
```

Parsing Expression Grammar

- Builds off Backus-Naur Form
- Uses ‘?’ for optional rules/token sets [but [] is also used informally].
- *Kleene Star* ‘*’ – means “zero-or-more” repetition.
- *Kleene Plus* ‘+’ – means “one-or-more” repetition.
- AND ‘&’ and NOT ‘!’ predicates for peeking/checking the next token without getting the next token.
- <u>We will use a variation of this for our calculator!</u>

Translating PEMDAS to PEG

- **PEMDAS** = Parentheses, Exponent, Multiplication, Division, Addition, and Subtraction.
- Addition and Subtraction have the lowest priority in the hierarchy in any given equation.
- **Two ways to parse a grammar**:
  - **<u>Top-Down</u>** → start from highest rule going down to the lowest.
  - **<u>Bottom-Up</u>** → start from lowest rule and work way up to the highest.
- **PEG is designed for Top-Down parsing so we will be doing top-down parsing.**
- Where do we start? **The definition of a value**.
  - Why? Because the value, whether a number or variable, has the highest priority in any expression.
- **For the code, add `#include <math.h>` at the top.**
  - `We're gonna need it.`

PEMDAS in PEG/EBNF Form

- Expr = AddExpr .
- AddExpr = MulExpr *('+' | '-' MulExpr) .
- MulExpr = PowExpr *('*' | '/' PowExpr) .
- PowExpr = TermExpr *('^' TermExpr) .
- TermExpr = number | 'e' | 'pi' .
- number = +[0-9] .
- How do we do parentheses?
  - **TermExpr = number | 'e' | 'pi' | '(' Expr ')' | '[' Expr ']' .**
  - Notice that with parentheses, we go back up to the **Expr** rule!
  - This is a form of top-down parsing called **Recursive Descent**.
  - **Kevin's Advice**: When making a top-down parser, code it "bottom-up", start with the lowest rule and work your way up.
- What about negative starting numbers like: '-2 + 4' ?
  - Needs a ***UnaryExpr*** but where does that go in the grammar?
- Food for thought: Where do functions go?

Parsing Terms

- All we're doing is returning the numerical value of a constant, whether it's named or numerical.
- We copy the current token.
- Have the Lexer move onto the next token.
- Then return a value based on what we currently got.
- We'll use infinity as our error value.

```c
/// TermExpr = number | 'e' | 'pi' | '(' Expr ')' | '[' Expr ']' .
/// number = [0-9]+ [ '.' [0-9]+ ] | [ [0-9]+ ] '.' [0-9]+ .
double parse_term(struct Lexer *lexer) {
	/// copy current token.
	struct Token const token = lexer->curr_token;
	lexer_get_token(lexer);
	switch( token.type ) {
		case TokenNum: {
			return token.value;
		}
		case TokenName: {
			char const *name = &lexer->input[token.start];
			if( !strncmp(name, "pi", token.num_chars) ) {
				return M_PI;
			} else if( !strncmp(name, "e", token.num_chars) ) {
				return M_E;
			}
			break;
			/// more code here later.
		}
		case TokenLParen: case TokenLBracket: {
			enum TokenType end_type = (token.type==TokenLParen)? TokenRParen :
			TokenRBracket;
			double result = parse_expr(lexer);
			if( lexer->curr_token.type==end_type ) {
				lexer_get_token(lexer);
				return result;
			}
			break;
		}
	}
	return INFINITY;
}
```

Parsing Negatives

- Instead of copying the current token, we only need to check if the lexer state's current token is a minus sign.
- If it is, then we accomplish the Kleene Star repetition by using recursion (calling the grammar rule function itself!).

```c
/// UnaryExpr = *( '-' ) TermExpr .
double parse_unary(struct Lexer *lexer) {
	if( lexer->curr_token.type==TokenMinus ) {
		lexer_get_token(lexer);
		return -parse_unary(lexer);
	}
	return parse_term(lexer);
}
```

Power Parsing 💪

- Instead of copying the current token, we get the address of the Lexer's current token and then access the address through a pointer.

- The advantage using a pointer is we don't have to copy the type of the current token during the loop as the get token function changes the current token's value every time.

```
/// PowExpr = UnaryExpr *( '^' UnaryExpr ).
double parse_pow(struct Lexer *lexer) {
	double result = parse_unary(lexer);
	struct Token const *curr_tok = &lexer->
	curr_token;
	while( curr_tok->type==TokenCarot ) {
		lexer_get_token(lexer);
		result = pow(result, parse_unary(lexer));
	}
	return result;
}
```

Multiplicative Parsing

```c
/// MulExpr = PowExpr *( ('*' | '/') PowExpr ) .
double parse_mul(struct Lexer *lexer) {
	double result = parse_pow(lexer);
	struct Token const *curr_tok = &lexer->curr_token;
	while( curr_tok->type==TokenStar || curr_tok->type==TokenSlash ) {
		enum TokenType const tt = curr_tok->type;
		lexer_get_token(lexer);
		switch( tt ) {
			case TokenStar:
				result *= parse_pow(lexer); break;
			case TokenSlash:
				result /= parse_pow(lexer); break;
		}
	}
	return result;
}
```

Additive Parsing

```c
/// AddExpr = MulExpr *( ('+' | '-') MulExpr ) .
double parse_add(struct Lexer *lexer) {
	double result = parse_mul(lexer);
	struct Token const *curr_tok = &lexer->
	curr_token;
	while( curr_tok->type==TokenPlus || curr_tok
	->type==TokenMinus ) {
		enum TokenType const tt = curr_tok->type;
		lexer_get_token(lexer);
		switch( tt ) {
			case TokenPlus:
				result += parse_mul(lexer); break;
			case TokenMinus:
				result -= parse_mul(lexer); break;
		}
	}
	return result;
}
```

One Function to Rule Them All

```c
/// Placed ABOVE all the parse_* functions.
double parse_add(struct Lexer *lexer);
double parse_mul(struct Lexer *lexer);
double parse_pow(struct Lexer *lexer);
double parse_unary(struct Lexer *lexer);
double parse_term(struct Lexer *lexer);

/// Expr = AddExpr .
double parse_expr(struct Lexer *lexer) {
    return parse_add(lexer);
}
```

Quick Review

```c
/// Placed ABOVE all the parse_* functions.
double parse_add(struct Lexer *lexer);
double parse_mul(struct Lexer *lexer);
double parse_pow(struct Lexer *lexer);
double parse_unary(struct Lexer *lexer);
double parse_term(struct Lexer *lexer);

/// Expr = AddExpr .
double parse_expr(struct Lexer *lexer) {…}

/// AddExpr = MulExpr *( ('+' | '-') MulExpr ) .
double parse_add(struct Lexer *lexer) {…}

/// MulExpr = PowExpr *( ('*' | '/') PowExpr ) .
double parse_mul(struct Lexer *lexer) {…}

/// PowExpr = UnaryExpr *( '^' UnaryExpr ).
double parse_pow(struct Lexer *lexer) {…}

/// UnaryExpr = *( '-' ) TermExpr .
double parse_unary(struct Lexer *lexer) {…}

/// TermExpr = number | 'e' | 'pi' | '(' Expr ')' | '[' Expr ']' .
/// number = [0-9]+ [ '.' [0-9]+ ] | [ [0-9]+ ] '.' [0-9]+ .
double parse_term(struct Lexer *lexer) {…}


int main(void) {
    enum{ LINE_SIZE = 2000 };
    char line[LINE_SIZE + 1] = {0};
    for(;;) { /// infinite program loop.
```

Unfinished Business

```c
int main(void) {
	enum{ LINE_SIZE = 2000 };
	char line[LINE_SIZE + 1] = {0};
	for(;;) { /// infinite program loop.
		puts("please enter an equation or 'q' to quit.");
		if( fgets(line, LINE_SIZE, stdin)==NULL ) {
			puts("fgets:: bad input");
			break;
		} else if( line[0]=='q' || line[0]=='Q' ) {
			puts("calculator program exiting.");
			break;
		}
		size_t const equation_len = strlen(line);
		struct Lexer lexer = {
			.input = line,
			.input_len = equation_len,
		};
		line[equation_len-1] = 0; /// remove stupid newline at end of equation input.
		lexer_get_token(&lexer);
		printf("result of equation '%s' = %f\n", line, parse_expr(&lexer));
	}
}
```

What about Math Functions though?

- What's a calculator having pi and Euler's constant but no functions to operate them with? Like ln(x), sin(x), etc.
- **Problem though**: Enforce parentheses or no?
- Sometimes we see, in class, equations like '`sin x`'.
- What if a user gives us input like "`ln e^5`".
  - **<u>Do we calculate it as `ln(e^5)` or `ln(e)^5`?</u>**
  - This is a concept in parsing called **<u>ambiguity</u>**.
- For our purposes we will allow function calls to have optional parentheses but this requires calculating adjustments depending whether there are parentheses or not.
- **Solution:**
  - **With parentheses → we parse the math function input as a term expression [this will treat the function input as an expression with parentheses].**
  - **Without parentheses → we parse the math function input as a power expression.**
- With the solution, the resolve the ambiguity of `ln e^5` by interpreting it as `ln(e^5)`.
- **<u>Note</u>**: With the solution we're going with, an input like `sin pi/2` will NOT be interpreted `sin(pi/2)` but as `sin(pi) / 2`.
- If this is undesired, you can relax the rule by going up to the multiplication or addition rule but it's not wise to resolve ambiguity by being too relaxed!

Extending Names Section for Math Functions

Add some of your own!

```c
/// inside function: 'parse_term'::
case TokenName: {
	bool has_parens = lexer->curr_token.type==TokenLParen || lexer->
	curr_token.type==TokenLBracket;
	char const *name = &lexer->input[token.start];
	if( !strncmp(name, "pi", token.num_chars) ) {
		return M_PI;
	} else if( !strncmp(name, "e", token.num_chars) ) {
		return M_E;
	} else if( !strncmp(name, "sin", token.num_chars) ) {
		return sin(has_parens? parse_term(lexer) : parse_pow(lexer));
	} else if( !strncmp(name, "ln", token.num_chars) ) {
		return log(has_parens? parse_term(lexer) : parse_pow(lexer));
	} else if( !strncmp(name, "arcsin", token.num_chars) ) {
		return asin(has_parens? parse_term(lexer) : parse_pow(lexer));
	} else if( !strncmp(name, "myfunctionhere", token.num_chars) ) {
		double result = has_parens? parse_term(lexer) : parse_pow(
		lexer);
		/// do something with result.
		return result;
	}
}
```

End of the
Text-Input Math Calculator Workshop

Next Workshop: Register Virtual Machine
(for CSC230)