CS 6015: Software Engineering

Spring 2024

Lecture 10: Parsing (Project Related)

This Week

- Documentation
- Let Binding (Project related)
- Parsing (Project related)

Next Week

- Parsing cont.
- Power of variables
- Libraries

Interpreter Command Line

```
$ ./msdscript --interp
_let x = (1 + (2))
_in  x * 3
9
$
```

Parsing

Parsing is the task of turning text into **`Expr`** objects

```
_let x = (1 + (2))
_in  x * 3
```

⬇

```
new Let("x",
        new Add(new Num(1), new Num(2)),
        new Mult(new Var("x"), new Num(3)));
```

Parsing *does not* imply interpreting, but it’s a good first step

Data Analysis for Parsing

Output: **Expr** ✓

Input: stream of characters

A stream of characters is either

- an empty stream
- a character followed by a stream of characters

```
_let x = (1 + (2))
_in  x * y
```

Handle one character at a time... not all that much help

Parsing Recipes

There's a whole big space of recipes for parsing

LALR(1), LL(k), PEG, GLR, SGLR, table-driven, recursive descent...

Parsing Recipes

There's a whole big space of recipes for parsing

LALR(1), LL(k), PEG, GLR, SGLR, table-driven, recursive descent...

MSDscript will be a compromise between nice-to-read and easy-to-parse

Parsing Anti-Pattern

First idea you may have: divide and conquer

```
parse_str("..... * .....")

= new Mult(parse_str("....."),
           parse_str("....."))
```

... does not work well

```
(1 * 3) * 2 + _let x = 1+2 _in 3*4
```

We'll stick to the stream-of-characters view

Parsing Numbers

$$\langle \text{expr} \rangle \;=\; \langle \text{number} \rangle$$

Parsing Numbers

$\langle\text{expr}\rangle$ = $\langle\text{number}\rangle$

sequence of digits: 0... 9

```
1352
```

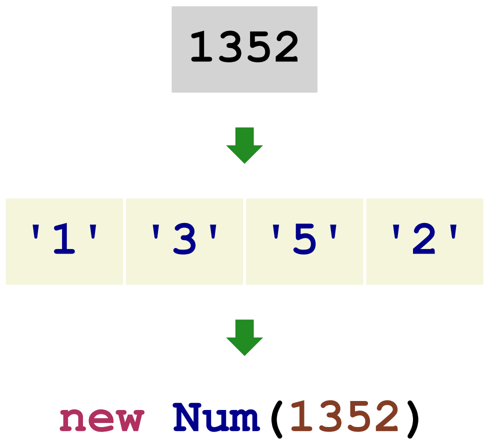
⬇

```
new Num(1352)
```

Parsing Numbers

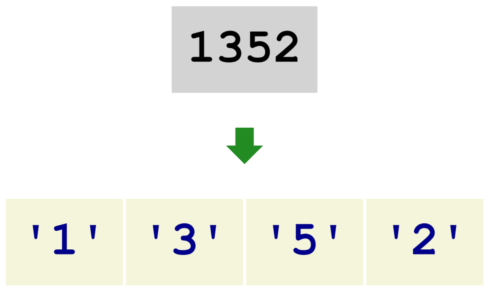
⟨expr⟩ = ⟨number⟩

sequence of digits: 0... 9



Parsing Numbers

⟨expr⟩ = ⟨number⟩ sequence of digits: 0… 9



```
in.get(); // = '1'
in.get(); // = '3'
in.get(); // = '5'
in.get(); // = '2'
in.get(); // = EOF
```

Parsing Numbers

First try:

```
Expr *parse_num(std::istream &in) {
   int n = 0;
   while (1) {
      int c = in.get();
      if (isdigit(c))
        n = n*10 + (c - '0');
      else
        break;
   }
   return new Num(n);
}
```

Parsing Numbers

First try:

```
Expr *parse_num(std::istream &in) {
    int n = 0;
    while
        in
        if

        el

    }
    retur
}
```

parse.cpp

```
....

// just for demo purposes
int main() {
  Expr *n = parse_num(std::cin);

  std::cout << n->to_pretty_string();
  std::cout << "\n";

  return 0;
}
```

Parsing Numbers

First try:

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break
  }
  return new
}
```

```
$ c++ parse.cpp expr.o cmdline.o
$ ./a.out
123
123
$ ./a.out
-123
0
```

Parsing Numbers

First try:

```
Expr *parse_num(std::istream &in) {
   int n = 0;
   while (1) {
      int c = in.get();
      if (isdigit(c))
        n = n*10 + (c - '0');
      else
        break;
   }
   return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    in.get(); // consume '-'
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool nega

  if (in.peek() == '-') {
    negative = true;
    in.get(); // consume '-'
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Like `in.get()`, but leaves character in stream

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    in.get(); // consume '-'
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

better to check!

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while
    int
    if
      n
    els
      b
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

```
static void consume(std::istream &in, int expect) {
  int c = in.get();
  if (c != expect)
    throw std::runtime_error("consume mismatch");
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.get();
    if (isdigit(c))
      n = n*10 + (c - '0');
    else
      break;
  }

  if (negative)
    n = -n;

  return new Num(
}
```

discarding `c` means we can’t tell

`-123`

from

`-123*`

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.peek();
    if (isdigit(c)) {
      consume(in, c);
      n = n*10 + (c - '0');
    } else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.peek();
    if (isdigit(c)) {
      consume(in, c);
      n = n*10 + (c - '0');
    } else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

General parsing strategy: peek to decide, then maybe consume

Parsing Numbers

```
Expr *parse_num(std::istream &in) {
  int n = 0;
  bool negative = false;

  if (in.peek() == '-') {
    negative = true;
    consume(in, '-');
  }

  while (1) {
    int c = in.peek();
    if (isdigit(c)) {
      consume(in, c);
      n = n*10 + (c - '0');
    } else
      break;
  }

  if (negative)
    n = -n;

  return new Num(n);
}
```

```
$ ./a.out
-123
-123
$ ./a.out
 -123
0
```

Ignoring Whitespace

```cpp
static void skip_whitespace(std::istream &in) {
  while (1) {
    int c = in.peek();
    if (!isspace(c))
      break;
    consume(in, c);
  }
}
```

Parsing Expressions

```
Expr *parse_expr(std::istream &in) {
  skip_whitespace(in);
  return parse_num(in);
}

int main() {
  while (1) {
    Expr *e = parse_expr(std::cin);

    e->pretty_print(std::cout);
    std::cout << "\n";

    skip_whitespace(std::cin);
    if (std::cin.eof())
      break;
  }

  return 0;
}
```

Parsing Expressions

```
Expr *parse_expr(std::istream &in) {
  skip_whitespace(in);
  return parse_num(in);
}

int main() {
  while (1) {
    Expr *e = parse_expr(std::cin);

    e->pretty_print(std::cout);
    std::cout << "\n";

    skip_whitespace(std::cin);
    if (std::cin.eof())
      break;
  }

  return 0;
}
```

```
$ ./a.out
123
123
-123
-123
x
0
0
0
0
```

Parsing Expressions

```
Expr *parse_expr(std::istream &in) {
  skip_whitespace(in);

  int c = in.peek();
  if ((c == '-') || isdigit(c))
    return parse_num(in);
  else {
    consume(in, c);
    throw std::runtime_error("invalid input");
  }
}
```

More Expressions

So, far, our parser supports just numbers:

```
123
-456
0
```

Let’s add support for parentheses:

```
(123)
(-456)
( (0 ))
```

Numbers and Parentheses

$$
\begin{array}{rcl}
\langle\text{expr}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)}
\end{array}
$$

Parentheses are not in **`Expr`**, because the **`Expr`** tree structure already handles grouping: it's ***abstract syntax***

The parser deals with characters in text, which is ***concrete syntax***

A grammar can be for abstract syntax or concrete syntax

Numbers and Parentheses

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
& | & \texttt{(}\ \langle \text{expr} \rangle\ \texttt{)}
\end{array}
$$

In concrete syntax, **`gray`** are literal characters to get

Whitespace can appear between any two things in the grammar

Numbers and Parentheses

$$\begin{tikzcd} \langle\text{expr}\rangle & = & \langle\text{number}\rangle \\ & \mid & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \arrow[ull, bend right] \end{tikzcd}$$

When `parse_expr` sees `(`, it should call itself

Parsing Expressions

```
Expr *parse_expr(std::istream &in) {
  skip_whitespace(in);

  int c = in.peek();
  if ((c == '-') || isdigit(c))
    return parse_num(in);
  else if (c == '(') {
    consume(in, '(');
    Expr *e = parse_expr(in);
    skip_whitespace(in);
    c = in.get();
    if (c != ')')
      throw std::runtime_error("missing close parenthesis");
    return e;
  } else {
    consume(in, c);
    throw std::runtime_error("invalid input");
  }
}
```

Parsing Addition

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
& | & \texttt{(}\ \langle \text{expr} \rangle\ \texttt{)} \\
& | & \langle \text{expr} \rangle\ \texttt{+}\ \langle \text{expr} \rangle
\end{array}
$$

Parsing Addition

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(}\ \langle \text{expr} \rangle\ \texttt{)} \\
 & | & \langle \text{expr} \rangle\ \texttt{+}\ \langle \text{expr} \rangle
\end{array}
$$

```
1 + 2 + 3
```

Parsing Addition

$$
\begin{array}{rcl}
\langle\text{expr}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \\
 & | & \langle\text{expr}\rangle\ \texttt{+}\ \langle\text{expr}\rangle
\end{array}
$$

```
1 + 2  +  3
```

Parsing Addition

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(}\ \langle \text{expr} \rangle\ \texttt{)} \\
 & | & \langle \text{expr} \rangle\ \texttt{+}\ \langle \text{expr} \rangle
\end{array}
$$

$$\boxed{\texttt{1}}\ \boxed{\texttt{+}}\ \boxed{\texttt{2 + 3}}$$

Parsing Addition

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(}\ \langle \text{expr} \rangle\ \texttt{)} \\
 & | & \langle \text{expr} \rangle\ \texttt{+}\ \langle \text{expr} \rangle
\end{array}
$$

Disallow immediate `+` here

```
1  +  2 + 3
```

Parsing Addition

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
 & | & \langle \text{addend} \rangle \ \texttt{+} \ \langle \text{expr} \rangle \\
\\
\langle \text{addend} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)}
\end{array}
$$

`1 + 2 + 3`

Parsing Addition

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)}
\end{array}
$$

$$\underbrace{\texttt{1 + 2}}_{\text{can't be } \langle\text{addend}\rangle}\texttt{ + 3}$$

Parsing Addition

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)}
\end{array}
$$

$$\texttt{1 + }\underbrace{\texttt{2 + 3}}_{\langle\text{expr}\rangle}$$

Parsing Addition

$$\begin{array}{lcl}
\langle \text{expr} \rangle & = & \langle \text{addend} \rangle \\
 & | & \langle \text{addend} \rangle \; \texttt{+} \; \langle \text{expr} \rangle \\
 & & \\
\langle \text{addend} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(} \; \langle \text{expr} \rangle \; \texttt{)}
\end{array}$$

Parsing Addition

```
Expr *parse_addend(std::istream &in) {
  skip_whitespace(in);

  int c = in.peek();
  if ((c == '-') || isdigit(c))
    return parse_num(in);
  else if (c == '(') {
    consume(in, '(');
    Expr *e = parse_expr(in);
    skip_whitespace(in);
    c = in.get();
    if (c != ')')
      throw std::runtime_error("missing close parenthesis");
    return e;
  } else {
    consume(in, c);
    throw std::runtime_error("invalid input");
  }
}
```

⟨expr⟩	=	⟨addend⟩
	|	⟨addend⟩ + ⟨expr⟩
⟨addend⟩	=	⟨number⟩
	|	(⟨expr⟩)

Parsing Addition

```
Expr *parse_addend(std::istream &in) {
  skip_whitespace(in);

    c = in.peek();
  if ((c == '-') || isdigit(c))
    return parse_num(in);
  else if (c == '(') {
    consume(in, '(');
    Expr *e = parse_expr(in);
    skip_whitespace(in);
    c = in.get();
    if (c != ')')
      throw std::runtime_error("missing close parenthesis");
    return e;
  } else {
    consume(in, c);
    throw std::runtime_error("invalid input");
  }
}
```

Changed the function name

$\langle$expr$\rangle$	=	$\langle$addend$\rangle$
	|	$\langle$addend$\rangle$ + $\langle$expr$\rangle$
$\langle$addend$\rangle$	=	$\langle$number$\rangle$
	|	($\langle$expr$\rangle$)

Parsing Addition

```
Expr *parse_addend(std::istream &in) {
  skip_whitespace(in);

  int c = in.peek();
  if ((c == '-') || isdigit(c))
    return parse_num(in);
  else if (c == '(') {
    consume(in, '(');
    Expr *e = parse_expr(in);
    skip_whitespace(in);
    c = in.get();
    if (c != ')')
      throw std::runtime_error("missing close parenthesis");
    return e;
  } else {
    consume(in, c);
    throw std::runtime_error("invalid input");
  }
}
```

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)}
\end{array}
$$

Still call `parse_expr` to parse parenthesized

Parsing Addition

```
static Expr *parse_expr(std::istream &in) {
  Expr *e;

  e = parse_addend(in);

  skip_whitespace(in);

  int c = in.peek();
  if (c == '+') {
    consume(in, '+');
    Expr *rhs = parse_expr(in);
    return new Add(e, rhs);
  } else
    return e;
}
```

$\langle \text{expr} \rangle$	=	$\langle \text{addend} \rangle$
	|	$\langle \text{addend} \rangle$ + $\langle \text{expr} \rangle$
$\langle \text{addend} \rangle$	=	$\langle \text{number} \rangle$
	|	($\langle \text{expr} \rangle$)

Parsing Multiplication

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
```

```
1 * 2 + 3 * 4
```

Parsing Multiplication

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle\ \texttt{*}\ \langle\text{addend}\rangle \\
\\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)}
\end{array}
$$

$$\texttt{1 * }\underbrace{\texttt{2 + 3 * 4}}_{\text{can't be } \langle\text{addend}\rangle}$$

Parsing Multiplication

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle \ \texttt{+} \ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle \ \texttt{*} \ \langle\text{addend}\rangle \\
\\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(} \ \langle\text{expr}\rangle \ \texttt{)}
\end{array}
$$

$$\underbrace{\texttt{1 * 2}}_{\langle\text{addend}\rangle} \ \texttt{+} \ \underbrace{\texttt{3 * 4}}_{\langle\text{addend}\rangle}$$

Parsing Multiplication

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle \ \texttt{+} \ \langle\text{expr}\rangle \\
 & & \\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle \ \texttt{*} \ \langle\text{addend}\rangle \\
 & & \\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(} \ \langle\text{expr}\rangle \ \texttt{)}
\end{array}
$$

```
1 * (2 + 3 * 4)
```

Parsing Multiplication

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle \; \texttt{+} \; \langle\text{expr}\rangle \\
 & & \\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle \; \texttt{*} \; \langle\text{addend}\rangle \\
 & & \\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(} \; \langle\text{expr}\rangle \; \texttt{)}
\end{array}
$$

$$\texttt{1 * }\underbrace{\underbrace{\texttt{(}\underbrace{\texttt{2 + 3 * 4}}_{\langle\text{expr}\rangle}\texttt{)}}_{\langle\text{multicand}\rangle}}_{\langle\text{addend}\rangle}$$

Parsing Multiplication

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & = & \langle \text{addend} \rangle \\
 & | & \langle \text{addend} \rangle \ \texttt{+} \ \langle \text{expr} \rangle \\[1em]
\langle \text{addend} \rangle & = & \langle \text{multicand} \rangle \\
 & | & \langle \text{multicand} \rangle \ \texttt{*} \ \langle \text{addend} \rangle \\[1em]
\langle \text{multicand} \rangle & = & \langle \text{number} \rangle \\
 & | & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)}
\end{array}
$$

- old `parse_addend` becomes `parse_multicand`
- new `parse_addend` calls `parse_multicand` and `parse_addend`

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Parsing Let

How about variables and `_let`?

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle\ \texttt{*}\ \langle\text{addend}\rangle \\
\\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \\
 & | & \langle\text{variable}\rangle \\
 & | & \texttt{\_let}\ \langle\text{variable}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{\_in}\ \langle\text{expr}\rangle
\end{array}
$$

Starts with `-` or `isdigit`

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )      Starts with (
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Parsing Let

How about variables and `_let`?

⟨expr⟩	=	⟨addend⟩
	|	⟨addend⟩ `+` ⟨expr⟩
⟨addend⟩	=	⟨multicand⟩
	|	⟨multicand⟩ `*` ⟨addend⟩
⟨multicand⟩	=	⟨number⟩
	|	`(` ⟨expr⟩ `)`
	|	⟨variable⟩
	|	`_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩

Let’s say only ASCII letters: `a`- `z` and `A`- `Z`

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Starts with `isalpha`

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Starts with `_`

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Should not allow immediate `_let`...

Parsing Let

How about variables and `_let`?

```
⟨expr⟩       =  ⟨addend⟩
             |  ⟨addend⟩ + ⟨expr⟩

⟨addend⟩     =  ⟨multicand⟩
             |  ⟨multicand⟩ * ⟨addend⟩

⟨multicand⟩  =  ⟨number⟩
             |  ( ⟨expr⟩ )
             |  ⟨variable⟩
             |  _let ⟨variable⟩ = ⟨expr⟩ _in ⟨expr⟩
```

Should not allow immediate `_let`...

... but `parse_expr` will consume `*`, anyway

Parsing Let

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & = & \langle\text{addend}\rangle \\
 & | & \langle\text{addend}\rangle\ \texttt{+}\ \langle\text{expr}\rangle \\
\\
\langle\text{addend}\rangle & = & \langle\text{multicand}\rangle \\
 & | & \langle\text{multicand}\rangle\ \texttt{*}\ \langle\text{addend}\rangle \\
\\
\langle\text{multicand}\rangle & = & \langle\text{number}\rangle \\
 & | & \texttt{(}\ \langle\text{expr}\rangle\ \texttt{)} \\
 & | & \langle\text{variable}\rangle \\
 & | & \texttt{\_let}\ \langle\text{variable}\rangle\ \texttt{=}\ \langle\text{expr}\rangle\ \texttt{\_in}\ \langle\text{expr}\rangle
\end{array}
$$

`parse_var` and `parse_let` helpers are a good idea

`parse_keyword` helper is also a good idea