SPECIFYING SYNTAX

Programming languages must be very well defined — there’s no room for ambiguity.

Language designers must use formal syntactic and semantic notation to specify the rules of a language.

In this lecture, we will focus on how syntax is specified.
We know from the previous lecture that the front-end of the compiler has three main phases:

- Scanning
- Parsing
- Semantic Analysis

Syntax Verification
SPECIFYING SYNTAX

• Scanning
  • Identifies the valid tokens, the basic building blocks, within a program.

• Parsing
  • Identifies the valid patterns of tokens, or constructs.

So how do we specify what a valid token is? Or what constitutes a valid construct?
REGULAR EXPRESSIONS

Tokens can be constructed from regular characters using just three rules:

1. Concatenation.
2. Alternation (choice among a finite set of alternatives).

Any set of strings that can be defined by these three rules is a regular set. Regular sets are generated by regular expressions.
Formally, all of the following are valid regular expressions (let $R$ and $S$ be regular expressions and let $\Sigma$ be a finite set of symbols):

- The empty set.
- The set containing the empty string $\varepsilon$.
- The set containing a single literal character $\alpha$ from the alphabet $\Sigma$.
- Concatenation: $RS$ is the set of strings obtained by concatenation of one string from $R$ with a string from $S$.
- Alternation: $R | S$ describes the union of $R$ and $S$.
- Kleene Closure: $R^*$ is the set of strings that can be obtained by concatenating any number of strings from $R$. 
REGULAR EXPRESSIONS

You can either use parentheses to avoid ambiguity or assume Kleene star has the highest priority, followed by concatenation then alternation.

Examples:

• \(a^* = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}\)

• \(a | b^* = \{\varepsilon, a, b, bb, bbb, bbbb, \ldots\}\)

• \((ab)^* = \{\varepsilon, ab, abab, ababab, abababab, \ldots\}\)

• \((a | b)^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}\)
Let’s look at a more practical example. Say we want to write a regular expression to identify valid numbers.

Some things to consider:

• Numbers can be any number of digits long, but must not start with 0.
• Numbers can be positive or negative.
• Numbers can be integers or real.
• Numbers can be represented by scientific notation (i.e. 2.9e8).
REGULAR EXPRESSIONS

number \rightarrow integer \mid real
integer \rightarrow non\_zero\_digit digit*
real \rightarrow integer exponent \mid decimal (exponent \mid \epsilon)
decimal \rightarrow (non\_zero\_digit digit*)\*(. digit \mid digit .) digit*
exponent \rightarrow (e \mid E) (\pm \mid \epsilon) integer
digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
non\_zero\_digit \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
So our number tokens are well-defined by the *number* symbol, which makes use of the other symbols to build larger expressions. Any valid pattern generated by expanding out the *number* symbol is a valid number. Note: while our rules build upon one another, no symbol is defined in terms of itself, even indirectly.

number $\rightarrow$ integer $|$ real
integer $\rightarrow$ non_zero_digit digit*
real $\rightarrow$ integer exponent $|$ decimal (exponent $|$ $\varepsilon$)
decimal $\rightarrow$ (non_zero_digit digit*)$^*$ (. digit $|$ digit . ) digit*
exponent $\rightarrow$ (e $|$ E) (+ $|$ - $|$ $\varepsilon$) integer
digit $\rightarrow$ 0 $|$ 1 $|$ 2 $|$ 3 $|$ 4 $|$ 5 $|$ 6 $|$ 7 $|$ 8 $|$ 9
non_zero_digit $\rightarrow$ 1 $|$ 2 $|$ 3 $|$ 4 $|$ 5 $|$ 6 $|$ 7 $|$ 8 $|$ 9
We can completely define our tokens in terms of regular expressions, but more complicated constructs necessitate the ability to self-reference.

This self-referencing ability takes the form of recursion.

The set of strings that can be defined by adding recursion to regular expressions is known as a Context-Free Language.

Context-Free Languages are generated by Context-Free Grammars.
We’ve seen a little bit of context-free grammars, but let’s flesh out the details. Context-free grammars are composed of rules known as *productions*. Each production has left-hand side symbols known as *non-terminals*, or *variables*. On the right-hand side, a production may contain *terminals* (tokens) or other non-terminals. One of the non-terminals is named the *start symbol*.

```
expr  \rightarrow id | number | - expr | ( expr ) | expr op expr
op    \rightarrow + | - | * | /
```

This notation is known as Backus-Naur Form.
DERIVATIONS

So, how do we use the context-free grammar to generate syntactically valid strings of terminals (or tokens)?

1. Begin with the start symbol.
2. Choose a production with the start symbol on the left side.
3. Replace the start symbol with the right side of the chosen production.
4. Choose a non-terminal A in the resulting string.
5. Replace A with the right side of a production whose left side is A.
6. Repeat 4 and 5 until no non-terminals remain.
Let's do a practice derivation with our grammar. We'll derive the string “(base1 + base2) * height/2”. The start symbol is expr.

Each string of symbols in the steps of the derivation is called a sentential form. The final sentential form is known as the yield.

expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /

expr → expr op expr
expr → expr op expr op expr
expr → expr op expr op number
expr → expr op expr / number
expr → expr op id / number
expr → expr * id / number
expr → ( expr ) * id / number
expr → ( expr op expr ) * id / number
expr → ( expr op id ) * id / number
expr → ( expr + id ) * id / number
expr → ( id + id ) * id / number
To save a little bit of room, we can write:

\[
\text{expr} \rightarrow * (\text{id} + \text{id}) * \text{id} / \text{number}
\]

“derives after zero or more replacements”

Note that in this derivation, we replaced the right-hand side consistently, leading to a right-most derivation. There are alternative derivation methods.
PARSE TREES FROM DERIVATIONS

expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /

expr ➔ expr op expr
  ➔ expr op expr op expr
  ➔ expr op expr op number
  ➔ expr op expr / number
  ➔ expr op id / number
  ➔ expr * id / number
  ➔ ( expr ) * id / number
  ➔ ( expr op expr ) * id / number
  ➔ ( expr op id ) * id / number
  ➔ ( expr + id ) * id / number
  ➔ ( id + id ) * id / number
PARSE TREES FROM DERIVATIONS

expr \rightarrow id \mid number \mid - expr \mid ( expr ) \mid expr \ op \ expr
op \rightarrow + \mid - \mid * \mid /

expr \rightarrow expr \ op \ expr
\rightarrow expr \ op \ expr \ op \ expr
\rightarrow expr \ op \ expr \ op \ number
\rightarrow expr \ op \ expr / number
\rightarrow expr \ op \ id / number
\rightarrow expr * id / number
\rightarrow ( expr ) * id / number
\rightarrow ( expr op expr ) * id / number
\rightarrow ( expr op id ) * id / number
\rightarrow ( expr + id ) * id / number
\rightarrow ( id + id ) * id / number
PARSE TREES FROM DERIVATIONS

\[ expr \rightarrow id \mid number \mid - expr \mid ( expr ) \mid expr \ op \ expr \]

\[ op \rightarrow + \mid - \mid * \mid / \]
**PARSE TREES FROM DERIVATIONS**

\[
\begin{align*}
\text{expr} & \rightarrow \text{id} \mid \text{number} \mid \text{- expr} \mid ( \text{expr} ) \mid \text{expr op expr} \\
\text{op} & \rightarrow + \mid - \mid * \mid / 
\end{align*}
\]
PARSE TREES FROM DERIVATIONS

\[
\begin{align*}
\text{expr} & \rightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid (\text{expr}) \mid \text{expr} \text{ op} \text{ expr} \\
\text{op} & \rightarrow + \mid - \mid * \mid /
\end{align*}
\]

expr → expr op expr
→ expr op expr op expr
→ expr op expr op number
→ expr op expr / number
→ expr op id / number
→ expr * id / number
→ ( expr ) * id / number
→ ( expr op expr ) * id / number
→ ( expr op id ) * id / number
→ ( expr + id ) * id / number
→ ( id + id ) * id / number
PARSE TREES FROM DERIVATIONS

expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /

expr → expr op expr
     → expr op expr op expr
     → expr op expr op number
     → expr op expr / number
     → expr op id / number
     → expr * id / number
     → ( expr ) * id / number
     → ( expr op expr ) * id / number
     → ( expr op id ) * id / number
     → ( expr + id ) * id / number
     → ( id + id ) * id / number

expr
     op
      *
     expr
      op
      expr
     id
      /
      num
PARSE TREES FROM DERIVATIONS

```
expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
```

```
expr  →  expr op expr
      →  expr op expr op expr
      →  expr op expr op number
      →  expr op expr / number
      →  expr op id / number
      →  expr * id / number
      →  ( expr ) * id / number
      →  ( expr op expr ) * id / number
      →  ( expr op id ) * id / number
      →  ( expr + id ) * id / number
      →  ( id + id ) * id / number
```
### Parse Trees from Derivations

#### Grammar Rules

```latex
expr \rightarrow id \mid number \mid - \ expr \mid ( \ expr ) \mid expr \ op \ expr
op \rightarrow + \mid - \mid * \mid /
```

#### Parse Tree

```
expr
  op
    expr
      op
        expr
          op
            expr
              op
                expr
                  op
                    expr
                      op
                        expr
                          op
                            expr
                              op
                                expr
                                    op
                                      expr
                                        op
                                          expr
                                            op
                                              expr
                                                op
                                                  expr
                                                    op
                                                      expr
                                                        op
                                                          expr
                                                           op
```

#### Parse Derivations

- `expr \rightarrow expr \ op \ expr`
- `expr \ op \ expr \ op \ expr`
- `expr \ op \ expr \ op \ number`
- `expr \ op \ expr \ / \ number`
- `expr \ op \ id \ / \ number`
- `expr \ * \ id \ / \ number`
- `( expr ) \ * \ id \ / \ number`
- `( expr \ op \ expr ) \ * \ id \ / \ number`
- `( expr \ op \ id ) \ * \ id \ / \ number`
- `( expr + \ id ) \ * \ id \ / \ number`
- `( id + \ id ) \ * \ id \ / \ number`

---

**Note:** The parse tree and derivations demonstrate the process of constructing a parse tree from a derivation of an expression according to the given grammar rules. Each rule is applied to generate a tree structure that represents the expression.
PARSE TREES FROM DERIVATIONS

\[ expr \rightarrow id \mid number \mid - \ expr \mid ( \ expr ) \mid expr \ op \ expr \]
\[ op \rightarrow + \mid - \mid \ast \mid / \]
**PARSE TREES FROM DERIVATIONS**

\[
\text{expr} \rightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid (\text{expr}) \mid \text{expr op expr}
\]

\[
\text{op} \rightarrow + \mid - \mid * \mid /
\]

- **Example Derivations:**
  - `expr` → `expr op expr`
  - `expr op expr` → `expr op expr op expr`
  - `expr op expr` → `expr op expr op number`
  - `expr op expr` → `expr op expr / number`
  - `expr op expr` → `expr op id / number`
  - `expr op expr` → `expr * id / number`
  - `(expr) * id / number`
  - `(expr op expr) * id / number`
  - `(expr op id) * id / number`
  - `(expr op id) * id / number`
  - `(id + id) * id / number`

**Parse Tree Diagram:**

```
  expr
     /    \
  op    expr
    / \
  *   expr
  /   / \
id  /   expr
    /   / \
   +   op   expr
      / \
     id num
```
PARSE TREES FROM DERIVATIONS

expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
Consider the following: “length * width * height”

From our grammar, we can generate two equally acceptable parse trees.
Consider the following: “length * width * height”

From our grammar, we can generate two equally acceptable parse trees.

Grammars that allow more than one parse tree for the same string are said to be ambiguous. Parsers must, in practice, generate special rules for disambiguation.
Context-free grammars can be structured such that derivations are more efficient for the compiler.

Take the example of arithmetic expressions. In most languages, multiplication and division take precedence over addition and subtraction. Also, associativity tells us that operators group left to right.

We could allow ambiguous derivations and let the compiler sort out the precedence later or we could just build it into the structure of the parse tree.
AMBIGUOUS DERIVATIONS

Previously, we had:

```
expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
```

Building in associativity and operator precedence:

```
expr → term | expr add_op term
term → factor | term mult_op factor
factor → id | number | - factor | ( expr )
add_op → + | -
mult_op → * | /
```
Previously, we had:

```
expr → id | number | - expr | ( expr ) | expr op expr
op → + | - | * | /
```

Building in associativity and operator precedence:

```
expr → term | expr add_op term
term → factor | term mult_op factor
factor → id | number | - factor | ( expr )
add_op → + | -
mult_op → * | /
```

Example: $3 + 4 \times 5$

```
expr
  +
  term
  factor
  *
  number
```

```
expr
  term
  mult_op
  factor
  number
```
Scanning
Finite Automata: NFAs and DFAs
Implementing a Scanner