Several interesting and intuitive languages are not regular

```
{ (^i )^i | i >= 0 }
```

This is fairly representative of several programming constructs, e.g., nested arithmetic constructs, nested scopes, nested if-then-else, etc. Programming languages usually have a natural recursive structure (often this is the structure of human thinking)

Regular expressions cannot represent this language. Intuitively, this language requires maintaining an unbounded amount of memory (specified by *i*), while FAs only have a bounded amount of memory (the number of states)

Parser

Input: A sequence of tokens as output from the lexer
Output: parse tree of the program

Example input: `if x == y then z = 1; else z = 2;`

Example output level 1: `if cond then statement else statement`

Example output level 2: `if-then-else statement`

Example output level 3: `statement`

The parse tree may be implicit, may not be output explicitly by the compiler.

For parsing, we need:

1. A language for describing valid strings of tokens
2. A method for distinguishing valid from invalid strings of tokens

Recursive structure of programming languages are a good fit for Context-Free grammars

```
EXPR =   if EXPR then EXPR else EXPR;
       | while EXPR ( EXPR )
```

A CFG consists of:

- A set of terminals T
- A set of non-terminals N
- A start symbol S \in N
- A set of productions:

  ```
  X -> Y1Y2..Yn
  ```

  - X (on the left-hand side) must be a non-terminal. That's what an LHS means.
  - Yi could be a terminal, a non-terminal, or the special symbol \epsilon

Strings of balanced parantheses:

```
S -> (S)
S -> \epsilon
```

Here,

```
N = {S}
T = {'(', ')'}
```

Productions can be read as replacement rules: the RHS can replace the LHS

1. Begin with a string with only the non-terminal S
2. Replace any non-terminal X in the string by the right-hand side of some production X -> Y1..Yn
3. Repeat (2) until there are no non-terminals

In other words, at any step of this *derivation* we perform the following replacement:

```
X1X2X3..Xi X Xi+1..Xn --> X1X2..Xi Y1Y2..Yk Xi+1..Xn
```

A derivation is of the form:

```
S --> ... -> \alpha0 --> ... --> \alphaN
N >= 0
```

Let G be a CFG with start symbol S. L(G) of G is:

```
{a1...an | ai \in T and S ---*--> a1...an}
```

Terminals are called so because there are no rules for replacing them

For a parser, the terminals are the tokens

A fragment of C (this notation is a more concise form of writing the productions, and is used in actual tools)

```
selection_statement:   IF '(' expression ')' statement ELSE statement
                     | IF '(' expression ')' statement
                     | SWITCH '(' expression ')' statement;

iteration_statement:   WHILE '(' expression ')' statement
                     | DO statement WHILE '(' expression ')' ';'
```

```
                    | FOR '(' expression_statement expression_statement ')' statement
                    | FOR '(' expression_statement expression_statement expression ')' statement
                    | FOR '(' declaration expression_statement ')' statement
                    | FOR '(' declaration expression_statement expression ')' statement;


expression_statement:   ';'
                     | expression ';';

statement:   labeled_statement
           | compound_statement
           | expression_statement
           | selection_statement
           | iteration_statement
           | jump_statement;
```

Some elements of this language?

Another example: arithmetic expressions

```
E --> E + E
E --> E * E
E --> ( E )
E --> Id
```

The idea of a CFG is a big step.

- Membership in a language is a yes/no answer. Also need a parse-tree
- Must handle errors gracefully. Give feedback to the programmer
- Need an implementation of the CFGs (e.g., bison generates an implementation given a grammar)

Form of the grammar is important

- Many grammars generate the same language
- Parsing tools are sensitive to the grammar. Only some grammars are parsable by some tools. Will see examples later

A derivation can be drawn as a tree

- Start symbol is tree's root
- For a production X --> Y1..Yn, add children Y1..Yn to node X

Example: `id * id + id`. Generate the derivation tree for the grammar of arithmetic expressions. This derivation tree is called the parse tree of this string. While building the tree, also write the derivation rules that have been used at each step to the left.

A parse tree has

- terminals at the leaves
- non-terminals at the interior nodes

An in-order traversal of the parse tree yields the original input

Multiple parse trees are possible for the same input (even for this example)

The parse tree shows the association of the operations, the original input does not.

The example we discussed is a *leftmost-derivation*: at each step, replace the left-most non-terminal. There is an equivalent notion of a *rightmost-derivation*.

A derivation defines a parse tree. But one parse tree can have multiple derivations. The leftmost and rightmost derivations are important from the point-of-view of parser implementation (as we will see later).

Note that the leftmost and rightmost derivations have the same parse tree. This is just about the order in which the branches of the parse tree were added.

Ambiguity in CFGs

Example: id*id + id

This string has two parse-trees in the grammar of arithmetic operations: (id * id) + id AND id * (id + id)

A grammar is *ambiguous* if it has more than one parse tree for some string.

- Equivalently, there is more than one right-most or left-most derivation for some string

Ambiguity may leave the meaning of the program ill-defined.

Several ways to handle ambiguity: One method is to rewrite the grammar unambiguously (by "stratifying" it)

```
E --> E' + E | E'
E' --> id * E' | id | (E) * E' | (E)
```

Show the unique left-most derivation for `id*id + id` using this new grammar.
Enforces precedence of * over +. E can only generate sum expressions over E'.

Another example:

```
E --> if E then E | if E then E else E | ...
```

String : if E1 then if E2 then E3 else E4
Ambiguity: which "if" does the "else" match with?
Typical rule: "else" matches the closest unmatched "if" (i.e., which does not already have an associated "else")

```
E -->   MIF   /* all if are matched with else */
         | UIF   /* some if is unmatched */

MIF -->   if E then MIF else MIF
           | ...

UIF -->   if E then MIF else UIF  /* the then expression is MIF, because if it were UIF then this else should have matched it! */
           | ...
```

Show the two parse trees, and show which one is rejected by this new grammar.

Impossible to convert automatically an ambiguous grammar to an unambiguous one

Used with care, ambiguity can sometimes simplify the grammar

- Sometimes allows more natural definitions
- We need disambiguation mechanisms

Most languages and parsing tools

- Use the more natural (ambiguous) grammar
- Along with disambiguating declarations

Example: E -> E + E | E * E | Int
Precedence: %left + (means that the "+" operator is left associative)
Precedence: %left * (means that the "*" operator is left associative)
The order of this specification says which one has higher precedence in case of ambiguity
Show how these precedence rules eliminate one of the parse trees

The parser will use these declarations to decide which move to take in case of ambiguity.

Error handling

Purpose of a compiler is:

- Detect non-valid programs
- Translate the valid ones

Many kinds of possible errors (e.g., in C):

- Lexing errors. e.g., abc$abc, detected by lexer
- Syntax errors. e.g., abc * % 123, detected by parser
- Semantic errors. e.g., int x; y = x(3), detected by type checker
- Correctness. e.g., quicksort. detected by tester/user. out of scope for a compiler

Error handlers should

- Report errors accurately and clearly
- Recover from an error quickly
- Not slow down compilation of valid code

Error handling modes

- Panic
- Error production
- Automatic local and global correction. Popular area of research at some point, but not mainstream.

Panic mode is the simplest

- When an error is encountered, discard tokens until a clear role is found
- Continue from there

Looking for "synchronizing tokens", typically the statement or expression terminators (e.g., ";" in C)

Example: (1 + + 2) + 3 The parser is going to get stuck at the second +. In panic mode recovery, skip ahead to next integer and then continue. (In this example, it would restart at 2, and treat it as 1 + 2, and then it would parse fine).

Bison: use the special terminator `error` to describe how much input to skip (error productions)

```
E --> int | E + E | (E) | error int | (error)
```

The fourth production says that if you see an error while trying to parse E, then the `error` symbol will match all the input until an int.

Similarly, the fourth production says that if you encounter an error while you are inside a paranthesized expression, you can discard everything until the closing bracket.

Error productions

- Specify known common mistakes in the grammar
- Example:
  - Write 5x instead of 5*x
  - Mathematicians complained that they were used to writing 5x but compiler gives parse error
  - Response: add a production E -> EE
- Disadvantage:
  - Complicates the grammar
  - Promotes mistakes (by making the syntax less restrictive)

- Common response by production compilers: warn about such usage but accept it anyway. *Warnings*

Error correction: find a "nearby" program. Notion of "edit distance"

- Try token insertions or deletions upto a certain edit distance exhaustively.
- Cons: hard to implement, significant slowdown in compilation, "nearby" is not necessarily the intended program
- Common response by production compilers today: suggestions to the programmer, e.g., OCaml
- PL/C is an example of a past error-correcting compiler
  - Motivation: compilation was quite slow. could take a day to recompile
  - Hence wanted to find as many errors as possible in one compilation cycle

Abstract syntax trees

Review

- A parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees (AST)
  - Like parse trees but ignore some details

Consider the grammar

```
E --> int | (E) | E + E
```

And the string

```
5 + (2 + 3)
```

After lexical analysis, a list of tokens

```
int5 '+' '(' 'int2' '+' 'int3' ')'
```

During parsing we build a parse tree *Draw a parse tree*root = E, second level expands E + E, third level expands the first E into int5 and the second E into (E), and so on... The parse tree is sufficient for compilation (it captures the nesting structure) but it has too much information, such as parantheses and single-successor nodes, etc. AST is a cleaned-up and more concise version of the parse tree. Parentheses are redundant because the tree structure tells us the grouping already

AST also captures the nesting structure, but it *abstracts* from the concrete syntax. It is more compact and easier to use. An important data structure in a compiler

```
(A): PLUS, (B), (C)
(B): 5
(C): PLUS (D), (E)
(D): 2
(E): 3
```

AST is an important data structure for the rest of the discussion