Backus-Naur Form
Announcements
Describing Code
Languages with Recursive Structure

Programming languages often have recursive **structure** (even if they do not support recursion). E.g., the calculator language was a tiny subset of Scheme that had only built-in procedures.

- Expressions are either numbers or call expressions.
- A call expression is +, -, *, or / followed by zero or more expressions.

\[
(+ (* 3 (+ (* 2 4) (+ 9 3))) (+ (* 0 2) 1))
\]

All calculator programs are sequence of these characters: ( ) + − * / . 0 1 2 3 4 5 6 7 8 9
But a valid calculator program must also have a tree structure and balanced parentheses.
Limitations of Regular Expressions

The parentheses language: an expression is zero or more expressions surrounded by <>

E.g., <<<><><><><><><><>><><>

The regular expression \[<>\]+ is too expressive; it matches => and <=.

(Demo)

\(<\(\)\)*\> matches <>, <<>>, and <<<>>, but not <<>>> or <<<<<>>

\(<\(<\)\)*\)> matches <>, <<>>, <<<>>, and <<<<<>>, but not <<<<<>>> or <<<<<<>>>>

Regular expressions cannot describe recursive structures of arbitrary depth.

(Therefore, a regular expression cannot describe the set of valid regular expressions!)
Context-Free Grammars
Grammars

A language has:

- **Syntax**: the set of allowed expressions in the language
- **Semantics**: the meaning of an expression

A grammar is a compact description of the syntax of a language.

A *regular language* is a language whose syntax can be described by a regular expression.

A *context-free language* has syntax that can be described by a *context-free grammar*.

- All of the features of a regular expression
- Can ensure that parentheses are balanced and properly nested
Backus-Naur Form

Backus-Naur form is a particular syntax for describing context-free grammars.

• Something like it was invented by John Backus to describe the syntax of ALGOL.

• Describing languages via context-free grammars is an older idea, formalized by Chomsky.

?start: expr
expr: OPEN CLOSE | OPEN exprs CLOSE
exprs: expr | expr exprs
OPEN: "<"
CLOSE: ">"

The Lark Python module is available on code.cs61a.org and has its own flavor of BNF.

Create a file on code.cs61a.org that starts with ?start:, and it will be processed by Lark.

(Demo)
Details of Backus-Naur Form in Lark

A special symbol \texttt{?start} corresponds to a complete expression.

Symbols in all caps are called terminals:

- Can only contain /regular expressions/, "text", and other TERMINALS
- No recursion is allowed within terminals

Unnamed literals within non-terminals do not show up in the parse tree.

\texttt{?start: numbers}

\texttt{numbers: INTEGER | numbers ," INTEGER}

\texttt{INTEGER: "0" | /-?[1-9]\d*/}

The \texttt{%ignore} directive omits those terminals in the final parse. E.g., \texttt{%ignore /\s+/}

(Demo)
Extended BNF
Extended BNF Operators

Extended BNF is not more expressive than BNF, but the grammar descriptions are shorter.

From the docs (lark-parser.readthedocs.io/en/latest/grammar.html#rules):

- (item item ..) - Group items
- [item item ..] - Maybe. Same as (item item ..)?
- item? - Zero or one instances of item ("maybe")
- item* - Zero or more instances of item
- item+ - One or more instances of item
- item ~ n - Exactly n instances of item
- item ~ n..m - Between n to m instances of item

EBNF notation appears in Python docs (docs.python.org/3/reference/expressions.html):

```
dict_display ::= "{" [key_datum_list | dict_comprehension] "}"  
dict_comprehension ::= expression ":" expression comp_for
```
Example: Calculator Language

A few more Lark specifics:

• Lark supports some common terminal types, such as numbers, via the `%import` directive.
• Symbol starting with `?` do not show up in the parse tree if they have exactly one child.

A grammar for Calculator:

```lark
%start: expr
%expr: NUMBER | call
call: "(" OPERATOR expr* ")"
OPERATOR: "+" | "-" | "*" | "/"

%ignore /\s+/
%import common.NUMBER
```

(Demo)
Ambiguity
Two Parses for the Same String

This grammar is ambiguous for 1+2*3:

```plaintext
?start: expr
?expr: NUMBER | expr OPERATOR expr
OPERATOR: "+" | "*"
%import common.NUMBER
```

Introducing symbols can eliminate ambiguity:

```plaintext
?start: expr
?expr: mul_expr | expr PLUS mul_expr
?mul_expr: NUMBER | mul_expr TIMES NUMBER
PLUS: "+"
TIMES: "*"
%import common.NUMBER
```