Lecture 32: BNF

Describing Language Syntax: BNF

- Among the many innovations introduced by the Algol 60 programming language, it introduced what is now the standard way to describe the syntax (or grammar) of a programming language.
- As a result, nearly all such descriptions use some form of *Backus-Naur Form (BNF)*.
- Example from the Python 3 documentation (slightly modified):

```
dict_display       ::=  "{" [key_list | dict_comprehension] "}"
key_list           ::=  key_datum ("," key_datum)* [","]
key_datum          ::=  expression ":" expression
                   |    "**" or_expr
dict_comprehension ::=  expression ":" expression comp_for
```

- This particular version of BNF resembles the one we'll use today: that used by the Python package *Lark*.
- (In fact the only difference is that while the Python docs use the traditional symbol '::=', Lark uses plain ':'.)

BNF vs. Regular Expressions

- BNF is a more powerful pattern language than the regular expressions that we saw in the last lecture.
- For example, regular expressions cannot accurately match a language (like Python) in which parentheses balance and can be arbitrarily nested.
- Consequently, the algorithms used to match BNF are more complex than those for regular expressions. (Take CS164 if you'd like to learn something about them).
- In formal terminology, we say that regular expressions describe sets of strings called *regular* or *type 3* languages, while BNF describes *context-free* or *type 2* languages.
- (Type 0 languages are all those that can be defined by some algorithm. Type 1 languages are somewhere in between types 0 and 2 in power.)

Basic BNF

- A BNF grammar consists of a set of grammar rules. Here, we'll use the syntax of Lark to write them (actually, I should say *metasyntax* of Lark, since it is the syntax *of a* syntax.)
- The basic (classical) form of a grammar rule is simply

  $\textit{symbol}_0$`:` $\textit{symbol}_1$ $\textit{symbol}_2$ $\cdots$ $\textit{symbol}_n$ `for` $n \geq 0$.
- Symbols represent sets of strings. They come in two flavors:
  - *Nonterminal symbols* are written as lower-case identifiers. $\textit{symbol}_0$ in a rule is always a nonterminal symbol.
  - *Terminal symbols* are quoted strings, regular expressions, or defined as upper-case names.
- The rule above means "a $\textit{symbol}_0$ *may be formed by* concatenating a $\textit{symbol}_1$, a $\textit{symbol}_2$,. . . , and a $\textit{symbol}_n$."
- To give multiple alternative rules for forming a given nonterminal, we can use '|', as for regular expressions:

```
number: octal_number | decimal_number | hexadecimal_number
```

Defining Terminals

- Terminal symbols are the base cases of the grammar. In the Scheme project, we called them *tokens*.
- In Lark grammars, they can be written as
  - Quoted strings (e.g., `"*"` or `"define"`), which simply match themselves.
  - Regular expressions surrounded by `"/"` on both sides (if you've used Perl, you've seen this notation). E.g., `/\d+/`.
  - Symbols written in upper case (e.g., `NUMBER`), which are defined by lexical rules, such as

    ```
    NUMBER: /\d+(\.\d+)/
    FRACTION: NUMBER "/" NUMBER
    ```

- Often, you'll want to define terminal symbols that should always be thrown away before doing any matching. Some very common ones are whitespace and comments.
- One does so by means of the `%ignore` directive:

  ```
  %ignore /\s+/          // Ignores all whitespace
  ```

Example I

- Here's a grammar for some very simple sentences:

```
// By default 'start' defines all the matched strings.
start: sentence
sentence: noun_phrase verb
noun: NOUN
noun_phrase: article noun
article : | ARTICLE     // The first option matches ""
verb: VERB

NOUN: "horse" | "dog" | "hamster"
ARTICLE: "a" | "the"
VERB: "stands" | "walks" | "jumps"
%ignore /\s+/
```

- For example, this grammar will match (or *accept*)

```
the horse jumps
a dog walks
hamster stands
```

Repeated Patterns

- In this purist form of BNF, one can *get* repetition by recursion, just as in Scheme.
- For example, to describe a list of one or more signed integers, one could write

```
numbers: numbers "," INTEGER | INTEGER
INTEGER: /-?\d+/
```

- *As* you can see, Lark is somewhat more clever about recursion than Python: you don't get an infinte loop even though we write the recursive case before the base case.

Nested Recursions

- How would you match any of the strings `()`, `(())`, `((()))`, etc.?
- This is an interesting set of strings because no regular expression can describe it.

```
???
```

Nested Recursions

- How would you match any of the strings `()`, `(())`, `((()))`, etc.?
- This is an interesting set of strings because no regular expression can describe it.

```
nest: "(" ")" | "(" nest ")"
```

- In other words, a nest is either '()' or the result of taking another string matched by nest and slapping parentheses around it.

Extended BNF (EBNF)

- Being a purist can be tedious. The repetition idiom illustrated by the numbers example would be clearer if we could directly say that a certain pattern is supposed to be repeated.
- So, it is common to use several of the notations that we used for regular expressions. The system will translate each of these into pure BNF behind our backs.

Notation	Meaning	Pure BNF Equivalent
`item*`	Zero or more items	`items:   | items item`
`item+`	One or more items	`items:  item | items item`
`[ item ]`	Optional item	`opt_item:   | item`
`item?`	Same as [item]	

Extended BNF (II)

- Also as in regular expressions, parentheses and square brackets group, so that you can write

  ```
  name : /\w+/
  number: /\d+/
  list: ( name | number )+
  ```

  to describe a list of one or more names and numbers.

- And you can write

  ```
  numbered_list : ( name [ ":" number ] )+
  ```

  to describe a sequence of one or more names, each optionally followed by a colon and number.

- How would you describe a list of zero or more names separated by commas (no comma at the end)?

Extended BNF (II)

- Also as in regular expressions, parentheses and square brackets group, so that you can write

  ```
  name : /\w+/
  number: /\d+/
  list: ( name | number )+
  ```

  to describe a list of one or more names and numbers.

- And you can write

  ```
  numbered_list : ( name [ ":" number ] )+
  ```

  to describe a sequence of one or more names, each optionally followed by a colon and number.

- How would you describe a list of zero or more names separated by commas (no comma at the end)?

  ```
  comma_separated_list : [ name ("," name)* ]
  ```

Larger Example: Calculator

- We showed you a calculator using Lisp notation. It's easy to describe with EBNF:

```
start: calc_expr

?calc_expr: NUMBER | calc_op   // We'll get to the "?" later

calc_op: "(" OPERATOR calc_expr* ")"

OPERATOR: "+" | "-" | "*" | "/"

%ignore /\s+/
// NUMBER is a terminal symbol defined in the Lark library.
// It described a decimal numeric literal (either integer
// or floating point).
%import common.NUMBER
```

Syntax Trees

- In most applications of BNF, we are interested not just in whether some text is grammatical, but also what its grammatical structure is.
- One standard representation of this structure is a kind of tree known, therefore, as a *syntax tree*.
- Lark produces these automatically. A Lark syntax tree is either
  - A Token (basically a kind of string) containing the text matching a terminal symbol, or
  - A Tree node whose label (called `.data`) is the name of a nonterminal, with zero or more children (`.children`, a list) that are Trees or Tokens.

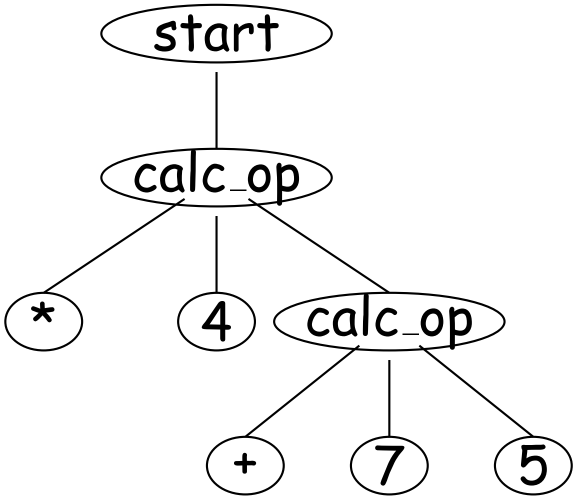
A Calculator Tree

```
start: calc_expr
?calc_expr: NUMBER | calc_op
calc_op: "(" OPERATOR calc_expr* ")"
OPERATOR: "+" | "-" | "*" | "/"
```

- With this grammar, parsing the text

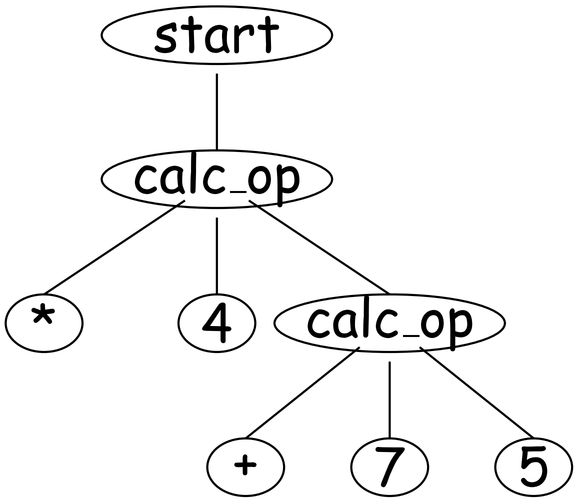
  ```
  (* 4 (+ 7 5))
  ```

  yields a Tree that looks like this:



A Calculator Tree (II)

```
start: calc_expr
?calc_expr: NUMBER | calc_op
calc_op: "(" OPERATOR calc_expr* ")"
OPERATOR: "+" | "-" | "*" | "/"
```



- In this tree, the parts that result from matching the rules for `start` and `calc_op` turn into tree nodes whose children are the Trees and Tokens that come from the right-hand sides of the rules.
- Leaves are nodes with no children and terminals (type Token).
- Lark by default removes parts of the rules that are quoted-string terminals (like `"("`), leaving named tokens (like NUMBER) or Tokens defined by regular expressions.
- It also removes any nodes whose rules start with '?' (like `calc_expr`) and have only one child, replacing them with that child.
- Because the tree is simplified, we call it an *abstract syntax tree.*

Evaluation

- Now that we have trees, we can do the same sorts of things we did in the scheme evaluator and calculator.
- For this purpose, Lark provides Transformers, which will convert the nodes of a tree in bottom-up fashion.
- Excerpt:

```
from Lark import Transformer
class Eval(Transformer):
    def start(self, args):
        return args[0]
    def calc_op(self, args):
        op = args[0]
        if op == ’+’:
            return sum(args[1:])
        elif op == ’-’:
            ...
    def NUMBER(self, num):
        return float(num)

evaluator = Eval()
print(evaluator.transform(tree))
```

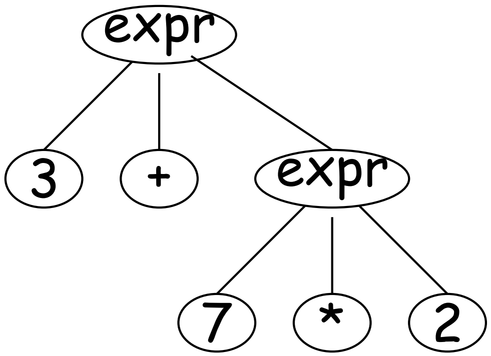
Ambiguity

- In the regular expressions `r"(ab|a)(ba|b)"`, it is ambiguous which group matches what on inputs like `"ab"`.
- There is a similar ambiguity in the case of BNF. A common example is the syntax of infix expressions (such as Python's):

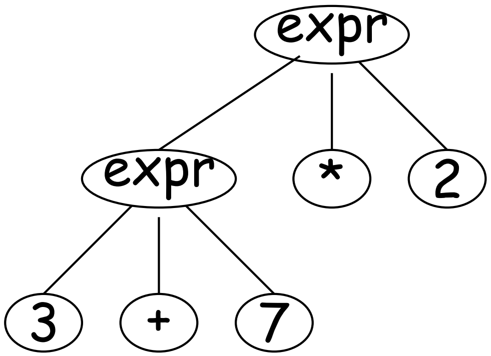
```
?start: expr
?expr: NUMBER | expr OPERATOR expr
OPERATOR: "+" | "-" | "*" | "/"
```

- What tree should I get for `3+7*2`? It obviously makes a difference when evaluating the expression.

Option 1



Option 2



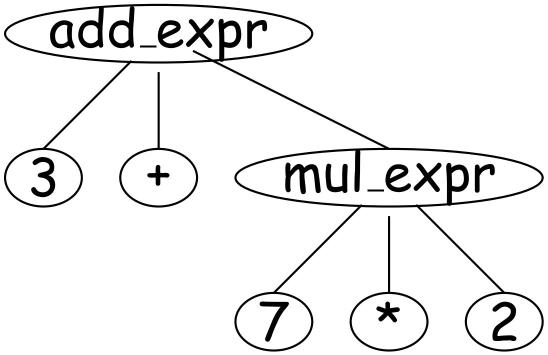
Ambiguity Resolution

- Some parser generators provide a way of specifying operator precedence.
- However, there is a traditional way to resolve things directly in BNF:

```
?start: expr
?expr: ?add_expr
?add_expr: mul_expr | add_expr ADDOP mul_expr
?mul_expr: NUMBER | mul_expr MULOP NUMBER
ADDOP: "+" | "-"
MULOP: "*" | "/"
```

- This treats an expression as a list of `mul_expr`s, separated by lower-precedence operators, where each `mul_expr` is a list of `NUMBER`s separated by higher-precedence operators.

- With this grammar, only one tree fits:


add_expr
3
+
mul_expr
7
*
2