Backus-Naur Form

Class outline:

- Backus-Naur Form
- (E)BNF syntax
- Parse tree formation
- Exercises
- Ambiguity

Backus-Naur Form

Describing language syntax

BNF was invented in 1960 to describe the ALGOL language and is now used to describe many programming languages.

An example BNF grammar from the Python docs:

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

A BNF grammar can be used as a form of documentation, or even as a way to automatically create a parser for a language.

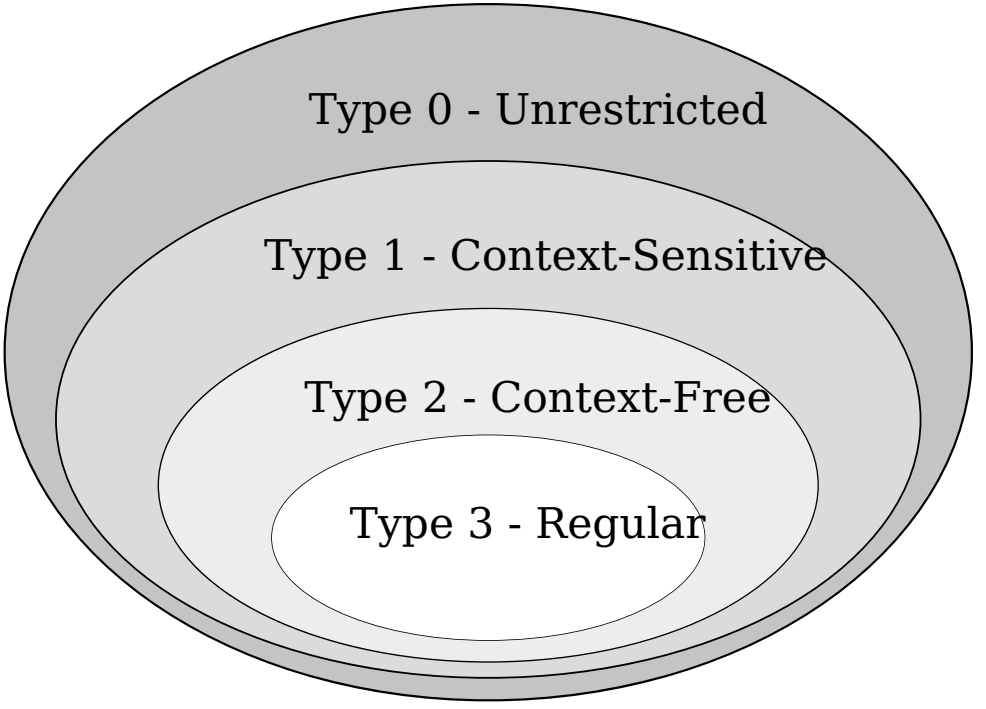
BNF vs. Regular expressions

BNF is more powerful than regular expressions. For example, regular expressions cannot accurately match a language (like Scheme) in which parentheses balance and can be arbitrarily nested.

BNF vs. Regular expressions

BNF is more powerful than regular expressions. For example, regular expressions cannot accurately match a language (like Scheme) in which parentheses balance and can be arbitrarily nested.

In formal language theory, BNF can describe "context-free languages" whereas regular expressions can only describe "regular languages".


Type 0 - Unrestricted
Type 1 - Context-Sensitive
Type 2 - Context-Free
Type 3 - Regular

Basic BNF

A BNF grammar consists of a set of grammar rules. We will specifically use the rule syntax supported by the Lark Python package.

The basic form of a grammar rule:

```
symbol0: symbol1 symbol2 ... symboln
```

Symbols represent sets of strings and come in 2 flavors:

- **Non-terminal symbols**: Can expand into either non-terminal symbols (themselves) or terminals.
- **Terminal symbols**: Strings (inside double quotes) or regular expressions (inside forward slashes).

To give multiple alternative rules for a non-terminal, use `|`:

```
symbol0: symbol1 | symbol2
```

BNF example

A simple grammar with three rules:

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

For the Lark library,

- Grammars need to start with a `start` symbol.
- Non-terminal symbol names are written in lowercase.
- Terminal symbols are written in UPPERCASE.

What strings are described by that grammar?

BNF example

A simple grammar with three rules:

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

For the Lark library,

- Grammars need to start with a `start` symbol.
- Non-terminal symbol names are written in lowercase.
- Terminal symbols are written in UPPERCASE.

What strings are described by that grammar?

```
10
10,-11
10,-11,12
```

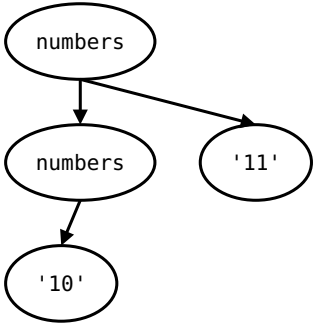
Trying out BNF grammars

You can paste a BNF grammar in code.cs61a.org, and it will be automatically recognized and processed by Lark as long as the first line starts with `?start:`.

If the grammar is parsed successfully, then you can type strings from the language in the prompt.

```
lark> 10,-11
```

If the string can be parsed according to the grammar, a parse tree appears!



Defining terminals

Terminals are the base cases of the grammar (like the tokens from the Scheme project).

In Lark grammars, they can be written as:

- Quoted strings which simply match themselves (e.g. `"*"` or `"define"`)
- Regular expressions surrounded by `/` on both sides (e.g. `/\d+/`)
- Symbols written in uppercase which are defined by lexical rules (e.g. `NUMBER: /\d+(\.\d+)/`

It's common to want to always ignore some terminals before matching. You can do that in Lark by adding an `%ignore` directive at the end of the grammar.

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

Example: Sentences

```
?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+/
```

What strings can this grammar parse?

Example: Sentences

```
?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+/
```

What strings can this grammar parse?

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

Repetition

EBNF is an extension to BNF that supports some shorthand notations for specifying how many of a particular symbol to match.

EBNF	Meaning	BNF equiv
`item*`	Zero or more items	`items: | items item`
`item+`	One or more items	`items: item | items item`
`item?`	Optional item	`optitem: | item`

All of our grammars for Lark can use EBNF shorthands.

Grouping

Parentheses can be used for grouping.

```
?start: list
list: ( NAME | NUM )+
NAME: /[a-zA-Z]+/
NUM: /\d+/

%ignore /\s/
```

Square brackets indicate an optional group.

```
numbered_list: ( NAME [ ":" NUM ] )+
```

Exercise: Describe a comma-separated list of zero or more names (no comma at the end).

Grouping

Parentheses can be used for grouping.

```
?start: list
list: ( NAME | NUM )+
NAME: /[a-zA-Z]+/
NUM: /\d+/

%ignore /\s/
```

Square brackets indicate an optional group.

```
numbered_list: ( NAME [ ":" NUM ] )+
```

Exercise: Describe a comma-separated list of zero or more names (no comma at the end).

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

Importing common terminals

Lark also provides pre-defined terminals for common types of data to match.

```
%import common.NUMBER
%import common.SIGNED_NUMBER
%import common.DIGIT
%import common.HEXDIGIT
```

See all here

Example: Calculator

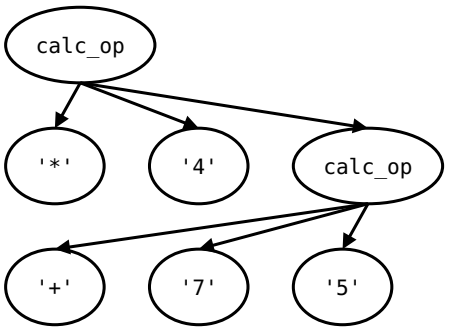
A BNF for the Calculator language:

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

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

Calculator tree breakdown

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



- Terminals are always leaf values, never branches.
- Lark removes unnamed literals entirely (like `"("`) but does show the values of named terminals (like `OPERATOR`) or unnamed regular expressions.
- Lark removes any nodes whose rules start with `?` and have only one child, replacing them with that child (like `calc_expr`).

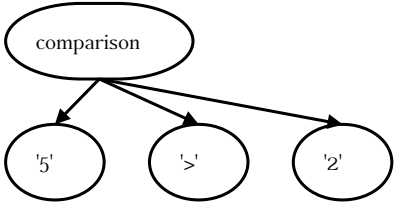
Because the tree is simplified, we call it an **abstract syntax tree**.

Exercises

Exercise: Python comparisons

Write a BNF that can parse simple Python comparisons between numbers: `5 > 2`, `3 < 5`, `32 == 33`, etc.

The comparison `5 > 2` should result in this parse tree:



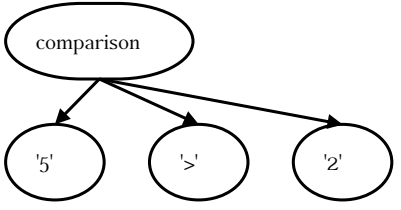

```
?start: comparison
comparison: ____________________________
__________: ____________________________

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

Exercise: Python comparisons (Solution)

Write a BNF that can parse simple Python comparisons between numbers: `5 > 2`, `3 < 5`, `32 == 33`, etc.

The comparison `5 > 2` should result in this parse tree:



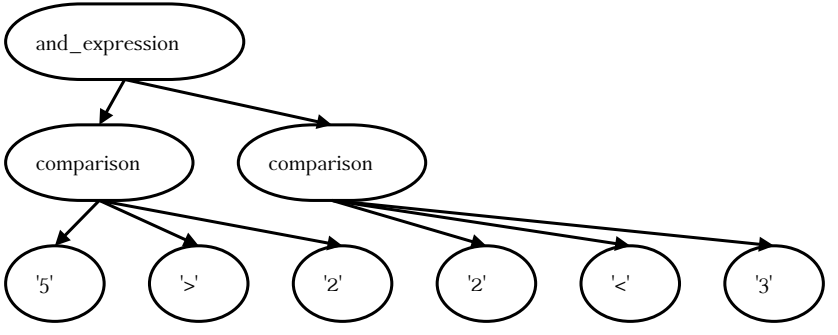

```
?start: comparison
comparison: NUMBER COMPARATOR NUMBER
COMPARATOR: "=="  | ">" | "<"

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

Exercise: Python ands

Write a BNF that can parse simple comparisons or Python `and` expressions with those simple comparisons:
`5 > 2`, `5 > 2 and 3 < 5`, `5 > 2 and 3 < 5 and 2 < 4`.

`5 > 2 and 2 < 3` should result in this parse tree:



Note: An `and` expression may itself contain nested `and`s.

Start from the previous solution.

Exercise: Python ands (Solution)

Write a BNF that can parse simple comparisons or Python `and` expressions with those simple comparisons:
`5 > 2`, `5 > 2 and 3 < 5`, `5 > 2 and 3 < 5 and 2 < 4`.

Note: An `and` expression may itself contain nested `and`s.

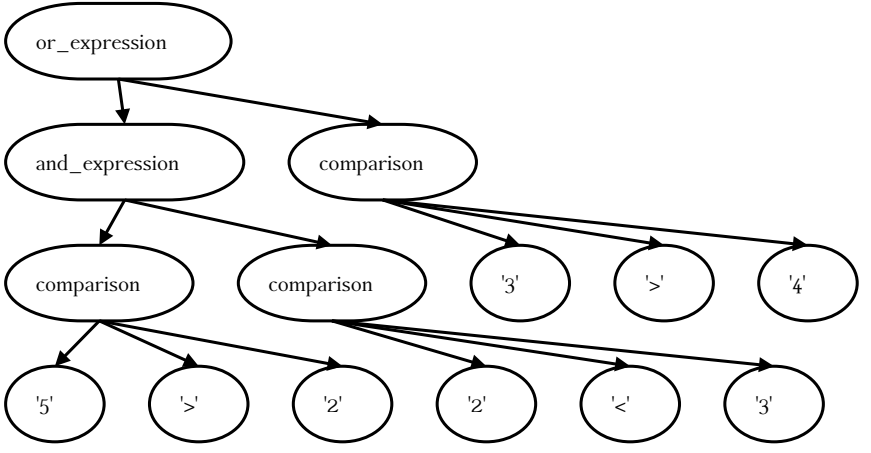
```
?start: expression
?expression: and_expression | comparison
and_expression: expression "and" expression
comparison: NUMBER COMPARATOR NUMBER
COMPARATOR: ">" | "<" | "=="

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

Exercise: Python and/ors

Add support for `or` expressions to previous BNF.
`5 > 2`, `5 > 2 or 3 < 5`, `5 > 2 and 3 < 5 or 2 < 4`.

`5 > 2 and 2 < 3 or 3 > 4` should result in this tree:



Exercise: Python and/ors (Solution)

Add support for `or` expressions to previous BNF.
`5 > 2`, `5 > 2 or 3 < 5`, `5 > 2 and 3 < 5 or 2 < 4`.

`5 > 2 and 2 < 3 or 3 > 4` should result in this tree:

```
?start: expression
?expression: or_expression | and_expression | comparison
or_expression: expression "or" expression
and_expression: expression "and" expression
comparison: NUMBER COMPARATOR NUMBER
COMPARATOR: ">" | "<" | "=="

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

Resolving ambiguity

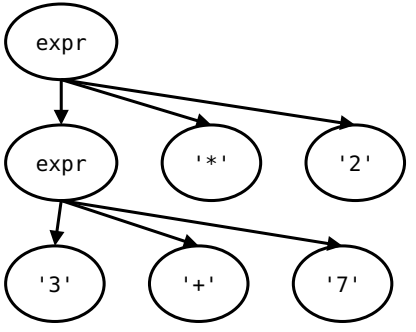
Ambiguity

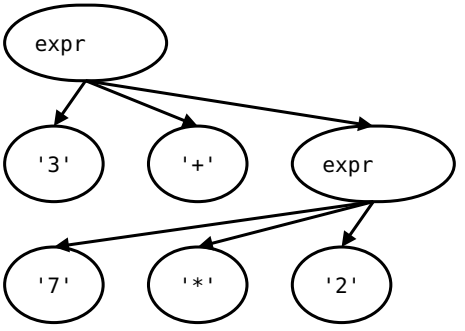
Ambiguity arises when a grammar supports multiple possible parses of the same string.

Python infix expression grammar:

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

What tree should we get for 3+7*2?



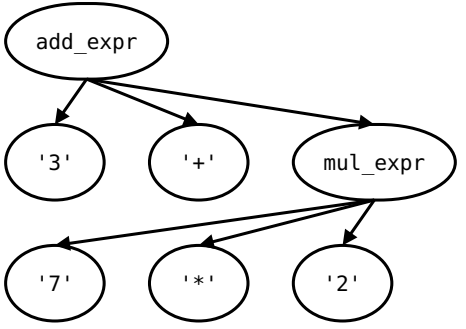

expr
'3'
'+'
expr
'7'
'*'
'2'

Ambiguity resolution

One way to resolve this ambiguity:

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

That grammar can only produce this parse tree:



BNF Tips

- Check your parse trees on code.cs61a.org
- Consider whether a ? should be in front of a non-terminal