Applications of RegEx + BNF

Class outline:

- Applications of RegEx
- Applications of BNF
  - Documentation
  - Parsers

Applications of RegEx

RegEx in Programs

Programs written in a general purpose language (like Python) often use regular expressions for pattern matching.

Example from the CS61A codebase:

```
def format_coursecode(course):
    """Formats a course code is a pretty way, separating the department from
    the course number.
    :param course: the course code, such as "cs61a"
    :return: prettified course code, such as "CS 61A"
    """
    m = re.match(r"([a-z]+)([0-9]+[a-z]?)", course)
    return m and (m.group(1) + " " + m.group(2)).upper()
```

RegEx for searching

Searching in VSCode for all uses of `re` methods:

```
\bre\.
```

Searching with grep for all uses of `re` methods:

```
grep -r --include=\*.py '\bre\.' .
```

RegEx in Spreadsheets

Google Spreadsheets includes functions like `REGEXMATCH` and `REGEXEXTRACT`.

Extracting matching patterns from a cell:

```
=REGEXEXTRACT(A2, ", ([\w\s]*)$")
```

See spreadsheet

RegEx in HTML

The HTML `input` tag is used for single-line form inputs.

```
<label>Username
    <input type="text">
</label>
```

Username

The `input` tag can specify a `pattern` attribute to restrict what input is considered valid.

```
<label>Username
    <input type="text" pattern="[a-zA-Z]+[a-zA-Z\d_]{5,}">
</label>
```

Username

RegEx in SQL

SQL is a way to query/update databases. Many SQL variants have support for searching using regular expressions.

Querying a public database using Google BigQuery:

```
SELECT place_name FROM `bigquery-public-data.geo_us_census_places.places_cal
    WHERE REGEXP_CONTAINS(place_name, r'\sCity$') LIMIT 10;
```

Querying Khan Academy data using BigQuery:

```
SELECT
    readable_id, edit_url, REGEXP_EXTRACT_ALL(perseus_content, r"[^!:]\[[^\]
    FROM
        content_streaming.ArticleRevision_edit_full
    WHERE
        subject_slug="ap-computer-science-principles"
        AND REGEXP_CONTAINS(perseus_content, r"[^!:]\[[^\]]*\]\([^)]*\)")
```

⚠ A word of caution ⚠

Regular expressions can be very useful. However:

- Very long regular expressions can be difficult for other programmers to read and modify.
  See also: Write-only
- Since regular expressions are declarative, it's not always clear how efficiently they'll be processed. Some processing can be so time-consuming, it can take down a server.
- Regular expressions can't parse everything! Don't write an HTML parser with regular expressions.

Applications of BNF

BNF for documentation

Where is BNF used?

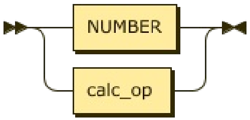
- Language specification: Python, CSS, SaSS, XML
- File formats: Google's robots.txt
- Protocols: Apache Kafka
- Parsers and compilers
- Text generation

You will likely use your BNF reading skills more often than your BNF writing skills.

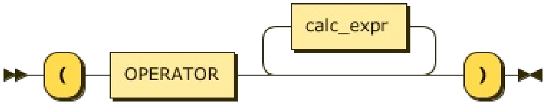
BNF syntax diagrams

A syntax diagram is a common way to represent BNF & other context-free grammars. Also known as railroad diagram.

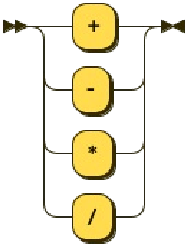
```
calc_expr: NUMBER | calc_op
```




```
calc_op: '(' OPERATOR calc_expr* ')'
```




```
OPERATOR: '+' | '-' | '*' | '/'
```



BNF for Python Integers

Adapted from the Python docs:

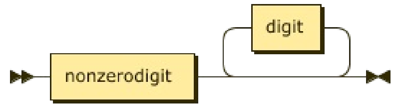
```
?start: integer
integer:  decinteger | bininteger | octinteger | hexinteger
decinteger:  nonzerodigit digit*
bininteger:  "0" ("b" | "B") bindigit+
octinteger:  "0" ("o" | "O") octdigit+
hexinteger:  "0" ("x" | "X") hexdigit+
nonzerodigit:  /[1-9]/
digit:  /[0-9]/
bindigit:  /[01]/
octdigit:  /[0-7]/
hexdigit:  digit | /[a-f]/ | /[A-F]/
```

What number formats can that parse?

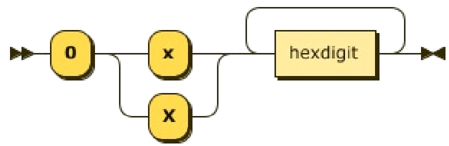
Try in code.cs61a.org!

Syntax diagram for Python integers

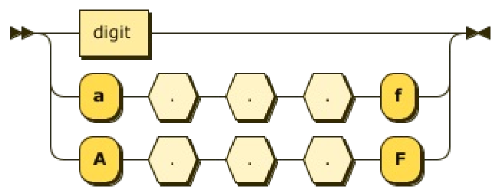
```
decinteger: nonzerodigit digit*
```



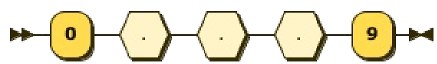

```
hexinteger: "0" ("x" | "X") hexdigit+
```




```
hexdigit: digit | /[a-f]/ | /[A-F]/
```




```
digit: /[0-9]/
```



BNF for Scheme expressions

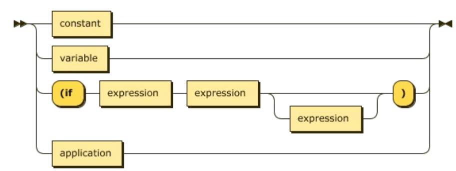
Adapted from the Scheme docs:

```
?start: expression
expression: constant | variable | "(if " expression expression expression? ")" |
constant: BOOLEAN | NUMBER
variable: identifier
application: "(" expression expression* ")"

identifier: initial subsequent* | "+" | "-" | "..."
initial: LETTER | "!" | "$" | "%" | "&" | "*" | "/" | ":" | "<" | "=" | ">" | "?
subsequent: initial | DIGIT | "." | "+" | "-"
LETTER: /[a-zA-z]/
DIGIT: /[0-9]/
BOOLEAN:  "#t" | "#f"

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

*This BNF does not include many of the special forms, for simplicity.

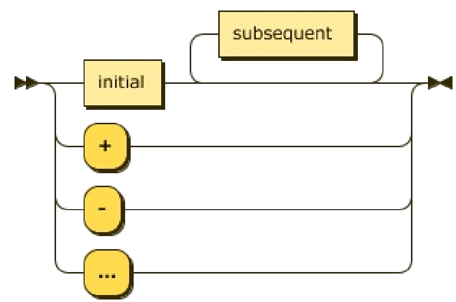
Syntax diagram for Scheme expressions

```
expression: constant | variable | "(if " expression
expression expression? ")" | application
```



```
application: "(" expression expression* ")"
```

```
identifier: initial subsequent* | "+" | "-" | "..."
```

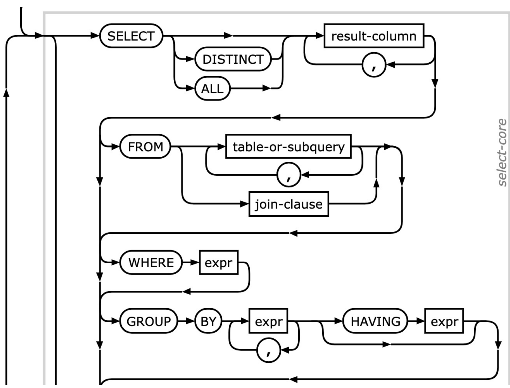


BNF for SQL

Adapted from the SQLite documentation:
(Uses a slightly different BNF syntax)

```
select_stmt ::= ( SELECT [ DISTINCT | ALL ] result_column ( ',' result_column )*
    [ FROM ( table_or_subquery ( ',' table_or_subquery ) * | join_clause ) ]
    [ WHERE expr ] [ GROUP BY expr ( ',' expr ) * [ HAVING expr ] ] |
    VALUES '(' expr ( ',' expr )* ')' ( ',' '(' expr ( ',' expr )* ')' )   )
    [ ORDER BY ordering_term ( ',' ordering_term ) * ] [ LIMIT expr [ ( OFFSET | ',' ) expr ] ]
```

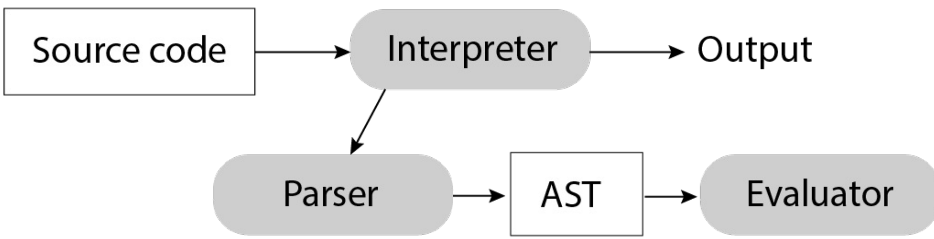
Syntax diagram from the SQLite documentation:



BNF for Parsers

Reminder: Interpreter phases

An interpreter first uses a parser to turn source code into an AST, and then uses an evaluator to turn the AST into an output.



Reminder: Calculator interpreter

The Calculator interpreter is a subset of the Scheme interpreter, using basically the same parser (from scheme_reader.py) but a simpler evaluation process.

From the `read_eval_print_loop` in `calc.py`:

```
expression = scheme_read(src)    # Returns a Pair
result = calc_eval(expression)   # Returns output
```

BNF-based interpreter, Pt 1

Replace `scheme_reader.py` with BNF + BNF engine (Lark)

```
from lark import Lark

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

    %ignore /\s+/
    %import common.NUMBER
"""
parser = Lark(grammar)
line = input("calc> ")
tree = parser.parse(line)
```

```
calc> (+ 1 2)
Tree('start', [Tree('calc_op', [Token('OPERATOR', '+'), Token('NUMBER', '1'), To
```

*The `Tree` class above is part of Lark; it's not the CS61A `Tree` class.

See full parser code.

BNF-based interpreter, Pt 2

Change evaluator to process Lark `Tree`s instead of `Pair`s.

Before:

```
def calc_eval(exp):
    if type(exp) in (int, float):
        return simplify(exp)
    elif isinstance(exp, Pair):
        operator = exp.first
        arguments = exp.second.map(calc_eval)
        return simplify(calc_apply(operator, arguments))
```

After:

```
def calc_eval(exp):
    if isinstance(exp, Token) and exp.type == 'NUMBER':
        return numberify(exp.value)
    elif isinstance(exp, Tree):
        operator = exp.children[0].value
        arguments = [calc_eval(child) for child in exp.children[1:]]
        return calc_apply(operator, arguments)
```

See full interpreter code.

BNF-based interpreter (Variant)

Another option for evaluation is to use the Lark `Transformer` class.

Replace `calc_eval`/`calc_apply` with:

```
class Eval(Transformer):
    def start(self, args):
        return args[0]

    def calc_op(self, args):
        op = args[0]
        operands = args[1:]
        if op == '+':
            return sum(operands)
        elif op == '-':
            if len(operands) == 1:
                return -operands[0]
            else:
                return operands[0] - sum(operands[1:])
        elif op == '*':
            return reduce(mul, operands)
        elif op == '/':
            return reduce(truediv, operands)

    def NUMBER(self, num):
        return numberify(num)
```

See full interpreter code.

BNF-based English parser

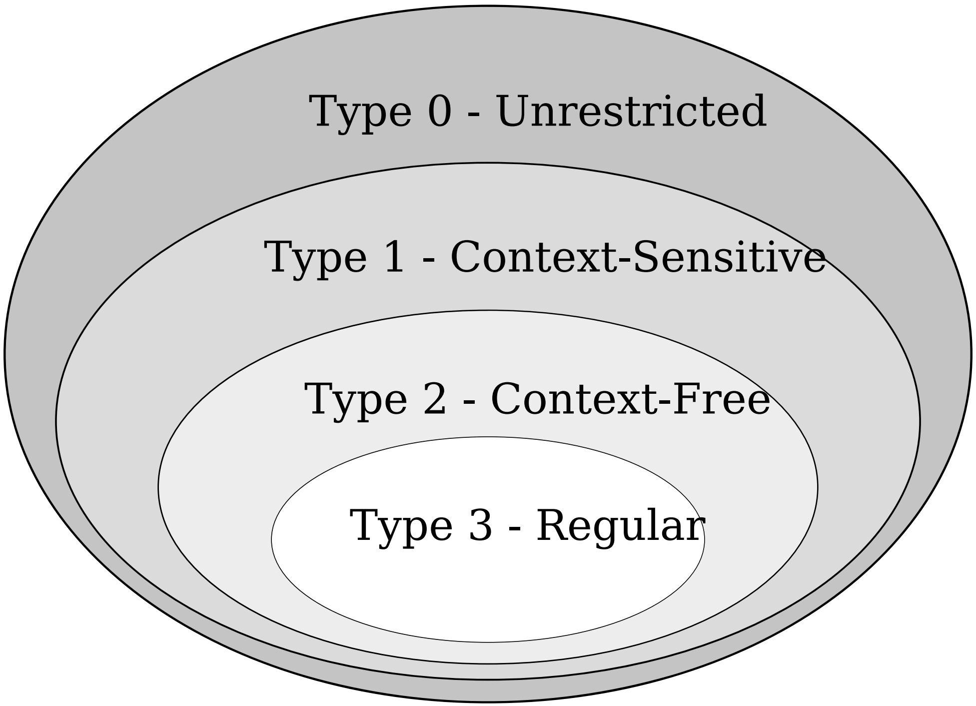
The NLTK Python library uses a BNF-like grammar for parsing sentences.

```
tokens = nltk.word_tokenize(sentence)
lil_grammar = nltk.CFG.fromstring("""
    S -> NP VP
    NP -> Det N | Det Adjs N
    Adjs -> Adj | Adjs Adj
    VP -> V | V NP | VP NP PP | V PP
    PP -> P NP
    Det -> 'the' | 'a'
    N -> 'fox' | 'dog' | 'cow'
    V -> 'jumped' | 'leaped'
    Adj -> 'brown' | 'lazy' | 'quick'
    P -> 'in' | 'over'
    """)
parser = nltk.ChartParser(lil_grammar)
tree = next(parser.parse(tokens))
```

Demo: NLTK Sentence Parsing

BNF in formal theory

CS172 discusses automata theory / language types.



For a quick version, watch this Computerphile video.