Overview

- *Typethon* is a *statically* and *strongly* typed general purpose programming language
- Derived from *Python*, it aims to be an improvement over the popular and beloved programming language
- Introduces new features:
  - Type safety:
    - use of allows for constants and immutability
    - explicit type of declared variables and constants, function parameters, and returns
    - static scoping using curly braces reduces type and scoping related bugs
  - Referencing:
    - memory address of a variable or object can be referenced using
    - use of pointers which point to the memory address of another variable
    - function arguments can be passed by reference which can make programs memory efficient by not taking up space when copying large data structures
- Programs in this language will have a `main()` function which will serve as the entry point

Example hello world program

```
def main() : none {
        print("hello world!")
}
```

Statements

- Variable declaration (`var <identifier> : <type>`)
- Pointer declaration(`ptr <indentifier> : <type>`)
- class declaration(`class <indetifier>`)
- object declaration(`var <identifier> = <Class>()`)
- Assignment (`<identifier> = <expression>`)
- If statement (`if` , `else` , `elif`)
- While statement (`while`)
- For statement (`for`)
- Function definition (`def`)
- Return statement (`return`)
- Continue statement (`continue`)
- Break statement (`break`)

Expressions

- Arithmetic expressions (`+` , `-` , `*` , `/` , `%`)
- Comparison expressions (`==` , `!=` , `<` , `>` , `<=` , `>=`)
- Boolean expressions (`and` , `or` , `not`)
- Function calls
- Literals (int, float, string, boolean)
- Collections (lists, dicts, sets, tuples)
- Range expressions (`..` , `..=`)
- Lambda expressions

Data Types

- Primitive types: int, float, str, chr, bool, none
- Collection types: list, dict, set, tuple

Identifiers

- Must start with a letter or underscore
- Can contain letters, numbers, and underscores
- used to denote variable, constants, objects, and functions

Keywords

- var, const, if, else, elif, while, for, fun, return, continue, break, true, false, lambda, in, not, and, or

```
<MainFunction> ::= "def" "main" "(" ")" ":" "none" "{" <Program>
"}"

<Program> ::= <StatementList>

<StatementList> ::= <Statement> | <Statement> <StatementList>

<Statement> ::= <VariableDeclaration>
                          | <ConstantDeclaration>
              | <Assignment>
              | <IfStatement>
              | <WhileStatement>
              | <FunctionDefinition>
              | <Comments>

<Comments> ::= <SingleLineComments> | <MultiLineComments>

<Comment> ::= "#" <AnythingButNewline>

<MultiLineComment> ::= <TripleQuotedString> <TripleQuotedString>
::= """ <Anything> """ | ''' <Anything> '''

<VariableDeclaration> ::= "var" <Identifier> ":" <Type>
```

```
                                                    | "var"
<Identifier> ":" <Type> "=" <Expression>
                                                    | "var"
<Identifier> = <Identifier> "(" [<ParameterList>] ")"

<ConstantDeclaration> ::= "const" <Identifier> : <Type>
                                                    | "const"
<Identifier> ":" <Type> "=" <Expression>
                                                    | "const"
<Identifier> = <Identifier> "(" [<ParameterList>] ")"

<PointerDeclaration> ::= "ptr" <Identifier> ":" <Type>

<Assignment> ::= <Identifier> "=" <Expression>

<IfStatement> ::= "if" "(" <BooleanExpression> ")" "{"
<StatementList> "}"
                | "if" "(" <BooleanExpression> ")" "{"
<StatementList> "}" "else" "{" <StatementList> "}"
                | "if" "(" <BooleanExpression> ")" "{"
<StatementList> "}" "elif" "{" <StatementList> "}"

<WhileStatement> ::= "while" "(" <BooleanExpression> ")" "{"
<StatementList> "}"

<ForStatement> ::= "for" "(" <Type> <identifier> "in"
(<RangeExpression> | <CollectionsType>) ")" "{" <StatementList> "}"

<ReturnStatement> ::= "return" <Expression>

<ContinueStatement> ::= "continue"
```

```
<BreakStatement> ::= "break"

<FunctionDefinition> ::= "def" <Identifier> "(" <ParameterList> ")"
":" <Type> "{" <StatementList> "}"

<FunctionApplication> ::= <Identifier> "(" (<Identifier>)* ")"

<ParameterList> ::= <Parameter> | <Parameter> "," <ParameterList>

<Parameter> ::= <Identifier> ":" <Type>

<Type> ::= "int"
                | "float"
                | "str"
                | "chr"
                | "bool"
                | <Identifier>
                | "none"
                | <CollectionsType>
                | "ptr"

<CollectionsType> := "list"
                                        | "dict"
                                        | "set"
                                        | "tuple"

<ListExpression> ::= "[" [<SeqExpression>] "]"

<TupleExpression> ::= "(" [<SeqExpression>] ")"

<SetExpression> ::= "[" [<SeqExpression] "]"
```

```
<DictExpression> ::= "{" [<KeyValuePairSequence>] "}"

<KeyValuePairSequence> ::= <KeyValuePairExpression> |
<KeyValuePairExpression> "," <KeyValuePairSequence>

<KeyValuePairExpression> ::= "Expression" ":" "Expression"

<SeqExpression> ::= <Expression> | <Expression> "," <SeqExpression>

<Expression> ::= <Term> (("+" | "-") <Term>)*
                                | <LambdaExpression>
                                | <ListExpression>
                                | <SetExpression>
                                | <TupleExpression>
                                | <DictExpression>
                                | <SequenceExpression>
                                | <KeyValuePairSequence>
                                | <KeyValuePairExpression>
                                | <BooleanExpression>
                                | <ComparisonExpression>
                                | <ReferenceExpression>

<ReferenceExpression> ::= "&" <Identifier>

<RangeExpression> ::= <Expression> ".." <Expression>
                    | <Expression> "..=" <Expression>

<ClassDefinition> ::= "class" <Identifier> ["(" <Identifier> ")"] "
{" <MemberList> "}"

<MemberList> ::= | <Member> | <Member> <MemberList>
```

```
<Member> ::= <VariableDeclaration> | <FunctionDefinition>

<LambdaExpression> ::= "lambda" <ParameterList> ":" <"Expression">

<BooleanExpression> ::= <BooleanTerm> | <BooleanExpression> "or"
<BooleanTerm>

<BooleanTerm> ::= <BooleanFactor> | <BooleanTerm> "and"
<BooleanFactor>

<BooleanFactor> ::= <BooleanLiteral> | <Identifier> | <Comparison>
| "(" <BooleanExpression> ")" | "not" <BooleanFactor>

<ComparisonExpression> ::= <Expression> <ComparisonOperator>
<Expression>

<ComparisonOperator> ::= "==" | "!=" | "<" | ">" | "<=" | ">="

<Term> ::= <Factor> (("*" | "/" | "%") <Factor>)*

<Factor> ::= <NumberLiteral>
                  | <identifier>
                  | "(" <Expression> ")"

<Literal> ::= <IntegerLiteral> | <FloatLiteral> | <StringLiteral> |
<BooleanLiteral>

<NumberLiteral> ::= <IntegerLiteral> | <FloatLiteral>

<IntegerLiteral> ::= [0-9]+

<FloatLiteral> ::= [0-9]+ "." [0-9]+
```

```
<StringLiteral> ::= "\"" <String> "\""

<String> ::= <Character> | <Character> <String>

<Character> ::= any printable character except "\""

<CharacterLiteral> ::= "'" <Character> "'"

<BooleanLiteral> ::= "true" | "false"

<Identifier> ::= [a-zA-Z_] [a-zA-Z0-9_]*
```

```
Dataview (inline field '='): Error:
-- PARSING FAILED ----------------------------------
---------------

> 1 | =
    | ^

Expected one of the following:

'(', 'null', boolean, date, duration, file link, list
('[1, 2, 3]'), negated field, number, object ('{ a:
1, b: 2 }'), string, variable
```