Skład grupy: Jakub Zborowski, Karol Woś, Sylwester Tylec

Temat: Tworzenie własnego języka programowania na podstawie języka python

Dokumentacja

Dokumentacja zawiera opis języka programowania stworzonego na podstawy przedmiotu Wytwarzanie aplikacji internetowych sterowanych modelami. Do utworzenia języka został wykorzystany język Python, w którym zostały zostały utworzone składnia i gramatyka języka. W budowaniu języka bazuje się na schematach funkcjonujących w języku bazowym, przystosowując je jednak do naszych potrzeb.

Pierwszym krokiem będzie utworzenie Lexera, czyli podprogramu interpretującego to, co użytkownik chce wprowadzić do programu. Innymi słowy to on decyduje o tym, jak język zrozumie wpisywane przez użytkownika komendy konsolowe. Stworzenie Lexera wymaga klasy Token, która intepretuje jakikolwiek wprowadzony w konsoli znak. Jest to „tlumacz” pomiędzy użytkownikiem, a metodami, które zostaną zaimplementowane do stworzenia języka.

```
TT_INT        = 'INT'
TT_FLOAT      = 'FLOAT'
TT_STRING     = 'STRING'
TT_IDENTIFIER = 'IDENTIFIER'
TT_KEYWORD    = 'KEYWORD'
TT_PLUS       = 'PLUS'
TT_MINUS      = 'MINUS'
TT_MUL        = 'MUL'
TT_DIV        = 'DIV'
TT_POW        = 'POW'
TT_EQ         = 'EQ'
TT_LPAREN     = 'LPAREN'
TT_RPAREN     = 'RPAREN'
TT_LSQUARE    = 'LSQUARE'
TT_RSQUARE    = 'RSQUARE'
TT_EE         = 'EE'
TT_NE         = 'NE'
TT_LT         = 'LT'
TT_GT         = 'GT'
TT_LTE        = 'LTE'
TT_GTE        = 'GTE'
TT_COMMA      = 'COMMA'
TT_ARROW      = 'ARROW'
TT_NEWLINE    = 'NEWLINE'
TT_EOF        = 'EOF'

KEYWORDS = [
  'VAR',
  'AND',
  'OR',
  'NOT',
  'IF',
  'ELIF',
  'ELSE',
  'FOR',
  'TO',
  'STEP',
  'WHILE',
  'FUN',
  'THEN',
  'END',
```

```
  'RETURN',
  'CONTINUE',
  'BREAK',
]

class Token:
  def __init__(self, type_, value=None, pos_start=None, pos_end=None):
    self.type = type_
    self.value = value

    if pos_start:
      self.pos_start = pos_start.copy()
      self.pos_end = pos_start.copy()
      self.pos_end.advance()

    if pos_end:
      self.pos_end = pos_end.copy()

  def matches(self, type_, value):
    return self.type == type_ and self.value == value

  def __repr__(self):
    if self.value: return f'{self.type}:{self.value}'
    return f'{self.type}'
```

Metoda klasy Lexer „skanuje” informacje wprowadzone przez użytkownika do konsoli znak po znaku i interpretuje je później na tokeny wedle ustalonych reguł. Znaki te są tłumaczone na tokeny, ktore mają przypisane już określone właściwości, co zostało zaprezentowane na rysunku 1. Należy przygotować jeszcze rozróżnienie na znaki i cyfry, ponieważ operacje na tych danych różnią się od siebie. Cyfry mogą być stało, bądź zmiennoprzecinkowe, należy więc przygotować metodę, która zintepretuje wpisany znak jako jeden z typów liczb. Metoda sprawdza, czy wprowadzone dane rzeczywiście są liczbą, w tym celu należy przygotować dodatkową tablicę, w której wypiszemy wszystkie cyfry, tj. od 0 do 9. Wprowadzone znaki są porównywane z tymi w tablicy. W przypadku, kiedy udało się znaleźć dopasowanie, zostało jeszcze stwierdzić, czy liczba jest stało, czy zmiennoprzecinkowa. W tym celu trzeba utworzyć instrukcję warunkową sprawdzającą występowanie we wprowadzonych danych kropki. W zależności od wyniku operacji porównania wprowadzone dane otrzymują jeden z tokenów. Tym sposobem zmienne zostają oznaczone jako typ int, bądź typ float. Następnie metoda zwraca odpowiedni token, z którego dalej będą korzystały inne metody języka.

```
class Lexer:
  def __init__(self, fn, text):
    self.fn = fn
    self.text = text
    self.pos = Position(-1, 0, -1, fn, text)
    self.current_char = None
    self.advance()

  def advance(self):
    self.pos.advance(self.current_char)
    self.current_char = self.text[self.pos.idx] if self.pos.idx < len(self.text) else None

  def make_tokens(self):
    tokens = []

    while self.current_char != None:
      if self.current_char in ' \t':
        self.advance()
      elif self.current_char in ';\n':
        tokens.append(Token(TT_NEWLINE, pos_start=self.pos))
        self.advance()
      elif self.current_char in DIGITS:
        tokens.append(self.make_number())
```

```
      elif self.current_char in LETTERS:
        tokens.append(self.make_identifier())
      elif self.current_char == '"':
        tokens.append(self.make_string())
      elif self.current_char == '+':
        tokens.append(Token(TT_PLUS, pos_start=self.pos))
        self.advance()
      elif self.current_char == '-':
        tokens.append(self.make_minus_or_arrow())
      elif self.current_char == '*':
        tokens.append(Token(TT_MUL, pos_start=self.pos))
        self.advance()
      elif self.current_char == '/':
        tokens.append(Token(TT_DIV, pos_start=self.pos))
        self.advance()
      elif self.current_char == '^':
        tokens.append(Token(TT_POW, pos_start=self.pos))
        self.advance()
      elif self.current_char == '(':
        tokens.append(Token(TT_LPAREN, pos_start=self.pos))
        self.advance()
      elif self.current_char == ')':
        tokens.append(Token(TT_RPAREN, pos_start=self.pos))
        self.advance()
      elif self.current_char == '[':
        tokens.append(Token(TT_LSQUARE, pos_start=self.pos))
        self.advance()
      elif self.current_char == ']':
        tokens.append(Token(TT_RSQUARE, pos_start=self.pos))
        self.advance()
      elif self.current_char == '!':
        token, error = self.make_not_equals()
        if error: return [], error
        tokens.append(token)
      elif self.current_char == '=':
        tokens.append(self.make_equals())
      elif self.current_char == '<':
        tokens.append(self.make_less_than())
      elif self.current_char == '>':
        tokens.append(self.make_greater_than())
      elif self.current_char == ',':
        tokens.append(Token(TT_COMMA, pos_start=self.pos))
        self.advance()
      else:
        pos_start = self.pos.copy()
        char = self.current_char
        self.advance()
        return [], IllegalCharError(pos_start, self.pos, "'" + char + "'")

    tokens.append(Token(TT_EOF, pos_start=self.pos))
    return tokens, None

  def make_number(self):
    num_str = ''
    dot_count = 0
    pos_start = self.pos.copy()

    while self.current_char != None and self.current_char in DIGITS + '.':
      if self.current_char == '.':
        if dot_count == 1: break
        dot_count += 1
      num_str += self.current_char
      self.advance()

    if dot_count == 0:
      return Token(TT_INT, int(num_str), pos_start, self.pos)
    else:
      return Token(TT_FLOAT, float(num_str), pos_start, self.pos)

  def make_string(self):
    string = ''
    pos_start = self.pos.copy()
    escape_character = False
```

```
    self.advance()

    escape_characters = {
      'n': '\n',
      't': '\t'
    }

    while self.current_char != None and (self.current_char != '"' or escape_character):
      if escape_character:
        string += escape_characters.get(self.current_char, self.current_char)
      else:
        if self.current_char == '\\':
          escape_character = True
        else:
          string += self.current_char
      self.advance()
      escape_character = False

    self.advance()
    return Token(TT_STRING, string, pos_start, self.pos)

  def make_identifier(self):
    id_str = ''
    pos_start = self.pos.copy()

    while self.current_char != None and self.current_char in LETTERS_DIGITS + '_':
      id_str += self.current_char
      self.advance()

    tok_type = TT_KEYWORD if id_str in KEYWORDS else TT_IDENTIFIER
    return Token(tok_type, id_str, pos_start, self.pos)

  def make_minus_or_arrow(self):
    tok_type = TT_MINUS
    pos_start = self.pos.copy()
    self.advance()

    if self.current_char == '>':
      self.advance()
      tok_type = TT_ARROW

    return Token(tok_type, pos_start=pos_start, pos_end=self.pos)

  def make_not_equals(self):
    pos_start = self.pos.copy()
    self.advance()

    if self.current_char == '=':
      self.advance()
      return Token(TT_NE, pos_start=pos_start, pos_end=self.pos), None

    self.advance()
    return None, ExpectedCharError(pos_start, self.pos, "'=' (after '!')")

  def make_equals(self):
    tok_type = TT_EQ
    pos_start = self.pos.copy()
    self.advance()

    if self.current_char == '=':
      self.advance()
      tok_type = TT_EE

    return Token(tok_type, pos_start=pos_start, pos_end=self.pos)

  def make_less_than(self):
    tok_type = TT_LT
    pos_start = self.pos.copy()
    self.advance()

    if self.current_char == '=':
      self.advance()
      tok_type = TT_LTE
```

```
    return Token(tok_type, pos_start=pos_start, pos_end=self.pos)

  def make_greater_than(self):
    tok_type = TT_GT
    pos_start = self.pos.copy()
    self.advance()

    if self.current_char == '=':
      self.advance()
      tok_type = TT_GTE

    return Token(tok_type, pos_start=pos_start, pos_end=self.pos)
```

Obsługa błędów powinna zawierać informację o niepoprawnie wykonanej operacji informując jednocześnie użytkownika o tym, co poszło nie tak. Dla zachowania przejrzystego stylu błędów należy dodać również informację o pliku i linii, w której odnaleziono przyczynę wywołania błędu.

```
class Error:
  def __init__(self, pos_start, pos_end, error_name, details):
    self.pos_start = pos_start
    self.pos_end = pos_end
    self.error_name = error_name
    self.details = details

  def as_string(self):
    result  = f'{self.error_name}: {self.details}\n'
    result += f'File {self.pos_start.fn}, line {self.pos_start.ln + 1}'
    result += '\n\n' + string_with_arrows(self.pos_start.ftxt, self.pos_start, self.pos_end)
    return result

class IllegalCharError(Error):
  def __init__(self, pos_start, pos_end, details):
    super().__init__(pos_start, pos_end, 'Illegal Character', details)

class ExpectedCharError(Error):
  def __init__(self, pos_start, pos_end, details):
    super().__init__(pos_start, pos_end, 'Expected Character', details)

class InvalidSyntaxError(Error):
  def __init__(self, pos_start, pos_end, details=''):
    super().__init__(pos_start, pos_end, 'Invalid Syntax', details)

class RTError(Error):
  def __init__(self, pos_start, pos_end, details, context):
    super().__init__(pos_start, pos_end, 'Runtime Error', details)
    self.context = context

  def as_string(self):
    result  = self.generate_traceback()
    result += f'{self.error_name}: {self.details}'
    result += '\n\n' + string_with_arrows(self.pos_start.ftxt, self.pos_start, self.pos_end)
    return result

  def generate_traceback(self):
    result = ''
    pos = self.pos_start
    ctx = self.context

    while ctx:
      result = f'  File {pos.fn}, line {str(pos.ln + 1)}, in {ctx.display_name}\n' + result
      pos = ctx.parent_entry_pos
      ctx = ctx.parent

    return 'Traceback (most recent call last):\n' + result
```

Klasa position zawiera metody służące do pobierania indeksu , pod którym znajduje się każdy z tokenów. Ułatwi to późniejsze poruszanie się po zmiennych i umożliwi zaproponowanie generowania eleganckiego komunikatu błędu.

```
class Position:
  def __init__(self, idx, ln, col, fn, ftxt):
    self.idx = idx
    self.ln = ln
    self.col = col
    self.fn = fn
    self.ftxt = ftxt

  def advance(self, current_char=None):
    self.idx += 1
    self.col += 1

    if current_char == '\n':
      self.ln += 1
      self.col = 0

    return self

  def copy(self):
    return Position(self.idx, self.ln, self.col, self.fn, self.ftxt)
```

Trzeba również przygotować metodę run, która wywoła instancję klasy lekser. Następnie metodę tę możemy wywoływać z zewnętrznego pliku powłoki po uprzednim imporcie pliku zawierającego logikę. Zostanie wyświetlony wynik, bądź zostanie zwrócony błąd.

```
global_symbol_table = SymbolTable()
global_symbol_table.set("NULL", Number.null)
global_symbol_table.set("FALSE", Number.false)
global_symbol_table.set("TRUE", Number.true)
global_symbol_table.set("MATH_PI", Number.math_PI)
global_symbol_table.set("PRINT", BuiltInFunction.print)
global_symbol_table.set("PRINT_RET", BuiltInFunction.print_ret)
global_symbol_table.set("INPUT", BuiltInFunction.input)
global_symbol_table.set("INPUT_INT", BuiltInFunction.input_int)
global_symbol_table.set("CLEAR", BuiltInFunction.clear)
global_symbol_table.set("CLS", BuiltInFunction.clear)
global_symbol_table.set("IS_NUM", BuiltInFunction.is_number)
global_symbol_table.set("IS_STR", BuiltInFunction.is_string)
global_symbol_table.set("IS_LIST", BuiltInFunction.is_list)
global_symbol_table.set("IS_FUN", BuiltInFunction.is_function)
global_symbol_table.set("APPEND", BuiltInFunction.append)
global_symbol_table.set("POP", BuiltInFunction.pop)
global_symbol_table.set("EXTEND", BuiltInFunction.extend)

def run(fn, text):
  # Generate tokens
  lexer = Lexer(fn, text)
  tokens, error = lexer.make_tokens()
  if error: return None, error

  # Generate AST
  parser = Parser(tokens)
  ast = parser.parse()
  if ast.error: return None, ast.error

  # Run program
  interpreter = Interpreter()
  context = Context('<program>')
  context.symbol_table = global_symbol_table
  result = interpreter.visit(ast.node, context)

  return result.value, result.error
```

```
basic > 1 + 2
[INT:1, PLUS, INT:2]
basic > 2.5 * 2.5
[FLOAT:2.5, MUL, FLOAT:2.5]
```

Jak widać na rysunku x język poprawnie interpretuje przekazywane mu zmienne, dając użytkownikowi informację zwrotną.

```
basic > 7 * $$$$$
Illegal Character: '$'
File <stdin>, line 1
```

Na rysunku x widać, że obsługa błędów informuje o niedozwolonym znaku i wskazuję linię kodu, w której wystąpił.

Kolejnym ważnym składnikiem języka jest parser. Decyduje on kolejności wykonywanych operacji. Dla przykładu w działaniach matematycznych operacje dzielenia i mnożenia mają wyższy priorytet od dodawania i odejmowania. Tym będzie zajmował się parser. Do poprawnego działania wykorzystuje on plik zawierający gramatykę języka.

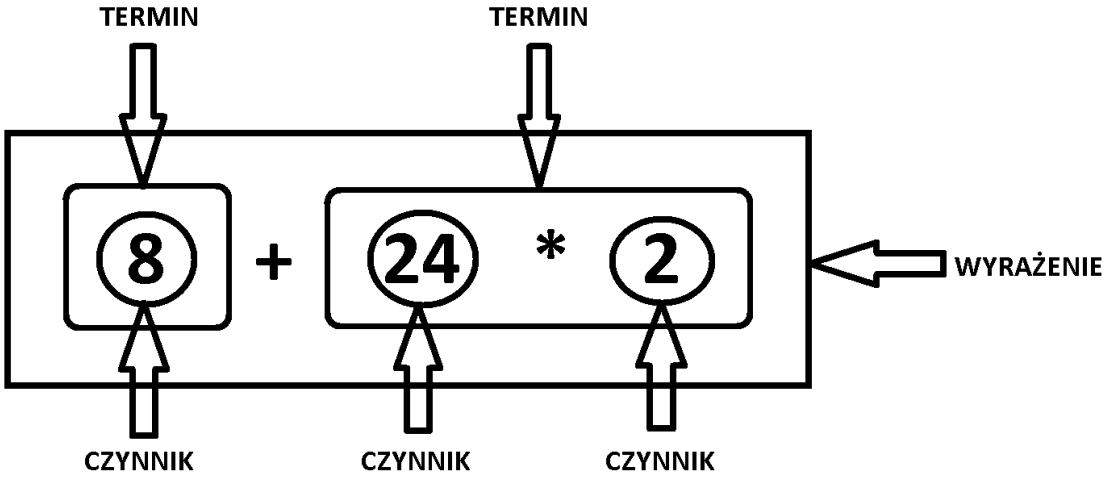
```
statements  : NEWLINE* statement (NEWLINE+ statement)* NEWLINE*

statement   : KEYWORD:RETURN expr?
            : KEYWORD:CONTINUE
            : KEYWORD:BREAK
            : expr

expr        : KEYWORD:VAR IDENTIFIER EQ expr
            : comp-expr ((KEYWORD:AND|KEYWORD:OR) comp-expr)*

comp-expr   : NOT comp-expr
            : arith-expr ((EE|LT|GT|LTE|GTE) arith-expr)*

arith-expr  : term ((PLUS|MINUS) term)*

term        : factor ((MUL|DIV) factor)*

factor      : (PLUS|MINUS) factor
            : power

power       : call (POW factor)*

call        : atom (LPAREN (expr (COMMA expr)*)? RPAREN)?

atom        : INT|FLOAT|STRING|IDENTIFIER
            : LPAREN expr RPAREN
            : list-expr
```

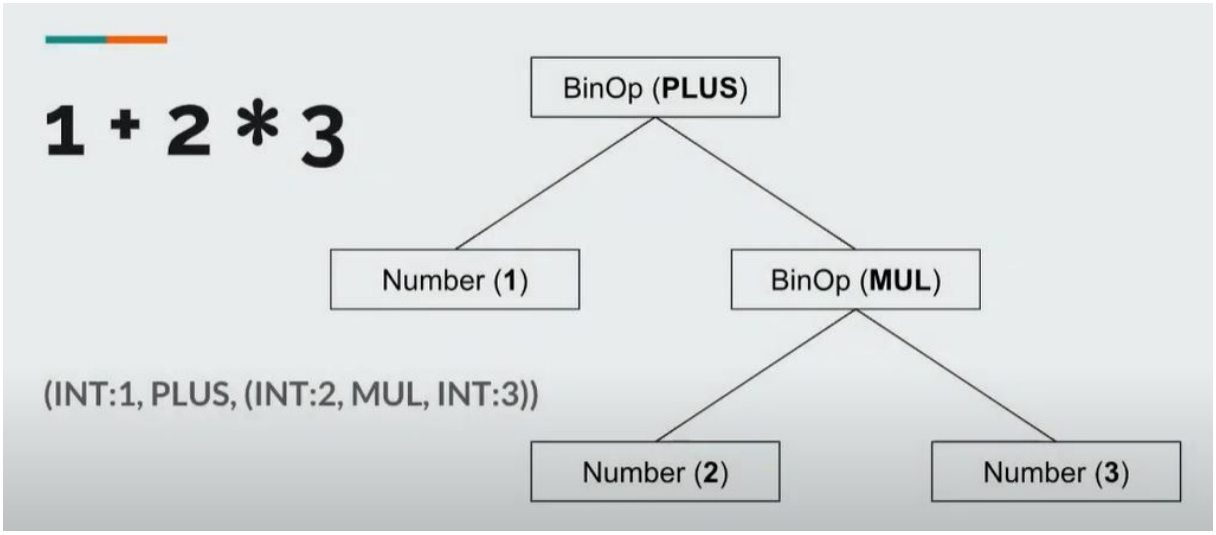
```
            : if-expr
            : for-expr
            : while-expr
            : func-def

list-expr   : LSQUARE (expr (COMMA expr)*)? RSQUARE

if-expr     : KEYWORD:IF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-b   : KEYWORD:ELIF expr KEYWORD:THEN
              (statement if-expr-b|if-expr-c?)
            | (NEWLINE statements KEYWORD:END|if-expr-b|if-expr-c)

if-expr-c   : KEYWORD:ELSE
              statement
            | (NEWLINE statements KEYWORD:END)

for-expr    : KEYWORD:FOR IDENTIFIER EQ expr KEYWORD:TO expr
              (KEYWORD:STEP expr)? KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

while-expr  : KEYWORD:WHILE expr KEYWORD:THEN
              statement
            | (NEWLINE statements KEYWORD:END)

func-def    : KEYWORD:FUN IDENTIFIER?
              LPAREN (IDENTIFIER (COMMA IDENTIFIER)*)? RPAREN
              (ARROW expr)
            | (NEWLINE statements KEYWORD:END)
```

Parser

Aby utworzyć własny język będzie nam do tego potrzebny parser, którego zadaniem jest zbudowanie drzewa składni z tokenów stworzonych przez lekser poprzez pilnowanie kolejności działań, sprawdzanie czy tokeny pasują do gramatyki naszego języka oraz czy generują odpowiednio drzewo .

Gdy parser napotyka złożone wyrażenie to wtedy rozdziela je na kilka osobnych terminów, a te natomiast są dzielone na poszczególne czynniki, które są jedynie liczbami.





Drzewo składni

Gramatyka jest podzielona na kilka różnych typów węzłów:

- numeryczne – przyjmuje odpowiedni token liczby, więc będzie to liczba całkowita lub zmiennoprzecinkowa,

- operacji binarnej – rozdziela operację(np: 2*3) na 3 czynniki: lewy węzeł(2), token operacji(*) i prawy węzeł(3)

-jednoargumentowe – sprawdza występujący znak przed liczbą(liczby dodatnie i ujemne)

Klasa parsera

Parser jest zmuszony do śledzenia bieżącego indeksu tokena podobnie jak lekser, a każda reguła z gramatyki ma odpowiadającą metodę w parserze.

```
def update_current_tok(self):
    if self.tok_idx >= 0 and self.tok_idx < len(self.tokens):
      self.current_tok = self.tokens[self.tok_idx]
    return self.current_tok
```

CZYNNIK

Na początku bierze bieżący token, następnie sprawdza czy ten token jest typu int albo float – jeśli jest to zwiększamy token o jeden i możemy zwrócić węzeł numeryczny. Dodatkowo jest sprawdzany znak występujący przed liczbą, aby były dostępne również liczby ujemne.

```
def make_number(self):
    num_str = ''
    dot_count = 0
    pos_start = self.pos.copy()

while self.current_char != None and self.current_char in DIGITS + '.':
      if self.current_char == '.':
        if dot_count == 1: break
        dot_count += 1
      num_str += self.current_char
      self.advance()

    if dot_count == 0:
      return Token(TT_INT, int(num_str), pos_start, self.pos)
    else:
      return Token(TT_FLOAT, float(num_str), pos_start, self.pos)

def advance(self):
    self.tok_idx += 1
    self.update_current_tok()
    return self.current_tok

 def atom(self):
    res = ParseResult()
    tok = self.current_tok

    if tok.type in (TT_INT, TT_FLOAT):
      res.register_advancement()
      self.advance()
      return res.success(NumberNode(tok))

def factor(self):
    res = ParseResult()
    tok = self.current_tok

    if tok.type in (TT_PLUS, TT_MINUS):
      res.register_advancement()
      self.advance()
      factor = res.register(self.factor())
      if res.error: return res
      return res.success(UnaryOpNode(tok, factor))

    return self.power()
```

TERMIN

Na początku szukamy lewego czynnika – korzystamy z funkcji czynnika, następnie w pętli sprawdzamy czy bieżący token jest mnożony czy dzielony(symbol * lub /) – jeśli tak jest to możemy pobrać token, a następnie pobieramy prawy czynnik. Więc teraz pomiędzy tymi dwoma czynnikami możemy utworzyć operację binarną – przekazujemy lewemu czynnikowi token operacji, a następnie prawy czynnik. Kiedy skończymy możemy po prostu zwrócić lewy czynnik, który jest teraz właściwie węzłem operacji binarnej. Używanie pętli pozwala nam na wiele operacji mnożenia bądź dzielenia w obrębie jednego terminu.

```
class BinOpNode:
  def __init__(self, left_node, op_tok, right_node):
    self.left_node = left_node
    self.op_tok = op_tok
    self.right_node = right_node

    self.pos_start = self.left_node.pos_start
    self.pos_end = self.right_node.pos_end

def term(self):
    return self.bin_op(self.factor, (TT_MUL, TT_DIV))
```

WYRAŻENIE

Działanie w tym przypadku jest bardzo podobne do tego z terminu, z tą różnicą że teraz szukamy plusa bądź minusa.

```
class BinOpNode:
  def __init__(self, left_node, op_tok, right_node):
    self.left_node = left_node
    self.op_tok = op_tok
    self.right_node = right_node

    self.pos_start = self.left_node.pos_start
    self.pos_end = self.right_node.pos_end

def arith_expr(self):
    return self.bin_op(self.term, (TT_PLUS, TT_MINUS))
```

PARSE RESULT

Parse Result jest to klasa, która śledzi czy wystąpił jakiś błąd, a jeśli tak to wskazuje na węzeł, w którym to się wydarzyło.

```
class ParseResult:
  def __init__(self):
    self.error = None
    self.node = None
    self.last_registered_advance_count = 0
    self.advance_count = 0
    self.to_reverse_count = 0

  def register_advancement(self):
    self.last_registered_advance_count = 1
    self.advance_count += 1

  def register(self, res):
    self.last_registered_advance_count = res.advance_count
    self.advance_count += res.advance_count
    if res.error: self.error = res.error
    return res.node

  def try_register(self, res):
    if res.error:
      self.to_reverse_count = res.advance_count
      return None
    return self.register(res)

  def success(self, node):
    self.node = node
    return self

  def failure(self, error):
    if not self.error or self.last_registered_advance_count == 0:
      self.error = error
    return self
```

Każdą operację zawijamy do rejestru, dzięki czemu będzie możliwość przypisania błędu do samego węzła, a nie do całego wyniku przejścia. Dodatkowo w kodzie zostały dodane wcześniej pokazane na listingach zmienne – pos_start i pos_end, które oznaczają konkretne miejsce w kodzie i dzięki temu wskazują miejsce błędu.

```
def bin_op(self, func_a, ops, func_b=None):
  if func_b == None:
    func_b = func_a

  res = ParseResult()
  left = res.register(func_a())
  if res.error: return res

  while self.current_tok.type in ops or (self.current_tok.type, self.current_tok.value) in
ops:
    op_tok = self.current_tok
    res.register_advancement()
    self.advance()
    right = res.register(func_b())
    if res.error: return res
    left = BinOpNode(left, op_tok, right)

  return res.success(left)
```

Parser korzysta również z tokenu końca pliku, dzięki któremu możemy sprawdzać czy gdy wystąpił błąd to czy nie nastąpił koniec pliku, co oznaczałoby wystąpienie błędu składni. Dzięki parserowi są również wyrzucane błędy, np.: brak spodziewanego znaku, błędny typ liczby czy brak prawego nawiasu po lewym.

```
  def parse(self):
    res = self.statements()
    if not res.error and self.current_tok.type != TT_EOF:
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        "Token cannot appear after previous tokens"
      ))
    return res


  expr = res.register(self.expr())
    if res.error:
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        "Expected 'RETURN', 'CONTINUE', 'BREAK', 'VAR', 'IF', 'FOR',
'WHILE', 'FUN', int, float, identifier, '+', '-', '(', '[' or 'NOT'"
      ))
    return res.success(expr)
```

Interpreter

Kolejnym krokiem niezbędnym do utworzenia własnego języka programowania jest stworzenie interpretera. Jest to program pozwalający na skompilowanie kodu języka, który będziemy tworzyć. Interpreter analizuje każdą linijkę kodu źródłowego programu a następnie wykonuje przeanalizowane fragmenty. Kod programu zostaje przetłumaczony do kodu maszynowego(lub kodu pośredniego) a następnie jest zapisywany w celu późniejszego wykonania.

Klasa interpreter posiada metodę visit która przyjmuje węzeł, następnie przetwarza węzeł i odwiedza wszystkie węzły potomne

Tworzymy definicje metody visit która użyje funkcji getattribute() którą przekazuje a następnie nazwę metody i metodę domyślną jeśli nie zostanie znaleziona żadna metoda. Teraz metoda zwraca węzeł.

```
class Interpreter:
  def visit(self, node, context):
    method_name = f'visit_{type(node).__name__}'
    method = getattr(self, method_name, self.no_visit_method)
    return method(node, context)
```

Poniżej metody znajduje się definicja metody no_visit_method komunikująca o braku powyższej metody, która polega na zgłoszeniu wyjątku wyświetlającego dokładną nazwę metody.

```
def no_visit_method(self, node, context):
    raise Exception(f'No visit_{type(node).__name__} method defined')
```

Następnie towrzymy kolejne węzły: numeru, operatora binarnego, jednoargumentowego, itd.

```
def visit_NumberNode(self, node, context):
    return RTResult().success(
      Number(node.tok.value).set_context(context).set_pos(node.pos_start, node.pos_end)
    )
def visit_UnaryOpNode(self, node, context):
    res = RTResult()
    number = res.register(self.visit(node.node, context))
    if res.should_return(): return res

    error = None

    if node.op_tok.type == TT_MINUS:
      number, error = number.multed_by(Number(-1))
    elif node.op_tok.matches(TT_KEYWORD, 'NOT'):
      number, error = number.notted()

    if error:
      return res.failure(error)
    else:
      return res.success(number.set_pos(node.pos_start, node.pos_end))
def visit_BinOpNode(self, node, context):
    res = RTResult()
    left = res.register(self.visit(node.left_node, context))
    if res.should_return(): return res
    right = res.register(self.visit(node.right_node, context))
    if res.should_return(): return res

    if node.op_tok.type == TT_PLUS:
```

```
      result, error = left.added_to(right)
    elif node.op_tok.type == TT_MINUS:
      result, error = left.subbed_by(right)
    elif node.op_tok.type == TT_MUL:
      result, error = left.multed_by(right)
    elif node.op_tok.type == TT_DIV:
      result, error = left.dived_by(right)
    elif node.op_tok.type == TT_POW:
      result, error = left.powed_by(right)
    elif node.op_tok.type == TT_EE:
      result, error = left.get_comparison_eq(right)
    elif node.op_tok.type == TT_NE:
      result, error = left.get_comparison_ne(right)
    elif node.op_tok.type == TT_LT:
      result, error = left.get_comparison_lt(right)
    elif node.op_tok.type == TT_GT:
      result, error = left.get_comparison_gt(right)
    elif node.op_tok.type == TT_LTE:
      result, error = left.get_comparison_lte(right)
    elif node.op_tok.type == TT_GTE:
      result, error = left.get_comparison_gte(right)
    elif node.op_tok.matches(TT_KEYWORD, 'AND'):
      result, error = left.anded_by(right)
    elif node.op_tok.matches(TT_KEYWORD, 'OR'):
      result, error = left.ored_by(right)

    if error:
      return res.failure(error)
    else:
      return res.success(result.set_pos(node.pos_start, node.pos_end))
```

Po utworzeniu węzłów w funkcji RUN tworzymy instancję interpretera

```
# Run program
  interpreter = Interpreter()
  context = Context('<program>')
  context.symbol_table = global_symbol_table
  result = interpreter.visit(ast.node, context)
```

Oczywiście program nie będzie działał bez stworzenia klas opisujących operacji na liczbach takich jak dodawanie, odejmowanie, mnożenie, dzielenie

```
def __init__(self, value):
    super().__init__()
    self.value = value

  def added_to(self, other):
    if isinstance(other, Number):
      return Number(self.value + other.value).set_context(self.context), None
    else:
      return None, Value.illegal_operation(self, other)

  def subbed_by(self, other):
    if isinstance(other, Number):
      return Number(self.value - other.value).set_context(self.context), None
    else:
```

```
      return None, Value.illegal_operation(self, other)
```

Funkcja IF

Składnia funkcji warunkowej if, która porównuje ze sobą poszczególne wartości wygląda następująco: Słowo kluczowe IF po którym następuje warunek, następnie słowo kluczowe THEN po którym następuje wyrażenie, które występuje jeśli warunek jest spełniony. Będzie również możliwość użycia słowa kluczowego ELIF po którym można wprowadzić kolejny warunek i wyrażenie. Drugą główną częścią funkcji if jest alternatywny wynik porównania, służy do tego słowo kluczowe ELSE po którym następuje wyrażenie alternatywne, które zostanie wykonane w przypadku, gdy żaden z poprzednich warunków nie został spełniony. Pierwszą czynnością przy dodawaniu funkcji do naszego języka jest dodanie do tokenów słów kluczowych używanych w funkcji:

```
KEYWORDS = [
  'VAR',
  'AND',
  'OR',
  'NOT',
  'IF',
  'ELIF',
  'ELSE']
```

Następnie piszemy kod umożliwiający działanie funkcji:

```
def if_expr_cases(self, case_keyword):
    res = ParseResult()
    cases = []
    else_case = None

    if not self.current_tok.matches(TT_KEYWORD, case_keyword):
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        f"Expected '{case_keyword}'"
      ))

    res.register_advancement()
    self.advance()

    condition = res.register(self.expr())
    if res.error: return res

    if not self.current_tok.matches(TT_KEYWORD, 'THEN'):
      return res.failure(InvalidSyntaxError(
        self.current_tok.pos_start, self.current_tok.pos_end,
        f"Expected 'THEN'"
      ))

    res.register_advancement()
    self.advance()

    if self.current_tok.type == TT_NEWLINE:
      res.register_advancement()
      self.advance()
```

```
      statements = res.register(self.statements())
      if res.error: return res
      cases.append((condition, statements, True))

      if self.current_tok.matches(TT_KEYWORD, 'END'):
        res.register_advancement()
        self.advance()
      else:
        all_cases = res.register(self.if_expr_b_or_c())
        if res.error: return res
        new_cases, else_case = all_cases
        cases.extend(new_cases)
    else:
      expr = res.register(self.statement())
      if res.error: return res
      cases.append((condition, expr, False))

      all_cases = res.register(self.if_expr_b_or_c())
      if res.error: return res
      new_cases, else_case = all_cases
      cases.extend(new_cases)

    return res.success((cases, else_case))
```

Ostatnią czynnością wykonywaną przy tworzeniu funkcji warunkowej if jest zaktualizowanie interpretera poprzez dodanie definicji naszej funkcji:

```
def visit_IfNode(self, node, context):
    res = RTResult()

    for condition, expr, should_return_null in node.cases:
      condition_value = res.register(self.visit(condition, context))
      if res.should_return(): return res

      if condition_value.is_true():
        expr_value = res.register(self.visit(expr, context))
        if res.should_return(): return res
        return res.success(Number.null if should_return_null else expr_value)

    if node.else_case:
      expr, should_return_null = node.else_case
      expr_value = res.register(self.visit(expr, context))
      if res.should_return(): return res
      return res.success(Number.null if should_return_null else expr_value)

    return res.success(Number.null)
```