Segundo Projeto: Analisador Sintático

O segundo projeto requer que você implemente um analisador sintático para a linguagem uChuck e gere uma árvore de sintaxe como saída (observe que a árvore de sintaxe abstrata será construída apenas no terceiro projeto). Para concluir este segundo projeto, você também usará o SLY, uma versão Python do conjunto de ferramentas lex/yacc com a mesma funcionalidade mas com uma interface mais amigável. Por favor, leia o conteúdo completo desta seção e complete cuidadosamente as etapas indicadas.

Visão Geral do Analisador Sintático: Construíndo uma Árvore de Sintaxe

Considere a seguinte gramática:

```
<program> ::= <statements> EOF

<statements> ::= { <statement> }+

<statement> ::= <print_statement>
              | <assign_statement>
              | <if_statement>

<print_statement> ::= PRINT <expr> SEMI

<assign_statement> ::= <identifier> EQUALS <expr> SEMI

<if_statement> ::= IF <expr> LBRACE <statements> RBRACE { ELSE LBRACE <statements> RBRACE }?

<expr> ::= <expr> PLUS <expr>
         | <expr> TIMES <expr>
         | <expr> EQ <expr>
         | <expr> NE <expr>
         | <constant>
         | <identifier>
         | LPAREN <expr> RPAREN

<identifier> ::= ID

<constant> ::= NUM
```

Nessa gramática, a seguinte sintaxe é usada:

```
 {símbolos}*  ==> Zero ou mais repetições de símbolos
 {símbolos}+  ==> Um ou mais repetições de símbolos
 {símbolos}?  ==> Zero ou uma ocorrência de símbolos (opcional)
 sim1 | sim2  ==> Ou sim1 ou sim2 (uma escolha)
```

Alguns exemplos de sentenças válidas para essa gramática são:

```
a = 3 * 4 + 5 ;
```

```
print a ;
```

```
a = 3;
b = 4 * a;
print (a+b);
```

```
a = 7;
b = 10;
c = 5;

if a*a == b*b + c*c {
   print 1;
```

```
} else {
   print 0;
}
```

Analisador Léxico

O primeiro passo é construir um analisador léxico para os terminais dessa gramática:

```
!pip install sly
```

```
from sly import Lexer

class MyLexer(Lexer):
    """A lexer for the Grammar."""

    # Reserved keywords
    keywords = {
        'print': "PRINT",
        'if': "IF",
        'else': "ELSE",
    }

    # All the tokens recognized by the lexer
    tokens = tuple(keywords.values()) + (
        # Identifiers
        "ID",
        # Constants
        "NUM",
        # Operators
        "PLUS",
        "TIMES",
        "EQ",
        "NE",
        # Assignment
        "EQUALS",
        # Delimeters
        "LPAREN",
        "RPAREN",  # ( )
        "LBRACE",
        "RBRACE",  # { }
        "SEMI",    # ;
    )

    # String containing ignored characters (between tokens)
    ignore = " \t"

    # Regular expression rules for tokens
    ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
    NUM = r'\d+'
    PLUS = r'\+'
    TIMES = r'\*'
    NE = r'!='
    EQ = r'=='
    EQUALS = r'='
    SEMI = r';'
    LPAREN = r'\('
    RPAREN = r'\)'
    LBRACE = r'{'
    RBRACE = r'}'

    # Special cases
    def ID(self, t):
        t.type = self.keywords.get(t.value, "ID")
        return t

    def NUM(self, t):
        t.value = int(t.value)
        return t

    # Define a rule so we can track line numbers
    @_(r'\n+')
    def ignore_newline(self, t):
        self.lineno += len(t.value)

    def find_column(self, token):
        """Find the column of the token in its line."""
        last_cr = self.text.rfind('\n', 0, token.index)
        return token.index - last_cr
```

```
    # Error handling rule
    def error(self, t):
        print("Illegal character %s at position (%d,%d)" % \
              (repr(t.value[0]), t.lineno, self.find_column(t)))
        self.index += 1


def main(args):
    if len(args) > 0:
        lex = MyLexer()
        for tok in lex.tokenize(args[0]):
            print(tok)


main(["a = 3 * 4 + 5 ;"])
```

Reconhecendo as sentenças

Agora, vamos desenvolver um analisador sintático para reconhecer sentenças dessa gramática e construir uma árvore de sintaxe para elas.

```
from sly import Parser

class MyParser(Parser):
    """A parser for the Grammar."""

    # Get the token list from the lexer (required)
    tokens = MyLexer.tokens

    def __init__(self):
        self.lexer = MyLexer()

    def parse(self, text, lineno=1, index=0):
        return super().parse(self.lexer.tokenize(text, lineno, index))

    # <program> ::= <statements> EOF
    @_('statements')
    def program(self, p):
        pass

    # <statements> ::= { <statement> }+
    @_('statement { statement }')
    def statements(self, p):
        pass

    # <statement> ::= <print_statement>
    #               | <assign_statement>
    #               | <if_statement>
    @_('print_statement',
       'assign_statement',
       'if_statement')
    def statement(self, p):
        pass

    # <print_statement> ::= PRINT <expr> SEMI
    @_('PRINT expr SEMI')
    def print_statement(self, p):
        pass

    # <assign_statement> ::= <identifier> EQUALS <expr> SEMI
    @_('identifier EQUALS expr SEMI')
    def assign_statement(self, p):
        pass

    # <if_statement> ::= IF <expr> LBRACE <statements> RBRACE { ELSE LBRACE <statements> RBRACE }?
    @_('IF expr LBRACE statements RBRACE [ ELSE LBRACE statements RBRACE ]')
    def if_statement(self, p):
        pass

    # <expr> ::= <expr> PLUS <expr>
    #          | <expr> TIMES <expr>
    #          | <expr> EQ <expr>
    #          | <expr> NE <expr>
    @_('expr PLUS expr',
       'expr TIMES expr',
       'expr EQ expr',
       'expr NE expr')
    def expr(self, p):
        pass
```

```
        # <expr> ::= <constant>
        #          | <identifier>
        @_('constant',
           'identifier')
        def expr(self, p):
            pass

        # <expr> ::= LPAREN <expr> RPAREN
        @_('LPAREN expr RPAREN')
        def expr(self, p):
            pass

        # <constant> ::= NUM
        @_('NUM')
        def constant(self, p):
            pass

        # <identifier> ::= ID
        @_('ID')
        def identifier(self, p):
            pass


    def build_tree(root):
        return '\n'.join(_build_tree(root))

    def _build_tree(node):
        if isinstance(node, list):
            if not node: return
            node = tuple(node)

        if not isinstance(node, tuple):
            yield " "+str(node)
            return

        values = [_build_tree(n) for n in node]
        if len(values) == 1:
            yield from build_lines('──', '  ', values[0])
            return

        start, *mid, end = values
        yield from build_lines('┬─', '│ ', start)
        for value in mid:
            yield from build_lines('├─', '│ ', value)
        yield from build_lines('└─', '  ', end)

    def build_lines(first, other, values):
        try:
            yield first + next(values)
            for value in values:
                yield other + value
        except StopIteration:
            return


    def main(args):
        if len(args) > 0:
            lex = MyLexer()
            parser = MyParser()
            st = parser.parse(args[0])
            if st is not None:
                print(build_tree(st))
```

Vamos ignorar os avisos por enquanto e tentar analisar uma sentença válida:

```
main(["a = 3 * 4 + 5 ;"])
```

Nada aconteceu! Vamos ver o que acontece com uma sentença inválida:

```
main(["a == 3 ;"])
```

Agora vamos adicionar informações para construir uma árvore de sintaxe:

```
from sly import Parser

class MyParser(Parser):
    """A parser for the Grammar."""
```

```
        # Get the token list from the lexer (required)
        tokens = MyLexer.tokens

        def __init__(self):
            self.lexer = MyLexer()

        def parse(self, text, lineno=1, index=0):
            return super().parse(self.lexer.tokenize(text, lineno, index))

        # <program> ::= <statements> EOF
        @_('statements')
        def program(self, p):
            return ('program', p.statements)

        # <statements> ::= { <statement> }+
        @_('statement { statement }')
        def statements(self, p):
            return [p.statement0] + p.statement1

        # <statement> ::= <print_statement>
        #               | <assign_statement>
        #               | <if_statement>
        @_('print_statement',
           'assign_statement',
           'if_statement')
        def statement(self, p):
            return p[0]

        # <print_statement> ::= PRINT <expr> SEMI
        @_('PRINT expr SEMI')
        def print_statement(self, p):
            return ('print', p.expr)

        # <assign_statement> ::= <identifier> EQUALS <expr> SEMI
        @_('identifier EQUALS expr SEMI')
        def assign_statement(self, p):
            return ('assign', p.identifier, p.expr)

        # <if_statement> ::= IF <expr> LBRACE <statements> RBRACE { ELSE LBRACE <statements> RBRACE }?
        @_('IF expr LBRACE statements RBRACE [ ELSE LBRACE statements RBRACE ]')
        def if_statement(self, p):
            return ('if', p.expr, p.statements0, p.statements1)

        # <expr> ::= <expr> PLUS <expr>
        #          | <expr> TIMES <expr>
        #          | <expr> EQ <expr>
        #          | <expr> NE <expr>
        @_('expr PLUS expr',
           'expr TIMES expr',
           'expr EQ expr',
           'expr NE expr')
        def expr(self, p):
            return ('expr: %s' % (p[1]), p.expr0, p.expr1)

        # <expr> ::= <constant>
        #          | <identifier>
        @_('constant',
           'identifier')
        def expr(self, p):
            return p[0]

        # <expr> ::= LPAREN <expr> RPAREN
        @_('LPAREN expr RPAREN')
        def expr(self, p):
            return p.expr

        # <constant> ::= NUM
        @_('NUM')
        def constant(self, p):
            return ('num: %s'% (p.NUM))

        # <identifier> ::= ID
        @_('ID')
        def identifier(self, p):
            return ('id: %s' % (p.ID))


main(["a = 3 * 4 + 5 ;"])
```

Construída a árvore de sintaxe, notamos que ela não está respeitando a precedência dos operadores. Precisamos indicar isso no analisador sintático. Isso é feito adicionando a variável `precedence` à classe do analisador. Consulte a seção "Lidando com Gramáticas Ambíguas" da

documentação do SLY.

Vamos também aproveitar e definir uma rotina de erro para o analisador sintático:

```
from sly import Parser

class MyParser(Parser):
    """A parser for the Grammar."""

    # Get the token list from the lexer (required)
    tokens = MyLexer.tokens

    precedence = (
        ('left', 'PLUS'),
        ('left', 'TIMES'),
        ('left', 'EQ'),
        ('left', 'NE'),
    )

    def __init__(self):
        self.lexer = MyLexer()

    def parse(self, text, lineno=1, index=0):
        return super().parse(self.lexer.tokenize(text, lineno, index))

    # Error handling rule
    def error(self, p):
        if p:
            if hasattr(p, 'lineno'):
                print("Error at line %d near the symbol %s " % (p.lineno, p.value))
            else:
                print("Error near the symbol %s" % p.value)
        else:
            print("Error at the end of input")

    # <program> ::= <statements>
    @_('statements')
    def program(self, p):
        return ('program', p.statements)

    # <statements> ::= { <statement> }+
    @_('statement { statement }')
    def statements(self, p):
        return [p.statement0] + p.statement1

    # <statement> ::= <print_statement>
    #               | <assign_statement>
    #               | <if_statement>
    @_('print_statement',
       'assign_statement',
       'if_statement')
    def statement(self, p):
        return p[0]

    # <print_statement> ::= PRINT <expr> SEMI
    @_('PRINT expr SEMI')
    def print_statement(self, p):
        return ('print', p.expr)

    # <assign_statement> ::= <identifier> EQUALS <expr> SEMI
    @_('identifier EQUALS expr SEMI')
    def assign_statement(self, p):
        return ('assign', p.identifier, p.expr)

    # <if_statement> ::= IF <expr> LBRACE <statements> RBRACE { ELSE LBRACE <statements> RBRACE }?
    @_('IF expr LBRACE statements RBRACE [ ELSE LBRACE statements RBRACE ]')
    def if_statement(self, p):
        return ('if', p.expr, p.statements0, p.statements1)

    # <expr> ::= <expr> PLUS <expr>
    #          | <expr> TIMES <expr>
    #          | <expr> EQ <expr>
    #          | <expr> NE <expr>
    @_('expr PLUS expr',
       'expr TIMES expr',
       'expr EQ expr',
       'expr NE expr')
    def expr(self, p):
        return ('expr: %s' % (p[1]), p.expr0, p.expr1)

    # <expr> ::= <constant>
    #          | <identifier>
    @_('constant',
```

```
    @_( constant ,
       'identifier')
    def expr(self, p):
        return p[0]

    # <expr> ::= LPAREN <expr> RPAREN
    @_('LPAREN expr RPAREN')
    def expr(self, p):
        return p.expr

    # <constant> ::= NUM
    @_('NUM')
    def constant(self, p):
        return ('num: %s'% (p.NUM))

    # <identifier> ::= ID
    @_('ID')
    def identifier(self, p):
        return ('id: %s' % (p.ID))
```

Com a definição de precedência de operadores, os conflitos empilha/reduz foram resolvidos. Vamos executar o analisador sintático para nossas sentenças de exemplo novamente:

```
main(["a = 3 * 4 + 5 ;"])
```

```
main(["a == 3 ; "])
```

Para fins de rastreamento de posição, você deve salvar o número de linha e coluna em cada produção. Consulte a seção "Número de Linha e Rastreamento de Posição" da documentação do SLY. Para fazer isso, sugiro extrair o número da linha e da coluna do símbolo terminal mais à esquerda de cada produção, se houver. Por exemplo:

```
    # Internal auxiliary methods
    def _token_coord(self, p):
        return p.lineno, self.lexer.find_column(p)


    # <assign_statement> ::= <identifier> EQUALS <expr> SEMI
    @_('identifier EQUALS expr SEMI')
    def assign_statement(self, p):
        return ('assign @ %d:%d' % self._token_coord(p), p.identifier, p.expr)
```

Vamos ajustar o analisador sintático para rastrear a posição de símbolos terminais e executá-lo em nossas sentenças de exemplo novamente:

```
from sly import Parser

class MyParser(Parser):
    """A parser for the Grammar."""

    # Get the token list from the lexer (required)
    tokens = MyLexer.tokens

    precedence = (
        ('left', 'PLUS'),
        ('left', 'TIMES'),
        ('left', 'EQ'),
        ('left', 'NE'),
    )

    def __init__(self):
        self.lexer = MyLexer()

    def parse(self, text, lineno=1, index=0):
        return super().parse(self.lexer.tokenize(text, lineno, index))

    # Internal auxiliary methods
    def _token_coord(self, p):
        return p.lineno, self.lexer.find_column(p)

    # Error handling rule
    def error(self, p):
        if p:
            if hasattr(p, 'lineno'):
                print("Error at line %d near the symbol %s " % (p.lineno, p.value))
            else:
```

```
                print("Error near the symbol %s" % p.value)
            else:
                print("Error at the end of input")

        # <program> ::= <statements>
        @_('statements')
        def program(self, p):
            return ('program', p.statements)

        # <statements> ::= { <statement> }+
        @_('statement { statement }')
        def statements(self, p):
            return [p.statement0] + p.statement1

        # <statement> ::= <print_statement>
        #               | <assign_statement>
        #               | <if_statement>
        @_('print_statement',
           'assign_statement',
           'if_statement')
        def statement(self, p):
            return p[0]

        # <print_statement> ::= PRINT <expr> SEMI
        @_('PRINT expr SEMI')
        def print_statement(self, p):
            return ('print @ %d:%d' % self._token_coord(p), p.expr)

        # <assign_statement> ::= <identifier> EQUALS <expr> SEMI
        @_('identifier EQUALS expr SEMI')
        def assign_statement(self, p):
            return ('assign @ %d:%d' % self._token_coord(p), p.identifier, p.expr)

        # <if_statement> ::= IF <expr> LBRACE <statements> RBRACE { ELSE LBRACE <statements> RBRACE }?
        @_('IF expr LBRACE statements RBRACE [ ELSE LBRACE statements RBRACE ]')
        def if_statement(self, p):
            return ('if @ %d:%d' % self._token_coord(p), p.expr, p.statements0, p.statements1)

        # <expr> ::= <expr> PLUS <expr>
        #          | <expr> TIMES <expr>
        #          | <expr> EQ <expr>
        #          | <expr> NE <expr>
        @_('expr PLUS expr',
           'expr TIMES expr',
           'expr EQ expr',
           'expr NE expr')
        def expr(self, p):
            return ('expr: %s @ %d:%d' % ((p[1],) + self._token_coord(p)), p.expr0, p.expr1)

        # <expr> ::= <constant>
        #          | <identifier>
        @_('constant',
           'identifier')
        def expr(self, p):
            return p[0]

        # <expr> ::= LPAREN <expr> RPAREN
        @_('LPAREN expr RPAREN')
        def expr(self, p):
            return p.expr

        # <constant> ::= NUM
        @_('NUM')
        def constant(self, p):
            return ('num: %s @ %d:%d' % ((p.NUM,) + self._token_coord(p)))

        # <identifier> ::= ID
        @_('ID')
        def identifier(self, p):
            return ('id: %s @ %d:%d' % ((p.ID,) +  self._token_coord(p)))


main(["a = 3 * 4 + 5 ;"])


main(["a = 3 * ;"])


main(["a == 3 ; "])


main(["print a ;"])
```

```
code = '''
a = 3;
b = 4 * a;
print (a+b);
'''


main([code])
```

```
code = '''
a = 7;
b = 10;
c = 5;

if a*a == b*b + c*c {
   print 1;
} else {
   print 0;
}
'''


main([code])
```

Escrevendo um Analisador Sintático para a Linguagem uChuck

Nesta etapa, você deve escrever uma versão preliminar de um analisador sintático para a linguagem uChuck. A especificação da gramática do uChuck em BNF está aqui. Sua tarefa é escrever regras de análise sintática usando o SLY. Para um melhor entendimento, estude o capítulo "Escrevendo um Analisador Sintático" da documentação do SLY.

Especificação

Sua tarefa é traduzir as regras listadas em uma gramática BNF em uma coleção de métodos decorados pelo decorador `@_()`. O nome de cada método deve corresponder ao nome da regra gramatical que está sendo analisada. O argumento para o decorador `@_()` é uma cadeia de caracteres que descreve o lado direito da gramática. Assim, uma regra gramatical como:

```
<program> ::= <statement_list> EOF
```

torna-se um método de classe do Python da forma:

```
class UChuckParser(Parser):
    """A parser for the uChuck language."""
    ...
    # <program> ::= <statement_list> EOF
    @_('statement_list')
    def program(self, p):
        return ('program', p.statement_list)
```

Para construir uma árvore de sintaxe, basta criar e retornar uma tupla ou lista em cada função de regra gramatical, como mostrado acima.

Seu objetivo, ao final deste segundo projeto, é reconhecer **sintaticamente** programas expressos na linguagem uChuck. Para isso, o ideal é que você faça com que sua gramática não apresente **nenhum** conflito empilha/reduz.

Sugestão: Você deve começar de forma simples e trabalhar incrementalmente até construir a gramática completa.

Esboço do Analisador Sintático

```
import sys
!pip install sly
from sly import Lexer, Parser
```

Copie o código da classe `UChuckLexer` que você escreveu no primeiro projeto e cole na célula abaixo.

```
class UChuckLexer(Lexer):
    """A lexer for the uChuck language."""
```

```
        def __init__(self, error_func):
            """Create a new Lexer.
            An error function. Will be called with an error
            message, line and column as arguments, in case of
            an error during lexing.
            """
            self.error_func = error_func

        # Reserved keywords
        keywords = {
            'while': "WHILE",
            'if': "IF",
            'else': "ELSE",
        }

        # All the tokens recognized by the lexer
        tokens = tuple(keywords.values()) + (
            # Identifiers
            "ID",
            # Constants
            "INT_VAL",
            "STRING_LIT",
        )

        # String containing ignored characters (between tokens)
        ignore = " \t"

        # Other ignored patterns
        ignore_newline = # <<< INCLUDE A REGEX HERE FOR NEWLINE >>>
        ignore_comment = # <<< INCLUDE A REGEX HERE FOR COMMENT >>>

        # Regular expression rules for tokens
        ID = # <<< INCLUDE A REGEX HERE FOR ID >>>
        INT_VAL = # <<< INCLUDE A REGEX HERE FOR INT_CONST >>>
        STRING_LIT = # <<< INCLUDE A REGEX HERE FOR CHAR_CONST >>>
        # <<< YOUR CODE HERE >>>

        # Special cases
        def ID(self, t):
          t.type = self.keywords.get(t.value, "ID")
          return t

        # Define a rule so we can track line numbers
        def ignore_newline(self, t):
            self.lineno += len(t.value)

        def ignore_comment(self, t):
            self.lineno += t.value.count("\n")

        def find_column(self, token):
            """Find the column of the token in its line."""
            last_cr = self.text.rfind('\n', 0, token.index)
            return token.index - last_cr

        # Internal auxiliary methods
        def _error(self, msg, token):
            location = self._make_location(token)
            self.error_func(msg, location[0], location[1])
            self.index += 1

        def _make_location(self, token):
            return token.lineno, self.find_column(token)

        # Error handling rule
        def error(self, t):
            msg = "Illegal character %s" % repr(t.value[0])
            self._error(msg, t)

        # Scanner (used only for test)
        def scan(self, text):
            output = ""
            for tok in self.tokenize(text):
                print(tok)
                output += str(tok) + "\n"
            return output


    class UChuckParser(Parser):
        """A parser for the uChuck language."""

        # Get the token list from the lexer (required)
        tokens = UChuckLexer.tokens
```

```
precedence = (
    # <<< YOUR CODE HERE >>>
)

def __init__(self, error_func=lambda msg, x, y: print("Lexical error: %s at %d:%d" % (msg, x, y), file=sys.stdout)):
    """Create a new Parser.
    An error function for the lexer.
    """
    self.lexer = UChuckLexer(error_func)

def parse(self, text, lineno=1, index=0):
    return super().parse(self.lexer.tokenize(text, lineno, index))

# Internal auxiliary methods
def _token_coord(self, p):
    return self.lexer._make_location(p)

# Error handling rule
def error(self, p):
    if p:
        if hasattr(p, 'lineno'):
            print("Error at line %d near the symbol %s " % (p.lineno, p.value))
        else:
            print("Error near the symbol %s" % p.value)
    else:
        print("Error at the end of input")

# <program> ::= <statement_list> EOF
@_('statement_list')
def program(self, p):
    return ('program', p.statement_list)

# <statement_list> ::= { <statement> }+
# <<< YOUR CODE HERE >>>

# <statement> ::= <expression_statement>
#               | <loop_statement>
#               | <selection_statement>
#               | <jump_statement>
#               | <code_segment>
# <<< YOUR CODE HERE >>>

# <jump_statement> ::= "break" ";"
#                    | "continue" ";"
# <<< YOUR CODE HERE >>>

# <selection_statement> ::= "if" "(" <expression> ")" <statement> { "else" <statement> }?
# <<< YOUR CODE HERE >>>

# <loop_statement> ::= "while" "(" <expression> ")" <statement>
# <<< YOUR CODE HERE >>>

# <code_segment> ::= "{" { <statement_list> }? "}"
# <<< YOUR CODE HERE >>>

# <expression_statement> ::= { <expression> }? ";"
# <<< YOUR CODE HERE >>>

# <expression> ::= <chuck_expression> { "," <chuck_expression> }*
# <<< YOUR CODE HERE >>>

# <chuck_expression> ::= { <chuck_expression> "=>" }? <decl_expression>
# <<< YOUR CODE HERE >>>

# <decl_expression> ::= <binary_expression>
#                     | <type_decl> <identifier>
# <<< YOUR CODE HERE >>>

# <type_decl> ::= "int"
#               | "float"
#               | <identifier>
# <<< YOUR CODE HERE >>>

# <binary_expression> ::= <unary_expression>
#                       | <binary_expression> "+"  <binary_expression>
#                       | <binary_expression> "-"  <binary_expression>
#                       | <binary_expression> "*"  <binary_expression>
#                       | <binary_expression> "/"  <binary_expression>
#                       | <binary_expression> "%"  <binary_expression>
#                       | <binary_expression> "<=" <binary_expression>
#                       | <binary_expression> "<"  <binary_expression>
#                       | <binary_expression> ">=" <binary_expression>
```

```
        #                       | <binary_expression> ">"  <binary_expression>
        #                       | <binary_expression> "==" <binary_expression>
        #                       | <binary_expression> "!=" <binary_expression>
        #                       | <binary_expression> "&&" <binary_expression>
        #                       | <binary_expression> "||" <binary_expression>
        # <<< YOUR CODE HERE >>>

        # <unary_expression> ::= <primary_expression>
        #                      | <unary_operator> <unary_expression>
        # <<< YOUR CODE HERE >>>

        # <unary_operator> ::= "+"
        #                    | "-"
        #                    | "!"
        # <<< YOUR CODE HERE >>>

        # <primary_expression> ::= <literal>
        #                        | <location>
        #                        | "<<<" <expression> ">>>"
        #                        | "(" <expression> ")"
        # <<< YOUR CODE HERE >>>

        # <literal> ::= <integer_value>
        #             | <float_value>
        #             | <string_literal>
        #             | "true"
        #             | "false"
        # <<< YOUR CODE HERE >>>

        # <location> ::= <identifier>
        # <<< YOUR CODE HERE >>>


    def build_tree(root):
        return '\n'.join(_build_tree(root))

    def _build_tree(node):
        if isinstance(node, list):
            if not node: return
            node = tuple(node)

        if not isinstance(node, tuple):
            yield " "+str(node)
            return

        values = [_build_tree(n) for n in node]
        if len(values) == 1:
            yield from build_lines('──', '  ', values[0])
            return

        start, *mid, end = values
        yield from build_lines('┬─', '│ ', start)
        for value in mid:
            yield from build_lines('├─', '│ ', value)
        yield from build_lines('└─', '  ', end)

    def build_lines(first, other, values):
        try:
            yield first + next(values)
            for value in values:
                yield other + value
        except StopIteration:
            return

    def print_error(msg, x, y):
        # use stdout to match with the output in the .out test files
        print("Lexical error: %s at %d:%d" % (msg, x, y), file=sys.stdout)

    def main(args):
        parser = UChuckParser(print_error)
        with open(args[0], 'r') if len(args) > 0 else sys.stdin as f:
            st = parser.parse(f.read())
            if st is not None:
                print(build_tree(st))
```

Teste

Para o desenvolvimento inicial, tente executar o analisador sintático em um arquivo de entrada de exemplo, como:

```
 /* print values of factorials */
 1 => int n;
 1 => int value;

 while( n < 10 )
 {
     value * n => value;
     <<< value >>>;
     n + 1 => n;
 }
```

```
%%file test.uck
/* print values of factorials */
1 => int n;
1 => int value;

while( n < 10 )
{
    value * n => value;
    <<< value >>>;
    n + 1 => n;
}
```

E o resultado será semelhante ao texto mostrado abaixo.

```
 ┬─ program
 └─┬─┬─ expr
   │ └─┬─ chuck_op @ 2:1
   │   ├─┬─ var_decl
   │   │ ├─ type: int @ 2:6
   │   │ └─ id: n @ 2:10
   │   └─ literal: int, 1 @ 2:1
   ├─┬─ expr
   │ └─┬─ chuck_op @ 3:1
   │   ├─┬─ var_decl
   │   │ ├─ type: int @ 3:6
   │   │ └─ id: value @ 3:10
   │   └─ literal: int, 1 @ 3:1
   └─┬─ while @ 5:1
     ├─┬─ binary_op: < @ 5:8
     │ ├─ location: n @ 5:8
     │ └─ literal: int, 10 @ 5:12
     └─┬─ stmt_list @ 6:1
       └─┬─┬─ expr
         │ └─┬─ chuck_op @ 7:2
         │   ├─ location: value @ 7:15
         │   └─┬─ binary_op: * @ 7:2
         │     ├─ location: value @ 7:2
         │     └─ location: n @ 7:10
         ├─┬─ expr
         │ └─┬─ print @ 8:2
         │   └─ location: value @ 8:6
         └─┬─ expr
           └─┬─ chuck_op @ 9:2
             ├─ location: n @ 9:11
             └─┬─ binary_op: + @ 9:2
               ├─ location: n @ 9:2
               └─ literal: int, 1 @ 9:6
```

```
main(["test.uck"])
```

Estude cuidadosamente a saída do analisador sintático e certifique-se de que ela faz sentido. Quando estiver razoavelmente satisfeito com a saída, tente executar alguns dos testes mais complicados projetados para testar vários cenários atípicos, fora do padrão esperado. Você pode usar como base os exemplos contidos aqui.

No AVA há um grande conjunto de testes para verificar seu código: confira-os para ver mais exemplos.

Envie seu trabalho

Depois de concluir esta tarefa, copie o código da classe `UChuckParser` e o submeta no AVA.

Anexo

A lista abaixo define os nós da árvore de sintaxe que devem ser retornados em cada regra da gramática:

```
program = tuple('program', statement_list)

statement_list = list(statement)

statement = expression_statement
          | loop_statement
          | selection_statement
          | jump_statement
          | code_segment

jump_statement = tuple('break @ lineno:column')
               | tuple('continue @ lineno:column')

selection_statement = tuple('if @ lineno:column', expression, statement0, statement1)

loop_statement = tuple('while @ lineno:column', expression, statement)

code_segment = tuple('stmt_list @ lineno:column', statement_list)

expression_statement = tuple('expr', expression)

expression = chuck_expression
           | tuple('expr_list', list(chuck_expression))

chuck_expression = tuple('chuck_op @ lineno:column', decl_expression, chuck_expression)
                 | decl_expression

decl_expression = binary_expression
                | tuple('var_decl', type_decl, tuple('id: ' + str(ID) + ' @ lineno:column'))

type_decl = tuple('type: int @ lineno:column')
          | tuple('type: float @ lineno:column')
          | tuple('type: ' + str(ID) + ' @ lineno:column')

binary_expression = unary_expression
                  | tuple('binary_op: + @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: - @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: * @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: / @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: % @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: <= @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: < @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: >= @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: > @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: == @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: != @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: && @ lineno:column', binary_expression0, binary_expression1)
                  | tuple('binary_op: || @ lineno:column', binary_expression0, binary_expression1)

unary_expression = primary_expression
                 | tuple('unary_op: ' + str(unary_operator), unary_expression)

unary_operator = tuple('+ @ lineno:column')
               | tuple('- @ lineno:column')
               | tuple('! @ lineno:column')

primary_expression = literal
                   | location
                   | tuple('print @ lineno:column', expression)
```

```
                    | expression

literal = tuple('literal: int, ' + str(INT_VAL) + ' @ lineno:column')
        | tuple('literal: float, ' + str(FLOAT_VAL) + ' @ lineno:column')
        | tuple('literal: string, ' + str(STRING_LIT) + ' @ lineno:column')
        | tuple('literal: int, 1 @ lineno:column')
        | tuple('literal: int, 0 @ lineno:column')

location = tuple('location: ' + str(ID) + ' @ lineno:column')
```

Um novo nó é criado sempre que o valor retornado for uma tupla do Python, por exemplo:

```
    # <program> ::= <statement_list> EOF
    @_('statement_list')
    def program(self, p):
        return ('program', p.statement_list)
```

Uma lista de nós é criada sempre que o valor retornado for uma lista do Python, por exemplo:

```
    # <statement_list> ::= { <statement> }+
    @_('statement { statement }')
    def statement_list(self, p):
        return [p.statement0] + p.statement1
```

Uma referência para um nó é criada sempre que o valor retornado em uma regra for o nome de outra regra.

Os valores indicados como `lineno` e `column` referem-se aos números de linha e coluna, respectivamente, do símbolo terminal mais à esquerda de cada produção, se houver.