Describing Syntax and Semantics

of

Programming Languages

Part I
Programming Language Description

Description must

• be **concise** and **understandable**
• be useful to both **programmers** and **language implementors**
• cover both
  • **syntax** (forms of expressions, statements, and program units) and
  • **semantics** (meanings of expressions, statements, and program units)

*Example:* Java while-statement

**Syntax:** while (boolean_expr) statement

**Semantics:** if boolean_expr is true then statement is executed and control returns to the expression to repeat the process; if boolean_expr is false then control is passed on to the statement following the while-statement.
Lexemes and Tokens

Lowest-level syntactic units are called **lexemes**. Lexemes include identifiers, literals, operators, special keywords etc.

A **token** is a category of the lexemes (i.e. similar lexemes belong to a token)

*Example*: Java statement: `index = 2 * count + 17;`

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>=</td>
<td>_EQUALS</td>
</tr>
<tr>
<td>2</td>
<td>NUMBER</td>
</tr>
<tr>
<td>*</td>
<td>MUL</td>
</tr>
<tr>
<td>count</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>+</td>
<td>PLUS</td>
</tr>
<tr>
<td>17</td>
<td>NUMBER</td>
</tr>
<tr>
<td>;</td>
<td>SEMI</td>
</tr>
</tbody>
</table>

**IDENTIFIER** tokens: `index, count`

**NUMBER** tokens: `2, 17`

remaining 4 lexemes (`=, *, +, ;`) are lone examples of their corresponding token!
Lexemes and Tokens: Another Example

Example: SQL statement

select sno, sname
from suppliers
where sname = 'Smith'

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>SELECT</td>
</tr>
<tr>
<td>sno</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>,</td>
<td>COMMA</td>
</tr>
<tr>
<td>sname</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>from</td>
<td>FROM</td>
</tr>
<tr>
<td>suppliers</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>where</td>
<td>WHERE</td>
</tr>
<tr>
<td>sname</td>
<td>IDENTIFIER</td>
</tr>
<tr>
<td>=</td>
<td>EQUALS</td>
</tr>
<tr>
<td>'Smith'</td>
<td>SLITERAL</td>
</tr>
</tbody>
</table>

IDENTIFIER tokens: sno, sname, suppliers

SLITERAL tokens: 'Smith'

remaining lexemes (select, from, where, , , =) are lone examples of their corresponding token!
Lexemes and Tokens: A third Example

*Example:* WAE expressions

```plaintext
{with {{x 5} {y 2}} {+ x y}};
```

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>LBRACE</td>
</tr>
<tr>
<td>with</td>
<td>WITH</td>
</tr>
<tr>
<td>{</td>
<td>LBRACE</td>
</tr>
<tr>
<td>x</td>
<td>ID</td>
</tr>
<tr>
<td>5</td>
<td>NUMBER</td>
</tr>
<tr>
<td>}</td>
<td>RBRACE</td>
</tr>
<tr>
<td>{</td>
<td>LBRACE</td>
</tr>
<tr>
<td>y</td>
<td>ID</td>
</tr>
<tr>
<td>2</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>}</td>
<td>RBRACE</td>
</tr>
<tr>
<td>}</td>
<td>RBRACE</td>
</tr>
<tr>
<td>{</td>
<td>LBRACE</td>
</tr>
<tr>
<td>+</td>
<td>PLUS</td>
</tr>
<tr>
<td>x</td>
<td>ID</td>
</tr>
<tr>
<td>y</td>
<td>ID</td>
</tr>
<tr>
<td>}</td>
<td>RBRACE</td>
</tr>
<tr>
<td>}</td>
<td>RBRACE</td>
</tr>
<tr>
<td>;</td>
<td>SEMI</td>
</tr>
</tbody>
</table>

**TOKENS:**

- LBRACE
- RBRACE
- PLUS
- MINUS
- TIMES
- DIV
- ID
- WITH
- IF
- NUMBER
- SEMI
Lexical Analyzer

A **lexical analyzer** is a **program** that reads an input program/expression/query and extracts each lexeme from it (classifying each as one of the tokens).

Two ways to write this lexical analyzer program:

1. Write it from scratch! i.e. choose your favorite programming language (python!) and write a program in python that reads input string (which contain the input program, expression, or query) and extracts the lexemes.

2. Use a code-generator (Lex, Yacc, PLY, ANTLR, Bison, ...) that reads a high-level specification (in the form of **regular expressions**) of all tokens and generates a lexical analyzer program for you!

3. We will see how to write the lexical analyzer from scratch later.

4. Now, we will learn how to do it using PLY: [http://www.dabeaz.com/ply/](http://www.dabeaz.com/ply/)
# Regular Expressions in Python

https://docs.python.org/3/library/re.html  
https://www.w3schools.com/python/python_regex.asp

Meta Characters used in Python regular expressions:

<table>
<thead>
<tr>
<th>Meta</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>A set of characters</td>
<td>[a-z], [0-9], [xyz012]</td>
</tr>
<tr>
<td>.</td>
<td>Any one character (except newline)</td>
<td>he..o,</td>
</tr>
<tr>
<td>^</td>
<td>starts with</td>
<td>^hello</td>
</tr>
<tr>
<td>$</td>
<td>ends with</td>
<td>world$</td>
</tr>
<tr>
<td>*</td>
<td>zero or more occurrences</td>
<td>[a-z]*</td>
</tr>
<tr>
<td>+</td>
<td>one or more occurrences</td>
<td>[a-zA-Z]+</td>
</tr>
<tr>
<td>?</td>
<td>one or zero occurrence</td>
<td>[+-]?</td>
</tr>
<tr>
<td>{}</td>
<td>specify number of occurrences</td>
<td>[0-9]{5}</td>
</tr>
<tr>
<td></td>
<td>either or</td>
<td>[a-z]+</td>
</tr>
<tr>
<td>()</td>
<td>capture and group</td>
<td>({0-9}{5}) use \1 \2 etc. to refer</td>
</tr>
<tr>
<td>\</td>
<td>begins special sequence; also used to escape meta characters</td>
<td>d, \w, etc. (see documentation)</td>
</tr>
</tbody>
</table>
**PLY (Python Lex/Yacc): WAE Lexer**

```python
import ply.lex as lex

reserved = { 'with': 'WITH', 'if': 'IF' }

tokens = ['NUMBER','ID','LBRACE','RBRACE','SEMI','PLUS','MINUS','TIMES','DIV'] + list(reserved.values())

t_LBRACE = r'{'
t_RBRACE = r'}'
t_SEMI = r';'
t_WITH = r'[wW][iI][tT][hH]'
t_IF = r'[iI][fF]'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*' t_DIV = r'/'

# Ignored characters
t_ignore = " \n\t"
t_ignore_COMMENT = r'\#.*'

def t_NUMBER(t):
    r'[-+]?[0-9]+(\.[0-9]+)?'
    t.value = float(t.value)
    t.type = 'NUMBER'
    return t

def t_ID(t):
    r'[a-zA-Z][_a-zA-Z0-9]*'
    t.type = reserved.get(t.value.lower(),'ID')
    return t

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
```

To install PLY:

```
pip install ply
```

or

```
pip3 install ply
```
WAE Lexer continued

• The lexer object has just two methods: `lexer.input(data)` and `lexer.token()`

• Usually, the Lexical Analyzer is used in tandem with a Parser (the parser calls `lexer.token()`).

• So, the code on this page is written just to debug the Lexical Analyzer.

• Once satisfied we can/should comment out this code.
WAE Lexer continued

{with {{x 5} {y 2}} {+ x y}};

The PLY Lexer program we wrote will generate the following sequence of pairs of token types and their values:

('LBRACE','{'), ('WITH','with'), ('LBRACE','{'), ('LBRACE','{'), ('ID','x'), ('NUMBER','5'), ('RBRACE','}'), ('LBRACE','{'), ('ID','y'), ('NUMBER','2'), ('RBRACE','{'), ('RBRACE','}'), ('LBRACE','{'), ('PLUS','+'), ('ID','x'), ('ID','y'), ('RBRACE','}'), ('RBRACE','}'), ('SEMI',';')

Let us see this program (WAELexer.py) in action!
Language Generators and Recognizers

Now that we know how to describe tokens of a program, let us learn how to describe a “valid” sequence of tokens that constitutes a program. A valid program is referred to as a sentence in formal language theory.

Two ways to describe the syntax:

(1) **Language Generator**: a mechanism that can be used to generate sentences of a language. This is usually referred to as a **Context-Free-Grammar** (CFG). Easier to understand.

(2) **Language Recognizer**: a mechanism that can be used to verify if a given string, p, of characters (grouped in a sequence of tokens) belongs to a language L. The syntax analyzer in a compiler is a language recognizer.

(3) There is a close connection between a language generator and a language recognizer.
Chomsky Hierarchy and Backus-Naur Form

- Chomsky, a noted Linguist, defined a hierarchy of language generator mechanisms or grammars for four different classes of languages. Two of them are used to describe the syntax of programming languages:
  
  - **Regular Grammars**: describe the tokens and are equivalent to regular expressions.
  
  - **Context-free Grammars**: describe the syntax of programming languages

- John Backus invented a similar mechanism, which was extended by Peter Naur later and this mechanism is referred to as the Backus-Naur Form (BNF)

- Both these mechanisms are similar and we may use CFG or BNF to refer to them interchangeably.
Fundamentals of Context Free Grammars

CFGs are a **meta-language** to describe another language. They are meta-languages for programming languages!

A context-free grammar $G$ has 4 components $(N,T,P,S)$:

1) $N$, a set of non-terminal symbols or just called **non-terminals**; these denote abstractions that stand for syntactic constructs in the programming language.

2) $T$, a set of terminal symbols or just called **terminals**; these denote the **tokens** of the programming language.

3) $P$, a set of **production rules** of the form

   \[ X \rightarrow \alpha \]

   where $X$ is a non-terminal and $\alpha$ (**definition** of $X$) is a string made up of terminals or non-terminals. The production rules define the “valid” sequence of tokens for the programming language.

4) $S$, a non-terminal, that is designated as the **start symbol**; this denotes the highest level abstraction standing for all possible programs in the programming language.
CFGs: Examples of Production rules

Note: We will use lower-case for non-terminals and upper-case for terminals.

(1) A Java assignment statement may be represented by the abstraction `assign`. The definition of `assign` may be given by the production rule

\[
\text{assign} \rightarrow \text{VAR} \text{EQUALS} \text{expression}
\]

(2) A Java if statement may be represented by the abstraction `ifstmt` and the following production rules:

\[
\text{ifstmt} \rightarrow \text{IF} \text{LPAREN} \text{logic_expr} \text{RPAREN} \text{stmt}
\]

\[
\text{ifstmt} \rightarrow \text{IF} \text{LPAREN} \text{logic_expr} \text{RPAREN} \text{stmt} \text{ELSE} \text{stmt}
\]

These two rules have the same LHS; They can be combined into one rule with “or” on the RHS:

\[
\text{ifstmt} \rightarrow \text{IF} \text{LPAREN} \text{logic_expr} \text{RPAREN} \text{stmt} \mid \text{IF} \text{LPAREN} \text{logic_expr} \text{RPAREN} \text{stmt} \text{ELSE} \text{stmt}
\]

In the above examples, we have to introduce production rules that define the various abstractions used such as `expression`, `logic_expr`, and `stmt`
CFGs: Examples of Production rules

(3) A list of identifiers in Java may be represented by the abstraction `ident_list`. The definition of `ident_list` can be given by the following **recursive** production rules:

\[
\text{ident_list} \rightarrow \text{IDENTIFIER} \\
\text{ident_list} \rightarrow \text{ident_list} \text{ COMMA IDENTIFIER}
\]

Notice that the second rule is recursive because the non-terminal `ident_list` on the LHS also appears in the RHS.

It is time to learn how these production rules are to be used! The production rules are a type of “replacement” or “rewrite” rules, where the LHS is replaced by the RHS. Consider the following replacements/rewrites starting with `ident_list`:

\[
\text{ident_list} \\
\Rightarrow \text{ident_list} \text{ COMMA IDENTIFIER} \\
\Rightarrow \text{ident_list} \text{ COMMA IDENTIFIER COMMA IDENTIFIER} \\
\Rightarrow \text{ident_list} \text{ COMMA IDENTIFIER COMMA IDENTIFIER COMMA IDENTIFIER} \\
\Rightarrow \text{IDENTIFIER COMMA IDENTIFIER COMMA IDENTIFIER COMMA IDENTIFIER}
\]

substituting these token types by their values, we may get: `x, y, z, u`
WAE PLY Grammar

Note: In PLY, we use : instead of →

**PRODUCTION RULES (P)**

<table>
<thead>
<tr>
<th>non-terminal</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>waeStart</td>
<td>: wae SEMI</td>
</tr>
<tr>
<td>wae</td>
<td>: NUMBER</td>
</tr>
<tr>
<td>wae</td>
<td>: ID</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE PLUS wae wae RBRACE</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE MINUS wae wae RBRACE</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE TIMES wae wae RBRACE</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE DIV wae wae RBRACE</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE IF wae wae wae RBRACE</td>
</tr>
<tr>
<td>wae</td>
<td>: LBRACE WITH LBRACE alist RBRACE wae RBRACE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>non-terminal</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>alist</td>
<td>: LBRACE ID wae RBRACE</td>
</tr>
<tr>
<td>alist</td>
<td>: LBRACE ID wae RBRACE alist</td>
</tr>
</tbody>
</table>

**TERMINALS (T)**

- LBRACE
- RBRACE
- PLUS
- MINUS
- TIMES
- DIV
- ID
- WITH
- IF
- NUMBER
- SEMI

**NON-TERMINALS (N)**

- waeStart
- wae
- alist

\[
\begin{align*}
\text{wae} : & \text{ LBRACE PLUS wae wae RBRACE} \\
\{ & + x y \} \\
\text{wae} : & \text{ LBRACE WITH LBRACE alist RBRACE wae RBRACE} \\
\{ & \text{ with } \{\{x 5\} \{y 2\}\} \{+ x y\}\} \\
\end{align*}
\]
Grammars and Derivations

The sentences of the language are generated through a sequence of applications of the production rules, starting with the start symbol. This sequence of rule applications is called a derivation. In a derivation, each successive string is derived from the previous string by replacing one of the nonterminals with one of that nonterminal's definitions.

Consider the string: \{+ x y\};

Here is a derivation for this string (starting from \texttt{waesStart} we are able to derive \{+ x y\};)

\[
\begin{align*}
\texttt{waesStart} & \Rightarrow \texttt{waes} ; \\
\Rightarrow & \{ + \texttt{waes} \ \texttt{waes} \} ; \\
\Rightarrow & \{ + \texttt{x} \ \texttt{waes} \} ; \\
\Rightarrow & \{ + \texttt{x} \ \texttt{y} \} ;
\end{align*}
\]

We have highlighted in red the non-terminal that is being replaced/rewritten. Since we have a successful derivation for the string, \{+ x y\}; we say that the string, \{+ x y\}; is a “valid” WAE expression.
Another Derivation Example

Consider the string: \{WITH \{{x 5} \{y 2}\} \{+ x y}\};
Here is a derivation for this string:

\[
\begin{align*}
\text{Production Rule Used} \\
waeStart : & \text{ wae SEMI} \\
wae : & \text{ LBRACE WITH LBRACE \alist \ wae \ RBRACE wae \ RBRACE} \\
\alist : & \text{ LBRACE ID wae RBRACE \alist} \\
wae : & \text{ NUMBER} \\
\alist : & \text{ LBRACE ID wae RBRACE} \\
wae : & \text{ NUMBER} \\
wae : & \text{ LBRACE PLUS wae wae RBRACE} \\
wae : & \text{ ID} \\
wae : & \text{ ID}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{Another Derivation Example}} \\
\text{Consider the string: } \{\text{WITH } \{{x 5}\ \{y 2\}\} \{+ x y\}\}; \\
\text{Here is a derivation for this string:}
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{Production Rule Used}} \\
waeStart : & \text{ wae SEMI} \\
wae : & \text{ LBRACE WITH LBRACE \alist \ wae \ RBRACE wae \ RBRACE} \\
\alist : & \text{ LBRACE ID wae RBRACE \alist} \\
wae : & \text{ NUMBER} \\
\alist : & \text{ LBRACE ID wae RBRACE} \\
wae : & \text{ NUMBER} \\
wae : & \text{ LBRACE PLUS wae wae RBRACE} \\
wae : & \text{ ID} \\
wae : & \text{ ID}
\end{align*}
\]
Derivations continued

- Each string in a derivation, including the start symbol, is referred to as a **sentential form**.
- A derivation continues until the sentential form does not contain any non-terminals.
- A **leftmost derivation** is one in which the replaced nonterminal is always the leftmost nonterminal.
- In addition to leftmost, a derivation may be **rightmost** or in an order that is **neither leftmost nor rightmost**.
- Derivation order has **no effect** on the language generated by a grammar.
- By choosing **alternative** rules with which to replace non-terminals in the derivation, different sentences in the language can be generated.
- By **exhaustively** choosing all combinations of choices, the entire language can be generated.
Another Grammar Example

PRODUCTION RULES:

<assign> : <id> = <expr>
<expr> : <id> + <expr>
<expr> : <id> * <expr>
<expr> : ( <expr> )
<expr> : <id>
{id> : A
{id> : B
{id> : C

A leftmost derivation for A = B * ( A + C )

<assign>
A = <expr>
A = <id> * <expr>
A = B * <expr>
A = B * ( <expr> )
A = B * ( <id> + <expr> )
A = B * ( A + <expr> )
A = B * ( A + <id> )
A = B * ( A + C )
Parse Tree

- A derivation can be represented graphically in the form of a **parse tree**.
- The **root node** is the **start symbol** of the grammar.
- Each step of the derivation **expands** a non-terminal node by creating one child node for each symbol in the RHS of the production rule used in the derivation.
- Every **internal node** is labeled with a **non-terminal** and every **leaf** is labeled with a **terminal**.
- A **pre-order traversal of just the leaves** is called the **yield** and should equal the terminal string whose derivation the parse tree represents.

\[
\begin{align*}
\text{waeStart} & \Rightarrow \text{wae} ; \\
& \Rightarrow \{ + \text{wae wae} \} ; \\
& \Rightarrow \{ + \text{x wae} \} ; \\
& \Rightarrow \{ + \text{x y} \} ;
\end{align*}
\]
\begin{align*}
\text{waestart} \\
\Rightarrow \text{ wae ;} \\
\Rightarrow \{ \text{ WITH } \{ \text{ alist } \} \text{ wae } \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x wae } \} \text{ alist } \} \text{ wae } \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \text{ alist } \} \text{ wae } \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \{ \text{ y wae } \} \} \text{ wae } \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \{ \text{ y 2 } \} \} \text{ wae } \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \{ \text{ y 2 } \} \} \{ + \text{ wae wae} \} \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \{ \text{ y 2 } \} \} \{ + \text{ x wae} \} \} ; \\
\Rightarrow \{ \text{ WITH } \{ \{ \text{ x 5 } \} \{ \text{ y 2 } \} \} \{ + \text{ x y} \} \} ;
\end{align*}
Parse Tree: A third example

\[
\begin{align*}
\text{<assign>} \\
\Rightarrow \text{<id>} &= \text{<expr>} \\
\Rightarrow A &= \text{<expr>} \\
\Rightarrow A &= \text{<id>} * \text{<expr>} \\
\Rightarrow A &= B * \text{<expr>} \\
\Rightarrow A &= B * ( \text{<expr>} ) \\
\Rightarrow A &= B * ( \text{<id>} + \text{<expr>} ) \\
\Rightarrow A &= B * ( A + \text{<expr>} ) \\
\Rightarrow A &= B * ( A + \text{id} ) \\
\Rightarrow A &= B * ( A + C )
\end{align*}
\]
PLY Parser

- In addition to the Lexer (ply.lex) module, PLY also provides a Parser module (ply.yacc)
- The Parser module requires a CFG specification of the language
- PLY automatically generates a Parser program from the CFG.
- The Parser program calls the PLY Lexer object (created by the Lexer module) to read tokens from the input string.
- The Parser program verifies that the input string can be derived from the grammar by trying to construct a parse tree.
- PLY also provides the ability to evaluate “attribute” values for non-terminals in the parse tree. This ability can be used by the programmer to construct a data structure that stores the essential parts of the input string. This data structure is sometimes called an abstract syntax tree
PLY Parser continued

- Each grammar rule is defined by a Python function where the docstring to that function contains the grammar rule.
- The Python function name must begin with a `p_` and it is typical to include the non-terminal on the LHS of the grammar rule as part of the function name.
- Here is one such function for the WAE Grammar:

```python
def p_wae_8(p):
    'wae : LBRACE WITH LBRACE alist RBRACE wae RBRACE'
    p[0] = ['with', p[4], p[6]]
```

- As can be observed, the function is named `p_wae_8`. The 8 is used to indicate that this is the 8th grammar rule with `wae` on the LHS.
- The second line is the docstring containing the grammar rule.
- The function has one parameter, `p`, which is a list of “values” of each of the symbols in the grammar rule. `p[0]` holds the value of the LHS non-terminal and `p[1]`, `p[2]`, etc. hold the values of the symbols of the RHS, as shown in the two comment lines.
PLY Parser continued

```python
def p_wae_8(p):
    'waes : LBRACE WITH LBRACE alist RBRACE wae RBRACE'
    # ^       ^     ^    ^     ^      ^     ^     ^
    p[0] = ['with', p[4], p[6]]
```

- For RHS tokens or terminals, the "value" of the corresponding p[i] is the same as the t.value attribute assigned in the lexer module.
- For RHS non-terminals, the value of the corresponding p[i] is determined by whatever is placed in p[0] in the function for the rule that is used in the derivation to replace this non-terminal. This value can be anything, decided by the programmer.

<table>
<thead>
<tr>
<th>p[i]</th>
<th>value of p[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[1]</td>
<td>&quot;{&quot;</td>
</tr>
<tr>
<td>p[2]</td>
<td>&quot;with&quot;</td>
</tr>
<tr>
<td>p[3]</td>
<td>&quot;{&quot;</td>
</tr>
<tr>
<td>p[4]</td>
<td>value assigned to p[0] in one of the alist-functions</td>
</tr>
<tr>
<td>p[5]</td>
<td>&quot;}&quot;</td>
</tr>
<tr>
<td>p[6]</td>
<td>value assigned to p[0] in one of the wae-functions</td>
</tr>
<tr>
<td>p[7]</td>
<td>&quot;}&quot;</td>
</tr>
</tbody>
</table>
import ply.yacc as yacc
from WAELexer import tokens

def p_waeStart(p):
    'waeStart : wae SEMI'
    p[0] = p[1]

def p_wae_1(p):
    'wae : NUMBER'
    p[0] = ['num',p[1]]

def p_wae_2(p):
    'wae : ID'
    p[0] = ['id',p[1]]

def p_wae_3(p):
    'wae : LBRACE PLUS wae wae RBRACE'
    p[0] = ['+',p[3],p[4]]

def p_wae_4(p):
    'wae : LBRACE MINUS wae wae RBRACE'
    p[0] = ['-',p[3],p[4]]

def p_wae_5(p):
    'wae : LBRACE TIMES wae wae RBRACE'
    p[0] = ['*',p[3],p[4]]

def p_wae_6(p):
    'wae : LBRACE DIV wae wae RBRACE'
    p[0] = ['/','p[3],p[4]]

def p_wae_7(p):
    'wae : LBRACE IF wae wae wae RBRACE'
    p[0] = ['if',p[3],p[4],p[5]]

def p_wae_8(p):
    'wae : LBRACE WITH LBRACE alist RBRACE wae RBRACE'
    p[0] = ['with',p[4],p[6]]
WAE Parser (continued)

WAEParser.py (continued)

```python
def p_alist_1(p):
    'alist : LBRACE ID wae RBRACE'
    p[0] = [[p[2],p[3]]]

def p_alist_2(p):
    'alist : LBRACE ID wae RBRACE alist'

def p_error(p):
    print("Syntax error in input!")

parser = yacc.yacc()
```

WAE.py (main program)

```python
from WAEParser import parser

def read_input():
    result = ''
    while True:
        data = input('WAE: ').strip()
        if ';' in data:
            i = data.index(';')
            result += data[0:i+1]
            break
        else:
            result += data + ' '  
    return result

def main():
    while True:
        data = read_input()
        if data == 'exit;':
            break
        try:
            tree = parser.parse(data)
        except Exception as inst:
            print(inst.args[0])
            continue
        print(tree)
```

WAEParser.py (continued)
**Grammar (subset)**

\[
\text{waeStart : wae SEMI} \\
\text{wae : ID} \\
\text{wae : LBRACE PLUS wae wae RBRACE}
\]

**Input String**

\[
\{ + x y \} ;
\]

**Parse Tree**

![Parse Tree Diagram]

**Derivation**

\[
\text{waeStart} \\
\Rightarrow \text{wae} ; \\
\Rightarrow \{ + \text{wae wae} \} ; \\
\Rightarrow \{ + x \text{wae} \} ; \\
\Rightarrow \{ + x y \} ;
\]

**PLY functions (subset)**

```python
def p_waeStart(p):
    'waeStart : wae SEMI'
    p[0] = p[1]

def p_wae_2(p):
    'wae : ID'
    p[0] = ['id',p[1]]

def p_wae_3(p):
    'wae : LBRACE PLUS wae wae RBRACE'
    p[0] = ['+',p[3],p[4]]
```
PLY: In a nutshell

Input \{+ 3 4\}

Output 7

Token Specification (Reg Exp) WAElexer.py

Language Specification (CFG) WAEParser.py

PLY

Parser (parser object)

Lexer (lexer object)

Main Program WAE.py