**Ian Graham Christensen**

**PROJECT 02 FOR COMPUTER SCIENCE 214 SECTION A AT CALVIN COLLEGE**

**February 13, 2018**

This week's project is to practice creating BNF definitions and work with derivations.

The are three key ideas to keep in mind:

* If a portion of a language construct is complicated, hide the complexity by creating a nonterminal for it, and then define the nonterminal (later).
* If a language construct is optional, create a nonterminal that has two productions, one of which defines the optional part, and the other of which is an Ø-production.
* If a language construct can be repeated, create a non-terminal that has two productions, one of which is a recursive production providing the repetition, and the other of which is an Ø-production by which the recursion can terminate.

1. Using these ideas, define the constructs below with BNF productions. Underline the terminals in your productions, to distinguish them from the non-terminals. You may assume that the following have already been defined:

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**a. A Java character literal**

<character literal> ::= **'** <single character> **'** | **'** <escape sequence> **'**

<single character> ::= <input character> except **'** and \

<escape sequence> ::= ‘ | \

<input character> ::= <letter>|<digit>|<input character>

**b. A Java character string literal**

<string literal> ::= **"** <string characters>?**"**

<string characters> ::= <string character> | <string characters> <string character>

<string character> ::= <input character> except **"** and **\** | <escape character>

<escape character> ::= ‘ | \

<input character> ::= <letter>|<digit>|<input character>

**c. A Java integer literal**  
  
<integer literal> ::= <decimal integer literal> | <hex integer literal> | <octal integer literal>

<decimal integer literal> ::= <decimal numeral> <integer type suffix>

<hex integer literal> ::= <hex numeral> <integer type suffix>

<octal integer literal> ::= <octal numeral> <integer type suffix>

<integer type suffix> ::= **l** | **L**

<decimal numeral> ::= 0 | <non zero digit> <digits>

<digits> ::= <digit> | <digits> <digit>

<digit> ::= 0 | <non zero digit>

<non zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hex numeral> ::= 0 x <hex digit> | 0 X <hex digit> | <hex numeral> <hex digit>

<hex digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

<octal numeral> ::= 0 <octal digit> | <octal numeral> <octal digit>

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

**d. A Java real (floating point) literal**

<floating-point literal> ::= <digits> **.** <digits> <exponent part> <float type suffix>

<exponent part> ::= <exponent indicator> <signed integer>

<exponent indicator> ::= e | E

<signed integer> ::= <sign>?<digits>

<sign> ::= + | -

<float type suffix> ::= f | F | d | D

<digits> ::= <digit> | <digits> <digit>

**e. A Java identifier:**

<identifier>::= <[identifier characters](https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-IdentifierChars)>

<identifier characters>::= <letter><digits> | <letter> <letters> | <letter>   
<digits> ::= <digit> | <digits> <digit>

<letters> ::= <letter> | <letters> <letter>

**f. A Java function declaration (prototype):**<method declaration> ::= <method header> <method body>

<method header> ::= <method modifiers> <result type> <method declarator> <throws>

<result type> ::= <type> | void

<method modifiers> ::= <method modifier> | <method modifiers> <method modifier>

<method modifier> ::= public | protected | private | static | abstract | final | synchronized | native

<method declarator> ::= <identifier> ( <formal parameter list> )

<method body> ::= <block> | ;

**g. A Java if statement (you may assume that the non-terminals <statement> and <expression> are defined elsewhere):**   
  
<if then statement>::= if ( <expression> ) <statement>

<if then else statement>::= if ( <expression> ) <statement no short if> else <statement>

<if then else statement no short if> ::= if ( <expression> ) <statement no short if> else <statement no short if>

**h. A Java while statement (you may assume that the non-terminals <statement> and <expression> are defined elsewhere):**  
<while statement> ::= while ( <expression> ) <statement>

1. **Prove that the following grammar is ambiguous:**

<S> ::= <A>  
<A> ::= <A> + <A> | <id>  
<id> ::= a | b | c

There are infinite parse trees that can be made using the grammar

For example a separate parse tree can be made for each letter:

|  |  |  |
| --- | --- | --- |
| <S> | <S> | <S> |
| <A> | <A> | <A> |
| <id> | <id> | <id> |
| a | b | c |

Infinite possibilities are available due to the <A> + <A> segment:

|  |  |  |
| --- | --- | --- |
| <S> | <S> | <S> |
| <A> | <A> | <A> |
| <A> + <A> | <A> + <A> | <A> + <A> |
| <id> <id> | <id> <A> + <A> | <id> <A> + <A> |
| a a | a <id> <id> | a <id> <A> + <A> |
|  | a a | a <id> <id> |
|  |  | a a |

The grammar is therefore ambiguous because it has infinite outcomes.

1. **Give a left-most derivation for A = A \* (B + C) using the following BNF grammar:**

<assign> ::= <id> = <expr>  
<id> ::= A | B | C  
<expr> ::= <expr> + <term> | <term>  
<term> ::= <term> \* <factor> | <factor>  
<factor> ::= ( <expr> ) | <id>

<assign>

<id> = <expr>

A <term>

<term> \* <factor>

<factor> (<expr>)

<id> <expr> + <term>

A <term> <factor>

<factor> <id>

<id> C

B