Yasemin Akın22101782 Section 003
Uygar Aras 22103277 Section 003
Feza Emir Çelik 22101910 Section 003

Name of Our Programming Language: Baljeet

Part A

Complete Backus-Naur Form (BNF) Grammar of Baljeet

Program

<program> ::= <statement list>

Literals (Constant integer values and Constant string values)

<sign> = <plus> | <minus>

<integer literal> ::= <digits> | <sign> <digits>

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

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

<string literal> ::= <double quotation mark><string characters><double quotation mark>

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

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

Variables, which may be assigned integer values

<variable initialization> ::= <variable> <equal sign> <initialization right side> | <variable declaration> <equal sign> <initialization left side>

<initialization right side> ::= <integer literal> | <variable> | <mathematical expression>

<variable declaration or argument> ::= *int* <identifier> | *const int* <identifier>

<variable> ::= <identifier>

Operators (the four arithmetic operators, modula and exponentiation)

<arithmetic operator> ::= <plus> | <minus> | <multiplication operator> | <division operator> | <modula operator> | <exponentiation operator>

<mathematical expression> ::= <term> | <term> <arithmetic operator> <mathematical expression>

Expressions may contain matching parentheses along with integer values, variables, operators, and function calls.

<logical expressions> ::= <logical expression> | <left parenthesis> <logical expression> <right parenthesis> <logical operator> <logical expressions> | <not> <logical expressions>

<logical expression> ::= <term> <comparison operator> <term>

<logical operator> ::= <and> | <or>

<term> ::= <variable> | <integer literal> | <function call> | <mathematical expression>

<comparison operator> ::= <less than> | <bigger than> | <less than or equal to> | <bigger than or equal to> | <equal to> | <not equal>

Conditional statements (e.g., if-then, if-then-else)

<conditional statement> ::= *if* <left parenthesis> <logical expressions> <right parenthesis> *then* <left curly brace> <statement list> <right curly brace> | *if* <left parenthesis> <logical expressions> <right parenthesis> *then* <left curly brace> <statement list> <right curly brace> *else* <left curly brace> <statement list> <right curly brace>

looping statement(s)

<looping statement> ::= <while loop> | <for loop> | <do-while loop>

<while loop> ::= *while* <left parenthesis> <logical expressions> <right parenthesis> <left curly brace> <statement list> <right curly brace>

<for loop> ::= *for* <left parenthesis> <variable initialization> <semicolon> <logical expressions> <semicolon> <for increment or decrement> <right parenthesis> <left curly brace> <statement list> <right curly brace>

<for increment or decrement> ::= <identifier> <plus> <plus> | <identifier> <minus> <minus> | <variable initialization>

<do-while loop> ::= *do* <left curly brace> <statement list> <right curly brace> *while* <left parenthesis> <logical expressions> <right parenthesis>

Input/output statements (string constants can be displayed on the console)

<input> ::= *input prompt* <string literal> *put* <left parenthesis> <variable> <right parenthesis>

<output> ::= *output* <left parenthesis> <output body> <right parenthesis>

<output body> ::= <integer output> | <string literal> | <integer output> <plus> <string literal> <output body> | <string literal> <plus> <integer output> <output body>

<integer output> ::= <variable> | <mathematical expression>

A data structure to store a collection of integer values

<data structure> ::= *int* <identifier> <left square bracket> <right square bracket> <equal sign> <left square bracket> <data> <right square bracket>

<data> ::= <integer literal> | <integer literal> <comma> <data>

<data access> ::= <identifier> <left square bracket> <integer literal> <right square bracket>

Function definitions (functions take any number of, and return a single, integer values)

<function call> ::= <identifier> <left parenthesis> <right parenthesis> | <identifier> <left parenthesis> <parameter list> <right parenthesis>

<parameter list> ::= <term> | <term> <comma> <parameter list>

<function definition> ::= *function* <return type> <identifier> <left parenthesis> <right parenthesis> <left curly brace> <statement list> <return> <right curly brace> | *function* <return type> <identifier> <left parenthesis> <argument list> <right parenthesis> <left curly brace> <statement list> <return> <right curly brace>

<return> ::= *returns* | *returns* <variable> | *returns* <integer literal> | *returns* <mathematical expression>

<argument list> ::= <variable declaration or argument> | <variable declaration or argument> <comma> <argument list>

<statement list> ::= <statement> <semicolon> | <statement> <semicolon> <statement list>

<statement> ::= <variable declaration or argument> | <variable initialization> | <conditional statement> | <looping statement> | <input> | <output> | <function call> | <comment> | <function definition> | <data structure> | <data access> | <mathematical expression> | <return>

<return type> ::= *void* | *intreturn*

Comments.

<comment> ::= <hashtag> <string characters>

<identifier>

<identifier> ::= <letters>

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

<letter> ::= a | b | … | z | A | B | … | Z

Symbols

<double quotation mark> ::= “

<equal sign> ::= =

<plus> ::= +

<minus> ::= -

<multiplication operator> ::= *

<division operator> ::= /

<modula operator> ::= %

<exponentiation operator> ::= ^

<left parenthesis> ::= (

<right parenthesis> ::=)

<comma> ::= ,

<left curly brace> ::= {

<right curly brace> ::= }

<semicolon> ::= ;

<left square bracket> ::= [

<right square bracket> ::=]

<hashtag> ::= #

<less than> ::= <

<bigger than> ::= >

<less than or equal to> ::= <=

<bigger than or equal to> ::= >=

<equal to> ::= ==

<not equal to> ::= !=

<not> ::= !

<and> ::= &&

<or> ::= ||

Keywords

<keyword> ::= *int* | *void* | *if* | *then* | *else* | *while* | *for* | *do* | *input* | *output* | *function* | *returns* | *put* | *prompt* | *const* | *intreturn*

Description of Each of Language Components of Baljeet

1. **<program> ::= <statement list>**
   This non-terminal line represents the structure of a program in Baljeet. It specifies that a program is created from a list of statements.

2. **<sign> = <plus> | <minus>**
   This indicates the sign of numbers in our language meaning numbers can be positive and negative.

3. **<integer literal> ::= <digits> | <sign> <digits>**
   This line shows how integers are formed in our language, can be with a preceding and or just as digits.

4. **<digits> ::= <digit> | <digit> <digits>**
   Explains how digits are formed in our language: a digit or a combination of digits.

5. **<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
   Defines what can be called as digits, the numbers.

6. **<string literal> ::= <double quotation mark><string characters><double quotation mark>**
   Explains how a string is defined in our language with preceding and exceeding quotation marks and combination of string characters.

7. **<string characters> ::= <string character> | <string character> <string characters>**
   Defines how string characters are formed, can be a single string character or a combination of string characters.

8. **<string character> ::= <letter> | <digit>**
   Defines what a string character can be, both a letter and a digit.

9. **<variable initialization> ::= <variable> <equal sign> <initialization right side> | <variable declaration> <equal sign> <initialization left side>**

This non-terminal is created to show how a variable can be initialized in our language. And what is composed on both sides.

10. **<initialization right side> ::= <integer literal> | <variable> | <mathematical expression>**

    This is the non-terminal of how the initialization right side is functioning and composed.

11. **<variable declaration or argument> ::= *int* <identifier> | *const int* <identifier>**

    This is the non-terminal way of defining how a variable can be defined in our language. It also shows that our language only consists of integers.

12. **<variable> ::= <identifier>**

    This is one of the most basic non-terminals in our language indicating that a variable is an identifier.

13. **<arithmetic operator> ::= <plus> | <minus> | <multiplication operator> | <division operator> | <modula operator> | <exponentiation operator>**

    This non-terminal is for listing the arithmetic operators that have been in use in our language.

14. **<mathematical expression> ::= <term> | <term> <arithmetic operator> <mathematical expression>**

    This non-terminal is used to describe how a mathematical expression is handled in our language, composed of term or term and an arithmetic operator and another mathematical expression. This is one of the recursive definitions in our language.

15. **<logical expressions> ::= <logical expression> | <left parenthesis> <logical expression> <right parenthesis> <logical operator> <logical expressions> | <not> <logical expressions>**

    This non-terminal is one of the other recursive definitions in our language; it explains how logical expressions are composed in our language. It can be composed of a single logical expression or a left parentheses a logical expression a right parentheses a logical operator and logical expressions.

16. **<logical expression> ::= <term> <comparison operator> <term>**

    This non-terminal definition explains how a logical expression is handled in our language. It is composed of a term, comparison operator, and another term at the end.

17. **<logical operator> ::= <and> | <or>**

    This non-terminal is for defining the logical operator types which are and & or.

18. **<term> ::= <variable> | <integer literal> | <function call> | <mathematical expression>**

    This non-terminal definition is for defining how a term is made up in our language. It can be a list of things such as variables, integer literal, function call and mathematical expression.

19. **<comparison operator> ::= <less than> | <bigger than> | <less than or equal to> | <bigger than or equal to> | <equal to> | <not equal>**

This non-terminal definition is defining how a comparison operator is made up in our language. It can be a list of things such as ‘less than’, ‘bigger than’, ‘less than or equal to’, ‘bigger than or equal to’, ‘equal to’, and ‘not equal’.

20. **<conditional statement> ::= *if* <left parenthesis> <logical expressions> <right parenthesis> *then* <left curly brace> <statement list> <right curly brace> | *if* <left parenthesis> <logical expressions> <right parenthesis> *then* <left curly brace> <statement list> <right curly brace> *else* <left curly brace> <statement list> <right curly brace>**

This definition for conditional statements is for defining the if-else structure which most of the languages have and their most basic and fundamental type of structures.

21. **<looping statement> ::= <while loop> | <for loop> | <do-while loop>**

This definition for looping statements is designed for how a loop is structured and what are the looping statements that can be used in our language.

22. **<while loop> ::= *while* <left parenthesis> <logical expressions> <right parenthesis> <left curly brace> <statement list> <right curly brace>**

This definition for ‘while loop’ is defining while loops our language will use which are composed of parentheses, logical expressions, and a statement list.

23. **<for loop> ::= *for* <left parenthesis> <variable initialization> <semicolon> <logical expressions> <semicolon> <for increment or decrement> <right parenthesis> <left curly brace> <statement list> <right curly brace> ,**

This is the for loop definition in our language, it explains the struct of the for loops which are composed of parentheses, variables, semicolons, ‘increments or decrements’, and curly bracelets.

24. **<for increment or decrement> ::= <identifier> <plus> <plus> | <identifier> <minus> <minus> | <variable initialization>**

This is the definition of incrementer and decrementer which is in use in our for loop definition in our languages.

25. **<do-while loop> ::= *do* <left curly brace> <statement list> <right curly brace> *while* <left parenthesis> <logical expressions> <right parenthesis>**

The non-terminal definition of do-while loop definition in our language is the follow up of the usual do-while loop definition which is mostly universal in all of the programming languages.

26. **<input> ::= input prompt <string literal> put <left parenthesis> <variable> <right parenthesis>**

The non-terminal definition of input in our language is composed of the input prompt part and the input is always defined as a string in Baljeet.

27. **<output> ::= output <left parenthesis> <output body> <right parenthesis>**

The non-terminal output definition in our language defines how an output is created by using a left parenthesis output body and a right parenthesis.

28. **<output body> ::= <integer output> | <string literal> | <integer output> <plus>**

This non-terminal explanation is for defining the inside structure of the output body in Baljeet; it is basically a list of options of integer output, string literal and integer output with a plus.

29. **<string literal> <output body> | <string literal> <plus> <integer output> <output body>**

This non-terminal is for defining what a string literal is in Baljeet.

30. **<integer output> ::= <variable> | <mathematical expression>**

This non-terminal is for defining what an integer output could contain in Baljeet.

31. **<data structure> ::= *int* <identifier> <left square bracket> <right square bracket> <equal sign> <left square bracket> <data> <right square bracket>**

This is one of the most essential non-terminal definitions in our language. It is for defining how a data structure is made, what could it contain and its structure.

32. **<data> ::= <integer literal> | <integer literal> <comma> <data>**

Data in Baljeet is defined by either integer literals or integer literal followed by another data (since this asks for another data while producing one, it will be recursive) separated with a comma.

33. **<data access> ::= <identifier> <left square bracket> <integer literal> <right square bracket>**

Data in our data structure can be accessed by using this rule, similar to Java. One should know the name and the index, from the data structure, of the data wanted. Then you can access it by saying "*name*[*index*]" according to this rule.

34. **<function call> ::= <identifier> <left parenthesis> <right parenthesis> | <identifier> <left parenthesis> <parameter list> <right parenthesis>**

Function calls in Baljeet are either identifiers with empty parentheses, or identifiers followed by parentheses filled with parameter lists.

35. **<parameter list> ::= <term> | <term> <comma> <parameter list>**

Parameter lists in Baljeet are defined by either terms, or terms with parameter lists separated with a comma.

36. **<function definition> ::= *function* <return type> <identifier> <left parenthesis> <right parenthesis> <left curly brace> <statement list><right curly brace> | *function* <return type> <identifier> <left parenthesis> <argument list> <right parenthesis> <left curly brace> <statement list><right curly brace>**

Functions are defined with the keyword *function.* Return type of the functions follows, then the name of the function. After in parentheses an optional argument list can come. The

function body must be surrounded with curly braces. Inside the braces, a statement list should be found.

37. **<return> ::= *returns* | *returns* <variable> | *returns* <integer literal> | *returns* <mathematical expression>**
    Returns in Baljeet are for returning variables, integer literals, or mathematical expressions.

38. **<argument list> ::= <variable declaration or argument> | <variable declaration or argument> <comma> <argument list>**
    Here the syntax of arguments that a function can have in its definition are defined. Argument definitions are the same as variable declarations. If a function has several arguments, they should be separated with a comma.

39. **<statement list> ::= <statement> <semicolon> | <statement> <semicolon> <statement list>**
    Every program in Baljeet is composed of several statements. Here we decompose statements to single statements. After each statement a semicolon is needed.

40. **<statement> ::= <variable declaration or argument> | <variable initialization> | <conditional statement> | <looping statement> | <input> | <output> | <function call> | <comment> | <function definition> | <data structure> | <data access> | <mathematical expression> | <return>**
    Statements in Baljeet can be variable declaration or arguments, variable initializations, conditional statements, looping statements, inputs, outputs, function calls, comments, function definitions, data structures, data accesses, mathematical expressions, or returns.

41. **<return type> ::= *void* | *intreturn***
    Return type in Baljeet will be either void or integer.

42. **<comment> ::= <hashtag> <string characters>**
    Comment syntax in Baljeet defined as “# (text)”. Text characters are limited to <letter> and <digit> rules, which are together defined as <string characters>.
43. **<identifier> ::= <letters>**
    This non-terminal rule defines how an <identifier> is structured. In our language, identifiers' building blocks are limited to upper case or lowercase letters.

44. **<letters> ::= <letter> | <letter> <letters>**
    This non-terminal rule defines the plural form of <letter>, indicating one or more letters come together to form <letters>.

45. **<letter> ::= a | b | … | z | A | B | … | Z**
    This terminal rule defines the letters available in Baljeet. Letters in the English alphabet, both upper and lower case, are defined.