CS 315 – Programming Languages

PROJECT 1

ZEML LANGUAGE

Zeynep Büşra Ziyagil,21802646,SEC01

Muhammed Maruf Şatır, 21702908, SEC01

Erdem Ege Eroğlu, 21601636, SEC01

Contents

1.The Complete BNF Description of ZEML Language

Program Definition
<program> ::= start<begin><statements><end>

<statements> ::=<statement>|<statements><statement>

<statement> ::= <comment_line> |
<expr><endstmt>|<matched_stmt>|<unmatched_stmt>|<endstmt>|<loops>|<funct_dec>|

<endstmt> ::= <semicolon>

<sentence> ::= <identifier><sentence>|<identifier>

Variable Identifiers

<type> ::= <primitiveType>

<primitiveType> ::= “boolean” | “char” | “string” | “float‟ | “integer”

<boolean > ::= <true> | <false>
<string> ::= <string_identifier><sentence><string_identifier>
<sentence> ::= <word> | <sentence><word>
| <sentence><symbol>|<sentence><digit>
<word> ::= <letter> | <word><letter> |<word><digit>
<char> ::= <char_identifier><letter><char_identifier>
| <char_identifier><digit><char_identifier>
<integer> ::= <sign>?<digit>
<float> ::= <integer>.<digit>

<expr>::= <expr>+<term>|<term>|<logical_expr>|<expr>| <func_call_dec>

<term> ::= <var> | <constant>
<var> ::= <identifier>
<identifier> ::= <word> | <identifier> <digit>|<word><identifier>

<true> ::= “true” | 1
<false> ::= “false” | 0

<constant> ::= <constantIdentifier><var>
<constantIdentifier> : #
<var_list> ::= <var> | <var><backslash><var_list>

Assignment operator

<assignmentOperator> ::= <equal>

Expressions (arithmetic, relational, boolean, their combination)

<and> ::= and

<or> ::= or

<equal> ::= ==

<not_equal> ::= !=

<not> ::= not

<logical_expr> ::=<integer> <<integer>|<integer> > <integer>|<integer>>=<integer>

|<integer> >= <integer> | <identifier> < <identifier> | <identifier> > <identifier>

| <identifier> <=<identifier> >= <identifier> | <identifier> <and> <identifier>

| <identifier><or><identifier> | <boolean> <and> <boolean>

| <boolean><or><boolean>|<boolean><equal><boolean>

|<boolean><not_equal><boolean>|<identifier><equal><identifier>

|<identifier><not_equal><identifier>

Precedence, associativity of the operators

<expr> ::= <expr>+<term>| <expr> - <term>| <term>

<term> ::= <term> * <factor>| <term> / <factor>| <factor>

<factor> ::= <LP><expr><RP> | <identifier>

Loops

<loops> ::= <while_stmt>|<doWhile_stmt>

<while_stmt> ::=

loop<LP>(<logical_expr>|<funct_call>)<RP><LB><block><RB>

<doWhile_stmt> ::= doloop<LB> <stmts> <RB> while <LP>
<logical_expr>|<funct_call>><RP>

conditional statements

<matched_stmt> ::= if<LP><logical_expr><RP><matched_stmt>else<matched_stmt> | <while_stmt> | <doWhile_stmt>

<unmatched_stmt> ::= if<LP><logical_expr><RP><stmt> | if<LP><logical_expr><RP> <matched_stmt> else <unmatched_stmt>

statements for input / output (to ask questions and read answers)

<input_stmt>::= zemin<LP><inputs><RP>

<inputs>::=<boolean>|<char>|<float>|<integer>|<string>

<output_stmt> ::=zemout<LP> <outputs><RP>

<outputs> ::= <assignmentValues> | <assignmentValues><plus><outputs>

Function definitions and function calls

<argumstype> ::= <identifier>|<boolean>|<integer>|<float>|<char>|<string>

<block> ::= pass <end_stmt>|<statements> return <argumstype>|<statements>

<complex_args> ::= <argumstype>, <complex_args>|<argumstype>

<args> ::= <identifier>|<complex_args>|

<funct_dec> ::= function <identifier> <LP><args><RP><LB><block><RB>

Primitive Functions

<primitivefunc>::= <gamemap> | <createavatar> | <addroom>

| <movedirectly> |<opendoor> |<pickupwlt>

| <talkto> | <jump>| <fight>|<createmonster>|<addList>

<createavatar>::=createavatar(<args>);

<addList>::=addList(<args>);

<createmonster>::=createmonster(<args>);

<gamemap>::=gameMap(<args>);

<addroom>::=addroom(<args>);

<movedirectly>::=movedirectly(<args>);

<opendoor>::=opendoor(<args>);

<pickupwlt>::=pickupwlt(<args>);

<talkto>::=talkto(<args>);

<jump>::=jump(<args>);

<fight>::=fight(<args>);

<eat>::=eat(<args>);

<buy>::=buy(<args>);

Comments

<comment> ::= <tilda><sentence><endline>

Declerations

<termDecleration> ::= <type><space><term><endstmt>

<initialization> ::= <term><assignmentOperator><assignmentValues><endstmt>

<termDeclarationwithInit> ::=

<type><space><term><assignmentOperator><assignmentValues><endstmt>

<assignmentValues> ::= <primitiveType>| <constant>

Symbols

<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

<symbol> :: = <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <comma> | <semicolon> |
<tilda> | <underscore> | <equal> | <dot> | <char_identifier> | <space> | <constantIdentifier>

<LP> ::= (

<RP> ::=)

<LB> ::= {

<RB> :: = }

<LSB> ::= [

<RSB> ::=]

<backslash>::=\

<comma> ::= ,

<semicolon> ::= ;

<tilda> ::= ~

<underscore> ::= _

<equal> ::= =

<string_identifier> ::= “

<char_identifier> ::= „

<space> ::= “ ”

<sign> ::= +|-

<hashtag> ::= #

<endline> ::= \n

<dot> ::= .

<constantIdentifier> : #

2.Description of Language Constructs

Program

<program> ::= start<begin><statements><end>

In ZEML, program construction provides to start and finish the program. Entire program takes place in between “begin” and “end” commands.

<statements> ::=<statement><statements>|<statement>

Statements command is indicating statement and it can work recursively.

<statement> ::= <comment_line> |
<expr><endstmt>|<matched_stmt>|<unmatched_stmt>|<endstmt>|<loops>|<funct_dec>|

Statement commands can have comment lines, expressions, match and unmatched statements, loops, functions and just end of the statement.

<endstmt> ::= <semicolon>

Semicolon implies the end of the expression. The next expression after a semicolon means, it's a new expression.

- **Variable Identifiers**

<type> ::= <primitiveType>

Type is a non-terminal and it holds all basic data types which is named as primitive type or sub-data type.

<primitiveType> ::= "boolean" | "char" | "string" | "float" | "integer"

"Primitive types" is terminal and it holds all primitive data types that are similar to java. Those primitive types in ZEML are boolean, char, string, float, integer. Boolean can have two values true or false. Char variables are for all characters between ' '. String variables are for string representation. Float variables are for representation of float and double values. Integer data type is able to hold integers only.

<boolean > ::= <true> | <false>

That is a non-terminal and stands for truth values which are true or false in ZEML language.

<string> ::= <string_identifier><sentence><string_identifier>

String variables are for string representation. This is also one of the non-terminal components of the syntax of ZEML. Each string should be in " ", because " " are string identifiers.

<sentence> ::= <word>|<sentence><word>|<sentence><symbol>|<sentence><digit>

Sentence is another non-terminal. It is not a primitive data type because it can hold a group of values. It can be used in commenting.

<word> ::= <letter> | <word><letter> |<word><digit>

That is a non-terminal that can stand for letter and digit. Also, it can produce a group of letters and digits by combining those two.

<char>::=
<char_identifier><letter><char_identifier>|<char_identifier><digit><char_identifier>

<char> is an non-terminal and created for characters in ZEML. The conditions are that being between char identifiers (' ') and being letter or digit.

<integer> ::= <sign>?<digit>|<sign>?<digit><integer>

Integer non-terminal is for all integers regardless of their sign. Also, it can hold a group of integers by combining them in ZEML.

<float> ::= <integer>.<digit>

Float variable is an non-terminal and holds float numbers eg.12.4

<expr>::= <expr>+<term>|<term>|<logical_expr>|<expr>| <func_call_dec>

Each expression means one component of a statement. It may include expression, function calls, and term.

<term> ::= <var> | <constant>

Term is a non-terminal and can represent two options. First one is var and the other one is constant values.

<var> ::= <identifier>

Var represents identifier.

<identifier> ::= <word> | <identifier> <digit>|<word><identifier>

Identifiers can represent three options, first one is a single word, second one is a group with identifiers and digits, and the final one is a group of words.

<true> ::= “true” | 1

In ZEML, true can be string and integer as true and 1.

<false> ::= “false” | 0

In ZEML, false can be string and integer as false and 0.

<constant> ::= <constantIdentifier><var>

In ZEML, when a constant is decelerating, it should have a hashtag sign az a constant identifier.

<var_list> ::= <var> | <var><backslash><var_list>

In ZEML, variables can be listed by separating by backslash.

- **Assignment Operator**

<assignmentOperator>::=<equal>

Equal sign is an assignment operator that is used for initializations, in ZEML.

- **Expressions (arithmetic, relational, boolean, their combination)**

<and> ::= and

And is a logic operator for and in ZEML, same as java’s &&.

<or> ::= or

Or is a logic operator for or in ZEML, same as java's ||.

<equal> ::= ==

Equal is a comparator operator for equal in ZEML, same as java's ==.

<not_equal> ::= !=

Not_equal is a comparator operator for not equal in ZEML, same as java's !=.

<not> ::= not

Equal is a logic operator for not in ZEML, same as java's !.

<logical_expr> ::= <integer> < <integer>|<integer> ><integer>|<integer>>=<integer>

|<integer> >= <integer> | <identifier> < <identifier> | <identifier> > <identifier>

| <identifier> <=<identifier> >= <identifier> | <identifier> <and> <identifier>

| <identifier><or><identifier> | <boolean> <and> <boolean>

| <boolean><or><boolean>|<boolean><equal><boolean>

|<boolean><not_equal><boolean>|<identifier><equal><identifier>

|<identifier><not_equal><identifier>

A logical expression might consist of relational expressions. Those are greater than, equal or greater than, less than, equal or less than, and equal. Also, it can boolean checking by operations.It is possible to combine different sorts of expressions into the same logical expression.

- **Precedence, associativity of the operators**

<expr> ::= <expr>+<term>| <expr> - <term>| <term>

This expression consists of terms and expression and it symbolizes the addition and subtraction operations.It provides precedence by term because on terms multiplication and division operations take place.

<term> ::= <term> * <factor>| <term> / <factor>| <factor>

This term multiplication and division operations take place.It consists of factors and on factors parenthesis takes place to enable precedence.

<factor> ::= <LP><expr><RP> | <identifier>

This expression is for precedence it enables parenthesis to control operation order.

- **Loops**

<loops> ::= <while_stmt>|<doWhile_stmt>

Loops are divided by two groups first one is while statement and second is do while statement, those two represent for while loop and do while loop successively.

<while_stmt> ::=

loop<LP>(<logical_expr>|<funct_call>)<RP><LB><block><RB>

This is a non-terminal and designed for syntax of while loop that is one of the looping statements. If, logical expression or the function call is satisfied, then the block starts running until the logical expression is no longer satisfied.

<doWhile_stmt> ::= doloop<LB> <stmts> <RB> while <LP> <logical_expr>|<funct_call>><RP>

This is a non-terminal and designed for syntax of do while loop that is one of the looping statements. First executes the block part, then checks the logical expression to continue as a loop. When the condition is no longer satisfied, the loop would be broken.

- **conditional statements**

<matched_stmt> ::= if<LP><logical_expr><RP><matched_stmt>else<matched_stmt> | <while_stmt> | <doWhile_stmt>

In ZEML, there are two conditional statements and the first one is a matched statement that is a non-terminal. It can take another match if-statement by means of recursive call. It works for both if and else parts. Also, Left and right parenthesis should be in the same number. What is more, looping originates from matched statements.

<unmatched_stmt> ::= if<LP><logical_expr><RP><stmt> | if<LP><logical_expr><RP> <matched_stmt> else <unmatched_stmt>

In ZEML, there are two conditional statements and the second one is an unmatched statement that is a non-terminal. This statement can hold if-statement with else part that contains an unmatched statement. This is also recursive regardless of the number of left and right parenthesis.

- **statements for input / output (to ask questions and read answers)**

<input_stmt>::= zemin<LP><inputs><RP>

The input will be entered by the user in between paranthesis.

<inputs>::=<boolean>|<char>|<float>|<integer>|<string>

This is used to specify input which can consist of many data types.

<output_stmt> ::=zemout<LP> <outputs><RP>

The output will be given which is written in between paranthesis.

<outputs> ::= <assignmentValues> | <assignmentValues><plus><outputs>

This is used to specify output which can consist of many data types and to combine them with plus sign.

- **Function definitions and function calls**

<argumstype>::=<identifier>|<boolean>|<integer>|<float>|<char>|<string>

This type is created to use in function call and declarations. Data types diversify and it simplifies it for function usage.

<block> ::= pass <end_stmt>|<statements> return <argumstype>|<statements>

This is used in functions. It enables to fill in between brackets. The return token is used when some functions needs to return a specified value.

<complex_args> ::= <argumstype>, <complex_args>|<argumstype>

<args> ::= <identifier>|<complex_args>|

These two argument definitions are specified to add multiple arguments to the function in between brackets.

<funct_dec> ::= function <identifier> <LP><args><RP><LB><block><RB>

This is the basic function declaration which consists of arguments and identifier whose keyword is “function”. Declaration ends with a block statement to enable return.

- **Primitive Functions**

<primitivefunc>::= <gamemap> | <createavatar> | <addroom> | <movedirectly> |<opendoor> |<pickupwlt> | <talkto> | <jump>| <fight>|<createmonster>|<addList>

Primitive function is used to define all the basic funtions in a spesific signature. The word used in the functions are reserved words. All the functions included are created for the game properties that will be written in this ZEML language.

<createavatar>::=createavatar(<args>);This is the non-terminal function **createavatar** to create avatar for the player.

<addList>::=addList(<args>);This is the non-terminal function **addList** to create add items to variable list.

<createmonster>::=createmonster(<args>);This is the non-terminal function **createmonster** to create the monster avatar will fight while the playing game.

<gamemap>::=gameMap(<args>);This is the non-terminal function **gamemap** to create the basic game map which avatar will move on top of.

<addroom>::=addroom(<args>);This is the non-terminal function **addroom** to add another room which player will get by doing the right moves.

<movedirectly>::=movedirectly(<args>);This is the non-terminal function **movedirectly** to make the avatar created move with a specified direction.

<opendoor>::=opendoor(<args>);This is the non-terminal function **opendoor** to get in to another room.

<pickupwlt>::=pickupwlt(<args>);This is the non-terminal function **pickupwlt** to pick up money and get more wealthy.

<talkto>::=talkto(<args>);This is the non-terminal function **talkto** to enable avatar to talk by using input tab.

<jump>::=jump(<args>);This is the non-terminal function **jump** to jump above the obstacles setted in game map inside the rooms.

<fight>::=fight(<args>);This is the non-terminal function **fight** to monsters in the game.

<eat>::=eat(<args>);This is the non-terminal function **eat** to enable avatar to gain more power by buying food and eating them.

<buy>::=buy(<args>);This is the non-terminal function **buy** to enable avatar to buy items such as food or tools.

- **Comments**

<comment> ::= <tilda><sentence><endline>
This specified non-terminal which uses tilda enables adding comments. The syntax comment logic is created there. Adding a tilda in front of a line directly starts the comment mode. When the program passes on another line, comment mode will end.

- **Declerations**

<termDecleration> ::= <type><space><term><endstmt>

This non- terminal is created to enable users to declare a term in the ZEML language. For this aim the user needs to write the type of the variable and end with semicolon.

<initialization> ::= <term><assignmentOperator><assignmentValues><endstmt>

This non-terminal is necessary for the initialization of the variable declared. After writing the term it is combined with the assignment operator. This provides the value to pass to the variable.It ends with a semicolon.

<termDeclarationwithInit> ::=

<type><space><term><assignmentOperator><assignmentValues><endstmt>

This enables language to declare and initialize simultaneously. It combines both of the assignment operation and declaring process in one line.It ends with a semicolon.

<assignmentValues> ::= <primitiveType>| <constant>

This specifies the values will be assigned by the assignment operator.As it can be understood values can be other primitive values or constants.

3.Nontrivial Tokens

pass: Token for empty blocks.

integer: Token reserved for integer data type.

string: Token reserved for string data type

boolean: Token reserved for boolean data type.

return: Token to return values within a function.

if: Token reserved for conditional if-else statements.

else: Token reserved for conditional if-else statements.

comment: Token reserved for comment keyword detection.

function: Token reserved for function declaration.

remove: Token reserved for deleting statements.

add: Token reserved for adding items.

loop: Token reserved for detection of a while loop.

doloop: Token reserved for detection of a do while loop.

zemout: Token reserved for printing contents to the console.

zemin: Token reserved for reading from an input stream.

endstmt: Token reserved for detecting the end of a statement.

and: Token reserved for and operator.

or : Token reserved for or operator.

not: Token reserved for not operator.

start: Token that start the program

identifier: Token reserved for variables,functions and lists. It can consist of letters or letters combined with digits.It makes language understandable for coders and provides easier syntax.

literals: These values are the basics of the program. They can be assigned to variables or get used when it's necessary.

reserved words: Primitive type functions such as**<gamemap>, <createavatar>, <addroom>, <movedirectly>,<opendoor>, <pickupwlt>, <talkto>, <jump>, <fight>, <addList>,<createmonster>** etc. ot keywords which indicate variables such as integer, boolean etc. are reserved words of ZEML language. This gives the ZEML laungage readable syntax. Thus it becomes more useful for programmers. System funtions are specified for game writing. Reserved words makes type checking easier and makes the programs written in ZEML trustworthy.

4.Evaluation Creatia

Readability

Names chosen for literals, their design patterns are similar to the well-known programming languages. Logic based operations such as mathematical operations have a similar pattern with what we use in real life. This makes programmes readable because it is similar to what we always exposed in real life readings. The syntax of the program is also based on simple expressions. The reserved words of the ZEML such as if, else, loop means literally what they say so, for example if is used for a conditional statement. That makes programs easy on the eyes. It has a limited amount of operations but they are simple enough to create many scenarios and that makes the language readable and flexible. ZEML language does not include small unnecessary details, it is as useful and simple as possible to be understood easily. Because it is a language designed for game creating its abstractions are made for this concept. For example, the fight function is a basic function of this language and while coding process takes place coder does not need to define this function from the basic, it is already ready to use.This makes ZEML free of unnecessary coding lines and abstraction makes code more readable. Concepts are simple and understandable. However this abstraction, which makes coding and reading easier, makes the programmes less reliable.That's because coders cannot directly see what primitive type data is used.

Writability

ZEML language is designed for arcade games. Therefore, the main strength of the language, it has some functions. That is why, programmers would be able to reach directly all the functions for some game basics. For instance, some of those functions are ate, jump, fight, etc. Fordrone is not an Object-Oriented-Language. Indeed, ZEML has no classes, objects, or polymorphism. Moreover, ZEML does not have the pointers.This makes programming easier. Users are able to access directly to some functions that are produced to make the game commands simple. Thanks to those functions our language becomes more suitable for non-expert level programmers. When the ZEML is examined on the basis of orthogonality, there is a limited set of data types and system functions and system operations. This enables the language being more flexible in terms of legality of operations.It has a limited amount of operations but they are simple enough to create many scenarios and that makes the language writable and flexible.

Reliability

For the ambiguity of the program, we decided to make the program clear in terms of reliability. Reliability of the language may be affected by the type checking but we did not use these types of mechanisms because of our mission of the language to become more useful in game making.Moreover, the exception handling might be a threat to the reliability of the language. We did not use the expectation handling mechanism in our program. Therefore, there could be ambiguous parts for the sake of simplicity.

Since we use abstract data types for increasing readability of our language, the reliability factor of the language will be affected by these abstract data types. However, the usage of the abstract data type is extremely common and because we are designing arcade games these types of abstract data types will increase the performance.