CS315 - Programming Languages
Project 1

İrem Vesile Aydın 21902914

Zeynep Derin Dedeler 21902896

Ece Kahraman 21801879

Contents

Name of the Language:

YASS (Yet Another Statement Set)

Complete BNF Description of YASS

1. Program

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

<statement_list> ::= <statement>  |  <statement> <statement_list>
```

2. Statements

```
<statement> ::= <conditional_statement>;
              | <assignment_statement>;
              | <comment_statement>
              | <del_statement>;
              | <loop_statement>
              | <declaration_statement>;
              | <return_statement>;
              | <func_call_statement>;

<conditional_statement> ::=
          if<LP>(<rel_expression>| <func_call>)<RP><block>else<block>
        | if<LP>(<rel_expression>| <func_call>)<RP><block>
        | if<LP>(<rel_expression>| <func_call>)<RP><block>
                                   (elif<LP><block><RP>)+(else<block>)?

<assignment_statement> ::=   <set_ident> =  <set_expression>
                         |   <var_ident> = (<element> | <string> | <int>)
                         |   <var_ident> = <int_expression_1>

<comment_statement> ::= // {<cap_letter> | <letter> | <digit> | <symbols> | <empty>
                          | <white_space> } \n

<del_statement> ::=  del <set_ident>
```

```
<loop_statement> ::= while <LP> <rel_expression> <RP> <block>
                   | while <LP> <bool> <RP> <block>
                   | while <LP> <func_call> <RP> <block>
                   | do <block> while<LP> <rel_expression> <RP>
                   | do <block> while <LP> <func_call> <RP>
                   | do <block> while <LP> <bool> <RP>


<declaration_statement> ::= <set_declaration>
                          | <func_declaration>
                          | <element_declaration>
                          | <string_declaration>
                          | <int_declaration>

<return_statement>::= return  (<type_ident> | <type_init>)

<func_call_statement> ::= <func_call>

<block> ::= <LB><statement_list>?<RB>

<empty> ::=
```

3. Declarations

```
<set_declaration>::= set  (<set_ident> =  <set_expression>
                          | <set_ident> = <set_init> | <set_ident>)

<func_declaration> ::=
                   function <func_ident><LP><typed_argument_list><RP><block>

<element_declaration> ::= element  <var_ident> (= <element>)?

<string_declaration> ::= string <var_ident> (= <string>)?

<int_declaration> ::= int <var_ident> ( = <int>
                                      | = <int_expression_1>
                                      | = <var_ident> )?
```

4. Expressions

```
<expression> ::= <set_expression> | <rel_expression> | <int_expression_1>

<int_expression_1> ::=  (<var_ident> | <int>)
                        | <int_expression_1> <low_prec_int_op> <int_expression_2>
                        | <int_expression_2>

<int_expression_2> ::= (<var_ident> | <int>)
                        | <int_expression_2> <high_prec_int_op> (<var_ident> | <int>)

<set_expression> ::= <set_ident> | <set_expression> <set_operator> <set_ident>

<rel_expression> ::= <set_ident> <rel_opreator> <set_ident>
                       | <var_ident> <int_rel_operator> (<var_ident> | <int>)
                       | <int> <int_rel_operator> <var_ident>

<argument_list>::=  | <empty>
                        | <type_init> | <type_ident>
                        | <type_ident>, <argument_list>
                        | <type_init> , <argument_list>


<typed_argument_list>::=  | <empty>
                                | <type_keyword> <type_ident>
                                | <type_keyword> <type_ident>, <typed_argument_list>

<element_list> ::= <element> | <element> , <element_list>
```

5. Variables and Initializers

```
<set_ident> ::= <cap_letter> {<letter>|<digit>}

<var_ident> ::= <letter>{<cap_letter> | <letter> | <digit>}

<set_init> ::= <LB>[ <element_list>]<RB>

<element> ::= '{<letter>|<digit>|<symbols>|<cap_letter>}'

<string>::= "{<cap_letter> | <letter> | <digit>|<symbols>}"

<int> ::= (-)?(<digit>)+
```

6. Functions

<primitive_func> ::= add | remove | print | printFile | isEmpty | getSize | contains | pop
 | readFile | read

<func_call> ::= <func_ident><LP>(<argument_list> | <expression>
 | <func_call>)<RP>
<func_ident>::= _<letter> {<letter>|<digit>} | _<primitive_func>

7. Operators

<operator> ::= <rel_operator> | <set_operator> | <int_operators>

<rel_operator> ::= <set_rel_operator> | <int_rel_operator>

<set_rel_operator> ::= << | >> | == | !=

<int_rel_operator> ::= < | > | <= | >= | == | !=

<set_operator> ::= or | and | sub

<int_operators> ::= <low_prec_int_op> | <high_prec_int_op>

<low_prec_int_op> ::= + | -

<high_prec_int_op> ::= * | /

8. Types

<type_ident> ::= <var_ident> | <set_ident>

<type_init> ::= <set_init> | <element> | <string> | <int>

9. Terminals

<letter> ::= a | b | c | d | e | f | g | h | i | j | l | m | n | o | p | q | r | s | t | u | y | v | w | x | y | z

<cap_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

<white_space> ::= | \t

<symbols>::= . | \ | / | ? | * | % | = | ! | - | _ | $ | + | ^ | # | (|) | { | } | [|] | é | & | < | > | : | ; | , | ' | @ | £ | æ | ß

<type_keyword> ::= string | set | element | int

<bool> ::= true | false

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

<LB> ::= {

<RB> ::= }

<LP> ::= (

<RP> ::=)

Explanations for YASS Nonterminals

1. Program

<program> ::= <statement_list>

Our language is essentially a list of statements.

<statement_list> ::= <statement> | <statement> <statement_list>

A statement list is composed of statements.

2. Statements

```
<statement> ::= <conditional_statement>;
              | <assignment_statement>;
              | <comment_statement>
              | <del_statement>;
              | <loop_statement>
              | <declaration_statement>;
              | <return_statement>;
              | <func_call_statement>;
```

A statement in our language can be expressed as one of the more specific types of statements. At this stage, each of the statements is ended with a semicolon except for **<comment_statement>** and **<loop_statement>**.

```
<conditional_statement> ::=
         if <LP>(<rel_expression> | <fun_call>)<RP><block> else <block>
       | if <LP>(<rel_expression> | <fun_call>)<RP><block>
       | if <LP>(<rel_expression> | <fun_call>)<RP> <block>
                                   (elif <LP><block><RP>)+ (else <block>)?
```

Conditional statements represent an if-else structure. The statement can consist of a single if block, a single if-else block, or one or more else if blocks after the if with the optional else block at the end. We shortened the term “else if” to “elif”. They can have releational expressions and booleans as arguments. Functions can be called inside parentheses if they return a boolean value.

<assignment_statement> ::= <set_ident> = <set_expression>
| <var_ident> = (<element> | <string> | <int>)
| <var_ident> = <int_expression_1>

Assignment statements are created to assign the result of a set expression to another set, assign a value or an integer expression to a variable.

<comment_statement>::= // {<cap_letter> | <letter> | <digit> | <symbols> | | \t}\n

This is to allow the users to comment. After “//” terminal, the statement accepts anystring until the newline terminal is entered.

<del_statement> ::= del <set_ident>

A delete statement allows the user to deallocate the memory space for a previously initialized set.

<loop_statement> ::= while <LP> <rel_expression> <RP> <block>
| while <LP> <bool> <RP> <block>
| while <LP> <func_call> <RP> <block>
| do <block> while<LP> <rel_expression> <RP>
| do <block> while <LP> <func_call> <RP>
| do <block> while <LP> <bool> <RP>

Loop statements consist of while loops and do-while loops. Each of the loops can take either a **<bool>** or a **<rel_expression>** as its parameter. Functions can be called inside parentheses if they return a boolean value. Loop blocks are enclosed by curly brackets.

<declaration_statement> ::= <set_declaration>
| <func_declaration>
| <element_declaration>
| <string_declaration>
| <int_declaration>

Decleration statements create variables and functions. These variables and functions can be used in other parts of the program. Variables can be declared without initiailzation and can be initialized later in the program.

<return_statement>::= return (<type_ident> | <type_init>)

Return statement is used to return a variable from the function.

<func_call_statement> ::= <func_call>

Function call statement is used to call user-defined or primitive functions. It has two levels to provide a function call inside another function's parameter list during a function call.

```
_print(_getSize(S12));
```

<block> ::= <LB><statement_list>?<RB>

Block is a list of statements enclosed by curly brackets. They are useful for conditional statements, loops and function declarations. The number of statements can be zero.

<empty> ::=

Empty

3. Declarations

<set_declaration>::= set (<assignment_statement>
| <set_ident> = <set_init> | <set_ident>)

Set declaration statement creates a set. It uses the *set* keyword. Sets can be assigned to a value or result of some set operations. Sets also can be declaret without initialization.

```
set S1 = { ‘1’, ‘a’ };
set S2 = S2 or S3;
set S4;
```

<func_declaration> ::=
function <func_ident><LP><typed_argument_list><RP><block>

Function declaration statement creates a function using the *function* keyword. Functions take a list of argument with their types. Functions have a list of statements in the curly brackets and they can return variables. All function identifiers start with underscore symbol. Function statements end with semicolon as other statements.

```
function _f1( set S1, element el, string str) { . . . };
```

<element_declaration> ::= element <var_ident> (= <element>)?

Element declaration statemen creates an element using the *element* keyword. <var_ident> is the identifier of the element. Elements can be initialized as they declared or can be initialized later. Element values must be in single quotation marks.

```
element e;
element e = ‘33’;
```

<string_declaration> ::= string <var_ident> (= <string>)?

String declaration statement creates an element using the *string* keyword. <var_ident> is the identifier of the string. Strings can be initialized as they declared or can be initialized later. String values must be in quotation marks.

<int_declaration> ::= int <var_ident> (= <int>
| = <int_expression_1>
| = <var_ident>)?

Integer declaration statement creates a variable with the keyword *int* and the identifier <var_ident>. An integer can be declared as a real integer, or an integer expression, or as another integer variable.

4. Expressions

<expression> ::= <set_expression> | <rel_expression> | <int_expression_1>

Expression has three types: set expression, relational expression, and integer expressions.

<set_expression> ::= <set_ident>|<set_expression><set_operator> <set_ident>

Set expression applies set operations to sets or just an identifier for the set. Set expessions can be assigned to sets or passed as an argument to functions.

<rel_expression> ::= <set_ident> <rel_opreator> <set_ident>

Relational expressions describes relations between two sets. Relational operators consist of:

"<<" : Proper subset
">>" : Proper superset
"==" : Set equality
All of them return booleans.

<int_expression_1> ::= (<var_ident> | <int>)
| <int_expression_1> <low_prec_int_op> <int_expression_2>
| <int_expression_2>

This expression allows a series of arithmetic operations to be applied on two integer variables, or an integer variable and an integer constant. It is a right recursive expression, and makes sure of **<low_prec_int_op>** operators are closer to the root of the parse tree and are later in the operator precedence.

<int_expression_2> ::= (<var_ident> | <int>)
| <int_expression_2> <high_prec_int_op> (<var_ident> | <int>)

This expression is the second part of the integer expressions. Makes sure of **<high_prec_int_op>** is farther from the parse tree root and has precedence over **<low_prec_int_op>** .

<argument_list>::= | <empty>
| <type_init> | <type_ident>
| <type_ident>, <argument_list>
| <type_init> , <argument_list>

This allows users to set the parameters of an function during the function call. This provides two types of method to initilize parameters. The user can choose between using an existing variable as an parameter or directly passing a value to the parameter in the same type.

<typed_argument_list> ::= | <empty>
| <type_keyword> <type_ident>
| <type_keyword> <type_ident>, <typed_argument_list>

While declarating a function it allows user to declarate some parameters. The user can choose between four primitive types; set, element, int and string.

<element_list> ::= <element> | <element> , <element_list>

This provide the user with the ability to list the elements while creating a set. Code readability is acquired by seperating elements with a comma.

5. Variables and Initializers

<set_ident> ::= <cap_letter> {<letter>|<digit>}

Set identifier begins with a capital letter. It is optionally followed by a combination of lowercase letters and digits.

```
set S12;
```

<var_ident> ::= <letter>{<cap_letter> | <letter> | <digit>}

Variable identifier begins with a lowercase letter and follwed by a combination of letters and digits. It idetifies strings and elements.

```
element eE1;
srting s99;
```

<set_init> ::= <LB>[<element_list>]<RB>

Set initializer initializes sets using an element list enclosed by curly brackets. If the element list does not contain any elements, the set is empty.

<element> ::= '{<letter>|<digit>|<symbols>|<cap_letter>}'

Element is a combination of letters, digits, symbols, and capital letters. Elements are used in sets but also can be declared separately.They are always enclosed by single quotation marks.

<string>::= "{<cap_letter> | <letter> | <digit> | <symbols>}"

String is a combination of letters, digits, symbols, and capital letters. Strings are used in argument lists but also can be declared separately. They are always enclosed by single quotation marks.

6. Functions

<func_ident>::= _<letter> {<letter>|<digit>} | _<primitive_func>

Function identifier consists of an underscore followed by lowercase letters and digits. It is used to call and define functions.

```
function _f1() { };
_f1();
```

<func_call>::=<func_ident><LP>(<argument_list>|<expression>|<func_call>)<RP>

This allows function calls inside the parameter list of another function call without putting a semicolon at the end, unlike a function call statement. Parameter list can also include and expression or an argument list.

<primitive_func> ::= add | remove | print | printFile | isEmpty | getSize |contains
| pop | readFile | read

We designed some primitive function for our language YASS.

add*:* This function adds an element to a set and returns true if operation is succssessful or returns false.

```
_add(Set1, ‘el’);
```

remove: This function removes an element from a set and returns true if operation is succssessful or returns false.

```
_remove(Primeset, ‘12’);
```

print: This function prints four primitive types to the enviroment console. It can prints strings, elements, integers and sets. It can takes either their identifiers or direct values.

```
_print(“Testing the string 12.3”);
_print( el12 );
_print( Set21 );
```

printFile: This function takes a file destination in the string form and prints content of the file to the enviroment console.

```
_printFile(“C:\Users\zeynep\Documents\zdd-Dersler\CS315”);
```

isEmpty: This function returns true if the set is empty. Otherwise it returns false.

```
_isEmpty(Set8);
```

getSize: This function gets a set as an parameter and return number of elements it contains in integer type.

```
_getSize(Set3);
```

contains: This function return true if the set contains the element. Otherwise it returns false.

```
_contains(Setsm1, ‘sm1’);
```

pop: This function pops the last inserted element from a set and returns true. If the set is empty it returns false.

```
_pop(Set21);
```

readFile: This function takes a file destionation and reads its content and returns it in a string form to the program

```
_readFile(“C:\Users\zeynep\Documents\zdd-Dersler\CS315”);
```

read: This function reads an input from the user. Element, integer and string can be passed as argument to the function. It reads from the keyboard and assigns the value to the passed argument.

```
_read(S1);
_read(elm);
```

7. Operators

<operator>::= <rel_opreator> | <set_operator> | <int_operators>

Operators have three major types. Relational operators return boolean. Set operators return modified sets. Int operators return modified integers.

<set_operator>::= or | and | sub

Set operators are described on sets and return set.
or : union operator
and : intersection operator
sub : subtracts one set from another

<set_rel_operator> ::= << | >> | == | !=

Set relational operators are described on sets and they return boolean.
<< : subset of
>> : superset of
== : equal to
!= : not equal to

<int_rel_operator> ::= < | > | <= | >= | == | !=

Int relational operators are described on integers and they return boolean.
< : less than
> : greater than
<= : less than or equal to
>= : greater than or equal to
== : equal to
!= : not equal to

<int_operators> ::= <low_prec_int_op> | <high_prec_int_op>

The language has two types of integer expressions with different precedence levels. <high_prec_int_op> has precedence over <low_prec_int_op>.

<low_prec_int_op> ::= + | -

Addition and subtraction operators on integers.

<high_prec_int_op> ::= * | /

Multiplication and division operators on integers.

Nontrivial Tokens

- *if:* Reserved word for if statements
- *else:* Reserved word for else statements
- *elif:* Reserved word for else if statements
- *do:* Reserved word for do statements
- *while:* Reserved word for while statements
- *return:* Reserved word for return statements
- *del:* Reserved word for delete statements of sets
- *function:* Reserved word for function declarations
- *set:* Reserved word for set data type declarations
- *element:* Reserved word for element data type declarations
- *string:* Reserved word for string data type declarations
- *int:* Reserved word for integer data type declarations
- *or:* Reserved word for the union of sets
- *and:* Reserved word for the intersection of sets
- *sub:* Reserved word for the difference of sets
- *add:* Reserved word for the primitive function to add an element to set
- *remove:* Reserved word for the primitive function to remove a specific element from the set
- *print:* Reserved word for the primitive function to print data types
- *printFile:* Reserved word for the primitive function to print a file
- *isEmpty:* Reserved word for the primitive function to determine if the set is empty
- *getSize:* Reserved word fot the primitive function to return the set size
- *contains:* Reserved word for the primitive function to decide an element exists
- *pop:* Reserved word for the primitive function to remove the last added element
- *readFile:* Reserved word for the primitive function to take a file as an input
- *read:* Reserved word for the primitive function to take an input from the user

Evaluation of the Language

First and foremost, it needs to be understood that the YASS language we have designed is not a general-purpose language. Consequently, we have refrained from implementing some features present in widely used languages, such as lists, and floats. Next to those, our language may seem bare-bones, however they should not be compared in the first place.

Readability

We paid attention to make the syntax very simple, but also similar to that of high-level languages most of us are used to. We kept less of primitive data types, because some of the known data types were irrelevant to our language. By providing some primitive functions as well as supporting user-defined functions, we attempted to created a flexible environment for the users.

Writability:

The things we applied to make the language readable also make it easy to learn, use and write. However it does not support abstraction as we believed would be irrelevant for sets, consequently, the user will not be writing as expressively as they would like to.

Reliability:

Aliasing is restricted since a set should not have more than one name. While comparing the equality of two sets, both sets are assigned their own memory spaces. Even they are essentially the same set with different orders to element list, the names still refer to different objects. We made sure to check the variable types while passing to functions and such. On the other hand, exception handling is low since there aren’t many exceptions to be made in our syntax.