Language Specification:

Language Definition:

Alphabet:

a. Upper (A-Z) and lower case letters (a-z) of the English alphabet

b. Underline character '\_';

c. Decimal digits (0-9);

Lexic:

Special symbols, representing:

- operators + - \* : <- < <= != == >= > && || newline

- separators ( ) [ ] { } ; space

- reserved words: char int boolean else if while read write

Identifiers

- a sequence of letters and digits, with the first character being an underscore; the rule is:

identifier ::= "\_" letter | "\_" letter{letter|digit}

letter ::= "A" | "B" |...| "Z" | "a" | "b" | ... | "z"

digit ::= "0" | "1" |...| "9"

Constants

1.integer - rule:

noconst ::= "+"no|-no|no

no ::= digit \ 0 | (digit \ 0){digit}

2.character

character ::= 'letter'|'digit'

3.boolean = "true" | "false"

Syntax:

The words - predefined tokens are specified between " and ":

Sintactical rules:

program ::= decllist ";" cmpdstmt

declaration ::= type identifier

type ::= "char" {"[" noconst "]"} | "int" {"[" noconst "]"} | booltype

booltype ::= "boolean" {"[" noconst "]"}

decllist ::= declaration | declaration ";" decllist

cmpdstmt ::= "{" stmtlist "}"

stmtlist ::= stmt | stmt ";" stmtlist

stmt ::= simplstmt | structstmt

simplstmt ::= assignstmt | iostmt

assignstmt ::= identifier "<-" expression

expression ::= (expression ("+" | "-") term) | term

term ::= (term ("\*" | ":") factor) | factor

factor ::= "(" expression ")" | identifier | noconst

iostmt ::= "read" "(" identifier ")" | "write" "(" expression ")"

structstmt ::= stmt | ifstmt | whilestmt

ifstmt ::= "if" "(" condition ")" stmtlist ["else" stmt]

whilestmt ::= "while" "(" condition ")" stmtlist

condition ::= (expression relation expression) | booltype

Lexical rules:

identifier ::= "\_" letter | "\_" letter{letter|digit}

letter ::= "A" | "B" |...| "Z" | "a" | "b" | ... | "z"

digit ::= "0" | "1" |...| "9"

relation ::= "<" | "<=" | "==" | "!=" | ">=" | ">"

The tokens are codified according to the following table:

- identifiers - code 0

- constants - code 1

- reserved words: each word has its own code

- operators: each operator has its own code

- separators: each separator has its own code

Codification:

Token type Code

identifier 0

constant 1

int 2

boolean 3

read 4

write 5

if 6

else 7

while 8

true 9

false 10

char 11

: 12

; 13

+ 14

\* 15

( 16

) 17

[ 18

] 19

{ 20

} 21

- 22

< 23

> 24

= 25

! 26

&& 27

|| 28

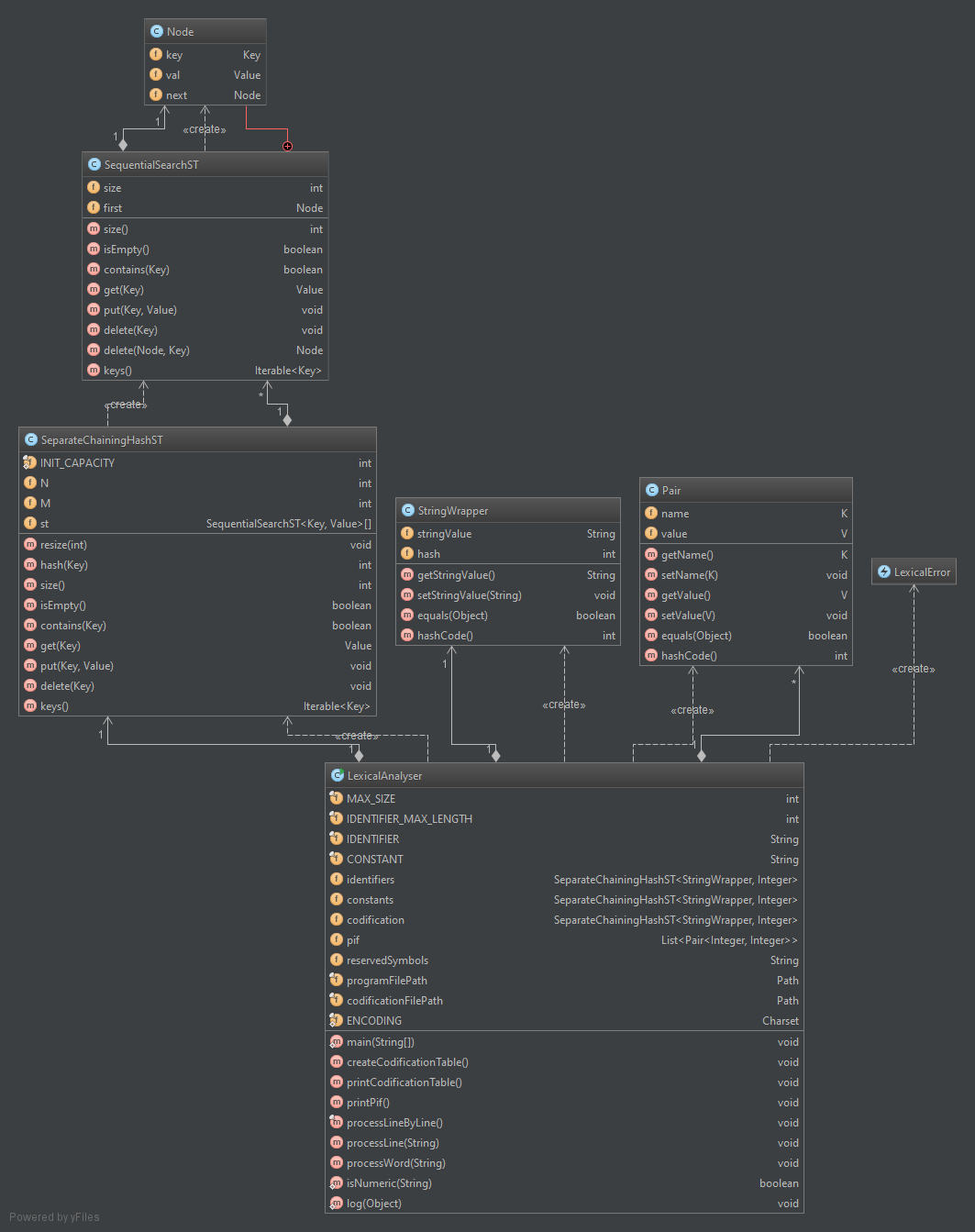
<- 29

== 30

!= 31

<= 32

>= 33



**SequentialSearchST**

The SequentialSearchST class represents an (unordered) symbol table of generic key-value pairs. It supports the usual put, get, contains, delete, size, and is-empty methods. It also provides a keys method for iterating over all of the keys.

public Value get(Key key)

Returns the value associated with the given key.

Input: key the key

Output: return the value associated with the given key if the key is in the symbol table and null if the key is not in the symbol table

public int size()

Output: Returns the number of key-value pairs in this symbol table.

public boolean isEmpty()

Output: return true if this symbol table is empty and false otherwise

public boolean contains(Key key)

Input: key the key

Output: return true if this symbol table contains key and false otherwise

public void put(Key key, Value val)

Inserts the key-value pair into the symbol table, overwriting the old value with the new value if the key is already in the symbol table. If the value is null, this effectively deletes the key from the symbol table.

Input: key the key

Output: val the value

public void delete(Key key)

Removes the key and associated value from the symbol table (if the key is in the symbol table).

Input: key the key

private Node delete(Node x, Key key)

Delete key in linked list beginning at Node x

public Iterable<Key> keys()

Returns all keys in the symbol table as an <tt>Iterable. To iterate over all of the keys in the symbol table named st, use the foreach notation: for (Key key : st.keys()).

Output: all keys in the sybol table as an Iterable

**SeparateChainingHashST**

The SeparateChainingHashST class represents a symbol table of generic key-value pairs. It supports the usual put, get, contains, delete, size, and is-empty methods. It also provides a keys method for iterating over all of the keys.

public int size()

Output: Returns the number of key-value pairs in this symbol table.

public boolean isEmpty()

Output: return true if this symbol table is empty and false otherwise

public boolean contains(Key key)

Input: key the key

Output: return true if this symbol table contains key and false otherwise

public Value get(Key key)

Returns the value associated with the given key.

Input: key the key

Output: return the value associated with the given key if the key is in the symbol table and null if the key is not in the symbol table

public void put(Key key, Value val)

Inserts the key-value pair into the symbol table, overwriting the old value with the new value if the key is already in the symbol table. If the value is null, this effectively deletes the key from the symbol table.

Input: key the key

Output: val the value

public void delete(Key key)

Removes the key and associated value from the symbol table (if the key is in the symbol table).

Input: key the key

public Iterable<Key> keys()

Output: Return keys in symbol table as an Iterable

**LexicalAnalyser**

private void createCodificationTable()

Reads the codification table from a text file and stores it in a map, with the key being the reserved word and the value the associated code

private final void processLineByLine()

Reads line by line the source program and process it

private void processLine(String aLine)

Splits the line in words, using a white space separator and process each word

private void processWord(String aWord)

Process a word and extract separators, operators, reserved words, constants and identifiers. If a word does not belong to any of these categories, throws an error