Syntax Analyzer

Compilers Project 2

Eric Kuxhausen

# Introduction

This project  
  
Methodology

This project

# Implementation

This project

# Discussion and Conclusions

This project

# References

* Compiler & Pascal References
  + Compilers Principles, Techniques, and Tools
    - Aho, Sethi, and Ullman.
    - ISBN 0201100886
  + Wikipedia page for Pascal
    - <http://en.wikipedia.org/wiki/Pascal_%28programming_language%29>
  + Christian Mann’s Pascal fuzz tester
    - http://personal.utulsa.edu/~christian-mann/cgi-bin/compilers/fuzz.cgi
* Java Programming References
  + Java 1.7 Docs
    - <http://docs.oracle.com/javase/7/docs>
  + StackOverflow.com

# Appendix I: Sample Inputs and Outputs

## reservedwords.txt

program PROGRAM 0

var VAR 0

array ARRAY 0

of OF 0

integer INT\_NAME 0

real REAL\_NAME 0

procedure PROC 0

begin BEGIN 0

end END 0

if IF 0

then THEN 0

else ELSE 0

while WHILE 0

do DO 0

call CALL 0

not NOT 0

or ADDOP 2

div MULOP 2

mod MULOP 3

and MULOP 4

## parseTest1.pas

program divests ( utopian ) ; var discarding : real ; var woofing :

array [ 95.322e9 .. 3586.94e10 ] of real ; var Stevens : integer ; var

indulging : real ; var salesman : integer ; var senders : real ; var

airline : integer ; begin end .

## parseTest1.listing

1 program divests ( utopian ) ; var discarding : real ; var woofing :

2 array [ 95.322e9 .. 3586.94e10 ] of real ; var Stevens : integer ; var

3 indulging : real ; var salesman : integer ; var senders : real ; var

4 airline : integer ; begin end .

## parseTest1.token

Line No. Lexeme TOKEN-TYPE ATTRIBUTE

1 program RESWRD PROGRAM

1 divests ID divests

1 ( OPENPAREN NULL

1 utopian ID utopian

1 ) CLOSEPAREN NULL

1 ; SEMICOLON NULL

1 var RESWRD VAR

1 discarding ID discarding

1 : COLON NULL

1 real RESWRD REAL\_NAME

1 ; SEMICOLON NULL

1 var RESWRD VAR

1 woofing ID woofing

1 : COLON NULL

2 array RESWRD ARRAY

2 [ OPENBRACKET NULL

2 95.322e9 NUM 95.322e9

2 .. DOTDOT NULL

2 3586.94e10 NUM 3586.94e10

2 ] CLOSEBRACKET NULL

2 of RESWRD OF

2 real RESWRD REAL\_NAME

2 ; SEMICOLON NULL

2 var RESWRD VAR

2 Stevens ID Stevens

2 : COLON NULL

2 integer RESWRD INT\_NAME

2 ; SEMICOLON NULL

2 var RESWRD VAR

3 indulging ID indulging

3 : COLON NULL

3 real RESWRD REAL\_NAME

3 ; SEMICOLON NULL

3 var RESWRD VAR

3 salesman ID salesman

3 : COLON NULL

3 integer RESWRD INT\_NAME

3 ; SEMICOLON NULL

3 var RESWRD VAR

3 senders ID senders

3 : COLON NULL

3 real RESWRD REAL\_NAME

3 ; SEMICOLON NULL

3 var RESWRD VAR

4 airline ID airline

4 : COLON NULL

4 integer RESWRD INT\_NAME

4 ; SEMICOLON NULL

4 begin RESWRD BEGIN

4 end RESWRD END

4 . EOF NULL

## allLexValid.pas

program stuff ( things );.,:

array [2..5] of integer real

call procedure begin end

:= if then else while do

= <> < <= >= >

+- or

\*/ div mod and

01 2 3.4 5.6E-7 8.9E1

## allLexValid.listing

1 program stuff ( things );.,:

SYNTAXERR: Expected { RESWRD VAR },{ RESWRD PROC },{ RESWRD BEGIN }encountered { EOF NULL }

2 array [2..5] of integer real

3 call procedure begin end

4 := if then else while do

5 = <> < <= >= >

6 +- or

7 \*/ div mod and

8 01 2 3.4 5.6E-7 8.9E1

## allLexValid.token

Line No. Lexeme TOKEN-TYPE ATTRIBUTE

1 program RESWRD PROGRAM

1 stuff ID stuff

1 ( OPENPAREN NULL

1 things ID things

1 ) CLOSEPAREN NULL

1 ; SEMICOLON NULL

1 . EOF NULL

1 . SYNTAXERR Expected { RESWRD VAR },{ RESWRD PROC },{ RESWRD BEGIN }encountered { EOF NULL }

1 , COMMA NULL

1 : COLON NULL

2 array RESWRD ARRAY

2 [ OPENBRACKET NULL

2 2 NUM 2

2 .. DOTDOT NULL

2 5 NUM 5

2 ] CLOSEBRACKET NULL

2 of RESWRD OF

2 integer RESWRD INT\_NAME

2 real RESWRD REAL\_NAME

3 call RESWRD CALL

3 procedure RESWRD PROC

3 begin RESWRD BEGIN

3 end RESWRD END

4 := ASSIGNOP NULL

4 if RESWRD IF

4 then RESWRD THEN

4 else RESWRD ELSE

4 while RESWRD WHILE

4 do RESWRD DO

5 = RELOP EQ

5 <> RELOP NEQ

5 < RELOP LT

5 <= RELOP LTE

5 >= RELOP GTE

5 > RELOP GT

6 + ADDOP PLUS

6 - ADDOP MINUS

6 or ADDOP OR

7 \* MULOP TIMES

7 / MULOP SLASH

7 div MULOP DIV

7 mod MULOP MOD

7 and MULOP AND

8 01 NUM 01

8 2 NUM 2

8 3.4 NUM 3.4

8 5.6E-7 NUM 5.6E-7

8 8.9E1 NUM 8.9E1

## semanticTest1.pas

program example(input, output);

var x: integer ;

var y: integer ;

var z: real ;

procedure gcd(a : integer ; b: real );

begin

if (b>0.0) or (x<>y) then x:=2 mod a

else z:= b/2.0

end ;

begin

x := 5;

call gcd(x, 3.0)

end .

## semanticTest1.listing

1 program example(input, output);

2 var x: integer ;

3 var y: integer ;

4 var z: real ;

5 procedure gcd(a : integer ; b: real );

6 begin

7 if (b>0.0) or (x<>y) then x:=2 mod a

8 else z:= b/2.0

9 end ;

10 begin

11 x := 5;

12 call gcd(x, 3.0)

13 end .

## semanticTest1.token

Line No. Lexeme TOKEN-TYPE ATTRIBUTE

1 program RESWRD PROGRAM

1 example ID example

1 ( OPENPAREN NULL

1 input ID input

1 , COMMA NULL

1 output ID output

1 ) CLOSEPAREN NULL

1 ; SEMICOLON NULL

2 var RESWRD VAR

2 x ID x

2 : COLON NULL

2 integer RESWRD INT\_NAME

2 ; SEMICOLON NULL

3 var RESWRD VAR

3 y ID y

3 : COLON NULL

3 integer RESWRD INT\_NAME

3 ; SEMICOLON NULL

4 var RESWRD VAR

4 z ID z

4 : COLON NULL

4 real RESWRD REAL\_NAME

4 ; SEMICOLON NULL

5 procedure RESWRD PROC

5 gcd ID gcd

5 ( OPENPAREN NULL

5 a ID a

5 : COLON NULL

5 integer RESWRD INT\_NAME

5 ; SEMICOLON NULL

5 b ID b

5 : COLON NULL

5 real RESWRD REAL\_NAME

5 ) CLOSEPAREN NULL

5 ; SEMICOLON NULL

6 begin RESWRD BEGIN

7 if RESWRD IF

7 ( OPENPAREN NULL

7 b ID b

7 > RELOP GT

7 0.0 NUM 0.0

7 ) CLOSEPAREN NULL

7 or ADDOP OR

7 ( OPENPAREN NULL

7 x ID x

7 <> RELOP NEQ

7 y ID y

7 ) CLOSEPAREN NULL

7 then RESWRD THEN

7 x ID x

7 := ASSIGNOP NULL

7 2 NUM 2

7 mod MULOP MOD

7 a ID a

8 else RESWRD ELSE

8 z ID z

8 := ASSIGNOP NULL

8 b ID b

8 / MULOP SLASH

8 2.0 NUM 2.0

9 end RESWRD END

9 ; SEMICOLON NULL

10 begin RESWRD BEGIN

11 x ID x

11 := ASSIGNOP NULL

11 5 NUM 5

11 ; SEMICOLON NULL

12 call RESWRD CALL

12 gcd ID gcd

12 ( OPENPAREN NULL

12 x ID x

12 , COMMA NULL

12 3.0 NUM 3.0

12 ) CLOSEPAREN NULL

13 end RESWRD END

13 . EOF NULL

# Appendix II: Program Listings

package kuxhausen;

import java.util.Scanner;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Project2 {

public static void main(String[] args) {

for (String filename : args) {

Scanner file = Lexar.getFile("input/" + filename + ".pas");

if (file != null) {

Lexar l = new Lexar(file);

Parser p = new Parser(l);

Utils.writeListingFile("output/" + filename + ".listing", p.getTokenList(),

l.getSourceBuffer());

Utils.writeTokenFile("output/" + filename + ".token", p.getTokenList());

}

}

}

}

package kuxhausen;

import java.util.ArrayList;

import kuxhausen.Token.ResWordAttr;

import kuxhausen.Token.TokType;

import kuxhausen.Token.\*;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Parser {

private Lexar mL;

/\*\*

\* current Token

\*/

private Token mT;

private SourcePointer mLine;

/\*\*

\* sync set for the current nonTerminal

\*/

private Token[] mSet;

private ArrayList<Token> mTokens = new ArrayList<Token>();

Parser(Lexar lex) {

mL = lex;

consumeToken();

program();

}

private void consumeToken() {

Token next = mL.getNextToken();

if (next == null) {

next = new Token(TokType.$, null, null, mLine);

}

mT = next;

mTokens.add(next);

mLine = next.position;

}

public ArrayList<Token> getTokenList() {

return mTokens;

}

private class SyntaxErr extends Exception {

}

Token pair(TokType type, Enum attr) {

return new Token(type, (attr != null) ? attr.ordinal() : -1, null, null);

}

public void match(TokType type, Enum attr) throws SyntaxErr {

Token desired = pair(type, attr);

if (mT.fullTypeMatch(desired)) {

consumeToken();

} else {

Token[] toks = {pair(type, attr)};

wanted(toks);

throw new SyntaxErr();

}

}

private void wanted(Token[] wanted) {

String message = generateErrorMessage(wanted);

mTokens.add(new Token(TokType.SYNTAXERR, message, mT.lexeme, mT.position));

}

private String generateErrorMessage(Token[] tokens) {

String result = "Expected ";

for (int i = 0; i < tokens.length; i++) {

result += (i > 0) ? "," : "";

result += "{ " + tokens[i].type.toString() + " " + tokens[i].getAttribute() + " }";

}

result += "encountered { " + mT.type.toString() + " " + mT.getAttribute() + " }";

return result;

}

private void sync() {

while (mT.type != TokType.$ && !inSet(mSet)) {

consumeToken();

}

}

private boolean inSet(Token[] syncSet) {

for (Token s : syncSet) {

if (mT.fullTypeMatch(s))

return true;

}

return false;

}

void program() {

mSet = new Token[] {};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROGRAM:

match(TokType.RESWRD, ResWordAttr.PROGRAM);

match(TokType.ID, null);

match(TokType.OPENPAREN, null);

identifierList();

match(TokType.CLOSEPAREN, null);

match(TokType.SEMICOLON, null);

programTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.PROGRAM)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void programTail() {

mSet = new Token[] {};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case VAR:

declarations();

programTailTail();

return;

case PROC:

subprogramDeclarations();

compoundStatement();

match(TokType.EOF, null);

return;

case BEGIN:

compoundStatement();

match(TokType.EOF, null);

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.VAR), pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void programTailTail() {

mSet = new Token[] {};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

subprogramDeclarations();

compoundStatement();

match(TokType.EOF, null);

return;

case BEGIN:

compoundStatement();

match(TokType.EOF, null);

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.PROC), pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void identifierList() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case ID:

match(TokType.ID, null);

identifierListTail();

return;

}

Token[] toks = {pair(TokType.ID, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void identifierListTail() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case CLOSEPAREN:

return;

case COMMA:

match(TokType.COMMA, null);

match(TokType.ID, null);

identifierListTail();

return;

}

Token[] toks = {pair(TokType.CLOSEPAREN, null), pair(TokType.COMMA, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void declarations() {

mSet =

new Token[] {pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case VAR:

match(TokType.RESWRD, ResWordAttr.VAR);

match(TokType.ID, null);

match(TokType.COLON, null);

type();

match(TokType.SEMICOLON, null);

declarationsTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.VAR)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void declarationsTail() {

mSet =

new Token[] {pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case VAR:

match(TokType.RESWRD, ResWordAttr.VAR);

match(TokType.ID, null);

match(TokType.COLON, null);

type();

match(TokType.SEMICOLON, null);

declarationsTail();

return;

case PROC:

return;

case BEGIN:

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.VAR), pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void type() {

mSet = new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case ARRAY:

match(TokType.RESWRD, ResWordAttr.ARRAY);

match(TokType.OPENBRACKET, null);

match(TokType.NUM, null);

match(TokType.DOTDOT, null);

match(TokType.NUM, null);

match(TokType.CLOSEBRACKET, null);

match(TokType.RESWRD, ResWordAttr.OF);

standardType();

return;

case INT\_NAME:

standardType();

return;

case REAL\_NAME:

standardType();

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.ARRAY), pair(TokType.RESWRD, ResWordAttr.INT\_NAME),

pair(TokType.RESWRD, ResWordAttr.REAL\_NAME)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void standardType() {

mSet = new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case INT\_NAME:

match(TokType.RESWRD, ResWordAttr.INT\_NAME);

return;

case REAL\_NAME:

match(TokType.RESWRD, ResWordAttr.REAL\_NAME);

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.INT\_NAME), pair(TokType.RESWRD, ResWordAttr.REAL\_NAME)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void subprogramDeclarations() {

mSet = new Token[] {pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

subprogramDeclaration();

match(TokType.SEMICOLON, null);

subprogramDeclarationsTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.PROC)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void subprogramDeclarationsTail() {

mSet = new Token[] {pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

subprogramDeclaration();

match(TokType.SEMICOLON, null);

subprogramDeclarationsTail();

return;

case BEGIN:

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.PROC), pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void subprogramDeclaration() {

mSet = new Token[] {pair(TokType.SEMICOLON, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

subprogramHead();

subprogramDeclarationTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.PROC)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void subprogramDeclarationTail() {

mSet = new Token[] {pair(TokType.SEMICOLON, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case VAR:

declarations();

subprogramDeclarationTailTail();

return;

case PROC:

subprogramDeclarations();

compoundStatement();

return;

case BEGIN:

compoundStatement();

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.VAR), pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void subprogramDeclarationTailTail() {

mSet = new Token[] {pair(TokType.SEMICOLON, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

subprogramDeclarations();

compoundStatement();

return;

case BEGIN:

compoundStatement();

return;

}

break;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.PROC), pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void subprogramHead() {

mSet =

new Token[] {pair(TokType.RESWRD, ResWordAttr.VAR), pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case PROC:

match(TokType.RESWRD, ResWordAttr.PROC);

match(TokType.ID, null);

subprogramHeadTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.PROC)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void subprogramHeadTail() {

mSet =

new Token[] {pair(TokType.RESWRD, ResWordAttr.VAR), pair(TokType.RESWRD, ResWordAttr.PROC),

pair(TokType.RESWRD, ResWordAttr.BEGIN)};

try {

switch (mT.type) {

case OPENPAREN:

arguments();

match(TokType.SEMICOLON, null);

return;

case SEMICOLON:

match(TokType.SEMICOLON, null);

return;

}

Token[] toks = {pair(TokType.OPENPAREN, null), pair(TokType.SEMICOLON, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void arguments() {

mSet = new Token[] {pair(TokType.SEMICOLON, null)};

try {

switch (mT.type) {

case OPENPAREN:

match(TokType.OPENPAREN, null);

parameterList();

match(TokType.CLOSEPAREN, null);

return;

}

Token[] toks = {pair(TokType.OPENPAREN, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void parameterList() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case ID:

match(TokType.ID, null);

match(TokType.COLON, null);

type();

parameterListTail();

return;

}

Token[] toks = {pair(TokType.ID, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void parameterListTail() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case CLOSEPAREN:

return;

case SEMICOLON:

match(TokType.SEMICOLON, null);

match(TokType.ID, null);

match(TokType.COLON, null);

type();

parameterListTail();

return;

}

Token[] toks = {pair(TokType.CLOSEPAREN, null), pair(TokType.SEMICOLON, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void compoundStatement() {

mSet = new Token[] {pair(TokType.EOF, null), pair(TokType.SEMICOLON, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case BEGIN:

match(TokType.RESWRD, ResWordAttr.BEGIN);

compoundStatementTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.BEGIN)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void compoundStatementTail() {

mSet = new Token[] {pair(TokType.EOF, null), pair(TokType.SEMICOLON, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case BEGIN:

optionalStatements();

match(TokType.RESWRD, ResWordAttr.END);

return;

case END:

match(TokType.RESWRD, ResWordAttr.END);

return;

case IF:

optionalStatements();

match(TokType.RESWRD, ResWordAttr.END);

return;

case WHILE:

optionalStatements();

match(TokType.RESWRD, ResWordAttr.END);

return;

case CALL:

optionalStatements();

match(TokType.RESWRD, ResWordAttr.END);

return;

}

break;

case ID:

optionalStatements();

match(TokType.RESWRD, ResWordAttr.END);

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.BEGIN), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.IF), pair(TokType.RESWRD, ResWordAttr.WHILE),

pair(TokType.RESWRD, ResWordAttr.CALL), pair(TokType.ID, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void optionalStatements() {

mSet = new Token[] {pair(TokType.RESWRD, ResWordAttr.END)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case BEGIN:

statementList();

return;

case IF:

statementList();

return;

case WHILE:

statementList();

return;

case CALL:

statementList();

return;

}

break;

case ID:

statementList();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.BEGIN), pair(TokType.RESWRD, ResWordAttr.IF),

pair(TokType.RESWRD, ResWordAttr.WHILE), pair(TokType.RESWRD, ResWordAttr.CALL),

pair(TokType.ID, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void statementList() {

mSet = new Token[] {pair(TokType.RESWRD, ResWordAttr.END)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case BEGIN:

statement();

statementListTail();

return;

case IF:

statement();

statementListTail();

return;

case WHILE:

statement();

statementListTail();

return;

case CALL:

statement();

statementListTail();

return;

}

break;

case ID:

statement();

statementListTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.BEGIN), pair(TokType.RESWRD, ResWordAttr.IF),

pair(TokType.RESWRD, ResWordAttr.WHILE), pair(TokType.RESWRD, ResWordAttr.CALL),

pair(TokType.ID, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void statementListTail() {

mSet = new Token[] {pair(TokType.RESWRD, ResWordAttr.END)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

}

break;

case SEMICOLON:

match(TokType.SEMICOLON, null);

statement();

statementListTail();

return;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.SEMICOLON, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void statement() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case BEGIN:

compoundStatement();

return;

case IF:

match(TokType.RESWRD, ResWordAttr.IF);

expression();

match(TokType.RESWRD, ResWordAttr.THEN);

statement();

statementTail();

return;

case WHILE:

match(TokType.RESWRD, ResWordAttr.WHILE);

expression();

match(TokType.RESWRD, ResWordAttr.DO);

statement();

return;

case CALL:

procedureStatment();

return;

}

break;

case ID:

variable();

match(TokType.ASSIGNOP, null);

expression();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.BEGIN), pair(TokType.RESWRD, ResWordAttr.IF),

pair(TokType.RESWRD, ResWordAttr.WHILE), pair(TokType.RESWRD, ResWordAttr.CALL),

pair(TokType.ID, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void statementTail() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case ELSE:

match(TokType.RESWRD, ResWordAttr.ELSE);

statement();

return;

}

break;

case SEMICOLON:

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.ELSE),

pair(TokType.SEMICOLON, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void variable() {

mSet = new Token[] {pair(TokType.ASSIGNOP, null)};

try {

switch (mT.type) {

case ID:

match(TokType.ID, null);

variableTail();

return;

}

Token[] toks = {pair(TokType.ID, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void variableTail() {

mSet = new Token[] {pair(TokType.ASSIGNOP, null)};

try {

switch (mT.type) {

case OPENBRACKET:

match(TokType.OPENBRACKET, null);

expression();

match(TokType.CLOSEBRACKET, null);

return;

case ASSIGNOP:

return;

}

Token[] toks = {pair(TokType.OPENBRACKET, null), pair(TokType.ASSIGNOP, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void procedureStatment() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case CALL:

match(TokType.RESWRD, ResWordAttr.CALL);

match(TokType.ID, null);

procedureStatementTail();

return;

}

break;

}

Token[] toks = {pair(TokType.RESWRD, ResWordAttr.CALL)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void procedureStatementTail() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case ELSE:

return;

}

break;

case OPENPAREN:

match(TokType.OPENPAREN, null);

expressionList();

match(TokType.CLOSEPAREN, null);

return;

case SEMICOLON:

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.ELSE),

pair(TokType.OPENPAREN, null), pair(TokType.SEMICOLON, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void expressionList() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case NOT:

expression();

expressionListTail();

return;

}

break;

case OPENPAREN:

expression();

expressionListTail();

return;

case ADDOP:

switch (AddopAttr.values()[(int) mT.attribute]) {

case PLUS:

expression();

expressionListTail();

return;

case MINUS:

expression();

expressionListTail();

return;

}

break;

case ID:

expression();

expressionListTail();

return;

case NUM:

expression();

expressionListTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.NOT), pair(TokType.OPENPAREN, null),

pair(TokType.ADDOP, AddopAttr.PLUS), pair(TokType.ADDOP, AddopAttr.MINUS),

pair(TokType.ID, null), pair(TokType.NUM, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void expressionListTail() {

mSet = new Token[] {pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case CLOSEPAREN:

return;

case COMMA:

match(TokType.COMMA, null);

expression();

expressionListTail();

return;

}

Token[] toks = {pair(TokType.CLOSEPAREN, null), pair(TokType.COMMA, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void expression() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case NOT:

simpleExpression();

expressionTail();

return;

}

break;

case OPENPAREN:

simpleExpression();

expressionTail();

return;

case ADDOP:

switch (AddopAttr.values()[(int) mT.attribute]) {

case PLUS:

simpleExpression();

expressionTail();

return;

case MINUS:

simpleExpression();

expressionTail();

return;

}

break;

case ID:

simpleExpression();

expressionTail();

return;

case NUM:

simpleExpression();

expressionTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.NOT), pair(TokType.OPENPAREN, null),

pair(TokType.ADDOP, AddopAttr.PLUS), pair(TokType.ADDOP, AddopAttr.MINUS),

pair(TokType.ID, null), pair(TokType.NUM, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void expressionTail() {

mSet =

new Token[] {pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case THEN:

return;

case ELSE:

return;

case DO:

return;

}

break;

case CLOSEPAREN:

return;

case SEMICOLON:

return;

case COMMA:

return;

case CLOSEBRACKET:

return;

case RELOP:

match(TokType.RELOP, null);

simpleExpression();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.DO),

pair(TokType.CLOSEPAREN, null), pair(TokType.SEMICOLON, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEBRACKET, null),

pair(TokType.RELOP, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void simpleExpression() {

mSet =

new Token[] {pair(TokType.RELOP, null), pair(TokType.SEMICOLON, null),

pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.ELSE),

pair(TokType.RESWRD, ResWordAttr.THEN), pair(TokType.CLOSEBRACKET, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEPAREN, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case NOT:

term();

simpleExpressionTail();

return;

}

break;

case OPENPAREN:

term();

simpleExpressionTail();

return;

case ADDOP:

switch (AddopAttr.values()[(int) mT.attribute]) {

case PLUS:

sign();

term();

simpleExpressionTail();

return;

case MINUS:

sign();

term();

simpleExpressionTail();

return;

}

break;

case ID:

term();

simpleExpressionTail();

return;

case NUM:

term();

simpleExpressionTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.NOT), pair(TokType.OPENPAREN, null),

pair(TokType.ADDOP, AddopAttr.PLUS), pair(TokType.ADDOP, AddopAttr.MINUS),

pair(TokType.ID, null), pair(TokType.NUM, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void simpleExpressionTail() {

mSet =

new Token[] {pair(TokType.RELOP, null), pair(TokType.SEMICOLON, null),

pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.ELSE),

pair(TokType.RESWRD, ResWordAttr.THEN), pair(TokType.CLOSEBRACKET, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case THEN:

return;

case ELSE:

return;

case DO:

return;

}

break;

case CLOSEPAREN:

return;

case SEMICOLON:

return;

case COMMA:

return;

case CLOSEBRACKET:

return;

case RELOP:

return;

case ADDOP:

match(TokType.ADDOP, null);

term();

simpleExpressionTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.DO),

pair(TokType.CLOSEPAREN, null), pair(TokType.SEMICOLON, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEBRACKET, null),

pair(TokType.RELOP, null), pair(TokType.ADDOP, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void term() {

mSet =

new Token[] {pair(TokType.ADDOP, null), pair(TokType.RELOP, null),

pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null)};

// try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case NOT:

factor();

termTail();

return;

}

break;

case OPENPAREN:

factor();

termTail();

return;

case ID:

factor();

termTail();

return;

case NUM:

factor();

termTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.NOT), pair(TokType.OPENPAREN, null),

pair(TokType.ID, null), pair(TokType.NUM, null)};

wanted(toks);

sync();

/\*

\* Unreachable } catch (ParErr e) { sync(); }

\*/

}

void termTail() {

mSet =

new Token[] {pair(TokType.ADDOP, null), pair(TokType.RELOP, null),

pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case THEN:

return;

case ELSE:

return;

case DO:

return;

}

break;

case CLOSEPAREN:

return;

case SEMICOLON:

return;

case COMMA:

return;

case CLOSEBRACKET:

return;

case RELOP:

return;

case ADDOP:

return;

case MULOP:

match(TokType.MULOP, null);

factor();

termTail();

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.DO),

pair(TokType.CLOSEPAREN, null), pair(TokType.SEMICOLON, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEBRACKET, null),

pair(TokType.RELOP, null), pair(TokType.ADDOP, null), pair(TokType.MULOP, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void factor() {

mSet =

new Token[] {pair(TokType.ADDOP, null), pair(TokType.RELOP, null),

pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null), pair(TokType.MULOP, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case NOT:

match(TokType.RESWRD, ResWordAttr.NOT);

factor();

return;

}

break;

case OPENPAREN:

match(TokType.OPENPAREN, null);

expression();

match(TokType.CLOSEPAREN, null);

return;

case ID:

match(TokType.ID, null);

factorTail();

return;

case NUM:

match(TokType.NUM, null);

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.NOT), pair(TokType.OPENPAREN, null),

pair(TokType.ID, null), pair(TokType.NUM, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void factorTail() {

mSet =

new Token[] {pair(TokType.ADDOP, null), pair(TokType.RELOP, null),

pair(TokType.SEMICOLON, null), pair(TokType.RESWRD, ResWordAttr.END),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.CLOSEBRACKET, null), pair(TokType.COMMA, null),

pair(TokType.CLOSEPAREN, null), pair(TokType.MULOP, null)};

try {

switch (mT.type) {

case RESWRD:

switch (ResWordAttr.values()[(int) mT.attribute]) {

case END:

return;

case THEN:

return;

case ELSE:

return;

case DO:

return;

}

break;

case CLOSEPAREN:

return;

case SEMICOLON:

return;

case COMMA:

return;

case CLOSEBRACKET:

return;

case RELOP:

return;

case ADDOP:

return;

case MULOP:

return;

case OPENBRACKET:

match(TokType.OPENBRACKET, null);

expression();

match(TokType.CLOSEBRACKET, null);

return;

}

Token[] toks =

{pair(TokType.RESWRD, ResWordAttr.END), pair(TokType.RESWRD, ResWordAttr.THEN),

pair(TokType.RESWRD, ResWordAttr.ELSE), pair(TokType.RESWRD, ResWordAttr.DO),

pair(TokType.CLOSEPAREN, null), pair(TokType.SEMICOLON, null),

pair(TokType.COMMA, null), pair(TokType.CLOSEBRACKET, null),

pair(TokType.RELOP, null), pair(TokType.ADDOP, null), pair(TokType.MULOP, null),

pair(TokType.OPENBRACKET, null)};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

void sign() {

mSet =

new Token[] {pair(TokType.ID, null), pair(TokType.NUM, null),

pair(TokType.OPENPAREN, null), pair(TokType.RESWRD, ResWordAttr.NOT)};

try {

switch (mT.type) {

case ADDOP:

switch (AddopAttr.values()[(int) mT.attribute]) {

case PLUS:

match(TokType.ADDOP, AddopAttr.PLUS);

return;

case MINUS:

match(TokType.ADDOP, AddopAttr.MINUS);

return;

}

break;

}

Token[] toks = {};

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

}

package kuxhausen;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Scanner;

import java.io.\*;

import static java.lang.System.out;

import static kuxhausen.Token.\*;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Lexar {

private HashMap<String, Token> reservedWordTable = new HashMap<String, Token>();

private SourceBuffer source = new SourceBuffer();

private SourcePointer srcPos = new SourcePointer();

private SymbolTable symbols = new SymbolTable();

private ArrayList<Token> tokens = new ArrayList<Token>();

public Lexar(Scanner file) {

loadReservedWordTable();

while (file.hasNextLine()) {

// Read source into buffer

// Per project spec, only consider upto 71 characters per line including \n

String line = file.nextLine();

source.addLine(line.substring(0, Math.min(71, line.length())) + "\n");

}

file.close();

}

private void loadReservedWordTable() {

try {

Scanner wordFile = new Scanner(new BufferedReader(new FileReader("input/reservedwords.txt")));

while (wordFile.hasNextLine() && wordFile.hasNext()) {

String lexeme = wordFile.next();

String resType = wordFile.next();

int attribute = wordFile.nextInt();

if (resType.equals(TokType.ADDOP.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.ADDOP, attribute, lexeme, srcPos));

} else if (resType.equals(TokType.MULOP.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.MULOP, attribute, lexeme, srcPos));

} else {

for (ResWordAttr tt : ResWordAttr.values()) {

if (resType.equals(tt.toString())) {

reservedWordTable.put(lexeme, new Token(TokType.RESWRD, tt.ordinal(), lexeme, srcPos));

}

}

}

}

wordFile.close();

out.println("successfully loaded " + reservedWordTable.size()

+ " reserved words from reservedwords.txt");

} catch (FileNotFoundException e) {

out.println("reservedwords.txt not found");

}

}

public Token getNextToken() {

Token result = null;

result = reservedWordsMachine();

if (result == null) {

whitespaceMachine();

if (!source.hasNext(srcPos)) // check there is more after removing whitespace

return result;

result = idMachine();

}

if (result == null) {

result = realMachine();

}

if (result == null) {

result = intMachine();

}

if (result == null) {

result = relopMachine();

}

if (result == null) {

result = catchAllMachine();

}

if (result != null)

tokens.add(result);

return result;

}

private boolean isWhiteSpace(char c) {

if (c == ' ' || c == '\t' || c == '\n')

return true;

return false;

}

private boolean isLetter(char c) {

if (c >= 'a' && c <= 'z')

return true;

if (c >= 'A' && c <= 'Z')

return true;

return false;

}

private boolean isDigit(char c) {

if (c >= '0' && c <= '9')

return true;

return false;

}

private boolean isEOF(char c) {

return (c == '.');

}

private Token reservedWordsMachine() {

SourcePointer backup = srcPos.clone();

// first consume whitespace expected before id / reserved words

boolean hasConsumedWhitespace = false;

if (this.srcPos.lineNum == 0 && srcPos.charInLineNum == 0) {

hasConsumedWhitespace = true; // whitespace not needed before first char in source

}

while (source.hasNext(srcPos) && isWhiteSpace(source.peek(srcPos))) {

source.advanceChar(srcPos);

hasConsumedWhitespace = true;

}

if (hasConsumedWhitespace) {

String candidate = "";

// next consume one letter

if (source.hasNext(srcPos) && isLetter(source.peek(srcPos))) {

candidate += source.advanceChar(srcPos);

// next consume any following letters or digits

while (source.hasNext(srcPos)

&& (isLetter(source.peek(srcPos)) || isDigit(source.peek(srcPos)))) {

candidate += source.advanceChar(srcPos);

}

// if candidate is followed by whitespace or EOF

if (source.hasNext(srcPos)

&& (isWhiteSpace(source.peek(srcPos)) || isEOF(source.peek(srcPos)))) {

// check reserved word table

if (reservedWordTable.containsKey(candidate)) {

Token result = reservedWordTable.get(candidate).clone();

result.position = srcPos.clone();

return result;

}

}

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token idMachine() {

SourcePointer backup = srcPos.clone();

String candidate = "";

// consume one letter

if (source.hasNext(srcPos) && isLetter(source.peek(srcPos))) {

candidate += source.advanceChar(srcPos);

// next consume any following letters or digits

while (source.hasNext(srcPos)

&& (isLetter(source.peek(srcPos)) || isDigit(source.peek(srcPos)))) {

candidate += source.advanceChar(srcPos);

}

if (candidate.length() > 10)

return new Token(TokType.LEXERR, "Invalid ID: too long", candidate, srcPos);

// Check add id to symbol table

Token t = new Token(TokType.ID, candidate, candidate, srcPos);

if (!symbols.table.containsKey(candidate))

symbols.table.put(candidate, t);

return t;

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

/\*\*

\* consumes whitespace

\*/

private void whitespaceMachine() {

while (source.hasNext(srcPos) && isWhiteSpace(source.peek(srcPos))) {

source.advanceChar(srcPos);

}

}

private Token relopMachine() {

SourcePointer backup = srcPos.clone();

if (source.hasNext(srcPos)) {

String lex = "" + source.advanceChar(srcPos);

switch (lex) {

case "=":

return new Token(TokType.RELOP, RelopAttr.EQ.ordinal(), lex, srcPos);

case "<":

if (source.hasNext(srcPos)) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '>') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.NEQ.ordinal(), lex, srcPos);

} else if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.LTE.ordinal(), lex, srcPos);

} else {

return new Token(TokType.RELOP, RelopAttr.LT.ordinal(), lex, srcPos);

}

}

break;

case ">":

if (source.hasNext(srcPos)) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.RELOP, RelopAttr.GTE.ordinal(), lex, srcPos);

} else {

return new Token(TokType.RELOP, RelopAttr.GT.ordinal(), lex, srcPos);

}

}

break;

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token intMachine() {

SourcePointer backup = srcPos.clone();

if (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

String lex = "" + source.advanceChar(srcPos);

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

lex += source.advanceChar(srcPos);

}

if (lex.startsWith("00"))

return new Token(TokType.LEXERR, "Invalid INT: multiple leading zeros", lex, srcPos);

if (lex.length() > 10)

return new Token(TokType.LEXERR, "Invalid INT: too long", lex, srcPos);

return new Token(TokType.NUM, lex, lex, srcPos);

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token realMachine() {

SourcePointer backup = srcPos.clone();

String lex = "";

int xCount = 0;

boolean hasDot = false;

int yCount = 0;

boolean hasExp = false;

int zCount = 0;

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

xCount++;

lex += source.advanceChar(srcPos);

}

if (source.hasNext(srcPos) && source.peek(srcPos) == '.') {

hasDot = true;

lex += source.advanceChar(srcPos);

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

yCount++;

lex += source.advanceChar(srcPos);

}

}

SourcePointer notLongBackup = srcPos.clone();

if (source.hasNext(srcPos) && (source.peek(srcPos) == 'E' || source.peek(srcPos) == 'e')) {

hasExp = true;

lex += source.advanceChar(srcPos);

if (source.hasNext(srcPos) && (source.peek(srcPos) == '+' || source.peek(srcPos) == '-')) {

lex += source.advanceChar(srcPos);

}

while (source.hasNext(srcPos) && isDigit(source.peek(srcPos))) {

zCount++;

lex += source.advanceChar(srcPos);

}

}

if (xCount > 0 && hasDot && yCount > 0) {

if (lex.startsWith("00"))

return new Token(TokType.LEXERR, "Invalid REAL: multiple leading zeros in xx", lex, srcPos);

if (xCount > 5)

return new Token(TokType.LEXERR, "Invalid REAL: xx too long", lex, srcPos);

if (yCount > 5)

return new Token(TokType.LEXERR, "Invalid REAL: yy too long", lex, srcPos);

if (hasExp && zCount > 0) {

if (zCount > 2)

return new Token(TokType.LEXERR, "Invalid REAL: zz too long", lex, srcPos);

else if (lex.substring(lex.length() - zCount).startsWith("00"))

return new Token(TokType.LEXERR, "Invalid REAL: multiple leading zeros in zz", lex, srcPos);

else

return new Token(TokType.NUM, lex, lex, srcPos);

} else {

srcPos = notLongBackup;

return new Token(TokType.NUM, lex, lex, srcPos);

}

}

// if no token matched, revert source pointer and return null

srcPos = backup;

return null;

}

private Token catchAllMachine() {

SourcePointer backup = srcPos.clone();

String lex = "" + source.advanceChar(srcPos);

switch (lex) {

case "(":

return new Token(TokType.OPENPAREN, null, lex, srcPos);

case ")":

return new Token(TokType.CLOSEPAREN, null, lex, srcPos);

case ";":

return new Token(TokType.SEMICOLON, null, lex, srcPos);

case ",":

return new Token(TokType.COMMA, null, lex, srcPos);

case "[":

return new Token(TokType.OPENBRACKET, null, lex, srcPos);

case "]":

return new Token(TokType.CLOSEBRACKET, null, lex, srcPos);

case "+":

return new Token(TokType.ADDOP, AddopAttr.PLUS.ordinal(), lex, srcPos);

case "-":

return new Token(TokType.ADDOP, AddopAttr.MINUS.ordinal(), lex, srcPos);

case "\*":

return new Token(TokType.MULOP, MulopAttr.TIMES.ordinal(), lex, srcPos);

case "/":

return new Token(TokType.MULOP, MulopAttr.SLASH.ordinal(), lex, srcPos);

}

if (lex.equals(":")) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '=') {

lex += source.advanceChar(srcPos);

return new Token(TokType.ASSIGNOP, null, lex, srcPos);

} else

return new Token(TokType.COLON, null, lex, srcPos);

} else if (lex.equals(".")) {

if (source.hasNext(srcPos) && source.peek(srcPos) == '.') {

lex += source.advanceChar(srcPos);

return new Token(TokType.DOTDOT, null, lex, srcPos);

} else {

return new Token(TokType.EOF, null, lex, srcPos);

}

}

Token err = new Token(TokType.LEXERR, "Unrecog Symbol", lex, srcPos);

return err;

}

public void computeProjectZero() {

for (int i = 0; i < source.getNumLines(); i++) {

out.print(i + ". " + source.getLine(i));

}

}

public static Scanner getFile(String filepath) {

try {

return new Scanner(new BufferedReader(new FileReader(filepath)));

} catch (FileNotFoundException e) {

out.println("Source not found at " + filepath);

return null;

}

}

public ArrayList<Token> getTokenList() {

return tokens;

}

public SourceBuffer getSourceBuffer() {

return source;

}

}

package kuxhausen;

import java.util.ArrayList;

/\*\*

\* @author Eric Kuxhausen Stores source code with the requirements of: preserving line numbers,

\* providing access by line number, and facilitating per-character linear traversal with

\* SourcePointers.

\*/

public class SourceBuffer {

private ArrayList<String> sourceBuffer = new ArrayList<String>();

public SourceBuffer() {}

public void addLine(String line) {

sourceBuffer.add(line);

}

public int getNumLines() {

return sourceBuffer.size();

}

public String getLine(int number) {

return sourceBuffer.get(number);

}

public boolean hasNext(SourcePointer position) {

if (position.lineNum < sourceBuffer.size()

&& position.charInLineNum < sourceBuffer.get(position.lineNum).length()) {

return true;

} else

return false;

}

/\*\*

\* guard with hasNextCharacter() to prevent out of bounds issues

\*/

public char peek(SourcePointer position) {

return sourceBuffer.get(position.lineNum).charAt(position.charInLineNum);

}

public char advanceChar(SourcePointer position) {

if (hasNext(position)) {

char result = peek(position);

if (position.charInLineNum < sourceBuffer.get(position.lineNum).length() - 1) {

position.charInLineNum++;

} else {

position.lineNum++;

position.charInLineNum = 0;

}

return result;

}

return 0;

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

public class SourcePointer implements Cloneable {

public int lineNum;

public int charInLineNum;

@Override

public SourcePointer clone() {

SourcePointer copy = new SourcePointer();

copy.lineNum = lineNum;

copy.charInLineNum = charInLineNum;

return copy;

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

import java.util.HashMap;

public class SymbolTable {

public HashMap<String, Token> table;

public SymbolTable() {

table = new HashMap<String, Token>();

}

}

package kuxhausen;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Token implements Cloneable {

public TokType type;

public Object attribute;

public String lexeme;

public SourcePointer position;

public Token(TokType t, int attr, String lex, SourcePointer pos) {

this(t, (Object) attr, lex, pos);

}

public Token(TokType t, String attr, String lex, SourcePointer pos) {

this(t, (Object) attr, lex, pos);

}

private Token(TokType t, Object attr, String lex, SourcePointer pos) {

type = t;

attribute = attr;

lexeme = lex;

position = (pos != null) ? pos.clone() : null;

}

public Token clone() {

return new Token(type, attribute, lexeme, position.clone());

}

public PasType getNumType() {

if (type == TokType.NUM) {

if (lexeme.contains("."))

return PasType.REAL;

else

return PasType.INT;

}

return PasType.ERR;

}

public String getAttribute() {

if (attribute != null) {

if (attribute instanceof Integer && (int) attribute != -1) {

switch (type) {

case RESWRD:

return ResWordAttr.values()[(int) attribute].toString();

case RELOP:

return RelopAttr.values()[(int) attribute].toString();

case ADDOP:

return AddopAttr.values()[(int) attribute].toString();

case MULOP:

return MulopAttr.values()[(int) attribute].toString();

}

} else if (!(attribute instanceof Integer)) {

return attribute.toString();

}

}

return "NULL";

}

public RelopAttr getRelop() {

return RelopAttr.values()[(int) attribute];

}

public MulopAttr getMulop() {

return MulopAttr.values()[(int) attribute];

}

public AddopAttr getAddop() {

return AddopAttr.values()[(int) attribute];

}

public boolean fullTypeMatch(Token other) {

if (type == other.type) {

// if one of these types, have to compare attributes as well

if (type == TokType.RESWRD || type == TokType.RELOP || type == TokType.ADDOP

|| type == TokType.MULOP) {

// unless the attribute wasn't specified, in which case it's a wildcard

if ((int) attribute == -1 || (int) other.attribute == -1) {

return true;

}

if (((int) attribute) == ((int) other.attribute)) {

return true;

}

} else {

return true;

}

}

return false;

}

public static enum TokType {

RESWRD, ID, EOF, NUM, RELOP, ADDOP, MULOP, LEXERR, SYNTAXERR, SEMANTICERR, OPENPAREN, CLOSEPAREN, SEMICOLON, COMMA, COLON, OPENBRACKET, DOTDOT, CLOSEBRACKET, ASSIGNOP, $

}

public static enum ResWordAttr {

PROGRAM, VAR, ARRAY, OF, INT\_NAME, REAL\_NAME, PROC, BEGIN, END, IF, THEN, ELSE, WHILE, DO, CALL, NOT

}

public static enum RelopAttr {

EQ, NEQ, LT, LTE, GTE, GT

}

public static enum AddopAttr {

PLUS, MINUS, OR

}

public static enum MulopAttr {

TIMES, SLASH, DIV, MOD, AND

}

}

package kuxhausen;

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.ArrayList;

import static kuxhausen.Token.\*;

public class Utils {

public static void writeListingFile(String filename, ArrayList<Token> tokens, SourceBuffer source) {

PrintWriter output = null;

try {

output = new PrintWriter(filename);

} catch (FileNotFoundException e) {

}

int lineNo = -1;

for (Token t : tokens) {

while (t.position.lineNum > lineNo) {

lineNo++;

output.print(String.format("%-8s", "" + (lineNo + 1)) + source.getLine(lineNo));

}

if (t.type == TokType.LEXERR)

output.println("LEXERR: " + t.attribute);

if (t.type == TokType.SYNTAXERR)

output.println("SYNTAXERR: " + t.attribute);

}

output.close();

}

public static void writeTokenFile(String filename, ArrayList<Token> tokens) {

PrintWriter output = null;

try {

output = new PrintWriter(filename);

} catch (FileNotFoundException e) {

}

String formatting = "%-10s%-20s%-20s%-10s";

output.println(String.format(formatting, "Line No.", "Lexeme", "TOKEN-TYPE", "ATTRIBUTE"));

for (Token t : tokens) {

if (t.type != TokType.$) {

output.println(String.format(formatting, (t.position.lineNum + 1), t.lexeme,

t.type.toString(), t.getAttribute()));

}

}

output.close();

}

}