Type and Scope Checking, Memory Address Computation

Compilers Project 3, 4

Eric Kuxhausen

Table of Contents

Introduction 2

L-Attributed Definition 2

Implementation 3

Discussion and Conclusions 3

References 3

Compiler & Pascal References 3

Java Programming References 4

Appendix I: Sample Inputs and Outputs 5

semanticTest1.pas 5

semanticTest1.listing 5

semanticTest1.token 5

semanticTest1.loc 7

semanticTest2.pas 7

semanticTest2.listing 7

semanticTest2.token 8

semanticTest2.loc 9

Appendix II: Program Listings 10

# Introduction

These projects consist of a semantic analyzer for our version of the pascal language. This Semantic Analyzer evaluates the tokens produced by the Lexar after they have already undergone syntax analysis by the Parser. The Semantic Analyzer performs declarations processing, where variables, procedure parameters, procedures, and programs are type checked and scope checked. Additionally, it performs type-checking across the rest of the parse tree, catching errors such as mixing incompatible types in expressions. Finally, it performs memory address computations for variables within their locals scopes.  
Methodology

## L-Attributed Definition

See attached handwritten pages

# Implementation

As with the previous projects, I have written my project in the java language, compiled with the standard oracle compiler for java 1.7. As the semantic analysis must be embedded in the parser, but I wanted my project to be backwards compatible with the previous project deliverables (i.e. still able to generate project 2 outputs with syntax but not semantic analysis), I embedded my semantic checking logic in a copy of the parser.

Pascal types used for type checking:

* INT
* REAL
* AINT
* AREAL
* PPINT
* PPREAL
* PPAINT
* PPAREAL
* PGPP // for the program parameters
* BOOL
* ERR
* ERRSTAR,

# Discussion and Conclusions

This project was conceptually challenging to start, but once I had figured out valid types for the lower productions, I became to understand the larger picture. I was somewhat unsure how to format the output of memory address computations, so I picked a format that shows scope nesting. Another major challenge of this project was creating sufficiently helpful error messages, especially in cases where multiple different error messages exist for the same subproduction, such as when any one input is wrong in a into a multi-input type checking expression.

# 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

## 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

## semanticTest1.loc

NEW SCOPE: example

0 x INT

4 y INT

8 z REAL

NEW SCOPE: gcd

END SCOPE

END SCOPE

## semanticTest2.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 (z) then x:=2 mod a

else f:= b/2

end ;

begin

x := 5;

call gcd(3.0, x)

end .

## semanticTest2.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 (z) then x:=2 mod a

SYNTAXERR: type errorBOOL OR REAL cannot be used together

SYNTAXERR: Type error: expected boolean expression, got ERR

8 else f:= b/2

SYNTAXERR: No var or proc\_param named f defined yet in this scope

9 end ;

SYNTAXERR: type errorREAL SLASH INT cannot be used together

10 begin

11 x := 5;

12 call gcd(3.0, x)

SYNTAXERR: Incorrect procedure param type: got REAL, expected INT

SYNTAXERR: Incorrect procedure param type: got INT, expected REAL

13 end .

## semanticTest2.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 z ID z

7 ) CLOSEPAREN NULL

7 then RESWRD THEN

7 SEMANTICERR type errorBOOL OR REAL cannot be used together

7 SEMANTICERR Type error: expected boolean expression, got ERR

7 x ID x

7 := ASSIGNOP NULL

7 2 NUM 2

7 mod MULOP MOD

7 a ID a

8 else RESWRD ELSE

8 f ID f

8 := ASSIGNOP NULL

8 f SEMANTICERR No var or proc\_param named f defined yet in this scope

8 b ID b

8 / MULOP SLASH

8 2 NUM 2

9 end RESWRD END

9 SEMANTICERR type errorREAL SLASH INT cannot be used together

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 3.0 NUM 3.0

12 , COMMA NULL

12 SEMANTICERR Incorrect procedure param type: got REAL, expected INT

12 x ID x

12 ) CLOSEPAREN NULL

12 SEMANTICERR Incorrect procedure param type: got INT, expected REAL

13 end RESWRD END

13 . EOF NULL

## semanticTest2.loc

NEW SCOPE: example

0 x INT

4 y INT

8 z REAL

NEW SCOPE: gcd

END SCOPE

END SCOPE

# Appendix II: Program Listings

package kuxhausen;

import java.util.Scanner;

/\*\*

\* @author Eric Kuxhausen

\*/

public class Project34 {

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);

DecoratedParser p = new DecoratedParser(l, ("output/" + filename + ".loc"));

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

l.getSourceBuffer());

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

}

}

}

}

package kuxhausen;

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.ArrayDeque;

import java.util.ArrayList;

import java.util.List;

import kuxhausen.Token.\*;

/\*\*

\* @author Eric Kuxhausen

\*/

public class DecoratedParser {

private Lexar mL;

/\*\*

\* current Token

\*/

private Token mT;

private Token mConsumed;

private SourcePointer mLine;

/\*\*

\* sync set for the current nonTerminal

\*/

private Token[] mSet;

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

private ArrayDeque<GreenNode> mScope = new ArrayDeque<GreenNode>();

{

GreenNode invisibleRoot = new GreenNode();

invisibleRoot.setName("");

mScope.add(invisibleRoot);

}

private PrintWriter output;

DecoratedParser(Lexar lex, String loc) {

try {

output = new PrintWriter(loc);

} catch (FileNotFoundException e) {

}

mL = lex;

consumeToken();

program();

exitScope();

output.close();

}

private void consumeToken() {

mConsumed = mT;

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;

}

public void checkAddGreen(String name) {

GreenNode green = new GreenNode();

green.setName(name);

boolean hasConflict = false;

for (GreenNode g : mScope) {

for (Node n : g.getChildren()) {

if (n instanceof GreenNode && n.getName().equals(name)) {

hasConflict = true;

}

}

}

if (hasConflict) {

mTokens.add(new Token(TokType.SEMANTICERR, "A program or procedure named " + green.getName()

+ " already defined in this scope", name, mLine));

// go ahead and add node anyway with modified name so that subtree can be typechecked

green.setName(green.getName() + "#");

}

mScope.getFirst().getChildren().add(green);

mScope.addFirst(green);

output.println(" NEW SCOPE: "+green.getName());

}

public void checkAddBlue(String name, PasType type) {

boolean hasConflict = false;

for (GreenNode g : mScope) {

for (Node n : g.getChildren()) {

if (n instanceof BlueNode && n.getName().equals(name)) {

hasConflict = true;

}

}

}

if (hasConflict) {

mTokens.add(new Token(TokType.SEMANTICERR, "A var or proc\_param named " + name

+ " already defined in this scope", name, mLine));

} else {

BlueNode b = new BlueNode();

b.setName(name);

b.setType(type);

mScope.getFirst().getChildren().add(b);

}

}

public void checkGreen(String name) {

for (GreenNode g : mScope) {

for (Node n : g.getChildren()) {

if (n instanceof GreenNode && n.getName().equals(name)) {

return;

}

}

}

mTokens.add(new Token(TokType.SEMANTICERR, "No program or procedured named " + name

+ " defined yet in this scope", name, mLine));

}

public List<BlueNode> getPPs(String name) {

ArrayList<BlueNode> result = new ArrayList<BlueNode>();

GreenNode parrent = null;

for (GreenNode g : mScope) {

for (Node n : g.getChildren()) {

if (n instanceof GreenNode && n.getName().equals(name)) {

parrent = (GreenNode) n;

}

}

}

if (parrent != null) {

for (Node n : parrent.getChildren()) {

if (n instanceof BlueNode) {

switch (((BlueNode) n).getType()) {

case PPAINT:

result.add((BlueNode) n);

break;

case PPAREAL:

result.add((BlueNode) n);

break;

case PPINT:

result.add((BlueNode) n);

break;

case PPREAL:

result.add((BlueNode) n);

break;

}

}

}

}

return result;

}

public PasType checkBlue(String name) {

for (GreenNode g : mScope) {

for (Node n : g.getChildren()) {

if (n instanceof BlueNode && n.getName().equals(name)) {

return ((BlueNode) n).getType();

}

}

}

mTokens.add(new Token(TokType.SEMANTICERR, "No var or proc\_param named " + name

+ " defined yet in this scope", name, mLine));

return PasType.ERR;

}

public void exitScope() {

mScope.removeFirst();

output.println(" END SCOPE");

}

public PasType reportErrStar(String msg) {

Token t = new Token(TokType.SEMANTICERR, msg, "", mLine);

mTokens.add(t);

return PasType.ERR;

}

public void computeOffset(Token id, TypeWidth tw) {

output.println(mScope.getFirst().scopeOffset+" "+id.lexeme+" "+tw.type.toString());

mScope.getFirst().scopeOffset +=tw.width;

}

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);

Token id = mConsumed;

checkAddGreen(id.lexeme);

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);

Token id = mConsumed;

checkAddBlue(id.lexeme, PasType.PGPP);

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);

Token id = mConsumed;

checkAddBlue(id.lexeme, PasType.PGPP);

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);

Token id = mConsumed;

match(TokType.COLON, null);

TypeWidth type = type();

match(TokType.SEMICOLON, null);

checkAddBlue(id.lexeme, type.type);

computeOffset(id, type);

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);

Token id = mConsumed;

match(TokType.COLON, null);

TypeWidth type = type();

match(TokType.SEMICOLON, null);

checkAddBlue(id.lexeme, type.type);

computeOffset(id, type);

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();

}

}

TypeWidth 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);

Token n1 = mConsumed;

PasType num1 = n1.getNumType();

match(TokType.DOTDOT, null);

match(TokType.NUM, null);

Token n2 = mConsumed;

PasType num2 = n2.getNumType();

match(TokType.CLOSEBRACKET, null);

match(TokType.RESWRD, ResWordAttr.OF);

PasType st = standardType();

if (num1 == PasType.INT && num2 == PasType.INT && st == PasType.INT) {

return new TypeWidth(PasType.AINT,

4 \* (1 + Integer.valueOf(n2.lexeme) - Integer.valueOf(n1.lexeme)));

} else if (num1 == PasType.INT && num2 == PasType.INT && st == PasType.REAL) {

return new TypeWidth(PasType.AREAL,

4 \* (1 + Integer.valueOf(n2.lexeme) - Integer.valueOf(n1.lexeme)));

}

case INT\_NAME:

standardType();

return new TypeWidth(PasType.INT, 4);

case REAL\_NAME:

standardType();

return new TypeWidth(PasType.REAL, 8);

}

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();

}

return new TypeWidth(PasType.ERR, 0);

}

PasType 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 PasType.INT;

case REAL\_NAME:

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

return PasType.REAL;

}

break;

}

Token[] toks =

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

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

return PasType.ERR;

}

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();

exitScope();

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);

Token id = mConsumed;

checkAddGreen(id.lexeme);

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);

Token id = mConsumed;

match(TokType.COLON, null);

PasType type = type().type;

PasType x = PasType.ERR;

switch (type) {

case INT:

x = PasType.PPINT;

break;

case REAL:

x = PasType.PPREAL;

break;

case AINT:

x = PasType.PPAINT;

break;

case AREAL:

x = PasType.PPAREAL;

break;

}

checkAddBlue(id.lexeme, x);

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);

Token id = mConsumed;

match(TokType.COLON, null);

PasType type = type().type;

PasType x = PasType.ERR;

switch (type) {

case INT:

x = PasType.PPINT;

break;

case REAL:

x = PasType.PPREAL;

break;

case AINT:

x = PasType.PPAINT;

break;

case AREAL:

x = PasType.PPAREAL;

break;

}

checkAddBlue(id.lexeme, x);

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);

PasType exp1 = expression();

if (exp1 != PasType.BOOL)

reportErrStar("Type error: expected boolean expression, got " + exp1.toString());

match(TokType.RESWRD, ResWordAttr.THEN);

statement();

statementTail();

return;

case WHILE:

match(TokType.RESWRD, ResWordAttr.WHILE);

PasType exp2 = expression();

if (exp2 != PasType.BOOL)

reportErrStar("Type error: expected boolean expression, got " + exp2.toString());

match(TokType.RESWRD, ResWordAttr.DO);

statement();

return;

case CALL:

procedureStatment();

return;

}

break;

case ID:

PasType var = variable();

match(TokType.ASSIGNOP, null);

PasType exp3 = expression();

if (var == PasType.ERR || exp3 == PasType.ERR)

return;

else if (var == PasType.INT && exp3 == PasType.INT)

return;

else if (var == PasType.REAL && exp3 == PasType.REAL)

return;

else

reportErrStar("Type error: cannot assign " + exp3 + " to a " + var);

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();

}

}

PasType variable() {

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

try {

switch (mT.type) {

case ID:

match(TokType.ID, null);

Token id = mConsumed;

PasType idT = checkBlue(id.lexeme);

PasType varT = variableTail();

if (idT == PasType.ERR || varT == PasType.ERR)

return PasType.ERR;

else if (varT == PasType.INT) {

switch (idT) {

case AINT:

return PasType.INT;

case PPAINT:

return PasType.INT;

case AREAL:

return PasType.REAL;

case PPAREAL:

return PasType.REAL;

default:

return reportErrStar("Array type expected, " + idT.toString() + " recieved");

}

} else if (varT == PasType.NULL) {

switch (idT) {

case INT:

return PasType.INT;

case PPINT:

return PasType.INT;

case REAL:

return PasType.REAL;

case PPREAL:

return PasType.REAL;

default:

return reportErrStar("Numeric type expected, " + idT.toString() + " recieved");

}

} else {

return reportErrStar("Invalid array index type, " + idT.toString() + " recieved");

}

}

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

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

return PasType.ERR;

}

PasType variableTail() {

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

try {

switch (mT.type) {

case OPENBRACKET:

match(TokType.OPENBRACKET, null);

PasType exp = expression();

match(TokType.CLOSEBRACKET, null);

return exp;

case ASSIGNOP:

return PasType.NULL;

}

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

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

return PasType.ERR;

}

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);

Token id = mConsumed;

checkGreen(id.lexeme);

int numDesired = getPPs(id.lexeme).size();

int numSeen = procedureStatementTail(new PPPair(id.lexeme, 0));

if (numDesired != numSeen)

reportErrStar("procedure " + id.lexeme + " called with " + numSeen

+ " params, yet expected " + numDesired);

return;

}

break;

}

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

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

}

int procedureStatementTail(PPPair i) {

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 0;

case ELSE:

return 0;

}

break;

case OPENPAREN:

match(TokType.OPENPAREN, null);

int listNum = expressionList(i);

match(TokType.CLOSEPAREN, null);

return listNum;

case SEMICOLON:

return 0;

}

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();

}

return 0;

}

int expressionList(PPPair i) {

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

// try {

switch (mT.type) {

case RESWRD:

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

case NOT:

PasType exp1 = expression();

if (getPPs(i.procName).size() <= i.paramNum)

reportErrStar("Unexpected procedure param of type" + exp1.toString());

else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp1)

reportErrStar("Incorrect procedure param type: got " + exp1 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

}

break;

case OPENPAREN:

PasType exp2 = expression();

if (getPPs(i.procName).size() <= i.paramNum)

reportErrStar("Unexpected procedure param of type" + exp2.toString());

else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp2)

reportErrStar("Incorrect procedure param type: got " + exp2 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

case ADDOP:

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

case PLUS:

PasType exp3 = expression();

if (getPPs(i.procName).size() <= i.paramNum)

reportErrStar("Unexpected procedure param of type" + exp3.toString());

else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp3)

reportErrStar("Incorrect procedure param type: got " + exp3 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

case MINUS:

PasType exp4 = expression();

if (getPPs(i.procName).size() <= i.paramNum)

reportErrStar("Unexpected procedure param of type" + exp4.toString());

else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp4)

reportErrStar("Incorrect procedure param type: got " + exp4 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

}

break;

case ID:

PasType exp5 = expression();

if (getPPs(i.procName).size() <= i.paramNum)

reportErrStar("Unexpected procedure param of type" + exp5.toString());

else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp5)

reportErrStar("Incorrect procedure param type: got " + exp5 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

case NUM:

PasType exp6 = expression();

if (getPPs(i.procName).size() <= i.paramNum) {

reportErrStar("Unexpected procedure param of type" + exp6.toString());

} else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp6) {

reportErrStar("Incorrect procedure param type: got " + exp6 + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

}

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

}

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(); }

\*/

return 0;

}

int expressionListTail(PPPair i) {

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

try {

switch (mT.type) {

case CLOSEPAREN:

return i.paramNum;

case COMMA:

match(TokType.COMMA, null);

PasType exp = expression();

if (getPPs(i.procName).size() <= i.paramNum) {

reportErrStar("Unexpected procedure param of type" + exp.toString());

} else if (getPPs(i.procName).get(i.paramNum).getPPFreeType() != exp) {

reportErrStar("Incorrect procedure param type: got " + exp + ", expected "

+ getPPs(i.procName).get(i.paramNum).getPPFreeType());

}

return expressionListTail(new PPPair(i.procName, i.paramNum + 1));

}

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

wanted(toks);

sync();

} catch (SyntaxErr e) {

sync();

}

return 0;

}

PasType 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:

PasType se1 = simpleExpression();

return expressionTail(se1);

}

break;

case OPENPAREN:

PasType se2 = simpleExpression();

return expressionTail(se2);

case ADDOP:

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

case PLUS:

PasType se3 = simpleExpression();

return expressionTail(se3);

case MINUS:

PasType se4 = simpleExpression();

return expressionTail(se4);

}

break;

case ID:

PasType se5 = simpleExpression();

return expressionTail(se5);

case NUM:

PasType se6 = simpleExpression();

return expressionTail(se6);

}

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(); }

\*/

return PasType.ERR;

}

PasType expressionTail(PasType i) {

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 i;

case THEN:

return i;

case ELSE:

return i;

case DO:

return i;

}

break;

case CLOSEPAREN:

return i;

case SEMICOLON:

return i;

case COMMA:

return i;

case CLOSEBRACKET:

return i;

case RELOP:

match(TokType.RELOP, null);

Token relop = mConsumed;

PasType se = simpleExpression();

if (i == PasType.ERR || se == PasType.ERR)

return PasType.ERR;

else if (i == PasType.BOOL && relop.getRelop() == RelopAttr.EQ && se == PasType.BOOL)

return PasType.BOOL;

else if (i == PasType.BOOL && relop.getRelop() == RelopAttr.NEQ && se == PasType.BOOL)

return PasType.BOOL;

else if (i == PasType.INT && se == PasType.INT)

return PasType.BOOL;

else if (i == PasType.REAL && se == PasType.REAL)

return PasType.BOOL;

else

return reportErrStar("type error" + i.toString() + " " + relop.getAttribute() + " "

+ se.toString() + " cannot be used together");

}

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();

}

return PasType.ERR;

}

PasType 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:

PasType term = term();

return simpleExpressionTail(term);

}

break;

case OPENPAREN:

PasType term1 = term();

return simpleExpressionTail(term1);

case ADDOP:

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

case PLUS:

sign();

PasType term2 = term();

PasType set1 = simpleExpressionTail(term2);

if (set1 == PasType.BOOL)

return reportErrStar("Expected num after +,- but recieved BOOL");

else

return set1;

case MINUS:

sign();

PasType term3 = term();

PasType set2 = simpleExpressionTail(term3);

if (set2 == PasType.BOOL)

return reportErrStar("Expected num after +,- but recieved BOOL");

else

return set2;

}

break;

case ID:

PasType term4 = term();

return simpleExpressionTail(term4);

case NUM:

PasType term5 = term();

return simpleExpressionTail(term5);

}

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(); }

\*/

return PasType.ERR;

}

PasType simpleExpressionTail(PasType i) {

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 i;

case THEN:

return i;

case ELSE:

return i;

case DO:

return i;

}

break;

case CLOSEPAREN:

return i;

case SEMICOLON:

return i;

case COMMA:

return i;

case CLOSEBRACKET:

return i;

case RELOP:

return i;

case ADDOP:

match(TokType.ADDOP, null);

Token addop = mConsumed;

PasType term = term();

PasType set1i;

if (i == PasType.ERR || term == PasType.ERR)

set1i = PasType.ERR;

else if (i == PasType.BOOL && addop.getAddop() == AddopAttr.OR && term == PasType.BOOL)

set1i = PasType.BOOL;

else if (i == PasType.INT && addop.getAddop() != AddopAttr.OR && term == PasType.INT)

set1i = PasType.INT;

else if (i == PasType.REAL && addop.getAddop() != AddopAttr.OR && term == PasType.REAL)

set1i = PasType.REAL;

else

set1i =

reportErrStar("type error" + i.toString() + " " + addop.getAttribute() + " "

+ term.toString() + " cannot be used together");

return simpleExpressionTail(set1i);

}

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();

}

return PasType.ERR;

}

PasType 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:

PasType fact = factor();

return termTail(fact);

}

break;

case OPENPAREN:

PasType fact1 = factor();

return termTail(fact1);

case ID:

PasType fact2 = factor();

return termTail(fact2);

case NUM:

PasType fact3 = factor();

return termTail(fact3);

}

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(); }

\*/

return PasType.ERR;

}

PasType termTail(PasType i) {

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 i;

case THEN:

return i;

case ELSE:

return i;

case DO:

return i;

}

break;

case CLOSEPAREN:

return i;

case SEMICOLON:

return i;

case COMMA:

return i;

case CLOSEBRACKET:

return i;

case RELOP:

return i;

case ADDOP:

return i;

case MULOP:

match(TokType.MULOP, null);

Token mulop = mConsumed;

PasType fact = factor();

PasType term1i;

if (i == PasType.ERR || fact == PasType.ERR)

term1i = PasType.ERR;

else if (i == PasType.BOOL && mulop.getMulop() == MulopAttr.AND && fact == PasType.BOOL)

term1i = PasType.BOOL;

else if (i == PasType.INT && mulop.getMulop() != MulopAttr.AND && fact == PasType.INT)

term1i = PasType.INT;

else if (i == PasType.REAL && mulop.getMulop() != MulopAttr.AND && fact == PasType.REAL)

term1i = PasType.REAL;

else

term1i =

reportErrStar("type error" + i.toString() + " " + mulop.getAttribute() + " "

+ fact.toString() + " cannot be used together");

return termTail(term1i);

}

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();

}

return PasType.ERR;

}

PasType 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);

PasType fOne = factor();

switch (fOne) {

case BOOL:

return PasType.BOOL;

case ERR:

return PasType.ERR;

default:

return reportErrStar("Attempted to use nonBoolean type as Boolean");

}

}

break;

case OPENPAREN:

match(TokType.OPENPAREN, null);

PasType eType = expression();

match(TokType.CLOSEPAREN, null);

return eType;

case ID:

match(TokType.ID, null);

PasType idType = checkBlue(mConsumed.getAttribute());

PasType fTail = factorTail();

if (fTail == PasType.ERR || idType == PasType.ERR)

return PasType.ERR;

if (fTail == PasType.INT) {

switch (idType) {

case AINT:

return PasType.INT;

case PPAINT:

return PasType.INT;

case AREAL:

return PasType.REAL;

case PPAREAL:

return PasType.REAL;

default:

return reportErrStar("Array type expected, " + idType.toString() + " recieved");

}

} else if (fTail == PasType.NULL) {

switch (idType) {

case INT:

return PasType.INT;

case PPINT:

return PasType.INT;

case REAL:

return PasType.REAL;

case PPREAL:

return PasType.REAL;

default:

return reportErrStar("Numeric type expected, " + idType.toString() + " recieved");

}

} else {

return reportErrStar("Invalid array index type, " + idType.toString() + " recieved");

}

case NUM:

match(TokType.NUM, null);

return mConsumed.getNumType();

}

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();

}

return PasType.ERR;

}

PasType 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 PasType.NULL;

case THEN:

return PasType.NULL;

case ELSE:

return PasType.NULL;

case DO:

return PasType.NULL;

}

break;

case CLOSEPAREN:

return PasType.NULL;

case SEMICOLON:

return PasType.NULL;

case COMMA:

return PasType.NULL;

case CLOSEBRACKET:

return PasType.NULL;

case RELOP:

return PasType.NULL;

case ADDOP:

return PasType.NULL;

case MULOP:

return PasType.NULL;

case OPENBRACKET:

match(TokType.OPENBRACKET, null);

PasType exp = expression();

match(TokType.CLOSEBRACKET, null);

return exp;

}

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();

}

return PasType.ERR;

}

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;

public enum PasType {

INT, REAL, AINT, AREAL, PPINT, PPREAL, PPAINT, PPAREAL, PGPP, BOOL, ERR, ERRSTAR, NULL,

}

package kuxhausen;

public class TypeWidth {

PasType type;

int width;

TypeWidth(PasType t, int w) {

type = t;

width = w;

}

}

package kuxhausen;

public class PPPair {

String procName;

int paramNum;

PPPair(String p, int n) {

procName = p;

paramNum = n;

}

}

package kuxhausen;

public interface Node {

abstract String getName();

abstract void setName(String name);

}

package kuxhausen;

public class BlueNode implements Node {

private String mName;

private PasType mType;

@Override

public String getName() {

return mName;

}

@Override

public void setName(String name) {

mName = name;

}

public PasType getType() {

return mType;

}

public PasType getPPFreeType(){

switch(mType){

case PPINT: return PasType.INT;

case PPREAL: return PasType.REAL;

case PPAINT: return PasType.AINT;

case PPAREAL: return PasType.AREAL;

default: return mType;

}

}

public void setType(PasType t) {

mType = t;

}

}

package kuxhausen;

import java.util.ArrayList;

public class GreenNode implements Node {

private String mName;

private ArrayList<Node> mChildren = new ArrayList<Node>();

public int scopeOffset;

@Override

public String getName() {

return mName;

}

@Override

public void setName(String name) {

mName = name;

}

public ArrayList<Node> getChildren() {

return mChildren;

}

}

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();

}

}