**Compiler Principle Experiment 2:**

**syntax analysis**

**1 Introduction**

Following the last experiment, we still use Cool, the Classroom Object Oriented Language, as our source language. In this experiment, we are going to construct a parser for Cool. We makes use of two tools: the parser generator named CUP and a package for manipulating trees. The output of our parser will be an abstract syntax tree (AST). We will construct this AST using semantic actions of the parser generator.

Also, after completing our parser, we will use lots of test programs to verify the completeness and correctness of it.

**2 The Grammars Definition of Cool**

Before we dive into the implementation of a Cool Parser, we need first specify the grammars of Cool exactly( as the requirement of this experiment, we use **Backus–Naur form**).

According to the Cool Reference Manual, we have these terminals and non-terminals below:

**terminal CLASS, ELSE, FI, IF, IN, INHERITS, LET, LET\_STMT, LOOP, POOL, THEN, WHILE;**

**terminal CASE, ESAC, OF, DARROW, NEW, ISVOID;**

**terminal ASSIGN, NOT, LE, ERROR;**

**terminal PLUS, DIV, MINUS, MULT, EQ, LT, DOT, NEG, COMMA, SEMI, COLON;**

**terminal LPAREN, RPAREN, AT, LBRACE, RBRACE;**

**terminal AbstractSymbol STR\_CONST, INT\_CONST;**

**terminal Boolean BOOL\_CONST;**

**terminal AbstractSymbol TYPEID, OBJECTID;**

**nonterminal programc program;**

**nonterminal Classes class\_list;**

**nonterminal class\_c class;**

Notice that we specify the type of some terminals and non-terminals. And also, we can add any non-terminals if we wish.

**The BNF:**

**( In our text book, the syntax of a BNF is non-terminal → terminal, however, in the real word the syntax of a BNF is slightly different.**

**See** [**https://en.wikipedia.org/wiki/Backus%E2%80%93Naur\_form**](https://en.wikipedia.org/wiki/Backus–Naur_form) **in detail.)**

**2.1 Program**

At the beginning, we have grammars like these:

**<program> ::= <class\_list>**

**<class\_list> ::= <class\_list> <class>**

**| <class>**

**2.2 Class**

According the cool reference manual, the class definitions have the form:

class <type> [ inherits <type> ] {

<feature\_list>

};

The notation [ ...] denotes an optional construct.

Hence the grammar of class is:

**<class> ::= CLASS TYPEID <inheritance> LBRACE <feature\_list\_opt> RBRACE SEMI**

And we add a new non-terminal – inheritance:

**<inheritance> ::= INHERITS TYPEID**

**| <empty>**

Then the feature\_list\_opt grammar:

**<feature\_list\_opt> ::= <feature\_list>**

**| <empty>**

**<feature\_list> ::= <feature\_list> <feature>**

**| <feature>**

In addition, we need add two non-terminals:

**nonterminal Features feature\_list;**

**nonterminal Features feature\_list\_opt;**

**2.3 Feature**

The body of a class definition consists of a list of feature definitions. A feature is **either an attribute or a method**.

An attribute of class A specifies a variable that is part of the state of objects of class A.

A method of class A is a procedure that may manipulate the variables and objects of class A.

Thus the feature grammar can be:

**<feature> ::= <attribute>**

**| <method>**

**2.4 Attribute**

**<attribute> ::= OBJECTID COLON TYPEID assign\_opt SEMI**

**<assign\_opt> ::= ASSIGN <expression>**

**| <empty>**

**2.5 Method**

**<method> ::= OBJECTID LPAREN <formal\_list\_opt> RPAREN COLON TYPEID LBRACE <****expression> RBRACE SEMI**

Again we need add some new non-terminals:

nonterminal Feature feature;

nonterminal Attribute attribute;

nonterminal Assign assign\_opt;

nonterminal Expression expression;

nonterminal Method method;

nonterminal Formals formal\_list\_opt;

Then we dive in the method grammars:

**<formal\_list\_opt> ::= <formal\_list>**

**| <empty>**

**<formal\_list> ::= <formal\_list> ,** **<****formal>**

**| <formal>**

**<formal> ::= OBJECTID COLON TYPEID**

Now we are getting into **the most complex part**. Expressions are the largest syntactic category in Cool.

**2.5.1 Constants**

The simplest expressions are constants. The **boolean** **constants** are true and false. **Integer** **constants** are unsigned strings of digits such as 0, 123, and 007. **String constants** are sequences of characters enclosed in double quotes, such as "This is a string." String constants may be at most 1024 characters long.

The constants belong to the basic classes Bool, Int, and String. The value of a constant is an object of the appropriate basic class.

The constants grammars:

**<constant> ::= STR\_CONST**

**| INT\_CONST**

**| BOOL\_CONST**

**<expression> ::= <constant>**

**| ...**

**2.5.2 Identifiers**

**<expression> ::= <constant>**

**| OBJECTID**

**| ...**

**2.5.3 Assignment**

An assignment has the form

<id> <- <expr>

The static type of the expression must conform to the declared type of the identifier. The value is the value of the expression. The static type of an assignment is the static type of <expr>.

**<expression> ::= <constant>**

**| OBJECTID**

**| OBJECTID ASSIGN <expression>**

**| ...**

**2.5.4 Dispatch**

There are **three forms** of dispatch (i.e. method call) in Cool. The three forms differ only in how the called method is selected. The most commonly used form of dispatch is

**<expr>.<id>(<expr>,...,<expr>)**

Consider the dispatch e0.f(e1 , . . . , en). To evaluate this expression, the arguments are evaluated in left-to-right order, from e1 to en. Next, e0 is evaluated and its class C noted (if e0 is void a runtime error is generated). Finally, the method f in class C is invoked, with the value of e0 bound to self in the body of f and the actual arguments bound to the formals as usual. The value of the expression is the value returned by the method invocation.

**The other forms of dispatch are:**

**<id>(<expr>,...,<expr>)**

**<expr>@<type>.id(<expr>,...,<expr>)**

The first form is shorthand for self.<id>(<expr>,...,<expr>).

The second form provides a way of accessing methods of parent classes that have been hidden by redefinitions in child classes. Instead of using the class of the leftmost expression to determine the method, the method of the class explicitly specified is used. For example, e@B.f() invokes the method f in class B on the object that is the value of e. For this form of dispatch, the static type to the left of “@”must conform to the type specified to the right of “@”.

Then, we get the grammars:

**<dispatch> ::= <expression> DOT OBJECTID LPAREN <expression\_list> RPAREN**

**| OBJECTID LPAREN <expression\_list> RPAREN**

**| <expression> AT TYPEID DOT OBJECTID LPAREN <expression\_list> RPAREN**

**<expression\_list> ::= <expression> COMMA <expression\_list>**

**| <expression>**

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| …**

**2.5.5 Conditionals**

A conditional has the form

**if <expr> then <expr> else <expr> fi**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| ...**

**2.5.6 Loops**

A loop has the form

**while <expr> loop <expr> pool**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| …**

**2.5.7 Blocks**

A block has the form

**{ <expr>; ... <expr>; }**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| …**

**<expressions> ::= <expressions> <expression> SEMI**

**| <expression> SEMI**

**2.5.8 Let**

A let expression has the form

**let <id1> : <type1> [ <- <expr1> ], ..., <idn> : <typen> [ <- <exprn> ] in <expr>**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| …**

**<let\_expression> ::= LET <nest\_lets> IN <expression>**

**<nest\_lets> ::= <formal> <assign\_opt> COMMA <nest\_lets>**

**| <formal> <assign\_opt>**

**2.5.9 Case**

A case expression has the form

**case <expr0> of**

**<id1> : <type1> => <expr1>;**

**. . .**

**<idn> : <typen> => <exprn>;**

**esac**

The grammar is:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| <case\_expression>**

**<case\_expression> ::= CASE <expression> OF <cases> ESAC**

**<cases> ::= <case\_branch>**

**| <case\_branch> cases**

**<case\_branch> ::= <formal> DARROW <expression> SEMI**

**2.5.10 New**

A new expression has the form

**new <type>**

**2.5.11 Isvoid**

The expression

**isvoid expr**

evaluates to true if expr is void and evaluates to false if expr is not void.

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| <case\_expression>**

**| NEW TYPEID**

**| ISVOID <expression>**

**2.5.12 Arithmetic and Comparison Operations**

Cool has four binary arithmetic operations: +, -, \*, /. The syntax is

expr1 <op> expr2

Cool has three comparison operations: <, <=, =.

Finally, there is one arithmetic and one logical unary operator.

The expression ~<expr> is the integer complement of <expr>.

The expression not <expr> is the boolean complement of <expr>.

The grammars:

**<arith> ::= <expression> PLUS <expression>**

**| <expression> MINUS <expression>**

**| <expression> MULT <expression>**

**| <expression> DIV <expression>**

**| NEG <expression>**

**<comp>** ::= **<expression> LT <expression>**

**| <expression> LE <expression>**

**| <expression> EQ <expression>**

**| NOT <expression>**

Also, we can have a pair of parentheses outside a expression.

| …

| LPAREN <expression> RPAREN

Hence, we get **the complete grammars of Cool**:

<program> ::= <class\_list>  
<class\_list> ::= <class\_list> <class>  
| <class>  
<class> ::= CLASS TYPEID <inheritance> LBRACE <feature\_list\_opt> RBRACE SEMI  
<inheritance> ::= INHERITS TYPEID  
| <empty>  
  
<feature\_list\_opt> ::= <feature\_list>  
| <empty>  
<feature\_list> ::= <feature\_list> <feature>  
| <feature>  
  
<feature> ::= <attribute>  
| <method>  
  
<attribute> ::= OBJECTID COLON TYPEID assign\_opt SEMI  
<assign\_opt> ::= ASSIGN <expression>  
| <empty>  
  
<method> ::= OBJECTID LPAREN <formal\_list\_opt> RPAREN COLON TYPEID LBRACE <expression> RBRACE SEMI  
<formal\_list\_opt> ::= <formal\_list>  
| <empty>  
<formal\_list> ::= <formal\_list> , <formal>  
| <formal>  
<formal> ::= OBJECTID COLON TYPEID  
  
  
<expression> ::= <constant>  
| OBJECTID <assign\_opt>  
| <dispatch>  
| IF <expression> THEN <expression> ELSE <expression> FI  
| WHILE <expression> LOOP <expression> POOL  
| LBRACE <expressions> RBRACE  
| <let\_expression>  
| <case\_expression>  
| NEW TYPEID  
| ISVOID <expression>  
| <arith>  
| <comp>  
| LPAREN <expression> RPAREN  
  
  
<constant> ::= STR\_CONST  
| INT\_CONST  
| BOOL\_CONST  
  
<dispatch> ::= <expression> DOT OBJECTID LPAREN <expression\_list> RPAREN  
| OBJECTID LPAREN <expression\_list> RPAREN  
| <expression> AT TYPEID DOT OBJECTID LPAREN <expression\_list> RPAREN  
  
<expression\_list> ::= <expression> COMMA <expression\_list>  
| <expression>  
  
<expressions> ::= <expressions> <expression> SEMI  
| <expression> SEMI  
  
<let\_expression> ::= LET <nest\_lets> IN <expression>  
<nest\_lets> ::= <formal> <assign\_opt> COMMA <nest\_lets>   
| <formal> <assign\_opt>  
  
<case\_expression> ::= CASE <expression> OF <cases> ESAC  
<cases> ::= <case\_branch>  
| <case\_branch> cases  
<case\_branch> ::= <formal> DARROW <expression> SEMI  
  
<arith> ::= <expression> PLUS <expression>  
| <expression> MINUS <expression>  
| <expression> MULT <expression>  
| <expression> DIV <expression>  
| NEG <expression>  
  
<comp> ::= <expression> LT <expression>  
| <expression> LE <expression>  
| <expression> EQ <expression>  
| NOT <expression>

**3 The grammars Implementation in CUP**

The complete CUP specification for our cool parser (with actions embedded at various points in the grammar) is shown below:

/\*

\* cool.cup

\* Parser definition for the COOL language.

\*

\*/

import java\_cup.runtime.\*;

/\* Stuff enclosed in {: :} is copied verbatim to the class containing

all parser actions. All the extra variables/functions you want to use

in the semantic actions should go here. Don't remove or modify anything

that was there initially. \*/

action code {:

int curr\_lineno() {

return ((CoolTokenLexer)parser.getScanner()).curr\_lineno();

}

AbstractSymbol curr\_filename() {

return ((CoolTokenLexer)parser.getScanner()).curr\_filename();

}

let createLet(int line\_no, Features features, Expression e, int counter){

if(counter == features.getLength()-1){

return new let(curr\_lineno(), ((attr) features.getNth(counter)).name, ((attr) features.getNth(counter)).type\_decl,

((attr) features.getNth(counter)).init, e);

}else {

return new let(curr\_lineno(), ((attr) features.getNth(counter)).name, ((attr) features.getNth(counter)).type\_decl,

((attr) features.getNth(counter)).init, createLet(line\_no, features, e, ++counter));

}

}

:}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* DONT CHANGE ANYTHING IN THIS SECTION \*/

parser code {:

int omerrs = 0;

public void syntax\_error(Symbol cur\_token) {

int lineno = action\_obj.curr\_lineno();

String filename = action\_obj.curr\_filename().getString();

System.err.print("\"" + filename + "\", line " + lineno +

": parse error at or near ");

Utilities.printToken(cur\_token);

omerrs++;

if (omerrs>50) {

System.err.println("More than 50 errors");

System.exit(1);

}

}

public void unrecovered\_syntax\_error(Symbol cur\_token) {

}

:}

/\* Declare the terminals; a few have types for associated lexemes. The

token ERROR is never used in the parser; thus, it is a parse error when

the lexer returns it. \*/

terminal CLASS, ELSE, FI, IF, IN, INHERITS, LET, LET\_STMT, LOOP, POOL, THEN, WHILE;

terminal CASE, ESAC, OF, DARROW, NEW, ISVOID;

terminal ASSIGN, NOT, LE, ERROR;

terminal PLUS, DIV, MINUS, MULT, EQ, LT, DOT, NEG, COMMA, SEMI, COLON;

terminal LPAREN, RPAREN, AT, LBRACE, RBRACE;

terminal AbstractSymbol STR\_CONST, INT\_CONST;

terminal Boolean BOOL\_CONST;

terminal AbstractSymbol TYPEID, OBJECTID;

/\* DON'T CHANGE ANYTHING ABOVE THIS LINE, OR YOUR PARSER WONT WORK \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Complete the nonterminal list below, giving a type for the semantic

value of each non terminal. (See the CUP documentation for details. \*/

nonterminal programc program;

nonterminal Classes class\_list;

nonterminal class\_c class;

nonterminal Features feature\_list;

nonterminal Features feature\_list\_opt, let\_attr\_list;

nonterminal Expression expr;

nonterminal Expressions actuals, exp\_list, blk\_expr;

nonterminal Feature feature, let\_attr;

nonterminal Formals optional\_formal\_list;

nonterminal formalc formal;

nonterminal Cases case\_list;

nonterminal Case case;

/\* Precedence declarations go here. \*/

/\* Order is lowest precedence top and highest predence bottom \*/

precedence right ASSIGN;

precedence right NOT; /\*doc didn't specify, so guessing \*/

precedence nonassoc LE, LT, EQ;

precedence left PLUS, MINUS;

precedence left MULT, DIV;

precedence right ISVOID; /\*doc didn't specify, so guessing \*/

precedence right NEG; /\*doc didn't specify, so guessing \*/

precedence nonassoc AT; /\*doc didn't specify, so guessing \*/

precedence left DOT; /\*doc didn't specify, so guessing \*/

program

::= class\_list:cl

{: RESULT = new programc(curr\_lineno(), cl); :}

| error

{: //System.err.println("Reached program error production");

RESULT = new programc(curr\_lineno(),

new Classes(curr\_lineno())); :}

;

class\_list

/\* single class \*/

::= class:c

{: RESULT = (new Classes(curr\_lineno())).appendElement(c); :}

/\* several classes \*/

| class\_list:cl class:c

{: RESULT = cl.appendElement(c); :}

;

/\* If no parent is specified, the class inherits from the Object class \*/

class

::= CLASS TYPEID:n LBRACE feature\_list\_opt:f RBRACE SEMI

{: RESULT = new class\_c(curr\_lineno(), n,

AbstractTable.idtable.addString("Object"),

f, curr\_filename()); :}

| CLASS TYPEID:n INHERITS TYPEID:p LBRACE feature\_list\_opt:f RBRACE SEMI

{: RESULT = new class\_c(curr\_lineno(), n, p, f, curr\_filename()); :}

| error SEMI

{: //System.err.println("Reached program error production");

RESULT = new class\_c(curr\_lineno(), AbstractTable.idtable.addString("Object"),

AbstractTable.idtable.addString("Object"),

new Features(curr\_lineno()), curr\_filename()); :}

;

/\* Feature list may be empty, but no empty features in list. \*/

feature\_list\_opt

::= /\* empty \*/

{: RESULT = new Features(curr\_lineno()); :}

| feature\_list:f

{: RESULT = f; :}

;

feature\_list

/\* single feature \*/

::= feature:f

{:

RESULT = new Features(curr\_lineno()).appendElement(f);:}

/\* several features \*/

| feature\_list:fl feature:f

{:

RESULT = fl.appendElement(f); :}

;

/\* features \*/

feature

::= OBJECTID:o LPAREN optional\_formal\_list:ofl RPAREN COLON TYPEID:t LBRACE expr:e RBRACE SEMI

{: //System.err.println("Reached method production");

RESULT = new method(curr\_lineno(), o, ofl, t, e);:}

| OBJECTID:o COLON TYPEID:t ASSIGN expr:e SEMI

{: //System.err.println("Reached first attr feature production");

RESULT = new attr(curr\_lineno(), o, t, e);:}

| OBJECTID:o COLON TYPEID:t SEMI

{: //System.err.println("Reached second attr feature production");

RESULT = new attr(curr\_lineno(), o, t, new no\_expr(curr\_lineno()));:}

| error SEMI

{: //System.err.println("Reached program error production");

RESULT = new attr(curr\_lineno(), AbstractTable.idtable.addString("self"),

AbstractTable.idtable.addString("Object"),

new no\_expr(curr\_lineno())); :}

;

let\_attr

::= OBJECTID:o COLON TYPEID:t ASSIGN expr:e

{: //System.err.println("Reached first attr feature production");

RESULT = new attr(curr\_lineno(), o, t, e);:}

| OBJECTID:o COLON TYPEID:t

{: //System.err.println("Reached second attr feature production");

RESULT = new attr(curr\_lineno(), o, t, new no\_expr(curr\_lineno()));:}

| error

{: //System.err.println("Reached program error production");

RESULT = new attr(curr\_lineno(), AbstractTable.idtable.addString("self"),

AbstractTable.idtable.addString("Object"),

new no\_expr(curr\_lineno())); :}

;

/\* formal lists and formals \*/

optional\_formal\_list

/\* single formal \*/

::= formal:f

{: //System.err.println("Reached first formal list production");

RESULT = new Formals(curr\_lineno()).appendElement(f);:}

/\* several formals \*/

| optional\_formal\_list:ofl COMMA formal:f

{: //System.err.println("Reached second formal list production");

RESULT = ofl.appendElement(f);:}

/\* no formals \*/

| /\* empty \*/

{: RESULT = new Formals(curr\_lineno());:}

;

/\* formals \*/

formal

::= OBJECTID:o COLON TYPEID:t

{: //System.err.println("Reached formal production");

RESULT = new formalc(curr\_lineno(), o, t);:}

//| error COLON TYPEID:t | OBJECTID:o COLON error | error COLON error

// {: //System.err.println("Reached program error production");

// RESULT = new attr(curr\_lineno(), AbstractTable.idtable.addString("self"),

// AbstractTable.idtable.addString("Object"),

// new no\_expr(curr\_lineno())); :}

;

/\* Expressions \*/

expr

::= OBJECTID:o ASSIGN expr:e

{: RESULT = new assign(curr\_lineno(), o, e); :}

| expr:e DOT OBJECTID:o actuals:a

{: RESULT = new dispatch(curr\_lineno(), e, o, a); :}

| expr:e AT TYPEID:t DOT OBJECTID:o actuals:a

{: RESULT = new static\_dispatch(curr\_lineno(), e, t, o, a); :}

| IF expr:e1 THEN expr:e2 ELSE expr:e3 FI

{: RESULT = new cond(curr\_lineno(), e1, e2, e3);:}

| WHILE expr:e1 LOOP expr:e2 POOL

{: RESULT = new loop(curr\_lineno(), e1, e2);:}

| LBRACE blk\_expr: e RBRACE

{: RESULT = new block(curr\_lineno(), e); :}

| LET let\_attr\_list:l IN expr:e

{: RESULT = createLet(curr\_lineno(), l, e, 0); :}

| CASE expr:e OF case\_list:c ESAC

{: RESULT = new typcase(curr\_lineno(), e, c); :}

| NEW TYPEID:t

{: RESULT = new new\_(curr\_lineno(), t);:}

| ISVOID expr:e

{: RESULT = new isvoid(curr\_lineno(), e);:}

| expr:e1 PLUS expr:e2

{: RESULT = new plus(curr\_lineno(), e1, e2); :}

| expr:e1 MINUS expr:e2

{: RESULT = new sub(curr\_lineno(), e1, e2); :}

| expr:e1 MULT expr:e2

{: RESULT = new mul(curr\_lineno(), e1, e2); :}

| expr:e1 DIV expr:e2

{: RESULT = new divide(curr\_lineno(), e1, e2); :}

| NEG expr:e

{: RESULT = new neg(curr\_lineno(), e); :}

| expr:e1 LT expr:e2

{: RESULT = new lt(curr\_lineno(), e1, e2); :}

| expr:e1 EQ expr:e2

{: RESULT = new eq(curr\_lineno(), e1, e2); :}

| expr:e1 LE expr:e2

{: RESULT = new leq(curr\_lineno(), e1, e2); :}

| NOT expr:e

{: RESULT = new comp(curr\_lineno(), e); :}

| LPAREN expr:e RPAREN

{: RESULT = e; :}

| INT\_CONST:i

{: RESULT = new int\_const(curr\_lineno(), i); :}

| STR\_CONST:s

{: RESULT = new string\_const(curr\_lineno(), s); :}

| BOOL\_CONST:b

{: RESULT = new bool\_const(curr\_lineno(), b); :}

| OBJECTID:o

{: RESULT = new object(curr\_lineno(), o); :}

| OBJECTID:n actuals:a

{: RESULT = new dispatch(curr\_lineno(),

new object(curr\_lineno(),

AbstractTable.idtable.addString("self")),

n, a); :}

;

/\* Optional Helper Expressions for expressions\*/

/\* let subexpression \*/

let\_attr\_list

::= let\_attr:f

{: //System.err.println("Reached first feature list production");

RESULT = new Features(curr\_lineno()).appendElement(f);:}

/\* several features \*/

| let\_attr\_list:ltl COMMA let\_attr:f

{: //System.err.println("Reached second feature list production");

RESULT = ltl.appendElement(f); :}

;

/\* list of one or more expressions for block \*/

blk\_expr

::= expr:e SEMI

{: RESULT = new Expressions(curr\_lineno()).appendElement(e); :}

| blk\_expr:b expr:e SEMI

{: RESULT = b.appendElement(e); :}

| error SEMI

{: //System.err.println("Reached program error production");

RESULT = new Expressions(curr\_lineno()); :}

;

/\* Cases \*/

case\_list

/\* One Case \*/

::= case:c

{: RESULT = new Cases(curr\_lineno()).appendElement(c); :}

| case\_list:c1 case:c2

{: RESULT = c1.appendElement(c2); :}

;

/\* Case Non-Terminal \*/

case

::= OBJECTID:o COLON TYPEID:t DARROW expr:e SEMI

{: RESULT = new branch(curr\_lineno(), o, t, e); :}

;

/\* Method arguments \*/

actuals

/\* No arguments \*/

::= LPAREN RPAREN

{: RESULT = new Expressions(curr\_lineno()); :}

/\* List of arguments \*/

| LPAREN exp\_list:el RPAREN

{: RESULT = el; :}

;

/\* Non-empty list of expressions \*/

exp\_list

/\* One expression \*/

::= expr:e

{: RESULT = (new Expressions(curr\_lineno())).appendElement(e); :}

/\* Several expressions \*/

| exp\_list:el COMMA expr:e

{: RESULT = el.appendElement(e); :}

;

**4 Test Cases**

We have totally 70 test programs. For convenience, we just show some of them in the document and give others in a zip file.

**arithprecedence.test**

class A {

f():Int{{

a+b-c;

a-b+c;

a+b\*c;

a\*b+c;

a+b/c;

a/b+c;

a-b\*c;

a\*b-c;

a-b/c;

a/b-c;

a\*b/c;

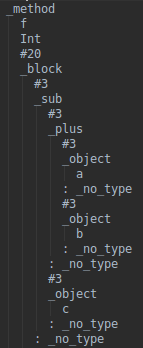
a/b\*c;

}};

};

The output:

we choose the first sentence: a+b-c



Let’s draw a tree according the output:

assignment.test

class Test {

foo:Int;

bar():Object{foo <- 3 };

};

assignment.test.out

#4

\_program

#4

\_class

Test

Object

"assignment.test"

(

#2

\_attr

foo

Int

#2

\_no\_expr

: \_no\_type

#3

\_method

bar

Object

#3

\_assign

foo

#3

\_int

3

: \_no\_type

: \_no\_type

)

**classnoname.test**

class {

foo:Int;

};

**classnoname.test.out**

"classnoname.test", line 1: syntax error at or near '{'

Compilation halted due to lex and parse errors

**staticdispatchnoargs.test**

class Test {

foo:Int;

bar():Object{self@Test.bar()};

};

**staticdispatchnoargs.test**

#4

\_program

#4

\_class

Test

Object

"staticdispatchnoargs.test"

(

#2

\_attr

foo

Int

#2

\_no\_expr

: \_no\_type

#3

\_method

bar

Object

#3

\_static\_dispatch

#3

\_object

self

: \_no\_type

Test

bar

(

)

: \_no\_type

)

**5 Problems During Experimenting**

**1.Forget that in a class definition, its features can be empty( no variables and methods).**

**Old one:**

<feature\_list> ::= <feature\_list> <feature>

| feature

**New one:**

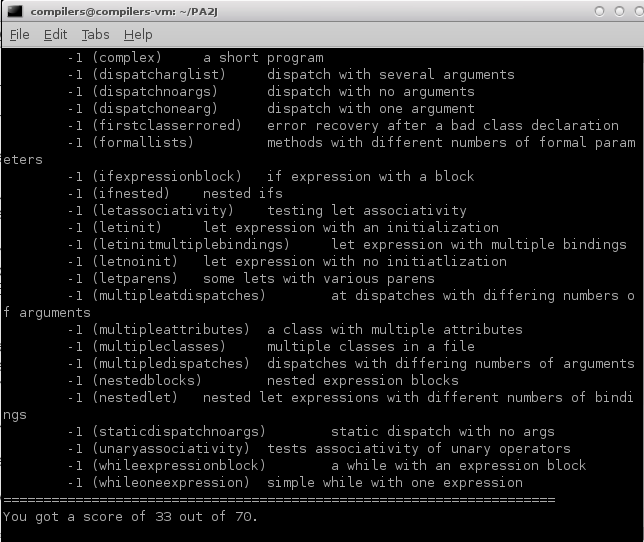
<feature\_list\_opt> ::= <feature\_list>

| <empty>

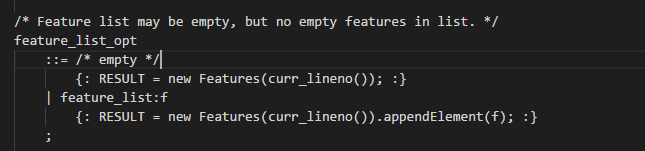
<feature\_list> ::= <feature\_list feature>

| feature

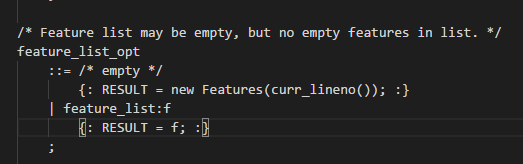
**2. The first time I tested my parser at which I thought I completed it, I got a lot of errors.**



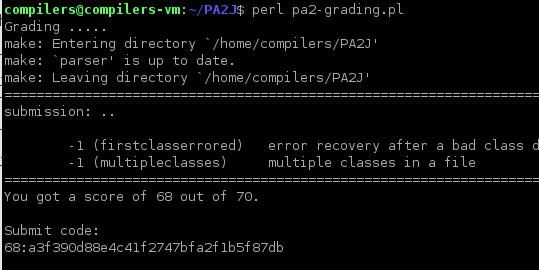
Then I find the problem is in the feature\_list\_opt grammar sepecification:



Noticed that the RESULT should be f itself rather than an Feature element.



**3 Then the test output shows below:**



This is the test programs:

**firstclasserrored.test**

class Foo inherits asdfjkasldfjdklaf;

class Bar {a:Int;};

class Baz {b():Int{B};};

**Multipleclasses.test**

class Test {

foo:Test;

bar():Int {5};

};

class Test extends Object {

foo:Test;

};

class Test {

foo:Test;

bar():Int {5};

};

class Test implements Nothing{

foo:Test;

bar():Int {5};

};

There were two test cases failed. From the test output, we can know that our parser lack the ability of error recovery. Therefore, we have to add some additional actions for handling errors.

