**Lex, Yacc版本**

Lex版本: flex 2.6.4

Yacc版本: bison (GNU Bison) 3.8.2

**作業平台**

Windows Subsysten for Linux 2(WSL2)。Version: 2.4.13.0

**執行方式**

Step1. 到存放B113040047.y及B113040047.l與它們的makefile的資料夾底下，執行: make

Step2. ./calc < [測試檔名.java]

**你如何處理這份規格書上的問題**

以”lines”作為起始的語法，並能用lines遞迴導出多個”class\_declaration”，藉此開始Java的語法(Java程式是以class為基礎組成)。

**2.1 Data types and declarations**

**變數宣告:**

分成四種狀況(以下四種狀況，都能以逗號將ID隔開，進行多個變數的宣告):

1. 靜態變數宣告(static): *static <變數型態> <ID>;*
2. 一般變數宣告: *<變數型態> <ID>;*
3. 陣列宣告: *<變數型態> [] <ID> = new [<int number>];*
4. 常數宣告: *final <變數型態> <ID> = <start value>*

**Classes and Objects**

**class宣告:**

*<class modifier> class <ID> <super> <interface> <classbody>*

其中，<super>和<interface>用於class的繼承或是interface的實作，可以選擇忽略。其語法如下所示:

*<super>: /\*empty\*/*

*| extends <ID>*

*<interface>: /\*empty\*/  
 | implement <ID>*

而class modifier則為:

*<class modifier>: /\*empty\*/*

*| public | abstract | final*

**object宣告:**

分成有無提供起始值兩種宣告方式

1. *<ID(for class name)> <ID(for object name)>*
2. *<ID(for class name)> <ID(for object name)> = new <ID(for class name)> ();*

**Fields宣告:**

分成class fields及instance fields

1. class fields: *static <變數型態> <ID>;*
2. instance fields:

再細分為三種狀況:

1. 普通宣告: *<變數型態> <ID*>;
2. 常數宣告: *final <變數型態> <ID> = <start value>;*
3. 陣列宣告: *<變數型態> [] <ID> = new [<int number>];*

**2.2 Methods**

依照有無提供method\_modifier以及回傳值型態分成三種狀況

1. <method\_modifier> <變數型態> <ID> (<argument>) <compound\_statement>

2. <變數型態> <ID> (<argument>) <compound\_statement>

3. <ID> (<argument>) <compound\_statement>

其中method\_modifier包含:

<method\_modifier>: public | protected | private

**2.3 Statements**

以<statement>包含以下四種statements(其他statement為避免conflict，而移置lower level)

<statement>: <compound statement> | <simple statement> |

<loop\_statement> | <condition statement>

**compound statement**

為了處理變數宣告及statement交替出現的情況，採用兩層的遞迴結構:

***<****compound statement>: {<compound\_statement body>}*

*<compound\_statement body>: <variable\_declaration> <compound\_statement\_body>*

*| <statement> <compound\_statement\_body>*

*| <object\_declaration> <compound\_statement\_body>*

*| <return\_statement>*

*| <class\_declaration> <compound\_statement\_body>*

**simple statement**

分成以下狀況:

* 1. <name> = <expression>;
  2. print(expression);
  3. read(expression);
  4. expression;
  5. ;

為避免conflict，將name++以及name--移至lower level

其中，name的定義如下:

<name>: ID\_TOK

| field\_or\_array\_item

**expression**

針對加減法、乘除法以及部分運算的優先度進行分層，因此呈現以下遞迴結構:

<expression>: <term>

| <expression> + <term>

| <expression> - <term>

<term>: <factor>

| <term> \* <factor>

| <term> / <factor>

<factor>: <ID>

| integer(純數字)

| float(純浮點數)

| string(純字串)

| true

| false

| (<expression >)

| <prefixOP> <name>

| <name> <postfixOP>

| interger(純數字) <postfixOP>

| <method\_invocation>

| <ID>.<ID>

| <array\_item>

其中，prefixOP, postfixOP, method\_invocation, array\_item的定義如下

prefixOP: ++ | -- | + | -

postfixOP: ++ | --

method\_invocation: <name> (<argument>)

array\_item: <ID>[integer(純數字)]

| <ID> [ <ID>]

**conditional statement**

針對有無使用else，分成兩種狀況

1. IF LPAREN <boolean\_expr> RPAREN <if\_else\_loop\_statement>
2. IF LPAREN <boolean\_expr> RPAREN <if\_else\_loop\_statement> <else>

而如果使用了else，會執行此文法，此文法包含了if…else的狀況以及if…else if…else的狀況:

<else>: else <if\_else\_loop\_statement>

| else <conditional\_statemen>

另外，boolean\_expr及if\_else\_loop\_statement與其中再使用到的infixOP的定義如下:

<boolean\_expr>: <expression> <infixOP> <expression>

<if\_else\_loop\_statement>: <simple\_statement>

| <compound\_statement>

<infixOP>: == | != | < | > | <= | >=

**loop statement**

分為for loop及while loop:

1. for(<forInitOpt>;<boolean\_expr>;<forUpdataOpt>) <if\_else\_loop\_statement>
2. while (<boolean\_expr>) <if\_else\_loop\_statement>

另外，forInitOpt及forUpdataOpt與其中為了讓for loop之中的宣告能宣告多個變數，而使用到的init\_item及init\_list的定義如下

<forInitOpt>: /\* empty \*/

| <init\_list>

<init\_item>: <name> = <expression>

| int <ID> = <expression> /\* 供 int i=0 這種 \*/

<init\_list>: <init\_item>

| <init\_list> , <init\_item>

<forUpdateOpt>: <name> ++

| <name> --

**return statement**

定義如下:

*return <expression> ;*

1. **Semantic definition**

藉由將變數儲存至ID table並配合stack的操作，來做到變數重複宣告的檢查

做法:

* 每當遇到”{“，就將目前儲存的ID數量存進stack中
* 當宣告變數時，就到ID table中，查找: 從最新放入table的變數，到stack的top所指向的ID，這之間的所有變數名稱是否與新宣告的變數名稱相同，若相同，則出現變數重複宣告的提示，若不相同，則放入ID tabla中
* 每當遇到”}”，就將: 從stack的top所指向的ID開始，把ID刪除，刪到最新的ID為止

此作法確保檢查時，只會檢查到指定scope的ID，並且已經結束他的生命週期的ID會被從ID table中刪除

1. **Error and recovery**

針對以下幾種狀況作出error recovery

(1) 檢查宣告的object type和賦予它的object type是否相同

e.g.

(O): Point lowerLeft = new Point();

(X): Point lowerLeft = new point();

1. 沒加分號

若是在需要放分號的地方沒有放分號而是其他的token，則會提示錯誤，而因為這時已經取得到其他文法的某個token，所以parser會一直往後取得token，直到碰到下一個分號，讓沒有放分號的語句的下一行失效。這個做法使仿造C compiler中，如果沒有加上分號，會在下一句提示錯誤

1. 重複變數宣告

如同semantic definition中所說到的，以stack以及ID table來做檢查

1. 變數宣告格式錯誤(例如多個變數名稱之間沒有加逗號)

在進行變數宣告時，若ID的後方不是接上賦予的值或是放上逗號來接續下一個變數名稱的宣告，則會提示invalid variable declaration。

e.g. int a b;

-> ID後方接上ID，不符合文法，提示invalid variable declaration

1. else without if

如果檢查到一個else statement出現在並非if statement後方的位置，則會提示錯誤: else without if

1. boolean expression錯誤

在需要放上boolean expression的文法中(例如conditional statement以及loop statement)，如果檢測到右括號，也就是boolean expression結束之處，如果前面所放的不是boolean expression，則會提示錯誤: invalid Boolean expression

**做這個作業所遇到的問題**

1. if和else優先級的差異

以前述的conditional statement的定義來實作時，一直遇到conflict，因為parser遇到conditional statement，他會有多種選擇，不知道該選擇哪條文法。這個問題後來在宣告區放上:

%nonassoc LOWER\_THAN\_ELSE

%nonassoc ELSE

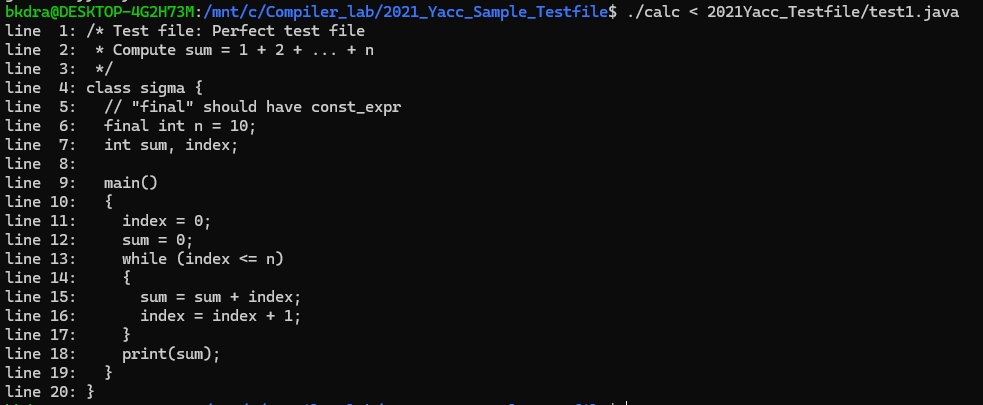
並在conditional statement的文法IF LPAREN <boolean\_expr> RPAREN <if\_else\_loop\_statement> 後方放上:%prec LOWER\_THAN\_ELSE後得到解決

1. 重複變數宣告的檢驗

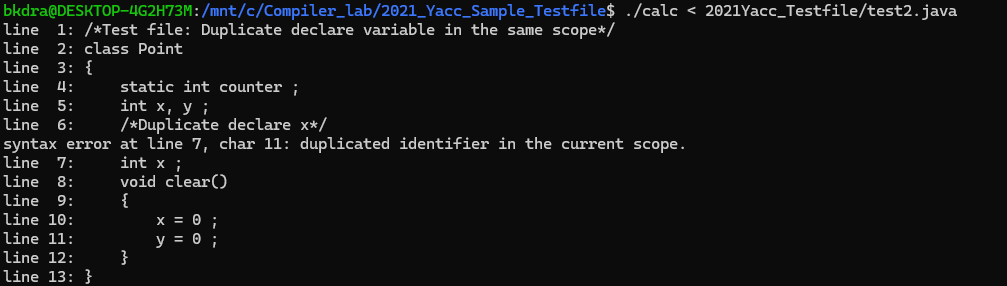
起初是針對method名稱以及class名稱等狀況來劃分變數名稱是否重複使用，但沒有考慮到在scope外層使用到的變數名稱，在scope內層仍可再使用。為了解決此問題，才改為目前的處理方式。

**所有測試檔執行出來的結果**

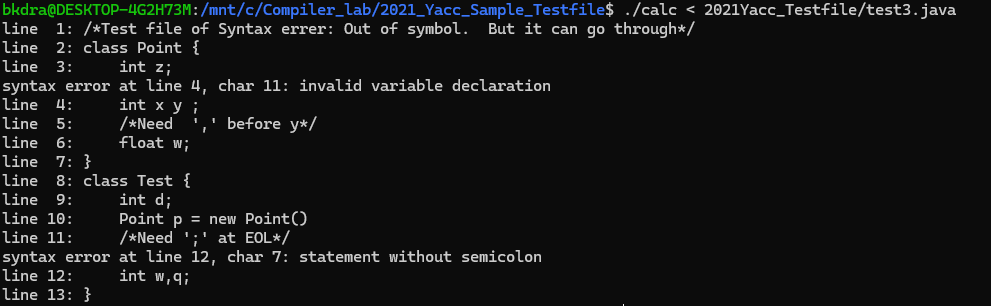
test1.java:



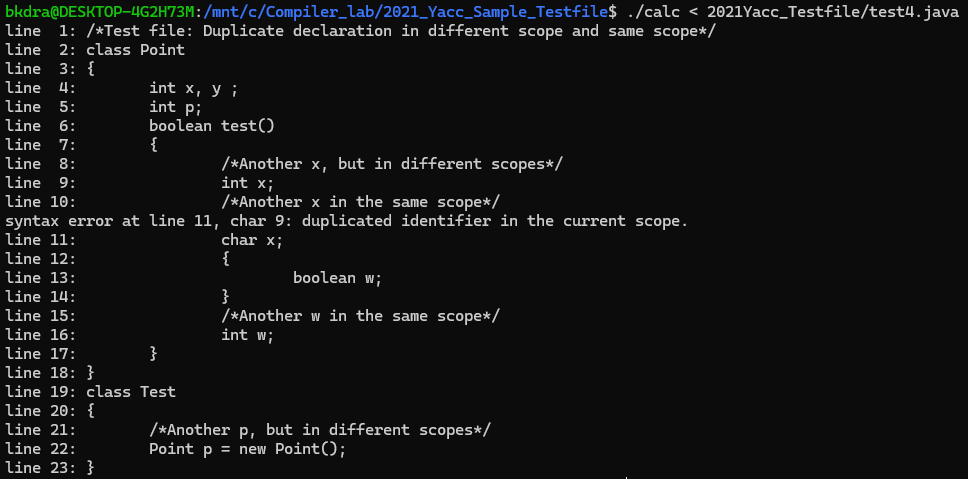
test2.java:



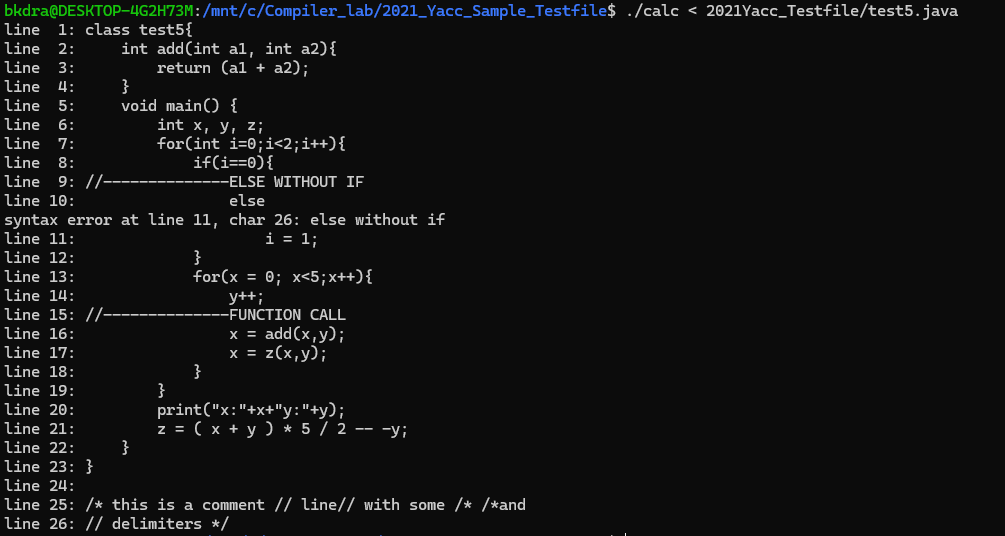
test3.java:



test4.java:



test5.java



test6.java:

