A file describing what changes I have to make to my scanner and parser since the previous version

xxx.l

xxx.y

（支援的額外功能：**int array** 的宣告及使用、**read** 的使用）

四資工三甲　B10615031　許晉捷

---

1. 在 scanner 中，新增 keyword token “length”：

```
220     /* TOKEN: "length" -> keywords */
221 length { bufText(yytext); token(LENGTH); }
```

^ 用於取得某個陣列的長度。

2. 在 parser 中，新增一系列用於產生 java assembly code 的函數：

```
// generate the java assembly code w/ a plain string
void genJavaa(const char*);

// generate the java assembly code w/ a formatted string ('f' stands for 'formatted')
void fgenJavaa(const char*, const char*, const char*, const char*, const char*, const char*);

// generate the java assembly code w/ an integer value ('i' stands for 'integer')
void igenJavaa(const char*, const int, const char*);

// generate the java assembly code for a relational operation ('r' stands for 'relational')
void rgenJavaa(const char*);

// generate the java assembly code for a read-clause ('u' stands for 'user' input)
void ugenJavaa(const char*);

// buffer the java assembly code w/ a plain string
void bufJavaa(const char*);

// buffer the java assembly code w/ a formatted string ('f' stands for 'formatted')
void fbufJavaa(const char*, const char*, const char*, const char*, const char*, const char*);

// buffer the java assembly code w/ an integer value ('i' stands for 'integer')
void ibufJavaa(const char*, const int, const char*);

// flush the buffered java assembly code
void flushJavaa();
```

^ genJavaa: 傳入什麼就產生什麼。
fgenJavaa: 格式化產生。
igenJavaa: 字串 + 整數 + 字串。
rgenJavaa: 產生 relational operation 的 java assembly code。
ugenJavaa: 產生 read statement 的 java assembly code。
bufJavaa: 將傳入字串 buffer 起來。
fbufJavaa: 格式化 buffer。
ibufJavaa: 字串 + 整數 + 字串（buffer）。
flushJavaa: 將所有 buffered 字串產生出來，並清空 buffer。

3. 在 parser 中所有有需要的 actions 中，加入上述的其中一種或多種 **xxxJavaa 函數**；例子請看以下第 5 條。

4. 在 parser 中，新增一些全域變數，用於輔助產生正確的 java assembly code：

```
// the counter for localizing the local variables to jvm
int localCounter = 0;

// in global or local currently
int inGlobal = 1;

// in a global's variable declaration or not
int inGlobalVarDclr = 0;

// the counter of labels
int labelCounter = 0;
```

^ localCounter: 輔助產生正確的 local 編號（每切換一個 method 就會重新計數）。
inGlobal: 判斷現在（parsing 中）是否處於全域的 scope 中。
inGlobalVarDclr: 判斷現在（parsing 中）是否處於全域變數的宣告句中。
labelCounter: 用於產生一般 labels 的計數器。

5. 在 parser 中，新增「取得陣列長度」的規則：

```
697 /* the length of an array */
698     | identifier PERIOD LENGTH {
699         Item* aItem = lookupInHashTable(ident);
700         if (aItem != NULL) {
701             igenJavaa("aload ", aItem->localNum, "\n");
702             genJavaa("arraylength\n");
703         }
704     }
705     ;
```

^ 規則：expr -> identifier ‘.’ “length”　　// identifier -> ID
例子：var arr: int[5]　　// 宣告一個長度為 5 的 int 陣列
println arr.length　　// 印出它的長度（會印出 5）

6. 在 symbol_table.hpp 中的 struct item 中新增以下三個成員：

```
24  // an item (data) of a hash-table node
25 typedef struct item {
26     // for checking if this item is a global item or not
27     int isGlobalFlag;
28
29     // the number for being stored in local variables in jvm
30     int localNum;
31
32     // the value for a constant item
33     int constantVal;
```

^ isGlobalFlag: 判斷這個 item 是否為全域變數/常數/陣列。
localNum: 這個 item 的 local 編號（僅當此 item 為**區域**變數/陣列時才有用）。
constantVal: 這個 item 的常數值（僅當此 item 為常數時才有用）。

7. 在 parser 中的 main yyparse()的前後分別操作「用於輸出.jasm 檔」的檔案指標：

```
823 // the main function
824 int main() {
825     // open the file pointer for outputting the java assembly code
826     fout = fopen("xxx.jasm", "w");
827
828     // start parsing
829     yyparse();
830
831     // close the file pointer
832     fclose(fout);
833
834     return 0;
835 }
```

^ yyparse()前：以 write 模式打開固定檔名為 **xxx.jasm** 的檔案指標。
yyparse()後：關閉該檔案指標。
實際的檔案輸出操作全部都會寫在上述的 **xxxJavaa** 系列函數中。