FDMJ2023 Grammar

The grammar

This is the mini programming language we are going to compile in our class. We aim to generate a compiler to translate any FDMJ2023 program into LLVM IR, as well as RPi Assembly.

```
Program -> MainMethod ClassDecl*

MainMethod -> public int main '(' ')' '{' VarDecl* Statement* '}'

ClassDecl -> public class id [extends id] '{' VarDecl* MethodDecl* '}'
     //[] means optional

VarDecl -> class id id ';' | int id ';' | int id '=' IntConst ';' |
   int '[' ']' id ';' | int '[' ']' id '=' '{' IntConstList '}' ';'

IntConst -> INT_CONST | '-' INT_CONST
IntConstList -> IntConst IntConstRest* | \empty
IntConstRest -> ',' IntConst

MethodDecl -> public Type id '(' FormalList ')' '{' VarDecl* Statement* '}'

FormalList -> Type id FormalRest* | \empty
FormalRest -> ',' Type id

Type -> class id | int | int '[' ']'

Statement ->
   '{' Statement* '}' |
   if '(' Exp ')' Statement else Statement |
   if '(' Exp ')' Statement |
   while '(' Exp ')' Statement |
   while '(' Exp ')' ';' |
   Exp = Exp ';' |
   ~~Exp '[' Exp ']' = Exp ';' | //the first Exp must be array /* Redundant */~~
   Exp '['  ']' = '{' ExpList '}' ';' |
        /* the first Exp must be array: get new array location,
           then assign values*/
   Exp '.' id '(' ExpList ')' ';' | //ignore the return value
   continue ';' | break ';' |
   return Exp ';' |
   putint '(' Exp ')' ';' | putch '(' Exp ')' ';' |
```

```
37      putarray '(' Exp ',' Exp ')' ';' |
38      starttime '(' ')' ';' | stoptime '(' ')' ';'
39
40  Exp -> Exp op Exp |
41      Exp '[' Exp ']' |
42      Exp '.' id '(' ExpList ')' |
43           //to call a class method, Exp must evaluate to an object
44      Exp '.' id | //to access a class varialbe
45                   //Exp must evaluate to an object
46      Exp '.' id '[' Exp ']' | //to access a class array /* Redundant */
47      INT_CONST |
48      true | false | length '(' Exp ')' |
49      id | this | new int '[' Exp ']' | new id '(' ')' |
50      '!' Exp | '-' Exp | '(' Exp ')' |
51      '(' '{' Statement* '}' Exp ')' |   //escape expression
52      getint '(' ')' | getch '(' ')' | getarray '(' Exp ')'
53
54  ExpList -> Exp ExpRest* | \empty
55  ExpRest -> ',' Exp
```

Notes:

The semantics of an FDMJ2023 program with the above grammar is similar to that for programming languages of C and Java. Here we give a few notes about it, and we will have more discussions during the semester.

- **Comments** may be included in an FDMJ2023 program in two ways:
  - All characters after "//" are treated as comments up to a newline.
  - All characters (including newline) after "/*" are treated as comments until "*/" is encountered.
  - Comments are not treated as part of the program.
- The root of the grammar is `Program`.
- The binary operations (`op`) are `+`, `-`, `*`, `/`, `||`, `&&`, `<`, `<=`, `>`, `>=`, `==`, `!=`.
- We use integer to "simulate" boolean values. When doing boolean operations (`||`, `&&`, `!`), any integer not equal to 0 is taken as true, and false otherwise. The result of a boolean operation and comparison operation is either integer `1` for true or integer `0` for false (hence, for example, `100 && 2` gives integer `1`, `1>2` gives integer `0`, and `!2` gives `0`.). The literals true and false are taken as integer `1` and `0`, respectively.
- `INT_CONST` is `[0-9]+`.

- `id` is any string consisting of `[a-z]`, `[A-Z]`, `[0-9]` and `_` (the underscore) of any length, with the restriction that it cannot be any of the keywords (terminal strings marked red) used in the above grammar, and it must start with a `[a-z]` or `[A-Z]`. The lower or upper case letters in an id are significant (e.g., `aB` and `ab` are two different ids).
- All arrays are in the heap memory. new returns pointer to the heap memory. The statement `id[]={exp1, ...,expn}` is to initialize a new array of size `n` in the heap memory.
- All the statements are executed from left to right, including the ones in the escape expression, and only impact the state after the point of the code. For example, if the initial value of `a` is `0`, then `a+2*({a=1; b=2} a+b)` gives `6`, but `2*({a=1; b=2} a+b)+a` gives `7`. Another example is: assume `id` is a class object which has a class variable `x` (with initial value `0`) and method `f` (which increments the value of `x` by 1 and returns `0`), then `id.x+2*id.f()` gives 0, while `2*id.f()+id.x` gives `1`.
- Boolean binary operations (`||` and `&&`) follow the "shortcut" semantics. For example, in `(true || ({a=1},false))`, `a=1` is not executed.

FDMJ2023-SLP：

```
1 Program(root) -> MainMethod
2 MainMethod -> public int main '(' ')' '{' Statement* '}'
3 Statement -> id = Exp ';' | putint '(' Exp ')' ';' | putch '(' Exp ')' ';'
4 Exp -> Exp op Exp | INT_CONST | id | '-' Exp | '(' Exp ')' |
5         '(' '{' Statement* '}' Exp ')'
```

Notes:

This is a proper subset of FDMJ2023. It's supposed to use for the first "compiler" of the class. We assume the allowed `op` are only `+`, `-`, `x`, `/`.