**FDMJ2024 Specification**

**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 FDMJ2024 program into LLVM IR, as well as RPi Assembly.

|  |
| --- |
| Program -> MainMethod ClassDecl\* //\* means 0 or more   MainMethod -> public int main '(' ')' '{' VarDecl\* Statement\* '}'    ClassDecl -> class id [extends id] '{' VarDecl\* MethodDecl\* '}'  //[] means optional    Type -> class id | int | int '[' ']' | float | float '[' ']'  VarDecl -> Type id ';' ｜ Type id '=' CONST ';' |   Type id '=' '{' ConstList '}' ';'     ConstList -> CONST ConstRest\* | \empty  ConstRest -> ',' CONST    MethodDecl -> public Type id '(' FormalList ')' '{' VarDecl\* Statement\* '}'    FormalList -> Type id FormalRest\* | \empty  FormalRest -> ',' Type id  Statement ->  '{' Statement\* '}' |  if '(' Exp ')' Statement else Statement |   if '(' Exp ')' Statement |  while '(' Exp ')' Statement |   while '(' Exp ')' ';' |  id = Exp ';' |  id '[' Exp ']' = Exp ';' |  id '[' ']' = '{' ExpList '}' ';' | /\* new id[], then assign values\*/  Exp '.' id '(' ExpList ')' ';' | //ignore the return value  continue ';' | break ';' |  return Exp ';' |  putint '(' Exp ')' ';' | putch '(' Exp ')' ';' |  putint '(' Exp ')' ';' | putch '(' Exp ')' ';' |  putarray '(' Exp ',' Exp ')' ';' |  starttime '(' ')' ';' | stoptime '(' ')' ';'    Exp -> Exp op Exp |  Exp '[' Exp ']' |  Exp '.' id '(' ExpList ')' | //to call a class method  Exp '.' id | //to access a class varialbe  CONST |  true | false |  id | this | new int '[' Exp ']' | new float '[' Exp ']' |  new id '(' ')' | '!' Exp | '-' Exp | '(' Exp ')' |  '(' '{' Statement\* '}' Exp ')' | //escape expression  getint '(' ')' | getch '(' ')' | getarray '(' Exp ')'    ExpList -> Exp ExpRest\* | \empty  ExpRest -> ',' Exp |

**Notes:**

The semantics of an FDMJ2024 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.

* The root of the grammar is Program.
* The binary operations (op) are +, -, \*, /, ||, &&, <, <=, >, >=, ==, !=.
* CONST is either an integer (+|-)[0-9]+ or a float number (+|-)[0-9]\*.[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.
* All variables in a class are taken as “public” variables (as defined in Java).
* A subclass inherits all the methods and variables from its superclass (if any, recursively). To simplify, we assume that a subclass may override methods declared in its ancestor classes, but must have the same signature (i.e., the same list of types, albeit may use different id’s). And a variable declared in a subclass cannot use the same id as any variable declared in any of its ancestor classes.