Compilers Project

MiniJava

Preface

MiniJava is a subset of Java. The meaning of a MiniJava program is given by its meaning as a Java program, Overloading is not allowed in MiniJava. The MiniJava statement System.out.println(...); can only print integers. The MiniJava expression e.length only applies to expressions of type int [].
So we have to build a compiler witch can understand MiniJava programs.
Compiler building have to be divided into sequential steps :

- Lexical Analysis.
- Parsing.
- Building abstract syntax tree (AST).
- Symantec Analysis.
- Code generation.

In this time we are going to do first three steps, and off course we are going to use some tools to help us in our work (Javacc, Jjtree) .
The first one "Javacc" or "Java compiler compiler ", Helps me to build my parser easily.
"Javacc" is LL(k) parser.
The second one "Jjtree" Helps me to build my abstract syntax tree (AST)

Tokens Definitions

Our problem starts with Tokens Detention.
Tokens are the terminals of our language, we categorized them into sets to let our program more flexible : Skips ,Comments ,Reserved words ,Operations ,Literals ,Identifiers ,Separators
more about this ? see Appendix A.

Language grammars

The grammars witch are given in the assignment has a lot of problems ☺, so we are going to redefine them, to see language grammars see Appendix B.

Ambiguity

The job of a parser is to read an input stream and determine whether or not the input stream conforms to the grammar.
This determination in its most general form can be quite time consuming.
What if there can be multiple equivalent grammars for the same input language? That's what we called it ambiguity.

There are two ways in which you make the choice decisions work properly :

- Modify the grammar to make it simpler.
- Insert hints at the more complicated choice points to help the parser make the right choices. "LOOKAHEAD".

And we are going to use them both to convert our language to the stander model.
Let's start now by defining the ambiguous grammars :

- ClassDecl() :
  LOOKAHEAD(DerivedClass()) DerivedClass() | Class()
- MethodBody() :
  LOOKAHEAD(VarDecl()) VarDecl() | Statment()
- Type() :
  boolean
  | id
  | **LOOKAHEAD**(int[]) int[]
  | int
- Statement():
  LOOKAHEAD(If (Exp) Statement else Statement) If (Exp) Statement else Statement
  | If (Exp) Statement
  | while (Exp) Statment
  | Print(Exp)
  | (**LOOKAHEAD**(2) Object.)? Id ([Exp]) = Exp ;
  | { (Statment)* }
- NewObject()
  new (int[Exp()] | id())
- IdentifierExp()
  LOOKAHEAD(Identifier() <DOT> Length()) Identifier() <DOT> Length()
  |
  [**LOOKAHEAD**(Object() <DOT>) Object() <DOT>] Identifier() [
  <LBRACKET> Exp() <RBRACKET>
  | <DOT> Length()
  | <LPAREN> ExpList() <RPAREN>]

Building the Tree

We are now ready to build the tree, and all we have to do is to modify the start grammar to return the root of our AST ,

```
SimpleNode Start() : {}
{
    Program()
    { return jjtThis; }
}
```

Before we start building the tree we have to know that jjtree is a tool that can build tree structure from our grammars, each grammar in our language well be a node in target tree, so we have to rewrite the grammars to be more flexible in the tree.

to see the full jjtree grammars go to Appendix C.

And now we can compile the file normally in some steps :

- Build the AST "Jjtree command" witch well gives me some classes to define our tree and a "jj" file witch contain the parser.
- Compiling the parser ,"javacc command", witch well gives me set of java source files.
- Generating Executable instance from the parser .

Finaly we are now ready to parse any MiniJava source file and build it's AST.

Tests

Let the test file be as the follow

```
class d{
      public static void main(String[] a){
      y.x = 10;
      //somtext
      int a = 1;
      int h = 15;
      int c = (true);
      boolean t = c+h;
      }
}

class hj extends s{
    int r =(1+true);
    public int d(int y,boolean b){
        int y = new hj();
        System.out.println(y);
        while(i<100){
        if(i/2==1){
            System.out.println(y);
        }else{
            System.out.println(u);
            }
        }
       return 1;
    }
    int u = this.x();
    int[] g = new rr();
}
```

The result of compiling this file is a huge tree , and that's why we didn't paste it here ☺

Semantic Analyze

In this stage we are aiming to check the validity of the source code , but before all let us answer this question "what it means that some code is valid or not ?"
After a lot of studies, analyzes and scientific researches !! we find this answer the code is semantically valid if it didn't have any of those :

- Declaring a variable of unknown type
- Using undeclared identifier
- Redeclaring a method or variable
- Calling non existing method
- Calling a method with invalid arguments
- Cyclic inheritance
- Type mismatching

And now we will see how the above constraints can be achieved at first we need a structure to store the data about the classes of the program so we suggest the ClassList structure witch store the following data of the every class

- Name
- Super class name
- Data members (name , type)
- Methods (name , argument list , return type)

And we fill this structure with data using a visitor called classDeclVisitor witch traverse over the AST and fill the class list with checking the following constraints :

- Any data member name couldn't be used twice
- Any method name couldn't be used twice (No overloading)
- The data members types must be known or defined
- The method return types must be known or defined
- Cyclic inheritance is banned
- Overriding is not allowed

Type checking

After filling the class list the type checking begins to implement the type checking we will use an assistant structure it is the Symbol Table and we will use it for storing the data members and local variables names and values of each instance of any class and for the main method after knowing the tasks of the Symbol Table we will now see the internal system of this structure

Symbol Table

The symbol table is just a combination of a Hash table & Stack , the hash table is used for storing the data and the stack is used for observing the scopes .
Every cell of the hash table is a list of symbols where the symbol represent a variable and holding the following info :

- Name
- Type
- Value
- Sub symbol table if the symbol is an instance of some class

The stack is used for storing the following data

- Variables names
- Flags of the scopes beginnings

We will use the symbol table like next steps

- Insert symbol
  - Creating symbol with the required data
  - Add the symbol to the hash table
  - Push the symbol name to the stack
- Lookup for symbol
  - Return the first occurrence of the symbol name
- Starting a scope
  - Push the starting scope flag to the stack
- Ending a scope
  - Pop all symbol until finding a starting scope flag
  - Remove symbols witch there names popped from stack

Now we will traverse over the AST and fill the symbol table with data members and local variables and simultaneously do the type checking we fill the Symbol Table & check types at same time to avoid cases like this

```
int  a = 3;
System.out.println( a * b );
int b = 0;
```

The type checking is done by visiting every node of the AST and match the types of the node sons

example : visiting the node ASTIf
this node has two sons
1. ASTExp
2. ASTStatement

Checking this node is by checking the ASTExp type if it is Boolean then
this node is valid semantically
else
it is invalid

another example : visiting the node ASTArrayAssignment
this node has three sons
1. ASTIdentifier
2. ASTExp1
3. ASTExp2

Checking this node is by Searching for the value of the ASTIdentifier in the symbol table
If the ASTIdentifier found in the Symbol Table
If the Symbol type is intArray
If ASTExp1,ASTExp2 type is int
this node is valid semantically
else else else it is invalid

Code Generation

The meaning of code generation is to convert the AST to micro instructions like ASM, MIPS, and we are going to generate a MIPS code.
But there is some changes occur on language grammars, you can see the new grammars in Appendix D.

The main idea is to traverse over AST and in each case we generate the suitable code, so we need to define a structure to implement the problem.

We will use processor registers `$t1` ..... `$tn` where $n \longrightarrow \infty$ to store variables and expressions values so we define a **`class`** `SymbolTable` witch store each variable with it's register (as the semantic symbol table) each element of this table is a `Symbol` object.

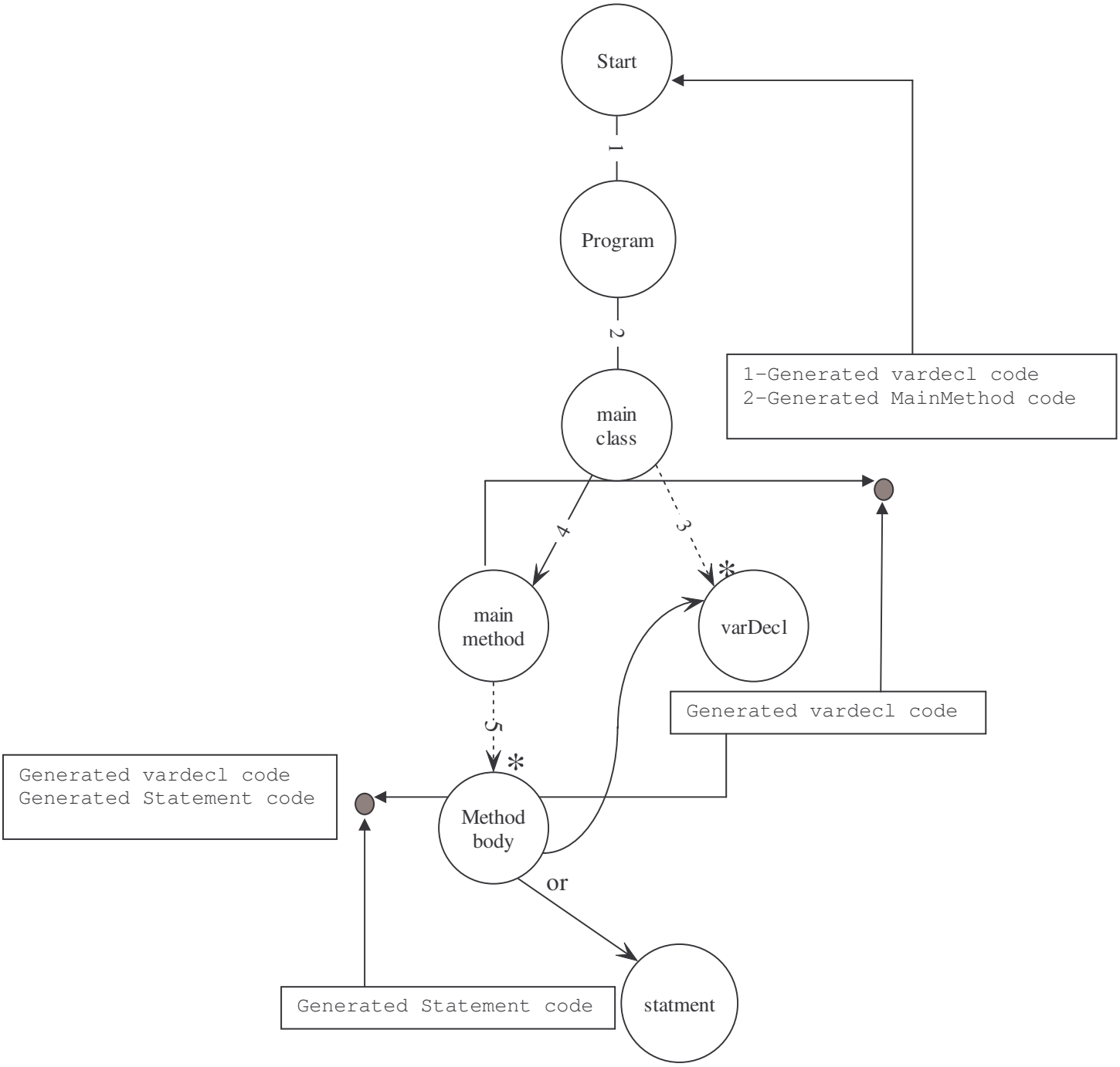
Symbol	
String RegName	Store register name.
String IdName	Store Identifier name.

To store each peace of the generated code we define `MIPSBlock`, and to store the name of the register witch contain result value of the block.

This class will be passed between the nodes of the AST , for example ASTExp node will pass it's MIPSBlock(*code + name of result register*) to her parent (maybe *while*, *if* ,...)

MIPSBlock	
String code	Store MIPS code block.
String regName	Store register name witch contains the value of the block.

To traverse over AST we need to implements `Visitor` interface, let's explain this by an example



As the figure above , we have to generate class data members code before we start generating main method code, those because main method maybe use them in it's body!.

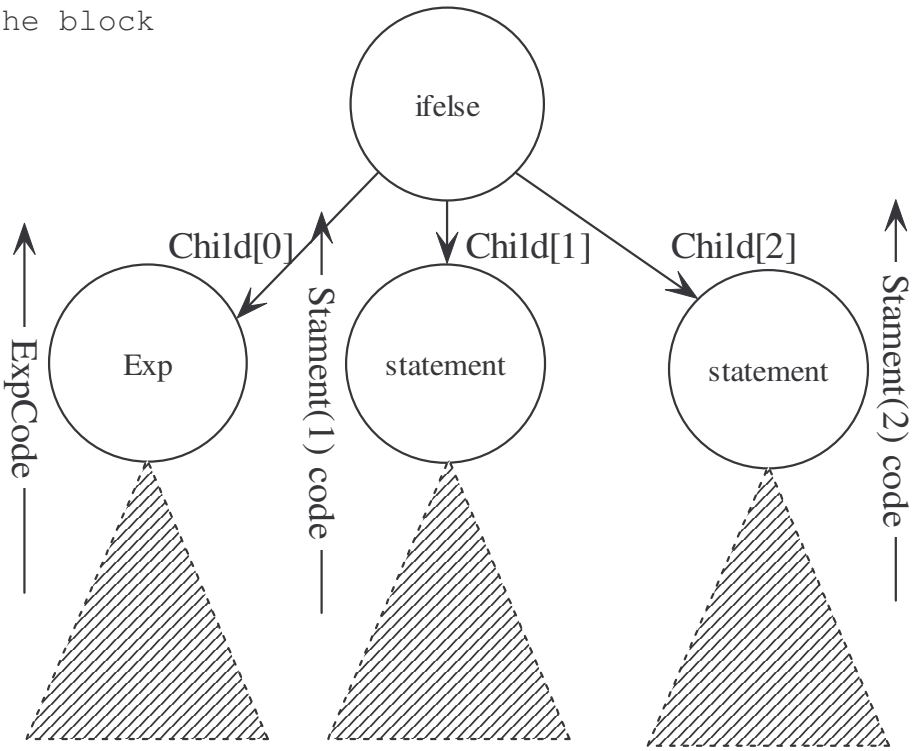
CodeGen for "if else" grammar

As the grammars **ASTifelse** node have 3 children

```
If exp
     Statement1
else
     Statement2
```

The returned MIPS code from this node will be generate as the follow

```
//Lab1 ← Generate-new-Label()
//Lab2 ← Generate-new-Label()
//past ExpMipsBlock.code

beqz ExpMipsBlock.regName , Lab1

//past ST1MipsBlock.code

j Lab2

Lab1

//past ST2MipsBlock.code

Lab2

Return the block
```



and we will implement the codegen for other grammars at the same way.

Tests

```
class Test
{
    public static void main(String[] args)
    {
        int x =-11;
        if (y > x )
        {
            x=y*2;
            System.out.println(x);
        }
        else
            x=7-y;
        while (x > 0)
        {
            x=x-1;
            System.out.println(x*y);
        }

    }
    int y=4+22;
}
```

```
  addi $t1 , $zero ,4
  addi $t2 , $zero ,22
  add $t1 , $t1 , $t2
  mov $t3 , $t1
  addi $t4 , $zero ,11
  sub $t4 , $zero , $t4
  mov $t5 , $t4
  sgt $t6 , $t3 , $t5
  beqz $t6 , Lable_1
  addi $t7 , $zero ,2
  mul $t3 , $t3 , $t7
  mov $t5 , $t3
  disp $t5
  j Lable_2Lable_1 :
  addi $t8 , $zero ,7
  sub $t8 , $t8 , $t3
  mov $t5 , $t8
Lable_2 :
Lable_3 :
  sgt $t10 , $t5 , $t9
  beqz $t10 , Lable_4
  addi $t11 , $zero ,1
  sub $t5 , $t5 , $t11
  mov $t5 , $t5
  mul $t5 , $t5 , $t3
  disp $t5
  j Lable_3
Lable_4 :
```

Appendix A. Tokens definition

```
TOKEN :
{
      < ID: <LETTER> (<LETTER>|<DIGIT>)* >
    | < #LETTER: ["_","a"-"z","A"-"Z"] >
    | < #DIGIT: ["0"-"9"] >
}
```

```
TOKEN :
{
      < INT: "int">
    | < INT_ARRAY: <INT><LBRACKET><RBRACKET> >
    | < BOOLEAN: "boolean">
    | < CLASS: "class">
    | < THIS: "this">
    | < EXTENDS: "extends">
    | < PUBLIC: "public">
    | < STATIC: "static">
    | < VOID: "void">
    | < RETURN: "return">
    | < MAIN: "main">
    | < IF: "if">
    | < ELSE: "else">
    | < WHILE: "while">
    | < PRINT: "System.out.println">
    | < NEW: "new">
    | < STRING: "String">
    | < LENGTH: "length">

}
```

```
TOKEN :
{
      < LPAREN: "(" >
    | < RPAREN: ")" >
    | < LBRACE: "{" >
    | < RBRACE: "}" >
    | < LBRACKET: "[" >
    | < RBRACKET: "]" >
    | < SEMICOLON: ";" >
    | < COMMA: "," >
    | < DOT: "." >
}
```

```
SKIP :
{
      " "
    | "\t"
    | "\n"
    | "\r"
}
```

```
TOKEN :
{
      < ASSIGN: "=" >
    | < GT: ">" >
    | < LT: "<" >
    | < NOT: "!" >
    | < EQ: "==" >
    | < OR: "||" >
    | < AND: "&&" >
    | < PLUS: "+" >
    | < MINUS: "-" >
    | < STAR: "*" >
    | < SLASH: "/" >

}
```

```
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
 |<FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
 |<MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}
```

```
TOKEN :
{
      < INTEGER_LITERAL: ["1"-"9"] (["0"-"9"])*>
    | < BOOLEAN_LITERAL: ("true"|"false")>
}
```


Appendix B. Languge grammars

Start ::= *Program*
Program ::= *MainClass (ClassDecl)**
MainClass ::= *Class Identifier <LBRACE> MainMethodHeader <LBRACE> (MethodBody)* <RBRACE> <RBRACE>*
MainMethodHeader ::= *<PUBLIC> <STATIC> <VOID> <MAIN> <LPAREN> <STRING> <LBRACKET> <RBRACKET> <ID> <RPAREN>*
ClassDecl ::= *Class Identifier (<EXTENDS> Identifier)? <LBRACE> (Body)* <RBRACE>*
Body ::= *(VarDecl | MethodDecl)*
VarDecl ::= *Type <ID> <ASSIGN> Exp <SEMICOLON>*
MethodDecl ::= *<PUBLIC> Type Identifier <LPAREN> FormalList <RPAREN> <LBRACE> (MethodBody)* <RETURN> Exp <SEMICOLON> <RBRACE>*
MethodBody ::= *(VarDecl | Statment)*
FormalList ::= *(Type Identifier (<COMMA> Type Identifier)*)?*
Type ::= *(BooleanType | Identifier | IntArrayType | IntegerType)*
Object ::= *(<THIS> | <ID>)*
Statment ::= *(<LBRACE> (Statment)* <RBRACE> | <IF> <LPAREN> Exp <RPAREN> Statment (<ELSE> Statment)? | <WHILE> <LPAREN> Exp <RPAREN> Statment | <PRINT> <LPAREN> Exp <RPAREN> <SEMICOLON> | (Object <DOT>)? <ID> (<ASSIGN> Exp | <LBRACKET> Exp <RBRACKET> <ASSIGN> Exp) <SEMICOLON>)*
Exp ::= *(OrExp | NewObject)*
OrExp ::= *AndExp (<OR> AndExp)**
AndExp ::= *NotExp (<AND> NotExp)**
NotExp ::= *(<NOT>)? LGEExp*
LGEExp ::= *AdditiveExpression ((<GT> | <LT> | <EQ>) AdditiveExpression)**
AdditiveExpression ::= *MultiplicativeExpression ((<PLUS> | <MINUS>) MultiplicativeExpression)**
MultiplicativeExpression ::= *UnaryExpression ((<STAR> | <SLASH>) UnaryExpression)**
UnaryExpression ::= *(Integer | Boolean | IdentifierExp | <LPAREN> Exp <RPAREN>)*
IdentifierExp ::= *(Object <DOT>)? Identifier (<LBRACKET> Exp <RBRACKET> | <DOT> <LENGTH> | <LPAREN> ExpList <RPAREN>)?*
NewObject ::= *<NEW> (Integer <LBRACKET> Exp <RBRACKET> | Identifier <LPAREN> <RPAREN>)*
ExpList ::= *(Exp (<COMMA> Exp)*)?*
Class ::= *<CLASS>*
Identifier ::= *<ID>*
Integer ::= *<INTEGER_LITERAL>*
IntegerType ::= *<INT>*
IntArrayType ::= *<INT_ARRAY>*
BooleanType ::= *<BOOLEAN>*
Boolean ::= *<BOOLEAN_LITERAL>*

Appendix C. jjtree Grammars

```
SimpleNode Start() : {}
{
    Program()
    { return jjtThis; }
}

void Program() : {}
{
    MainClass()( LOOKAHEAD(DerivedClass())
DerivedClass() | Class() )*
}

void MainClass() : {}
{
    <CLASS> Identifier() <LBRACE>
    <PUBLIC> <STATIC> <VOID> <MAIN>
<LPAREN> <STRING> <LBRACKET>
    <RBRACKET> <ID> <RPAREN> <LBRACE>
MainMethod() <RBRACE> <RBRACE>
}

void MainMethod():{}
{
    (MethodBody())*
}

void Class() : {}
{
    <CLASS> Identifier() <LBRACE> ( Body()
)* <RBRACE>
}

void DerivedClass() : {}
{
    <CLASS> Identifier()  <EXTENDS>
Identifier() <LBRACE> (Body() )* <RBRACE>
}

void Body() : {}
{
      VarDecl()
    | MethodDecl()
}

void VarDecl() : {}
{
    Type() Identifier() <ASSIGN> Exp()
<SEMICOLON>
}

void MethodDecl() : {}
{
    <PUBLIC> Type() Identifier() <LPAREN>
FormalList() <RPAREN> <LBRACE>
MethodBoodyBlock() <RETURN> Exp()
<SEMICOLON> <RBRACE>
}

void MethodBoodyBlock():{}
{
     (MethodBody())*
}

void MethodBody() : {}
{
      LOOKAHEAD(VarDecl())  VarDecl()
    | Statment()
}

void FormalList() : {}
{
    [  Parameter() ( <COMMA> Parameter() )* ]
}

void Parameter() : {}
{
    Type() Identifier()
}

void Type() : {}
{
      BooleanType()
    | Identifier()
    | LOOKAHEAD(IntArrayType())
IntArrayType()
    | IntegerType()
}

void Object() : {}
{
      This()
    | Identifier()
}

void Statment() : {}
{
     LOOKAHEAD(IfElse()) IfElse()
    | If()
    | While()
    | Print()
    | Assignment()
    | <LBRACE> (Statment())* <RBRACE>
}

void UnaryAssignment() : {}
{
    [ LOOKAHEAD(2) Object() <DOT> ]
Identifier() <ASSIGN> Exp()
}

void ArrayAssignment() : {}
{
   [ LOOKAHEAD(2) Object() <DOT> ]
Identifier()<LBRACKET> Exp() <RBRACKET>
<ASSIGN> Exp()
}

void Assignment() : {}
{
    (LOOKAHEAD(
ArrayAssignment())ArrayAssignment() |
UnaryAssignment()) <SEMICOLON>
}
```

```
void If() : {}
{
    <IF> <LPAREN> Exp() <RPAREN>
Statment()
}

void IfElse() : {}
{
    <IF> <LPAREN> Exp() <RPAREN>
Statment() <ELSE> Statment()
}

void Print() : {}
{
    <PRINT> <LPAREN> Exp() <RPAREN>
<SEMICOLON>
}

void While() : {}
{
    <WHILE> <LPAREN> Exp() <RPAREN>
Statment()
}

void Exp() : {}
{
      OrExp()
    | NewObject()
}

void OrExp() : {}
{
    AndExp() ( Or() AndExp() )*
}

void AndExp() : {}
{
    NotExp() ( And() NotExp())*
}

void NotExp() : {}
{
    [ Not() ] LGEExp()
}

void LGEExp() : {}
{
    AdditiveExpression()( (
GreaterThan() | LessThan() | Equal()
) AdditiveExpression())*
}


void Integer() : {}
{
    <INTEGER_LITERAL>
}

void IntegerType() : {}
{
    <INT>
}
```

```
void AdditiveExpression() : {}
{
    MultiplicativeExpression() ( (
Plus() | Minus() )
MultiplicativeExpression() )*
}

void MultiplicativeExpression() : {}
{
    UnaryExpression() ( ( Star() |
Slash() ) UnaryExpression() )*
}

void UnaryExpression() : {}
{
   [ Minus() ] Element()
}

void Element() : {}
{
      Integer()
    | Boolean()
    | IdentifierExp()
    | <LPAREN> Exp() <RPAREN>
}

void IdentifierExp() : {}
{
    LOOKAHEAD( Identifier() <DOT>
Length() )  Identifier() <DOT> Length()
    |
    [ LOOKAHEAD(Object() <DOT>) Object()
<DOT> ] Identifier() [   <LBRACKET>
Exp() <RBRACKET>

| <DOT> Length()

| <LPAREN> ExpList() <RPAREN>
]
}

void NewObject() : {}
{
    <NEW> (IntegerType() <LBRACKET>
Exp() <RBRACKET> | Identifier() <LPAREN>
<RPAREN> )
}

void ExpList() : {}
{
  [ Exp() ( <COMMA> Exp() )* ]
}

void Classs() : {}
{
    <CLASS>
}

void Identifier() : {}
{
    <ID>
}
```


```
void IntArrayType() : {}
{
    <INT_ARRAY>
}

void BooleanType() : {}
{
    <BOOLEAN>
}

void Boolean() : {}
{
    <BOOLEAN_LITERAL>
}

void Not() : {}
{
    <NOT>
}

void Or() : {}
{
    <OR>
}

void And() : {}
{
    <AND>
}

void Length() : {}
{
    <LENGTH>
}
```

```
void Star() : {}
{
    <STAR>
}
void Plus() : {}
{
    <PLUS>
}

void Minus() : {}
{
    <MINUS>
}

void Assign() : {}
{
    <ASSIGN>
}

void Slash() : {}
{
    <SLASH>
}

void Equal() : {}
{
    <EQ>
}

void LessThan() : {}
{
    <LT>
}

void GreaterThan() : {}
{
    <GT>
}

void This() : {}
{
    <THIS>
}
```