# MicroJ

## Final Report

Zichen Yang, zyang13@tufts.edu
Chenxuan Liu, cliu26@tufts.edu
Wei Sheng, wsheng03@tufts.edu
Weishi Ding, wding01@tufts.edu

April 2023

# Contents


**1 Introduction** **4**

**2 Features of MicroJ** **5**
2.1 Flexible Class, Function and Interface Definition . . . . . 5
2.2 Function Overloading . . . . . 5
2.3 Arrays . . . . . 5
2.4 OOP: Static/Non-Static, Public/Private, Inheritance, and Polymorphism . . . . . 6
2.5 Interfaces with Inheritance . . . . . 6

**3 Language Tutorial** **7**
3.1 Environment Setup . . . . . 7
3.2 Using the Compiler . . . . . 7
3.3 Brief introduction of how to use MicroJ . . . . . 7
3.3.1 A sample MicroJ ”Hello World” Program . . . . . 8
3.3.2 Variables and Statements . . . . . 8
3.3.3 Define and use functions . . . . . 8
3.3.4 Define and use interfaces . . . . . 9
3.3.5 Define and use classes . . . . . 9

**4 Language Reference Manual** **12**
4.1 Preface . . . . . 12
4.2 Credits . . . . . 12
4.3 Lsexical . . . . . 12
4.3.1 Comments . . . . . 12
4.3.2 Identifiers . . . . . 13
4.3.3 Keywords . . . . . 13
4.3.4 Literals . . . . . 13
4.4 Types, Values, and Variables . . . . . 15
4.4.1 Primitive Types, Values, and Variables . . . . . 15
4.4.2 Reference Types, Values, and Variables . . . . . 16
4.4.3 Default Values . . . . . 17
4.4.4 Global Variables . . . . . 17
4.5 Expressions . . . . . 18
4.5.1 Literal . . . . . 19
4.5.2 Binop Expression . . . . . 19
4.5.3 Unop Expression . . . . . 21
4.5.4 Funcall Expression . . . . . 21
4.5.5 New Funcall Expression . . . . . 21

4.5.6 New Array Expression . . . . . 22
4.5.7 Array Indexing Expression . . . . . 22
4.5.8 Access Expression . . . . . 22
4.5.9 Assign Expression . . . . . 23
4.5.10 Precedence Grouping Expression . . . . . 29
4.5.11 Operator precedence . . . . . 29
4.6 Statements . . . . . 30
4.6.1 Expression Statement . . . . . 30
4.6.2 Block Statement . . . . . 31
4.6.3 If Statement . . . . . 31
4.6.4 For Statement . . . . . 31
4.6.5 While Statement . . . . . 32
4.6.6 Control Flow Statement . . . . . 32
4.6.7 Return statement . . . . . 33
4.7 Functions . . . . . 33
4.7.1 Function definitions . . . . . 33
4.7.2 Function call . . . . . 34
4.8 Interfaces . . . . . 34
4.8.1 Syntax of Interfaces . . . . . 34
4.8.2 Abstract Methods . . . . . 35
4.8.3 Interface Inheritance . . . . . 35
4.9 Classes . . . . . 36
4.9.1 Syntax of Classes . . . . . 36
4.9.2 Access modifiers . . . . . 37
4.9.3 Static keyword . . . . . 37
4.9.4 Fields . . . . . 37
4.9.5 Methods . . . . . 37
4.9.6 Constructors . . . . . 37
4.9.7 Class Inheritance . . . . . 38
4.9.8 Polymorphism . . . . . 39
4.10 Appendix . . . . . 42
4.10.1 Expressions . . . . . 42
4.10.2 Declarations . . . . . 43
4.10.3 Statements . . . . . 44
4.10.4 Function Definition . . . . . 44
4.10.5 Interface Definition . . . . . 45
4.10.6 Class Definition . . . . . 45

**5 Project Plan** **46**
5.1 Project Timeline . . . . . 46
5.2 Contribution . . . . . 48
5.2.1 Zichen Yang . . . . . 48
5.2.2 Chenxuan Liu . . . . . 48
5.2.3 Wei Sheng . . . . . 48
5.2.4 Weishi Ding . . . . . 48
5.3 Software Development Environment . . . . . 49

**6 System Architecture and Design** **50**
6.1 Overall Architecture . . . 50
6.2 Different Components of the Compiler . . . 51
6.2.1 Scanner . . . 51
6.2.2 Parser . . . 52
6.2.3 Semantic Checker . . . 52
6.2.4 Code Generator . . . 53

**7 Test Plan** **55**
7.1 Unit test, integration test and Automation . . . 55
7.1.1 Unit test . . . 55
7.1.2 Integration test . . . 55
7.1.3 Automation . . . 57
7.2 Responsibilities . . . 57
7.3 Source Code and the LLVM code Generated . . . 57
7.3.1 check static and access control of class . . . 57
7.3.2 Test interface implementation and class extension . . . 61
7.3.3 Test polymorphism . . . 63

**8 Lessons Learnt** **68**
8.1 Zichen Yang . . . 68
8.2 Sheng Wei . . . 69
8.3 Chenxuan Liu . . . 69
8.4 Weishi Ding . . . 70

**A Appendix** **71**
A.1 Translator . . . 71
A.1.1 toplevel.ml . . . 71
A.1.2 scanner.mll . . . 72
A.1.3 parser.mly . . . 74
A.1.4 ast.ml . . . 79
A.1.5 semant.ml . . . 85
A.1.6 sast.ml . . . 105
A.1.7 codegen.ml . . . 110
A.2 Killer Apps . . . 132
A.2.1 Inheritance showcase . . . 132
A.2.2 Interface showcase . . . 133
A.2.3 Polymorphism showcase . . . 134

# Chapter 1

# Introduction

The primary purpose and motivation behind MicroJ was to provide a simplified programming language for software development. Our team recognized that while Java is a popular language, its complexity can sometimes be a barrier to entry for new programmers. We wanted to create a language that would be more accessible to those who are just starting out in programming, while still providing essential OOP features.

MicroJ is a great choice for a wide range of software development use cases, including scripting, and basic desktop applications. With its simplified syntax and essential OOP features, developers can easily create objects and classes with inheritance, making it easier to create modular, maintainable code. While MicroJ may not be suitable for complex mobile application development, game development, and scientific computing, it is a great choice for those who are new to programming or looking for a simpler alternative to Java.

To fully understand MicroJ, it is important to have a basic understanding of programming concepts such as OOP, inheritance, and arrays. While MicroJ is designed to be accessible to new programmers, some familiarity with these concepts will be helpful in fully leveraging the language's capabilities.

In conclusion, MicroJ is a powerful, yet easy-to-use programming language that provides a simplified alternative to Java. With its familiar syntax and essential OOP features, MicroJ is a great choice for a wide range of software development use cases, from scripting to basic desktop applications. Whether you are a new programmer looking to learn the basics of OOP or an experienced developer seeking a simpler alternative to Java, MicroJ is a language that is sure to meet your needs.

# Chapter 2

# Features of MicroJ

MicroJ is a powerful programming language. In this chapter, we will introduce some of its key features.

## 2.1 Flexible Class, Function and Interface Definition

One of the most notable features of our compiler is the ability to define classes, functions and interfaces anywhere in the user's code. This is similar to Python's approach to function and class definition. With MircoJ, users can define a function or class at the bottom of the user's code and still use it anywhere in the file. This feature allows for more flexibility in code organization and makes it easier to write and manage large code bases.

## 2.2 Function Overloading

Another great feature of MicroJ is the support for function overloading. Using Micro-J, users can define multiple functions with the same name, as long as their signatures are different. In other words, users can define two or more functions with the same name, which will be distinguished by their parameters. This feature makes it easier to write clean and concise code, as users can use the same function name for functions with similar functionality, without the need to create a different function name each time.

## 2.3 Arrays

Array is a built-in data-structure in MicroJ. In addition to basic array types such as integer array, MicroJ supports object arrays. Object arrays are arrays that store objects instead of primitive data types. MicroJ also supports array indexing, which is a critical feature for working with arrays. Array indexing allows users to access individual elements within an array using an index number. This feature is particularly useful when working with large data sets or when the user need to perform operations on specific elements of an array.

## 2.4 OOP: Static/Non-Static, Public/Private, Inheritance, and Polymorphism

Our compiler also supports object-oriented programming (OOP) concepts such as static/non-static, public/private, inheritance, and polymorphism. With MicroJ, users can create static and non-static methods and variables, set access modifiers to control visibility, and use inheritance and polymorphism to create hierarchies of classes that share common attributes and behaviors.

## 2.5 Interfaces with Inheritance

Our compiler also supports interfaces with inheritance. Interfaces are a powerful feature of OOP that allow users to define a set of methods that a class must implement. With inheritance, users can define an interface that inherits from multiple interfaces, providing more flexibility in users' code designs. This feature is particularly useful when working with complex systems that require a high degree of modularity and extensibility.

In conclusion, MicroJ offers a range of features that let users write flexible, efficient, and powerful code.

# Chapter 3

# Language Tutorial

## 3.1 Environment Setup

Users can set up the environment by using the following command:

```
sudo apt-get install llvm-14 m4 cmake opam
opam install ocaml dune
```

Packages version:
llvm-14 14.0.0
m4 1.4.18
cmake 3.22.1
opam 2.1.2
dune 3.7.1
ocaml 4.13.1

## 3.2 Using the Compiler

Users can use LLoutPut.sh to execute the code by typing: ./LLoutPut.sh {$1}
Users should replace {$1} by the name of the code file in the Test directory
example: ./LLoutPut.sh Test1

## 3.3 Brief introduction of how to use MicroJ

The grammar of MicroJ is like the mixture of Java, C++, and Python. The layout of a MicroJ program is similar to a C++ program. If you are familiar with those languages, you can easily hand on MicroJ. The main components of a MicroJ program include:

- Global variable definition
- Function definition
- Class definition
- Interface definition
- A main function as entry point

### 3.3.1 A sample MicroJ "Hello World" Program

Here's a sample "Hello World" program written in MicroJ:

```
int main(){
    print("Hello World");
}
```

The output of this code:
"Hello World"

### 3.3.2 Variables and Statements

MicroJ provides a range of data types including bool, int, double, string, Object, and Array. It also supports for, while, and if statements for users to write their programs. For example:

```
int main(){
    int[] intArray := new int[10];
    int i;

    for(i = 0; i < 10; i = i+1){
        intArray[i] := i;
    }

    for(i = 0; i < 10; i = i+1){
        if(i == 5){
            break;
        }
    }

    print(intArray[i]);
}
```

The output of this code:
5

### 3.3.3 Define and use functions

Functions can be defined and called by providing their name and arguments, and the order in which functions are defined does not matter. Users are able to call functions even if they are defined after the function call. For example:

```
int main(){
    printString("hello world");
}

string printString(string s){
    print(s);

    return s;
}
```

The output of this code:
"Hello World"

### 3.3.4 Define and use interfaces

An interface defines a set of abstract methods that a class can implement. Interfaces can also extend other interfaces, and a class that implements an interface must provide concrete implementations for all its abstract methods. Same as function, the order of interfaces definition does not matter. For example:

```
interface Window{
    void window();
}

interface Wheel{
    void wheel();
}

interface Factory extends Window, Wheel{

}

class Car implements Factory{
    constructor Car(){

    }

    void window(Car self){
        print("6");
    }

    void wheel(Car self){
        print("4");
    }
}

int main(){
    Car c : = new Car();
    c.window();
    c.wheel();
}
```

The output of this code:
6
4

### 3.3.5 Define and use classes

A class defines a set of methods and fields. A class can extend exactly one class and can implement multi interfaces. Users can create a class instance by the "new" keyword with the call of a constructor. Same as function, the order of class definition is not matter. For example:

```
class Father {
    public static int fPubStat;
    private static int fPriStat;

    public int fPub;
    private int fPri;

    constructor Father(int fPub, int fPri, int fPubStat, int fPriStat){

    }

    constructor Father(){

    }

    public static void fPublicStaticPrint(){
        print("public static");
    }

    private static void fPrivatetStaticPrint(){
        print("private static");
    }

    public void fPublicPrint(Father self){
        print("public");
    }

    private void fPrivatePrint(Father self){
        print("private");
    }
}

class Son extends Father{
    constructor Son(){

    }

    public void sTestPublic(Son self){
        Father.fPublicStaticPrint();
        self.fPublicPrint();
        print(self.fPub);

        Father f := new Father();
        f.fPublicPrint();
        print(f.fPub);
    }

    public void sTestPrivate(Son self){
        Father.fPrivatetStaticPrint();
```

```
        self.fPrivatePrint();
        print(self.fPri);

        Father f := new Father();
        f.fPrivatePrint();
        print(f.fPri);
    }
}

int main(){
    Son s := new Son();
    s.sTestPublic();
    s.sTestPrivate();
}
```

The output of this code:
"public static"
"public"
0
"public"
0
"private static"
"private"
0
"private"
0

# Chapter 4

# Language Reference Manual

## 4.1 Preface

This is a reference manual for the **MicroJ** programming language. This document aims to document syntax and grammar for the language, and also provide some small program examples.
**MicroJ** is an object-orientated language with more flexibility than traditional object-orientated languages like **Java** or **C++**.
In this language reference manual, the prose would appear as regular texts in Computer Modern font without any special styling.
*Language Grammar rules* would follow *BNF* form, *placeholders* appearing in *italic font*.
Keywords and other predefined token, like {} that users need to use in the program are shown in sans serif font

```
Code Blocks will be shown in green backgrounds
```

## 4.2 Credits

**MicroJ** is inspired by the **Java** programming language, the **Python** programming language and the **C++** programming language. The design of this language reference manual is inspired by the **GNU C Language Reference Manual**.

## 4.3 Lsexical

### 4.3.1 Comments

Multiline comments start with /* and end with */. All the text in between will be ignored.

**Grammar of Comments:**

$\langle \textit{comments} \rangle$ ::= /* $\langle \textit{any UTF\_8 characters} \rangle$+ */

**Example Code of Comment in MicroJ:**

```
/* This is a comment in MicroJ */
```

### 4.3.2 Identifiers

Identifiers are sequences of characters used for naming variables, functions, and self-defined data types. It can include letters, decimal digits, and the underscore character _in identifiers. The first character of an identifier cannot be a digit.

**Grammar of Identifier:**

$\langle identifier \rangle$ ::= $\langle letter \rangle$ $\langle identifier\text{-}tail \rangle$

$\langle identifier\text{-}tail \rangle$ ::= $\langle letter\text{-}or\text{-}digit\text{-}or\text{-}underscore \rangle$ $\langle identifier\text{-}tail \rangle$

$\langle letter\text{-}or\text{-}digit\text{-}or\text{-}underscore \rangle$ ::= $\langle letter \rangle$ | $\langle digit \rangle$ | _

$\langle letter \rangle$ ::= a | b | c | ... | x | y | z | A | B | C | ... | X | Y | Z

$\langle digit \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**Example Code of Identifier in MicroJ:**

```
variableName; variable_name; ClassName; ClassName2;
```

### 4.3.3 Keywords

Keywords are special identifiers reserved as part of the programming language MicroJ itself. Users cannot use Keywords for other purposes.
Here is a list of all keywords in MicroJ:

$\langle keywords \rangle$ := break | continue | return
| for | while
| if | else
| interface | class | new | static | constructor
| implements | extends
| private | protected | public
| int | string | double | bool | void

### 4.3.4 Literals

A literal is the source code representation of a value of a primitive type.

#### 4.3.4.1 Integer Literals

An integer is a sequence of digits with or without a negative sign, the prefix 0 will be ignored.

**Grammar of Integer Literals:**

$\langle$*integer-literals*$\rangle$ ::= [-]$\langle$*digit*$\rangle$+

**Example Code of Integer Literal in MicroJ:**

```
1; -1; -20; 100; 00000313;
```

#### 4.3.4.2 Double Literals

A float number is a sequence of digits followed by a . and a sequence of digits. The first sequence represents the integer part of the floating point number . represents the decimal point, and the second sequence represents the fractional part.

**Grammar of Double Literals:**

$\langle$*double-literals*$\rangle$ ::= [-]$\langle$*digit*$\rangle$+ . $\langle$*digit*$\rangle$+

**Example Code of Double Literal in MicroJ:**

```
1.2312323; 1.0; -909.1233; 0.112312; -0.0123;
```

#### 4.3.4.3 Boolean Literals

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters.

**Grammar of Boolean Literals:**

$\langle$*bool-literals*$\rangle$ ::= true | false

**Example Code of Boolean Literal in MicroJ:**

```
true; false;
```

#### 4.3.4.4 String Literals

A string literal consists of zero or more UTF_8 characters without double quote characters, enclosed in double quotes.

**Grammar of Boolean Literals:**

$\langle$*string-literals*$\rangle$ ::= " $\langle$*any UTF_8 characters*$\rangle$+ "

**Example Code of String Literal in MicroJ:**

```
"Hello, world!";
```

#### 4.3.4.5 Separators

A separator separates tokens, and they are the single-character tokens below:
( ) [ ] { } ; ,

#### 4.3.4.6 Operators

An operator is a special token that represents an operation, such as addition, assignment, or value comparison operation. Operators and their usage are revisited in chapter Expression. The operators are the tokens below:

```
=
< > >= <= != == && ||
+ - * / .
```

## 4.4 Types, Values, and Variables

**Grammar of Data Types:**

⟨*variable-type*⟩ ::= int | double | bool | string | ⟨*object*⟩ | ⟨*array-type*⟩

⟨*array-type*⟩ ::= ⟨*variable-type*⟩ [ ]

⟨*object*⟩ ::= ⟨*identifier*⟩

Note that, *identifier* refers to a class name. And class is discussed in Chapter 7 Classes.
Notice that MircoJ doesn't support N-dimension array type.

### 4.4.1 Primitive Types, Values, and Variables

**Grammar of Defining Variables with Primitive Types:**

⟨*primitive-type-variable*⟩ ::= ⟨*primitive-type*⟩ ⟨*identifier*⟩ [: = ⟨*expression*⟩]

⟨*primitive-type*⟩ ::= int | double | bool | string

More detailed description of Expression could be found in the Expression Chapter.

#### 4.4.1.1 Integral Type and Values

The values of the integral types are integers in the range: $-2^{63}$ *to* $2^{63} - 1$

#### 4.4.1.2 Double Type and Values

The values of the double type are real numbers in the approximate range: $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$.

#### 4.4.1.3 Boolean Type and Values

The bool type represents a logical quantity with two possible values, indicated by the literals true and false.

#### 4.4.1.4 The String Type and Values

In MircoJ, a string is zero or more UTF_8 characters without double quote characters, enclosed in double quotes.

### 4.4.2 Reference Types, Values, and Variables

There are two reference types in MircoJ: class and array.

#### 4.4.2.1 Class

**Grammar for Declaring and Initializing Class Variables:**

$\langle \textit{class-type-variable} \rangle$ ::= $\langle \textit{object} \rangle \langle \textit{identifier} \rangle$ [ : = new $\langle \textit{object} \rangle$ ( [$\langle \textit{expression} \rangle$+] ) ]

**Example Code of Class Type Variables:**

```
/* define a class */
class ClassName {
    constructor ClassName() {}
}
int main() {
    /*declare a class type variable*/
    ClassName variableName;

    /*define an object reference and initialize it with constructor*/
    ClassName variableName : = new ClassName();
}
```

More detailed description of Class could be found in the Class Chapter.

#### 4.4.2.2 Array

An array is a container that can hold a collection of values of the same type. The values in an array are stored in contiguous memory locations and can be accessed using an index, which is a zero-based integer that represents the position of the element in the array.

**Grammar for Declaring and Initializing Array:**

$\langle \textit{array-type-variable} \rangle$ ::= $\langle \textit{array-type} \rangle$ $\langle \textit{identifier} \rangle$ [ : = new $\langle \textit{variable-type} \rangle$ [ $\langle \textit{expression} \rangle$ ] ]

Note that, variable type only refer to int, bool, double, string and object types. Also, expression must have a return value of integer.

#### 4.4.2.3 Access Array Elements

Users can access the value stored at the specific index of the array.

**Grammar of Access Array Elements:**

$\langle \textit{array-access} \rangle$ ::= $\langle \textit{identifier} \rangle$ [ $\langle \textit{expression} \rangle$ ]

Note that, expression must have a return value of integer. By accessing the value stored at the specific index of the array, users can either get its value or modify its value.

**Example Code of Declaration of Array Type Variables and Array Access:**

```
class Student {
    constructor Student(){}
}


int main(){
    /* declaration and initialization of primitive array */
    int[] intArray;

    /* initialization array with speific dimension */
    int[] intArray2 : = new int[10];

    /* declaration and initialization of reference array */
    Student[] studentArray : = new Student[10];

    /* declare variable James */
    Student James : = new Student();

    /* access and modify the value at a specific index of an array */
    studentArray[2] : = James;

    /* access and get value at specific index of an array */
    Student Amy : = studentArray[2];
}
```

### 4.4.3 Default Values

MircoJ allows variables to be defined without initialization. The default value of each type is listed below:

- int type: 0
- double type: 0.0
- bool type: false
- string type: null
- reference type: null

### 4.4.4 Global Variables

MicroJ allows to define global variables. Users can define integer, double, bool. Note that a global variable of string type cannot be initialized in a global context, but it can be initialized in main() function.

**Example Code of Global Variables**

```
int counter : = 0;
bool flag : = true;
double data : = 3.14;
string global_str;  /* String type variable cannot be initialized here */

int main(){
    global_str = "Hello World";   /* Can be initialized here */
    while ( counter < 5 && flag ){
        counter = counter + 1;
        data = data * 2.0;
        if (data == 4.0 * 3.14) {
            flag = false;
        }
    }
    print(counter);
    /* The result should be 2 */
    print(flag);
    /* The result should be 0 */
    print(data);
    /* The result should be 12.560000 */
}
```

## 4.5 Expressions

An expression is made up of one or more operands and zero or more operators. Operands refer to objects, variables, literals, and function calls. Operator refers to an operation to be performed on the operands.

**Grammar of Expressions:**

$\langle expression \rangle$ ::= $\langle literal \rangle$
| $\langle identifier \rangle$
| $\langle binop\text{-}expression \rangle$
| $\langle unop\text{-}expression \rangle$
| $\langle funcall\text{-}expression \rangle$
| $\langle new\text{-}funcall\text{-}expression \rangle$
| $\langle new\text{-}array\text{-}expression \rangle$
| $\langle array\text{-}indexing\text{-}expression \rangle$
| $\langle assign\text{-}expression \rangle$
| $\langle access\text{-}expression \rangle$
| $\langle precedence\text{-}grouping\text{-}expression \rangle$

### 4.5.1 Literal

In **MicroJ**, literals are of the following types: integer, double, boolean, string.

#### Grammar of Literal

$\langle literal \rangle$ ::= | $\langle$*integer-literals*$\rangle$
| $\langle$*double-literals*$\rangle$
| $\langle$*boolean-literals*$\rangle$
| $\langle$*string-literals*$\rangle$

For integer literal, see more detail in section Integer Literals.
For double literal, see more detail in section Double Literals.
For boolean literal, see more detail in section Boolean Literals.
For string literal, see more detail in section String Literals.

### 4.5.2 Binop Expression

#### Grammar of Binop Expression

$\langle$*binop-expression*$\rangle$ ::= $\langle$*arithmetic-operator-expression*$\rangle$
| $\langle$*relational-operator-expression*$\rangle$

#### 4.5.2.1 Arithmetic Operator Expression

Notice that *expression* in the grammar are expression of integer number type or double number type.

#### Grammar of Arithmetic Operator Expression

$\langle$*arithmetic-operators-expression*$\rangle$ ::= $\langle$*expression*$\rangle$ `+` $\langle$*expression*$\rangle$
| $\langle$*expression*$\rangle$ `-` $\langle$*expression*$\rangle$
| $\langle$*expression*$\rangle$ `*` $\langle$*expression*$\rangle$
| $\langle$*expression*$\rangle$ `/` $\langle$*expression*$\rangle$

**Example Code of Arithmetic Operators in MicroJ:**

```
int main(){
    /* Addition */
    print(4 + 3);   /* The result should be 7. */
    print(12.6 + 9.84); /* The result should be 22.440000. */

    /* Subtraction*/
    print(4 - 3);   /* The result should be 1. */
    print(12.6 - 9.84);    /* The result should be 2.760000. */

    /* Multiplication*/
    print(4 * 3);  /* The result should be 12. */
    print(12.6 * 9.84);    /* The result should be 123.984000 */
```

```
    /*Division*/
    /* Integer division will remove the trailing numerals to 0,
    for example 4 / 3 is 1. */
    print(4 / 3);   /* The result should be 1 */
    print(12.6 / 9.84);   /* The result should be 1.2804878049 */
}
```

#### 4.5.2.2 Relational Operator Expression

Users may use relational operators to test the relation between two operands of the same type, and the operands can be of integer type, double type, boolean type, string type, object type, and array type.

Specifically, users can test whether one integer type value or floating point type value is less than (<), greater than (>), less than or equals to (<=), or greater than or equal to (>=) another integer type value or double type value; but user will not be able to perform these operations on boolean type values, string type values and object type values.

User can test whether two values of the above type are the same using equal (==) operator, or not the same using the not equal (`!=`) operator.

Note that the only operators work on object type and array type are (==) and (!=), which will compare the operands' addresses.

**Grammar of Relational Operator Expression:**
Notice that the two *expression* being compared must be of the same type.

⟨*relational-operator-expression*⟩ ::= ⟨*expression*⟩ `==` ⟨*expression*⟩
| ⟨*expression*⟩ `!=` ⟨*expression*⟩
| ⟨*expression*⟩ < ⟨*expression*⟩
| ⟨*expression*⟩ > ⟨*expression*⟩
| ⟨*expression*⟩ <= ⟨*expression*⟩
| ⟨*expression*⟩ >= ⟨*expression*⟩

**Example Code of Comparison Operators of Integer and Double types:**

```
int main(){
    print(3 != 5); /* should return 1*/

    print(3 != 3); /* should return 0 */

    print(4.12 == 4.12); /* should return 1 */

    print(3 >= 5); /* should return 0 */

    print(3.14159 > 2.0) ; /* should return 1 */
}
```

### 4.5.3 Unop Expression

In MicroJ, there are only two unary operators in our compiler: - and !.

**Grammar of Unop Expression:**

$\langle unop\text{-}expression \rangle$ ::= - $\langle expression \rangle$

| ! $\langle expression \rangle$

**Example Code of Unop Expression:**

```
int main(){
    int a : = 2;
    print(-a); /* THe result should be -2. */
    int b : = 0;
    print(-b); /* The result should be 0. */

    bool c : = true;
    print(!c) /* The result should be 0 */
}
```

### 4.5.4 Funcall Expression

Regardless of whether the function takes in zero or more arguments, user need to call the function by its name followed by () brackets.

**Grammar of Function Call:**

$\langle funcall\text{-}expression \rangle$ ::= $\langle identifier \rangle$ ( [ $\langle expression\text{-}list \rangle$ ] )

**Example Code of Function Call:**

```
int IntegerAdd(int x, int y) {
    return x + y;
}
int main(){
    /* call a function */
    print(IntegerAdd(3, 4));
}
```

### 4.5.5 New Funcall Expression

A Function Call followed by new keyword must be a constructor function, which will create a class type instance.

**Grammar of New Funcall Expression:**

$\langle new\text{-}funcall\text{-}expression \rangle$ ::= new $\langle object \rangle$ ( [$\langle expression \rangle$+] )

To see more on this topic, please refers to Class Type Variables.

### 4.5.6 New Array Expression

User can create a new Array of specific type by using `new` keyword followed by a Literal type or Object type, then followed by [ ] keyword, where an expression of integer type representing the length of the array will be within [ ] keyword.

**Grammar of New Array Expression:**

⟨*new-array-expression*⟩ ::= `new` ⟨*variable-type*⟩ [ ⟨*expression*⟩ ]

To see more on this topic, please refers to Array Type Variables.

### 4.5.7 Array Indexing Expression

Users can access the value stored at an index of of an array element by using the name of the variable strong the array, followed by [ ] keyword, where an expression of integer type representing the index will be within [ ] keyword.

**Grammar of Array Indexing Expression:**

⟨*array-indexing-expression*⟩ ::= ⟨*identifier*⟩ [ ⟨*expression*⟩ ]

To see more on this topic, please refers to See Access Array Elements.

### 4.5.8 Access Expression

User can access fields and methods of class type variables. Accessing methods of class type variables is to make a function of that method.

**Grammar of Access Expression**

⟨*access-expression*⟩ ::= ⟨*field-access*⟩
| ⟨*method-access*⟩

⟨*field-access*⟩ :: = ⟨*regular-field-access*⟩
| ⟨*array-indexing-field-access*⟩

⟨*regular-field-access*⟩ ::= ⟨*identifier*⟩ . ⟨*identifier*⟩
| ⟨*array-indexing-expression*⟩ . ⟨*identifier*⟩

⟨*array-indexing-field-access*⟩ ::= ⟨*identifier*⟩ . ⟨*array-indexing-expression*⟩
| ⟨*array-indexing-expression*⟩ . ⟨*array-indexing-expression*⟩

⟨*method-access*⟩ :: = ⟨*identifier*⟩ . ⟨*identifier*⟩ ( [ ⟨*expression-list*⟩ ] )
| ⟨*array-indexing-expression*⟩ . ⟨*identifier*⟩ ( [ ⟨*expression-list*⟩ ] )

**Example Code of Access expression:**

```
class MyClass {
    bool[] bArray;
    double d;

    constructor MyClass(){

    }

    void method(MyClass self){

    }
}

void main(){
    MyClass[] cArray : = new MyClass[3];
    cArray[0] : = new MyClass();
    MyClass c : = new MyClass();

    c.d;
    cArray[0].d;

    c.bArray[0];
    cArray[0].bArray[0];

    c.method();
    cArray[0].method();
}
```

### 4.5.9 Assign Expression

Assignment operators store values in variables. There are two assignment operator in MicroJ, = and : =. They store the value of the operand on the right into the variable on the left. The left operand to an assign operator cannot be a literal.

**Grammar of Assign Expression:**

$\langle$*assign-expression*$\rangle$ ::= $\langle$*regular-assign-expression*$\rangle$
| $\langle$*definition-assign-expression*$\rangle$

#### 4.5.9.1 Regular Assign Expression

Regular Assign Expressions are expressions used when user have already define a variable of certain type and are now assigning value to the variable. Primitive types and Class types all use (=) operator when performing definition and assignment at the same type. User should note that, assigning value to Array or Array Indexing also uses (: =) operator, and they are incorporated in the discussion of Definition Assignment Expression immediately following current section.

**Grammar of Regular Assign Expression:**

⟨*regular-assign-expression*⟩ ::= ⟨*identifier-assign-expression*⟩
| ⟨*field-assign-expression*⟩

⟨*identifier-assign-expression*⟩ ::= ⟨*identifier*⟩ = ⟨*literal*⟩
| ⟨*identifier*⟩ = ⟨*identifier*⟩
| ⟨*identifier*⟩ = ⟨*binop-expression*⟩
| ⟨*identifier*⟩ = ⟨*unop-expression*⟩
| ⟨*identifier*⟩ = ⟨*funcall-expression*⟩
| ⟨*identifier*⟩ = ⟨*new-funcall-expression*⟩
| ⟨*identifier*⟩ = ⟨*new-array-expression*⟩
| ⟨*identifier*⟩ = ⟨*array-indexing-expression*⟩
| ⟨*identifier*⟩ = ⟨*access-expression*⟩

⟨*field-assign-expression*⟩ ::= ⟨*field-access*⟩ = ⟨*literal*⟩
| ⟨*field-access*⟩ = ⟨*identifier*⟩
| ⟨*field-access*⟩ = ⟨*binop-expression*⟩
| ⟨*field-access*⟩ = ⟨*unop-expression*⟩
| ⟨*field-access*⟩ = ⟨*funcall-expression*⟩
| ⟨*field-access*⟩ = ⟨*new-funcall-expression*⟩
| ⟨*field-access*⟩ = ⟨*array-indexing-expression*⟩
| ⟨*field-access*⟩ = ⟨*new-array-expression*⟩
| ⟨*field-access*⟩ = ⟨*access-expression*⟩

User should note that in the above BNF, if *field-access* is of type *array-indexing-expression*, which represents array indexing, it will be matched to the Definition Assign Expression case, which will be discussed in the following chapter, only when *array-indexing-expression* matches *identifier. array-indexing-expression*; in the other case, it will match to the Regular Assign Expression.

**Example Code of Identifier Assign Expression:**

```
/* Examples for identifier assign expression */
/* Test for identifier assign */
class TestClass{
    int temp;
    constructor TestClass(int temp){}
}

int time2 (int a){
    return a * 2;
}

```

```
int main(){
    /* identifier  = literal  */
    int a;
    a = 3;
    print(a);
    /* The output should be 3. */

    /* identifier  = identifier  */
    int b;
    b = a;
    print(b);
    /* The output should be 3. */

    /* identifier  = binop-expression */
    bool c;
    c = true && false;
    print(c);
    /* The output should be 0. */

    /* identifier  = unop-expression */
    c = !c;
    print(c);
    /* The output should be 1. */

    /* identifier  = funcall-expression */
    a = time2(a);
    print(a);
    /* The output should be 6. */

    /* identifier  = new-funcall-expression */
    TestClass d;
    d = new TestClass(1);
    /* identifier  = new-array-expression */
    int[] arr;
    arr = new int[10];
    arr[3] : = 2;
    print(arr[3]);
    /* The output should be 2. */

    /* identifier  = array-indexing-expression */
    a = arr[3];
    print(a);
    /* The output should be 2. */
    int temp;
    /* identifier  = access-expression */
    temp = d.temp;
    print(temp);
    /* The output should be 1. */

}
```

**Example Code of Field Assign Expression:**

```
class myClass{
    int a;
    int b;
    bool c;
    bool d;
    myClass2 my2;
    constructor myClass(int a, int b, bool c, bool d){}
}

class myClass2{
    int a2;
    int b2;
    int[] arr2;
    constructor myClass2(int a2, int b2){}
}

int time2 (int a){
    return a * 2;
}

int main(){
    myClass temp1 : = new myClass(1, 2, true, false);
    /* field-access = literal */
    temp1.a = 10;
    print(temp1.a);
    /* The oupt put should be 10. */

    /* field-access = identifier */
    int e : = 2;
    temp1.a = e;
    print(temp1.a);
    /* The output should be 2. */

    /* field-access = binop-expression */
    temp1.b = 4 + 7;
    print(temp1.b);
    /* The output should be 11 */

    /* field-access = unop-expression */
    temp1.c = !temp1.c;
    print(temp1.c);
    /* The output should be 0 */

    /* field-access = funcall-expression */
    temp1.a = time2(temp1.a);
    print(temp1.a);
    /* The output should be 4. */

    /* field-access = new-funcall-expression */
```

```
    temp1.my2 = new myClass2(3,4);
    /* field-access = new-array-expression */
    myClass2 temp2 : = new myClass2(5, 6);
    /* field-access = access-expression */
    temp2.a2 = temp1.a;
    print(temp2.a2);
    /* The output should be 4. */

    /* Here's an example of identifier.array-indexing-expression */
    /* Which will be discussed in the next chapter */
    /* temp2.arr2[0] : = 2; */
}
```

#### 4.5.9.2 Definition Assign Expression

Definition Assign Expressions are expressions used when users define a variable of certain type and assign value at the same type. Primitive types and Class types all use (: =) operator when performing definition and assignment at the same type. User should note that, assigning value to Array or Array Indexing also uses (: =) operator, and they are incorporated in the BNF forms below.

**Grammar of Definition Assign Expression:**

$\langle$*definition-assign-expression*$\rangle$ ::= $\langle$*regular-defasn*$\rangle$
| $\langle$*array-indexing-defasn*$\rangle$

$\langle$*regular-defasn*$\rangle$ ::= $\langle$*variable-type*$\rangle$ $\langle$*identifier*$\rangle$ : = $\langle$*literal*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*identifier*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*binop-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*unop-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*funcall-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*new-funcall-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*array-indexing-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*new-array-expression*$\rangle$
| $\langle$*variable-type*$\rangle\langle$*identifier*$\rangle$ : = $\langle$*access-expression*$\rangle$

$\langle$*array-indexing-defasn*$\rangle$ ::= $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*literal*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*identifier*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*binop-expression*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*unop-expression*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*funcall-expression*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*new-funcall-expression*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*array-indexing-expression*$\rangle$
| $\langle$*array-indexing-expression*$\rangle$ : = $\langle$*access-expression*$\rangle$

**Example Code of Definition Assign Expression:**

```
/* <definition-assign-expression> ::= <regular-defasn> | <array-indexing-
   defasn> */

/* <regular-defasn> ::= <variable-type> <identifier> := ... */

/* <array-indexing-defasn> ::= <array-indexing-expression> := ... */

class TestClass{
    int temp;
    constructor TestClass(int temp){}
}

int time2 (int a){
    return a * 2;
}

int main(){
    /* <variable-type> <identifier> : = <literal> */
    int i := 3;
    print(i);

    /* <variable-type> <identifier> : = <identifier> */
    int j := 5;
    int m := j;
    print(m);

    /* <variable-type> <identifier> : = <binop-expression> */
    int k := 2 + 3;
    print(k);

    /* <variable-type> <identifier> : = <unop-expression> */
    int y := - 3;
    print(y);

    /* <variable-type> <identifier> : = <funcall-expression> */
    int z := time2(2);
    print(z);

    /* <variable-type> <identifier> : = <new-funcall-expression> */
    TestClass obj1 := new TestClass(5);
    /* <variable-type> <identifier> : = <new-array-expression> */
    int[] arr1 := new int[3];

    /* <variable-type> <identifier> : = <access-expression> */
    int acc1 := obj1.temp;
    print(acc1);

    /* <array-indexing-expression> : = <literal> */
    arr1[0] := 3;
```

```
    print(arr1[0]);

    /* <array-indexing-expression> : = <identifier> */
    arr1[0] := k;
    print(arr1[0]);

    /* <array-indexing-expression> : = <binop-expression> */
    arr1[0] := 2 * 3;
    print(arr1[0]);

    /* <array-indexing-expression> : = <unop-expression> */
    arr1[1] := -3;
    print(arr1[1]);

    /* <array-indexing-expression> : = <funcall-expression> */
    arr1[1] := time2(3);
    print(arr1[1]);

    /* <array-indexing-expression> : = <new-funcall-expression> */
    TestClass[] arr2 := new TestClass[3];
    arr2[0] := new TestClass(7);
    print(arr2[0].temp);

    /* <array-indexing-expression> : = <array-indexing-expression> */
    arr1[2] := arr1[1];
    print(arr1[2]);

    /* <array-indexing-expression> : = <access-expression> */
    arr1[2] := obj1.temp;
    print(arr1[2]);
}
```

### 4.5.10 Precedence Grouping Expression

User can group expressions inside () to ensure operations grouped within the round brackets will be evaluated before other operations in the whole expression.

**Grammar of Precedence Grouping Expression:**

$\langle$*precedence-group-expression*$\rangle$ ::= ( $\langle$*expression*$\rangle$ )

### 4.5.11 Operator precedence

When users define an expression containing multiple operations, such as a + b * c, the operators will be grouped by precedence. In the example above, b * c will be grouped together, and the outcome of evaluating b * c will then be added to a. The list below will provide information on the precedence of different types of expressions, with the highest precedence on top and descending precedence to the bottom.

1. Function call, array indexing, object membership access

2. Unary operators: logical negation, unary negative
3. Multiplication and Division
4. Addition and Subtraction
5. Greater-than, Less-than, Greater-than-or-equal-to, Less-than-or-equal-to
6. Equality and Inequality
7. Logical AND
8. Logical OR
9. Conditional expressions: if, while, for
10. Assignment expression

## 4.6 Statements

Statements are usually used as control flows of a MicroJ program. A statement can also be a simple expression.

**Grammar of Statement:**

$\langle statement \rangle$ ::= $\langle expression\text{-}statement \rangle$
| $\langle block\text{-}statement \rangle$
| $\langle if\text{-}statement \rangle$
| $\langle for\text{-}statement \rangle$
| $\langle while\text{-}statement \rangle$
| $\langle control\text{-}flow\text{-}statement \rangle$
| $\langle return\text{-}statement \rangle$

Note that, *if-statement*, *for-statement*, *while-statement*, *control-flow-statement*, and *return-statement* will be introduced in this chapter.

### 4.6.1 Expression Statement

This subsection introduce a statement can be a single expression with a semi-colon.

**Grammar of Expression Statement:**

$\langle expression\text{-}statement \rangle$ ::= $\langle expression \rangle$;

### 4.6.2 Block Statement

It is a list of statements wrapped in {}. It is usually used in the body of if statement, for statement, and while statement.

**Grammar of Block Statement:**

$\langle$*block-statement*$\rangle$ ::= {[$\langle$*statement*$\rangle$+]}

### 4.6.3 If Statement

Users can use the if statement to execute a portion of the program conditionally.

**Grammar of If Statement:**

$\langle$*if-statement*$\rangle$ ::= if ( $\langle$*expression*$\rangle$ ) [$\langle$*statement*$\rangle$] [else $\langle$*statement*$\rangle$]

Notice that, expression here must be eventually evaluated to a bool. If condition *expression* evaluates to true, the *then statement* will be executed; otherwise, the *else statement* will be executed. Notice that *else statement* is optional; users can also choose to wrap *then-statement* and *else-statement* in {} to create a statement block for multiple *statements*.

**Example Code of If Statement:**

```
int main() {
    int x : = 4;
    if (x == 4) {
        print("x is 4");
    } else {
        x = 3;
        print("x is now 3");
    }
}
```

### 4.6.4 For Statement

The for statement is a loop statement in which the users can execute expressions by a specific number of times.

**Grammar of For Statement:**

$\langle$*for-statement*$\rangle$ ::= for ( [$\langle$*expression*$\rangle$]; $\langle$*expression*$\rangle$ ; [$\langle$*expression*$\rangle$] ) $\langle$*statement*$\rangle$

The *for statement* will first evaluate the first *expression*, which is the initial condition, then it will evaluate the second *expression*, which evaluates to a boolean type outcome. If the second *expression* evaluates to false, the for loop ends, and programs resume after the for loop; if the second *expression* evaluates to *true*, the *statement* will be executed, and then the last *expression* will be evaluated, and finally the loop starts with evaluate first *expression* again. Note that, the first *expression* and the third *expression* are optional.

**Example Code of For Statement:**

```
int main() {
    for (int x : = 0; x < 10; x = x + 1) {
        print(x);
    }
}
```

### 4.6.5 While Statement

The *while statement* is a loop statement with terminal condition at the beginning of the loop structure.

**Grammar of While Statement:**

$\langle$*while-statement*$\rangle$ ::= while ( $\langle$*expression*$\rangle$ ) $\langle$*statement*$\rangle$

The while statement will first evaluate the *expression*, which must return boolean type outcome. If the *expression* evaluates to true, the *statement* will then be executed, and the *expression* is evaluated again. If *expression* evaluates to false, the while loop will stop to execute and the program will resume after the *while statement*.

**Example Code of While Statement:**

```
int main() {
    int count : = 0;
    while (count < 10) {
        print(count);
        count = count + 1;
    }
}
```

### 4.6.6 Control Flow Statement

Users can use the *break statement* to terminate a *while* or *for statement*. The grammar of the *break statement* is trivial, and an actual example is more helpful. Users can use the *continue statement* to terminate an iteration of the while or for loop, and begin the next iteration. We use an actual code example to demonstrate the usage of *continue*.

**Grammar of Control Flow Statement:**

$\langle$*control-flow-statement*$\rangle$ ::= break;
| continue;

**Example Code of Control Flow Statement:**

```
int main() {
    for (int i : = 0; i < 5; i = i + 1) {
        if (i == 3) {
            break;
        }
```

```
        else print(i);
    }

    for (int i : = 0; i < 5; i = i + 1) {
        if (i == 3) {
            i = i + 1;
            continue;
        }
        else print(i);
    }
}
```

The example of *break* will print numbers from 0 to 2; when x reaches 3, x == 3 is true, and *break* is evaluated, which will terminate the for loop. The example of *continue* will print numbers from 0 to 5 except for 3; when x == 3; *continue* is evaluated and the loop begins the next iteration.

### 4.6.7 Return statement

Users can use the *return statement* to exit a function and return an expression.

**Grammar of Return Statement:**

$\langle \textit{return-statement} \rangle$ ::= return ;
| return $\langle \textit{expression} \rangle$;

Notice that *expression* here could be empty, depending on whether the function has a void return type.

## 4.7 Functions

Users can call a function by using the name of the function and giving it the parameters required by its function definition. A function call can be an expression or as an operand of an expression.

### 4.7.1 Function definitions

*Function definition* contains the following information: function's name, return type, parameters with their names and types, and the body of the function, which is the actual procedure encapsulated by the function. The function body is actually a *block statement*.

**Grammar of Function Definition:**

$\langle \textit{function-definition} \rangle$ ::= $\langle \textit{return-type} \rangle$ $\langle \textit{identifier} \rangle$ ( [$\langle \textit{parameter} \rangle$+]) $\langle \textit{statement} \rangle$

$\langle \textit{return-type} \rangle$ ::= $\langle \textit{variable-type} \rangle$ | void

$\langle \textit{parameter} \rangle$ ::= $\langle \textit{variable-type} \rangle$ $\langle \textit{identifier} \rangle$

**Example Code of a Simple Function Definition:**

```
int square(int x) {
    return x * x;
}
```

### 4.7.2 Function call

Users can call a function by using the name of the function and giving it the parameters required by its function definition. A *function call* can be an expression or its return type as an operand of an expression.

**Grammar of Function Definition:**

$\langle$*function-call*$\rangle$ ::= $\langle$*identifier*$\rangle$ ( [$\langle$*expression*$\rangle$+] )

**Example Code of a Simple Function Call:**

```
int square(int x) {
    return x * x;
}

int main() {
    square(3); /* This should return 9 */
}
```

Functions with multiple parameters need to be called with parameters that match the function definition, separated by a comma.

**Example Code of Function Call with Multiply Parameters:**

```
int mean (int x, int y) {
    return (x + y) / 2;
}

int main() {
    mean(3, 3); /* This should return 3 */
}
```

## 4.8 Interfaces

An *interface* is a collection of *abstract methods* that can be implemented by a class. An *interface* can also extend one or more *interfaces*. All methods in the *interface* are public, Users don't need to write access controllers explicitly.

### 4.8.1 Syntax of Interfaces

**Grammar of Interfaces:**

$\langle$*interface*$\rangle$ ::= interface $\langle$*identifier*$\rangle$ [extends $\langle$*identifier*$\rangle$ + ]
{ [$\langle$*abstract-method*$\rangle$]+ }

Note that, *abstract-method* are explained in 8.2.

**Example Code of an Interface:**

```
interface collegeStudent extends Student {
    /*Abstract Methods*/
    void setName(string name);
    void printName(string name);
}
int main(){}
```

### 4.8.2 Abstract Methods

The *abstract method* only contains the method signature without the implementation of the method. The method in interface doesn't need to have self as a parameter, but self parameter is necessary when *abstract methods* are implemented in a class.

**Grammar of Abstract Method:**

$\langle \textit{abstract-method} \rangle$ ::= $\langle \textit{return-type} \rangle$ $\langle \textit{identifier} \rangle$ ( [$\langle \textit{parameter} \rangle$+] ) ;

**Example Code of an Interface Definition with Abstract Methods:**

```
interface collegeStudent {
    /*Abstract Methods*/
    /* Don't need to include self parameter here */
    void setName(string name);

    /* Don't need to include self parameter here */
    int score();
}
int main(){}
```

### 4.8.3 Interface Inheritance

An *interface* can also extend one or more *interfaces*. The same method signatures will be merged.

**Example Code of Interface Inheritance:**

```
/*Test interface extension*/
interface Factory extends Window, Wheel, Name{

}

interface Window{
    void window();
}

interface Wheel{
    void wheel();
}

interface Name{
```

```
    void name();
}

int main(){}
```

## 4.9 Classes

A *class* is a blueprint or a template for creating objects that define the properties and behaviors of those objects. A *class* encapsulates the data and the methods that operate on that data into a single unit, providing a clean and modular way to structure a program.

### 4.9.1 Syntax of Classes

**Grammar of Class Definition:**

$$\langle \textit{class-syntax} \rangle ::= \textsf{class} \ \langle \textit{identifier} \rangle \ [\textsf{extends} \ \langle \textit{identifier} \rangle] \ [\textsf{implement} \ \langle \textit{identifier} \rangle +]$$
$$\{ \ \ [\langle \textit{class-body} \rangle \ +] \ \}$$

$$\langle \textit{class-body} \rangle ::= \langle \textit{field} \rangle$$
$$| \ \langle \textit{constructor} \rangle$$
$$| \ \langle \textit{method} \rangle$$

Note that, *field*, *constructor*, and *method* will be introduced later in this chapter.

**Example Code of a Simple Class Definition:**

```
class Point {
    int x;
    int y;
    constructor Point ( int x , int y ){}
    void moveX ( Point self , int distance ){
        self . x = self . x + distance ;
    }
    void moveY ( Point self , int distance ){
        self . y = self . y + distance ;
    }
    void pirntPoint(Point self){
        print(self.x);
        print(self.y);
    }
}
int main () {
    Point p : = new Point (0 , 0);
    p.pirntPoint();
}
```

### 4.9.2 Access modifiers

*Access modifiers* are keywords that determine the visibility and accessibility of fields, methods, and other members of a class. There are three access modifiers in MircoJ:
*public*: A *public* member can be accessed from anywhere, by any code.
*private*: A *private* member can be accessed within its own class and all its subclasses.

**Grammar of Access Modifier:**

$\langle access\text{-}modifier \rangle$ ::= public | private

### 4.9.3 Static keyword

A *static* method or field belongs to the class and not to any instance of the class. It can be called only by using the class name.

### 4.9.4 Fields

A *field* is a variable that belongs to a class or an object. It represents the state or data of the object.
**Grammar of Field:**

$\langle field \rangle$ ::= [$\langle access\text{-}modifier \rangle$] [static] $\langle identifier \rangle$;

Note that, the expression here should be a define and assign case in expression.

### 4.9.5 Methods

A *method* is a block of code that performs a specific task or action. It is similar to the function described before. But it is declared within the body of a class. It can also be modified by access modifiers and static keywords. In MicroJ, self is a conventional name that must be given as the first parameter of a non-static *method* within a class. The self parameter represents the instance of the class on which the *method* is called. When calling a *method* on a class type instance, MicroJ will automatically pass the instance as the first argument to the *method*, which is captured by the self parameter.

**Grammar of Method:**

$\langle method \rangle$ ::= [$\langle access\text{-}modifier \rangle$] [static] $\langle function\text{-}definition \rangle$

Note that the function-definition here refer to the content in chapter 6 Functions.

### 4.9.6 Constructors

*Constructors* are a unique type of method that does not have a return type and cannot be modified by static keywords or access modifiers. The purpose of a *constructor* is to assign initial values to the fields of an object. The body of the *constructor* should be empty, and the parameters of the *constructor* should have the same names as the class's fields it wants to initialize. It is required to define at least one *constructor* for a class. (Note that, the static string can not be initialized in a *constructor*. Depending on the LLVM version, for LLVM 14.00, non-static string cannot be initialized as well. On LLVM 16.0, non-static string can be initialized. There is no guarantee for consistent performance.)

**Grammar of Constructor:**

⟨*constructor*⟩ ::= constructor ⟨*identifier*⟩ ( [⟨*parameter*⟩+] ) ⟨*statement*⟩

**Example Code of a Complex Class Definition:**

```
class Animal {
    private static string species;
    public int age;
    double size;
    static string name;
    constructor Animal() {}
    constructor Animal(int age) {}
    /* the parameter has the same name as its field 'age */

    void eat(Animal self) {
        print("I eat meat");
    }
    static void sleep() {
        print("I'm sleeping");
    }
}

int main() {
    Animal lion : = new Animal(3);
    lion.eat();
    Animal.sleep();
    Animal.name = "King";
    print(Animal.name);
}
```

### 4.9.7 Class Inheritance

A *class* in MircoJ can extend only one superclass but can implement multiple interfaces. When a *class* implements an interface, it must provide implementations for all the methods specified by that interface.

**Example Code of Class and Interface Inheritance:**

```
class Coordinate{
    int x;
    int y;
    constructor Coordinate(int x, int y){}
    void printLocation(Coordinate self){
        print("X-cordinate:");
```

```
        print(self.x);
        print("Y-cordinate:");
        print(self.y);
    }
}

interface operation{
    void moveX(int distance);
    void moveY(int distance);
}

class Point extends Coordinate implements operation{
    constructor Point(int x, int y) {}

    public void moveX(Point self, int distance) {
        self.x = self.x + distance;
    }

    public void moveY(Point self, int distance) {
        self.y = self.y + distance;
    }

    public void printLocation(Point self){
        print("Point Location:");
    }
}

int main(){
    Coordinate zero : = new Point(0, 0);
    Coordinate orign : = new Coordinate(0,0);
    zero.moveX(3);

    zero.printLocation();
    /* The out put shoud be "Point Location:" */
    orign.printLocation();
    /* The out put should be:
    "X-cordinate:"
    0
    "Y-cordinate:"
    0
    */
}
```

### 4.9.8 Polymorphism

In MicroJ, polymorphism is supported, which means that you can assign a subclass instance to a superclass pointer. The method that will be called will depend on the instance that the superclass pointer is referring to at runtime.

**Example Code of Polymorphism:**

```
/*Test polymorphism*/
class Animal {
    constructor Animal(){}

    void printName(Animal self){}
}

class Cat extends Animal{
    constructor Cat(){}

    void printName(Cat self){
        print("Cat");
    }
}

class Bird extends Animal{
    constructor Bird(){}

    void printName(Bird self){
        print("Bird");
    }
}

class Dog extends Animal{
    constructor Dog(){}

    void printName(Dog self){
        print("Dog");
    }
}



int main() {
    Animal c : = new Cat();
    Animal b : = new Bird();
    Animal d : = new Dog();

    c.printName();  /* This should print Cat */
    b.printName();  /* This should print Bird */
    d.printName();  /* This should print Dog */
}
```

Notice that object arrays doesn't support polymorphism right now.

**Example Code of Failure Polymorphism:**

```
/*Test polymorphism*/
class Animal {
    constructor Animal(){}
```

```
    void printName(Animal self){

    }
}

class Cat extends Animal{
    constructor Cat(){}

    void printName(Cat self){
        print("Cat");
    }
}

class Bird extends Animal{
    constructor Bird(){}

    void printName(Bird self){
        print("Bird");
    }
}

class Dog extends Animal{
    constructor Dog(){}

    void printName(Dog self){
        print("Dog");
    }
}

int main() {
    Animal c := new Cat();
    Animal b := new Bird();
    Animal d := new Dog();

    Animal[] array := new Animal[3];
    array[0] := c;
    array[1] := b;
    array[2] := d;
    for(int i := 0; i < 3; i = i+1){
        array[i].printName();
    }
}
```

## 4.10 Appendix

Syntax Summary

### 4.10.1 Expressions

```
expression:
    primary
    unop expression
    expression binop expression
    lvalue = expression


primary:
    identifier
    literal
    ( expression )
    primary [ expression ]
    primary ( expression-list-opt )
    lvalue . identifier
    new type [ expression ]
    new primary ( expression-list-opt )


left-value:
    identifier
    primary [ expression ]
    left-value . identifier
    ( left-value )
```

The primary expression operators:

() [] .

have the highest priority and group left-to-right.
The unary operators:

\- !

have priority below the primary operator but higher than any binary operator, and group right-to-left.
The binary operators and the conditional are grouped left-to-right, and have priority decreasing as indicated:

**binop**:

* /

\+ -

> < >= <=

== !=

&& ||

Assignment operator has precedence below all the above operators, and it is grouped right to left:

asnop:

=

### 4.10.2 Declarations

```
declaration:
    decl-specifiers  declarator

decl-specifiers:
    access-control-option  field-modifier-option  type-specifier

type-specifier:
    int
    bool
    double
    string
    Object
    int[]
    double[]
    string[]
    Object[]


declarator:
    identifier
```

The access-control contains:

**public  private  protected**

Each declaration has one access-control flag. If the access-control is not specified, the default value is **public**.

The field-modifier contains:

**static**

If the field-modifier is not specified, the default value is flag is non-static.

### 4.10.3 Statements

```
statement:
    expression
    statement-list
    if (expression) statement
    if (expression) statement else statement
    for (expression-option; expression; expression option) statement
    while (expression) statement
    return  expression option
    control-flow

control-flow:
    break
    continue


statement-list:
    statement
    statement statement-list
```

### 4.10.4 Function Definition

function-definition:
    access-control-option **static**$_{option}$ type-specifier function-declarator function-body

function-declarator:
    declarator parameter-list$_{option}$

parameter-list:
    identifier
    identifier parameter-list

function-body:
    declaration-list$_{option}$ statement-list

declaration-list:
    declaration
    declaration declaration-list

### 4.10.5 Interface Definition

`interface-definition:`

  **interface** `identifier` `interface-inheritance`$_{option}$ **{** `interface-body` **}**

`declaration-list:`

  `declaration-list`$_{option}$ `statement-list`

`declaration-list:`

  `declaration`

  `declaration declaration-list`

### 4.10.6 Class Definition

`class-definition:`

  **class** `identifier` `class-inheritance`$_{option}$ **{** `class-body` **}**

`class-inheritance:`

  **extends** `identifier` **implements** `identifier`

`class-body:`

  `declaration-list`$_{option}$ `function-definition-list`$_{option}$ `constructor`

`constructor:`

  **constructor** `identifier` `parameter-list`$_{option}$ `statement-list`$_{option}$

# Chapter 5

# Project Plan

## 5.1 Project Timeline

| Time | Description |
|---|---|
| Jan. $24^{th}$ | We discussed the features that our compiler should have, and each group member conducted personal research on the feature(s) we selected to implement. |
| Jan. $29^{th}$ | We completed the proposal and decided on the features that our compiler will eventually have, which include class, inheritance, and arrays. |
| Feb. $8^{th}$ | We worked on the parser and scanner in MicroC and discussed the basic syntax of our language. |
| Feb. $10^{th}$ | We used the syntax discussed earlier to create the first draft of our compiler's abstract syntax tree (AST). |
| Feb. $16^{th}$ | We tested the complete version of the parser and discovered 72 conflicts. |
| Feb. $18^{th}$ | Debugged the conflicts in parser. Left 28 conflicts. |
| Feb. $20^{th}$ | Debugged the conflicts in parser. Left 2 conflicts. |
| Feb $22^{nd}$ | The last conflict was can not be fixed by adding precedence, fixed this one by using a new keyword. All conflicts were fixed. |
| Feb. $24^{th}$ | Discussed the structure of the LRM. |
| Feb. $25^{th}$ | First draft of LRM completed and reviewed. |
| Feb. $26^{th}$ | Final draft of LRM completed. |
| Mar. $4^{th}$ | Created the semantic analysis based on the structure of MicroC. Created the sast. |
| Mar. $12^{th}$ | Created two scans in semantic analysis. |
| Mar. $14^{th}$ | Went over the code-gen in MicroC as team and discussed the how to approach it in our compiler. |
| Mar. $17^{th}$ | Finished the semantic analysis for expression, and statement. |
| Mar. $19^{th}$ | Finished the semantic analysis for function and its return type. |
| Mar. $25^{th}$ | Finished the code-gen for part of expression and function.(Could define int, double, bool, and string; Also can use print function.) |
| Mar. $26^{th}$ | Completed the relevant test and shell script. |
| Mar. $27^{th}$ | Group meeting and reviewed the submission for Hello world. |

| | |
|---|---|
| Apr. 1st | Fixed the syntax in parser and the new syntax looks more like java and easy to understand. |
| Apr. 4th | Completed the statement in code-gen. |
| Apr. 8th | Completed the semantic analysis for class and interface. |
| Apr. 10th | Separated into two groups and implemented the break & continue and array features in code-gen. |
| Apr. 13th | Completed the part of the expression (customized function call, binop, and uop). |
| Apr. 14th | break & continue completed and tested. |
| Apr. 16th | Array feature completed and tested. |
| Apr. 16th | Started implementing Class in Codegen. Finished architecture design. |
| Apr. 17th | Finished class struct type. |
| Apr. 19th | Finished constructor. |
| Apr. 20th | Separated in two group, one group implemented the class in code-gen and the other group fixed the the LRM. |
| Apr. 20th | Extended class to support non-static field. In function expr, start implementing Access case to support accessing a class's non-static field. Tested this feature with simple testcases. |
| Apr.21st | Extended class to support static field. We found there were some design flaws in semantic analysis that caused huge pain for implementing static field. So, fixed semantic analysis first. |
| Apr. 22nd | Started writing the Final Report |
| Apr. 22nd | Finished static field. |
| Apr. 23rd | Extended class to support methods. However, our implementation didn't support a method to access its fields. |
| Apr. 24th | Tried using other ways to implement class methods so that it could access its fields. But, we failed. |
| Apr. 25th | After consulting the instructors, we finally decided to add 'self' keyword as a parameter into each non-static method in a class. Non-static class methods finished. In function expr, extend Access case to support accessing a class's non-static method. |
| Apr. 26th | Extended class to support static fields. In function expr, extended Access case to support accessing a class's static field. Tested this feature with simple test cases. |
| Apr. 27th | Extended class to support static methods. In function expr, extended Access case to support accessing a class's static method. Tested this feature with simple test cases. |
| Apr. 28th | Extended the class to support private keywords. This feature was handled by semantic analysis, but we again found some design flaws that made it hard to add this new feature. So, fixed semantic analysis first, and then implement private keyword. Tested this feature with simple test cases. |
| Apr. 29th | Finished implementing object array. |
| May 2nd | Group meeting, reviewed the entire project and report. |
| Apr. 30th - May 5th | Started stress testing and fixed bugs along the way. |

## 5.2 Contribution

### 5.2.1 Zichen Yang

**Role:** Developer, Project Manager

- Design the syntax of AST.
- Code in every aspect of the project.
- Code the important feature of class and inheritance.

### 5.2.2 Chenxuan Liu

**Role:** Project Manager

- Design the syntax of AST.
- Code in the ast, parser, scanner, sast and semant.
- Organize, and make plan for the next meeting.
- Write part of the final report.

### 5.2.3 Wei Sheng

**Role:** Designer

- Design the syntax of AST.
- Code in parser and code-gen.
- Add array feature and update the structure of parser.
- Deign the important feature of OOP (including class, inheritance and interface) and the implementation of array.
- Design the semantic analysis two scan structure.
- Write part of the final report.

### 5.2.4 Weishi Ding

**Role:** Quality Assurance (QA) Tester

- Design the syntax of AST.
- Coding the important feature of array in code-gen.
- Create stressed test for every new feature are created.
- Write the first draft of LRM.
- Write part of the final report.

## 5.3 Software Development Environment

- Libraries and Language: We used OCaml version 4.13.1, OCaml LLVM version 14.0.0, opam 2.1.2, and dune version 3.7.1.
- Version control: We didn't use GitHub as our version control tool. We save the latest version of each group coding in one member's personal computer. We only use GitHub to store every deliverable submission.
- Group Coding: We use VS Code studio live share plug-in as the main tool for meeting and coding together.
- Text Editor: VS Code.
- Operating System: Ubuntu 22.04.0 LTS, OS 13.3.1

# Chapter 6

# System Architecture and Design

## 6.1 Overall Architecture

The MicroJ compiler follows a four-stage process to compile source code into executable code. Each stage corresponds to a specific Ocaml program, as described in Figure 6.1:

- Scanning: The source code is processed by the scanner.mll program, which converts it into a token stream.
- Parsing: The parser.mly program then processes the token stream to generate an Abstract Syntax Tree (AST).
- Semantic analysis: The semant.ml program checks the AST for semantic errors and generates a Semantically-checked Abstract Syntax Tree (SAST).
- Code generation: The SAST is used by the code generation phase to create LLVM IR code (.ll file), which is compiled to assembly code (.s file) and then to a binary executable (.exe) using a C compiler.

Once the binary executable is generated, it can be executed by the operating system.

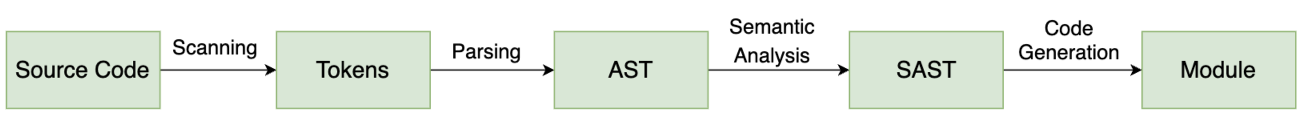


Figure 6.1: Compilation Pipeline

We utilize ocamllex to perform the scanning phase, and ocamlyacc to perform the semantic analysis phase, and we comply with standard practice in both scanning phase and parsing phase. Our creativity goes to the semantic analysis phase and code generation phase.

MicroJ has a unique feature that allows users to define classes, functions and interfaces in any order they wish. This differs from the C programming language, where users need to provide function declarations at the top of the C file if they want to use a function before its definition. With

MicroJ, users can use a function, class, or interface before the place where it's defined, giving MicroJ a more modern programming experience similar to Java and Python.

To achieve this flexibility, MicroJ utilizes a two-pass scanning strategy during the Semantic Analysis phase, shown in Figure 6.2. The first scan collects all classes, functions, and interfaces defined by the user and puts them in Class Map, Function Map, and Interface Map, respectively. With that information in hand, the second scan checks all class methods, globals, and function bodies, and generates SAST along the way.

For more details on the two scans, refer to section 6.2.3 Semantic Checker.

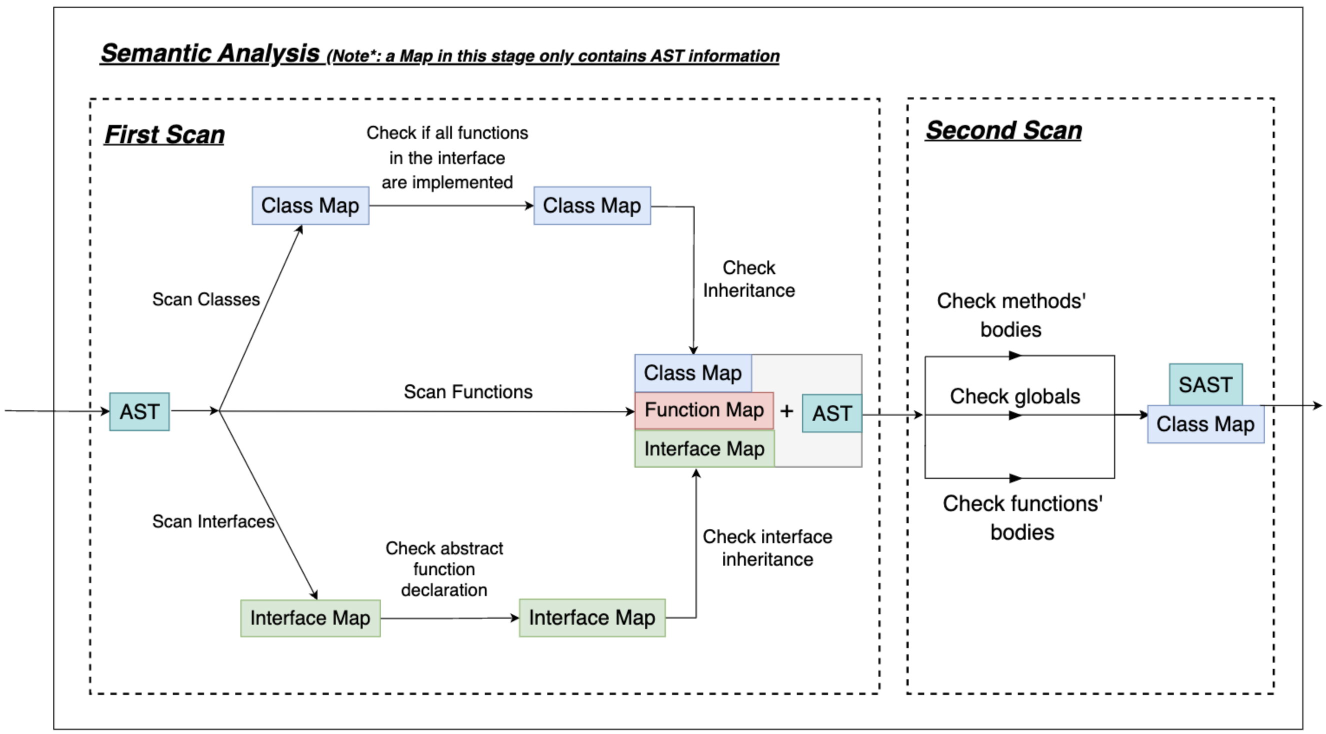


Figure 6.2: Architecture of Semantic Analysis Phase

We also utilize a two-pass scanning strategy in the code generation phase, as illustrated in Figure 6.3, where the first scan will construct Globals Map', Function Map', and Class Map'; in the second scan, those maps will be used for look up when building method bodies and function bodies. More detailed explanation of the two scans here is also discussed in 6.2.4 Code Generation.

## 6.2 Different Components of the Compiler

### 6.2.1 Scanner

The scanner (scanner.mll) receives MicroJ source code supplied by the user, and it produces tokenized output. In this step, whitespaces and comments (only multi-line comment is allowed in MicroJ) in the source code will not be tokenized. Input that does not correspond to valid token in MicroJ language will be rejected by the scanner.

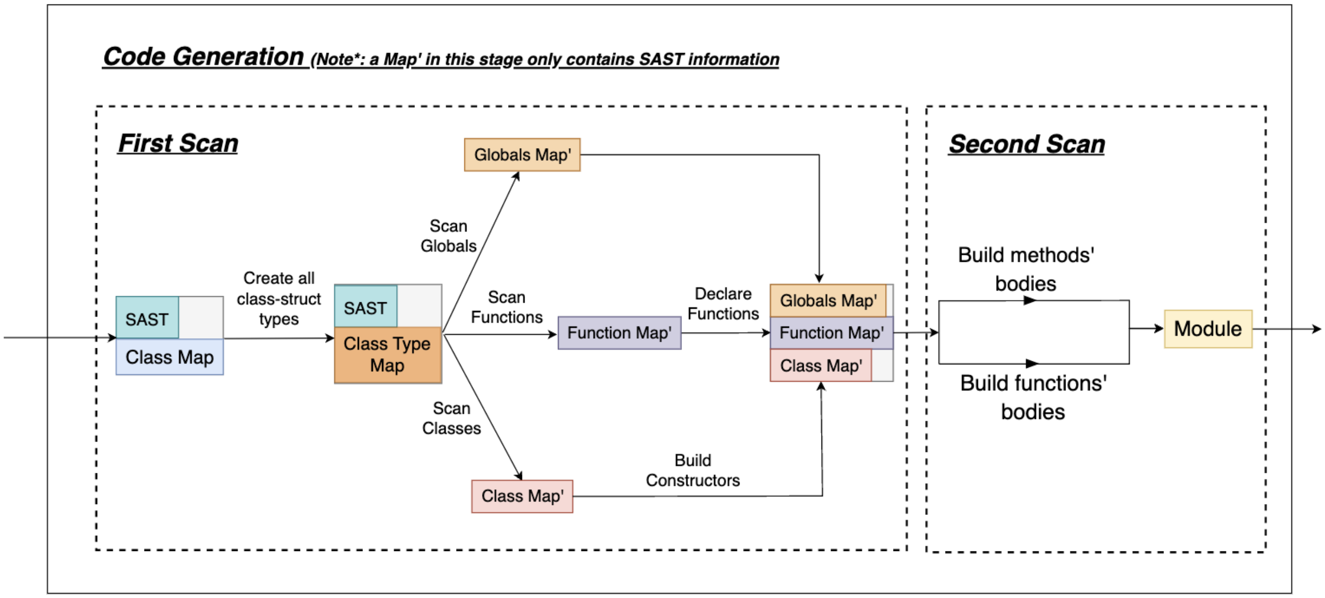


Figure 6.3: Architecture of Code Generation Phase

Author: Team.

### 6.2.2 Parser

The parser (parser.mll) receives the token stream and generates the AST. All components of the MicroJ AST are defined in file (ast.mll), and based on those definitions, the parser will produce the AST of the source program.

Author: Team.

### 6.2.3 Semantic Checker

The semantic checker (semant.ml) takes in an AST generated from the parser and generates a sementially-checked SAST for every node of the AST.

A MicroJ source program that successfully goes through this step is guaranteed to be syntactically and semantically right, which means that:

- All global and local variables have correct types according to their declaration, value assignment and usage
- All functions have the correct return type as declaration
- All classes have a least one constructor
- All classes have implemented all functions defined by the interfaces they implement
- All interfaces have valid abstract declarations

- Valid inheritance of interfaces
- Valid inheritance of classes

In the first scanning step, upon seeing a node representing a function in AST, the semantic checker will check whether there already exists the same function signature (function name and type of arguments) in the Function Map; if not, the function will be added to the Function map. Upon seeing a node representing a class in the AST, the semantic checker will first check if a class with the same name has been already defined. If not, it will then check whether all methods in the interfaces (could be none) this class implements have been defined in itself; if both conditions are satisfied, the semantic checker will put the class info (including fields, methods, constructors) into the Class Map. Upon seeing a node representing an Interface in the AST, the semantic checker will check whether there's an interface with the same name has been defined, and if all abstract function definitions in the interface body are valid (i.e. it is not allowed for two abstract functions to have the same function signature, which compose of function name, parameter list and return type); if both condition are met, the interface info (abstract function definitions) will be put into the Interface Map. Readers should note that Class Map, Function Map, and Interface Map in the semantic checker phase only contains AST information.

In the second scan, the semantic checker will check global variable definitions, class methods' bodies and functions' bodies. When checking global variables, the semantic checker will ensure its definition and assignment are of valid types; if a global variable passes the check, it will be built as a node in the SAST. When checking function bodies, the semantic checker will check whether all statements within the function body are semantically valid; it will also check if all variables used in the method body are within scope by checking arguments, globals and locals. If so, the function body will be built into SAST. When checking methods' bodies, the semantic checker will collect arguments of the method, fields of the current class, fields of the parent class, global variables and local variables, then, combined with the above information, utilize the semantic rules for function bodies to check the method body; if such check passes, it will generate a node in SAST representing a method.

Using the two-pass scanning strategy, the semantic checker will have collected all information of classes, interfaces, functions and globals that are scattered in the AST, and use it to build the at the second scan. This allows the user to have the freedom of defining classes, functions and interfaces in any order they want.

Author: Team.

### 6.2.4 Code Generator

The Code Generator (codegen.ml) is responsible for taking in an SAST from the semantic checker, and generating LLVM instructions. We also utilize a two-pass scanning strategy here, where the first scan is responsible for collecting information from the SAST and the Class Map generated from the previous phase, and also building constructors of each class, and the second scan is responsible for building the LLVM instructions for functions' bodies and class methods' bodies.

In the first scan, the code generator will first construct distinct LLVM struct types for all classes using the information in the Class Map generated from the semantic analysis, as each class type will have different struct size based on fields it has. Then, the code generator will collect information of global variables, functions and classes from the SAST, because which the function bodies and method bodies will rely on. The code generator will place global variables in the Globals Map', functions in the Function Map', and classes in the Class Map'. Users should notice a Map' in the code generation phase only contains SAST information. It's worth mentioning that interfaces are discarded after semantic analysis because an interface only contains abstract function definitions without a body, which means no LLVM instructions need to be generated for an interface.

In the second scan, using the maps just created, the code generator will iteratively build function bodies and method bodies and generate the module for execution.

Author: Team. In Particular, Zichen Yang designed the implementation strategy of our OOP feature, and contributed significantly to the OOP feature implementation.

# Chapter 7

# Test Plan

## 7.1 Unit test, integration test and Automation

### 7.1.1 Unit test

For each functionality of our language, we create a separate test file. We execute the code using ./LLouPut.sh and compare the output with our expected output to ensure that the functionality is working as expected.
If the program generates the expected output, we keep the test case and create a corresponding .out file to store the expected output. We perform both positive and negative tests, where positive tests are those where the program runs as expected, while negative tests are those where we expect the program to fail. Code of ./LLouPut.sh:

```
dune clean
dune build
_build/default/src/toplevel.exe -a "./test/$1" > "Result.ast"
_build/default/src/toplevel.exe -s "./test/$1" > "Result.sast"
_build/default/src/toplevel.exe -l "./test/$1" > "Result.ll"
llc -relocation-model=pic "Result.ll" > "Result.s"
cc -o "Result.exe" "Result.s"
"./Result.exe"
```

### 7.1.2 Integration test

We use ./testall.sh to execute all test cases we had written and compare with the output generated with the corresponding .out files.
Code of ./testall.sh:

```
# If a filename is passed as an argument, test only that file.
# Otherwise, test all files.
if [ $# -eq 1 ]; then
    files=("$1")
else
    files=("FailTest1" "FailTest2" "FailTest3" "FailTest4"
           "Test1" "Test2" "Test3" "Test4" "Test5" "Test6" "Test7" "Test8
              " "Test9" "Test10"
```

```
            "Test11" "Test12" "Test13" "Test14" "Test15" "Test16" "Test17"
                "Test18" "Test19" "Test20")
fi

# Test each file in the array.
for f in "${files[@]}"; do
    if [[ "$f" == FailTest* ]]; then
        expected_output="./test/$f.out"
        actual_output=$(dune exec -- ./src/toplevel.exe -l "./test/$f"
           2>&1 )
        if [[ "$actual_output" != "$(cat $expected_output)" ]]; then
            echo "Failed Test failed as expected: $f"
        else
            echo "Failed Test fail as expected: $f"
        fi
    elif [[ "$f" == Test* ]]; then
        expected_output="./test/$f.out"
        dune exec -- ./src/toplevel.exe -l "./test/$f" > "test.ll"
        llc -relocation-model=pic "test.ll" > "test.s"
        cc -o "test.exe" "test.s"
        actual_output=$(./test.exe)
        if [[ "$actual_output" != "$(cat $expected_output)" ]]; then
            echo "Test failed: $f"
        else
            echo "Test passed: $f"
        fi
    fi
done
```

Part of the output of the Integration test

```
./testall.sh
Failed Test failed as expected: FailTest1
Failed Test fail as expected: FailTest2
Failed Test failed as expected: FailTest3
Failed Test failed as expected: FailTest4
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test1
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test2
Done: 62% (34/54, 20 left) (jobs: 0)Test passed: Test3
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test4
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test5
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test6
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test7
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test8
Done: 69% (34/49, 15 left) (jobs: 0)Test passed: Test9
Done: 69% (34/49, 15 left) (jobs: 0)Test failed: Test10
```

### 7.1.3 Automation

Whenever we introduce a new feature, we use ./testall to execute all the test cases we have created. This is done to ensure that the newly released feature does not impact the existing ones.

## 7.2 Responsibilities

Chenxuan Liu: Wrote test code for testing lexing and parsing.
Wei Sheng: Wrote some test code for OOP features and expressions.
Weishi Ding: Wrote the shell script for testing and test code for statements.
Zichen Yang: Wrote some test code for OOP features.

## 7.3 Source Code and the LLVM code Generated

### 7.3.1 check static and access control of class

**Source Code**

```
class Father {
    public static int pubStat;
    private static int priStat;

    public int pub;
    private int pri;

    constructor Father(int pub, int pri, int pubStat, int priStat){

    }

    public static void publicStaticPrint(){
        print("public static");
    }

    private static void privatetStaticPrint(){
        print("private static");
    }

    public void publicPrint(Father self){
        print("public");
    }

    private void privatePrint(Father self){
        print("private");
    }
}


int main() {
    Father f : = new Father(1,2,3,4);
```

```
    print(Father.pubStat);
    print(f.pub);

    Father.pubStat = 10;
    f.pub = 20;
    print(Father.pubStat);
    print(f.pub);


    Father.publicStaticPrint();
    f.publicPrint();
}
```

**Generated LLVM code**

```
; ModuleID = 'MicroJ'
source_filename = "MicroJ"

@pubStat = global i64 0
@priStat = global i64 0
@my_struct_ptr = global { i64, i64, i64, i64 }* null
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.4 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.5 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str = private unnamed_addr constant [10 x i8] c"\22private\22\00", align
    1
@fmt.6 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.7 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.8 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.9 = private unnamed_addr constant [9 x i8] c"\22public\22\00", align
    1
@fmt.10 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.11 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.12 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.13 = private unnamed_addr constant [17 x i8] c"\22private static
   \22\00", align 1
@fmt.14 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.15 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.16 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.17 = private unnamed_addr constant [16 x i8] c"\22public static
   \22\00", align 1

declare i32 @printf(i8*, ...)

define { i64, i64, i64, i64 }* @Father(i64 %0, i64 %1, i64 %2, i64 %3) {
entry:
```

```
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i64, i64, i64, i64
     }* getelementptr ({ i64, i64, i64, i64 }, { i64, i64, i64, i64 }*
     null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to { i64, i64, i64, i64 }*
  store { i64, i64, i64, i64 }* %struct_ptr, { i64, i64, i64, i64 }**
     @my_struct_ptr, align 8
  %field_ptr = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64,
      i64, i64 }* %struct_ptr, i32 0, i32 0
  store i64 0, i64* %field_ptr, align 4
  %field_ptr1 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %struct_ptr, i32 0, i32 1
  store i64 0, i64* %field_ptr1, align 4
  %field_ptr2 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %struct_ptr, i32 0, i32 2
  store i64 0, i64* %field_ptr2, align 4
  %field_ptr3 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %struct_ptr, i32 0, i32 3
  store i64 0, i64* %field_ptr3, align 4
  %field_ptr4 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %struct_ptr, i32 0, i32 2
  store i64 %0, i64* %field_ptr4, align 4
  %field_ptr5 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %struct_ptr, i32 0, i32 0
  store i64 %1, i64* %field_ptr5, align 4
  store i64 %2, i64* @pubStat, align 4
  store i64 %3, i64* @priStat, align 4
  ret { i64, i64, i64, i64 }* %struct_ptr
}

define void @publicStaticPrint() {
entry:
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.16, i32 0, i32 0), i8* getelementptr inbounds
     ([16 x i8], [16 x i8]* @str.17, i32 0, i32 0))
  ret void
}

define void @privatetStaticPrint() {
entry:
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.12, i32 0, i32 0), i8* getelementptr inbounds
     ([17 x i8], [17 x i8]* @str.13, i32 0, i32 0))
  ret void
}

define void @publicPrint({ i64, i64, i64, i64 }* %self) {
entry:
  %self1 = alloca { i64, i64, i64, i64 }*, align 8
  store { i64, i64, i64, i64 }* %self, { i64, i64, i64, i64 }** %self1,
     align 8
```

```
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.8, i32 0, i32 0), i8* getelementptr inbounds ([9
      x i8], [9 x i8]* @str.9, i32 0, i32 0))
  ret void
}

define void @privatePrint({ i64, i64, i64, i64 }* %self) {
entry:
  %self1 = alloca { i64, i64, i64, i64 }*, align 8
  store { i64, i64, i64, i64 }* %self, { i64, i64, i64, i64 }** %self1,
     align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.5, i32 0, i32 0), i8* getelementptr inbounds
     ([10 x i8], [10 x i8]* @str, i32 0, i32 0))
  ret void
}

declare noalias i8* @malloc(i32)

define i64 @main() {
entry:
  %f = alloca { i64, i64, i64, i64 }*, align 8
  %Father_result = call { i64, i64, i64, i64 }* @Father(i64 1, i64 2, i64
      3, i64 4)
  store { i64, i64, i64, i64 }* %Father_result, { i64, i64, i64, i64 }**
     %f, align 8
  %static_field = load i64, i64* @pubStat, align 4
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %static_field)
  %f1 = load { i64, i64, i64, i64 }*, { i64, i64, i64, i64 }** %f, align
     8
  %field_ptr = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64,
      i64, i64 }* %f1, i32 0, i32 2
  %0 = load i64, i64* %field_ptr, align 4
  %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
      i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %0)
  store i64 10, i64* @pubStat, align 4
  %f3 = load { i64, i64, i64, i64 }*, { i64, i64, i64, i64 }** %f, align
     8
  %field_ptr4 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %f3, i32 0, i32 2
  store i64 20, i64* %field_ptr4, align 4
  %static_field5 = load i64, i64* @pubStat, align 4
  %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
      i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %static_field5)
  %f7 = load { i64, i64, i64, i64 }*, { i64, i64, i64, i64 }** %f, align
     8
  %field_ptr8 = getelementptr inbounds { i64, i64, i64, i64 }, { i64, i64
     , i64, i64 }* %f7, i32 0, i32 2
  %1 = load i64, i64* %field_ptr8, align 4
```

```
  %printf9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
      i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %1)
  call void @publicStaticPrint()
  %f10 = load { i64, i64, i64, i64 }*, { i64, i64, i64, i64 }** %f, align
      8
  call void @publicPrint({ i64, i64, i64, i64 }* %f10)
  ret i64 0
}
```

### 7.3.2 Test interface implementation and class extension

**Source Code**

```
interface Behaviour extends bark{
}

interface bark{
    void bark();
}

class Animal {
    int leg;

    constructor Animal(){

    }
}

class Dog extends Animal implements Behaviour{
    constructor Dog(int leg){

    }

    void bark(Dog self){
        print("wang wang");
    }

}

int main(){
    Dog d : = new Dog(4);

    print(d.leg);
    d.bark();
}
```

**Generated LLVM code**

```
; ModuleID = 'MicroJ'
source_filename = "MicroJ"

@my_struct_ptr = global { i64 }* null
@my_struct_ptr.1 = global { i64 }* null
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.5 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.6 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str = private unnamed_addr constant [12 x i8] c"\22wang wang\22\00",
   align 1

declare i32 @printf(i8*, ...)

define { i64 }* @Animal() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i64 }*
     getelementptr ({ i64 }, { i64 }* null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to { i64 }*
  store { i64 }* %struct_ptr, { i64 }** @my_struct_ptr, align 8
  %field_ptr = getelementptr inbounds { i64 }, { i64 }* %struct_ptr, i32
     0, i32 0
  store i64 0, i64* %field_ptr, align 4
  ret { i64 }* %struct_ptr
}

declare noalias i8* @malloc(i32)

define { i64 }* @Dog(i64 %0) {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i64 }*
     getelementptr ({ i64 }, { i64 }* null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to { i64 }*
  store { i64 }* %struct_ptr, { i64 }** @my_struct_ptr.1, align 8
  %field_ptr = getelementptr inbounds { i64 }, { i64 }* %struct_ptr, i32
     0, i32 0
  store i64 0, i64* %field_ptr, align 4
  %field_ptr1 = getelementptr inbounds { i64 }, { i64 }* %struct_ptr, i32
      0, i32 0
  store i64 %0, i64* %field_ptr1, align 4
  ret { i64 }* %struct_ptr
}

define void @bark({ i64 }* %self) {
entry:
  %self1 = alloca { i64 }*, align 8
  store { i64 }* %self, { i64 }** %self1, align 8
```

```
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.6, i32 0, i32 0), i8* getelementptr inbounds
     ([12 x i8], [12 x i8]* @str, i32 0, i32 0))
  ret void
}

define i64 @main() {
entry:
  %d = alloca { i64 }*, align 8
  %Dog_result = call { i64 }* @Dog(i64 4)
  store { i64 }* %Dog_result, { i64 }** %d, align 8
  %d1 = load { i64 }*, { i64 }** %d, align 8
  %field_ptr = getelementptr inbounds { i64 }, { i64 }* %d1, i32 0, i32 0
  %0 = load i64, i64* %field_ptr, align 4
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt, i32 0, i32 0), i64 %0)
  %d2 = load { i64 }*, { i64 }** %d, align 8
  call void @bark({ i64 }* %d2)
  ret i64 0
}
```

### 7.3.3 Test polymorphism

**Source Code**

```
class Animal {
    constructor Animal(){

    }

    void printName(Animal self){

    }
}

class Cat extends Animal{
    constructor Cat(){

    }

    void printName(Cat self){
        print("Cat");
    }
}

class Bird extends Animal{
    constructor Bird(){

    }
```

```
    void printName(Bird self){
        print("Bird");
    }
}

class Dog extends Animal{
    constructor Dog(){

    }

    void printName(Dog self){
        print("Dog");
    }
}



int main() {
    Animal c : = new Cat();
    Animal b : = new Bird();
    Animal d : = new Dog();

    c.printName();
    b.printName();
    d.printName();
}
```

**Generated LLVM code**

```
; ModuleID = 'MicroJ'
source_filename = "MicroJ"

@my_struct_ptr = global {}* null
@my_struct_ptr.2 = global {}* null
@my_struct_ptr.4 = global {}* null
@my_struct_ptr.6 = global {}* null
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.7 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.8 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.9 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.10 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.11 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.12 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.13 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.14 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str = private unnamed_addr constant [7 x i8] c"\22Bird\22\00", align 1
@fmt.15 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.16 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
```

```
@fmt.17 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.18 = private unnamed_addr constant [6 x i8] c"\22Cat\22\00", align 1
@fmt.19 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.20 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@fmt.21 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.22 = private unnamed_addr constant [6 x i8] c"\22Dog\22\00", align 1

declare i32 @printf(i8*, ...)

define {}* @Animal() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({}* getelementptr
     ({}, {}* null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to {}*
  store {}* %struct_ptr, {}** @my_struct_ptr, align 8
  ret {}* %struct_ptr
}

define void @printName({}* %self) {
entry:
  %self1 = alloca {}*, align 8
  store {}* %self, {}** %self1, align 8
  ret void
}

declare noalias i8* @malloc(i32)

define {}* @Cat() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({}* getelementptr
     ({}, {}* null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to {}*
  store {}* %struct_ptr, {}** @my_struct_ptr.2, align 8
  ret {}* %struct_ptr
}

define void @printName.1({}* %self) {
entry:
  %self1 = alloca {}*, align 8
  store {}* %self, {}** %self1, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.17, i32 0, i32 0), i8* getelementptr inbounds
     ([6 x i8], [6 x i8]* @str.18, i32 0, i32 0))
  ret void
}

define {}* @Bird() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({}* getelementptr
     ({}, {}* null, i32 1) to i32))
```

```
  %struct_ptr = bitcast i8* %malloccall to {}*
  store {}* %struct_ptr, {}** @my_struct_ptr.4, align 8
  ret {}* %struct_ptr
}

define void @printName.3({}* %self) {
entry:
  %self1 = alloca {}*, align 8
  store {}* %self, {}** %self1, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.14, i32 0, i32 0), i8* getelementptr inbounds
     ([7 x i8], [7 x i8]* @str, i32 0, i32 0))
  ret void
}

define {}* @Dog() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({}* getelementptr
     ({}, {}* null, i32 1) to i32))
  %struct_ptr = bitcast i8* %malloccall to {}*
  store {}* %struct_ptr, {}** @my_struct_ptr.6, align 8
  ret {}* %struct_ptr
}

define void @printName.5({}* %self) {
entry:
  %self1 = alloca {}*, align 8
  store {}* %self, {}** %self1, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
     i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds
     ([6 x i8], [6 x i8]* @str.22, i32 0, i32 0))
  ret void
}

define i64 @main() {
entry:
  %c = alloca {}*, align 8
  %Cat_result = call {}* @Cat()
  store {}* %Cat_result, {}** %c, align 8
  %b = alloca {}*, align 8
  %Bird_result = call {}* @Bird()
  store {}* %Bird_result, {}** %b, align 8
  %d = alloca {}*, align 8
  %Dog_result = call {}* @Dog()
  store {}* %Dog_result, {}** %d, align 8
  %c1 = load {}*, {}** %c, align 8
  call void @printName.1({}* %c1)
  %b2 = load {}*, {}** %b, align 8
  call void @printName.3({}* %b2)
  %d3 = load {}*, {}** %d, align 8
```

```
  call void @printName.5({}* %d3)
  ret i64 0
}
```

# Chapter 8

# Lessons Learnt

## 8.1 Zichen Yang

This project has given me a lot of experience in designing and implementing the architecture of a large project, where code written two months earlier needs to accommodate future changes. I have also gained valuable lessons on how to manage a team and a long-term project.

I am proud of our reuse of functions throughout the semantic checker and code generation. For instance, I reused the helper function of building a function body when building a class method; despite their differences, we use boolean optional flags to indicate whether we are building a class method or a function. This designing and implementing process is mentally challenging and takes a lot of effort, as I have to plan ahead for modules that have not yet been implemented, but it is great practice for anticipating future requirements. There are still some areas for improvement with this practice, as I attempted but failed to reuse the helper function of building the function body to build a class's constructor.

I contributed a lot to the AST and parser design, and I later found that some AST syntax were not appropriate and could be redesigned in a better way. But, it was too late to make this change, as too many places would need to be modified to accommodate for this simple change. Also, I also found that we had to add more parsing rules to support more features, but it actually introduced lots of conflicts in the parser, which were extremely hard to fix. We ended up with a really weird syntax in assignment expression that made MicroJ a little hard to use. If I would improve MicroJ afterwards, I would definitely strive to solve this parser problem.

Also, I gained a lot of experience with the map data structure in OCaml, which is the core data structure used throughout our semantic checker and code generation. I also learned and applied techniques to bypass functional language restrictions. For instance, to tackle immutable data restraint, we use ref to create mutable references.

Regarding project and team management, one of the hardest lessons we learned is that it is crucial to ensure the team is developing in a stable environment. We spent significant efforts trying to install LLVM natively, but all attempts failed, so we switched to using docker. However, we then switched to virtual machines running Ubuntu. Unfortunately, the virtual machine's stability is not trustworthy: on the afternoon of the due date of the extended test suite, one teammate's VM's disk image got corrupted, leading to catastrophic, non-recoverable data loss. The whole team then had to work around the clock to rebuild the progress we had made from our last archive. All of this could have been avoided if we had developed natively with docker.

Speaking of project pacing, the team tends to accumulate work before a deliverable due date, which causes stressful long hours of work and zero buffer time, especially with our language reference manual and final compiler. I learned a valuable lesson that the team should have a coherent sense of time management, and as the team manager, I should set up an internal timeline to allow more buffer time and avoid fatigue, which is crucial in this semester marathon.

## 8.2 Sheng Wei

Before taking this class, I had not studied Programming Languages. However, through hands-on project development, I learned functional programming, which was challenging yet enlightening. In addition, I gained knowledge about memory allocation during compile time and runtime when working with LLVM, as well as the information that can be known only during runtime and prior to runtime. This connection to Computation Theory, which I am also studying this semester, helped me understand the underlying concepts better.

During the project, I served as the language expert and gained a more in-depth understanding of the Object-Oriented Programming (OOP) paradigm. As we were modeling our language's behavior on Java's OOP design, I became more familiar with the intricacies of Java's OOP design. I worked on many corner cases of OOP, such as assigning an instance variable of a class the value of an instance's corresponding uninitialized value. Through this experience, I gained a better understanding of how OOP should function and how I can use OOP better in Java and C++.

While working on the project, I spent considerable time attempting to implement an array out-of-bound guard in the semantic analysis phase. However, I came to the realization that it is equivalent to solving the halting problem, which is a fundamental limit in computation theory. This experience allowed me to witness firsthand how theoretical challenges can manifest in real-world engineering problems.

## 8.3 Chenxuan Liu

Throughout the course of this project, I gained a deep understanding of the various phases involved in compilation, as well as how to implement a compiler step-by-step. By working hands-on with the scanner, parser, semantic checker, and code generator, I was able to develop a detailed understanding of the technical details of each component. This included everything from the process of tokenizing and parsing code, to generating an Abstract Syntax Tree (AST) and translating it into a Static Abstract Syntax Tree (SAST).

In particular, I gained valuable experience with debugging parsers, and became adept at locating and resolving shifts and reducing conflicts. I developed a systematic approach for resolving these issues that our team uses, starting by gradually removing rules until we could identify the problematic one. If changing precedence did not solve the problem, we would then attempt to modify the input token, such as having a different form for assignment operators when used in simple assignment versus definition and assignment.

I also became skilled at translating between tokens and parsing rules, which was particularly challenging given the complexity of our Object-Oriented Programming (OOP) language. Attention to detail was essential when working with the token stream and drafting parsing rules, and I learned to approach this task with a focus on invariants in order to ensure clarity around what each token represented, such as a type or a name.

## 8.4 Weishi Ding

There are several valuable lessons learned around the development of a large project.

Firstly, a consistent development environment is crucial for long-term success. I made the decision to switch the development environment from Docker to a Virtual Machine without considering the potential consequences, and my VM crashed on the due date of the extended test suite, causing the team to have to redo a lot of progress.

This leads to another learning lesson: version control and backup are a must-have for a long-term project. Our team rarely used git to perform version control. Although we set up a git repo, we quickly grew tired of having to set up branches, commit, push, and resolve conflicts, and we saved our progress locally, only pushing to git after achieving a major milestone. This practice proved to have almost catastrophic consequences when my VM crashed, and significant progress was lost. None of us had encountered such a dire situation, and this experience taught us the necessity of version control and backups.

Finally, as for technical lessons learned from doing the project, I gained a holistic view of how compilers work and how to use LLVM API in a functional setting, which is a valuable experience. I saw how functional programming could be limited when changing states are inevitable, and the scope of variables and functions quickly becomes less manageable. Taking this course right after Programming Language, where I once thought functional languages were the cure to all runtime bugs, I think this project experience made me truly understand the statement "there is no silver bullet in software engineering".

# Appendix A

# Appendix

## A.1 Translator

### A.1.1 toplevel.ml

```
(* Description: Top-level of the MicroJ compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR
      ,
   and dump the module
   Class: COMP107
   Author: Zichen Yang, Weishi Ding
   Date: 4-15-2023*)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist =
    [
      ("-a", Arg.Unit (set_action Ast), "Print the AST");
      ("-s", Arg.Unit (set_action Sast), "Print the SAST");
      ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR
         ");
      ( "-c",
        Arg.Unit (set_action Compile),
        "Check and print the generated LLVM IR (default)" );
    ]
  in
  let usage_msg = "usage: ./bin/toplevel.exe [-a|-s|-l|-c] ./test/[test.
     mj]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
     usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
```

```
  match !action with
  | Ast -> print_string (Ast.string_of_program ast)
  | _ -> (
      let (sast, classMap) = Semant.creatDefMaps ast in
      match !action with
      | Ast -> ()
      | Sast -> print_string (Sast.string_of_sprogram sast)
      | LLVM_IR ->
          print_string (Llvm.string_of_llmodule (Codegen.translate (sast,
              classMap)))
      | Compile ->
          let m = Codegen.translate (sast, classMap) in
          Llvm_analysis.assert_valid_module m;
          print_string (Llvm.string_of_llmodule m))
```

### A.1.2 scanner.mll

```
(* Description:  Ocamllex scanner for MicroJ
   Class: COMP107
   Author: Zichen Yang, Chenxuan Liu, Weishi Ding, Wei Shen
   Date: 4-25-2023 *)

{ open Parser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
(* | "//"     { comment lexbuf }  *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LSQUARE }
| ']'      { RSQUARE }
| ';'      { SEMI   }
| ':'      { COLUMN }
| ','      { COMMA  }
| '.'      { DOT    }

(* binOp *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
```

```
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }

(* unOp *)
| "!"      { NOT }

(* reserved keywords *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "public" { PUBLIC }
| "private" { PRIVATE }
| "protect" { PROTECT }
| "static" { STATIC }
| "null"   {NULL}
| "break" { BREAK }
| "continue" { CONTINUE }

(* reserved type keywords *)
| "int"    { INT }
| "bool"   { BOOL }
| "double"  { DOUBLE }
| "string" { STRING }
(* | "int[]"    { INTLIST }
| "bool[]"   { BOOLLIST }
| "double[]"  { DOUBLELIST }
| "string[]" { STRINGLIST } *)
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }

(* OOP reserved keywords *)
| "class" { CLASS }
| "interface" { INTERFACE }
| "constructor" {CONSTRUCTOR }
| "super"   { SUPER }
| "new" { NEW }
| "implements" { IMPLEMENTS }
| "extends" { EXTENDS }
| "is" { IS }
| "this" {THIS}

(* Literal *)
```

```
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* as lxm { DLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '\"' [^'\"']* '\"' as lxm  {STRINGLIT(lxm)}

(* end file and edge case *)
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
(* | '\n' { token lexbuf } *)
| eof  { token lexbuf }
| _    { comment lexbuf }
```

### A.1.3 parser.mly

```
(* Description:  Parser for MicroJ
   Class: COMP107
   Author: Zichen Yang, Chenxuan Liu
   Date: 5-3-2023 *)

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA PLUS MINUS
   TIMES DIVIDE ASSIGN DOT CONSTRUCTOR COLUMN

%token NOT EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL DOUBLE VOID STRING BREAK
   CONTINUE INTLIST BOOLLIST DOUBLELIST STRINGLIST
%token CLASS INTERFACE NEW IMPLEMENTS EXTENDS IS PUBLIC PRIVATE PROTECT
   STATIC THIS NULL SETDIMENSION SUPER
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID DLIT STRINGLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE DEF
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
```

```
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%left DOT
%nonassoc CLASSNAME


%%

program:
    programComp_list EOF { List.rev $1 }

programComp_list:
    /* nothing */    { [] }
    | programComp_list programComp { $2 :: $1 }

programComp:
    stmt {Stmt $1}
    // stmt_list    {Stmt (List.rev $1)}
  | fundef       {Fun  ($1)}
  | classdef     {Class($1)}
  | interfacedef {Interface($1)}

fundef:
    typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
        {{  ty = $1;
            id = $2;
            args = List.rev $4;
            body = List.rev $7}}

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

formal_list:
    typ ID                   { [($1,$2)]     }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

classdef:
    CLASS ID father_opt interface_opt LBRACE class_stmt_list RBRACE
        {{ id = $2;
           father =  $3;
           interface = $4;
           body = List.rev $6;}}

father_opt:
      EXTENDS ID  {Some $2}
    | {None}

interface_opt:
```

```
      IMPLEMENTS id_list {Some (List.rev $2)}
    | {None}

id_list:
      ID      {[$1]}
    | id_list COMMA ID  {$3 :: $1}

interfacedef:
    INTERFACE ID extend_mem_opt LBRACE absFundef_list RBRACE
        {{
           id = $2;
           extend_members = $3;
           body = List.rev $5;
        }}

extend_mem_opt:
    | EXTENDS id_list {Some (List.rev $2)}
    | {None}

absFundef_list:
      /* nothing */ {[]}
    | absFundef_list absFundef {$2 :: $1}

absFundef:
    fieldMod typ ID LPAREN formals_opt RPAREN SEMI
        {{
            fieldM = $1;
            ty = $2;
            id = $3;
            args = List.rev $5;
        }}

class_stmt_list:
    /* nothing */         {[]}
    | class_stmt_list class_stmt {$2 :: $1}

class_stmt:
      CONSTRUCTOR typ LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
       {ConstructorDef ($2, List.rev $4, List.rev $7)}
    | accControl fieldMod fundef {MethodDef ($1, $2, $3)}
    | accControl fieldMod typ ID def_stmt SEMI {FieldDef ($1, $2, $3, $4,
        $5)}


accControl:
      PUBLIC  {Some Public}
    | PRIVATE {Some Private}
    | PROTECT {Some Protect}
    | {None}
```

```
fieldMod:
      STATIC  {Some Static}
    | {None}
/*
empty_square_list:
      LSQUARE RSQUARE   {["[]"]}
    | empty_square_list LSQUARE RSQUARE {"[]" :: $1}
*/

list_type:
    INT LSQUARE RSQUARE %prec NOT{IntList }
  | BOOL LSQUARE RSQUARE  %prec NOT{BoolList }
  | DOUBLE LSQUARE RSQUARE  %prec NOT{DoubleList }
  | STRING LSQUARE RSQUARE  %prec NOT{StringList }
  | ID LSQUARE RSQUARE %prec CLASSNAME{ObjectList ($1)}

single_type:
    INT  {Int}
  | BOOL {Bool}
  | DOUBLE {Double}
  | STRING {String}
  | ID %prec CLASSNAME{Object $1}

typ:
    single_type {$1}
  | list_type {$1}
  | VOID {Void}


stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

controlFlow:
    BREAK    {Break}
  | CONTINUE {Continue}

stmt:
    expr SEMI                              { Expr $1}
  | RETURN expr_opt SEMI                   { Return $2}
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | LBRACE stmt_list RBRACE                { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                           { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt          { While($3, $5)         }
  | controlFlow SEMI                       { ControlFlow($1)}
```

```
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

/* expr that can be used as an index */
index_expr:
    LITERAL          { Literal($1)           }
  | ID     %prec NOELSE          { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr DOT expr    { Access ($1, $3)        }
  // | typ  DOT expr    { StaticAccess($1, $3)   }
  | funcall {$1}
  | MINUS expr %prec NOT { Unop(Neg, $2)      }


funcall:
   ID LPAREN args_opt RPAREN { Call($1, $3)  }


/* expr about define and assign */
def_asn_expr:
    expr ASSIGN expr   { Asn($1, $3)            }
  | typ ID def_stmt { DefAsn($1, $2, $3)}


expr:
    THIS             {This}
  | SUPER            {Super}
  | index_expr       {$1}
  | DLIT                   { Dliteral($1)          }
  | BLIT             { BoolLit($1)            }
  | STRINGLIT        { StringLiteral($1)      }
  | NULL             { Null }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq,   $3)   }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr LEQ    expr { Binop($1, Leq,   $3)   }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | NOT expr         { Unop(Not, $2)          }
  | def_asn_expr     { $1 }
  | LPAREN expr RPAREN {ParenExp $2                  }
  | NEW funcall         { NewExpr($2) }
  //| LSQUARE expr_list_opt RSQUARE {ListExpr ($2)}
  | NEW single_type LSQUARE expr RSQUARE {NewArray($2, $4)}
```

```
  | ID square_list def_stmt {Indexing($1, List.rev $2, $3)}


square_list:
   LSQUARE index_expr RSQUARE   {[$2]}
  | square_list LSQUARE index_expr RSQUARE {$3 :: $1}

/*
expr_list_opt:
    expr_list {Some $1}
    | {None}


expr_list:
      expr  {[$1]}
    | expr_list COMMA expr {$1 @ [$3]}
*/

/* optional define statement */
def_stmt:
    | COLUMN ASSIGN expr {Some $3}
    | {None}


args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }


args_list:
    expr                    { [$1] }
  | args_list COMMA expr { $3 :: $1 }
```

### A.1.4 ast.ml

```
(* Description:  Abstract Syntax Tree and its unpaser
   Class: COMP107
   Author: Zichen Yang, Chenxuan Liu, Wei Shen, Weishi Ding
   Date: 4-25-2023 *)
type op =
  | Add
  | Sub
  | Mult
  | Div
  | Equal
  | Neq
  | Less
  | Leq
  | Greater
```

```
  | Geq
  | And
  | Or

type uop = Neg | Not
type fieldModifier = Static
type accControl = Public | Private | Protect

type typ =
  | Int
  | Bool
  | Double
  | Void
  | String
  | Object of string
  | IntList
  | BoolList
  | DoubleList
  | StringList
  | ObjectList of string
(* IntList | BoolList | StringList| DoubleList *)

type controlFlow = Break | Continue
type bind = typ * string

type expr =
  | Literal of int
  | Dliteral of string
  | BoolLit of bool
  | StringLiteral of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Access of expr * expr
  (* | StaticAccess of typ * expr *)
  | ObjMethod of typ * string * expr list
  | DefAsn of typ * string * expr option
  | Asn of expr * expr
  | Call of string * expr list
  | ListExpr of expr list option
  | Indexing of string * expr list * expr option
  | ParenExp of expr
  | NewExpr of expr
  | NewArray of typ * expr
  | Null
  | This
  | Super
  | Noexpr

type stmt =
```

```
  | Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | ControlFlow of controlFlow
  | NoStmt

type fundef = {
  ty : typ;
  id : string;
  args : (typ * string) list;
  body : stmt list;
}

type classStmt =
  | ConstructorDef of typ * (typ * string) list * stmt list
  | FieldDef of
      accControl option * fieldModifier option * typ * string * expr
         option
  | MethodDef of accControl option * fieldModifier option * fundef

type classdef = {
  id : string;
  father : string option;
  interface : string list option;
  body : classStmt list;
}

type absFunDef = {
  fieldM : fieldModifier option;
  ty : typ;
  id : string;
  args : (typ * string) list;
}

type interfaceDef = {
  id : string;
  extend_members : string list option;
  body : absFunDef list;
}

type programComp =
  | Stmt of stmt
  | Fun of fundef
  | Class of classdef
  | Interface of interfaceDef

type program = programComp list
```

```
(* Unparser function *)
let string_of_op = function
  | Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function Neg -> "-" | Not -> "!"

let string_of_type = function
  | Int -> "int"
  | Bool -> "bool"
  | Double -> "double"
  | Void -> "void"
  | String -> "string"
  | Object s -> s
  (* | IntList -> "int[]"
     | DoubleList -> "double[]"
     | BoolList -> "bool[]"
     | StringList -> "string[]" *)
  | IntList -> "int []"
  | DoubleList -> "double []"
  | BoolList -> "bool []"
  | StringList -> "string []"
  | ObjectList s -> s ^ " []"

let rec string_of_expr = function
  | Literal l -> string_of_int l
  | Dliteral l -> l
  | BoolLit true -> "true"
  | BoolLit false -> "false"
  | StringLiteral l -> l
  | Id l -> l
  | Binop (e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop (o, e) -> string_of_uop o ^ string_of_expr e
  | Access (e1, e2) -> string_of_expr e1 ^ "." ^ string_of_expr e2
  (* | StaticAccess(t, e) -> "Static Access\n" *)
  | ObjMethod (t, meth, el) ->
      string_of_type t ^ "." ^ meth ^ "("
      ^ String.concat ", " (List.map string_of_expr el)
```

```
      ^ ")"
  (* int a = 1 *)
  (* | PreDefAsn(t, n, e) ->
     (match e with
       Some value -> string_of_type t ^ " " ^ n ^ " = " ^ string_of_expr
          value
     | None -> string_of_type t ^ " " ^ n) *)
  (* Dog a *)
  (* | ObjDef(t, n) -> string_of_type t ^ " " ^ n *)
  (* Dog a =  new Dog() *)
  (* | ObjDefAsn(t, n, e) -> string_of_type t ^ " " ^ n ^ " = new " ^
     string_of_expr e *)
  | Asn (e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  (* | ObjAsn(n, e) -> n ^ " = new " ^ string_of_expr e *)
  | Call (n, el) ->
      n ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | ListExpr el -> (
      match el with
      | Some lis -> "[" ^ String.concat ", " (List.map string_of_expr lis
         ) ^ "]"
      | None -> "[]")
  | Indexing (id, idx, exp) -> (
      id
      ^ String.concat ""
          (List.map (function s -> "[" ^ string_of_expr s ^ "]") idx)
      ^ match exp with Some e -> " = " ^ string_of_expr e | None -> "")
  (* | ObjListDef(classname, sql, varname) -> classname ^ String.concat
     "" sql ^ " " ^ varname *)
  | ParenExp e -> "(" ^ string_of_expr e ^ ")"
  (* | SetDim (es) -> "setDim(" ^ String.concat ", " (List.map
     string_of_expr es) ^ ")" *)
  | NewExpr e -> "new " ^ string_of_expr e
  | DefAsn (ty, id, e) -> (
      string_of_type ty ^ " " ^ id
      ^ match e with Some e' -> " = " ^ string_of_expr e' | None -> "")
  | This -> "this"
  | Null -> "null"
  | Super -> "super"
  | Noexpr -> ""
  | NewArray (typ, expr) ->
      "new " ^ string_of_type typ ^ "[" ^ string_of_expr expr ^ "]"

let rec string_of_stmt = function
  | Block stmts ->
      "{\n    " ^ String.concat "\n    " (List.map string_of_stmt stmts)
         ^ "\n}"
  | Expr expr -> string_of_expr expr ^ ";"
  | Return expr -> "return " ^ string_of_expr expr ^ ";"
  | If (e, s, Block []) -> "if (" ^ string_of_expr e ^ ") " ^
     string_of_stmt s
```

```
  | If (e, s1, s2) ->
      "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ " else "
      ^ string_of_stmt s2 ^ "\n    "
  | For (e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; "
      ^ string_of_expr e3 ^ ") " ^ string_of_stmt s
  | While (e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt
     s
  | NoStmt -> ""
  | ControlFlow Break -> "break;"
  | ControlFlow Continue -> "continue;"

let string_of_fundef (fd : fundef) =
  string_of_type fd.ty ^ " " ^ fd.id ^ "("
  ^ String.concat ", "
      (List.map (function t, s -> string_of_type t ^ " " ^ s) fd.args)
  ^ ") {\n    "
  ^ String.concat "\n    " (List.map string_of_stmt fd.body)
  ^ "\n}\n"

let string_of_ac = function
  | Some Public -> "public"
  | Some Private -> "private"
  | Some Protect -> "protect"
  | None -> ""

let string_of_fm = function Some Static -> "static" | None -> ""

let string_of_classStmt = function
  | ConstructorDef (s, bind1, st_list) ->
      "constructor " ^ string_of_type s ^ "("
      ^ String.concat ", "
          (List.map (function t, s -> string_of_type t ^ " " ^ s) bind1)
      ^ ") {\n    "
      ^ String.concat "\n" (List.map string_of_stmt st_list)
      ^ ";\n}"
  | FieldDef (ac, fm, t, s, e) -> (
      match e with
      | Some exp ->
          string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^ string_of_type
             t ^ " "
          ^ s ^ " = " ^ string_of_expr exp ^ ";"
      | None ->
          string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^ string_of_type
             t ^ " "
          ^ s ^ ";")
  | MethodDef (ac, fm, fd) ->
      string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^ string_of_fundef fd
```

```
let string_of_father = function Some value -> "extends " ^ value | None
   -> ""

let string_of_abs_interface = function
  | Some value -> "extends " ^ String.concat ", " value
  | None -> ""

let string_of_class_interface = function
  | Some value -> "implements " ^ String.concat ", " value
  | None -> ""

let string_of_classdef (cd : classdef) =
  "class " ^ cd.id ^ " " ^ string_of_father cd.father ^ " "
  ^ string_of_class_interface cd.interface
  ^ " {\n    "
  ^ String.concat "\n    " (List.map string_of_classStmt cd.body)
  ^ "\n}\n"

let string_of_absfundef absfundef =
  string_of_fm absfundef.fieldM
  ^ " "
  ^ string_of_type absfundef.ty
  ^ " " ^ absfundef.id ^ "("
  ^ String.concat ";\n"
      (List.map (function t, s -> string_of_type t ^ " " ^ s) absfundef.
         args)
  ^ ");"

let string_of_interfacedef interfacedef =
  "interface " ^ interfacedef.id ^ " "
  ^ string_of_abs_interface interfacedef.extend_members
  ^ " {\n    "
  ^ String.concat "\n   " (List.map string_of_absfundef interfacedef.body
     )
  ^ "\n}"

let string_of_programcomp = function
  | Stmt s -> string_of_stmt s
  | Fun f -> string_of_fundef f
  | Class c -> string_of_classdef c
  | Interface i -> string_of_interfacedef i

let string_of_program program =
  String.concat "\n" (List.map string_of_programcomp program)
```

### A.1.5 semant.ml

```
(* Description:  Semantic Analysis of MicroJ
   Class: COMP107
```

```
   Author: Zichen Yang, Chenxuan Liu, Wei Shen
   Date: 5-4-2023 *)

open Sast
open Ast

let compare_typ t1 t2 =
  match (t1, t2) with
  | Int, Int -> 0
  | Bool, Bool -> 0
  | Double, Double -> 0
  | Void, Void -> 0
  | String, String -> 0
  | Object s1, Object s2 -> String.compare s1 s2
  | _ -> compare t1 t2

let rec listcompare cmp l1 l2 =
  match (l1, l2) with
  | [], [] -> 0
  | [], _ :: _ -> -1
  | _ :: _, [] -> 1
  | x1 :: l1', x2 :: l2' -> (
      match cmp x1 x2 with 0 -> listcompare cmp l1' l2' | res -> res)

module FunSig = struct
  type t = string * typ list

  let compare (s1, ts1) (s2, ts2) =
    match String.compare s1 s2 with
    | 0 -> listcompare compare_typ ts1 ts2
    | c -> c
end

module FunSigMap = Map.Make (FunSig)
module StringMap = Map.Make (String)

exception Exception of string

(************************** check class def below
   ***********************************)
let get_formals_type args =
  List.rev (List.fold_left (fun accu (typ, _id) -> typ :: accu) [] args)

(*add function signiture to the funMap*)
let check_funDef funMap = function
  | Fun fundef ->
      let key : FunSig.t = (fundef.id, get_formals_type fundef.args) in
      if FunSigMap.mem key funMap then
        raise (Exception ("Function " ^ fundef.id ^ " cannot be redefined
            ."))
```

```
      else
        let funMap = FunSigMap.add key fundef funMap in
        funMap
  | _ -> funMap

let get_accControl e =
  match e with
  | Public -> "public"
  | Private -> "private"
  | Protect -> "protect"

(*creat field map for a given class*)
let check_class_field (fieldmap,sfieldmap, methodmap, smethodmap,
   constructormap) input =
  match input with
  (* typ and e is unchecked here*)
  | FieldDef (acc_opt, f_opt, typ, id, _e) ->
      let accflag =
        match acc_opt with Some e -> get_accControl e | None -> "public"
      in

      (match f_opt with Some Static -> (* if static, check sfieldmap*)
        if StringMap.mem id sfieldmap then
            raise (Exception ("Field " ^ id ^ " cannot be redefined"))
        else
          let fieldmap' = StringMap.add id (accflag, "static", typ)
             fieldmap in
          let sfieldmap' = StringMap.add id (accflag, "static", typ)
             sfieldmap in
            (fieldmap',sfieldmap', methodmap, smethodmap, constructormap)
      | None ->                   (* if nonstatic, check fieldmap *) (*
         actually both nonstatic and static are in fieldmap; sfieldmap
         only has static*)
        if StringMap.mem id fieldmap then
          raise (Exception ("Field " ^ id ^ " cannot be redefined"))
        else
          let fieldmap' = StringMap.add id (accflag, "nonstatic", typ)
             fieldmap in
            (fieldmap',sfieldmap, methodmap, smethodmap, constructormap))
  | _ -> raise (Exception "Expect a FieldDef, but someting else is
     provided")

(*creat method map for a given class*)
let check_class_method (fieldmap,sfieldmap, methodmap, smethodmap,
   constructormap) input =
  match input with
  (* typ and e is unchecked here*)
  | MethodDef (acc_opt, f_opt, funDef) ->
      let accflag =
        match acc_opt with Some e -> get_accControl e | None -> "public"
```

```
      in
      let fflag = match f_opt with Some _ -> "static" | None -> "
         nonstatic" in
      let type_list = get_formals_type funDef.args in
      let _ = if List.length type_list = 0 && fflag = "nonstatic" then
         raise (Exception ("Need parameter 'self' in the method '" ^
         funDef.id ^ "'")) in
        (match fflag with "nonstatic" ->     (* if nonstatic, then check
           in methodmap*)
          let key : FunSig.t =  (funDef.id, List.tl type_list) in
            if FunSigMap.mem key methodmap then
              raise
                (Exception
                  ("Method " ^ funDef.id ^ " ("
                  ^ String.concat ", "
                      (List.map string_of_type (get_formals_type funDef.
                         args))
                  ^ ") cannot be redefined"))
            else
              let methodmap' = FunSigMap.add key (accflag, fflag, funDef)
                  methodmap in
                (fieldmap,sfieldmap, methodmap', smethodmap,
                   constructormap)
        | "static" ->                        (* if static, then check in
           smethodmap*)
            let key : FunSig.t =  (funDef.id, type_list) in
            if FunSigMap.mem key smethodmap then
              raise
                (Exception
                  ("Method " ^ funDef.id ^ " ("
                  ^ String.concat ", "
                      (List.map string_of_type (get_formals_type funDef.
                         args))
                  ^ ") cannot be redefined"))
            else
              let methodmap' = FunSigMap.add key (accflag, fflag, funDef)
                  methodmap in
              let smethodmap' = FunSigMap.add key (accflag, fflag, funDef
                 ) smethodmap in
                (fieldmap,sfieldmap, methodmap', smethodmap',
                   constructormap)
        | _ -> raise (Exception ("Impossible field modifier '" ^ fflag ^
           "'")))
  | _ -> raise (Exception "Expect a MethodDef, but someting else is
     provided")

(*creat constructor map for a given class*)
let check_class_constructor (fieldmap,sfieldmap, methodmap, smethodmap,
   constructormap) input =
  match input with
```

```
  (* typ and e is unchecked here*)
  | ConstructorDef (typ, args, body) ->
      let key : FunSig.t = (string_of_type typ, get_formals_type args) in
      if FunSigMap.mem key constructormap then
        raise
          (Exception ("Method " ^ string_of_type typ ^ " cannot be
             redefined"))
      else
        let funDef : fundef = { ty = typ; id = string_of_type typ; args;
           body } in
        let constructormap' = FunSigMap.add key ("public", "nonstatic",
           funDef) constructormap in
          (fieldmap,sfieldmap, methodmap, smethodmap, constructormap')
  | _ ->
      raise (Exception "Expect a ConstructorDef, but someting else is
         provided")

(*creat filed, counstructor map, method map for a class body*)
let check_class_body body =
  let check_one (fieldmap,sfieldmap, methodmap, smethodmap,
     constructormap) e =
    match e with
    | ConstructorDef _ ->
        check_class_constructor (fieldmap,sfieldmap, methodmap,
           smethodmap, constructormap) e
    | FieldDef _ -> check_class_field (fieldmap,sfieldmap, methodmap,
       smethodmap, constructormap) e
    | MethodDef _ -> check_class_method (fieldmap,sfieldmap, methodmap,
       smethodmap, constructormap) e
  in
  List.fold_left check_one
    (StringMap.empty, StringMap.empty, FunSigMap.empty, FunSigMap.empty,
       FunSigMap.empty)
    body

(*creat all class's map, key: className :: value:(fieldMap, methodMap,
   constructorMap, classDef)*)
let check_classDef classMap = function
  | Class cdf ->
      (* let () = print_string ("Define class " ^ cdf.id ^ "\n") in *)
      if StringMap.mem cdf.id classMap then
        raise (Exception ("Class " ^ cdf.id ^ " cannot be redefined."))
      else
        let (fieldmap,sfieldmap, methodmap, smethodmap, constructormap) =
            check_class_body cdf.body in
        let classMap =
          StringMap.add cdf.id
          (fieldmap,sfieldmap, methodmap, smethodmap, constructormap, cdf
             )
            classMap
```

```
        in
        if FunSigMap.cardinal constructormap < 1 then
          raise (Exception ("No constructor has been defined for " ^ cdf.
             id))
        else classMap
  | _ -> classMap

(************************** check interface def below
   ************************************)
(*creat absMethodMap for given interface*)
let check_interface_absFunDef map (absfundef : absFunDef) =
  let key : FunSig.t = (absfundef.id, get_formals_type absfundef.args) in
  if FunSigMap.mem key map then
    raise
      (Exception ("Abstract function " ^ absfundef.id ^ " cannot be
         redefined."))
  else
    let fflag =
      match absfundef.fieldM with Some _ -> "static" | None -> "nonstatic
         "
    in
    FunSigMap.add key ("public", fflag, absfundef) map

(*creat absMethodMap for a interfaceBody*)
let check_interface_body body =
  List.fold_left check_interface_absFunDef FunSigMap.empty body

(*creat all interface's map. key: interfaceName :: value absMethodMap*)
let check_interfaceDef interfaceMap = function
  | Interface interdf ->
      if StringMap.mem interdf.id interfaceMap then
        raise (Exception ("Interface " ^ interdf.id ^ " cannot be
           redefined."))
      else
        let map = check_interface_body interdf.body in
        let interfaceMap =
          StringMap.add interdf.id (map, interdf) interfaceMap
        in
        interfaceMap
  | _ -> interfaceMap

(********************this is a wapper to check class, interface and
   global functions**********************)
(* create interface map, function map, class map for program *)
let creatDefMaps program =
  let check_one (classMap, funMap, interfaceMap) e =
    match e with
    | Class _ ->
        let classMap' = check_classDef classMap e in
        (classMap', funMap, interfaceMap)
```

```
    | Fun _ ->
        let funMap' = check_funDef funMap e in
        (classMap, funMap', interfaceMap)
    | Interface _ ->
        let interfaceMap' = check_interfaceDef interfaceMap e in
        (classMap, funMap, interfaceMap')
    | _ -> (classMap, funMap, interfaceMap)
  in
  let classMap, funMap, interfaceMap =
    List.fold_left check_one
      (StringMap.empty, FunSigMap.empty, StringMap.empty)
      program
  in
  let _ = if (FunSigMap.mem ("main", []) funMap) then () else raise (
     Exception ("No main() function defined")) in
  (************************* check class extends class, implements
     interface ***********************************)
  (*check if super class exist*)
  let check_class_father classdef =
    match classdef.father with
    | Some father -> StringMap.mem father classMap
    | None -> true
  in

  (********* check if the class has implemented all methods in the
     interface it implements*********)
  let rec check_if_absMethods_implemented (classdef : classdef)
      (interfacedef : string) =
    if not (StringMap.mem interfacedef interfaceMap) then
      raise (Exception ("Interfaces" ^ " of " ^ classdef.id ^ " undefined
         "))
    else
      let _,_, mMap,_, _, _ = StringMap.find classdef.id classMap
      and absMap, interfaceDef = StringMap.find interfacedef interfaceMap
          in
      let () =
          FunSigMap.iter
            (fun key _value ->
              if not (FunSigMap.mem key mMap) then
                raise
                  (Exception
                    ("Function '" ^ fst key ^ "' in Interface '" ^
                       interfacedef
                    ^ "' is not defined in Class '" ^ classdef.id ^ "'"))
              else
                let md, s, _ = FunSigMap.find key mMap in
                if not (md = "public" && s = "nonstatic") then
                  raise
                    (Exception
                      (fst key ^ " of interface " ^ interfacedef
```

```
                    ^ " is not defined in class " ^ classdef.id)))
          absMap
    in (* also need to check if interfaces it extends are implemented
       in this class*)
      (match interfaceDef.extend_members  with
        None -> ()
      | Some fathers -> List.iter (check_if_absMethods_implemented
         classdef) fathers)

in
(*check if interface that extended exist*)
let check_class_interface classdef =
  match classdef.interface with
  | Some interfaces ->
      List.iter (check_if_absMethods_implemented classdef) interfaces
  | None -> ()
in
let check_class_inheritance _ (_,_,_, _, _, classdef) =
  let father = match classdef.father with Some name -> name | None ->
     "" in
  if not (check_class_father classdef) then
    raise
      (Exception ("Super class" ^ father ^ " of " ^ classdef.id ^ "
         undefined"))
  else check_class_interface classdef
in
let () = StringMap.iter check_class_inheritance classMap in

(* check if all father is defined, and return unit*)

(************************** check interface extends interface
   ***********************************)
(*check if interface that extended exist*)
let check_interface_interface (interfaceDef : interfaceDef) =
  match interfaceDef.extend_members with
  | Some interfaces ->
      List.for_all
        (fun interface -> StringMap.mem interface interfaceMap)
        interfaces
  | None -> true
in
let check_interface_inheritance _ (_, interfaceDef) =
  if not (check_interface_interface interfaceDef) then
    raise (Exception ("Interfaces" ^ " of " ^ interfaceDef.id ^ "
       undefined"))
in
let () = StringMap.iter check_interface_inheritance interfaceMap in

(************* Add built-in function into the funMap ***************)
let add_builtin_fundef map
```

```
    ((return_type : typ), (fname : string), (formal_list_t : typ list))
        =
  let argsList = List.map (fun typ -> (typ, "x")) formal_list_t in
  FunSigMap.add (fname, formal_list_t)
    { ty = return_type; id = fname; args = argsList; body = [] }
    map
in
let funMap =
  List.fold_left add_builtin_fundef funMap
    [
      (Void, "print", [ Int ]);
      (Void, "print", [ Bool ]);
      (Void, "print", [ Double ]);
      (Void, "print", [ String ]);
      (Void, "print", [ Object "Animal" ]);
      (* only for testing*)
      (Void, "print", [ IntList ]);
      (String, "charAt", [ String; Int ]);
    ]
in
(* let () = print_int (FunSigMap.cardinal funMap) in  *)
(************* Now it's time to produce sast ***************)
(************** Declare a global variable map*****************)
let globalvarMap = ref StringMap.empty in
let add_funsig map acc =
  FunSigMap.fold (fun key value acum -> FunSigMap.add key value acum)
     map acc
in
(* add all entry of map ino acc*)
(* build a global constructor map by iterating through the classMap,
   find all consMap, and integrate them into a global consMap*)
let constructorMap =
  StringMap.fold
    (fun _key (_,_,_, _, consM, _) acc -> add_funsig consM acc)
    classMap FunSigMap.empty
in
let type_of_identifier s map =
  (* this function is to find the type of a variable, raise NotFound*)
  try StringMap.find s !map
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
in
let rec check_assign lvaluet rvaluet err =
  (*this function checks if type1 = type2*)
  match lvaluet with
  | Object s1 -> (
      match rvaluet with
      | Object s2 -> (
          let _, _,_, _, _, cdf =
            try StringMap.find s2 classMap
            with Not_found -> raise (Failure "Unknown Class instance")
```

```
          in
          if cdf.id = s1 then Void
          else
            match cdf.father with
            | Some super ->
                let rt = Object super in
                check_assign lvaluet rt err
            | None ->
              (* let _ = raise (Exception (s2)) in *)
                raise
                  (Failure
                     ("Right side object instance is not " ^ s1 ^ "
                        subclass"))
          )
      | _ ->
          raise (Failure "Assigning non object to a variable of object
             type"))
  | _ -> if lvaluet = rvaluet then Void else raise (Failure err)
in
let elemTy ty =
  match ty with
  | IntList -> Int
  | BoolList -> Bool
  | DoubleList -> Double
  | StringList -> String
  | ObjectList s -> Object s
  | _ -> raise (Exception "Not an array of proper type")
in
let rec expr e map (inclass : bool)=
  match e with
  | Literal l -> (Int, SLiteral l)
  | Dliteral s -> (Double, SDliteral s)
  | BoolLit b -> (Bool, SBoolLit b)
  | StringLiteral s -> (String, SStringLiteral s)
  | Id s -> (type_of_identifier s map, SId s) (******possible bug
     ******)
  | Asn (e1, e2) as ex ->
      let lt, e1' = expr e1 map inclass and rt, e2' = expr e2 map
         inclass in
      let err =
        "illegal assignment " ^ string_of_type lt ^ " = " ^
           string_of_type rt
        ^ " in " ^ string_of_expr ex
      in
      (check_assign lt rt err, SAsn ((rt, e1'), (rt, e2')))
  | Binop (e1, op, e2) as e ->
      let t1, e1' = expr e1 map inclass and t2, e2' = expr e2 map
         inclass in
      let same = t1 = t2 in
      let ty =
```

```
        match op with
        | (Add | Sub | Mult | Div) when same && t1 = Int -> Int
        | (Add | Sub | Mult | Div) when same && t1 = Double -> Double
        | (Equal | Neq) when same -> Bool
        | (Less | Leq | Greater | Geq) when same && (t1 = Int || t1 =
           Double)
          ->
            Bool
        | (And | Or) when same && t1 = Bool -> Bool
        | _ ->
            raise
              (Failure
                 ("illegal binary operator " ^ string_of_type t1 ^ " "
                ^ string_of_op op ^ " " ^ string_of_type t2 ^ " in "
                ^ string_of_expr e))
      in
      (ty, SBinop ((t1, e1'), op, (t2, e2')))
  | Unop (op, e) as ex ->
      let t, e' = expr e map inclass in
      let ty =
        match op with
        | Neg when t = Int || t = Double -> t
        | Not when t = Bool -> Bool
        | _ ->
            raise
              (Failure
                 ("illegal unary operator " ^ string_of_uop op
                ^ string_of_type t ^ " in " ^ string_of_expr ex))
      in
      (ty, SUnop (op, (t, e')))
  | Access (e1, e2) -> (
      let rec find_field (key : string) cname =
        (*recursive function to find a field *)
        let fieldM', _,_,_, _, cdf = StringMap.find cname classMap in
        try StringMap.find key fieldM'
        with Not_found -> (
          match cdf.father with
          | Some fname -> find_field key fname
          | None -> raise (Failure ("Field " ^ key ^ " is undefined")))
      in
      let rec find_method key cname =
        (*recursive function to find a method*)
        let _,_, methodM', _,_, cdf = StringMap.find cname classMap in
        try FunSigMap.find key methodM'
        with Not_found -> (
          match cdf.father with
          | Some fname -> find_method key fname
          | None -> raise (Failure ("Function " ^ fst key ^ " is
             undefinedddd")))
      in
```

```
        let field_or_method e cname vname static =
          match e with
          | Id fieldname ->
              let accflag, fflag, ty = find_field fieldname cname in
              if inclass = false then
                if static = true then
                  if accflag = "public" && fflag = "static" then
                    (ty, SAccess ((Object cname, SId vname), (ty, SId
                       fieldname)))
                  else raise (Failure ("Field " ^ fieldname ^ " is
                     undefineddd"))
                else if accflag = "public" && fflag = "nonstatic" then
                  (ty, SAccess ((Object cname, SId vname), (ty, SId
                     fieldname)))
                else raise (Failure ("Field " ^ fieldname ^ " is
                   undefined1"))
              else
                if static = true then
                  if fflag = "static" then
                    (ty, SAccess ((Object cname, SId vname), (ty, SId
                       fieldname)))
                  else raise (Failure ("Field " ^ fieldname ^ " is
                     undefined"))
                else if fflag = "nonstatic" then
                  (ty, SAccess ((Object cname, SId vname), (ty, SId
                     fieldname)))
                else raise (Failure ("Field " ^ fieldname ^ " is
                   undefined"))

          | Call (funcname, args) ->
              let args_types =
                List.rev
                  (List.fold_left
                     (fun acc e ->
                        let t', _e' = expr e map inclass in
                        t' :: acc)
                     [] args)
              in
              let key = (funcname, args_types) in
              let accflag, fflag, fdf = find_method key cname in
              if inclass = false then
                if static = true then
                  if accflag = "public" && fflag = "static" then
                    (fdf.ty, SAccess( (Object cname, SId vname),
                            (fdf.ty, SCall (funcname, List.rev (List.
                               fold_left
                                                                (fun acc e
                                                                   -> expr
                                                                   e map
                                                                   inclass
```

```
                                                                :: acc)
                                                            [] args))))
                                                                )
            else raise (Failure ("Function " ^ fst key ^ " is
               undefined"))
          else if accflag = "public" && fflag = "nonstatic" then
              (fdf.ty, SAccess ( (Object cname, SId vname),
                      (fdf.ty, SCall (funcname, List.rev
                                                  (List.fold_left
                                                    (fun acc e ->
                                                       expr e map
                                                       inclass :: acc
                                                       )
                                                    [] args)))))
          else raise (Failure ("Function " ^ fst key ^ " is
             undefined"))
        else (* if not in class, then everything can be access*)
          if static = true then
            if fflag = "static" then
              (fdf.ty, SAccess( (Object cname, SId vname),
                      (fdf.ty, SCall (funcname, List.rev (List.
                         fold_left
                                                            (fun acc e
                                                               -> expr
                                                               e map
                                                               inclass
                                                               :: acc)
                                                            [] args))))
                                                               )
            else raise (Failure ("Function " ^ fst key ^ " is
               undefined"))
          else if fflag = "nonstatic" then
              (fdf.ty, SAccess ( (Object cname, SId vname),
                      (fdf.ty, SCall (funcname, List.rev
                                                  (List.fold_left
                                                    (fun acc e ->
                                                       expr e map
                                                       inclass :: acc
                                                       )
                                                    [] args)))))
          else raise (Failure ("Function " ^ fst key ^ " is
             undefined"))
    | Indexing (fieldname, _, _) ->
         let accflag, fflag, ty = find_field fieldname cname in
        if inclass = false then
          if static = true then
            if accflag = "public" && fflag = "static" then
              let _ = map := StringMap.add fieldname ty !map in
              let ty',sexpr' = expr e map inclass in
```

```
                (ty', SAccess ((Object cname, SId vname), (ty',
                   sexpr')))
              else raise (Failure ("Field " ^ fieldname ^ " is
                 undefined"))
            else if accflag = "public" && fflag = "nonstatic" then
              let _ = map := StringMap.add fieldname ty !map in
              let ty',sexpr' = expr e map inclass in
                (ty', SAccess ((Object cname, SId vname), (ty',sexpr
                   ')))
            else raise (Failure ("Field " ^ fieldname ^ " is
               undefined1"))
          else
            if static = true then
              if fflag = "static" then
                let _ = map := StringMap.add fieldname ty !map in
                let ty',sexpr' = expr e map inclass in
                  (ty', SAccess ((Object cname, SId vname), (ty',
                     sexpr')))
              else raise (Failure ("Field " ^ fieldname ^ " is
                 undefined"))
            else if fflag = "nonstatic" then
              let _ = map := StringMap.add fieldname ty !map in
              let ty',sexpr' = expr e map inclass in
                  (ty', SAccess ((Object cname, SId vname), (ty',
                     sexpr')))
            else raise (Failure ("Field " ^ fieldname ^ " is
               undefined"))
      | _ -> raise (Failure "Expect Id/Call on right-hand side of
         Access")
    in
    match e1 with
    | Id n ->
        if StringMap.mem n classMap then
          (*if it's a class name rather than a var name*)
          field_or_method e2 n n true
        else
          let ty, _sexp = expr e1 map inclass in
          field_or_method e2 (string_of_type ty) n false
    | Indexing(id, _e1', None) ->
        let ty, sexp = expr e1 map inclass in
          (match field_or_method e2 (string_of_type ty) id false with
            (ty', SAccess(_, sexp2)) -> (ty', SAccess((ty, sexp),
               sexp2))
          | _ -> raise (Failure "Expect an object type on left-hand
             side of Access"))
    | Access(_, _) ->
        let ty, sexp = expr e1 map inclass in
          (match field_or_method e2 (string_of_type ty) "" false with
            (ty', SAccess(_, sexp2)) -> (ty', SAccess((ty, sexp),
               sexp2))
```

```
          | _ -> raise (Failure "Expect an object type on left-hand
             side of Access"))
          (* (ty, SAccess(expr e3 map inclass, expr e4 map inclass))
             *)
    | _ -> raise (Failure "Expect an object type on left-hand side of
        Access"))
| DefAsn (t, n, e) -> (
    (* cannot check duplicates because a local variable can have the
       same name as a global var*)
    match e with
    | Some e' ->
        let t1, e1 = expr e' map inclass in
        let err =
          "illegal assignment " ^ string_of_type t ^ " = "
          ^ string_of_type t1 ^ " in " ^ string_of_expr e'
        in
        let rt =
          try check_assign t t1 err
          with Failure _ ->
            raise (Failure (string_of_type t ^ string_of_type t1 ^  "
               DefAsn two sides doesn't match"))
        in
        if rt = Void then (
          (* equal to check_assign() *) (*type is determined bt the
             type on the right handside*)
          map := StringMap.add n t1 !map;
          (* print_int (StringMap.cardinal !globalvarMap); *)
          (Void, SDefAsn (t1, n, (t1, e1))))
        else
          raise
            (Exception
               ("Expect " ^ string_of_type t ^ " but provide "
              ^ string_of_type t1))
    | None ->
        map := StringMap.add n t !map;
        (Void, SDefAsn (t, n, (t, SNull))))
| Call (name, args) ->
    let checked_args =
      List.rev
        (List.fold_left
           (fun acc e ->
             let t', _e' = expr e map inclass in
             t' :: acc)
           [] args)
    in
    let key = (name, checked_args) in
    if FunSigMap.mem key funMap then
      let fundef = FunSigMap.find key funMap in
      ( fundef.ty,
        SCall
```

```
            ( name,
              List.rev
                (List.fold_left (fun acc e -> expr e map inclass :: acc
                   ) [] args) ) )
      (* in raise (Exception (List.iter (fun arg -> match arg with
         SId _ -> raise (Exception "SId case") | _ -> raise (
         Exception "Failed")) argslis))  *)
    else raise (Exception ("Function " ^ name ^ "() is undefined."))
| NewExpr e -> (
    match e with
    | Call (name, args) ->
        let checked_args =
          List.rev
            (List.fold_left
               (fun acc e ->
                 let t', _e' = expr e map inclass in
                 t' :: acc)
               [] args)
        in
        let key = (name, checked_args) in
        if FunSigMap.mem key constructorMap then
          let _, _, fundef = FunSigMap.find key constructorMap in
          ( fundef.ty,
            SNewExpr
              ( fundef.ty,
                SCall
                  ( name,
                    List.rev
                      (List.fold_left
                         (fun acc e -> expr e map inclass :: acc)
                         [] args) ) ) )
        else raise (Exception ("Counstructor " ^ name ^ "() is
           undefined."))
    | _ -> raise (Exception "Function call is required after NEW
       keyword."))
| Null -> (Void, SNull)
| Super -> (Void, SSuper)
| NewArray (ty, e) -> (
    let t', e' = expr e map inclass in
    match t' with
    | Int -> (
        match ty with
        | Int -> (IntList, SNewArray (ty, (t', e')))
        | Bool -> (BoolList, SNewArray (ty, (t', e')))
        | Double -> (DoubleList, SNewArray (ty, (t', e')))
        | String -> (StringList, SNewArray (ty, (t', e')))
        | Object s -> (ObjectList s, SNewArray (ty, (t', e')))
        | _ as s ->
            raise (Exception ("There is no array of" ^ string_of_type
                s)))
```

```
      | _ -> raise (Exception "Index must be integer."))
  | Noexpr -> (Void, SNoexpr)
  | Indexing (id, expl, expo) -> (
      match expl with
      | exp :: _exps -> (
          let t, e = expr exp map inclass and idtype, _ = expr (Id id)
             map inclass in
          match t with
          | Int -> (
              match expo with
              | Some expo' ->
                  let t1' = elemTy idtype and t2', _e2' = expr expo'
                     map inclass in
                  (* let _ = raise (Exception (string_of_type t2')) in
                     *)
                  ( check_assign t1' t2'
                      "Incompatible assignment of list element.",
                    SIndexing (id, (t, e), Some (expr expo' map inclass
                      )) )
              | None -> (elemTy idtype, SIndexing (id, (t, e), None)))
          | _ -> raise (Exception "Index must be integer."))
      | _ -> raise (Exception "n dimension array not allowed"))
  | ParenExp e -> expr e map inclass
  | _ -> raise (Exception "wait for implementation")
in

(*****************This function is used to check if an expression
   produce a boolean value ***********)
let check_bool_expr e map inclass =
  let t', e' = expr e map inclass
  and err = "expected Boolean expression in " ^ string_of_expr e in
  if t' != Bool then raise (Failure err) else (t', e')
in
(*****************This function is used to check statement excluding
   Return ***********)
let rec check_stmt e map (fundef : fundef) (inclass : bool) =
  match e with
  | Expr e -> SExpr (expr e map inclass)
  | If (p, b1, b2) ->
      SIf
        ( check_bool_expr p map inclass,
          check_stmt b1 map fundef inclass,
          check_stmt b2 map fundef inclass)
  | For (e1, e2, e3, st) ->
      let res = expr e1 map inclass in
      SFor
        ( res ,
          check_bool_expr e2 map inclass,
          expr e3 map inclass,
          check_stmt st map fundef inclass)
```

```
    | While (p, s) -> SWhile (check_bool_expr p map inclass, check_stmt s
        map fundef inclass)
    | Block sl ->
        let rec check_stmt_list es map fundef inclass=
          match es with
          | [ (Return _ as s) ] -> [ check_stmt s map fundef inclass]
          | Return _ :: _ -> raise (Failure "nothing may follow a return
             ")
          | Block sl :: ss ->
              check_stmt_list (sl @ ss) map fundef inclass(* Flatten
                 blocks *)
          | s :: ss -> check_stmt s map fundef inclass :: check_stmt_list
              ss map fundef inclass
          | [] -> []
        in
        SBlock (check_stmt_list sl map fundef inclass)
    | ControlFlow cf -> SControlFlow cf
    | NoStmt -> SNoStmt
    | Return ex ->
        let t, e' = expr ex map inclass in
        if t = fundef.ty then SReturn (t, e')
        else
          raise
            (Failure
               ("return gives " ^ string_of_type t ^ " expected "
              ^ string_of_type fundef.ty ^ " in " ^ string_of_expr ex))
in

(****************Check global statements***********)
let check_global_stmt = function
    | Return _ -> raise (Exception "Cannot not return outside of a
       function.")
    | _ as s ->
        let empty_fundef = { ty = Void; id = "NIL"; args = []; body = []
           } in
        SStmt (check_stmt s globalvarMap empty_fundef false)
in

let check_binds (kind : string) (to_check : bind list) =
  let name_compare (_, n1) (_, n2) = compare n1 n2 in
  let check_it checked binding =
    let void_err = "illegal void " ^ kind ^ " " ^ snd binding
    and dup_err = "duplicate " ^ kind ^ " " ^ snd binding in
    match binding with
    (* No void bindings *)
    | Void, _ -> raise (Failure void_err)
    | _, n1 -> (
        match checked with
        (* No duplicate bindings *)
        | (_, n2) :: _ when n1 = n2 -> raise (Failure dup_err)
```

```
        | _ -> binding :: checked)
  in

  let _ = List.fold_left check_it [] (List.sort name_compare to_check)
     in
  to_check
in
(**********check function body************)
let check_function_implement (fundef : fundef) symbols (inclass : bool)
   =
  let funbody =
    List.rev
      (List.fold_left
         (fun acc e -> check_stmt e symbols fundef inclass :: acc)
         [] fundef.body)
  in
  (* let () = print_int (StringMap.cardinal !symbols) in *)
  (* let () = print_int (List.length funbody) in *)
  let fundef' : sfundef =
    { ty = fundef.ty; id = fundef.id; args = fundef.args; body =
       funbody }
  in
  SFun fundef'
in
(******** check a class's methods, constructor and field ********)
let check_class_implement (cdf : classdef) =
  let class_stmts = cdf.body
  and fMap, _, _,_,_, _ = StringMap.find cdf.id classMap
  and globals' = !globalvarMap in
  let check_body_implement = function
    | MethodDef (acc, fm, fdf) -> (
        let args' = check_binds "local" fdf.args in
        let symbols_wt_f =
          List.fold_left
            (fun m (ty, name) -> StringMap.add name ty m)
            globals' args'
        in
        let symbols_w_f =
          StringMap.fold
            (fun key (_, _, ty) acc -> StringMap.add key ty acc)
            fMap symbols_wt_f
        in
        let symbols =
          match cdf.father with
          | None -> ref symbols_w_f
          | Some name ->
              let fMap, _,_,_, _, _ = StringMap.find name classMap in
              ref
                (StringMap.fold
                   (fun key (_, _, ty) acc -> StringMap.add key ty acc)
```

```
                 fMap symbols_w_f)
      in
      let res = check_function_implement fdf symbols true in
      match res with
      | SFun sfdf -> SMethodDef (acc, fm, sfdf)
      | _ -> raise (Exception ("Errors in " ^ fdf.id ^ "'s body")))
  | ConstructorDef (t, binds, slist) -> (
      if
        not (string_of_type t = cdf.id)
        (**** check if constructor name is identical to class name
           ****)
      then
        raise
          (Exception
             ("Constructor name is " ^ string_of_type t ^ ", but
                expect "
            ^ cdf.id))
      else
        let (fdf : fundef) =
          { ty = t; id = string_of_type t; args = binds; body = slist
            }
        in
        let args' = check_binds "local" fdf.args in
        let symbols_wt_f =
          List.fold_left
            (fun m (ty, name) -> StringMap.add name ty m)
            globals' args'
        in
        let symbols =
          ref
            (StringMap.fold
               (fun key (_, _, ty) acc -> StringMap.add key ty acc)
               fMap symbols_wt_f)
        in
        let res = check_function_implement fdf symbols true in
        match res with
        | SFun sfdf -> SConstructorDef sfdf
        | _ -> raise (Exception "ConstructorDef Error"))
  | FieldDef (acc, fmd, t, name, e) ->
      let symbols_w_f =
        StringMap.fold
          (fun key (_, _, ty) acc -> StringMap.add key ty acc)
          fMap globals'
      in
      let symbols =
        match cdf.father with
        | None -> ref symbols_w_f
        | Some name ->
            let fMap, _, _,_,_, _ = StringMap.find name classMap in
            ref
```

```
                  (StringMap.fold
                     (fun key (_, _, ty) acc -> StringMap.add key ty acc)
                     fMap symbols_w_f)
          in
          SFieldDef (acc, fmd, t, name, expr (DefAsn (t, name, e))
             symbols true)
    in

    let body' = List.map check_body_implement class_stmts in
    let (scdf : sclassdef) =
      {
        id = cdf.id;
        father = cdf.father;
        interface = cdf.interface;
        body = body';
      }
    in
    SClass scdf
  in
  let check_all = function
    | Stmt e -> check_global_stmt e
    | Fun fundef ->
        let globals' = !globalvarMap in
        let args' = check_binds "local" fundef.args in
        let symbols =
          ref
            (List.fold_left
               (fun m (ty, name) -> StringMap.add name ty m)
               globals' args')
        in
        check_function_implement fundef symbols false
    | Class classdef -> check_class_implement classdef
    | Interface inter -> SInterface inter
  in
  (List.map check_all program, classMap)
```

### A.1.6 sast.ml

```
(* Description:  Semantically-checked Abstract Syntax Tree and its
   unparser
   Class: COMP107
   Author: Zichen Yang, Chenxuan Liu, Wei Shen
   Date: 4-25-2023 *)
open Ast

type sexpr = typ * sx

and sx =
  | SLiteral of int
```

```
  | SDliteral of string
  | SBoolLit of bool
  | SStringLiteral of string
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  (* | SObjListDef of string * string list * string *)
  | SAccess of sexpr * sexpr
  | SObjMethod of typ * string * sexpr list
  (* | SObjDef of typ * string *)
  | SDefAsn of typ * string * sexpr
  (* | SPreDefAsn of typ * string * expr option *)
  (* | SObjDefAsn of typ * string * expr *)
  | SAsn of sexpr * sexpr
  (* | SObjAsn of string * expr *)
  | SCall of string * sexpr list
  | SListExpr of sexpr list option
  | SIndexing of string * sexpr * sexpr option
  | SParenExp of sexpr
  (* | SSetDim of expr list *)
  | SNewExpr of sexpr
  | SNewArray of typ * sexpr
  | SNull
  | SThis
  | SSuper
  | SNoexpr

type sstmt =
  | SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SControlFlow of controlFlow
  | SNoStmt

type sfundef = {
  ty : typ;
  id : string;
  args : (typ * string) list;
  body : sstmt list;
}

type sclassStmt =
  | SConstructorDef of sfundef
  | SFieldDef of accControl option * fieldModifier option * typ * string
     * sexpr
  | SMethodDef of accControl option * fieldModifier option * sfundef
```

```
type sclassdef = {
  id : string;
  father : string option;
  interface : string list option;
  body : sclassStmt list;
}

type sabsFunDef = {
  fieldM : fieldModifier option;
  ty : typ;
  id : string;
  args : (typ * string) list;
}

type sinterfaceDef = {
  id : string;
  extend_members : string list option;
  body : sabsFunDef list;
}

type sprogramComp =
  | SStmt of sstmt
  | SFun of sfundef
  | SClass of sclassdef
  | SInterface of interfaceDef

type sprogram = sprogramComp list

(* Unparser function *)
let rec string_of_sexpr (t, e) =
  "(" ^ string_of_type t ^ " : "
  ^ (match e with
    | SLiteral l -> string_of_int l
    | SDliteral l -> l
    | SBoolLit true -> "true"
    | SBoolLit false -> "false"
    | SStringLiteral l -> l
    | SId l -> l
    | SBinop (e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr
          e2
    | SUnop (o, e) -> string_of_uop o ^ string_of_sexpr e
    | SAccess (e1, e2) -> string_of_sexpr e1 ^ "." ^ string_of_sexpr e2
    | SObjMethod (name, meth, el) ->
      string_of_type name ^ "." ^ meth ^ "("
      ^ String.concat ", " (List.map string_of_sexpr el)
      ^ ")"
    | SAsn (e1, e2) -> string_of_sexpr e1 ^ " = " ^ string_of_sexpr e2
    | SCall (n, el) ->
      n ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
```

```
    | SListExpr el -> (
        match el with
        | Some lis ->
            "[" ^ String.concat ", " (List.map string_of_sexpr lis) ^ "]"
        | None -> "[]")
    | SIndexing (id, idx, exp) -> (
        id ^ string_of_sexpr idx
        ^ match exp with Some e -> " = " ^ string_of_sexpr e | None ->
           "")
    | SParenExp e -> "(" ^ string_of_sexpr e ^ ")"
    | SNewExpr e -> "new " ^ string_of_sexpr e
    | SDefAsn (ty, id, e) -> string_of_type ty ^ " " ^ id ^
       string_of_sexpr e
    | SThis -> "this"
    | SNull -> "null"
    | SSuper -> "super"
    | SNewArray (t, e) ->
        "new " ^ string_of_type t ^ " [" ^ string_of_sexpr e ^ "]"
    | SNoexpr -> "")
  ^ ")"

let rec string_of_sstmt = function
  | SBlock stmts ->
      "{\n    "
      ^ String.concat "\n    " (List.map string_of_sstmt stmts)
      ^ "\n}"
  | SExpr expr -> string_of_sexpr expr ^ ";"
  | SReturn expr -> "return " ^ string_of_sexpr expr ^ ";"
  | SIf (e, s, SBlock []) ->
      "if (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
  | SIf (e, s1, s2) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s1 ^ " else "
      ^ string_of_sstmt s2 ^ "\n    "
  | SFor (e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; "
      ^ string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
  | SWhile (e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^
     string_of_sstmt s
  | SControlFlow Break -> "break;"
  | SControlFlow Continue -> "continue;"
  | SNoStmt -> ""

let string_of_sfundef (fd : sfundef) =
  string_of_type fd.ty ^ " " ^ fd.id ^ "("
  ^ String.concat ", "
      (List.map (function t, s -> string_of_type t ^ " " ^ s) fd.args)
  ^ ") {\n    "
  ^ String.concat "\n    " (List.map string_of_sstmt fd.body)
  ^ "\n}\n"
```

```
let string_of_sclassStmt = function
  | SConstructorDef sfundef ->
      "constructor " ^ string_of_type sfundef.ty ^ "("
      ^ String.concat ", "
          (List.map
             (function t, s -> string_of_type t ^ " " ^ s)
             sfundef.args)
      ^ ") {\n    "
      ^ String.concat "\n" (List.map string_of_sstmt sfundef.body)
      ^ ";\n}"
  | SFieldDef (ac, fm, t, s, e) ->
      string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^ string_of_type t ^
         " " ^ s
      ^ " = " ^ string_of_sexpr e ^ ";"
      (* (match e with
         | Some exp -> string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^
           string_of_type t ^ " " ^ s ^ " = " ^ string_of_sexpr exp ^
           ";"
         | None -> string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^
           string_of_type t ^ " " ^ s ^ ";") *)
  | SMethodDef (ac, fm, fd) ->
      string_of_ac ac ^ " " ^ string_of_fm fm ^ " " ^ string_of_sfundef
         fd

let string_of_sclassdef (cd : sclassdef) =
  "class " ^ cd.id ^ " " ^ string_of_father cd.father ^ " "
  ^ string_of_class_interface cd.interface
  ^ " {\n    "
  ^ String.concat "\n    " (List.map string_of_sclassStmt cd.body)
  ^ "\n}\n"

let string_of_sabsfundef (sabsfundef : sabsFunDef) =
  string_of_fm sabsfundef.fieldM
  ^ " "
  ^ string_of_type sabsfundef.ty
  ^ " " ^ sabsfundef.id ^ "("
  ^ String.concat ";\n"
      (List.map (function t, s -> string_of_type t ^ " " ^ s) sabsfundef.
         args)
  ^ ");"

let string_of_sinterfacedef sinterfacedef =
  "interface " ^ sinterfacedef.id ^ " "
  ^ string_of_abs_interface sinterfacedef.extend_members
  ^ " {\n    "
  ^ String.concat "\n   " (List.map string_of_sabsfundef sinterfacedef.
     body)
  ^ "\n}"

let string_of_sprogramcomp = function
```

```
  | SStmt s -> string_of_sstmt s
  | SFun f -> string_of_sfundef f
  | SClass c -> string_of_sclassdef c
  | SInterface i -> string_of_interfacedef i

let string_of_sprogram program =
  String.concat "\n" (List.map string_of_sprogramcomp program)
```

### A.1.7 codegen.ml

```
(* Description:  Code generation of MicroJ
   Class: COMP107
   Author: Zichen Yang, Chenxuan Liu, Wei Shen, Weishi Ding
   Date: 5-4-2023 *)

(* open Llvm *)
open Sast
open Semant
module L = Llvm
module A = Ast
module StringMap = Map.Make (String)
module FunSigMap = Map.Make (FunSig)

let translate ((program : sprogramComp list), classMap_from_sement) =
  (* print_string "In codegen" *)
  let context = L.global_context () in
  let funMap = ref FunSigMap.empty
  and globalvarMap = ref StringMap.empty
  and classMap = ref StringMap.empty
  and classTypeMap = ref StringMap.empty  (*store the strcut_type of each
      class; nonstatic only*)
  and classTypeListMap = ref StringMap.empty in(*store the type_list of
     each class; nonstatic only*)

  (* Add types to the context *)
  let string_t = L.i8_type context (*string_ptr*)
  and bool_t = L.i1_type context (*bool*)
  and i32_t = L.i32_type context
  and int_t = L.i64_type context (*int*)
  and void_t = L.void_type context
  and double_t = L.double_type context
  and the_module = L.create_module context "MicroJ" in
  (*initialize value according to the given type*)
  let init_val ty =
      match ty with
        A.Int -> L.const_int int_t 0
      | A.Bool -> L.const_int int_t 0
      | A.Double -> L.const_float double_t 0.0
```

```
    | A.String -> L.const_stringz context "" (**** could be wrong here
       ***)
    | A.BoolList -> L.const_null (L.pointer_type bool_t)
    | A.IntList -> L.const_null (L.pointer_type int_t)
    | A.DoubleList -> Llvm.const_null (L.pointer_type double_t)
    | A.StringList -> L.const_null (L.pointer_type((L.pointer_type
       string_t)))
    | Object s -> let struct_type = try StringMap.find s !classTypeMap
                                with Not_found -> raise (Exception ("
                                   Type " ^ s ^ " is undefined"))
               in
                 L.const_null (L.pointer_type struct_type)
    | ObjectList s -> let struct_type = try StringMap.find s !
       classTypeMap
                                  with Not_found -> raise (Exception
                                     ("Type " ^ s ^ " is undefined"))
                 in
                   L.const_null (L.pointer_type (L.pointer_type
                      struct_type))
      | _ -> raise (Exception "invalid type")
  in
(* Covert MicroJ types to LLVM types *)
let ltype_of_typ = function
  | A.String -> L.pointer_type string_t
  | A.Int -> int_t
  | A.Void -> void_t
  | A.Bool -> bool_t
  | A.Double -> double_t
  | IntList -> L.pointer_type int_t
  | DoubleList -> L.pointer_type double_t
  | BoolList -> L.pointer_type bool_t
  | StringList -> L.pointer_type (L.pointer_type string_t)
  | Object s -> let struct_type = try StringMap.find s !classTypeMap
                              with Not_found -> raise Not_found
               in
                 L.pointer_type struct_type
  | ObjectList s -> let struct_type = try StringMap.find s !
     classTypeMap
                               with Not_found -> raise Not_found
               in
                 L.pointer_type (L.pointer_type struct_type)

in
 (*** Add all class type into the classTypeMap ****)
let rec find_all_fields cname =
  let fieldM',_,_, _, _, (cdf : A.classdef) = StringMap.find cname
     classMap_from_sement in
  let type_list =
    List.rev
      (StringMap.fold
```

```
          (fun _key (_, _, ty) acc -> ty :: acc)
          fieldM' [])
   in
   match cdf.father with
       None -> type_list
     | Some cname' -> type_list @ find_all_fields cname'
 in
 let rec add_class_struct_type className (fieldM,_,_,_,_,(cdf:A.classdef
    ))=
   let type_list' =
     List.rev
       (StringMap.fold
          (fun _key (_, _, ty) acc -> ty :: acc )
          fieldM [])
   in
   let type_list = type_list' @ (match cdf.father with None -> [] | Some
       cname -> find_all_fields cname) in
   let struct_type = L.struct_type context
           (Array.of_list (List.map
               (fun ty -> try ltype_of_typ ty with
                   Not_found -> match ty with
                     Object s | ObjectList s -> if (StringMap.mem s
                        classMap_from_sement) then
                       let _ = add_class_struct_type s (StringMap.find s
                           classMap_from_sement) in
                        ltype_of_typ ty
                       else raise (Exception ("Type " ^ A.string_of_type
                           ty ^ " is undefined"))
                     | _ -> raise (Exception ("Type " ^ A.string_of_type
                        ty ^ " is undefined"))) type_list)) in
   let _ = classTypeMap := StringMap.add className struct_type !
      classTypeMap in
     classTypeListMap := StringMap.add className type_list !
        classTypeListMap
 in
 let _ = StringMap.iter add_class_struct_type classMap_from_sement in
 let printf_t : L.lltype =
   L.var_arg_function_type i32_t [| L.pointer_type string_t |]
 in
 let printf_func : L.llvalue =
   L.declare_function "printf" printf_t the_module
 in
 (**************** Build global variables ***************)
 let global_vars vardef =
   match vardef with
   | SStmt (SExpr (_, SDefAsn (_, name, (t, value)))) ->
       let init =
         match t with
         | A.Double -> (
             match value with
```

```
              | SNull -> L.const_float (ltype_of_typ t) 0.0
              | SDliteral s ->
                  L.const_float (ltype_of_typ t) (float_of_string s)
              | _ -> raise (Failure "Expected Dliteral or SNull"))
          | A.Int -> (
              match value with
              | SNull -> L.const_int (ltype_of_typ t) 0
              | SLiteral i -> L.const_int (ltype_of_typ t) i
              | _ -> raise (Failure "Expected Sliteral or SNull"))
          | A.Bool -> (
              match value with
              | SNull -> L.const_int (ltype_of_typ t) 0
              | SBoolLit b -> L.const_int bool_t (if b then 1 else 0)
              | _ -> raise (Failure "Expected Sliteral or SNull"))
          | A.String -> (
              match value with
              (*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!constant string is not
                 generated properly!!!!!!!!!!!!!!!!!!!!!!!!!!*)
              | SNull -> L.const_null (L.pointer_type string_t)
              | SStringLiteral s -> L.const_stringz context s
              | _ -> raise (Failure "Expected SStringliteral or SNull"))
          | _ -> raise (Failure "Not implemented yet")
        in

        (globalvarMap : L.llvalue StringMap.t ref)
        := StringMap.add name
             (L.define_global name init the_module)
             !globalvarMap
    | _ -> raise (Failure "Only global variable definitions are allowed
        outside main")
  in
  let function_decl sx =
    match sx with
    | SFun sfundef ->
        let name = sfundef.id
        and args_types = List.map (fun (t, _) -> t) sfundef.args
        and args_types' =
          Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) sfundef.
             args)
        in
        let key : FunSig.t = (name, args_types) in
        let ftype = L.function_type (ltype_of_typ sfundef.ty) args_types'
            in
        funMap :=
          FunSigMap.add key
            (L.define_function name ftype the_module, sfundef)
            !funMap
    | _ -> raise (Exception "Expected SFun")
  in
```

```
(* find_method function will recursivly find the method. It will lookup
    in its father until raise Not found *)
  (* Both static and nonstatic method are handled here *)
  let rec find_method key cname (static : bool) =
    if static = false then
      let _,_, _, methodM,_, (cdf : sclassdef) = StringMap.find cname !
         classMap in
      try FunSigMap.find key methodM
      with Not_found -> (
        match cdf.father with
        | Some fname -> find_method key fname static
        | None -> raise (Failure ("Function " ^ fst key ^ " is
            undefined")))
    else
      let _,_, _,_, smethodM, (cdf : sclassdef) = StringMap.find cname
         !classMap in
      try FunSigMap.find key smethodM
      with Not_found -> (
        match cdf.father with
        | Some fname -> find_method key fname static
        | None -> raise (Failure ("Function " ^ fst key ^ " is
            undefined")))
  in

(*********find_field function will recursivly find the method. It will
    lookup in its father until raise Not found***********)
let rec find_field (key : string) cname (index, answer) =
  let fieldM', _, _,_,_, (cdf : sclassdef) = StringMap.find cname !
     classMap in
  let (new_index,new_answer) = (StringMap.fold
              (fun fieldname (_,fflag,_,_) (index', answer') ->
                if fieldname = key && fflag = "nonstatic" then (index'
                   + 1, index')
                else (index' + 1, answer'))
              fieldM' (index, answer))
  in
    if new_answer = -1 then
      match cdf.father with
      | Some fname -> find_field key fname (new_index,new_answer)
      | None -> answer
    else
      new_answer

in
let rec find_static_field (key : string) cname  =
  let _, sfield, _,_,_, (cdf : sclassdef) = StringMap.find cname !
     classMap in
      let value = try StringMap.find key sfield with
          Not_found -> (
              match cdf.father with
```

```
                None -> raise Not_found
              | Some fname -> find_static_field key fname
          ) in
        value
in

let build_function_body (the_function, (sfundef : sfundef)) =
  let builder = L.builder_at_end context (L.entry_block the_function)
     in
  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and double_format_str = L.build_global_stringptr "%f\n" "fmt" builder
  and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
      in
  let local_vars =
    let add_formal m (t, n) p =
      let () = L.set_value_name n p in
      let local = L.build_alloca (ltype_of_typ t) n builder in
      let _ = L.build_store p local builder in
      StringMap.add n local m
    in
    ref
      (List.fold_left2 add_formal StringMap.empty sfundef.args
         (Array.to_list (L.params the_function)))
  in
  let lookup n =
    try StringMap.find n !local_vars
    with Not_found -> (
      try StringMap.find n !globalvarMap
      with Not_found -> raise (Failure ("undeclared identifier " ^ n)))
  in
  let rec expr builder ((_, e) : sexpr) =
    (***** helper function for SAccess *****)
    (******** field_or_method returns a pointers to the field of a
       struct *******)
    let get_index se builder =
      let llv = expr builder se in llv
    in
    let field_or_method e funM fieldM struct_ptr' (scdf : sclassdef) (
       static : bool)=
      match e with
      | _, SCall (fname, args) ->
          let fdef, (fdecl : sfundef) =
            (*need to add the self type into the key if it's a
               nonstatic method*)
            let key = match static with true -> get_formals_type args
                                        | false -> (A.Object scdf.id) ::
                                           (get_formals_type args)
            in
            try FunSigMap.find (fname, key) funM
            with Not_found ->
```

```
            match scdf.father with
              None -> raise (Exception ("Function " ^ fname ^ "
                 undefined"))
            | Some cname -> let key' = if static = true then (
                 get_formals_type args)
                                         else (A.Object cname) :: (
                                            get_formals_type args) in
                                             find_method (fname, key')
                                                cname static
        in
        let llargs =
            (* give the function call self argument if its nonstatic
               *)
            match static with false -> struct_ptr' :: (List.rev (List
               .map (expr builder) (List.rev args)))
                            | true -> (List.rev (List.map (expr
                               builder) (List.rev args)))  in
        let result =
          match fdecl.ty with A.Void -> "" | _ -> fname ^ "_result"
        in
          L.build_call fdef (Array.of_list llargs) result builder
    | _, SId name ->
        let (idx, ans) =
            (StringMap.fold
               (fun key _value (index, answer) ->
                 if key = name then (index + 1, index)
                 else (index + 1, answer))
               fieldM (0, -1))
        in
        let k = match ans
          with -1 ->
              (match scdf.father with
                None -> raise (Exception ("Field " ^ name ^ "
                   undefined"))
              | Some cname -> let k' = find_field name cname (idx,
                 ans) in
                                if k' = -1
                                  then raise (Failure ("Field " ^
                                     name ^ " is undefined called"))
                                else k')
          | _ -> ans
        in
        let field_ptr = L.build_struct_gep struct_ptr' k "field_ptr"
           builder in
          field_ptr
    | _, SIndexing(name, e', e_op) ->
        let (idx, ans) =
            (StringMap.fold
               (fun key _value (index, answer) ->
                 if key = name then (index + 1, index)
```

```
                    else (index + 1, answer))
                  fieldM (0, -1))
           in
           let k = match ans
             with -1 ->
                 (match scdf.father with
                   None -> raise (Exception ("Field " ^ name ^ "
                      undefined"))
                 | Some cname -> let k' = find_field name cname (idx,
                      ans) in
                                    if k' = -1
                                      then raise (Failure ("Field " ^
                                         name ^ " is undefined"))
                                    else k')
             | _ -> ans
           in
           let field_ptr = (L.build_struct_gep struct_ptr' k "field_ptr"
               builder )in
           let index = get_index e' builder in
           (* let llvalueIndex = [| L.const_int int_t (Int64.to_int
              index) |] in *)
           let element =
             L.build_gep field_ptr [|index|] "array" builder
           in
           (match e_op with
           | Some sexp ->
               let e' = expr builder sexp in
               L.build_store e' element builder
           | None -> element)
       | _ -> raise (Exception "Cannot access things other than field or
           method")
     in
     match e with
     | SLiteral i -> L.const_int int_t i
     | SStringLiteral s ->
         let str_ptr = L.build_global_stringptr s "str" builder in
         let str_ptr_cast =
           L.build_bitcast str_ptr (L.pointer_type string_t) "strcast"
              builder
         in
         str_ptr_cast
     | SBoolLit b -> L.const_int bool_t (if b then 1 else 0)
     | SDliteral l -> L.const_float_of_string double_t l
     | SNoexpr -> L.const_int i32_t 0
     | SId s -> L.build_load (lookup s) s builder
     | SDefAsn (t, name, sexpr) ->
         let t', e = sexpr in
         let local = L.build_alloca (ltype_of_typ t) name builder in
         (* let () = prerr_endline(L.string_of_llvalue local)  in *)
         let _ = local_vars := StringMap.add name local !local_vars in
```

```
        let value =
          match t' with
          | Int | Bool -> (
              match e with
              | SNull -> L.const_int (ltype_of_typ t) 0
              | sexpr -> expr builder (t, sexpr))
          | String -> (
              match e with
              | SNull -> L.const_null (L.pointer_type string_t)
              | sexpr -> expr builder (t, sexpr))
          | Double -> (
              match e with
              | SNull -> L.const_float (ltype_of_typ t) 0.0
              | sexpr -> expr builder (t, sexpr))
          | IntList -> (
              match e with
              | SNull -> L.const_null (L.pointer_type int_t)
              | sexpr -> expr builder (t, sexpr))
          | DoubleList -> (
              match e with
              | SNull -> L.const_null (L.pointer_type double_t)
              | sexpr -> expr builder (t, sexpr))
          | BoolList -> (
              match e with
              | SNull -> L.const_null (L.pointer_type bool_t)
              | sexpr -> expr builder (t, sexpr))
          | StringList -> (
              match e with
              | SNull -> L.const_null (L.pointer_type string_t)
              | sexpr -> expr builder (t, sexpr))
          | Object _ as ob -> (
              match e with
              | SNull -> L.const_null (L.pointer_type (ltype_of_typ ob)
                 )
              | sexpr -> expr builder (t, sexpr))
          | ObjectList _ as oblist -> (
              match e with
              | SNull -> L.const_null (L.pointer_type ((L.pointer_type
                  (ltype_of_typ oblist))))
              | sexpr -> expr builder (t, sexpr))
          | _ -> raise (Exception "SDefAsn not implemented")
        in
        L.build_store value local builder
    | SCall ("charAt", [ (_, name); e ]) -> (
        let index = get_index e builder in
        (* let llvalueIndex = [| L.const_int int_t (Int64.to_int index)
            |] in *)
        match name with
        | SId name ->
            let get_array = L.build_load (lookup name) name builder in
```

```
          let get_element =
            L.build_gep get_array [|index|] "array" builder
          in
          let asic = L.build_load get_element name builder in
          (* let asic_64 = L.build_zext asic int_t "sb" builder in *)
          let () = raise (Failure (string_of_bool (L.is_constant asic
             ))) in
          let ocaml_int64 =
            match L.int64_of_const asic with
            | None -> raise (Failure "index must be integer")
            | Some v -> v
          in
          let ocaml_char = Char.chr (Int64.to_int ocaml_int64) in
          let ocaml_string = String.make 1 ocaml_char in
          let sl = SStringLiteral ocaml_string in
          expr builder (A.String, sl)
      | _ -> raise (Exception "Expected a ID of string"))
  | SCall ("print", [ (t, e') ]) -> (
      match t with
      | A.Int | A.Bool | A.IntList ->
          L.build_call printf_func
            [| int_format_str; expr builder (t, e') |]
            "printf" builder
      | A.String ->
          L.build_call printf_func
            [| string_format_str; expr builder (t, e') |]
            "printf" builder
      | A.Double | A.DoubleList ->
          L.build_call printf_func
            [| double_format_str; expr builder (t, e') |]
            "printf" builder
      | A.Object "Animal" ->
          L.build_call printf_func
            [| int_format_str; expr builder (t, e') |]
            "printf" builder
      | _ -> raise (Exception "Unmatched print function"))
  | SCall (f, args) ->
      let get_formals_type args =
        List.rev
          (List.fold_left (fun accu (typ, _id) -> typ :: accu) []
             args)
      in
      let fdef, fdecl =
        try FunSigMap.find (f, get_formals_type args) !funMap
        with Not_found ->
          raise (Exception ("Function " ^ f ^ " undefined"))
      in
      let llargs = List.rev (List.map (expr builder) (List.rev args))
          in
      let result =
```

```
          match fdecl.ty with A.Void -> "" | _ -> f ^ "_result"
        in
          L.build_call fdef (Array.of_list llargs) result builder
    | SNewExpr (_ty, e) -> (
        match e with
        | SCall (fname, args) ->
            let _,_, constM, _, _,_=
              try StringMap.find fname !classMap
              with Not_found ->
                raise (Exception ("Class " ^ fname ^ " undefined"))
            in
            let fdef, (fdecl : sfundef) =
              try FunSigMap.find (fname, get_formals_type args) constM
              with Not_found ->
                raise (Exception ("Constructor " ^ fname ^ " undefined
                   "))
            in
            let llargs = List.rev (List.map (expr builder) (List.rev
               args)) in
            let result =
              match fdecl.ty with A.Void -> "" | _ -> fname ^ "_result"
            in
              L.build_call fdef (Array.of_list llargs) result builder
        | _ -> raise (Exception "Expect a constructor after 'new'
           keyword\n"))
    | SAccess (e1, e2) -> (
        match e1 with
        | ty, SId name ->
            if StringMap.mem name !classMap then
              (*if it's a class name rather than a var name, then it
                 can access static field or method*)
              let _, sfieldM,_,_,smethodM, scdf = StringMap.find name !
                 classMap in
                (match e2 with
                  (_ ,SCall _) -> field_or_method e2 smethodM sfieldM (
                     L.const_null int_t) scdf true
                | (_, SId field_name) ->
                  let field_ptr = try StringMap.find field_name sfieldM
                      with Not_found ->
                        (match scdf.father with
                          None -> raise (Exception ("Static Field " ^
                             name ^ " undefined"))
                        | Some fname -> try find_static_field
                            field_name fname
                                    with Not_found -> raise (
                                       Exception ("Static field " ^
                                        field_name ^ " is undefined
                                       ")))
                  in
                  L.build_load field_ptr "static_field" builder
```

```
          | (_, SIndexing(field_name, e', e_op)) ->
            let field_ptr = try StringMap.find field_name sfieldM
                with Not_found ->
                  (match scdf.father with
                    None -> raise (Exception ("Static Field " ^
                       name ^ " undefined"))
                  | Some fname -> try find_static_field
                     field_name fname
                                  with Not_found -> raise (
                                     Exception ("Static field " ^
                                      field_name ^ " is undefined
                                     ")))
            in
            let index = get_index e' builder in
            (* let llvalueIndex = [| L.const_int int_t (Int64.
               to_int index) |] in *)
            let element =
              L.build_gep field_ptr [|index|] "array" builder
            in
            (match e_op with
            | Some sexp ->
                let e' = expr builder sexp in
                L.build_store e' element builder
            | None -> L.build_load element field_name builder)
          | _ -> raise (Exception "Cannot access things other
             than field or method"))
        (* The last argument is useless here, only to make the
           compiler happy*)
      else
        let struct_ptr = L.build_load (lookup name) name builder
           in
        let (fieldM : 'a StringMap.t), _,_, methodM,_, scdf=
          try StringMap.find (A.string_of_type ty) !classMap
          with Not_found ->
            raise
              (Exception
                 ("Class " ^ A.string_of_type ty ^ " is undefined
                    "))
        in
         (match e2 with (_ ,SCall _) -> field_or_method e2
            methodM fieldM struct_ptr scdf false
          | (_, SId _) -> L.build_load (field_or_method e2
             methodM fieldM struct_ptr scdf false) "" builder
          | (_, SIndexing(name,_,None)) -> L.build_load (
             field_or_method e2 methodM fieldM struct_ptr scdf
             false) name builder
          | (_, SIndexing(_,_,Some _)) -> field_or_method e2
             methodM fieldM struct_ptr scdf false
          | _ -> raise (Exception "Cannot access things other
             than field or method"))
```

```
      | ty, SIndexing (_, _, sexpr_o) ->
          (match sexpr_o with Some _ -> raise (Exception ("Invalid
             DefAsn"))
            | None ->
              let struct_ptr = expr builder e1 in
              let (fieldM : 'a StringMap.t), _,_, methodM,_, scdf=
                try StringMap.find (A.string_of_type ty) !classMap
                with Not_found ->
                  raise
                    (Exception
                      ("Class " ^ A.string_of_type ty ^ " is
                         undefined"))
                in
              (match e2 with (_ ,SCall _) -> field_or_method e2
                 methodM fieldM struct_ptr scdf false
                | (_, SId _) -> L.build_load (field_or_method e2
                   methodM fieldM struct_ptr scdf false) "" builder
                | (_, SIndexing(name,_,None)) -> L.build_load (
                   field_or_method e2 methodM fieldM struct_ptr scdf
                   false) name builder
                | (_, SIndexing(_,_,Some _)) -> field_or_method e2
                   methodM fieldM struct_ptr scdf false
                | _ -> raise (Exception "Cannot access things other
                   than field or method")))
      | _ -> raise (Failure "Expect an object type on left-hand side
          of Access"))
  | SAsn (e1, e2) ->
    (match e1 with (_, SId s) ->
      let e2' = expr builder e2 in
      let _ = L.build_store e2' (lookup s) builder in
        e2'
    | (ty, SAccess((cty, SId name), ((_, SId field_name) as e'')))->
        (***Now only allows to modify a field. Modify method is not
       allowed ***)
      if (StringMap.mem name !classMap) = false then
        let struct_ptr = L.build_load (lookup name) name builder in
        let fieldM,_, _, methodM,_, scdf =
          try StringMap.find (A.string_of_type cty) !classMap
          with Not_found ->
            raise
              (Exception
                ("Class " ^ A.string_of_type ty ^ " is undefined"))
        in
        let target_ptr = field_or_method e'' methodM fieldM
           struct_ptr scdf false in
        let e2' = expr builder e2 in
        let _ = L.build_store e2' target_ptr builder in
          e2'
      else
        let _, sfieldM,_,_,_, scdf = StringMap.find name !classMap in
```

```
            let value = try StringMap.find field_name sfieldM
                with Not_found ->
                  (match scdf.father with
                    None -> raise (Exception ("Static Field " ^ name
                       ^ " undefined"))
                  | Some fname -> find_static_field field_name fname)
            in
            let e2' = expr builder e2 in
            let _ = L.build_store e2' value builder in
              e2'
    | (ty, SAccess((cty, SIndexing _) as e3, ((_, SId _) as e''))) ->
        let struct_ptr = expr builder e3 in
        let fieldM,_, _, methodM,_, scdf =
        try StringMap.find (A.string_of_type cty) !classMap
            with Not_found ->
              raise (Exception ("Class " ^ A.string_of_type ty ^ " is
                  undefined"))
        in
        let target_ptr = field_or_method e'' methodM fieldM
           struct_ptr scdf false in
        let e2' = expr builder e2 in
        let _ = L.build_store e2' target_ptr builder in
          e2'
    | (ty, SAccess((cty, SIndexing _) as e3, ((_cty', SIndexing _) as
        e4))) ->
      let struct_ptr = expr builder e3 in
      let (fieldM : 'a StringMap.t), _,_, methodM,_, scdf=
        try StringMap.find (A.string_of_type cty) !classMap
        with Not_found ->
          raise
            (Exception
              ("Class " ^ A.string_of_type ty ^ " is undefined"))
        in
        let target_ptr = field_or_method e4 methodM fieldM struct_ptr
            scdf false in
        let e2' = expr builder e2 in
        let _ = L.build_store e2' target_ptr builder in
          e2'
    | _ -> raise (Exception "Bad left-hand side of assign"))
  | SBinop (e1, op, e2) ->
      let t, _ = e1 and e1' = expr builder e1 and e2' = expr builder
         e2 in
      (match t with A.Double ->
        (match op with
        | A.Add -> L.build_fadd
        | A.Sub -> L.build_fsub
        | A.Mult -> L.build_fmul
        | A.Div -> L.build_fdiv
        | A.Equal -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq -> L.build_fcmp L.Fcmp.One
```

```
          | A.Less -> L.build_fcmp L.Fcmp.Olt
          | A.Leq -> L.build_fcmp L.Fcmp.Ole
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt
          | A.Geq -> L.build_fcmp L.Fcmp.Oge
          | A.And | A.Or ->
              raise
                (Failure
                   "internal error: semant should have rejected and/or
                      on \
                    float"))
            e1' e2' "tmp" builder
        | A.Int ->
          (match op with
          | A.Add -> L.build_add
          | A.Sub -> L.build_sub
          | A.Mult -> L.build_mul
          | A.Div -> L.build_sdiv
          | A.And -> L.build_and
          | A.Or -> L.build_or
          | A.Equal -> L.build_icmp L.Icmp.Eq
          | A.Neq -> L.build_icmp L.Icmp.Ne
          | A.Less -> L.build_icmp L.Icmp.Slt
          | A.Leq -> L.build_icmp L.Icmp.Sle
          | A.Greater -> L.build_icmp L.Icmp.Sgt
          | A.Geq -> L.build_icmp L.Icmp.Sge)
            e1' e2' "tmp" builder
        | A.Bool ->
          (match op with
          | A.And -> L.build_and
          | A.Or -> L.build_or
          | A.Equal -> L.build_icmp L.Icmp.Eq
          | A.Neq -> L.build_icmp L.Icmp.Ne
          | _ -> raise (Exception "invalid binop on bool type"))
            e1' e2' "tmp" builder
        |  _ -> (match op with
          | A.Equal -> L.build_icmp L.Icmp.Eq
          | A.Neq -> L.build_icmp L.Icmp.Ne
          | _ -> raise (Exception "Cannot apply binod other then '=='
             and '!=' on Objects or Arrays"))
            e1' e2' "tmp" builder)
    | SUnop (op, e) ->
        let t, _ = e in
        let e' = expr builder e in
        (match op with
        | A.Neg when t = A.Double -> L.build_fneg
        | A.Neg -> L.build_neg
        | A.Not -> L.build_not)
          e' "tmp" builder
    | SNewArray (ty, sexpr) -> (
        match ty with
```

```
        | Int ->
            let index = get_index sexpr builder in
            (* let llvalueIndex = L.const_int int_t (Int64.to_int index
               ) in *)
            L.build_array_malloc int_t index "array" builder
        | Double ->
            let index = get_index sexpr builder in
            (* let llvalueIndex = L.const_int int_t (Int64.to_int index
               ) in *)
            L.build_array_malloc double_t index "array" builder
        | String ->
            let index = get_index sexpr builder in
            let string_Ptr = L.pointer_type (string_t) in
            (* let llvalueIndex = L.const_int int_t (Int64.to_int index
               ) in *)
            L.build_array_malloc string_Ptr index "array" builder
        | Bool ->
            let index = get_index sexpr builder in
            (* let llvalueIndex = L.const_int int_t (Int64.to_int index
               ) in *)
            L.build_array_malloc bool_t index "array" builder
        | Object _ as ob ->
            let index = get_index sexpr builder in
            (* let llvalueIndex = L.const_int int_t (Int64.to_int index
               ) in *)
            L.build_array_malloc (ltype_of_typ ob) index "array"
               builder
        | _ -> raise (Exception "Wait for implement"))
    | SIndexing (id, sexpr, sexpr_o) -> (
        let index = get_index sexpr builder in
        (* let llvalueIndex = [| L.const_int int_t (Int64.to_int index)
            |] in *)
        let array = L.build_load (lookup id) id builder in
        let element =
          L.build_gep array [|index|] "array" builder
        in
        match sexpr_o with
        | Some sexp ->
            let e' = expr builder sexp in
            L.build_store e' element builder
        | None -> L.build_load element id builder)
    | _ -> raise (Exception "New Need Implementation")
  in
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
    | Some _ -> () (***if the current block already has a terminator
       ***)
    | None -> ignore (instr builder)
    (***if the current block doesn't have a terminator***)
  in
```

```
let rec stmt builder block = function
  | SBlock sl ->
      List.fold_left (fun accb st -> stmt accb block st) builder sl
  | SReturn e ->
      let _ =
        match sfundef.ty with
        (* Special "return nothing" instr *)
        | A.Void -> L.build_ret_void builder (* Build return
           statement *)
        | _ -> L.build_ret (expr builder e) builder
      in
      builder
  | SControlFlow Break -> (
      match block with
      | None -> raise (Exception "No destination for break")
      | Some bb_list ->
          let _ = L.build_br (fst bb_list) builder in
          builder)
  | SControlFlow Continue -> (
      match block with
      | None -> raise (Exception "No destination for continue")
      | Some bb_list ->
          let _ = L.build_br (snd bb_list) builder in
          builder)
  | SExpr e ->
      let _ = expr builder e in
      builder
  | SIf (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
      (* Add "merge" basic block to our function's list of blocks *)
      let merge_bb = L.append_block context "merge" the_function in
      (* Partial function used to generate branch to merge block *)
      let branch_instr = L.build_br merge_bb in

      (* Same for "then" basic block *)
      let then_bb = L.append_block context "then" the_function in
      (* Position builder in "then" block and build the statement *)
      let then_builder =
        stmt (L.builder_at_end context then_bb) block then_stmt
      in
      (* Add a branch to the "then" block (to the merge block)
         if a terminator doesn't already exist for the "then" block
            *)
      let () = add_terminal then_builder branch_instr in

      (* Identical to stuff we did for "then" *)
      let else_bb = L.append_block context "else" the_function in
      let else_builder =
        stmt (L.builder_at_end context else_bb) block else_stmt
      in
```

```
        let () = add_terminal else_builder branch_instr in

        (* Generate initial branch instruction perform the selection of
            "then"
           or "else". Note we're using the builder we had access to at
              the start
           of this alternative. *)
        let _ = L.build_cond_br bool_val then_bb else_bb builder in
        (* Move to the merge block for further instruction building *)
        L.builder_at_end context merge_bb
    | SWhile (predicate, body) ->
        (* First create basic block for condition instructions -- this
           will
           serve as destination in the case of a loop *)
        let pred_bb = L.append_block context "while" the_function in
        (* In current block, branch to predicate to execute the
           condition *)
        let _ = L.build_br pred_bb builder in

        (* Create the body's block, generate the code for it, and add a
            branch
           back to the predicate block (we always jump back at the end
              of a while
           loop's body, unless we returned or something) *)
        let body_bb = L.append_block context "while_body" the_function
           in
        let merge_bb = L.append_block context "merge" the_function in
        let while_builder =
          stmt
            (L.builder_at_end context body_bb)
            (Some (merge_bb, pred_bb))
            body
        in
        let () = add_terminal while_builder (L.build_br pred_bb) in

        (* Generate the predicate code in the predicate block *)
        let pred_builder = L.builder_at_end context pred_bb in
        let bool_val = expr pred_builder predicate in

        (* Hook everything up *)
        let _ = L.build_cond_br bool_val body_bb merge_bb pred_builder
           in
        L.builder_at_end context merge_bb
    | SFor (e1, e2, e3, body) ->
        (*semant can not regconized var in e1 e2 e3*)
        stmt builder block
          (SBlock [ SExpr e1; SWhile (e2, SBlock [ body; SExpr e3 ]) ])
    | _ -> raise (Exception "stmt function needs Implementation")
  in
  let builder = stmt builder None (SBlock sfundef.body) in
```

```
    add_terminal builder
      (match sfundef.ty with
      | A.Void -> L.build_ret_void
      | A.Double -> L.build_ret (L.const_float double_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in
  (* build a class definition*)
  let class_decl (c : sclassdef) =
    (* Step 1: scan through the sclassdef and creat three maps*)
    let scanning (fieldM, sfieldM, constM, methodM, smethodM) (s :
       sclassStmt) =
      match s with
      | SConstructorDef sfundef ->
          (* add to local methodMap*)
          let name = sfundef.id
          and args_types = List.map (fun (t, _) -> t) sfundef.args
          and args_types' =
            Array.of_list (List.map (fun (t, _) -> ltype_of_typ t)
               sfundef.args)
          in
          let key : FunSig.t = (name, args_types) in
          let ftype = L.function_type (ltype_of_typ sfundef.ty)
             args_types' in
          let constM =
            FunSigMap.add key
              (L.define_function name ftype the_module, sfundef)
              constM
          in
          (fieldM, sfieldM, constM, methodM, smethodM)
      | SFieldDef (accflag, fflag, ty, id, e) ->
          (* add a field to local fieldmap*)
          let accflag' = A.string_of_ac accflag in
          (match fflag with
            None ->
              let fieldM' = StringMap.add id (accflag', "nonstatic", ty,
                 e) fieldM in
                (fieldM', sfieldM, constM, methodM,smethodM)
            | Some _ ->
              let fieldM' = StringMap.add id (accflag', "static", ty, e)
                 fieldM in
              let sfieldM = StringMap.add id (L.define_global id (
                 init_val ty) the_module) sfieldM in
               (fieldM', sfieldM, constM, methodM,smethodM))


      | SMethodDef (_accflag, fflag, sfundef) ->
          (* let accflag' =  A.string_of_ac accflag in
             let fflag'   = match fflag with None -> "nonstatic" | Some _
                 -> "static" in *)
          let name = sfundef.id in
```

```
      let args_types =(List.map (fun (t, _) -> t) sfundef.args) in
      let args_types' =
        Array.of_list (List.map (fun (t, _) -> ltype_of_typ t)
           sfundef.args)
      in
      let key : FunSig.t = (name, args_types) in
      let ftype = L.function_type (ltype_of_typ sfundef.ty)
         args_types' in
        match fflag with
          None ->
            let methodM =
              FunSigMap.add key (L.define_function name ftype
                 the_module, sfundef) methodM
            in
            (fieldM, sfieldM, constM, methodM, smethodM)
        | Some _->
            let smethodM =
              FunSigMap.add key (L.define_function name ftype
                 the_module, sfundef) smethodM
            in
            (fieldM, sfieldM, constM, methodM, smethodM)

  in
  let (fieldM, sfieldM, constM, methodM, smethodM) =
    (* perform scanning here *)
    List.fold_left scanning
      (StringMap.empty, StringMap.empty,FunSigMap.empty, FunSigMap.
         empty, FunSigMap.empty)
      c.body
  in
  let _ = classMap := StringMap.add c.id (fieldM,sfieldM, constM,
     methodM, smethodM, c) !classMap in
  (*add the new class into classMap*)
  (* now use the field map to build the struct*)
  (*get the current struct type*)
  let type_list = StringMap.find c.id !classTypeListMap in
  let struct_type = StringMap.find c.id !classTypeMap in
  (*store the struct_ptr of current class, it will be used in build
     method*)
  let current_struct_ptr = L.define_global "my_struct_ptr" (L.
     const_null (L.pointer_type struct_type)) the_module in
  let build_constructor_body (the_function, (sfundef : sfundef)) =
    let builder = L.builder_at_end context (L.entry_block the_function)
       in
    (* let _ = classTypeMap := StringMap.add c.id struct_type !
       classTypeMap in *)
    let params = Array.to_list (L.params the_function) in
    let fieldName_param_pair = List.map2 (fun x y -> (snd x,y)) sfundef
       .args params in
    let struct_ptr = L.build_malloc struct_type "struct_ptr" builder in
```

```
      let _ = L.build_store struct_ptr current_struct_ptr builder in
      let initialize_field k ty =
        let field_ptr = L.build_struct_gep struct_ptr k "field_ptr"
           builder in
        let init = init_val ty in
        let _ = L.build_store init field_ptr builder in
          k + 1
      in
      let _ = List.fold_left initialize_field 0 type_list in (*
         initialize all field to *)
      let _ = List.iter (fun (fieldname,value) ->
         (*If constructor initialize all fields*)
            (* let k = snd (StringMap.fold (fun key _v (index, answer) ->
                          if key = fieldname then (index + 1, index)
                          else (index + 1, answer))
                        fieldM' (0, -1)) in  *)
           let k = (find_field fieldname c.id (0,-1)) in
               if k != -1 then
                 let field_ptr = L.build_struct_gep struct_ptr k "
                    field_ptr" builder in
                   ignore (L.build_store value field_ptr builder)

               else
                 let field_ptr = find_static_field fieldname c.id in
                   ignore (L.build_store value field_ptr builder))
                      fieldName_param_pair
      in
        L.build_ret struct_ptr builder (* return the pointer to the
           struct*)
    in
      FunSigMap.iter (fun _key value -> ignore (build_constructor_body
         value)) constM (* build all constructors here*)


  in
  (*body of classdecl*)
  let generate_all = function
    | SStmt _e as s -> global_vars s
    | SFun _sfundef as s -> function_decl s
    | SClass sclassdef -> class_decl sclassdef
    | _ -> ()
  in
  let _ = List.iter generate_all program in
  let _ = FunSigMap.iter (fun _key value -> build_function_body value) !
     funMap in
  let build_method_body (the_function, (sfundef : sfundef)) =
    build_function_body (the_function, sfundef)  (* build the method body
       *)
   (* let _ = raise (Exception( "Lengrth of globalvarMap: " ^
      string_of_int (StringMap.cardinal !globalvarMap)) in *)
```

```
in
let _ = StringMap.iter (fun _key (_,_,_,methodM, smethodM, _) ->
  let _ = FunSigMap.iter (fun _key value -> build_method_body value)
     methodM in(*iteratively build methods*)
    FunSigMap.iter (fun _key value -> build_method_body value) smethodM
       ) !classMap
in
the_module
```

## A.2 Killer Apps

### A.2.1 Inheritance showcase

```
/*Test class inheritance*/
class Father {
    public static int fPubStat;
    private static int fPriStat;

    public int fPub;
    private int fPri;

    constructor Father(int fPub, int fPri, int fPubStat, int fPriStat){

    }

    constructor Father(){

    }

    public static void fPublicStaticPrint(){
        print("public static");
    }

    private static void fPrivatetStaticPrint(){
        print("private static");
    }

    public void fPublicPrint(Father self){
        print("public");
    }

    private void fPrivatePrint(Father self){
        print("private");
    }
}

class Son extends Father{
    constructor Son(){

    }

    public void sTestPublic(Son self){
        Father.fPublicStaticPrint();
        self.fPublicPrint();
        print(self.fPub);

        Father f := new Father();
        f.fPublicPrint();
        print(f.fPub);
```

```
    }

    public void sTestPrivate(Son self){
        Father.fPrivatetStaticPrint();
        self.fPrivatePrint();
        print(self.fPri);

        Father f : = new Father();
        f.fPrivatePrint();
        print(f.fPri);
    }
}

int main(){
    Son s : = new Son();
    s.sTestPublic();
    s.sTestPrivate();
}
```

### A.2.2 Interface showcase

```
/*Test interface implementation and class extension*/

interface Behaviour extends bark{
}

interface bark{
    void bark();
}

class Animal {
    int leg;

    constructor Animal(){

    }
}

class Dog extends Animal implements Behaviour{
    constructor Dog(int leg){

    }

    void bark(Dog self){
        print("wang wang");
    }

}
```

```
int main(){
    Dog d : = new Dog(4);

    print(d.leg);
    d.bark();
}
```

### A.2.3 Polymorphism showcase

```
/*Test polymorphism*/
class Animal {
    constructor Animal(){

    }

    void printName(Animal self){

    }
}

class Cat extends Animal{
    constructor Cat(){

    }

    void printName(Cat self){
        print("Cat");
    }
}

class Bird extends Animal{
    constructor Bird(){

    }

    void printName(Bird self){
        print("Bird");
    }
}

class Dog extends Animal{
    constructor Dog(){

    }

    void printName(Dog self){
        print("Dog");
    }
}
```

```
int main() {
    Animal c : = new Cat();
    Animal b : = new Bird();
    Animal d : = new Dog();

    c.printName();
    b.printName();
    d.printName();
}
```