```
        if ("script" in langName)
            if ("Java" in langName)
                "Dynamic"
            else
                "Probably Dynamic"
        else
            "Unknown"
}

findLangType("Java")
findLangType("Javascript")
findLangType("Typescript")
```

Output (Java):

```
public class Main {
    public static void main(String[] args) {
        System.out.println(findLangType("Java"));
        System.out.println(findLangType("Javascript"));
        System.out.println(findLangType("Typescript"));
    }

    public static String findLangType(String langName) {
        if (langName.equals("Java")) {
            return "Static";
        }
        else {
            if (langName.contains("script")) {
                if (langName.contains("Java")) {
                    return "Dynamic";
                }
                else {
                    return "Probably Dynamic";
                }
            }
            else {
                return "Unknown";
            }
        }
    }
}
```

Homework 2 – MiniJava Static Checking (Semantic Analysis)

This homework introduces your semester project, which consists of building a compiler for MiniJava, a subset of Java. Mir programs can be compiled by a full Java compiler like javac.

Here is a partial, textual description of the language. Much of it **can be safely ignored** (most things are well defined in the requirement that each MiniJava program is also a Java program):

MiniJava is fully object-oriented, like Java. It does not allow global functions, only classes, fields and methods. The basic types ar is an array of int. You can build classes that contain fields of these basic types or of other classes. Classes contain methods wit types, etc. MiniJava supports single inheritance but not interfaces. It does not support function overloading, which means that unique. In addition, all methods are inherently polymorphic (i.e., “virtual” in C++ terminology). This means that foo can be defi same return type and arguments as in the parent, but it is an error if it exists with other arguments or return type in the parent. return type--there are no void methods. Fields in the base and derived class are allowed to have the same names, and are essen All MiniJava methods are “public” and all fields “protected”. A class method cannot access fields of another class, with the Methods are visible, however. A class's own methods can be called via “this”. E.g., this.foo(5) calls the object's own foo method, of object a. Local variables are defined only at the beginning of a method. A name cannot be repeated in local variables (of the repeated in fields (of the same class). A local variable x shadows a field x of the surrounding class. In MiniJava, constructors an The new operator calls a default void constructor. In addition, there are no inner classes and there are no static methods or fiel static method “main” is handled specially in the grammar. A MiniJava program is a file that begins with a special class that c specific arguments that are not used. The special class has no fields. After it, other classes are defined that can have fields and m Notably, an A class can contain a field of type B, where B is defined later in the file. But when we have "class B extends A", A you'll notice in the grammar, MiniJava offers very simple ways to construct expressions and only allows < comparisons. There ar + 2 + 3, but a method call on one object may be used as an argument for another method call. In terms of logical operators, M ("&&") and the logical not ("!"). For int arrays, the assignment and [] operators are allowed, as well as the a.length expression, wh We have “while” and “if” code blocks. The latter are always followed by an “else”. Finally, the assignment "A a = new B();" when B same applies when a method expects a parameter of type A and a B instance is given instead. The MiniJava grammar in BNF can make small changes to grammar, but you must accept everything that MiniJava accepts and reject anything that is reject Making changes is not recommended because it will make your job harder in subsequent homework assignments. Normally grammar.

The MiniJava grammar in JavaCC form is here. You will use the JTB tool to convert it into a grammar that produces class hierar or more visitors who will take control over the MiniJava input file and will tell whether it is semantically correct, or will print an err for the compiler to report precisely what error it encountered and compilation can end at the first error. But you should not miss er programs.

The visitors you will build should be subclasses of the visitors generated by JTB, but they may also contain methods and fields to checking, to transfer information from one visitor to the next, etc. In the end, you will have a Main class that runs the semantic an was produced by JavaCC and executing the visitors you wrote. You will turn in your grammar file, if you have made changes, oth by JavaCC and JTB alongside your own classes that implement the visitors, etc. and a Main. The Main should parse and statica that are given as arguments.

Also, for every MiniJava file, your program should store and print some useful data for every class such as the names and the method this class contains. For MiniJava we have only three types of variables (int, boolean and pointers). Ints are stored in 4 pointers in 8 bytes (we consider functions and int arrays as pointers). Corresponding offsets are shown in the example below:

Input:

```
class Main {
    public static void main(String[] a) {System.out.println(new A().foo());}
}


class A {
    int i;
    boolean flag;
    int j;
    public int foo() {}
    public boolean fa() {}
}

class B extends A {
    A type;
    int k;
    public boolean bla() {}
    public int foo() {}
}
```

Output:

```
A.i : 0
A.flag : 4
A.j : 5
A.foo : 0
A.fa: 8
B.type : 9
B.k : 17
B.bla : 16
```

There will be a tutorial for JavaCC and JTB. You can use these files as MiniJava examples and to test your program. Obviously own files, however the homework will be graded purely on how your compiler performs on all the files we will test it against (bo others). You can share ideas and test files, but you are not allowed to share code.

Your program should run as follows:

```
java [MainClassName] [file1] [file2] ... [fileN]
```

That is, your program must perform semantic analysis on all files given as arguments. May the Force be with you!

Homework 3 - Generating intermediate code (MiniJava -> LLVM)

In this part of the project you have to write visitors that convert MiniJava code into the intermediate representation used by th MiniJava language is the same as in the previous exercise. The LLVM language is documented in the LLVM Language Refere use only a subset of the instructions.

Types

Some of the available types that might be useful are:

- `i1` - a single bit, used for booleans (practically takes up one byte)
- `i8` - a single byte
- `i8*` - similar to a char* pointer
- `i32` - a single integer
- `i32*` - a pointer to an integer, can be used to point to an integer array
- static arrays, e.g., `[20 x i8]` - a constant array of 20 characters

Instructions to be used

- `declare` is used for the declaration of external methods. Only a few specific methods (e.g., `calloc` , `printf`) need t
  Example: `declare i32 @puts(i8*)`
- `define` is used for defining our own methods. The return and argument types need to be specified, and the metho instruction of the same type.
  Example: `define i32 @main(i32 %argc, i8** argv) {...}`
- `ret` is the return instruction. It is used to return the control flow and a value to the caller of the current function. Example:
- `alloca` is used to allocate space on the stack of the current function for local variables. It returns a *pointer* to the given t the method returns.
  Example: `%ptr = alloca i32`
- `store` is used to store a value to a memory location. The parameters are the value to be stored and a pointer to the mem
  Example: `store i32 %val, i32* %ptr`
- `load` is used to load a value from a memory location. The parameters are the type of the value and a pointer to the mem
  Example: `%val = load i32, i32* %ptr`
- `call` is used to call a method. The result can be assigned to a register. (LLVM bitcode temporary variables are called "re parameters (with their types) need to be specified.
  Example: `%result = call i8* @calloc(i32 1, i32 %val)`
- `add, and, sub, mul, xor` are used for mathematical operations. The result is the same type as the operands.
  Example: `%sum = add i32 %a, %b`

- `icmp` is used for comparing two operands. `icmp slt` for instance does a signed comparison of the operands and operand is less than the second, otherwise `i1 0` .
  Example: `%case = icmp slt i32 %a, %b`

- `br` with a `i1` operand and two labels will jump to the first label if the `i1` is one, and to the second label otherwise.
  Example: `br i1 %case, label %if, label %else`

- `br` with only a single label will jump to that label.
  Example: `br label %goto`

- `label:` declares a label with the given name. The instruction before declaring a label needs to be a `br` operation, eve to the label.
  Example: `label123:`

- `bitcast` is used to cast between different pointer types. It takes the value and type to be cast, and the type that it will be
  Example: `%ptr = bitcast i32* %ptr2 to i8**`

- `getelementptr` is used to get the pointer to an element of an array from a pointer to that array and the index of the pointer to the type that is passed as the first parameter (in the case below it's an `i8*`). This example is like doing `ptr_i` still need to do a `load` to get the actual value at that position).
  Example: `%ptr_idx = getelementptr i8, i8* %ptr, i32 %idx`

- `constant` is used to define a constant, such as a string. The size of the constant needs to be declared too. In the ex bytes (`[12 x i8]`). The result is a pointer to the given type (in the example below, `@.str` is a `[12 x i8]*`).
  Example: `@.str = constant [12 x i8] c"Hello world\00"`

- `global` is used for declaring global variables - something you will need to do for creating v-tables. Just like `constant` given type.
  Example:
  `@.vtable = global [2 x i8*] [i8* bitcast (i32 ()* @func1 to i8*), i8* bitcast (i8* (i32, i`

- `phi` is used for selecting a value from previous basic blocks, depending on which one was executed before the current b the first in a basic block. It takes as arguments a list of pairs. Each pair contains the value to be selected and the predecess necessary in single-assignment languages, in places where multiple control-flow paths join, such as if-else statements, i from the different paths. In the context of the exercise, you will need this for short-circuiting and (&&) expressions.
  Example:

```
br i1 1, label %lb1, label %lb2
lb1:
    %a = add i32 0, 100
    br label %lb3
lb2:
    %b = add i32 0, 200
    br label %lb3
lb3:
    %c = phi i32 [%a, %lb1], [%b, %lb2]
```

V-table

If you do not remember or haven't seen how a virtual table (v-table) is constructed, essentially it is a table of function pointers, pc an object. The v-table defines an address for each dynamic function the object supports. Consider a function `foo` in position 0 table (with actual offset 8). If a method is overridden, the overriding version is inserted in the same location of the virtual table as calls are implemented by finding the address of the function to call through the virtual table. If we wanted to depict this in C, located at location `x` and we are calling `foo` which is in the 3rd position (offset 16) of the v-table. The address of the function memory location `(*x) + 16` .

Execution

You will need to execute the produced LLVM IR files in order to see that their output is the same as compiling the input java file with `java` . To do that, you will need Clang with version >=4.0.0. You may download it on your Linux machine, or use it via SSH

In Ubuntu Trusty

1. `sudo apt update && sudo apt install clang-4.0`
2. Save the code to a file (e.g. `ex.ll`)
3. `clang-4.0 -o out1 ex.ll`
4. `./out1`

In linuxvm machines

1. `/home/users/thp06/clang/clang -o out1 ex.ll`
2. `./out1`

Deliverable

Your program should run as follows:
`java [MainClassName] [file1.java] [file2.java] ... [fileN.java]`
That is, your program must compile to LLVM IR all .java files given as arguments. Moreover, the outputs must be stored `file2.ll` , ... `fileN.ll` respectively.

Tips

- You will need to use a lot of registers in order to 'glue' expressions together. This means that each visitor will produce the cc expression to a register, and then return the name of that register so that other expressions may use it, if necessary.
- Registers are single-assignment. This means you can only write to them once (but read any number of times). This also imp used for local variables of the source program. Instead, you will allocate space on the stack using `alloca` and keep the a use the `load` and `store` instructions to read and write to that local variable.

- Because registers are single-assignment, you will probably need to keep a counter to produce new ones. For example, you form `%_1`, `%_2`, etc.
- You will only support compilation to a 64-bit architecture: pointers are 8-bytes long.
- Everything new in Java is initialized to zeroes.
- Memory allocated with `@calloc` will leak since you're not implementing a Garbage Collector, but that's fine for this homew
- You will need to check each array access in order not to write or read beyond the limits of an array. If an illegal read/write is message "Out of bounds" and the program will exit (you may call the `@throw_oob` defined below for that). Of course, you an array for that.
- You will also need to check if an array is allocated with a negative length, and do the same process as above in that case.
- You may see some examples of LLVM code produced for different Java input files here (corresponding to the earlier MiniJa
- You may define the following helper methods once in your output files, in order to be able to call `@calloc`, `@print_int`

```
 declare i8* @calloc(i32, i32)
declare i32 @printf(i8*, ...)
declare void @exit(i32)

@_cint = constant [4 x i8] c"%d\0a\00"
@_cOOB = constant [15 x i8] c"Out of bounds\0a\00"
define void @print_int(i32 %i) {
    %_str = bitcast [4 x i8]* @_cint to i8*
    call i32 (i8*, ...) @printf(i8* %_str, i32 %i)
    ret void
}

define void @throw_oob() {
    %_str = bitcast [15 x i8]* @_cOOB to i8*
    call i32 (i8*, ...) @printf(i8* %_str)
    call void @exit(i32 1)
    ret void
}
```

Example program

The program below demonstrates all of the above instructions. It creates an array of 3 methods (add, sub and mul), calls all of th and prints the results.

```
 @.funcs = global [3 x i8*] [i8* bitcast (i32 (i32*, i32*)* @add to i8*),
                             i8* bitcast (i32 (i32*, i32*)* @sub to i8*),
                             i8* bitcast (i32 (i32*, i32*)* @mul to i8*)]

declare i32 @printf(i8*, ...)
@.comp_str = constant [15 x i8] c"%d %c %d = %d\0A\00"
@.ret_val = constant [20 x i8] c"Returned value: %d\0A\00"

define i32 @main() {
    ; allocate local variables
    %ptr_a = alloca i32
    %ptr_b = alloca i32
    %count = alloca i32

    ; initialize var values
    store i32 100, i32* %ptr_a
    store i32 50, i32* %ptr_b
    store i32 0, i32* %count
    br label %loopstart

  loopstart:
    ;load %i from %count
    %i = load i32, i32* %count
    ; while %i < 3
    %fin = icmp slt i32 %i, 3
    br i1 %fin, label %next, label %end

  next:
    ; get pointer to %i'th element of the @.funcs array
    %func_ptr = getelementptr [3 x i8*], [3 x i8*]* @.funcs, i32 0, i32 %i
    ; load %i'th element that contains an i8* to the method
    %func_addr = load i8*, i8** %func_ptr
    ; cast i8* to actual method type in order to call it
    %func = bitcast i8* %func_addr to i32 (i32*, i32*)*
    ; call casted method
    %result = call i32 %func(i32* %ptr_a, i32* %ptr_b)

    ; print result
    %str = bitcast [20 x i8]* @.ret_val to i8*
    call i32 (i8*, ...) @printf(i8* %str, i32 %result)

    ; increase %i and store to %count
    %next_i = add i32 %i, 1
    store i32 %next_i, i32* %count
    ; go to loopstart
    br label %loopstart

  end:
    ret i32 0
}
```

```
define i32 @add(i32* %a, i32* %b) {
    %str = bitcast [15 x i8]* @.comp_str to i8*

    ; load values from addresses
    %val_a = load i32, i32* %a
    %val_b = load i32, i32* %b

    ; add them and print the result
    %res = add i32 %val_a, %val_b
    call i32 (i8*, ...) @printf(i8* %str, i32 %val_a, [1 x i8] c"+", i32 %val_b, i32 %res)

    ; return the result
    ret i32 %res
}

define i32 @sub(i32* %a, i32* %b) {
    ; similar as above
    %str = bitcast [15 x i8]* @.comp_str to i8*
    %val_a = load i32, i32* %a
    %val_b = load i32, i32* %b
    %res = sub i32 %val_a, %val_b
    call i32 (i8*, ...) @printf(i8* %str, i32 %val_a, [1 x i8] c"-", i32 %val_b, i32 %res)
    ret i32 %res
}

define i32 @mul(i32* %a, i32* %b) {
    ; similar as above
    %str = bitcast [15 x i8]* @.comp_str to i8*
    %val_a = load i32, i32* %a
    %val_b = load i32, i32* %b
    %res = mul i32 %val_a, %val_b
    call i32 (i8*, ...) @printf(i8* %str, i32 %val_a, [1 x i8] c"*", i32 %val_b, i32 %res)
    ret i32 %res
}
```