

K31 Compilers

Spring Semester 2019

June 3, 2019

Compilers

Course Description

Lectures

Tutorials

Project

Piazza course link

Compiler Tools

Course Project

Design and implementation of a compiler for the MiniJava language (a small subset of Java)

To implement the compiler you will use the tools JavaCC and JTB

The implementation for phases 2 and 3 of the project will be done in Java utilizing the visitor pattern

Homework	Description
<u>1</u>	Implementation of a LL(1) parser for a simple calculator and a translator to Java for a language for operations
<u>2</u>	Semantic Check (MiniJava)
<u>3</u>	Generating intermediate code (MiniJava -> LLVM)

Homework 1 - LL(1) Calculator Parser - Translator to Java

Part 1

For the first part of this homework you should implement a simple calculator. The calculator should accept expressions and XOR(^) operators, as well as parentheses. The grammar (for single-digit numbers) is summarized in:

exp -> num | exp op exp | (exp)

op -> ^ | &

num -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

You need to change this grammar to support priority between the two operators, to remove the left recursion for LL

This part of the homework is divided in two parts:

1. For practice, you can write the FIRST+ & FOLLOW sets for the LL(1) version of the above grammar. In the end, you should create a single lookahead table (include a row for every derivation in your final grammar). This part will not be graded.
2. You have to write a recursive descent parser in Java that reads expressions and computes the values or reports a syntax error. You don't need to identify blank space or multi-digit numbers. You can read the symbols one by one (using a function). The expression must end with a newline or EOF.

Part 2

In the second part of this homework you will implement a parser and translator for a language supporting strings, function definitions and calls, conditionals (if-else i.e., every "if" statement) and the following logical expressions:

- is-prefix-of (string1 prefix string2): Whether string1 is a prefix of string2.
- is-suffix-of (string1 suffix string2): Whether string1 is a suffix of string2.

All values in the language are strings.

Your parser, based on a context-free grammar, will translate the input language into Java. You will use JavaCUP combined either with a hand-written lexer or a generated-one (e.g., using JFlex, which is encouraged).

You will infer the desired syntax of the input and output languages from the examples below. The output language is Java, compiled using the "javac" command and executed using the "java" command or online Java compilers like [this](#), if you prefer.

There is no need to perform type checking for the argument types or a check for the number of function arguments. The program input will always be semantically correct.

Note that each file of Java source code you produce must have the same name as the public Java class in it. For example, if the public class is "Main" and the generated files are "Main.java". In order to compile a file named Main.java you need to execute: javac Main.java. In order to execute the produced Main.class file you need to execute: java Main.

To execute the program successfully, the "Main" class of your Java program must have a method with the following signature: main(String[] args), which will be the main method of your program, containing all the translated statements of the input program. For each function declaration of the input program, the translated Java program must contain an equivalent static method. Keep in mind that in the input language the function declarations must precede all statements.

Example #1

Input:

```

name() {
    "John"
}

surname() {
    "Doe"
}

fullname(first_name, sep, last_name) {
    first_name + sep + last_name
}

name()
surname()
fullname(name(), " ", surname())

```

Output (Java):

```

public class Main {
    public static void main(String[] args) {
        System.out.println(name());
        System.out.println(surname());
        System.out.println(fullname(name(), " ", surname()));
    }

    public static String name() {
        return "John";
    }

    public static String surname() {
        return "Doe";
    }

    public static String fullname(String first_name, String sep, String last_name) {
        return first_name + sep + last_name;
    }
}

```

Example #2

Input:

```

name() {
    "John"
}

repeat(x) {
    x + x
}

cond_repeat(c, x) {
    if (c prefix "yes")
        if("yes" prefix c)
            repeat(x)
        else
            x
    else
        x
}

cond_repeat("yes", name())
cond_repeat("no", "Jane")

```

Example #3

Input:

```

findLangType(langName) {
    if ("Java" prefix langName)
        if(langName prefix "Java")
            "Static"
        else
            if("script" suffix langName)
                "Dynamic"
            else
                "Unknown"
    else
        if ("script" suffix langName)

```

```

        "Probably Dynamic"
    else
        "Unknown"
}

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

```

Homework 2 – MiniJava Static Checking (Semantic Analysis)

This homework introduces your semester project, which consists of building a compiler for MiniJava, a subset of that its 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 from 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. It has boolean, and int [] which is an array of int. You can build classes that contain fields of these basic types or of methods with arguments of basic or class types, etc.
- MiniJava supports single inheritance but not interfaces. It does not support function overloading, which means that a method is unique. In addition, all methods are inherently polymorphic (i.e., "virtual" in C++ terminology). This means that a subclass can override a method if it has the same return type and argument types (ordered) as in the parent, but it is an error if it has a different return type or argument types. Also all methods must have a return type--there are no void methods. Fields in a class are allowed to have the same names, and are essentially different fields.
- All MiniJava methods are "public" and all fields "protected". A class method cannot access fields of another class or superclass. Methods are visible, however. A class's own methods can be called via "this". E.g., this.foo(5) calls the foo method, a.foo(5) calls the foo method of object a. Local variables are defined only at the beginning of a method and are repeated in local variables (of the same method) and cannot be repeated in fields (of the same class). A local variable can be the same as the surrounding class.
- In MiniJava, constructors and destructors are not defined. The new operator calls a default void constructor for each class and there are no static methods or fields. By exception, the pseudo-static method "main" is handled. A MiniJava program is a file that begins with a special class that contains the main method and specific arguments. This special class has no fields. After it, other classes are defined that can have fields and methods. Notably, an A class can contain a field of type B, where B is defined later in the file. But when we have a field of type B defined before B. As you'll notice in the grammar, MiniJava offers very simple ways to construct expressions. There are no lists of operations, e.g., 1 + 2 + 3, but a method call on one object may be used as an expression. In terms of logical operators, MiniJava allows the logical and ("&&") and the logical not ("!"). For array operators are allowed, as well as the a.length expression, which returns the size of array a. We have "new B()" for creating a new B object. The latter are always followed by an "else". Finally, the assignment "A a = new B();" when B extends A is correct. A method expects a parameter of type A and a B instance is given instead.

The MiniJava grammar in BNF can be downloaded [here](#). You can make small changes to grammar, but you must accept and reject anything that is rejected by the full Java language. Making changes is not recommended because of subsequent homework assignments. Normally you won't need to touch the grammar.

The MiniJava grammar in JavaCC form is [here](#). You will use the JTB tool to convert it into a grammar that produces a visitor who will write one or more visitors who will take control over the MiniJava input file and will tell whether it is semantically correct. It isn't necessary for the compiler to report precisely what error it encountered and compilation can end without missing errors or reporting errors in correct programs.

The visitors you will build should be subclasses of the visitors generated by JTB, but they may also contain methods during static checking, to transfer information from one visitor to the next, etc. In the end, you will have a Main class that initiates the parser that was produced by JavaCC and executing the visitors you wrote. You will turn in your code, made changes, otherwise just the code produced by JavaCC and JTB alongside your own classes that implement the Main. The Main should parse and statically check all the MiniJava files 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 variables and methods this class contains. For MiniJava we have only three types of variables (int, boolean and pointers). Pointers are in 1 byte and pointers in 8 bytes (we consider functions and int arrays as pointers). Corresponding offsets are given below:

Input:

```

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

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

```

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 (you can make up your own files, however the homework will be graded purely on how your compiler performs on all the files (including the above sample files and 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 of the LLVM project. The MiniJava language is the same as in the previous exercise. The LLVM language is documented in the [Manual](#), although you will 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`) are declared. Example: `declare i32 @puts(i8*)`
- **define** is used for defining our own methods. The return and argument types need to be specified, and the `ret` 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.
- **alloca** is used to allocate space on the stack of the current function for local variables. It returns a *pointer* to the allocated space, which is freed when 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 memory location. 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 memory location. 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 used for the result and 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: if the first 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 if it is zero. 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** or **goto** 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.
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. It also takes a pointer to the type that is passed as the first parameter (in the case below it's an `i8*`).
`ptr_idx = &ptr[idx]` in C (you 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. The string is 12 bytes (`[12 x i8]`). The result is a pointer to the given type (in the example below, `@.str` is a pointer to `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 declare a pointer to the given type.
Example:
`@.vtable = global [2 x i8*] [i8* bitcast (i32 ()* @func1 to i8*), i8* bitcast (i8* (i32 ()*) to i8*)]`
- **phi** is used for selecting a value from previous basic blocks, depending on which one was executed. Instructions must be the first in a basic block. It takes as arguments a list of pairs. Each pair contains the predecessor block for that value. This is necessary in single-assignment languages, in places where multiple assignments occur, as if-else statements, if one wants to select a value from the different paths. In the context of the exercise, it is used for circuiting and (&&) expressions.
Example:

```
br i1 1, label %b1, label %b2
b1:
  %a = add i32 0, 100
  br label %b3
b2:
  %b = add i32 0, 200
  br label %b3
b3:
  %c = phi i32 [%a, %b1], [%b, %b2]
```

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. The v-table defines an address for each dynamic function the object supports. Consider the example of a `bar` method in position 1 of the table (with actual offset 8). If a method is overridden, the overriding version is inserted into the virtual table as the overridden version. Virtual calls are implemented by finding the address of the function to call from the v-table. To depict this in C, imagine that object `obj` is located at location `x` and we are calling `foo` which is in the v-table. The address of the function that is going to be called is in 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 program and executing it with `java`. To do that, you will need Clang with version `>=4.0.0`. You may download it on your Linux machine or use the linuxvm machines.

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 `file1.ll`, `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 return the value of an expression to a register, and then return the name of that register so that other expressions can use it.
- Registers are single-assignment. This means you can only write to them once (but read any number of times). This is necessary for the compiler to be able to use registers for local variables of the source program. Instead, you will allocate space on the stack and store the address in a register. You will 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, registers of the 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
- You will need to check each array access in order not to write or read beyond the limits of an array. If an illegal access will print the message "Out of bounds" and the program will exit (you may call the `@throw_oob` defined below to know the length of 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
- You may see some examples of LLVM code produced for different Java input files [here](#) (corresponding to the from HW2).
- You may define the following helper methods once in your output files, in order to be able to call `@calloc`, `@throw_oob`.

```
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) arguments 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
}
```