Scala−−, a type inferred language Project Report

Da Liu dl2997@columbia.edu

Contents

1 Background

Scala is becoming drawing attentions in daily production among various industries. Being as a general purpose programming language, it is influenced by many ancestors including, Erlang, Haskell, Java, Lisp, OCaml, Scheme, and Smalltalk. Scala has many attractive features, such as cross-platform taking advantage of JVM; as well as with higher level abstraction agnostic to the developer providing immutability with persistent data structures, pattern matching, type inference, higher order functions, lazy evaluation and many other functional programming features. . Scala is truly good at breaking down complex large scale projects into manageable small solutions that work together in a functional fashion.

LLVM as a very powerful compiler infrastructure providing many advantages in compiler optimization and interfacing with many languages to ease the compiler backend writing magnificently.

2 Introduction

Scala−− is a prototype towards to be a full-fledged production-ready functional programming language with full support of current version of Scala. It has fast startup time and potentially be able to leverage LLVM optimization/analyse. The prototype compiler translates source code with a subset of Scala syntax to LLVM IR, The intermediate representation from frontend of the compiler is implemented in OCaml, machine code from LLVM eventually running on all LLVM-supported architectures.

The "frontend" means the beginning part of the compiler, which consists of lexical analyzer, tokenizer, abstract syntax tree generator, semantic checker, the end product (intermediate representation) of this compiler frontend is the input of the subsequent "backend" compiler, LLVM language binding is used to create this part. So taking advantage of LLVM, the final output from this compiler is low level machine code-generated executables; i.e., writing clean and concise, easy to maintain Scala code with functional programming features, while achieving performance approaching to assembly languages.

3 Type Inference

Type inference for programming languages is based on logical systems and constraint resolution. It is highly idispendisable in conpempotary higher-order, polymorphic languages, improved local type inference in

Scala and advanced type specification system in Elixir/Erlang for example. Hindley-Milner typing rules are the central concept of many *complete* type inference algorithms which have been implemented in many type inferred languages including the aforementioned ones as well as in `ML`, `OCaml` and `Haskell`. `Scala`$^{--}$ has the potential of having the full implementation of a complete global type inference algorithm with respect to Hindley-Milner rules, in ordre to better infer all the forms of values including function values and potentially class objects when object-oriented element is introduced to the languager, and/or generalized algebric data types (GADT); it is possible but hard, for the latter case in the general sence. The following equations abstract the fundamental idea of the Hindley-Milner type inference rules.

$$\frac{\tau \prec \Gamma(x)}{\Gamma \vdash_{\mathrm{HM}} \ x : \tau} \ \textsc{Var}$$

$$\frac{\Gamma \vdash_{\mathrm{HM}} \ e_1 : \tau_1 \to \tau_2 \ \ \Gamma \vdash_{\mathrm{HM}} \ e_2 : \tau_1}{\Gamma \vdash_{\mathrm{HM}} \ e_1 \ e_2 : \tau_2} \ \textsc{App}$$

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \ \vdash_{\mathrm{HM}} \ e : \tau_2}{\Gamma \ \vdash_{\mathrm{HM}} \ \lambda x \to \ e : \tau_1 \to \tau_2} \ \textsc{Abs}$$

$$\frac{\Gamma \vdash_{\mathrm{HM}} \ e_1 : \tau_1 \ \Gamma \backslash x \cup \{x : generalize(\Gamma, \tau_1)\} \vdash_{\mathrm{HM}} \ e_2 : \tau_2}{\Gamma \ \vdash_{\mathrm{HM}} \ \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ \textsc{Let}$$

In the Hindley-Milner rules, each one is responsible for one basic language constructs, illustrated for type variables, applications, lambda and let expression; in general, under type environement Γ, a type τ can be assigned to an expression e.

4 Language Prototype Features

The LLVM-Scala−− *Prototype* is succinct, statically typed, functional programming, JVM-free language with relatively sophisticated type system.

The Prototype shares the same file suffix .scala as the Scala programming language. Only a small subset of Scala features is supported in this Prototype, which were documented in this *Proposal*; all others that were not mentioned in this Proposal is not be supported or be supported in a very limited way. From the language processing mode point of view, only compilation is supported in the Prototype. Object-oriented feature is omitted. For instance, there is no `trait`, or `class`. Another example was supporting `import` in a limited way, in that only *supported* library or Prototype-compatible code would be allowed to imported into a Prototype code and there is no definition of `package`. Further more, there is no support for named arguments, variable number of arguments, companion objects, object comparison, nested functions, nested classes, Actors, file I/O, or exception handling, The undecidable supporting features is be documented in future report when necessary, for instance, the range difference between the Prototype and Scala for Int values.

5 Project Design

- Compiler
  - LLVM-IR
  - Support x86 and ARM
  - Support a subset of Scala features[1]
- Benchmarking
  - Testing code
    - Basic algorithms written in Scala, or the targeting Scala-like language
    - Algorithm testing code counterpart in OCaml and/or other programming languages
  - Testing compiler For comparison purposes, to see the differences between the project and existing compilers and interpreters
    - target compiler (LLVM)
    - gcc/g++
    - Sun Java Development Kit (JDK)
    - Perl
    - Python
  - Benchmark utilities

6 Implementation

Figure 6.1 shows the `Scala−−` implementation work flow as well as the data flow in accord to a typical compile working cycle.

llvm::PassManager can be used to do automatic code flow optimization with LLVM. However, in the current, it is still working in progress. A container for all of the passes which can be run. The passes are basically work on the IR to either do some analysis, which need not alter the IR, or some optimization or transform, which may alter the IR. Custom optimizations also beyond of the consideration of the project. Howver, it might be useful to know that there is a pre-built pass which simply allows you to output the IR into a human readable representation of LLVM assembly, llvm::createPrintModulePass. Also as already shown in Figure 6.1, this design follows a very typical flow of compiler front end.

6.1 Benchmarking results

Simple Hello World programs print out to `STDOUT` and recursive version of Fibonacci number 20 calculation were implemented in Scala and Scala−−, respectively, with the difference being the `object` header for the Scala language as shown in the Figure 6.2 and Figure 6.3. The Fibonacci was running for 200 cycles with `bash for-loop` control. Both programs output were redirected to `/dev/null` and `sync`-ed

[1] Details mentioned in the section of *Language Prototype Features*

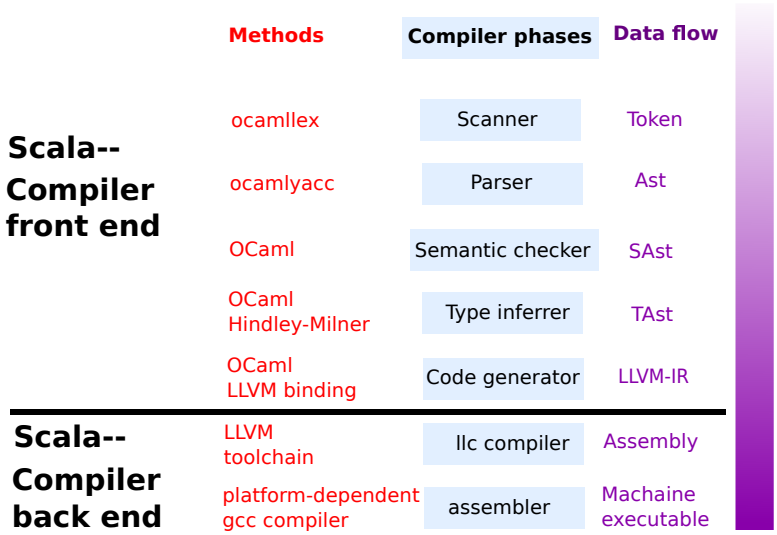

Figure 6.1: Schemetic shows `Scala−−` compiler architecture with OCaml and LLVM toolchains.

before each run; each ran three times for compilation and execution, respectively. Note tha `sbt` can boost Scala performance to some extent.

A typical timing routine for compilation processes was used as the following for Scala−−:

```
 sync && time ./scalal.native -c < bench-hello.scala >
     bench-hello.ll; llc bench-hello.ll -o hello.s;gcc
     hello.s;
and
 sync && time ./scalal.native -c < bench-fib.scala >
     bench-fib.ll;llc bench-hello.ll -o hello.s;gcc
     hello.s.
```

A typical timing routine for compilation processes was used as the following for Scala:

```
sync && time scalac bench-hello.scala;
sync && time scalac bench-fib.scala.
```

Similarily, a typical timing routine for execution processes was as the following for Scala−−:

```
sync && time ./hello.out /dev/null,
sync && time for i in {1..200};do ./fib.out >
    /dev/null;done}.
sync && time for i in {1..200};do scala Fib >
    /dev/null;done}.
```

Above also shows similar routes used for benchmarking Scala execution.

Here the Fibonacci benchmarking code for `Scala` and `Scala−−` is also shown in Listing 6.1 and Listing 6.1, respectively.

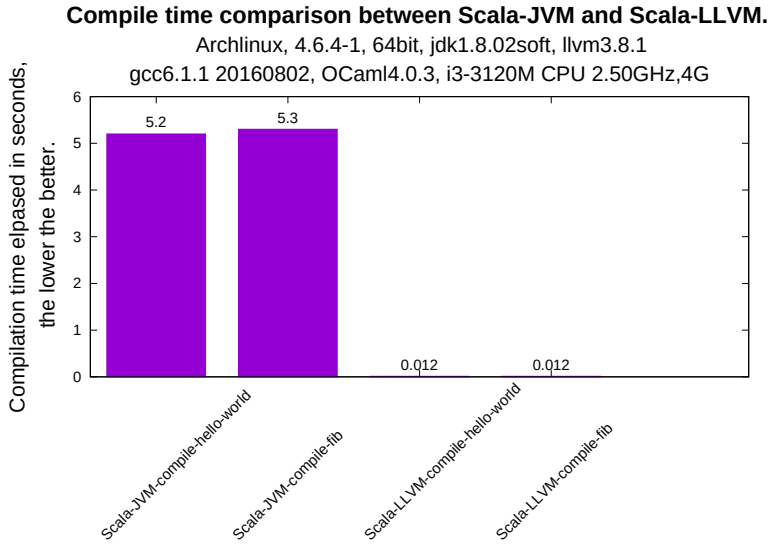

Figure 6.2: Benchmarking shows comparison of compile time between JVM-based Scala and LLVM-based Scala−−

```
object Fib {
  def fib (x : Int) :Int = {
    if (x < 2) return 1
    return fib(x - 1) + fib(x -2)
  }
  def main (args: Array[String]) {
    print(fib(20));
  }
}
```

```
/* Recursion verion of Fibonacci calculation. */
def fib = (var x : int) : int
{
 if (x < 2) return 1;
 return fib(x-1) + fib(x-2);
}

def main = () : int
{
 print(fib(20)); // Calculate the 20th Fibonacci number

 return 0;
}
```

Listing 1: LLVM-IR generated from Scala–

```
; ModuleID = 'ScalaL'

@fmt = private unnamed_addr constant [4 x i8]
    c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8]
```

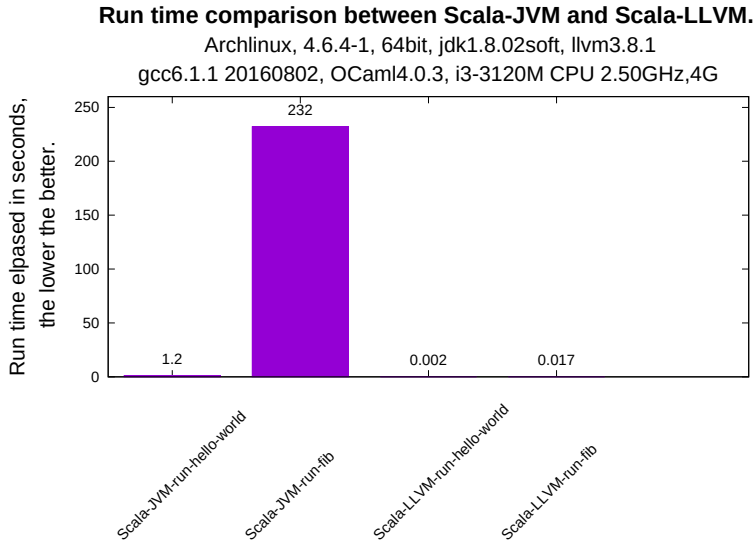

Figure 6.3: Benchmarking shows comparison of execution time between JVM-based Scala and LLVM-based Scala−−

```
    c"%d\0A\00"

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

define i32 @main() {
entry:
  %fib_result = call i32 @fib(i32 20)
  %printf = call i32 (i8*, ...) @printf(i8*
      getelementptr inbounds ([4 x i8], [4 x i8]* @fmt,
      i32 0, i32 0), i32 %fib_result)
  ret i32 0
}

define i32 @fib(i32 %x) {
entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %x2 = load i32, i32* %x1
  %tmp = icmp slt i32 %x2, 2
  br i1 %tmp, label %then, label %else

merge:                                    ; preds = %else
  %x3 = load i32, i32* %x1
  %tmp4 = sub i32 %x3, 1
  %fib_result = call i32 @fib(i32 %tmp4)
  %x5 = load i32, i32* %x1
  %tmp6 = sub i32 %x5, 2
  %fib_result7 = call i32 @fib(i32 %tmp6)
  %tmp8 = add i32 %fib_result, %fib_result7
  ret i32 %tmp8

then:                                     ; preds = %entry
```

```
  ret i32 1

else:                                             ; preds = %entry
  br label %merge
}
```

Listing 2: Assembly code generated from llc

```
   .text
   .file "a.ll"
   .globl main
   .align 16, 0x90
   .type main,@function
main:                                   # @main
   .cfi_startproc
# BB#0:                                 # %entry
   pushq %rax
.Ltmp0:
   .cfi_def_cfa_offset 16
   movl $20, %edi
   callq fib
   movl %eax, %ecx
   movl $.Lfmt, %edi
   xorl %eax, %eax
   movl %ecx, %esi
   callq printf
   xorl %eax, %eax
   popq %rcx
   retq
.Lfunc_end0:
   .size main, .Lfunc_end0-main
   .cfi_endproc

   .globl fib
   .align 16, 0x90
   .type fib,@function
fib:                                    # @fib
   .cfi_startproc
# BB#0:                                 # %entry
   pushq %rbx
.Ltmp1:
   .cfi_def_cfa_offset 16
   subq $16, %rsp
.Ltmp2:
   .cfi_def_cfa_offset 32
.Ltmp3:
   .cfi_offset %rbx, -16
   movl %edi, 12(%rsp)
   cmpl $2, %edi
   jge  .LBB1_1
# BB#2:                                 # %then
   movl $1, %eax
   jmp  .LBB1_3
.LBB1_1:                                # %merge
   movl 12(%rsp), %edi
   decl %edi
```

```
  callq fib
  movl %eax, %ebx
  movl 12(%rsp), %edi
  addl $-2, %edi
  callq fib
  addl %ebx, %eax
.LBB1_3:                                # %then
  addq $16, %rsp
  popq %rbx
  retq
.Lfunc_end1:
  .size fib, .Lfunc_end1-fib
  .cfi_endproc

  .type .Lfmt,@object    # @fmt
  .section .rodata.str1.1,"aMS",@progbits,1
.Lfmt:
  .asciz "%d\n"
  .size .Lfmt, 4

  .type .Lfmt.1,@object # @fmt.1
.Lfmt.1:
  .asciz "%d\n"
  .size .Lfmt.1, 4


  .section ".note.GNU-stack","",@progbits
```

7 Discussion

7.1 About scalac

One trivial aspect of the motivation of the project is to improve my user experience of Scala programming language. There are some documentation of the slowness of the Scala-JVM compiler elsewhere on the internet:

> There are two aspects to the (lack of) speed for the Scala compiler. **Greater startup overhead** Scalac itself consists of a LOT of classes which have to be loaded and jit-compiled Scalac has to search the classpath for all root packages and files. Depending on the size of your classpath this can take one to three extra seconds. Overall, expect a startup overhead of scalac of 4-8 seconds, longer if you run it the first time so disk-caches are not filled. **Type inference is costly**, in particular if it involves implicit search. Scalac has to do type checking twice; once according to Scala's rules and a second time after erasure according to Java's rules. Besides type checking there are about 15 transformation steps to go from Scala to Java, which all take time.
>
> answered Aug 31 '10 at 18:54 on stackoverflow.com
>
> Martin Odersky

8 Lessons learned

Planning is very important, some features of OCaml language intriged me deeply that I would like to spend more time digging such as GADT. During the process of the compiler implementation, I found a good tooling is a key to be efficient. Two of among those nice things I found are `utop` and `merlin` giving the types of expressions on-the-fly with hotkeys in both vim or emacs, this is really useful when debugging.

9 Language Specifications

9.1 Introdution

This manual describes Scala−−, a primitive static scoping, strong static typing, type inferred, functional programming language with immutable data structures, simple pattern matching and specialized emphasis on type inference. Scala−− written in OCaml syntactically resembles a small subset of Scala functionalities with educational purpose, that compiles to the LLVM Intermediate Representation. Several standard library functions and data structures including `map`, `filter`, List and Map are implemented in the Scala−−, providing relatively advanced immutabe operations, demonstrated the possibility of becoming a general purpose programming language. This manual describes in detail the lexical conventions, type systems, scoping rules, standard library functions, data structures and the gramar of the Scala−− language.

9.2 Lexical syntax

9.2.1 Identifiers

In Scala−−, there is only one way to form an identifier. It can start with a letter which can be followed by an arbitrary sequence of letters and digits, which may be followed by underscore '_' characters and following similar sequences, which can be defined by the following regular expression:

```
['a' - 'z' 'A' - 'Z' ]['a' - 'z' 'A' - 'Z' '0' - '9'
'_' ]*
```

9.2.2 Keywords

Scala−− has a set of reserved keywords that can not be used as identifiers.

Statements and blocks The following keywords indicate types of statement or blocks:

```
var val
```

Functions The following keyword is reserved for function-related purpose:

```
def main
```

Pattern matching The following keywords are reserved for pattern matching-related purposes:

```
match with case => _
```

Control flow The following keywords are used for control flow:

```
if else return for do while yield <-
```

Types The following primitive type keywords resemble Scala's object type:

```
type Int Float Char String Boolean Nil => :
```

Built-in functions The following keywords are reserved for built-in functions and/or data structures:

```
random, max, min, floor, map, filter, partition, println,
to, until, Map, List, Array, Tuple, insert, remove, ::,
++, ++=, ., ->
```

9.2.3 Literals

Scala−− supports boolean, integer, float, character, string, escape sequence, symbol literals.

Integer The following regular expression defines a decimal digit:

```
digit = ['0' - '9']
```

An integer of type Int is a 64-bit signed immutable value consisting of at least one digit taking the following form:

```
digit+
```

Floating point Floating point numbers can be represented as exponentials as the following regular exressions:

```
exp = 'e' ['+' '-']?  ['0' - '9']+
```

So in general, floating point numbers take the following regular expression:

```
digit'.'digit[exp] | '.'digit[exp] | digit exp | digit
[exp]
```

Boolean Boolean literals are the following:

```
boolean = ["true" | "false"]
```

Character Characters are single, 8-bit, ASCII symbols.

```
char = ['a' - 'z'] | ['A' - 'Z']
```

String A string literal is a sequence of characters in double quotes.

Escape sequence The following escape sequences are recognized in character and string literals:

\b	backspace
\t	horizontal tab
\n	linefeed
\r	carriage return
\"	double quote

\'	single quote
\\	backslash

9.2.4 Punctions

Punctions group and/or separate the primary expressions consisting of above-mentioned identifiers and literals.

`()` can indicates a list of arguments in a function declaration or function call; it can also be position access operator in built-in facilities of Array, Map and List; it can also be the boundry symbols of built-in facilities of Array, Map, List and Tuple.

`{}` defines statement blocks.

`,` represents the separator between a list of arguments in functions or a list of items in built-in data structures.

`;` is a newline character separating expressions and statements.

9.2.5 Comments and whitespace

Comments in Scala−− start with /* and terminate with */, where multiple line comments are not allowed to be nested. Single line comments take the form of //.

9.2.6 Operations

Scala−− supports several operations including artithmetic and booleans literals.

Value binding A single equsl sign indicates assignment operation or in an assignment or declaration statement:

`=`

Operators The following binary operators are supported in Scala−−:

```
/, *, %
==, !=, <=, <, >, >=
&, |
```

The following unary operators are supported:

```
!, ~
```

The following operators can be either bintary or unary, depening the context:

```
+, -
```

9.3 Syntax

9.3.1 Program structure

Scala−− program consists of decarations which are made of optional global variable and/or newline-separated and/or optionally semi-colon-separated function decarations as well as function bodies which may include variable assignment or nested function bodies.

```
program:
  declaration
  program declaration
declaration:
```

```
      fundec
      vardec newline
      newline
```

Variable declarations and type inference Variables can be declared and initialized globally, or locally in a function body:
`<var|val> <id-list> [ : var-type ] [ = value ] [;] <nl>,`
where `"nl"` represents newline.

Type inference Instead of *advanced local type inference* algorithm employed in Scala, Scala−− experimented type inference using one type of *complete type inference* algorithm – *Hindley-Milner type inference* algorithm which has been broadly adopted in a spate of contemporary type inferred functional programming languages such as Standard ML, OCaml and Haskell.

A variable can be declaraed as the following:

```
val myInt : Int = 17
val myFloat : Float = 3.14
val myChar : Char = 'c'
val myString : String = "Hello World!"
val myBoolean : Boolean = true
val myList : List[Int] = List(1,1,2,3,5,8)
val myMap : Map = Map( "Static typing" -> "OCaml",
    "Dynamic typing" -> "Elixir")
```

The above expressions are equvivlent to the following:

```
val myInt = 17
val myFloat = 3.14
val myChar = 'c'
val myString = "Hello World!"
val myBoolean = true
val myList = List(1,1,2,3,5,8)
val myMap = Map( "Static typing" -> "OCaml", "Dynamic
    typing" -> "Elixir")
```

Correctness of type inference also hold true when apply to function declarations regarding formal arguments and function return type which is documented in following sections.

Mutable and immutable variables Immutable variables are defined with keyword `val`, while mutable variables are defined with keyword `var` as the following:

```
val immutList = List("I" "Can" "NOT" "Be" "Modified"
    "!")
var mutString = "I am ok to be changed."
```

Function declarations Functions are defined in the following way:

```
def func-id (formal-listopt) [ :  var-typeopt ] block
```

Here `def` is a keyword starting a function declaration or definition. func-id is the identifier of a instance of the function. `block` contains the function body. `formal-listopt` is optionally required when decaring or defining a function, containing the formal arguments of `var-type`. HM type inference algorithm applies here.

```
  formal-list:
    formal-type-cluster
    formal-list, formal-type-cluster
  formal-type-cluster:
    var-type
    id-list var-type
```

For instance, a function can be declaraed with explicit specification of the argument types:

```
/* Return summation of two integer numbers */
def sumOfSquares(x: Int, y: Int): Int = {
   val x2 = x * x
      val y2 = y * y
      x2 + y2
}
```

9.3.2 Expressions

Primary expressions Primary expressions consist of literals and parathesized expression.

Precedence of operations

9.3.3 Statements

Assignments Assignment of variables requires using `=` keyword. For example:

```
val anInt = 5
var aChar = 'w'
```

9.3.4 Blocks and control flow

Conditional in Scala−−:

```
  // If statements are like Java except they return a
      value like the ternary operator
  // Conditional operators: ==, !=, >, <, <=, >=
  // Logical operators: &&, ||, !

  val age = 18
  val canVote = if (age >= 18) "yes" else "no"

  // {} is required in the REPL, but not otherwise
  if ((age >= 5) && (age <= 6)) {
```

```
    println("Go to Kindergarten")
  } else if ((age > 6) && (age <= 7)) {
    println("Go to Grade 1")
  } else {
    println("Go to Grade " + (age - 5))
  }

  true || false
  !(true)
```

There are for-loop, while-loop, if-else statement in Scala−−.

```
  /* Scala-- while-loop example: */
    var i = 0;
    while (i <= 5) {
      println(i)
      i += 1
    }

  /* Scala-- do-while-loop example: */
    do {
      println(i)
      i += 1
    } while (i <= 9)

  /* Scala-- for-loop example: */
    for (i <- 1 to 10) {
      println(i)
    }

  // until is often used to loop through strings or
      arrays
       val randLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
       //for (i <- 0 to (randLetters.length - 1)) {
       for (i <- 0 until randLetters.length) {
  // Get the letter in the index of the String
         println(randLetters(i))
       }

  // Used to iterate through a list
       val aList = List(1,2,3,4,5)
       for (i <- aList) {
         println("List iitem " + i)
       }

  // Store even numbers in a list
       var evenList = for {
         i <- 1 to 20
  // You can put as many conditions here separated with
      semicolons as you need
         if (i % 2) == 0
         if (i % 3) == 0
       } yield i

       println("Even list:")
       for (i <- evenList)
         println(i)
```

```
// This loop assigns a value to the 1st variable and
    it retains that value until the 2nd finishes its
    cycle and then it iterates
     for (i <- 1 to 5; j <- 6 to 10) {
       println("i: " + i)
       println("j: " + j)
     }
```

9.4 Scoping rules

Scala−− uses static scoping or lexical scoping just as Scala does. The scope of a variable or a function is limited to the block where it is declared, if it is not a global variable or function. That means, the policy of Scala−− allows its compiler to decide language issues; at *compile time* instead of *run time* the language issues are determined.

9.5 Standard library and collections

Several standard library functions and data structures are implemented in Scala−− to provide feature-rich coding experience.

9.5.1 println, map and filter

The following code shows an example using map and filter:

```
    // "_" is used to indicate the meant function when
        declared in order for subsequent passing
        function as an argument
    val log10Func = log10 _
    println("Log10 is: " + log10Func(1000))
    // Apply a function to all items of a list with map
    List(1000.0,
        10000.0).map(log10Func).foreach(println)

    // Filter passes only those values that meet a
        condition
    List(1,2,3,4).filter(_ % 2 == 0).foreach(println)
```

9.5.2 Array

Built-in Array has several operations to ease the manipulation, that includes `insert`, `remove`, `empty`, `+=`, `++=`.

```
    // Create and initialize array
    val friends = Array("Bob", "Tom")
    // Change the value in an array
    friends(0) = "Sue"
    println("Best friends: " + friends(0))
    // Create an ArrayBuffer
    val friends2 = ArrayBuffer[String]()
    // Add an item to the 1st index
    friends2.insert(0, "Phil")
    // Add item to the next available slot
```

```
    friends2 += "Mark"
    // Add multiple values to the next available slot
    friends2 ++= Array("Susy", "Paul")
    // Add items starting at 2nd slot
    friends2.insert(1, "Mike", "Sally", "Sam")
    // Remove the 2nd element
    friends2.remove(1)
    // Remove two elements starting at the 2nd index
    friends2.remove(1, 2)
```

9.5.3 List

List examplifies as the following:

```
    val aList = List(1,2,3,4,5)
```

9.6 Code example

9.6.1 Hello World

```
/* Hello World Example in Scala-- */
/**
 * Author: ________________
 * Description: ___________
 * Last modified:__________
 * Usage: _________________
 */
 // Function entry point starts here:
 def main(args: Array[String]) {
    pritnln("Hello, world!")
 }
```

9.6.2 QuickSort

```
    /* Quick sort in Scala-- */
    // main function invokes pre-defined function(s)
        in the same file
    def quickSort(a: List[Double]): List[Double] = a
        match {
       case Nil    => Nil
       case x :: xs =>
          val (lt, gt) = xs.partition(_ < x)
          quickSort(lt) ++ List(x) ++ quickSort(gt)
    }
    def main(args: Array[String] {
       println("Quick sort demo: " +
           quickSort(List(-2.718, 3, 3.1, 3,
           3.1415926, 14, 3.1415)))
    }
```

10 Reference

10.1 Type inference

1. M. Odersky, C. Zenger, M. Zenger. Colored Local Type Inference. *In POPL-01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 41-53, New York, NY, USA, 2001. ACM.
2. V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. *Mathematical Foundations of Computer Science*, 1-23, 2006.
3. M. Odersky, The Scala Language Spcification, *Programming Methods Laboratory*, EPFL, Switzerland, 2014.

10.2 Scala programming language

1. Martin Odersky, The Scala Language Specification, *Programming Methods Laboratory*, 2014
2. http://www.scala-lang.org/files/archive/spec/2.11/
3. http://docs.scala-lang.org/cheatsheets/

10.3 Scala programming language development

1. https://wiki.scala-lang.org/display/SIW/Compiler+Walk-Through
2. http://www.scala-lang.org/old/node/215.html
3. http://www.scala-lang.org/contribute/hacker-guide.html
4. https://github.com/lampepfl/dotty.git

10.4 Compile Scala to LLVM

1. http://vmkit.llvm.org/
2. https://github.com/scala-native/scala-native
3. https://github.com/greedy/scala
4. https://code.google.com/archive/p/slem/

1. http://lampwww.epfl.ch/ magarcia/ScalaCompilerCornerReloaded/

10.5 Benchmarking

1. http://benchmarksgame.alioth.debian.org/

11 Source code listing

```
(* Ocamllex scanner for ScalaL *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"   { comment lexbuf }    (* Comments *)
| '('    { LPAREN }
| ')'    { RPAREN }
| '{'    { LBRACE }
| '}'    { RBRACE }
| ';'    { SEMI }
| ','    { COMMA }
| '+'    { PLUS }
| '-'    { MINUS }
| '*'    { TIMES }
| '/'    { DIVIDE }
| '='    { ASSIGN }
| "=="   { EQ }
| "!="   { NEQ }
| '<'    { LT }
| "<="   { LEQ }
| ">"    { GT }
| ">="   { GEQ }
| "&&"   { AND }
| "||"   { OR }
| "!"    { NOT }
| "if"   { IF }
| "else" { ELSE }
| "for" { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "int" { INT }
| "bool" { BOOL }
| "float" { FLOAT }
| "unit" { UNIT }
| "true" { TRUE }
| "false" { FALSE }
| "def" { DEF }
| ":"    { COLON }
| "var" { VAR }
| ['0'-'9']+ as lxm { INT_LIT(int_of_string lxm) }
| ['0'-'9']*['.']['0'-'9']+ as lxm { FLOAT_LIT
    (float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as
    lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
    Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _  { comment lexbuf }
```

```
/* Ocamlyacc parser for ScalaL */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT UNIT
%token <int> INT_LIT
%token <float> FLOAT_LIT
%token <string> ID
%token DEF COLON VAR
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls vdecl { ($2 :: fst $1), snd $1 }
 | decls fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   | DEF ID ASSIGN LPAREN formals_opt RPAREN COLON typ
       LBRACE vdecl_list stmt_list RBRACE
    { { typ = $8;
    fname = $2;
    formals = $5;
    locals = List.rev $10;
    body = List.rev $11 } }
    /* XXX
   | DEF ID ASSIGN LPAREN formals_opt RPAREN LBRACE
       vdecl_list stmt_list RBRACE
    { { typinfer = TPlhdr; (* XXX Type place holder *)
    fname = $2;
    formals = $5;
    locals = List.rev $8;
    body = List.rev $9 } }
```

```
*/

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

formal_list:
   VAR ID COLON typ          { [($4,$2)] }
  | formal_list COMMA VAR ID COLON typ { ($6,$4) :: $1 }

typ:
   INT { Int }
  | BOOL { Bool }
  | FLOAT { Float }
  | UNIT { Unit }

vdecl_list:
   /* nothing */ { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   VAR ID COLON typ SEMI { ($4, $2) }

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

stmt:
   expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
      Block([])) }
  | 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) }

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

expr:
  | INT_LIT     { IntLit($1) }
  | FLOAT_LIT    { FloatLit($1) }
  | TRUE        { BoolLit(true) }
  | FALSE       { BoolLit(false) }
  | ID          { 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 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) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr     { Unop(Not, $2) }
  | ID COLON typ ASSIGN expr { Assign($1, $3, $5) } /*
      XXX */
  | ID ASSIGN expr { Assign($1, TPlhdr, $3) } /* XXX */
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list { List.rev $1 }

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

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less |
    Leq | Greater | Geq |
        And | Or

type uop = Neg | Not

type typ = Int | Bool | Float | Unit | TPlhdr

type bind = typ * string

type expr =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * typ * expr (* XXX *)
  | Call of string * expr list
  | 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

type func_decl = {
    typ : typ;
    fname : string;
```

```
   formals : bind list;
   locals : bind list;
   body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

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_typ = function
   Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Unit -> "unit"
  | TPlhdr -> "typeplaceholder" (* XXX *)

let rec string_of_expr = function
   IntLit(l) -> string_of_int l
  | FloatLit (f) -> string_of_float f
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | 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
  | Assign(v, t, e) -> v ^ string_of_typ t ^ " = " ^
     string_of_expr e (* XXX *)
  | Call(f, el) ->
     f ^ "(" ^ String.concat ", " (List.map
         string_of_expr el) ^ ")"
  | Noexpr -> ""

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

```
    ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^
    ")\n" ^ string_of_stmt s
  | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
   string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | 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

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^
   id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd
    fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl
    fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^
    "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

```
(* Semantic checking for the ScalaL compiler *)

open Ast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns unit if
    successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function
      *)

let check (globals, functions) =

  (* Raise an exception if the given list has a
     duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
   n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf
       n1))
     | _ :: t -> helper t
     | [] -> ()
    in helper (List.sort compare list)
  in
```

```
(* Raise an exception if a given binding is to a unit
    type *)
let check_not_unit exceptf = function
   (Unit, n) -> raise (Failure (exceptf n))
  | _ -> ()
in

(* Raise an exception of the given rvalue type cannot
    be assigned to
   the given lvalue type *)
let check_assign lvaluet rvaluet err =
   if lvaluet == rvaluet then lvaluet else raise err
in

(**** Checking Global Variables ****)

List.iter (check_not_unit (fun n -> "illegal unit
    global " ^ n)) globals;

report_duplicate (fun n -> "duplicate global " ^ n)
    (List.map snd globals);

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname)
    functions)
then raise (Failure ("function print may not be
    defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls = StringMap.add "print"
   { typ = Unit; fname = "print"; formals = [(Int,
       "x")];
     locals = []; body = [] } (StringMap.singleton
        "printb"
   { typ = Unit; fname = "printb"; formals = [(Bool,
       "x")];
     locals = []; body = [] })
 in

let function_decls = List.fold_left (fun m fd ->
    StringMap.add fd.fname fd m)
                 built_in_decls functions
in

let function_decl s = try StringMap.find s
    function_decls
    with Not_found -> raise (Failure ("unrecognized
        function " ^ s))
in

let _ = function_decl "main" in (* Ensure "main" is
    defined *)
```

```
let check_function func =

  List.iter (check_not_unit (fun n -> "illegal unit
      formal " ^ n ^
    " in " ^ func.fname)) func.formals;

  report_duplicate (fun n -> "duplicate formal " ^ n ^
      " in " ^ func.fname)
    (List.map snd func.formals);

  List.iter (check_not_unit (fun n -> "illegal unit
      local " ^ n ^
    " in " ^ func.fname)) func.locals;

  report_duplicate (fun n -> "duplicate local " ^ n ^
      " in " ^ func.fname)
    (List.map snd func.locals);

  (* Type of each variable (global, formal, or local *)
  let symbols = List.fold_left (fun m (t, n) ->
      StringMap.add n t m)
 StringMap.empty (globals @ func.formals @ func.locals
     )
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared
        identifier " ^ s))
  in

  (* Return the type of an expression or throw an
      exception *)
  let rec expr = function
   | IntLit _ -> Int
    | FloatLit _ -> Float
    | BoolLit _ -> Bool
    | Id s -> type_of_identifier s
    | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2
       = expr e2 in
 (match op with
      Add | Sub | Mult | Div when t1 = Int && t2 = Int
          -> Int
 | Equal | Neq when t1 = t2 -> Bool
 | Less | Leq | Greater | Geq when t1 = Int && t2 =
     Int -> Bool
 | And | Or when t1 = Bool && t2 = Bool -> Bool
     | _ -> raise (Failure ("illegal binary operator "
         ^
          string_of_typ t1 ^ " " ^ string_of_op op ^ "
              " ^
          string_of_typ t2 ^ " in " ^ string_of_expr e))
     )
    | Unop(op, e) as ex -> let t = expr e in
  (match op with
    Neg when t = Int -> Int
  | Not when t = Bool -> Bool
```

```
      | _ -> raise (Failure ("illegal unary operator "
          ^ string_of_uop op ^
        string_of_typ t ^ " in " ^ string_of_expr ex)))
    | Noexpr -> Unit
    | Assign(var, t, e) as ex -> let lt =
      type_of_identifier var (*XXX*)
                    and rt = expr e in
     check_assign lt rt (Failure ("illegal assignment
        " ^ string_of_typ lt ^
           " = " ^ string_of_typ rt ^ " in " ^
          string_of_expr ex))
    | Call(fname, actuals) as call -> let fd =
      function_decl fname in
     if List.length actuals != List.length fd.formals
         then
       raise (Failure ("expecting " ^ string_of_int
         (List.length fd.formals) ^ " arguments in " ^
            string_of_expr call))
     else
       List.iter2 (fun (ft, _) e -> let et = expr e in
         ignore (check_assign ft et
           (Failure ("illegal actual argument found "
              ^ string_of_typ et ^
           " expected " ^ string_of_typ ft ^ " in " ^
              string_of_expr e))))
         fd.formals actuals;
       fd.typ
  in

  let check_bool_expr e = if expr e != Bool
   then raise (Failure ("expected Boolean expression in
       " ^ string_of_expr e))
   else () in

  (* Verify a statement or throw an exception *)
  let rec stmt = function
 Block sl -> let rec check_block = function
       [Return _ as s] -> stmt s
     | Return _ :: _ -> raise (Failure "nothing may
         follow a return")
     | Block sl :: ss -> check_block (sl @ ss)
     | s :: ss -> stmt s ; check_block ss
     | [] -> ()
    in check_block sl
    | Expr e -> ignore (expr e)
    | Return e -> let t = expr e in if t = func.typ
      then () else
     raise (Failure ("return gives " ^ string_of_typ t
        ^ " expected " ^
                string_of_typ func.typ ^ " in " ^
                    string_of_expr e))

    | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt
      b2
    | For(e1, e2, e3, st) -> ignore (expr e1);
      check_bool_expr e2;
                    ignore (expr e3); stmt st
```

```
    | While(p, s) -> check_bool_expr p; stmt s
  in

  stmt (Block func.body)

 in
 List.iter check_function functions
```

```
open Llvm_target
open Llvm_scalar_opts
open Llvm
open Llvm_executionengine (* FIXME not working *)

module L = Llvm
module A = Ast

module StringMap = Map.Make(String)

let translate (globals, functions) =
 let context = L.global_context () in
 let the_module = L.create_module context "ScalaL"
 and i32_t = L.i32_type context
 and i8_t = L.i8_type context
 and i1_t = L.i1_type context
 and float_t = L.float_type context
 and unit_t = L.void_type context in

 let ltype_of_typ = function
    A.Int -> i32_t
  | A.Bool -> i1_t
  | A.Float -> float_t
  | A.Unit -> unit_t in


 (* Declare each global variable; remember its value in
    a map *)
 let global_vars =
  let global_var m (t, n) =
    let init = L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init
        the_module) m in
  List.fold_left global_var StringMap.empty globals in

 (* Declare printf(), which the print built-in function
    will call *)
 let printf_t = L.var_arg_function_type i32_t [|
    L.pointer_type i8_t |] in
 let printf_func = L.declare_function "printf" printf_t
    the_module in

 (* Define each function (arguments and return type) so
    we can call it *)
 let function_decls =
  let function_decl m fdecl =
    let name = fdecl.A.fname
    and formal_types =
```

```
 Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
     fdecl.A.formals)
   in let ftype = L.function_type (ltype_of_typ
       fdecl.A.typ) formal_types in
   StringMap.add name (L.define_function name ftype
       the_module, fdecl) m in
 List.fold_left function_decl StringMap.empty
     functions in

(* Fill in the body of the given function *)
let build_function_body fdecl =
 let (the_function, _) = StringMap.find fdecl.A.fname
     function_decls in
 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 in

 (* Construct the function's "locals": formal
     arguments and locally
    declared variables. Allocate each on the stack,
        initialize their
    value, if appropriate, and remember their values
        in the "locals" map *)
 let local_vars =
   let add_formal m (t, n) p = L.set_value_name n p;
 let local = L.build_alloca (ltype_of_typ t) n builder
     in
 ignore (L.build_store p local builder);
 StringMap.add n local m in

   let add_local m (t, n) =
 let local_var = L.build_alloca (ltype_of_typ t) n
     builder
 in StringMap.add n local_var m in

   let formals = List.fold_left2 add_formal
       StringMap.empty fdecl.A.formals
      (Array.to_list (L.params the_function)) in
   List.fold_left add_local formals fdecl.A.locals in

 (* Return the value for a variable or formal
     argument *)
 let lookup n = try StringMap.find n local_vars
            with Not_found -> StringMap.find n
                global_vars
 in

 (* Construct code for an expression; return its
     value *)
 let rec expr builder = function
  | A.IntLit i -> L.const_int i32_t i
   | A.FloatLit f -> L.const_float float_t f
   | A.BoolLit b -> L.const_int i1_t (if b then 1 else
       0)
   | A.Noexpr -> L.const_int i32_t 0
```

```
    | A.Id s -> L.build_load (lookup s) s builder
    | A.Binop (e1, op, e2) ->
   let e1' = expr builder e1
   and e2' = expr builder e2 in
   (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.Unop(op, e) ->
   let e' = expr builder e in
   (match op with
     A.Neg   -> L.build_neg
       | A.Not   -> L.build_not) e' "tmp" builder
    | A.Assign (s,t, e) -> let e' = expr builder e in
                ignore (L.build_store e' (lookup s)
                    builder); e'
    | A.Call ("print", [e]) | A.Call ("printb", [e]) ->
   L.build_call printf_func [| int_format_str ; (expr
       builder e) |]
     "printf" builder
    | A.Call (f, act) ->
      let (fdef, fdecl) = StringMap.find f
          function_decls in
   let actuals = List.rev (List.map (expr builder)
      (List.rev act)) in
   let result = (match fdecl.A.typ with A.Unit -> ""
                                      | _ -> f ^ "_result") in
      L.build_call fdef (Array.of_list actuals) result
          builder
   in

   (* Invoke "f builder" if the current block doesn't
      already
     have a terminal (e.g., a branch). *)
   let add_terminal builder f =
    match L.block_terminator (L.insertion_block
        builder) with
  Some _ -> ()
    | None -> ignore (f builder) in

   (* Build the code for the given statement; return
      the builder for
     the statement's successor *)
   let rec stmt builder = function
  A.Block sl -> List.fold_left stmt builder sl
    | A.Expr e -> ignore (expr builder e); builder
    | A.Return e -> ignore (match fdecl.A.typ with
```

```
   A.Unit -> L.build_ret_void builder
  | _ -> L.build_ret (expr builder e) builder); builder
    | A.If (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
  let merge_bb = L.append_block context "merge"
      the_function in

  let then_bb = L.append_block context "then"
      the_function in
  add_terminal (stmt (L.builder_at_end context
      then_bb) then_stmt)
    (L.build_br merge_bb);

  let else_bb = L.append_block context "else"
      the_function in
  add_terminal (stmt (L.builder_at_end context
      else_bb) else_stmt)
    (L.build_br merge_bb);

  ignore (L.build_cond_br bool_val then_bb else_bb
      builder);
  L.builder_at_end context merge_bb

    | A.While (predicate, body) ->
   let pred_bb = L.append_block context "while"
       the_function in
   ignore (L.build_br pred_bb builder);

   let body_bb = L.append_block context "while_body"
       the_function in
   add_terminal (stmt (L.builder_at_end context
       body_bb) body)
     (L.build_br pred_bb);

   let pred_builder = L.builder_at_end context pred_bb
       in
   let bool_val = expr pred_builder predicate in

   let merge_bb = L.append_block context "merge"
       the_function in
   ignore (L.build_cond_br bool_val body_bb merge_bb
       pred_builder);
   L.builder_at_end context merge_bb

    | A.For (e1, e2, e3, body) -> stmt builder
     ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body
       ; A.Expr e3]) ] )
  in

  (* Build the code for each statement in the function
      *)
  let builder = stmt builder (A.Block fdecl.A.body) in

  (* Add a return if the last block falls off the end
      *)
  add_terminal builder (match fdecl.A.typ with
     A.Unit -> L.build_ret_void
```

```
    | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

 List.iter build_function_body functions;


 (* Optimization *)
 let the_fpm = PassManager.create_function the_module in
 add_instruction_combination the_fpm;
 add_reassociation the_fpm;
 add_gvn the_fpm;
 add_cfg_simplification the_fpm;
 ignore(PassManager.initialize the_fpm);
 let _ PassManager.run_function functions the_fpm;

 (*
 FIXME
 *)
 (* dump_module the_module *)
 the_module
```

```
open Core.Std

open Ast

val interpreted_of_ast : Ast.expr_typed ->
    Ast.expr_typed
```

```
open Core.Std
open Ast

(* GADT experimentation *)

type _ typ =
   | GBool : bool -> bool typ
   | GInt : int -> typ
   | GChar : char -> typ
   | GFloat : float -> typ

type _ texpr =
   | GTyp : 'a typ -> 'a typ
   | GIf : bool texpr * 'a texpr * 'a texpr -> 'a texpr
   | GEq : 'a texpr * 'a texpr -> bool texpr
   | GLt : 'a texpr * 'a texpr -> bool texpr
   | GGt : 'a texpr * 'a texpr -> bool texpr
   | GEq : 'a texpr * 'a texpr -> bool texpr
   | GNeq : 'a texpr * 'a texpr -> bool texpr
   | GSub : 'a texpr * 'a texpr -> texpr
   | GMul : 'a texpr * 'a texpr -> texpr
   | GDiv : 'a texpr * 'a texpr -> texpr
   | GAdd : 'a texpr * 'a texpr -> texpr
   | GMod : 'a texpr * 'a texpr -> texpr

let rec inter : type a. a texpr -> a = function
  | GTyp (GBool b) -> b
```

```
  | GTyp (GInt i) -> i
  | GIf (b, l, r) -> if inter b then inter l else inter
     r
  | GEq (a, b) -> (inter a) = (inter b)
  | GLt (a,b) -> a < b
  | GGt (a,b) -> a > b
   | GEq (a, b) -> a = b
   | GSub (a, b) -> a -b
   | GAdd (a, b) -> a +b
   | GMul (a, b) -> a *b
   | GDiv (a, b) -> a /b
   | GMod (a, b) -> a %b
   | _ -> failwith (sprintf "[ERROR]: something wrong "
       (Ast.string_of_op op) (Ast.string_of_type t1))
```

```
(*
 * Description:
 * This is the entry point / top level of Scala-lite
 * handeling user input arguemnt option inlucindg the
    folowing
 * compilation
 * interpreter
 * otuput LLVM-IR
 * otput ast
 *
 * Author; DL, <dl2997@columbia.edu>
 *
 * Last modified: 20160811
 *
 * Dependencies:
 *   llvm3.8.1
 *   OCaml4.0.3.0
 *   Ocaml llvm binding module
 *
 * Usage:
 *      $ cd ../src
 *      $ make 2>&1 | tee make$(date +s%).log
 *      $ ./testall.sh
 *
 * NOTE: this compiler experiment originally was largely
    built based upon
 * microC by SE from Columbia University; hence the
    Licesnc of this compiler would
 * adopt similar Licence of microC, if there is any.
 *)
open Llvm
open Llvm_target
open Llvm_scalar_opts

type action = Ast | LLVM_IR | Scalalc | Scalali

let _ =
 let action = if Array.length Sys.argv > 1 then
   List.assoc Sys.argv.(1) [ ("-a", Ast); (* Print the
      AST only *)
           ("-l", LLVM_IR); (* Generate LLVM, don't
```

```
              check *)
            ("-i", Scalali)
          ("-c", Scalalc) ] (* Generate, check LLVM IR
              *)
else Compile in
let lexbuf = Lexing.from_channel stdin in
let ast = Parser.program Scanner.token lexbuf in
Semant.check ast;
match action with
  Ast -> print_string (Ast.string_of_program ast)
| LLVM_IR -> print_string (Llvm.string_of_llmodule
    (Codegen.translate ast))
| Scalali -> Interpreter.main_loop (Codegen.translate
    ast)
| Scalalc -> let m = Codegen.translate ast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```