Compiler Construction Project

Josef Svenningsson / Alex Gerdes

Spring 2016 – v1.2

Contents

1 Change Log

1.1 v1.2 – April 11

Small corrections

1.2 v1.1 – March 18

Initial version

2 Project summary

This document describes the compiler project that you will do as the main part of the examination for the Compiler construction course. The project is done individually or in groups of two students (recommended). The project is split into three parts:

1. Front end for the language Javalette, i.e. lexical analysis, parsing, building abstract syntax, type-checking and a few static checks. This part builds mainly on knowledge that you should have acquired previously, e.g., in the course Programming Language Technology.
2. A back end that generates code for LLVM (the Low Level Virtual Machine). LLVM are described in more detail later in this document.
3. Extensions to the base language. There are several optional extensions that you may choose to do for the third part of the project, as detailed below.

2.1 Submission deadlines

There are three submission deadlines, one for each part of the project:

- Submission A: At this point you must submit the first part, i.e., a working compiler that can parse and typecheck all programs in the base language and statically receject illegal programs. Deadline: 24 April, 2016.
- Submission B: The second part of the project is a complete compiler that can compile and run all programs in the base language. Deadline: May 15, 2016.
- Submission C: The third and last part consist of at least one extension to the base language, which is required to pass the course. More extensions can be implemented to get a higher grade. More information below. Deadline: 29 May, 2016.

In addition to these three submissions, examination includes a brief oral exam after submission C. The dates are posted on the course home page as well.

2.2 Extensions, credits and grades

The options for an extended project are to extend the source language with e.g. arrays and for loops, structures and pointers, object-oriented features or to generate native x86 code. There is also one essay project you can do which involves studying optimizations in the LLVM framework. You do not need to decide in advance how ambitious you will be; instead you should finish each stage before you attempt an extension.

In submission C, each of the seven tasks described in sections 5.1–5.7 gives one credit if implemented as described.

To pass the course and get grade 3 (or G, if you are a GU student), you need to submit working solutions in all submissions, implement at least on language extension in submission C, and pass the oral exam. To get grade 4, you must earn three credits; grade 5 (VG for GU students) requires five credits.

If you are only looking to pass the course and only get one credit then the project of studying an LLVM optimization, described in section 5.7 is not enough. You must implement at least one language extension to Javalette in order to pass the course.

Part of the goal of a project course like this is that you shall deliver working code on time. Thus, credits will be awarded for working extensions submitted before the deadline. We may allow resubmissions for minor bugfixes, but no partial credits will be awarded for partial solutions.

Finally, we note that we are making major simplifications in a compiler project by using virtual machines like LLVM as targets, rather than a real machine. This means that you can produce simple-minded code and rely on the respective target tools to do optimization and JIT compilation to machine code. The final lectures in the course will discuss these issues, but they will not be covered in depth. On the other hand, for most compiling purposes the road we take is the most effective. This leaves fine-tuning of optimization to LLVM tools, allowing many source languages and front-ends to profit from this effort.

3 The language Javalette

Javalette is a simple imperative language. It is almost a subset of C (see below). It can also be easily translated to Java (see below).

Javalette is not a realistic language for production use. However, it is big enough to allow for a core compiler project that illustrates all phases in compilation. It also forms a basis for extensions in several directions.

The basic language has no heap-allocated data. However, the extensions involve (Java-like) arrays, structures and objects, all of which are allocated on the heap. The extended language is designed to be garbage-collected, but you will not implement garbage collection as part of your project.

The description in this document is intentionally a bit vague and based on examples; it is part of your task to define the language precisely. However, the language is also partly defined by a collection of test programs (see below), on which the behaviour of your compiler is specified.

3.1 Some small Javalette programs

Let's start with a couple of small programs. First, here is how to say hello to the world:

```
// Hello world program

int main () {
  printString("Hello world!") ;
  return 0 ;
}
```

A program that prints the even numbers smaller than 10 is

```
int main () {
  int i = 0 ;
  while (i < 10) {
    if (i % 2 == 0) printInt(i) ;
    i++ ;
  }
  return 0 ;
}
```

Finally, we show the factorial function in both iterative and recursive style:

```
int main () {
  printInt(fact(7)) ;
  printInt(factr(7)) ;
  return 0 ;
```

```
}

// iterative factorial

int fact (int n) {
  int i,r ;
  i = 1 ;
  r = 1 ;
  while (i <= n) {
    r = r * i ;
    i++ ;
  }
  return r ;
}

// recursive factorial

int factr (int n) {
  if (n < 2)
    return 1 ;
  else
    return n * factr(n-1) ;
}
```

3.2 Program structure

A Javalette program is a sequence of *function definitions*.

A function definition has a *return type*, a *name*, a *parameter list*, and a *body* consisting of a *block*.

The names of the functions defined in a program must be different (i.e, there is no overloading).

One function must have the name `main`. Its return type must be `int` and its parameter list empty. Execution of a program consists of executing `main`.

A function whose return type is not `void` *must* return a value of its return type. The compiler must check that it is not possible that execution of the function terminates without passing a `return` statement. This check may be conservative, i.e. reject as incorrect certain functions that actually would always return a value. A typical case could be to reject a function ending with an `if`-statement where only one branch returns, without considering the possibility that the test expression might always evaluate to the same value, avoiding the branch without return. However, your check must correctly decide the control flow when the test expression is the literal `true` or the literal `false`. A function, whose return type is `void`, may, on the other hand, omit the `return` statement completely.

Functions can be *mutually recursive*, i.e. call each other. There is no prescribed order between function definitions (i.e., a call to a function may appear in the program before the function definition).

There are no modules or other separate compilation facilities; we consider only one-file programs.

3.3 Types

Basic Javalette types are `int`, `double`, `boolean` and `void`. Values of types `int`, `double` and `boolean` are denoted by literals (see below). `void` has no values and no literals.

No coercions (casts) are performed between types. Note this: it is NOT considered an improvement to your compiler to add implicit casts. In fact, some of the test programs check that you do not allow casts.

In the type checker, it is useful to have a notion of a *function type*, which is a pair consisting of the value type and the list of parameter types.

3.4 Statements

The following are the forms of statements in Javalette; we indicate syntax using BNFC notation, where we use `Ident`, `Exp` and `Stmt` to indicate a variable, expression and statement, respectively. Terminals are given within quotes. For simplicity, we sometimes deviate here from the actual provided grammar file.

- *Empty statement*: `";"`
- *Variable declarations*: `Type Ident ";"`

  Comment: Several variables may be declared simultaneously, as in `int i, j;` and initial values may be specified, as in `int n = 0;`
- *Assignments*: `Ident "=" Exp ";"`
- *Increments and decrements*: `Ident "++" ";"` and `Ident "--" ";"`

  Comment: Only for variables of type `int`; can be seen as sugar for assignments.
- *Conditionals*: `"if" "(" Exp ")" Stmt "else" Stmt`

  Comment: Can be without the `else` part.
- *While loops* : `"while" "(" Exp ")" Stmt`
- *Returns*: `"return" Exp ";"`

  Comment: No `Exp` for type `void`.

- *Expressions of type* `void`: `Exp ";"`

  Comment: The expression here will be a call to a void function (no other expressions have type `void`).

- *Blocks*: `"{" [Stmt] "}"`

  Comment: A function body is a statement of this form.

Declarations may appear anywhere within a block, but a variable must be declared before it is used.

A variable declared in an outer scope may be redeclared in a block; the new declaration then shadows the previous declaration for the rest of the block.

A variable can only be declared once in a block.

If no initial value is given in a variable declaration, the value of the variable is initialized to `0` for type `int`, `0.0` for type `double` and `false` for type `boolean`. Note that this is different from Java, where local variables must be explicitly initialized.

3.5 Expressions

Expressions in Javalette have the following forms:

- *Literals*: Integer, double, and Boolean literals (see below).
- *Variables.*
- *Binary operators*: `+`, `-`, `*`, `/` and `%`. Types are as expected; all except `%` are overloaded. Precedence and associativity as in C and Java.
- *Relational expressions*: `==`, `!=`, `<`, `<=`, `>` and `>=`. All overloaded as expected.
- *Disjunctions and conjunctions*: `||` and `&&`. These operators have *lazy semantics*, i.e.,
  - In `a && b`, if `a` evaluates to `false`, `b` is not evaluated and the value of the whole expression is `false`.
  - In `a || b`, if `a` evaluates to `true`, `b` is not evaluated and the value of the whole expression is `true`.
- *Unary operators*: `-` and `!` (negation of `int` and `double`, negation of `boolean`).
- Function calls.

3.6 Lexical details

Some of the tokens in Javalette are

- *Integer literals*: sequence of digits, e.g. `123`.
- *Float (double) literals*: digits with a decimal point, e.g. `3.14`, possibly with an exponent (positive or negative), e.g. `1.6e-48`.
- *Boolean literals*: `true` and `false`.
- *String literals*: ASCII characters in double quotes, e.g. `"Hello world"` (escapes as usual: `\n \t \" \\`). Can only be used in calls of primitive function `printString`.
- *Identifiers*: a letter followed by an optional sequence of letters, digits, and underscores.
- *Reserved words*: These include `while`, `if`, `else` and `return`.

Comments in Javalette are enclosed between `/*` and `*/` or extend from `//` to the end of line, or from `#` to the end of line (to treat C preprocessor directives as comments).

3.7 Primitive functions

For input and output, Javalette programs may use the following functions:

```
void printInt (int n)
void printDouble(double x)
void printString(String s)
int readInt()
double readDouble()
```

Note that there is no type of strings in Javalette, so the only argument that can be given to `printString` is a string literal.

The print functions print their arguments terminated by newline and the read functions will only read one number per line. This is obviously rudimentary, but enough for our purposes.

These functions are not directly implemented in the virtual machines we use. We will provide them using other means, as detailed below.

3.8 Parameter passing

All parameters are passed by value, i.e. the value of the actual parameter is computed and copied into the formal parameter before the subroutine is executed. Parameters act as local variables within the subroutine, i.e. they can be assigned to.

3.9 Javalette, C, and Java

Javalette programs can be compiled by a C compiler (`gcc`) if prefixed by suitable preprocessor directives and macro definitions, e.g.

```
#include <stdio.h>
#define printInt(k) printf("%d\n", k)
#define boolean int
#define true 1
```

In addition, function definitions must be reordered so that definition precedes use, mutual recursion must be resolved by extra type signatures and variable declarations moved to the beginnings of blocks.

Javalette programs can be compiled by a Java compiler (`javac`) by wrapping all functions in a class as `public static` methods and adding one more `main` method that calls your `main`:

```
public static void main (String [] args) {
  main();
}
```

Using a C compiler or Java compiler is a good way to understand what a program means even before you have written the full compiler. It can be useful to test the programs produced by your compiler with the result of the C- and/or Java compiler.

4 The front end

Your first task is to implement a compiler front end for Javalette, i.e.

1. Define suitable data types/classes for representing Javalette abstract syntax.
2. Implement a lexer and parser that builds abstract syntax from strings.
3. Implement a type checker that checks that programs are type-correct.
4. Implement a main program that calls lexer, parser and type checker, and reports errors.

These tasks are very well understood; there is a well-developed theory and, for steps 1 and 2, convenient tools exist that do most of the work. You should be familiar with these theories and tools and we expect you to complete the front end during the first week of the course.

We recommend that you use the BNF converter and either Alex and Happy (if you decide to implement your compiler in Haskell) or JLex and Cup (if you use Java). We may also allow other implementation languages and tools, but we can not guarantee support, and you must discuss your choice with Alex before you start. This is to make sure that we will be able to run your compiler and that you will not use inferior tools.

We provide a BNFC source file `Javalette.cf` that you may use. If you already have a BNFC file for a similar language that you want to reuse you may do so, but you must make sure that you modify it to pass the test suite for this course.

We will accept a small number of shift/reduce conflicts in your parser; your documentation must describe these and argue that they are harmless. Reduce/reduce conflicts are not allowed. The provided BNFC file has the standard dangling-else shift/reduce conflict.

One thing to note is that it may be useful to implement the type-checker as a function, which traverses the syntax *and returns its input* if the program is type-correct. The reason for this is that you may actually want to modify this and decorate the syntax trees with more information during type-checking for later use by the code generator. One example of such decoration can be to annotate all subexpressions with type information; this will be useful during code generation.

To do this, you can add one further form of expression to your BNFC source, namely a type-annotated expression.

5 Extensions

This section describes optional extensions that you may implement to learn more, get credits and thus a higher final grade. You may choose different combinations of the extensions.

In this section we specify the requirements on the extensions. Some implementation hints are given in section 8 and in the lecture notes.

5.1 One-dimensional arrays and for loops

The basic Javalette language has no heap-allocated data, so memory management consists only of managing the run-time stack. In this extension you will add one-dimensional arrays to basic Javalette. To get the credit, you must implement this in the front end and in the respective back end.

Arrays are Java-like, i.e. variables of array type contain a reference to the actual array, which is allocated on the heap. Arrays are explicitly created using a `new` construct and variables of array type have an attribute, `length`, which is accessed using dot notation.

Some examples of array declarations in the extension are

```
int[] a ;
double[] b;
```

Creating an array may or may not be combined with the declaration:

```
a = new int[20];
int[] c = new int[30];
```

After the above code, `a.length` evaluates to 20 and `a` refers to an array of 20 integer values, indexed from 0 to 19 (indexing always starts at 0).

Functions may have arrays as arguments and return arrays as results:

```
int[] sum (int[] a, int[] b) {
  int[] res = new int [a.length];
  int i = 0;
  while (i < a.length) {
    res[i] = a[i] + b[i];
    i++;
  }
  return res;
}
```

Array parameters are passed by value, i.e. the reference is copied into the parameter.

One new form of expressions is added, namely indexing, as shown in the example. Indexed expressions may also occur as L-values, i.e. as left hand sides of assignment statements. An array can be filled with values by assigning each individual element, as in function `sum`. But one can also assign references as in C or Java:

```
c = a;
```

The extension also includes implementation of a simple form of `foreach`-loop to iterate over arrays. If *expr* is an expression of type *t*`[]`, the following is a new form of statement:

`for` (*t var* : *expr*) *stmt*

The variable *var* of type *t* assumes the values *expr*[0], *expr*[1] and so on and the *stmt* is executed for each value. The scope of *var* is just *stmt*.

This form of loop is very convenient when you want to iterate over an array and access the elements, but it is not useful when you need to assign values to the elements. For this, we still have to rely on the `while` loop. The traditional `for` loop would be attractive here, but we cannot implement everything.

Test files for this extension are in subdirectory `extensions/arrays1`.

5.2 Multidimensional arrays

In this extension you add arrays with an arbitrary number of indices. Just as in Java, an array of type `int[][]` is a one-dimensional array, each of whose elements

is a one-dimensional array of integers. Declaration, creation and indexing is as expected:

```
int[][] matrix = new int[10][20];
int[][][] pixels;
...
matrix[i][j] =  2 * matrix[i][j];
```

You must specify the number of elements in each dimension when creating an array. For a two-dimensional rectangular array such as `matrix`, the number of elements in the two dimensions are `matrix.length` and `matrix[0].length`, respectively.

5.3 Dynamic data structures

In this extension you will implement a simple form of dynamic data structures, which is enough to implement lists and trees. The source language extensions are the following:

- Two new forms of top-level definitions are added (in the basic language there are only function definitions):
  1. *Structure definitions*, as examplified by

     ```
     struct Node {
        int elem;
        list next;
     };
     ```

  2. *Pointer type definitions*, as examplified by

     ```
     typedef struct Node *list;
     ```

     Note that this second form is intended to be very restricted. We can only use it to introduce new types that represent pointers to structures. Thus this form of definition is completely fixed except for the names of the structure and the new type. Note also that, following the spirit of Javalette, the order of definitions is arbitrary.

- Three new forms of expression are introduced:
  1. *Heap object creation*, examplified by `new Node`, where `new` is a new reserved word. A new block of heap memory is allocated and the expression returns a pointer to that memory. The type of this expression is thus the type of pointers to `Node`, i.e. `list`.
  2. *Pointer dereferencing*, examplified by `xs->next`. This returns the content of the field `next` of the heap node pointed to by `xs`.

3. *Null pointers*, examplified by `(list)null`. Note that the pointer type must be explicitly mentioned here, using syntax similar to casts (remember that there are no casts in Javalette).

- Finally, pointer dereferencing may also be used as L-values and thus occur to the left of an assignment statement, as in

  ```
  xs->elem = 3;
  ```

Here is an example of a complete program in the extended language:

```
typedef struct Node *list;

struct Node {
  int elem;
  list next;
};


int main() {
  printInt(length(fromTo(1,100)));
  return 0;
}

list cons (int x, list xs) {
  list n;
  n = new Node;
  n->elem = x;
  n->next = xs;
  return n;
}

list fromTo (int m, int n) {
  if (m>n)
    return (list)null;
  else
    return cons (m,fromTo (m+1,n));
}

int length (list xs) {
  int res = 0;
  while (xs != (list)null) {
    res++;
    xs = xs->next;
  }
  return res;
}
```

This and a few other test programs can be found in the `extensions/pointers` subdirectory of the test suite.

5.4 Object-orientation

This extension adds classes and objects to basic Javalette. From a language design point of view, it is not clear that you would want both this and the previous extension in the same language, but here we disregard this.

Here is a first simple program in the proposed extension:

```
class Counter {
  int val;

  void incr () {val++; return;}
  int value () {return val;}

}

int main () {
  Counter c;
  c = new Counter;
  c.incr();
  c.incr();
  c.incr();
  int x = c.value();
  printInt(x);
  return 0;
}
```

We define a class `Counter`, and in `main` create an object and call its methods a couple of times. The program writes 3 to `stdout`.

The source language extensions, from basic Javalette, are

- A new form of top-level definition: a *class declaration*. A class has a number of instance variables and a number of methods.

  Instance variables are private, i.e. are *only* visible within the methods of the class. We could not have written `c.val` in `main`.

  All methods are public; there is no way to define private methods. It would not be difficult in principle to allow this, but we must limit the task.

  There is always only one implicit constructor method in a class, with no arguments. Instance variables are, as all variables in Javalette, initialized to default values: numbers to 0, booleans to false and object references to null.

We support a simple form of single inheritance: a class may extend another one:

```
class Point2 {
  int x;
  int y;

  void move (int dx, int dy) {
     x = x + dx;
     y = y + dy;
  }

  int getX () { return x; }

  int getY () { return y; }
}

class Point3 extends Point2 {
  int z;

  void moveZ (int dz) {
    z = z + dz;
  }

  int getZ () { return z; }

}

int main () {
  Point2 p;

  Point3 q = new Point3;

  q.move(2,4);
  q.moveZ(7);
  p = q;

  p.move(3,5);

  printInt(p.getX());
  printInt(p.getY());
  printInt(q.getZ());

  return 0;
}
```

Here `Point3` is a subclass of `Point2`. The program above prints 5, 9 and

7.

Classes are types; we can declare variables to be (references to) objects of a certain class. Note that we have subtyping: we can do the assignment `p = q;` . The reverse assignment, `q = p;` would be a type error. We have a strong restriction, though: we will *not* allow overriding of methods. Thus there is no need for dynamic dispatch; all method calls can be statically determined.

- There are four new forms of expression:
  1. `"new" Ident` creates a new object, with fields initialized as described above.
  2. `Expr "." Expr`, is a method call; the first expression must evaluate to an object reference and the second to a call of a method of that object.
  3. `"(" Ident ") null"` is the null reference of the indicated class/type.
  4. `"self"` is, within the methods of a class, a reference to the current object. All calls to other, sibling methods of the class must be indicated as such using `self`, as in `self.isEmpty()` from one of the test files. This requirement is natural, since the extended Javalette, in contrast to Java, has free functions that are not methods of any class.

5.5 Object orientation with dynamic dispatch

The restriction not to allow method override is of course severe. In this extension the restriction is removed and subclassing with inheritance and method override implemented. This requires a major change of implementation as compared to the previous extension. It is no longer possible to decide statically which code to run when a message is sent to an object. Thus each object at runtime must have a link to a class descriptor, a struct with pointers to the code of the methods of the class. These class descriptor are linked together in a list, where a class descriptor has a link to the descriptor of its superclass. This list is searched at runtime for the proper method to execute. All this is discussed more during the lectures.

5.6 Native x86 code generation.

This extension is to produce native assembler code for a real machine, preferrably x86. We may accept code generators for other architectures, but *you* need to think of how we can test your extension. Before you attempt to write a backend for another architecture, discuss your choice with Alex and explain the testing procedure.

5.7 Study of LLVM optimization

We offer one possibility to get a credit that does not involve implementing a Javalette extension. This is to do a more thorough study of the LLVM framework and write a report of 4-5 pages. More precisely the task is as follows:

Look at the list of available optimization passes and choose at least three of these for further study. Mail Alex to agree that your choice is suitable (do this *before* you start to work on the extension!).

For each pass you must:

- Describe the optimization briefly; what kind of analysis is involved, how is code transformed?
- Find a Javalette program that is suitable to illustrate the optimization. List the program, the LLVM code generated by your compiler and the LLVM code that results by using `opt` to apply this pass (and only this pass). In addition to the code listing, explain how the general description in the previous item will actually give the indicated result. Part of the task is to find a program where the pass has an interesting effect.

We emphasize again that if you are only looking to pass the course and only get one credit then this project is not enough. You have to implement at least one extension to Javalette in order to pass the course.

5.8 Further possibilities

We are willing to give credits also to other extensions, which are not as well defined. If you want to do one of these and get credit, you must discuss it with Alex in advance. Here are some possibilities:

- Allow functions to be statically nested.
- Implement higher order functions, using either closures or defunctionalization.
- A simple module system. Details on module systems will be provided in the lectures.
- Implement exceptions, which can be thrown and caught.
- Implement some form of garbage collection.
- Provide a predefined type of lists with list comprehensions, similar to what is available in Python.
- Implement a backend for another architecture, such as ARM. It is important that you provide some way for the grader to test programs.

6 Testing the project

Needless to say, you should test your project extensively. We provide a testsuite of programs and will run your compiler on these. You may download the testsuite from the course web site. The testsuite contains both correct programs (in subdirectory `testsuite/good`) and illegal programs (in subdirectory `testsuite/bad`). For the good programs the correct output is provided in files with suffix `.output`. The bad programs contain examples of both lexical, syntactical and type errors.

Already after having produced the parser you should therefore write a main program and try to parse all the test programs. The same holds for the type checker and so on. When you only have the parser, you will of course pass some bad programs; those that are syntactically correct but have type errors.

Summarizing, your compiler must

- accept and be able to compile all of the files `testsuite/good/*.jl`. For these files, the compiler must print a line containing only `OK` to standard error, optionally followed by arbitrary output, such as a syntax tree or other messages. The compiler must then exit with the exit code 0.
- reject all of the files in `testsuite/bad/*.jl`. For these files, the compiler must print `ERROR` as the first line to standard error and then give an informative error message. The compiler must then exit with an exit code other than 0.

Furthermore, for correct programs, your compiled programs, must run and give correct output.

6.1 Automated testing

We also provide a program that automatically compiles and runs all the test programs. Before submission you **must** run that program to verify that your compiler behaves correctly. Our first action when we receive your submission is to run these tests. If this run fails, we will reject your submission without further checks, so you must make sure that this step works. See appendix B for details. Unfortunately, we cannot supply a working test driver for the Windows platform. If you have a Windows machine, you may do most of the development, including manual testing, on that machine, but for final testing you should transfer your project to our lab machines and run the test driver.

The test driver runs each good program and compares its output with the corresponding `.output` file. If the program needs input, this is taken from the `.input` file. Note that the test driver handles this; your generated code should read from `stdin` and write to `stdout`.

The tests are of course not exhaustive. It is quite possible that the grader will discover bugs in your code even if it passes all tests.

7 Code generation: LLVM

In the second part of the course you will change your target to LLVM, redo the back end and optionally extend the Javalette language.

LLVM (Low Level Virtual Machine) is both an intermediate representation language and a compiler infrastructure, i.e. a collection of software components for manipulating (e.g. optimizing) LLVM code and backends for various architectures. LLVM has a large user base and is actively developed. A lot of information and code to download can be found at the LLVM web site `http://www.llvm.org`.

Also LLVM code comes in two formats, a human-readable assembler format (stored in `.ll` files) and a binary bitcode format (stored in `.bc` files). Your compiler will produce the assembler format and you will use the LLVM assembler `llvm-as` to produce binary files for execution.

In addition to the assembler, the LLVM infrastructure consists of a large number of tools for optimizing, linking, JIT-compiling and manipulating bitcode. One consequence is that a compiler writer may produce very simple-minded LLVM code and leave to the LLVM tools to improve code when needed. Of course, similar remarks apply to JVM code.

7.1 LLVM code

The LLVM virtual machine is a *register machine*, with an infinite supply of typed, virtual registers. The LLVM intermediate language is a version of *three-address code* with arithmetic instructions that take operands from two registers and place the result in a third register. LLVM code must be in SSA (static single assignment) form, i.e. each virtual register may only be assigned once in the program text.

The LLVM language is typed, and all instructions contain type information. This "high-level" information, together with the "low-level" nature of the virtual machine, gives LLVM a distinctive flavour.

The LLVM web site provides a wealth of information, including language references, tutorials, tool manuals etc. There will also be lectures focusing on code generation for LLVM.

7.2 The structure of a LLVM file

There is less overhead in the LLVM file. But, since the language is typed, we must inform the tools of the types of the primitive functions:

```
declare void @printInt(i32)
declare void @printDouble(double)
```

```
declare void @printString(i8*)
declare i32 @readInt()
declare double @readDouble()
```

Here `i32` is the type of 32 bit integers and `i8*` is the type of a pointer to an 8 bit integer (i.e., to a character). Note that function names in LLVM always start with `@`.

Before running a compiled Javalette program, `myfile.bc` must be linked with `runtime.bc`, a file implementing the primitive functions, which we will provide. In fact, this file is produced by giving `clang` a simple C file with definitions such as

```
void printInt(int x) {
  printf("%d\n",x);
}
```

7.3 An example

We give the same example as we did for JVM, but now in the LLVM version, `myfile.ll` :

```
define i32 @main() {
entry:  %t0 = call i32 @fact(i32 7)             ; function call
        call void @printInt(i32 %t0)
        ret  i32 0

}

define i32 @fact(i32 %__p__n) {
entry:  %n = alloca i32                         ; allocate a variable on stack
        store i32 %__p__n , i32* %n             ; store parameter
        %i = alloca i32
        %r = alloca i32
        store i32 1 , i32* %i                   ; store initial values
        store i32 1 , i32* %r
        br label %lab0                          ; branch to lab0

lab0:   %t0 = load i32* %i                      ; load i
        %t1 = load i32* %n                      ; and n
        %t2 = icmp sle i32 %t0 , %t1            ; boolean %t2 will hold i <= n
        br i1 %t2 , label %lab1 , label %lab2   ; branch depending on %t2

lab1:   %t3 = load i32* %r
        %t4 = load i32* %i
        %t5 = mul i32 %t3 , %t4                 ; compute i * r
        store i32 %t5 , i32* %r                 ; store product
```

```
        %t6 = load i32* %i                      ; fetch i,
        %t7 = add i32 %t6 , 1                   ; add 1
        store i32 %t7 , i32* %i                 ; and store
        br label %lab0

lab2:   %t8 = load i32* %r
        ret  i32 %t8

}
```

We note several things:

- Registers and local variables have names starting with `%`.
- The syntax for function calls uses conventional parameter lists (with type info for each parameter).
- Booleans have type `i1`, one bit integers.
- After initialization, we branch explicitly to `lab0`, rather than just falling through.

7.4 LLVM tools

Your compiler will generate a text file with LLVM code, which is conventionally stored in files with suffix `.ll`. There are then several tools you might use:

- The *assembler* `llvm-as`, which translates the file to an equivalent binary format, called the *bitcode* format, stored in files with suffix `.bc` This is just a more efficient form for further processing. There is a *disassembler* `llvm-dis` that translates in the opposite direction.
- The *linker* `llvm-link`, which can be used to link together e.g. `main.bc` with bitcode file `runtime.bc` that defines the function `@printInt` and the other IO functions. By default, two files are written, `a.out` and `a.out.bc`. As one can guess from the suffix, `a.out.bc` is a bitcode file which contains the definitions from all the input bitcode files.
- The *interpreter/JIT compiler* `lli`, which directly executes its bitcode file argument, using a Just-In-Time compiler.
- The *static compiler* `llc`, which translates the file to a native assembler file for any of the supported architectures. It can also produce native object files using the flag `-filetype=obj`
- The *analyzer/optimizer* `opt`, which can perform a wide range of code optimizations of bitcode.

Here are the steps you can use to produce an executable file from within your compiler:

- Your compiler produces an llvm file, let's call it `prog.ll`.
- Convert the file to bitcode format using `llvm-as`. For our example file, issue the command `llvm-as prog.ll`. This produces the file `prog.bc`.
- Link the bitcode file with the runtime file using `llvm-link`. This step requires that you give the name of the output file using the `-o` flag. For example we can name the output file `main.bc` like so: `llvm-link prog.bc runtime.bc -o main.bc`.
- Generate a native object file using `llc`. By default `llc` will produce assembler output, but by using the flag `-filetype=obj` it will produce an object file. The invocation will look like this: `llc -filetype=obj main.bc`
- Produce an executable. The LLVM toolchain does not have support for this. The easiest way to produce an executable from an object file is to invoke a C compiler, like so: `gcc main.o`. This will produce the executable file `a.out`. If you want to change the name of the output, use the flag `-o`.

  Under the hood `gcc` calls the native linker `ld` but we do not recommend that you do that, because it requires specifying extra libraries and possibly adding paths. GCC solves all that for us.

7.5 Optimizations

To wet your appetite, let us see how the LLVM code can be optimized:

```
proj> cat myfile.ll | llvm-as | opt -std-compile-opts | llvm-dis
declare void @printInt(i32)

define i32 @main() {
entry:
  tail call void @printInt(i32 5040)
  ret i32 0
}

define i32 @fact(i32 %__p__n) nounwind readnone {
entry:
  %t23 = icmp slt i32 %__p__n, 1
  br i1 %t23, label %lab2, label %lab1

lab1:
  %indvar = phi i32 [ 0, %entry ], [ %i.01, %lab1 ]
  %r.02 = phi i32 [ 1, %entry ], [ %t5, %lab1 ]
  %i.01 = add i32 %indvar, 1
  %t5 = mul i32 %r.02, %i.01
  %t7 = add i32 %indvar, 2
```

```
  %t2 = icmp sgt i32 %t7, %__p__n
  br i1 %t2, label %lab2, label %lab1

lab2:
  %r.0.lcssa = phi i32 [ 1, %entry ], [ %t5, %lab1 ]
  ret i32 %r.0.lcssa
}
```

The first line above is the Unix command to do the optimization. We `cat` the LLVM assembly code file and pipe it through the assembler, the optimizer and the disassembler. The result is an optimized file, where we observe:

- In `main`, the call `fact(7)` has been completely computed to the result 5040. The function `fact` is not necessary anymore, but remains, since we have not declared that `fact` is local to this file (one can do that).
- The definition of `fact` has been considerably optimized. In particular, there is no more any use of memory; the whole computation takes place in registers.
- We will explain the `phi` instruction in the lectures; the effect of the first instruction is that the value of `%indvar` will be 0 if control comes to `%lab1` from the block labelled `%entry` (i.e. the first time) and the value will be the value of `%i.01` if control comes from the block labelled `%lab1` (i.e. all other times). The `phi` instruction makes it possible to enforce the SSA form; there is only one assignment in the text to `%indvar`.

If we save the optimized code in `myfileOpt.bc` (without disassembling it), we can link it together with the runtime using `llvm-link myfileOpt.bc runtime.bc -o a.out.bc`. If we disassemble the resulting file `a.out.bc`, we get (we have edited the file slightly in inessential ways):

```
@fstr = internal constant [4 x i8] c"%d\0A\00"

define i32 @main() nounwind {
entry:
        %t0 = getelementptr [4 x i8]* @fstr, i32 0, i32 0
        %t1 = call i32 (i8*, ...)* @printf(i8* %t0, i32 5040) nounwind
        ret i32 0
}

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

What remains is a definition of the format string `@fstr` as a global constant (`\0A` is `\n`), the `getelementpointer` instruction that returns a pointer to the beginning of the format string and a call to `printf` with the result value. Note that the call to `printInt` has been inlined, i.e. replaced by a call to `printf`; so linking includes optimizations across files.

We can now run `a.out.bc` using the just-in-time compiler `lli`. Or, if we prefer, we can produce native assembler code with `llc`. On my x86 machine, this gives

```
        .text
        .align  4,0x90
        .globl  _main
_main:
        subl    $12, %esp
        movl    $5040, 4(%esp)
        movl    $_fstr, (%esp)
        call    _printf
        xorl    %eax, %eax
        addl    $12, %esp
        ret
        .cstring
_fstr:                          ## fstr
        .asciz  "%d\n"
```

8 Hints for the extensions

8.1 One-dimensional arrays

To implement this extension, the expression `new int[e]` will need to allocate memory on the heap for the array itself and for the length attribute. Further, the array elements must be accessed by indexing.

LLVM provides support for built-in arrays, but these are not automatically heap-allocated. Instead, explicit pointers must be used. Thus, an array will have the LLVM type `{i32, [0 x` t `] }*`, where t is the LLVM type of the elements. The first `i32` component holds the length; the second the array elements themselves. The number of elements in the array is here indicated to be 0; it is thus your responsibility to make sure to allocate enough memory. For memory allocation you should use the C function `calloc`, which initializes allocated memory to 0. You must add a type declaration for `calloc`, but you do not need to worry about it at link time; LLVM:s linker includes `stdlib`.

Indexing uses the `getelementptr` instruction, which is discussed in detail in the lectures.

The LLVM does not include a runtime system with garbage collection. Thus, this extension should really include some means for reclaiming heap memory that is no longer needed. The simplest would be to add a statement form `free(a)`, where `a` is an array variable. This would be straightforward to implement, but is *not* necessary to get the credit.

More challenging would be to add automatic garbage collection. LLVM offers

some support for this. If you are interested in doing this, we are willing to give further credits for that task.

8.2 Multidimensional arrays

This extension involves more work than the previous one. In particular, you must understand the `getelementpointer` instruction fully and you must generate code to iteratively allocate heap memory for subarrays.

8.3 Structures and object-orientation.

Techniques to do these extensions are discussed in the lectures.

From an implementation point of view, we recommend that you start with the extension with pointers and structures. You can then reuse much of the machinery developed to implement also the first OO extension. In fact, one attractive way to implement the object extension is by doing a source language translation to Javalette with pointers and structures.

The full OO extension requires more sophisticated techniques, to properly deal with dynamic dispatch.

8.4 Native code generation

The starting point for this extension could be your LLVM code, but you could also start directly from the abstract syntax. Of course, within the scope of this course you will not be able to produce a code generator that can compete with `llc`, but it may anyhow be rewarding to do also this final piece of the compiler yourself.

One major addition here is to handle function calls properly. Unlike JVM and LLVM, which both provide some support for function calls, you will now have to handle all the machinery with activation records, calling conventions, and jumping to the proper code before and after the call.

There are several assemblers for x86 available and even different syntax versions. We recommend that you use the NASM assembler and that you read Paul Carter's PC assembly tutorial (linked from course web site) before you start the project, unless you are already familiar with x86 architecture. We do not have strong requirements on code quality for your code generator; straightforward code generation is acceptable. In particular, you do not need to implement register allocation to improve your code. This will also have serious negative consequences for the performance of your code. Indeed, a preferrable way to get native code is to use a framework like LLVM, which provides an extensive infrastructure for code manipulation.

An introduction to x86 assembler will be given in the lectures.

9 Collaboration and academic honesty

As mentioned before, you work individually or in groups of two in this project. You must develop your own code, and you are *not* allowed to share your code with other students or to get, or even look at, code developed by them. On the other hand, we encourage discussions among participants in the course about the project. As long as you follow the simple and absolute rule not to share code, we have no objections to questions asked and answered at a conceptual level.

If you do get significant help from some other participant, it is natural to acknowledge this in your documentation file.

Don't be a cheater.

A Submission format

1. You prepare your submission by creating a new empty directory, subsequently called the root directory. In this directory you create three subdirectories, `doc`, `lib` and `src`.

2. The root directory must, after building as below, contain the executable compiler `jlc`. The compiler is used as follows:

   - For submission B and C, the command `jlc myFile.jl` produces the executable file `a.out`. Your code will generate `myFile.ll`; then your compiler will call `llvm-as` and `llvm-link` to assemble and link.
   - Optionally, if you have chosen to implement a native code generator, the executable is `jlc_`*ARCH*, where *ARCH* is a CPU architecture such as x86. The command `jlc_`*ARCH* `myFile.ll` should produce an assembly file `myFile.s` and an executable file `a.out`.

   The compiler may be a shell script that calls files in `src`, or a symbolic link.

3. The subdirectory `src` must contain all necessary source code. In this directory, one should be able to build your compiler using `gmake`. Thus the directory contains

   - The BNF Converter source file, if the tool has been used.
   - Alex/Lex/JLex and Happy/Yacc/Cup source files.
   - Program modules: abstract syntax, lexer, parser, type checker, code generator, top level program.
   - A `Makefile` for building the compiler from source. The `Makefile` should at least have these targets:
     - A default target (the one that is run when the command `gmake` is issued). This target should compile all source files in the compiler, and any runtime library files. It does not need to regenerate any source files generated by BNFC, or by any of the parser and lexer generators. After running `gmake` in the source directory, the compiler in the root directory should work without further actions from the user.
     - A `clean` target that removes all files produced during building.

   Note that `src` should *not* contain obsolete files, such as backup files, bytecode files or other unnecessary files.

4. In the `lib` directory you place the following files, as needed:

   - For submission B, you place here `runtime.bc`, an LLVM bitcode file with the same functions. This file will be provided by us.

- If you choose to do a native code generator for submission C, you place here also the corresponding file `runtime.o`, produced by writing and compiling a C file.
- The `doc` directory must contain one file in html, ps, plain ascii, or pdf format (proprietary formats not allowed), with the following content:
  - An explanation of how the compiler is used (what options, what output, etc)
  - A specification of the Javalette language (if produced by BNF converter, you may just refer to your BNFC source file).
  - A list of shift/reduce conficts in your parser, if you have such conflicts, and an analysis of them.
  - For submission C, an explicit list of extensions implemented.
  - If applicable, a list of features *not* implemented and the reason why.

5. If your compiler `jlc` is a shell script, you may also place this file here before building the tar ball.
6. When you have prepared everything, you create a compressed tar ball:

   ```
   tar -zcf partA-1.tar.gz doc lib src
   ```

   This produces the file `partA-1.tar.gz` that you upload to Fire. We suggest the naming scheme `partX-Y.tar.gz` where X is A or B and Y is your version number (Y=1 the first time you submit, and if your submission is rejected and you must resubmit, the next has Y=2 etc).

   If you prefer, you may compress with `bzip2` instead of `gzip`.

B The tester

The tester is provided as a gzipped tar ball, which can be downloaded from the course web site.

This directory contains a test driver (`Grade.hs`, `RunCommand.hs` and `KompTest.hs`) that can be used to run the tests for your project. The subdirectory `testsuite` contains Javalette test programs.

B.1 Installation

The tester requires a Linux (or Mac OS X, or other Unix) environment and that the Haskell compiler GHC is available. You will have to do

```
make
```

in this directory to compile the test program. The result is the executable program `Grade` in the same directory. To use the tester after installation does not require any Haskell knowledge.

If you work on your own Windows machine, we cannot assist you in making the tester work. You should anyhow download the tester to get access to the testsuite in directory `testsuite`. Before submitting, you must upload your project to the lab machines and verify the submission.

B.2 Running the tests

Assume that your submission directory is `dir` and that your compiler is called `jlc`. Assume also that `dir/lib` contains the runtime support file (`runtime.bc` for submission B, and possibly `runtime.o` for submission C).

The test driver takes a number of options and two directories as command line arguments. The possible options are

`-s` <*name*>	The name of your compiler is *name* (default is `jlc`).
`-b JVM`	Target files are JVM `.class` file.
`-b LLVM`	Target files are LLVM `.bc` files.
`-b x86`	Target files are x86 `.o` files.
`-x` <*extension*>	Implemented extensions.

The first of the two path arguments specifies where to find the directory `testsuite` which contains the testsuite. The second specifies your submission directory. Thus, if your compiler is called `jlc` (as it should!), you may place yourself in the tester directory and run

```
./Grade . dir
```

to compile all the basic Javalette programs. The tester will *not* attempt to run the good programs, so this is suitable for testing your compiler for submission A. Note that it is essential that your compiler writes one line to `stderr`, containing `OK` for correct programs and `ERROR` for incorrect programs.

To also *run* the good programs, you must specify the backend as indicated above, i.e. for submission B

```
./Grade -b LLVM . dir
```

The test driver will report its activities in compiling the test programs and running the good ones. If your compiler is correct, output will end as follows:

```
Summary:
 0 compiling basic javalette (48/48)
 0 running basic (LLVM) (22/22)

Credits total: 0
```

All 48 test programs were compiled and gave correct indication OK or ERROR to stderr. The 22 correct programs were run and gave correct output. Since no extensions were tested, no credits were earned.

You may specify several extensions, as follows:

`arrays1`	One-dimensional arrays and for loop.
`arrays2`	Multi-dimensional arrays.
`pointers`	Structures and pointers.
`objects1`	Classes and objects, without method override.
`objects2`	Method override, dynamic dispatch.

B.3 Testing your submission

Your submission must be structured as specified in Appendix A. We suggest that, after having prepared your tar ball, you place it in an empty directory `dir1` and run

```
./Grade -b LLVM . dir1
```

from the test driver directory. The grading program, when it finds a tar ball in the submission directory, starts by extracting and building your compiler, before running the test suite. This is how we test your submission, so you can check that building succeeds before you submit.