Project Overview

The project has three parts: a frontend for the Javalette language (Part A), a backend with code generation for LLVM (Part B), and exter[illegible]r (Part C).

The project is to be done in[illegible]s of two. A group's final grade is based on the number of extensions completed for P[illegible]based on [a system of credits per implemented extension].

Pages

- [Submission format]
- [The Javalette Language]
  - [Example programs]
  - [Program structure]
  - [Types]
  - [Statements]
  - [Expressions]
  - [Lexical details]
  - [Primitive functions]
  - [Parameter passing]
  - [Javalette, C, and Java]
- [Frontend]
- [Code generation (LLVM)]
- [Extensions]
  - [Arrays I]
  - [Arrays II]
  - [Pointers]
  - [OOP I]
  - [OOP II]
  - [Higher-order functions]
  - [x86 generation]
  - [Optimization study]
  - [Other extensions]
- [Extension hints]
- [Testing]

Collaboration and academic honesty

We take academic honesty seriously. As mentioned above, students work individually or in groups of two on this project. Each individual/group must develop their own code, and are *not allowed to share code with other students or to get, or even look at, code developed by others*. Having said that, we do encourage discussions among participants in the course about the project at a conceptual level. Students who get significant help from others must make a note of this and acknowledge (in their documentation file) those who helped.

Don't be a cheater.

Code generation: LLVM

Once the front end is complete, your next task is to implement code generation for LLVM, i.e. your task is to make your compiler genera[illegible] given Javalette source code.

LLVM

LLVM (Low Level Virtual Ma[illegible]rmediate representation language and a compiler infrastructure, i.e. a collecti[illegible]onents for manipulating (e.g. optimizing) LLVM code and backends for various archit[illegible]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`. Make sure your LLVM version is compatible with the one used in the testing Docker has only guaranteed support for this particular version.

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.

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.

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);
}
```

An example

The following LLVM code demonstrates some of the language features in LLVM. It also serves as an example of what kind of code a Javalette compiler could generate for the `fact` function described here.

```
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, i32* %i                  ; load i
        %t1 = load i32, 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, i32* %r
        %t4 = load i32, i32* %i
        %t5 = mul i32 %t3 , %t4                  ; compute i * r
        store i32 %t5 , i32* %r                  ; store product
        %t6 = load i32, 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, i32* %r
        ret  i32 %t8
```

```
}
```


We note several things:

- Registers and local variables have names starting with `%`.
- The syntax for functio… onal parameter lists (with type info for each parameter).
- Booleans have type `i`…
- After initialization, we … `ab0`, rather than just falling through.

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 the 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 (JIT) 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.
- The wrapper `clang` which uses various of the above tools together to provide a similar interface to `GCC`.

Note that some installations of LLVM require a version number after the tool name, for example `llvm-as-3.8` instead of `llvm-as`.

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`
- Finally, produce an executable. The simplest way to do this is with `clang main.o`

A simpler alternative to the above steps is to let `clang` run the various LLVM tools, with `clang prog.ll runtime.bc`

Also note that the testing framework will call LLVM itself, and will link in the runtime library as well. For the purposes of assignment submission, your compiler need only produce an LLVM file (the equivalent of `prog.ll` above).

Optimizations

To whet your appetite, let u[illegible]code can be optimized:

```
> cat myfile.ll | llv[illegible]ompile-opts | llvm-dis
```

```
declare void @printIn[illegible]

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 could 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[illegible] a.out.bc
```

If we disassemble the resul[illegible]e 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 assembly code with `llc`. On a 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"
```

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.

This page specifies the requirements on the extensions. Some implementation hints are given on a separate page for [extension hints](#) and in the lecture notes.

Credits for extensions: ea[illegible]xtensions gives one credit point. Extensions that are non-standard in this sense are t[illegible]*neration* and some projects within the *further possibilities* section. The *native x86 code*[illegible] in that it gives two credits in itself and an extra credit for each of the standard extens[illegible] to the x86 code generator. Example: a student can collect 5 credits as follows.

- one-dimensional array[illegible]erator (1 credit)
- multi-dimensional arrays for LLVM code generator (1 credit)
- native x86 code generation (2 credits)
- one-dimensional arrays for x86 code generator (1 credit)

The course homepage explains how credits translate into course grades.

One-dimensional arrays and for loops (arrays1)

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: 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. The semantics follows Java arrays.

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). It is not required to generate bounds-checking code.

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;
}
```

One new form of expressio [illegible] ndexing, 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;
```

Arrays can be passed as parameters to functions, and returned from functions. When passed or returned, or assigned to a variable as above, it is a reference that is copied, not the contents of the array. The following function returns 3.

```
int return3 (void) {
  int[] arr = new int [1];
  int[] arr2 = new int [2];

  arr2 = arr;
  arr2[0] = 3;
  return arr[0];
}
```

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.

The length of an array is of type int. The `new` syntax for creating a new array is an expression. It can take any (integer type) expression as the new length, and it can be used in other locations than initialisers.

The array type does not support any other operations. There is no need for an equality or less-than test for the array type, for instance.

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

Multi-dimensional arrays (arrays2)

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 arr ... ements is a one-dimensional array of integers. Declaration, creation and indexing is as

```
int[][] matrix = new
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.

Dynamic data structures (pointers)

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 cor e extended language:

```
typedef struct Node *

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.

Object-orientation (objects1)

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 and 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 () { r

  int getY () { r
}

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 [illegible]ass, a reference to the current object. All calls to other, sibling methods c[illegible]ndicated as such using `self`, as in `self.isEmpty()` from one of the test fil[illegible]is natural, since the extended Javalette, in contrast to Java, has free functions[illegible]ds of any class.

Object orienta[illegible]ynamic dispatch (objects2)

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.

Higher-order functions (functions)

This extension adds non-polymorphic function values to Javalette. Functions become first class, i.e., functions can take functions as arguments and return functions as results. Javalette remains call-by-value.

```
int apply(fn(int) -> int f, int x) {
  return f(x);
}

fn(int) -> int compose(fn(int) -> int f, fn(int) -> int g) {
  return \(int x) -> int: f(g(x));
}

int main() {
  int inc(int x) {
    return x + 1;
  }
  fn(int) -> int times2 = \(int x) -> int: x * 2;

  printInt(apply(compose(inc, times2), 3));
  printInt(apply(compose(times2, inc), 3));

  return 0;
}
```

This language extension adds:

- function definitions as non-top-level definitions e.g. `inc` above

- function types e.g. `fn(int) -> int`
- lambda expression e.g. `\(int x) -> int: x * 2`

It is recommended that this extension is done after the `pointers` extension. The best way to implement function values is via closures, which are discussed in the later lectures.

Native x86 code generation

This extension is to produce [illegible] code for a real machine, preferrably x86. We may accept code generators for other archite[illegible] to think of how we can test your extension. Before you attempt to write a backend [illegible]ture, discuss your choice with the lecturer and explain the testing procedure.

Note that this extension gives you *two* credits, but it is not enough to just implement a naïve code generator. You must also implement some sort of optimization, such as register allocation or peephole optimization. Talk to the lecturer about which optimization(s) to implement before attempting the x86 code generator. The x86 code generation extension acts also as a kind of multiplier, that is, implementing another extension, for example arrays, will give you two credits instead of one. This fair because you need to generate code for both LLVM and x86.

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. Email the lecturer 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.

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 the lecturer in advance. Here are some possibilities:

- Implement an optimisation such as common-subexpression elimination, dead-code elimination, or loop-invaraint code motion as a Javelette-to-Javalette code transformation.
- Provide a predefined type of lists with list comprehensions, similar to what is available in Python.
- Allow functions to be statically nested.

- 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.
- Implement a backend for another architecture, such as RISC-V. It is important that you provide some way for the grader to t

The front

Your first task is to impleme end for Javalette:

1. Define suitable data ty esenting 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 to build your lexer and parser. We also recommend you use `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 the lecturer 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.

Hints for the extensions

The two simplest extensions are: the one-dimensional arrays extension and the dynamic structures (i.e. the pointers) extension.

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 [illegible]laration for `calloc`, but you do not need to worry about it at link time; LLVM:s linker i[illegible]

Indexing uses the `geteleme[illegible]` [illegible]which is discussed in detail in the lectures.

The LLVM does not include [illegible]th garbage collection. Thus, this extension should really include some means for re[illegible]ry 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.

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.

Structures/pointers 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.

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 LLVM (or the Java virtual machine (JVM), which provides 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 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. However, you must implement some form of quality improving optimisiation, e.g. a simple version of register allocation.

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

The Javalette language

Javalette is a simple impera[illegible]most a subset of C (see below). It can also be easily translated to Java (see belo[illegible]

Javalette is not a realistic la[illegible]n use. However, it is big enough to allow for a core compiler project that illustrates all p[illegible] It also forms a basis for extensions in several directions.

The basic language has no [illegible] 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.

Example 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) ;
}
```


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`. 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.

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 the notion of a *function type*, which is a pair consisting of the value type and the list of parameter types.

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.

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 [illegible]
- *Relational expressions*: [illegible] nd `>=`. All overloaded as expected.
- *Disjunctions and conjun*[illegible] These operators have *lazy semantics*, i.e.,
  - In `a && b`, if `a` e[illegible] `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.

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: \verb#\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).

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 are no variables of type string 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.

Parameter passing

All parameters are passed b[illegible]e of the actual parameter is computed and copied into the formal parameter before th[illegible]uted. Parameters act as local variables within the subroutine, i.e., they can be[illegible]

Javalette, C an[illegible]

Javalette programs can be c[illegible]piler (`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.

Submission format

Your submission should contain the following directories and files:

- Subdirectories `src`, `lib`, `doc`.
- A `Makefile` for building your compiler.

Submission contents

1. The `Makefile` should contain the targets `all` and `clean`.
   - The target `all` should build your compiler, and should create the following executables in the submission root:
     - (Assignment A): An executable `jlc`.
     - (Assignment A, B): An executable `jlc` for the LLVM backend.

        - (Optional, Assignment C): Executables `jlc_x86` or `jlc_x64` for the native 32/64-bit backends.
    - The target `clean` should remove all build artefacts.

2. The subdirectory `src` should contain:
    - all source code re[illegible] submission;
    - the grammar file [illegible]te.cf); and
    - *nothing else* (esp. [illegible]nerated code, etc).
3. The subdirectory `lib` [illegible]
    - (Assignment B, C) [illegible]eeded for your LLVM backend.
    - (Optional, Assign[illegible]e.s needed for your x86-32 or x86-64 backend.
4. The subdirectory `doc` should contain one plain ascii file 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.

Testing your submission

Please test your compiler before submission; see the section on testing. A submission that does not pass the test suite will be rejected immediately.

Testing the project

Needless to say, you should test your project extensively. We provide a test suite of programs and will run your compiler on these.

You can download the test suite from the course web site and run it locally or on a Chalmers machine (e.g., `remote11` or a Linux lab machine). The test suite 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.

Automated testing

Please see the test suite for instructions on how to test your submission. You **must** verify that your compiler passes the test suite before submission. Any submission that does not pass the test suite will be rejected immediately.