

# JLC - A Javalette Compiler

Sven Andersson

19860708-4632 – andsve@student.chalmers.se

Jhonny Göransson

19840611-8235 – jhonny@student.chalmers.se

2010-05-20

Revision 1

## 1 The JLC Compiler

### Building

To build the JLC binary execute;

```
$ cd src/
```

```
$ make
```

It will produce a `jlc` binary in the project root directory.

The following command will remove all files produced during building;

```
$ make clean
```

### Usage

```
$ ./jlc filename.jl
```

Will generate `filename.ll` (LLVM instruction file) and then translate this file into LLVM bitcode and output this to `./filename.bc`. The LLVM bitcode (`.bc`) must be linked together with the runtime library so that the read/print functions can be used. This is done by executing the `llvm` command `llvm-lc filename.bc lib/runtime.bc`.

### How it works

The compiler executes in the following 4 steps:

1. **Lexer** – Generates an abstract syntax tree from the source file.
2. **Typechecker** – Checks the code tree for type errors.
3. **(LLVM) Compiler** – Generates a `.ll` file with LLVM<sup>1</sup> instructions
4. **LLVM Bitcode Generation** – Writes and translates the instructions to LLVM-bitcode

### Extensions Implemented

None.

---

<sup>1</sup><http://llvm.org/>

## 2 BNFC Results

We get one shift/reduce conflict from BNFC, which comes from the "dangling" if/if-else rule.

## 3 The Javalette Language

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

### The lexical structure of javalette

#### Identifiers

Identifiers  $\langle Ident \rangle$  are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ ' ,`, reserved words excluded.

#### Literals

Integer literals  $\langle Int \rangle$  are nonempty sequences of digits.

Double-precision float literals  $\langle Double \rangle$  have the structure indicated by the regular expression  $\langle digit \rangle + \langle ' \rangle \langle digit \rangle + ( \langle ' e ' \rangle - ? \langle digit \rangle + ) ?$  i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

String literals  $\langle String \rangle$  have the form `"x"`, where *x* is any sequence of any characters except `"` unless preceded by `\`.

#### Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in javalette are the following:

<code>boolean</code>	<code>double</code>	<code>else</code>
<code>false</code>	<code>if</code>	<code>int</code>
<code>return</code>	<code>true</code>	<code>void</code>
<code>while</code>		

The symbols used in javalette are the following:

<code>(</code>	<code>)</code>	<code>,</code>
<code>{</code>	<code>}</code>	<code>;</code>
<code>=</code>	<code>++</code>	<code>--</code>
<code>-</code>	<code>!</code>	<code>&amp;&amp;</code>
<code>  </code>	<code>+</code>	<code>*</code>
<code>/</code>	<code>%</code>	<code>&lt;</code>
<code>&lt;=</code>	<code>&gt;</code>	<code>&gt;=</code>
<code>==</code>	<code>!=</code>	

## Comments

Single-line comments begin with #, //.

Multiple-line comments are enclosed with /\* and \*/.

## The syntactic structure of javalette

Non-terminals are enclosed between  $\langle$  and  $\rangle$ . The symbols  $::=$  (production),  $|$  (union) and  $\epsilon$  (empty rule) belong to the BNF notation. All other symbols are terminals.

```
 $\langle Program \rangle ::= \langle ListTopDef \rangle$ 

 $\langle TopDef \rangle ::= \langle Type \rangle \langle Ident \rangle ( \langle ListArg \rangle ) \langle Block \rangle$ 

 $\langle ListTopDef \rangle ::= \langle TopDef \rangle$ 
                    $| \langle TopDef \rangle \langle ListTopDef \rangle$ 

 $\langle Arg \rangle ::= \langle Type \rangle \langle Ident \rangle$ 

 $\langle ListArg \rangle ::= \epsilon$ 
                 $| \langle Arg \rangle$ 
                 $| \langle Arg \rangle , \langle ListArg \rangle$ 

 $\langle Stmt \rangle ::= ;$ 
             $| \langle Block \rangle$ 
             $| \langle Type \rangle \langle ListItem \rangle ;$ 
             $| \langle Ident \rangle = \langle Expr \rangle ;$ 
             $| \langle Ident \rangle ++ ;$ 
             $| \langle Ident \rangle -- ;$ 
             $| \text{return } \langle Expr \rangle ;$ 
             $| \text{return} ;$ 
             $| \text{if } ( \langle Expr \rangle ) \langle Stmt \rangle$ 
             $| \text{if } ( \langle Expr \rangle ) \langle Stmt \rangle \text{ else } \langle Stmt \rangle$ 
             $| \text{while } ( \langle Expr \rangle ) \langle Stmt \rangle$ 
             $| \langle Expr \rangle ;$ 

 $\langle Block \rangle ::= \{ \langle ListStmt \rangle \}$ 

 $\langle ListStmt \rangle ::= \epsilon$ 
                 $| \langle Stmt \rangle \langle ListStmt \rangle$ 

 $\langle Item \rangle ::= \langle Ident \rangle$ 
             $| \langle Ident \rangle = \langle Expr \rangle$ 

 $\langle ListItem \rangle ::= \langle Item \rangle$ 
                 $| \langle Item \rangle , \langle ListItem \rangle$ 

 $\langle Type \rangle ::= \text{int}$ 
             $| \text{double}$ 
             $| \text{boolean}$ 
             $| \text{void}$ 

 $\langle ListType \rangle ::= \epsilon$ 
                 $| \langle Type \rangle$ 
                 $| \langle Type \rangle , \langle ListType \rangle$ 
```

```

⟨Expr6⟩ ::=  ⟨Ident⟩
           |  ⟨Integer⟩
           |  ⟨Double⟩
           |  true
           |  false
           |  ⟨Ident⟩ ( ⟨ListExpr⟩ )
           |  ⟨Ident⟩ ( ⟨String⟩ )
           |  ( ⟨Expr⟩ )

⟨Expr5⟩ ::=  − ⟨Expr6⟩
           |  ! ⟨Expr6⟩
           |  ⟨Expr6⟩

⟨Expr4⟩ ::=  ⟨Expr4⟩ ⟨MulOp⟩ ⟨Expr5⟩
           |  ⟨Expr5⟩

⟨Expr3⟩ ::=  ⟨Expr3⟩ ⟨AddOp⟩ ⟨Expr4⟩
           |  ⟨Expr4⟩

⟨Expr2⟩ ::=  ⟨Expr2⟩ ⟨RelOp⟩ ⟨Expr3⟩
           |  ⟨Expr3⟩

⟨Expr1⟩ ::=  ⟨Expr2⟩ && ⟨Expr1⟩
           |  ⟨Expr2⟩

⟨Expr⟩  ::=  ⟨Expr1⟩ || ⟨Expr⟩
           |  ⟨Expr1⟩

⟨ListExpr⟩ ::=  ϵ
               |  ⟨Expr⟩
               |  ⟨Expr⟩ , ⟨ListExpr⟩

⟨AddOp⟩ ::=  +
           |  −

⟨MulOp⟩ ::=  *
           |  /
           |  %

⟨RelOp⟩ ::=  <
           |  <=
           |  >
           |  >=
           |  ==
           |  !=

```