Javalette Compiler

Javalette Compiler or **jlc** is a compiler for a small C-like language.

What?

A compiler written in haskell for the small C-like langauge called Javalette. This is for the course **TDA283, Compiler Construction** at Chalmers University of Technology, Sweden.

Copyright & License

Authors

The authors and contributors of this project are:

Björn Tropf

<bjoerntropf@gmail.com> (bjoerntropf at gmail dot com)

Mazdak Farrokhzad

<twingoow@gmail.com> (twingoow at gmail dot com)

Installation & Building

Requirements

The recommended versions below are the versions which the project was tested with and is known to be working.

- **make** or gradle
- **haskell**, recommended: 7.10.2
- **cabal**, recommended: 1.22.4.0
- **bnfc** >= 2.8
- **alex**, recommended: 3.1.4
- **happy**, recommended: 1.19.5
- **pandoc**, recommended: 1.17.0.2, for building documentation.

Building, from git repository:

First clone the repository to your local machine:

```
git clone https://github.com/Centril/TDA283-compiler-construction
```

Assuming that:

```
cd TDA283-Compiler-Construction
```

If you are building from the git repository with gradle, you can do:

```
./gradlew build
```

To build and run tests.

or alternatively using make:

```
make all
```

Building, from src/ repository (using make):

Assuming that:

```
cd src/
```

```
make all
```

Javalette Grammar conflicts

shift-reduce

The standard dangling else:

```
if ( cond ) { s; }
if ( cond ) { s1; } else { s2; }
```

reduce-reduce

There are no reduce-reduce conflicts.

Javalette language specification

The lexical structure of Javalette

Identifiers

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

Literals

Integer literals *Integer* are nonempty sequences of digits.

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

String literals *String* 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:

`String`	`boolean`	`double`	`else`
`false`	`if`	`int`	`return`
`true`	`void`	`while`	

The symbols used in Javalette are the following:

(	)	,	{	
}	;	=	++	
–	-	!	&&	
		+	*	/
%	<	<=	>	
>=	==	!=		

Comments

Single-line comments begin with #, //.Multiple-line comments are enclosed with /* and */.

The syntactic structure of Javalette

Non-terminals are enclosed between < and >. The symbols -> (production), \| (union) and **eps** (empty rule) belong to the BNF notation. All other symbols are terminals.

Program	->	TopDef
TopDef	->	*Type Ident* `(` Arg `)` *Block*
	->	*TopDef* TopDef
	|	*TopDef* TopDef
Arg	->	*Type Ident*
	->	**eps**
Arg	|	*Arg*
	|	*Arg* `,` Arg
Block	->	`{` Stmt `}`
	->	**eps**
Stmt	|	*Stmt* Stmt
Stmt	->	`;`
	|	*Block*
	|	*Type* Item `;`

	\/	*Ident* `=` *Expr* `;`
	\/	*Ident* `++` `;`
	\/	*Ident* `--` `;`
	\/	`return` *Expr* `;`
	\/	`return` `;`
	\/	`if` `(` *Expr* `)` *Stmt*
	\/	`if` `(` *Expr* `)` *Stmt* `else` *Stmt*
	\/	`while` `(` *Expr* `)` *Stmt*
	\/	*Expr* `;`
Item	->	*Ident*
	\/	*Ident* `=` *Expr*
	->	*Item* Item
	\/	*Item* `,` Item
Type	->	`int`
	\/	`double`
	\/	`boolean`
	\/	`void`
	->	**eps** Type
	\/	*Type*
	\/	*Type* `,` Type
Expr6	->	*Ident*
	\/	*Integer*
	\/	*Double*
	\/	`true`
	\/	`false`
	\/	*Ident* `(` Expr `)`
	\/	*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*
	->	**eps**
Expr		
	\/	*Expr*
	\/	*Expr* `,` Expr
AddOp	->	`+`
	\/	`-`
MulOp	->	`*`
	\/	`/`
	\/	`%`
RelOp	->	`<`
	\/	`<=`
	\/	`>`
	\/	`>=`
	\/	`==`
	\/	`!=`