**Language Documentation**

ODang is an object-oriented language written in Java that compiles to Javascript. We chose Java due to it being the language we were all strongest in. We did not design this language with any specific goal in mind other than to learn about compilers and practice compiler design.

We based our concrete grammar on Java’s syntax specification (<https://cs.au.dk/~amoeller/RegAut/JavaBNF.html>). ODang supports only boolean, string, and integer primitives; we didn’t see any reason to support other numerical types like byte, short, long, float, or double since ODang isn’t going to be used for any professional purposes anytime soon, and while primitive chars aren’t supported, strings can be used to handle text. It also supports if/else statements, while loops, for loops, println statements, and calls to super classes. We do not provide support for packages, access modifiers, garbage collection, arrays, switch/case statements, try/catch blocks. Our notable features include object-oriented classes and subtyping using prototype-based inheritance.

**Lessons Learned**

In terms of management, we didn’t have as much organization and communication as we should’ve had initially, as we kept doing things that someone else might’ve been working on unknowingly, so when changes finally did get pushed, a lot of the work another team member might’ve done would’ve just gone to waste. However, it did get better as the project went on as we continued running into these problems.

Next, we should have focused more on creating a concrete syntax of our language that would be simple to program instead of relying on the above link. It would have made things easier in the long run.

In addition, we would have probably chosen a different target language than Javascript since we were not at all that familiar with it as a group in the end.

Implementation-wise, we definitely should have had a better grasp of how we wanted the compiler to behave. As the codebase entered into thousands-of-lines territory, it became clear that some of our implemented features did not really scale well in regards to both future components and testing, and the solution became a combination of reworks and “hacky” fixes, based on how big the changes would have to be.

**Getting Started**

You can either:

1. Download the ODang.jar and run it with java -jar ODang.jar
2. Download the source and run it with Maven using mvn exec:java -D exec.mainClass=ODang

**Features**

Odang implements object-oriented classes and subtyping into JavaScript using prototype based inheritance. Notable features:

* Object-oriented classes
* Subtyping
* Strong types

**Known Limitations**

* Does not feature garbage collection
* Casts can only be made to Strings and Classes
* Classes cannot contain more than one constructor
* Does not contain access modifiers
* Does not contain bitwise operators

**Code Snippets**

ODang uses a Java-like syntax with some simplifications so much of the syntax is similar to Java except for the above limitations.

Legal types: int, boolean, String, ClassType

*Class Definition*

* class Foo {}
* class Foo extends Bar {}

*Class Instance Creation*

* Foo bar = new Foo();
* Foo bar = new Foo(int x, String y);

*Method Declaration*

* int plusOne(int x) {
* return x + 1;
* }
* class Foo {
* int x;
* int getX() {
* return this.x;
* }
* }

*Field declarations and access*

* class Foo {
* int x = 2;
* }
* class Bar {
* Foo obj = new Foo();
* int y = obj.x + 2;
* }

*Statements*

* if (i == 5) {
* i++;
* } else {
* i--;
* }
* while(true) {
* println(“Ha ha ha.”);
* }
* for (int i = 0; i < 10; i++){
* println((String)i);
* }

**Formal Syntax**

Program:

<compilation unit> ::= <class decs>?

Declarations:

<class decs> ::= <class dec> | <class decs> <class dec>

<class dec> ::= class <identifier> <super>? <class body>

<super> ::= extends <class type>

<class body> ::= { <class body decs>? }

<class body decs> ::= <class body dec> | <class body decs> <class body dec>

<class body dec> ::= <class member dec> | <constructor dec>

<class member dec> ::= <field dec> | <method dec>

<constructor dec> ::= <constructor declarator> <constructor body>

<constructor declarator> ::= <identifier> ( <formal param list>? )

<formal param list> ::= <formal param> | <formal param list> , <formal param>

<formal param> ::= <type> <identifier>

<constructor body> ::= { <explicit constructor invocation>? <block stmts> }

<explicit constructor invocation> ::= super ( <argument list>? )

<field dec> ::= <type> <var declarators> ;

<var declarators> ::= <var declarator> | <var declarators> , <var declarator>

<var declarator> ::= <identifier> | <identifier> = <expr>

<method dec> ::= <method header> <method body>

<method header> ::= <result type> <method declarator>

<result type> ::= <type> | void

<method declarator> ::= <identifier> ( <formal param list>? )

<method body> ::= <block> | ;

Statements:

<block> ::= { <block statements>? }

<block statements> ::= <block stmt> | <block stmts> <block stmt>

<block stmt> ::= <local vardec stmt> | <stmt>

<local vardec stmt> ::= <local vardec> ;

<local vardec> ::= <type> <vardeclarators>

<stmt> ::= <stmt without trailing substmt> | <if then else stmt> |

           <while stmt> | <for stmt> | <print stmt>

<stmt without trailing substmt> ::= <block> | <empty stmt> | <expr stmt> |

           <break stmt> | <return stmt>

<empty stmt> ::= ;

<expr stmt> ::= <stmt expr> ;

<stmt expr> ::= <assignment> | <preincrement expr> | <predecrement expr> |

                <postdecrement expr> | <postincrement expr> |

                <method invocation> | <class instance creation expr>

<print stmt> ::= println ( <expr> )

<if then else stmt> ::= if ( <expr> ) <block> else <block>

<while stmt> ::= while ( <expr> ) <block>

<for stmt> ::= for (<for init>? ; <expr>? ; <for update>? ) <block>

<for init ::= <stmt expr list> | <local vardec>

<for update> ::= <stmt expr list>

<stmt expr list> ::= <stmt expr> | <stmt expr list>

<break stmt> ::= break <identifier>? ;

<return stmt> ::= return <expr>? ;

Types:

<type> ::= <primitive type> | <class type>

<primitive type> ::= <numeric type> | boolean

<numeric type> ::= int | str

<class type> ::= <identifier>

Expressions:

<expr> ::= <assignment expr>

<assignment expr> ::= <equality expr> | <assignment>

<assignment> ::= <left side> <assignment op> <assignment expr>

<left side> ::= <expr name> | <field access>

<assignment op> ::= = | += | -=

<equality expr> ::= <relational expr> | <equality expr> == <relational expr>

                          <equality expr> != <relational expr>

<relational expr> ::= <additive expr> | <relational expr> < <additive expr> |

                            <relational expr> > <additive expr>

<additive expr> ::= <multiplicative expr> | <additive expr> + <multiplicative expr>

                    <additive expr> - <multiplicative expr>

<multiplicative expr> ::= <unary expr> | <multiplicative expr> \* <unary expr> |

                          <multiplicative expr> / <unary expr>

<cast expr> ::= ( <primitive type> ) <unary expr> | ( <class type> ) <unary expr>

<unary expr> ::= <unary expr no incr decr> | <predecrement expr> | <preincrement expr> |

<predecrement expr> ::= -- <unary expr>

<preincrement expr ::= ++ <unary expr>

<unary expr no incr decr> ::= <postfix expr> | - <unary exp> | ! <unary expr> | <cast expr>

<postfix expr> ::= <primary> | <postincrement expr> | <postdecrement expr>

<postdecrement expr> ::= <primary> --

<postincrement expr> ::= <primary> ++

<method invocation> ::= <method name> (<argument list>? ) | <field access> ( <argument list>?)

<field access> ::= <primary> . <identifier> | super . <identifier>

<method name> ::= <identifier>

<primary> ::= <literal> | this | ( expr ) | <class instance creation expr> | <method invocation>

<class instance creation expr> ::= new <class type> ( <argument list>? )

<argument list> ::= <expr> | <argument list> , <expr>

Literals:

<literal> ::= <identifier> | <string> | <integer> | <boolean literal> | <null literal>

<identifier> E Identifier

<string> E String

<integer> E Integer

<boolean literal> ::= true | false

<null literal> ::= null

<operator> E Operator