**Language Design Proposal: ODANG**

**Student Names**: Charles Dang, Giovanni Orozco, Marius Larnoey

**Language Name**: ODANG

**Compiler Implementation Language and Reasoning**: Java. We already know how to code in Java.

**Target Language**: Javascript

**Language Description**: Object-oriented programming. We’re going to create a Java-like language that is more optimized and useful.

**Planned Restrictions**: No garbage collection.

**Abstract Syntax**:

var is a variable

i is an integer

str is a string

classname is the name of the class

methodname is the name of the method

type ::= Int | String| Char | Boolean | Void

op ::= \* | / | + | - | = | % | > | < | ! | ++ | --

expr ::= var | str | i |

this | // reference to current object

println(expr\*) | //prints text on console

expr op expr | //arithmetic operations

expr.methodname(expr\*) | //call a method with any number of expressions as parameters

new classname(expr\*) /\*create a new class with any number of expressions as parameters\*/

(type) expr // cast the expression to a type, runtime error if cast to something it isn’t

vardec ::= type var // declare a variable

booleanstmt ::= var < int || var > int | var != int || var == int /\* boolean statement that compares values \*/

stmt := vardec = expr; | //simple variable assignment

var = expr; | assign an expression to a variable

while (expr) stmt | // while loop

for (var = expr; booleanstmt; var++ || var--) stmt | // for loop

break; | // break

{ stmt\* } | // block of statements

if (expr) stmt\* else stmt | // if statement

return expr; | // return expression

return; // return Void

methoddef ::= methodname(vardec\*) stmt /\* a method definition that contains any possible number of variable declarations separated by commas as parameters\*/

classdef ::= class classname extends classname //implies subtyping

vardec\*

classname (vardec\*) stmt /\* a constructor that has any number of variables that are separated by commas\*/

[classname operator op (classname &var) stmt] // implement operator overloading

methoddef\*

}

program ::= classdef\*

**Computation Abstraction Non-Trivial Feature**: Object-oriented classes

**Non-Trivial Feature #2**: Subtyping

**Non-Trivial Feature #3**: Operator Overloading

**Work Planned for Custom Component**: Subtyping will be left for the end since we need object-oriented classes implemented first

Concrete grammar:

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> | <method overload>

<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> ::= this ( <argument list>? ) | 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> | ;

<method overload> ::= <overload declarator> <overload body>

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

<overload 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