Language Documentation
for group 'Oligarchy'
COMP 430 Spring '20

Why this language and why this language design?

A large part of what drove us to create this language was for the learning experience we as a team would face. By introducing concepts both familiar and unfamiliar, we hoped to learn more about how compilers and its components work.

Some aspects of our language were chosen due to their simple and fundamental nature. This meant keeping our design close to a syntax that is familiar. Doing so should also help to make the language more accessible to the general audience of programmers.

Problems our language may help solve are those most affected by expression, sub-classes, and higher order functions. Since our language is partly inspired by object oriented programming, we provided elementary aspects of inherited variables and methods which give the user some flexibility when structuring classes and objects, by way of extending the acquired properties and behaviors of another class. Such functionalities mentioned help to provide another level of abstraction, which is useful when dealing with complexities in implementations.

Some components were scrapped, such as sub-typing, due to time restraints and implementation issues. Some restrictions were planned, such as no optimization, global variables, and data type restrictions. Outside variables must be named differently from local variables. Variable declarations are not "smart" and must be declared explicitly. This was done for the sake of simplicity. Similarly, booleans are restricted to "true" and "false" and do not allow for other tokens/values. Since we are using a low level target language, we will not have accessors as well.

Code snippets in your language highlighting features and edge cases, along with relevant explanation.

The following code snippets are merely a small sample of what our programming language is capable of. It is meant to highlight specific features and test cases that help to provide some complexity and justification to our language design.

```
int x = 1;

int y = x + 2;
```

Code Example of expression of variable declaration and addition binary operation along with assignment.

```
int w = (5 * 10) + 3*(7) / 10 + 12-10;
```

Code Example of variable declaration and assignment using various binary operations (multiplication, division, addition, and subtraction). This also exercises the operation priority.

```
class classOne {
    int a = 3;
    constructor (int num) int b = 4;
    int methodOne(int num2) int c = 5; return;
    }

class classTwo extend classOne {
    int d = 6;
    constructor (int num3) int e = 7;
    int methodTwo(int num4) int f = 8; return;
    }
```

Code example of class inheritance. Also makes use of constructor method, class method, and class variable declaration.

```
classOne.methodOne;
```

Code example of method called on a class. Returns void or a type.

```
if (i = 10 | i = 5){
    return true;
    }
else{
    return false;
    }
```

Code example of an if/else statement. If condition contains a boolean expression. Example shows use of return statement.

```
for (int i = 0; i <= 10; i+1) { w = 10 }
```

Code example a for loop. For loops can take in a variable declaration and assignment, a boolean, and an expression. Brackets hold a body of statements.

Known Limitations.

Some limitations were expected. We didn't plan to have optimizations. There was no "this" feature in Classes, so outside variables have to be named differently from local variables. Runtime errors will occur when casting variables of different types or logical operations are performed on numbers.

The grammar we created introduced additional limitations. Some options that could have been provided to the user, and may have subsequently given more flexibility, had to be abandoned due to time constraints and feasibility. For example, when creating the grammar we neglected to provide an option for the user to have getters and setters. This ultimately impacts their ability to make use of

instance variables and parameters.

Limitations in specific features such as High Order Functions and types of Expressions have been brought to light during development of code generation. Many corrections to such limitations required a complete reworking of some aspects of previous components, including the syntax tree, parser, and type-checker.

Additionally, we chose not to include memory de-allocation and generics into our language. Such features were deemed unfeasible considering our existing workload and the limited time of a single semester to work in.

Knowing what you know now, what would you do differently?

Considering the difficulty in implementing multiple features in a low level, target language, having the user take on some things may help lessen the load on we the developers. Even if this resulted in the language becoming more cumbersome for the user (such as LISP may be for some), it would have eased the complexity of many aspects of the compiler, such as code generation and testing.

Some components would have been easier to implement by putting certain requirements on the user. By making the user write out more information in their code, the token generator and type-checker would have more details regarding their context.

Much of our difficulty stemmed from providing a runnable compiler at end. This may have been helped by starting with a simple program in our designed language and then slowly introducing additional features into the program.

A concrete grammar turned out to be very important. Some components, such as the parser, would have benefited greatly because as changes or updates were needed, things became more confusing.

Changes in our procedures and tools may have helped as well. Being able to write out test cases first may have given an easier handle on how the code was to be written. Using Scala was new for all team members. Some features of the language, and tools available, could have made things much easier and more efficient. Being able to make use of Scala Test and SBT better would have helped tremendously.

While choosing a low level, target language was educating, it was also stressful. Without a great deal of help from the professor, it would have been too far removed from the scope of most team members. Given that the code generation was at the end of the semester, it unfortunately coincided with unforeseen circumstances (such as the school shutdown) and other classes finals. This created a big spike of necessary learning and work within a short period of time that was already compacted. Hence, changes to planned code generation would have stabilized some of that work.

How do I compile your compiler?

Todo.

How do I run your compiler?

Todo.

Formal syntax definition.

var is a variable

classname is the name of a class

func is the name of a method

str is a string

int is an integer

Boolean is true or false

type ::= Int | Boolean |

Void | [Void will only be used as a return type]

Class [Class type includes Object and String]

math ::= + | - | * | / | ^ | < | > | <= | >= [Arithmetic operations]

logic ::= && | || | == [something that needs logic on each side to evaluate]

exp ::= var | str | i | [Variables, strings, and integers are expressions]

Boolean |

exp_1 logic exp_2 |

print(exp) | [Prints something to the terminal]

exp_1 math exp_2 | [Arithmetic operations : adds planned restriction]

exp_1.methodname(exp_2*) | [Calls a method]

new classname(exp*) | [Creates a new instance of a class]

(type)exp | [Casts an expression as a type : bad casts are in planned restrictions]

(type var) => exp | [variable is in scope and will probably be used in the exp]

exp_1(exp_2) [how to call a high order function]

vardec ::= var [Variable declaration : add back changes]

stmt ::= exp; |

var = exp; | vardec = exp; | [Assignment]

for(vardec^; exp; stmt^) stmt | [^ optional omit them to be treated as a while loop]

break; | [break]

{ stmt* } | [block]

if (exp) stmt else stmt | [if/else]

return exp; | [return an expression]

return; [return Void]

methoddef ::= type methodname(vardec*) stmt [vardecs are comma-separated]

instancedec ::= vardec; [instance variable declaration]

classdef ::= class classname extends^ classname^ { [^ indicates optional]

instancedec*

constructor(vardec*) stmt [vardecs are comma-sep]

methoddef*

}

program ::= classdef* exp [exp is entry point]