JK-Java

Documentation for a compiler created by Jason Dang and Kodi Winterer

Table of Contents

Language Design

- Why this language? - The main reason we wanted to choose and design a language like JK-Java was because we wanted to learn more about object oriented programming and some of its features like access modifiers and generics, so we chose to design a stripped-down version of Java. Hence the name, JK-Java (JK being our first name initials).
    - Like mentioned above, JK-Java is a stripped-down version of Java, most notably retaining its object oriented nature but missing some features such as booleans, conditionals, for/while loops, etc.
    - Our implementation language for this compiler was Java, because both of us were comfortable with producing large amounts of code in Java given its extensive use here at CSUN in previous courses. We both came to an agreement in having our target language in C because we both had varying prior experiences with C.
- Why this language design? - Mainly, our language is based on the object oriented nature of Java, so it aims to cut down on some features while implementing key features such as generics, access modifiers, and class based inheritance.
    - Many things were restricted throughout our design and implementation process of our compiler, including conditionals and for/while loops near the beginning of our design process because they weren't necessary in accomplishing the non-trivial features of our language.
    - In addition, we eventually decided to add missing components that we originally didn't include in our formal written abstract syntax, including constructors in class definition, and a program definition after noticing we didn't have those initially in parsing/typechecker.
    - Some things that we had changed from our original syntax definition for JK-Java includes: only allowing variable parameters in expressions (e.g. println(var) only as opposed to println("...") or println(1)), adding in constructors that were missing before, a program definition, and more discussed below in Known Limitations.

Features

Sample "Hello World" program in JK-Java

```
String s;
s = "Hello World";
println(s);
```

- Object oriented class based inheritance - Classes of objects can be created just like in Java, and can inherit from another from the extends keyword. Classes have instance variables which are initialized in constructors, as well as methods which take in statements. Statements can be added outside of classes after all class declarations are added.

Sample class Person with instance variables age and name with getter and setter, also statements outside of class. Method call works since it is public.

```
public class Student{
        private int age;
        private String name;
        public Student(int a, int n){
                this.age=a;
                this.name=n;
        }
        public int getAge(){
                return this.age;
        }
        public int getName(){
                return this.name;
        }
        public void setAge(int a){
                this.age=a;
        }
        public void setName(int n){
                this.name=n;
        }
}
int age;
age = 18;
String name;
name = "Joe";
Student s;
S = new.Student(age, name);
int newage;
newage = 19;
s.setAge(newage);
```

Simple example of class based inheritance in which Student inherits from Person

```
public class Person{
        private String name;
        public Person(String n){
                this.name=n;
        }
}
public class Student extends Person{
        private String name;
        private int schoolYear;
        public Student(String n, int s){
                this.name=n;
                this.schoolYear=s;
        }
}
```

- Access Modifiers - Public/private access modifiers are used in method calling, private methods cannot be accessed outside of the classes they are created in.
    - Examples of this usage is shown above in both examples with classes, in which public/private access modifiers are used for classes, instance variables, and methods.

Quick example of code that will fail since the last statement is calling a method that is private.

```
public class Person{
        private int age;
        public Person(int a){
                this.age=a;
        }
        private void resetAgeToOne(){
                this.age = 1;
        }
}
int age;
Person p;
p = new.Person(age);
p.resetAgeToOne();
```

- Generics - Generic programming is added in our compiler to allow for later declaration of types in classes. Included in areas of our syntax including class definition, new class object expressions, types (T and classname<type>), etc.

Simple example of generics used in a class.

```
public class Student<A>{
        private A variable;
        public Student(A var){
                this.variable = var;
        }
        public A getVar(){
                return this.variable;
        }
}
String name;
name = "John";
Student<String> s;
s = new.Student<String>(name);
```

Known Limitations

- What is missing?
    - Not necessarily a missing feature, but our language does not allow for direct calls to instance variables at this point, so access modifiers on instance variables are practically meaningless as of right now.

Direct calls to instance variables like this in Java do not work in JK-Java (assuming age is public in class Student)

```
int age;
Student s;
s = new.Student(age);
int temp;
temp = s.age;
```

-
    - We included access modifiers on classes in the beginning of our abstract syntax design, but we decided to scrap it although the keywords are still there, as our language is rather limited to actually use it.
- What is hard to accomplish with JK-Java?
    - Like mentioned many times in other sections, utilization of our language is very limited due to the fact that we are missing booleans, conditionals, loops, arrays, etc.
    - Statements must be added after class declaration in our language syntax, so this is a limitation with freedom in ordering code.
        - This can be seen in some of the examples of code snippets under features.
    - Variable declaration and assignment must be done in separate lines according to our syntax.

Short example of assignment of variables taking up lines inefficiently

```
//In JK-Java

int age;
age = 2;

//Must be done, as opposed to what you can do in Java with:

int age = 2;
```

-
    - No use of super(), so variables must be initialized again in child classes
    - Newlines in translated code are currently not working so it can be hard to read.

- Known bugs as of <5/16/19>

    - Typechecker allows duplicate constructors (same number/type of instance variables)
    - Typechecker does not check child class constructor if it initializes the parents' instance variables or not (e.g. Student with int age, Junior which extends from Student should always initialize age but typechecker right now does not check for it)
    - Our language should not allow access to instance variables within the same class in methods/constructors without usage of the this.var expression. However typechecker currently allows that in constructors (methods work fine in this regard).

<u>What would we do differently?</u>

- In regards to JK-Java's design, I don't think that we would've wanted to change much considering the fact that we chose this language design because we were familiar with Java and wanted to learn more about object oriented programming as a result.
    - Obviously many of the cuts we had to make throughout our implementation process (e.g. no conditionals and loops, unary minus, booleans, etc.) were necessary in order to make deadlines or make our job easier in implementing the more important non-trivial features
    - In terms of our C translation design, there isn't much we would've changed in terms of the actual design, but we would have tried to plan ahead much earlier in learning why/how C works in order to accomplish the task of creating a design. Since both of us initially did not have this understanding, it was difficult to make a design in the initial deadline for abstractions of computations.
- Honestly, if we were to redo this entire project, we wouldn't change any of the build tool we used (Eclipse) mainly because both of us were used to using it and using it to test all of the individual components of our compiler. Our target language would also not be changed because we had chosen C for the main reason that both of us had some experience with C going into the class.
    - Although we had later trouble with analyzing and learning more about v-tables and pointers in C, I don't think learning another language that either of us may not have known prior to this class would have been a good idea, considering the amount of work that went into translation into C anyways.
- In the beginning of our project implementation process, we had several issues with learning how to use GitHub comfortably, as well as learning to work in a way that enables us to work on the project at the same time without any merge issues.

    - In addition, this would include problems with either of us being left in the dark mainly because we weren't committing as much as needed for the other to understand the code that was necessary for earlier deadlines.
    - Going into code generation, both of us were severely unprepared in knowing what needed to be done for C translation, so there was communication issues when it came to the third deadline with abstractions of computations in using v-tables and understanding the use of pointers in C overall.
- We also had some issues with missing components that should have been in our language to begin with for practical use
    - This includes independent function calls, lists of parameters instead of single parameters, constructors, etc.
        - These have had to be retroactively added into our compiler's many parts
- Overall, the main thing that we would've kept in mind if we had a redo in this project is to do more research in translating components of our language into C, as early as possible to prevent further confusion down the line. Also, there would have been stricter deadlines for better time management as our communication on this wasn't as clear as possible. Otherwise, there isn't much that would have changed knowing what we know now, besides trying to get more group members since only having two people was a struggle for both of us.
    - Also, better communication in between commits may have been something to keep in mind, especially for earlier deadlines while still learning how to use Github as a group effectively.

Compilation Instructions

Our compiler was written and built mainly using Eclipse for compilation, so compiling it in Eclipse would probably be preferable in ensuring that everything works the way that it should with JUnit included.

Runtime Instructions

Navigate to the Main file under *JK-Java/Main/JK-Lexer* and make sure "input.txt" is in the command line argument. Type into the input.txt file what needs to be compiled from JK-Java, run the Main file, and an output.txt file should appear in the project directory with the translated C code (An example JK-Java program is already loaded into input.txt for reference).

Formal Syntax

VAR is variable
INT is an int
STRING is a string

```
CLASSNAME is the name of a class
METHODNAME is the name of a method
type::= INT | VOID | T                          Built in types, with generic type T
        classname |                         Class type including Object and String
        classname<type>                     Generic typing in class object represented by <>
op::= + | - | * | /                         Arithmetic operation
exp::= VAR | STRING | INT | <type> |            Variables, strings, integers, generics are
expressions
        this.VAR |                          My instance, must be used to call instance variables
        exp op exp |                        Arithmetic operation
        VAR.METHODNAME(VAR) |               Calls method
        new.CLASSNAME(VAR)    |             Creates new instance of class
        new.CLASSNAME<type>(VAR)            Creates new instance of a generic class
vardec::=type VAR;                          Declares variable
stmt::=vardec;          |                       Declares variable
        VAR=exp; |                          Assignment
        return exp; |                       return an expression
        return;                             return Void
        println(exp) |                      Print statement
accessModifier::= public | private          Access modifiers
methoddef::=accessModifier type METHODNAME(vardec*) stmt Creates method
instancedec::= accessModifier vardec;                    Declares instance of variable
classdef::= accessModifier class CLASSNAME{ |            Normal class declaration
             accessModifier class CLASSNAME<type>{ |     Generic class declaration
             accessModifier class CLASSNAME extends CLASSNAME{  Class with inheritance
                instancedec*
                constructor(vardec*) stmt
                methoddef*
             }
program::= classdef*  stmt*                              Program Definition
```