WAC(Wow Awesome Code)

Documentation for a language and compiler (WAC).
Developed by: Sarah Canto, Adrian Hernandez, Cesar Barrera, Ruben Perez

Table of Contents

Language Design

- Why this language? - We chose WAC to be a basic form of object oriented programing; similar to a small subset of Java. We did this to highlight the specific OOP features we wanted to tackle. We are all familiar with object oriented programming languages from previous classes so we thought it would be a way to deeper understand this subject we all have studied.
- Why this language design? - As mentioned above, a few key features were chosen for WAC from a superset of all OOP features so that we can focus on those specifically and how they would translate to a non-object oriented language(C in our specific case). Our syntax is defined without any left recursion to avoid issues in parsing and some things have been altered/omitted for ease of compilation. For example: forcing variable instantiation at time of declaration, only allowing super to be used for invoking a parent constructor, and requiring that method calls be only used on variables therefore not allowing chaining on '.'.

Features

- Class-based inheritance with objects and subtyping - Our language supports class based inheritance; you can create different classes, use inheritance for instance variables and methods, and create objects from inherited classes.

```
class Animal {
      String name = " ";

      Animal(String name) {
           this.name = name;
      }

     Boolean eats() {
          return true;
     }
}
```

Above is an example of creating a parent class

```
class Dog extends Animal {
     Int numOfLegs = 0;

      Dog(String name, Int numOfLegs) {
           super(name);
           this.numOfLegs = numOfLegs;
     }

     String noise() {
          return "bark";
     }
}
```

Above is an example of creating a child class that extends from the earlier defined parent class.

```
Dog Steve = new Dog("Steve", 4);
Boolean retvalEats = Steve.eats();
String retvalNoise = Steve.noise();
println(retvalEats, “, “, retvalNoise);
```

Above is an example of creating an object and calling its normal and inherited methods.

After execution it should print:

```
true, bark
```

```
Dog Steve = new Dog("Steve", 4);
println(Steve.numOfLegs);
println(Steve.name);
```

Above is an example of creating an object of a subclass and accessing its instance variables

After execution it should print:

```
4
Steve
```

- Method Overloading - Our language also supports method overloading. The constraint of our method overloading feature is that two methods that have the same name must have a different number of parameters

```
Int getNum(Int x) {
    return x;
}

Int getNum(Int x, Int y) {
    if (x < y) {
        return x;
    }
}
```

Above is an example of method overloading in our language

- What is missing?
    - The main thing that's missing in our compiler is a code generator. Due to time constraints we were not able to make a code generator.
- What is hard to do with WAC?/Quirks in our language/compiler
    - Data privacy; since our language doesn't contain access modifiers all variables and methods are set to public by default. This limits the ability to hide data by using private/protected access modifiers.
    - A quirk with our language syntax is in our method calls. Our method calls are of the form: var.methodname(primary_exp*) which means that even to make a local method call to a method inside of the current class you're in you have to call it with the class name.
      Ex: findNum(x) VS main.findNum(x)
    - Checking for return from methods. Our language currently checks return type matching through the typechecker but it does not actually check whether a method definitely returns, maybe returns, or never returns. This was specified as undefined behavior and left up to the programmer.
    - Since we are using a Java HashMap in our typechecker, we have "last one wins" semantics when declaring variables.
- Known bugs
    - There is a bug in our tokenizer regarding tokenizing variables of type "classname". Since we used the idea of stripping all whitespace from the input file, tokenizing something like: Dog steve should tokenize as two variable tokens 'Dog' and 'steve' but in actuality it currently tokenizes as one variable token 'Dogsteve'. This was caused by altering our language

syntax after developing our tokenizer and not having time to go back and refactor the tokenizer.
- When we created our parser we allowed a class definition to exclude the ‘extends’ keyword and parent class. But our typechecker actually requires that everything has a parent class (using Object as the base class). Therefore if we try to typecheck an AST for a class that doesn’t contain an instance variable for the parent class name it doesn’t typecheck correctly.

Known Limitations

- Would we design anything differently?
    - For the most part, our group is happy with what we have accomplished and would not design anything differently other than add the code generator which we did not have time for because without that, we are unable to run the code end to end and have to work with each piece of the compiler individually. We think the implementation of our program was fairly simple but at the same time, posed challenges that we had to overcome thus increasing our programming skills.
- Different build tools? Target language?
    - Our group was happy with the build tools used. We can’t know if we would change the target language because we never got to make the code generator.
- Management and team development issues?
    - Luckily our group communicated well and we were able to split the work. Sarah was the lead of our group because she had the most experience and she lent her hand to others in the group to help with development issues we ran into. We ran into a couple issues with GitHub when others would accidentally push bad code to a branch and we had to roll branches back but other than those couple of instances, everything was pretty standard. We had to deal with errors that came with adapting Dewey’s code to our implementation but that is expected. Overall it was a good experience.

What would we do differently?

- For any changes regarding our language (WAC). There haven't been any real issues with our overall language decision. We were most familiar with both Java and C, so this seemed to be the most appropriate decision. Most of us had

familiarity with Java and the idea of Object Oriented Programming, so this wasn't a far jump from our comfort zone.
    - Main improvement would be each group member getting more familiar with the languages being used to better be able to translate ideas and algorithms into actual working code.
- As for project management, when developing WAC, we could have chosen a shared IDE or at least become more familiar with each other's IDEs and their pros and cons.
    - As we began our project, we encountered many compatibility issues when pulling and merging repositories. The different IDEs would cause conflicts, as some IDEs would create additional files specific to the IDE and some IDEs would format code comments in a different manner. This wasn't a big deal, but it did cause time set backs when trying to get the project started.
- Another improvement would be getting familiar with Github and managing our Github commits differently. We have had a couple issues with commits. We've had moments when commits were accidentally pushed to the incorrect branches, which would cause delays during our meetings. So learning our way around Github could prevent mistakes earlier on in the project.
- Time management could have been improved on. Our compiler does not have a code generator due to time limitations. The class was falling behind in general, so this aspect was a bit out of the group's control. But during the final stretch, we could have better organized our time to get some progress done on the code generator.

Compilation Instructions

- We used Maven as our build tool and JUnit as our testing framework. Our code was compiled through terminal using maven commands(namely mvn compile and mvn test for testing). Our file directory on git is reflective of the directory structure needed for using Maven. "mvn compile" should be run from the /WAC folder and a .pom file specific to your machine needs to be included in that directory.

Runtime Instructions

- Since we don't have a functioning code generator there is currently no way to run our compiler from end to end.
- Each part of the compiler takes in something and (possibly)returns something else:

-Tokenizer: Takes in a string that represents the entire program and returns a list of tokens.
-Parser: Takes in a list of tokens and returns a list of class definition ASTs and a list of statement ASTs
-Typechecker: Takes in the above mentioned lists of ASTs and ensures that it's well typed

Formal Syntax Definition

```
var is a variable
int is an integer
str is a string
methodname is the name of a method
classname is the name of the class
constructor is the name of the object
'*' represents the traditional Kleene star meaning

type ::= Int | Boolean | String | classname     // basic types
primary_exp ::= var | str | int | true | false  //base case values
multiplicative_op ::= * | /      // highest precedence operators
multiplicative_exp ::= primary_exp (multiplicative_op primary_exp)*
                          // highest precedence expressions
additive_op ::= + | -     // middle precedence operators
additive_exp ::= multiplicative_exp (additive_op multiplicative_exp)*
                          // middle precedence expressions
comparison_op ::= < | > | == | !=
comparison_exp ::= additive_exp | additive_exp comparison_op
additive_exp              // lowest precedence expressions
exp ::= comparison_exp | var.methodname(primary_exp*) | new
classname(exp*)
                          // recursive expressions | non recursive
                          expressions
                               // both lists are comma separated
vardec ::= type var = exp;     // variable declaration
param ::= type var             // parameters
stmt ::= vardec |         // variable declaration
         var = exp; |     // changing the value of a previously
                          // instantiated variable
         while (exp)  stmt  |         // while loops
         break; |                     // break
         if (exp) stmt else stmt |    // if/else
         return exp; |                // return an expression
         { stmt* }                    // block
         println(exp*); |            // printing expression
                                     // exps are comma separated
         super(var); |               // Invoke parent constructor
         this.var = var; |           // refers to the current
instance
         exp;                  // gives entry to exp and includes ;

methoddef ::= type methodname(param*) stmt // method declaration
                                         //params are comma
separated
classdef ::= class classname extends classname {
              vardec*
              constructor(param*) stmt      // params are comma
separated
              methoddef*
         }
         // creates new class instance (extends classname is
     optional)
program ::== classdef* stmt*
```