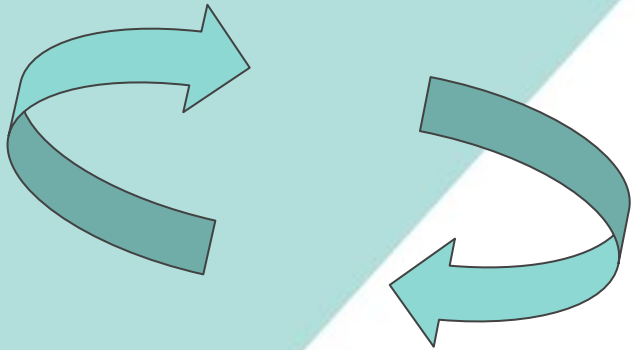
CSC 372 Translator Project



Madeline DeLeon, Savannah Rabasa,
Madison Ryan, Simi Saha

Language Design

For our project, we chose Java as our target language

Since Java is:

- Object-Oriented
- Imperative
- Compiled
- Strongly Typed
- Statically Typed
- Type Annotations

Our language must be:

- Not Object Oriented
- *We will keep Imperative
- Interpreted
- *We will keep Strongly Typed
- Dynamically Typed
- Type Inference

Language Design

What does this mean for our language?

- **Not Object Oriented** -> no objects, classes, or methods
- **Imperative** -> program specifies how a task should be done
- **Interpreted** -> programs are translated line by line
- **Strongly Typed** -> lots of type checking is done
- **Dynamically Typed** -> type checking is done at runtime
- **Type Inference** -> interpreter infers types from context

Language Design

We recognize that a language that meets all of these requirements is very similar to existing programming languages such as Python, Ruby, and JavaScript

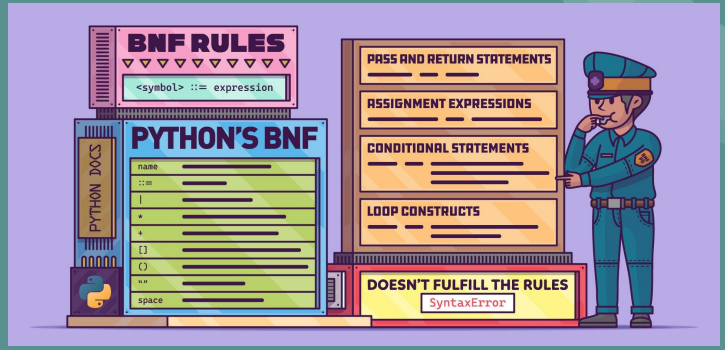
- Object-Oriented
- **Imperative**
- **Interpreted**
- **Strongly Typed**
- **Dynamically Typed**
- **Type Inference**

- Object-Oriented
- **Imperative**
- **Interpreted**
- **Strongly Typed**
- **Dynamically Typed**
- **Type Inference**

- Object-Oriented
- **Imperative**
- **Interpreted**
- Weakly Typed
- **Dynamically Typed**
- **Type Inference**

The Process

The first thing we did was write the grammar.
We started by writing a grammar for Java syntax that covered the basic requirements. Once we had that basic foundation, we were able to change up the syntax and format to match what we wanted our language to look like.



The challenge we ran into at this point was making sure that our grammar was unambiguous. We had to rework some non-terminals a few times to ensure that undesired behavior was impossible. For example, we broke <block> into <block> and <block2> so that you cannot declare a function within an if statement, loop, etc.

The Process

Here is the entirety of our grammar. A closer look can be found here: https://docs.google.com/document/d/1Ou1KlAhwrNJciJRnLzglz_gmpN0XCmkqhSR0MLGjvn0/edit?usp=sharing

```
<block> := <func_dec> <block> | <func_dec> | <block2>
<block2> := <block> <line> | <block> <loop> | <block> <cond_expr> | <loop> | <line>
            | <cond_expr>
<line> := <print>\n | <var_assign>\n | <func_call>\n | \n
<func_call> := <func>(<args>) | <func>()
<func_dec> := func <func>(<params>) { <block2> <return>}\n
            | func <func>() { <block2> <return>}\n
<return> := return | return <int> | return <bool> | return <string> | return <var>
<args> := <args>,<var> | <args>,<mult_div> | <args>,<condition> | <args>,<string>
            | <var> | <mult_div> | <condition> | <string>
<params> := <params>,<var> | <var>
<func> := (?!while|for|input|print)([a-zA-Z])+\w*
<print> := display(<condition>) | display(<mult_div>) | display(<string>) | display(<var>)
            | displayLine(<condition>) | displayLine(<mult_div>) | displayLine(<string>)
            | displayLine(<var>)
<loop> := loop(<mult_div>) {<block2>} | loop(<mult_div>,<mult_div>) {<block2>}
            | loop(<condition>) {<block2>}
<cond_expr> := if (<condition>) { <block2> }\n else { <block2> }\n
            | if (<condition>) {<block2>}\n
<condition> := <comp_expr> | <and_or>
```

```
<comp_expr> := <mult_div> > <mult_div> |  <mult_div> < <mult_div>
            | <mult_div> >= <mult_div> |  <mult_div> <= <mult_div>
            |  <mult_div> == <mult_div> | <and_or> == <and_or> | (<comp_expr>)
<and_or> := <and_or> and <not_expr> | <and_or> or <not_expr> | <not_expr>
<not_expr> := not <not_expr> | <bool_expr>
<bool_expr> := ( <condition> )  | <var> | <bool>
<mult_div> := <mult_div> * <add_sub> | <mult_div> / <add_sub>
            | <mult_div> % <add_sub> | <add_sub>
<add_sub> := <add_sub> + <int_expr> | <add_sub> - <int_expr> | <int_expr>
<int_expr> := ( <mult_div> ) | <var> | <int>
<var_assign> := <var> = <string> | <var> = inputStr() | <var> = intpuInt()
            | <var> = <mult_div> | <var> = <condition> | <var> = <func_call>
<var> = ([a-zA-Z])+\w*
<int> := ([1-9])+\d* | 0
<bool> := true |  false
<string> := “\w+”
```

The Process

The next step was to start writing the translator. We split the workload by trying to divide the grammar into sections that did not overlap. For example, subtrees of non-terminals in our syntax tree, such as the <mult_div> and <and_or> subtrees, were considered a section, and became their own java file for parsing and translating within the project.

```
<mult_div> := <mult_div> * <add_sub> | <mult_div> / <add_sub>
                | <mult_div> % <add_sub> | <add_sub>
<add_sub> := <add_sub> + <int_expr> | <add_sub> - <int_expr> | <int_expr>
<int_expr> := ( <mult_div> ) | <var> | <int>
```

```
<and_or> := <and_or> and <not_expr> | <and_or> or <not_expr> | <not_expr>
<not_expr> := not <not_expr> | <bool_expr>
<bool_expr> := ( <condition> )  | <var> | <bool>
```

The Process

Getting each individual part to work wasn't too bad, but the real challenge came when we had to combine them into one unit. At one point, we ran into problems with stack overflow and infinite recursion from too many classes calling each other.

```
<and_or> := <and_or> and <not_expr> | <and_or> or <not_expr> | <not_expr>
<not_expr> := not <not_expr> | <bool_expr>
<bool_expr> := ( <condition> )  | <var> | <bool>
```

One example of this was within the AndOr.java file. It was originally written so that the file would call Condition.java if parentheses were found when parsing, but this led to stack overflow and infinite recursion because Condition.java also calls AndOr.java, which created endless circular calls.

The Process

We also ran into difficulty parsing multi-line code blocks for <cond_expr>, <loop>, and <func_dec>. They all require multiple lines to be considered together when parsed.

```
*/
public static boolean funcHelper(String line, Scanner reader) {
    // parse header to validate
    if(!func.parseCmd(line)) {
        System.out.println(func.result);
        System.exit(status:0);
    }
    boolean inFunc = true;

    // while in func (stack not empty)
    while(reader.hasNext() && inFunc) {
        line = reader.nextLine().trim();

        if(line.trim().equals(anObject:"")) {
            func.translated += "\n";
        }
        // handles final func return
        else if(func.parseReturn(line)) {
            if(reader.nextLine().trim().equals(anObject:"}")) {
                inFunc = false;
                func.result += func.retResult;
                func.translated += func.retTranslated;
            }
            else {
                System.out.println("Failed to parse '" + func.name + "()'. Must have '}' on line after final function return.");
                System.exit(status:0);
            }
        }
        // parses & translates any loops in the func
        else if (line.contains(s:"loop(")) {
            String loopBlock = buildBlock(line, reader);
            if (loop.parseCmd(loopBlock)) {
                System.out.println(loop.result);
                func.result += loop.result;
                func.translated += loop.translated;
            }
            else {
                System.out.println(loop.result);
                System.exit(status:0);
            }
        }
        // parses & translates any if-stmts in the func
```

<func_dec> is an independent block that follows strict syntax, so we used a simple helper function that parses mostly normally but with an added return line check

```
178     public static boolean funcHelper(String line, Scanner reader) {
246         // handles if file ends before func closed
247         if(inFunc) {
248             System.out.println("Failed to parse '" + func.name + "'. Function must b
249             System.exit(status:0);
250         }
251
252         // makes sure func was complete before saving
253         if(func.endFunc()) {
254             func.addToFuncs();
255             return true;
256         }
257         else {
258             return false;
259         }
```

The Process

We also ran into difficulty parsing multi-line code blocks, <cond_expr>, <loop>, and <func_dec>. They all require multiple lines to be considered together when parsed.

Conditionals and loops must be able to nest and only be ended with a curly brace, so we made a separate helper function to put together the lines. If-else conditionals specifically also require two separate blocks, which was added as a special case.

```
public static String buildBlock(String firstLine, Scanner in) {
    try {
        String result = firstLine + "\n";
        Stack<String> stack = new Stack<>();
        stack.push(item:"{");

        // adds lines to block until balanced curly braces
        while (!stack.empty()) {
            String cur = in.nextLine().trim();
            if (cur.contains(s:"{")) {
                stack.push(item:"{");
            }
            if (cur.contains(s:"}")) {
                stack.pop();
            }

            result += cur + "\n";

            // checks for else & adds else block
            if (cur.contains(s:"}") && in.hasNext(pattern:"\\s*else\\s*.*")) {
                result += buildBlock(in.nextLine(), in);
            }
        }
        return result;
    } catch (Exception e) {
        System.out.println(x:"Failed to parse: Program does not contain enough
        System.exit(status:0);
    }
    return "";
}
```

The Process

The Translator.java file was the final step, and is the main file in this project. It runs in the console and asks for the user to input the name of the file that they would like to translate. The translator then begins its process, creating a new file in Java with the same name as the given file (ex. test1.txt → test1.java). The given file is parsed for non-terminals, and then translated into Java, the code being added to the newly created file. If there are any errors in the Boa in the given file, the translator returns an error in the console explaining what the error is.

Boa Tutorial

We decided to name our language Boa because it looks visually similar to Python, but functions differently. Here is a guide on the syntax:

- Strings, integers, and booleans are represented in the usual way

Strings: “Hello, world!”, “Enter your name: “, “Alice”, etc.
Integers: 5, 1220, -3, etc.
Booleans: true, false

- Variable assignment is done using the equals sign “=”

x = “hi”
y = 2
z = true

Boa Tutorial

- Arithmetic, comparison, and boolean operators are the same standard operators

+, -, *, /, %

>=, <=, ==, !=, >, <

and, or, not

- Print statements work like print and println in Java

display(x) -> print without newline character at the end

displayLine(x) -> print with newline character

Boa Tutorial

- If-else statements are similar to Java in that they require parentheses around the condition and curly braces around each block

```
if (<condition>) {
        …
}
else {
        …
}
```

Boa Tutorial

- Loops can either have one or two parameters. Loops with one parameter can either be a while loop, or a loop from 0 to the specified integer. Loops with two parameters are “for i in range(start, end)” type loops, but you cannot access the “i”

```
loop(i > 0) {
        …// runs while the condition is true
}
```

```
loop(2) {
        …// will run twice
}
```

```
loop(0,10) {
        …// will run ten times
}
```

Boa Tutorial

- Function definitions require the keyword “func”, as well as curly braces around the block

```
func double(x) {
    return x*2
}
```

- Function calls are the standard syntax where you write the name of the function, and then insert your arguments in the parentheses

```
x = double(4)
```

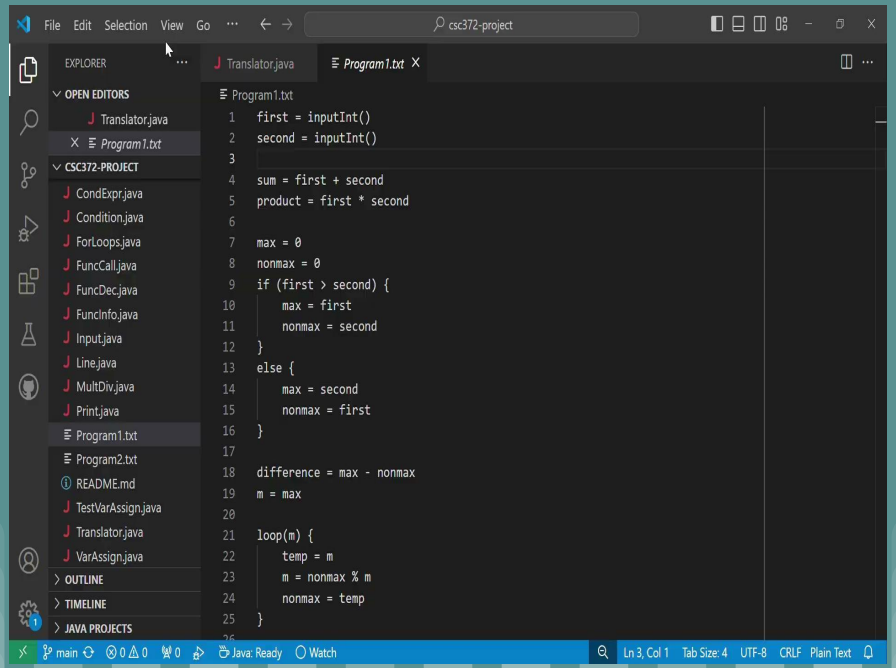
Translating a file

Once you have written a program in Boa, you can use our Translator to convert it to Java.

First, put the file you want to translate into the same directory as Translator.java. Then, run Translator.java; In the output terminal, it will prompt you to enter the name of the file you want to translate. Type in the name of your program (including the file extension) and press enter. The parsing process will be printed to the terminal, and a Java file with the same name as your original program will be created. If your program has no errors, the new file will be created successfully. If the parsing and translating fails, an error message(s) explaining what went wrong will be printed to the terminal.

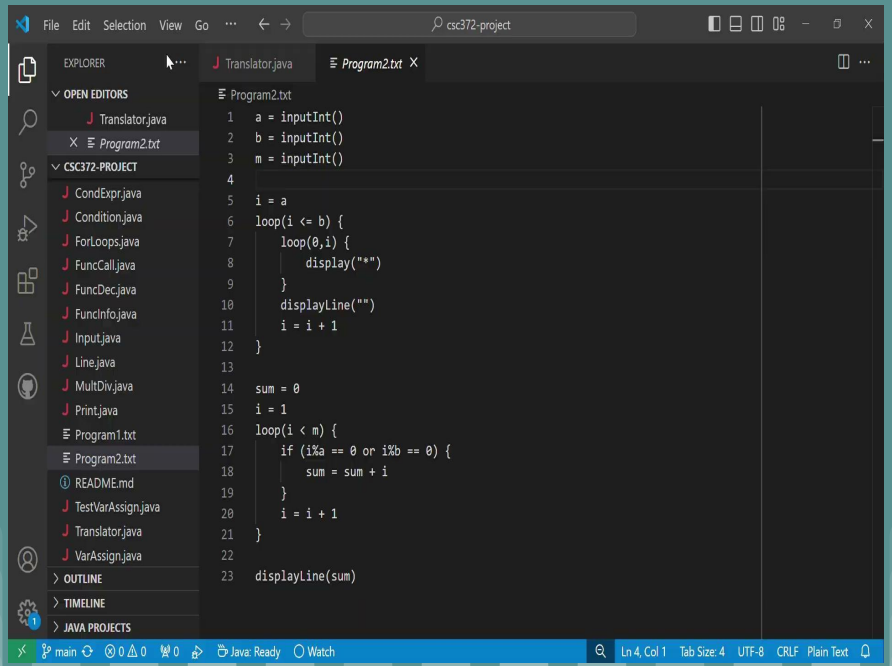
Translating a file: Examples

The video below shows the Translation of Program1.txt, one of the programs written in our language by another group



Translating a file: Examples

The video below shows the Translation of Program2.txt, one of the programs written in our language by another group

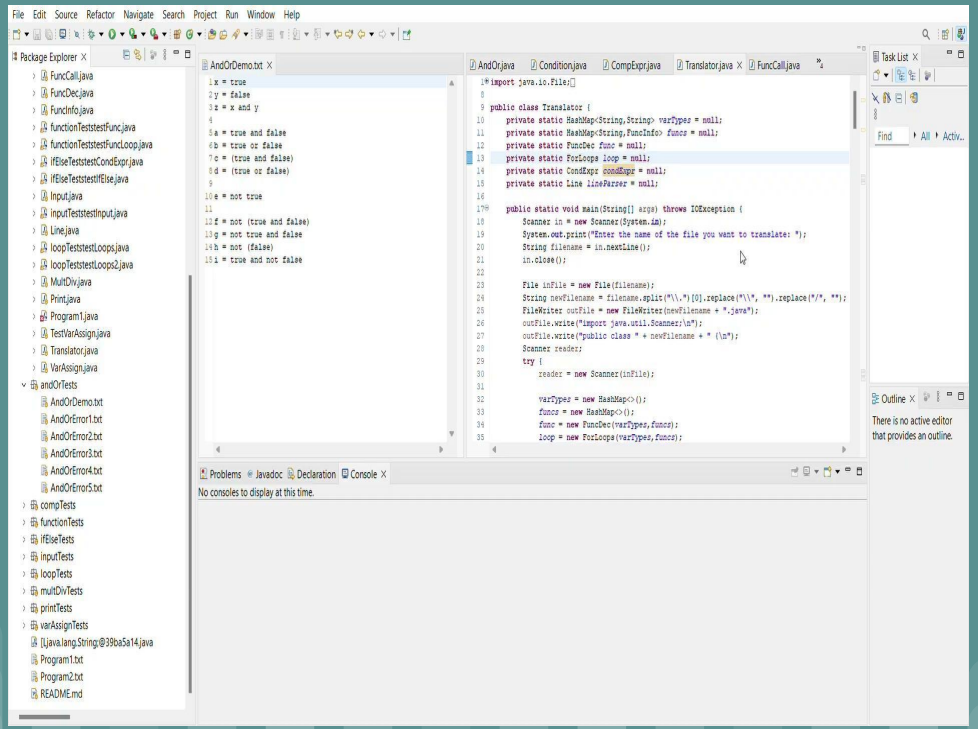


Sample Programs

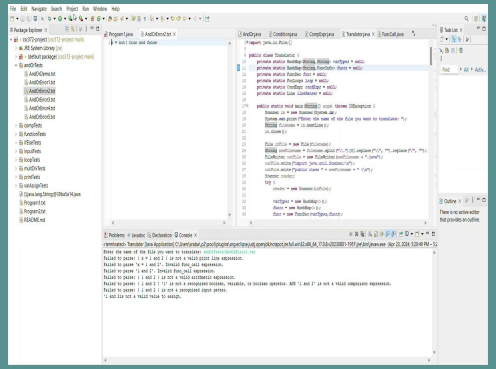
We wrote several sample programs for each separate class of our translator. In the next few slides, we will demonstrate some of these programs with a video. However, since we wrote so many - including the files we wrote for showcasing error messaging - we won’t show every single sample program. Those programs can be translated as explained before.

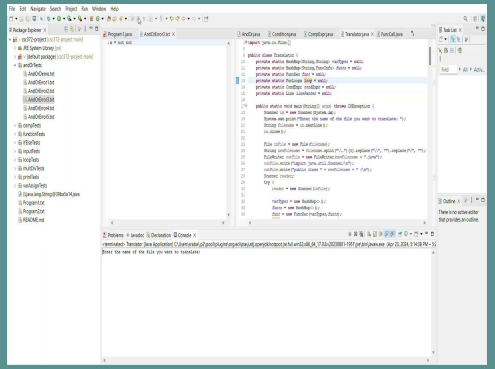
Sample Programs: AndOr

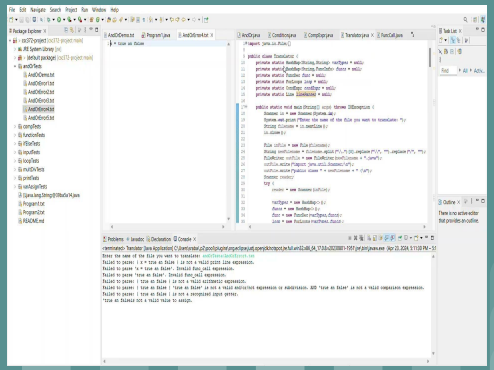
This video showcases the proper functionality of the AndOr.java class, which includes and, or, and not boolean expressions



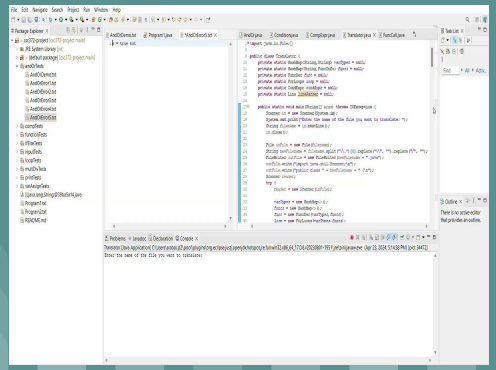
Sample Programs: AndOr Errors





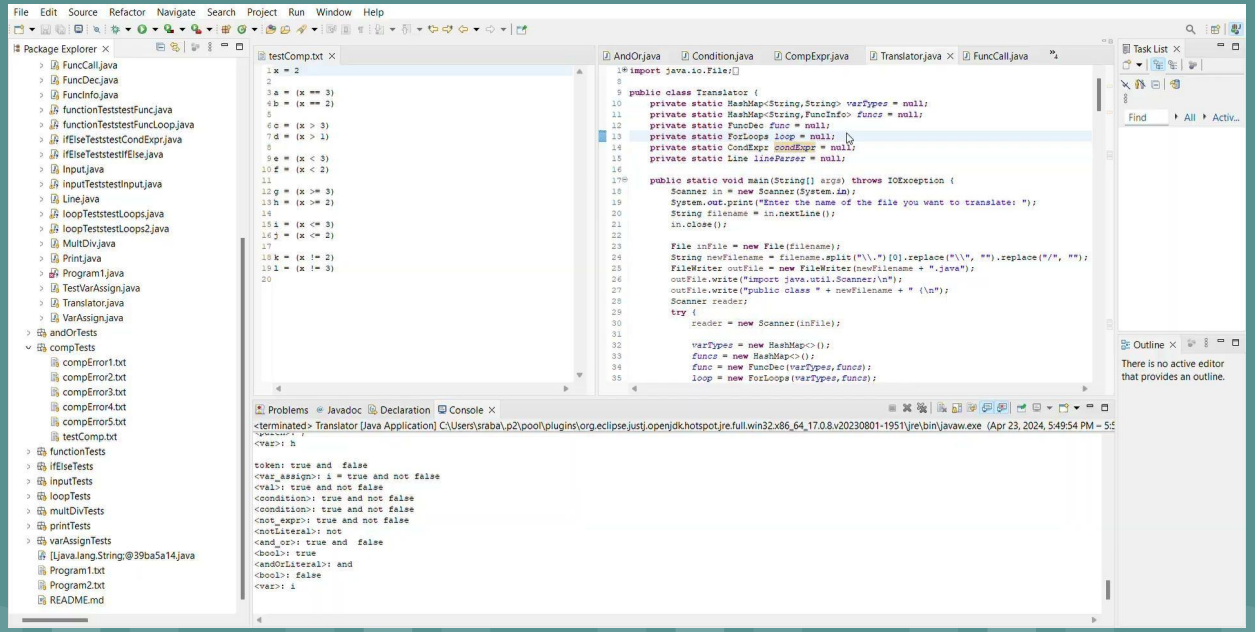


These videos show the error messages when parsing incorrectly written and/or/not statements



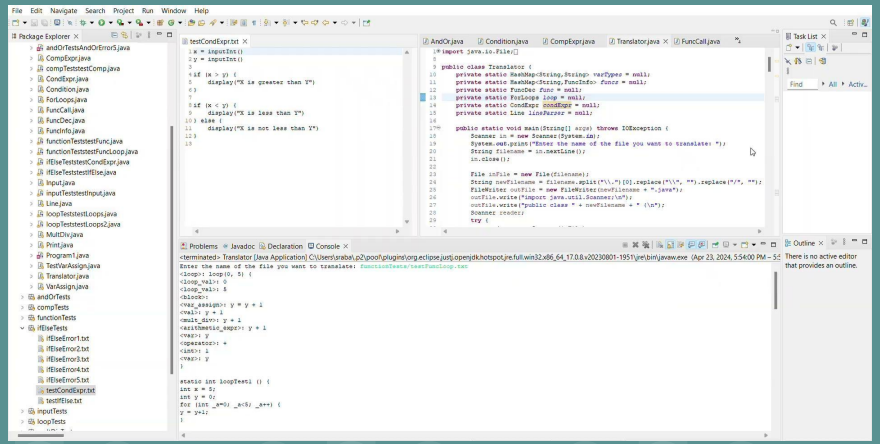
Sample Programs: Comp Tutorial

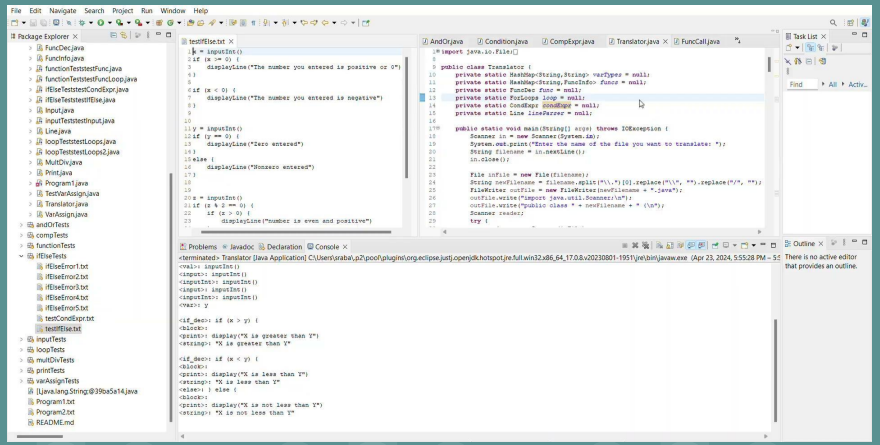
This video showcases the proper functionality of the CompExpr.java class, which includes comparison expressions using >, <, ==, !=, >=, and <=



Sample Programs: If/Else

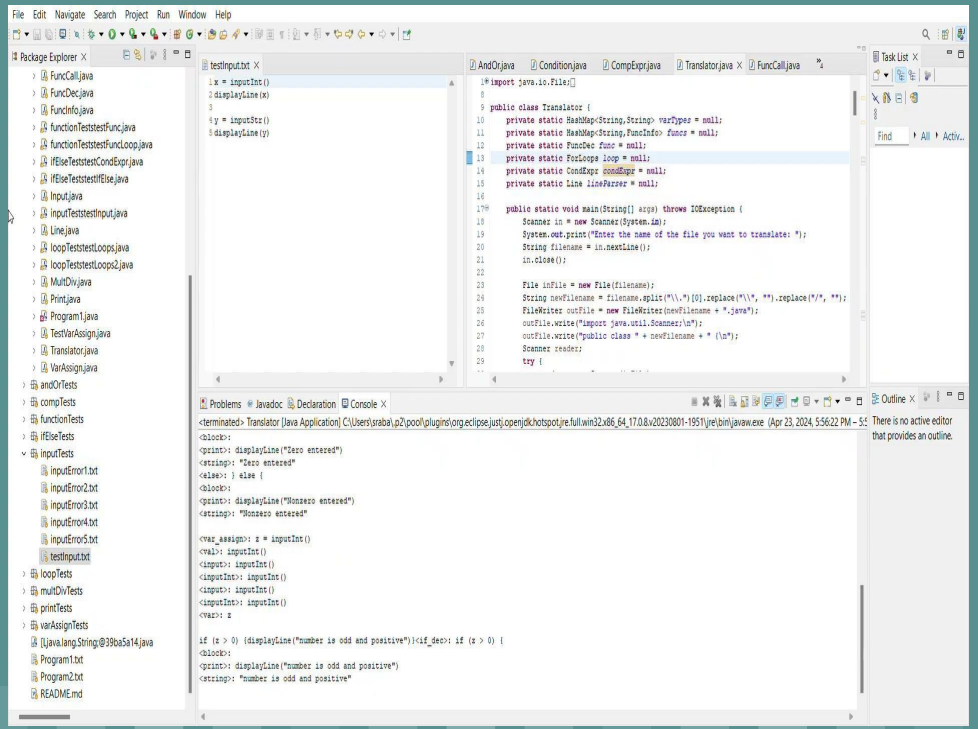
These videos showcase the proper functionality of the CondExpr.java class, which parses if/else statements, handles nesting, and parses the inner block(s)





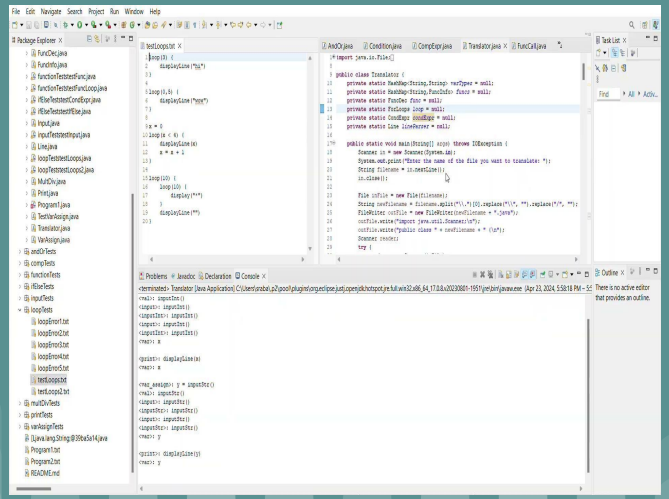
Sample Programs: Input

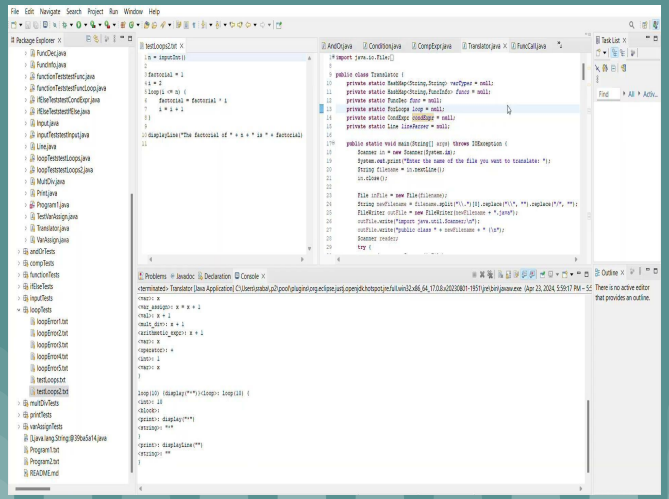
This video showcases the proper functionality of the Input.java class, which parses input statements in Boa



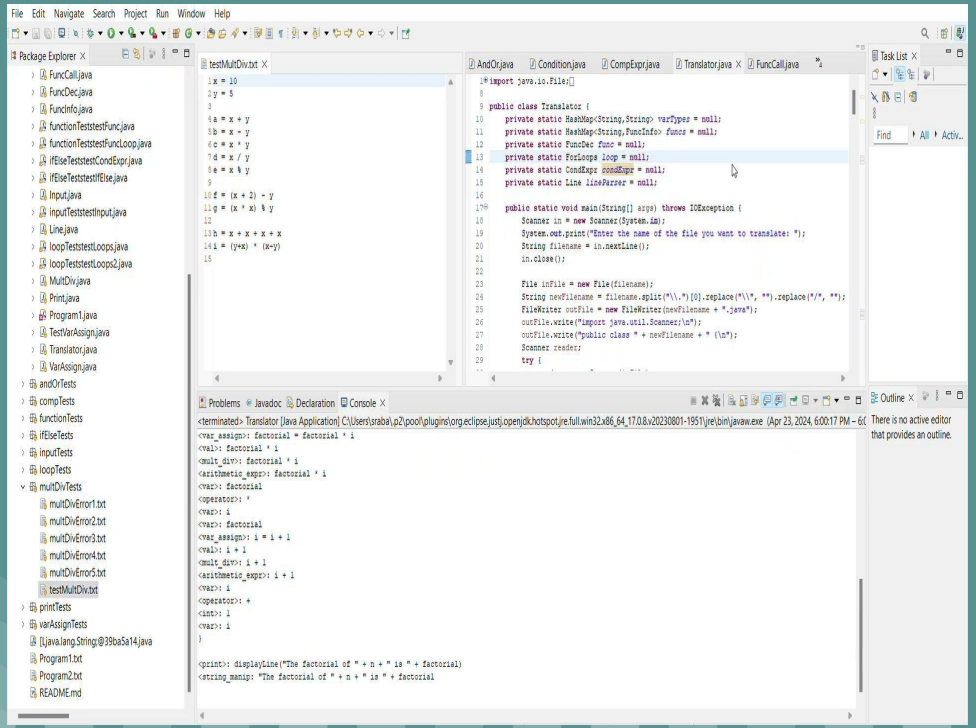
Sample Programs: Loops

These videos showcase the proper functionality of the ForLoops.java class, which parses loops in Boa and the contents nested within the loop.





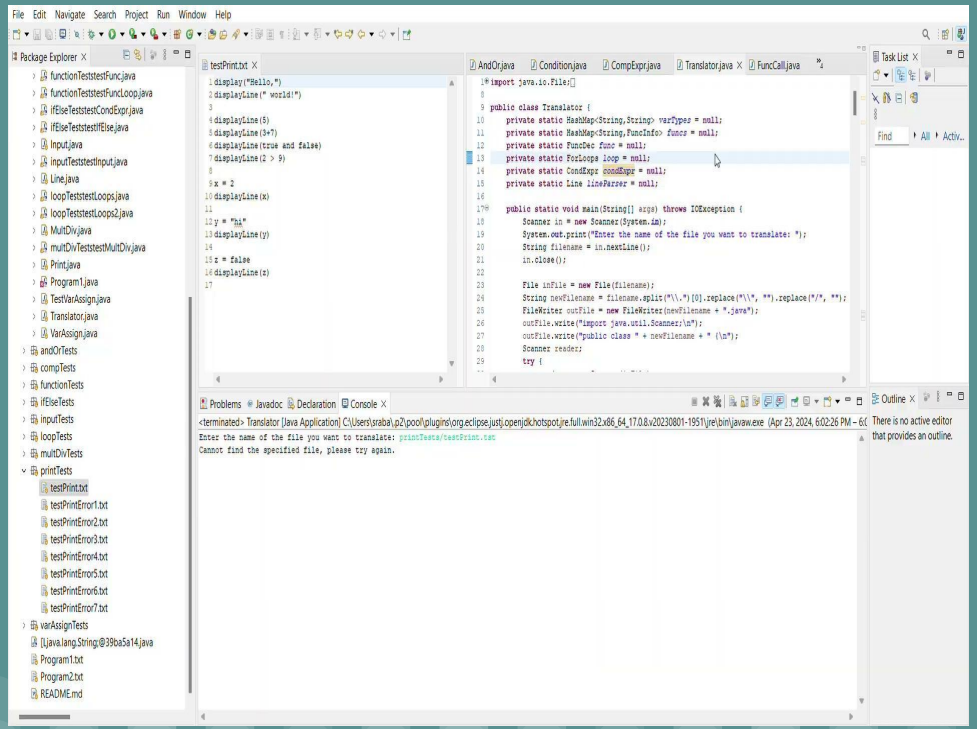
Sample Programs: Mult/Div



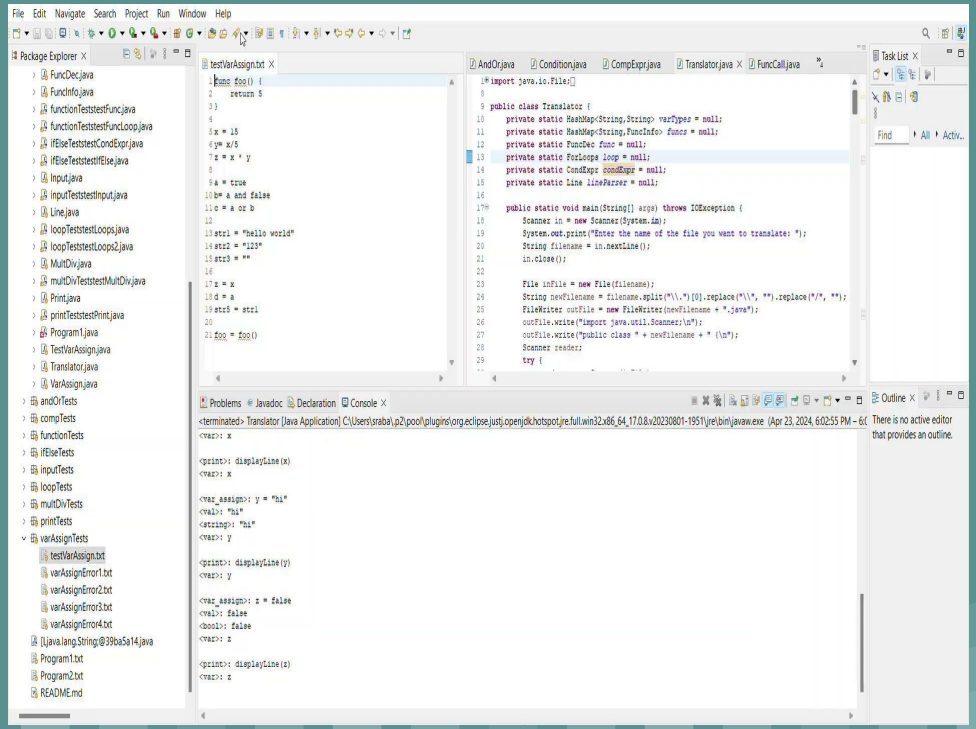
This video showcases the proper functionality of the MultDiv.java class, which includes arithmetic expressions such as +, -, *, /, %

Sample Programs: Print

This video showcases the proper functionality of the Print.java class, which parses Boa print statements, display() and displayLine()



Sample Programs: VarAssign



This video showcases the proper functionality of the VarAssign.java class, which shows the assignment of function calls, booleans, ints, strings, etc. to variables

Additional Features

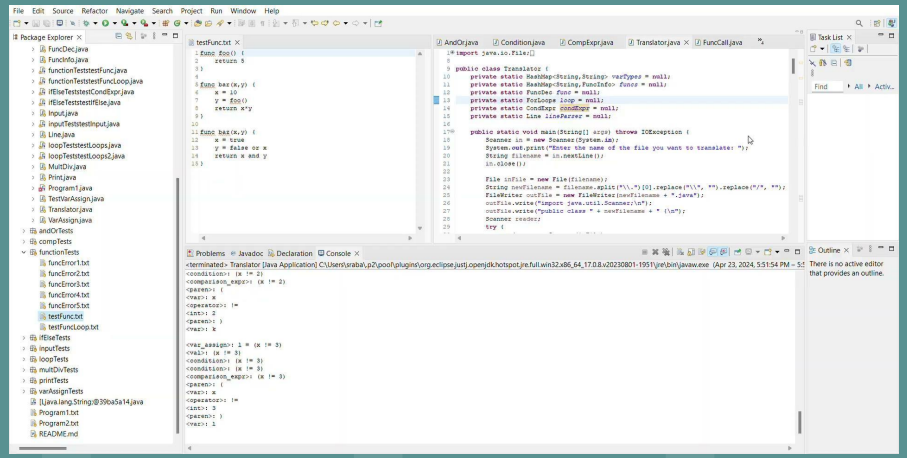
One additional feature we added to our language was function declarations and function calls. Here is what they look like within our grammar:

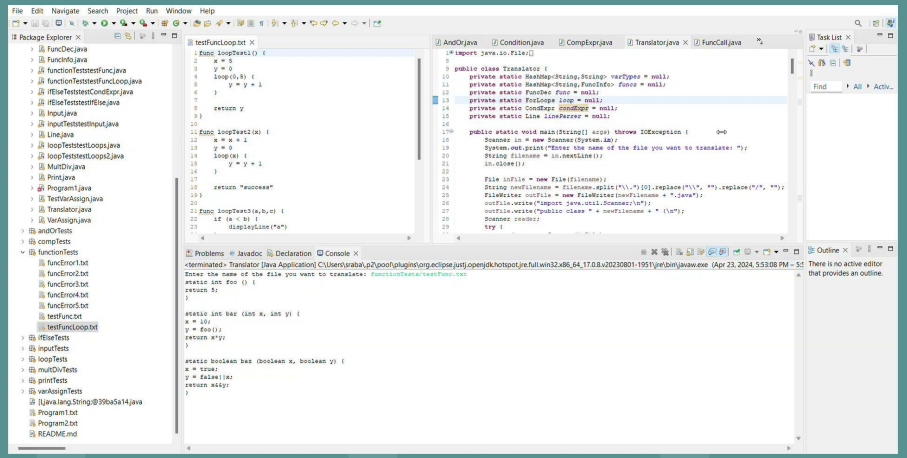
```
<func_call> := <func>(<args>) | <func>()
<func_dec> := func <func>(<params>) { <block2> <return>}\n
                | func <func>() { <block2> <return>}\n
<return> := return | return <int> | return <bool> | return <string> | return <var>
<args> := <args>,<var> | <args>,<mult_div> | <args>,<condition> | <args>,<string>
                | <var> | <mult_div> | <condition> | <string>
<params> := <params>,<var> | <var>
<func> := (?!loop|if|input|inputInt|display|displayLine)([a-zA-Z])+\w*
```

This means that within your Boa file, you can write a function and call it, as well as assigning its return value to a variable. An example of this can be seen in the videos on the next slide, which highlight the use of a test file with a function declaration and function call in Boa.

Sample Programs: Functions

These videos showcase the proper functionality of the FuncDec.java and FuncCall.java classes, which include function declarations and function calls





Additional Features

Another feature we added to our translator was showing the step-by-step parsing process in the console. When translating a file, the console shows each line's parsing process and breakdown in detail.

Some examples of the parsing shown when translating Program1.txt

```
<var_assign>: second = inputInt()
<val>: inputInt()
<input>: inputInt()
<inputInt>: inputInt()
<input>: inputInt()
<inputInt>: inputInt()
<var>: second

<var_assign>: sum = first + second
<val>: first + second
<mult_div>: first + second
<arithmetic_expr>: first + second
<var>: first
<operator>: +
<var>: second
<var>: sum
```

```
<if_dec>: if (first > second) {
<block>:
<var_assign>: max = first
<val>: first
<var>: first
<var>: max
<var_assign>: nonmax = second
<val>: second
<var>: second
<var>: nonmax
<else>: } else {
<block>:
<var_assign>: max = second
<val>: second
<var>: second
<var>: max
<var_assign>: nonmax = first
<val>: first
<var>: first
<var>: nonmax
```

Additional Features

We also included more detailed error messages into our translator. If a line is reached that cannot be parsed by the Boa grammar, the console then prints the error message, stating why the given line cannot be translated.

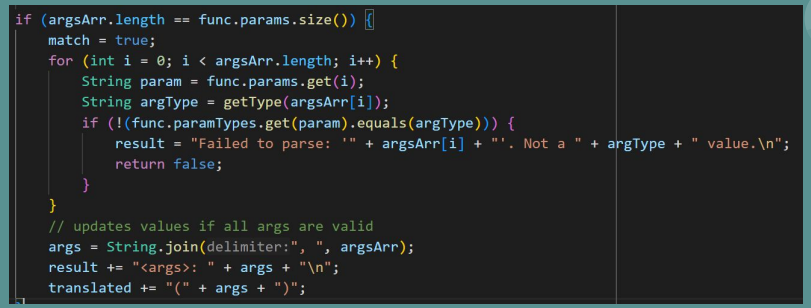
Some examples from our Print and FuncDec error programs:

```
Enter the name of the file you want to translate: printTests/testPrintError7.txt
Failed to parse: { 1a } is not a recognized printable value.
```

```
Enter the name of the file you want to translate: functionTests/funcError2.txt
Failed to parse 'foo'. Function must be closed with '}'.
```

Additional Features

The last additional feature we include was a type checking system. Our parser infers types from regex pattern matching for primitive values, and in <var_assign> maps those types to variables. When values and variables are used in expressions, their types are checked to ensure they match the grammar. For example, <mult_div> is an int expression so only int values or variables can be used with it. The only exception is parameters, whose values must be inferred based on how they are used. For example, if a parameter is used in a <mult_div> expression then it is typed as int.



```
if (argsArr.length == func.params.size()) {
    match = true;
    for (int i = 0; i < argsArr.length; i++) {
        String param = func.params.get(i);
        String argType = getType(argsArr[i]);
        if (!(func.paramTypes.get(param).equals(argType))) {
            result = "Failed to parse: '" + argsArr[i] + "'. Not a " + argType + " value.\n";
            return false;
        }
    }
    // updates values if all args are valid
    args = String.join(delimiter:", ", argsArr);
    result += "<args>: " + args + "\n";
    translated += "(" + args + ")";
```

Potential Changes in Continued Development

Something useful to add to our translator if we were to continue to develop it further would be to translate the file into Java with better formatting. The way our translator currently works, everything is aligned all the way to the left of the file. Including proper indentation within brackets would make it easier to read after the translation process without needing any extra steps/work.

```
import java.util.Scanner;
public class Program2 {
Run | Debug
public static void main(String[] args) {
int a = new Scanner(System.in).nextInt();;
int b = new Scanner(System.in).nextInt();;
int m = new Scanner(System.in).nextInt();;

int i = a;
while (i<=b) {
for (int _b=0; _b<i; _b++) {
System.out.print(s:"*");
}
System.out.println(x:"");
i = i+1;
}

int sum = 0;
i = 1;
while (i<m) {
if (i%a==0||i%b==0) {
sum = sum+i;
}
i = i+1;
}
```

Potential Changes in Continued Development

Currently, our language is very syntactically strict. This is especially true for code blocks like with if-blocks, loops, and especially functions. Very specific formatting is currently required for spacing, bracket placement, etc. If we were to continue developing our language and translator, incorporating more flexibility into its structure would be helpful.

We would also want to implement data structures like lists, arrays and trees to handle multiple items and introduce their functionality.

It would also be useful to create built in functions for common operations like sorting or mapping, or something like Math class for handling things like square root and absolute value.

Thank you!