JUDO

DANILO HIDALGO AND CHRISTIAN TREJO

OVERVIEW

- Judo is statically typed and uses type inference
- Created by Danilo Hidalgo and Christian Trejo in 2021 and maintained by CSC372
- Similar to Python and pseudocode in syntax.
- Judo = “J” for code translated to Java + “udo” for Pseudocode syntax
- Voted ”best language ever created” since 2021.

JUDO PHILOSOPHY

- Multi-paradigm: Imperative, sequential
- Type safety
  - Strongly typed
  - No coercion
- Efficiency
  - Translates to Java code, which can be compiled and run

WHY JUDO?

- Before Judo, there was a gap for an easily understood, strongly typed language that combines several syntactically easy aspects from other language.

"Judo is a programming language focused on the user experience that will ensure quality coding habits (i.e., indentation and variable usage).

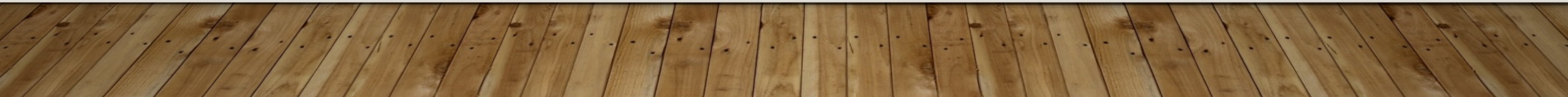
-Judo Creators

- Although Judo is relatively new, we expect it to take off and be used as an introductory programming language.

INFLUENCES

- Judo has a combination of Python and pseudocode syntax
- Types are the typical integers, strings, and arrays
- Type checking inspired by Java

SYNTAX AND SEMANTICS



”HELLO WORLD”

Things to note:

- Don’t need to declare a main()
- **Whitespace indentation is crucial**
- Statements cannot have a trailing semicolon
- *outln* will print the outline with a trailing newline
- Comments start with ?

Judo Code

```
? No trailing newline
out("Hello World")

? Trailing newline
outln("Hello World")
```

BASIC TYPES

Judo allows three types: integers, strings, and Booleans.

The following grammar was used to define Integers, Strings, and Booleans:

<integer> ::= <digit> | <integer><digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<string_literal> ::= “<string>”

<string> ::= <char><string> | <char>

<char> ::= <charLetter>|<digit>| @|#|$|%|^|&|*|(|)|-|=|[|]|{|}|\\|||'|;|:|<|>|,|.|?|/|`|~|"

<bool> ::= T | F

VARIABLE ASSIGNMENT AND REASSIGNMENT

The following grammar was used to define variable assignment:

<vars_assignment> ::= let <var> = <value>

The following grammar was used to define variable reassignment:

<var_reassignment> ::= <var> = <value>

where:

<value> ::= <var> | <integer> | <string_literal> | <ray> | <bool> | <or_expr>

VARIABLE ASSIGNMENT AND REASSIGNMENT

To declare a variable, use the keyword *let*.

Available types: integer, string, boolean, int array, string array and boolean array.

To reassign a variable, the new value must be of the same type.

```
? Variable Assignment
let a = 10                ? Integer
let b = "apple pie"       ? String literal
let c = i{100}            ? int array of length 100
let d = s{55}             ? string array of length 55
let e = T                 ? Boolean value True
let f = F                 ? Boolean value False
let g = T or F            ? Boolean value T
let h = T and F           ? Boolean value F
let i = not T             ? Boolean value F

? Variable Reassignment
a = 15
b = "asdf"
```

INTEGER EXPRESSIONS

Precedence (lowest to highest): + -, mod * /, - (unary), ( )

- The following grammar was used to define variable assignment:

<expr> ::= <expr> + <mmd_expr> | <expr> - <mmd_expr> | <mmd_expr>

<mmd_expr> ::= <mmd_expr>*<root> | <mmd_expr>/<root> |

<mmd_expr> mod <root> | <root>

<root> ::= <integer> | <var> | (<expr>) | -<expr>

VALID MATH EXPRESSIONS

- Valid math expressions in Judo include variables, addition, subtraction, multiplication, division, modulus, parentheses, integers, and variables.
- Whitespace is not an issue within the math expressions either.
- Here are a few valid expressions:
  - X
  - -1
  - 1
  - -(1)
  - 100
  - 1+1
  - 1+-1
  - 1 - 1
  - 1 * -1
  - 1 / 1
  - 1/1/x/1
  - 100 mod 1000
  - 10 * (1 / (val * 3))
  - (8 - 1 + 3) * 6 - ((3 +y) * 2)

INVALID MATH EXPRESSIONS

- The Judo translator will catch invalid math expressions such as incomplete expressions using +, -, *, /, and mod.
- Also, unmatched parentheses will be caught too
- Here are a few examples of invalid math expressions:
  - +
  - 1++1
  - (1/5
  - 100.1 * 1, 90 */ 10
  - 100 mod (2 * (4/10)

BOOLEANS AND BOOLEAN EXPRESSIONS

- Booleans: T is True and F is False
- Precedence (lowest) or, and, not (highest)
- The following grammar was used to define Booleans and Boolean expressions:

<bool> ::= T|F

<or_expr> ::= <or_expr> or <and_expr> | <and_expr>

<and_expr> ::= <and_expr> and <not_expr> | <not_expr>

<not_expr> ::= not <bool_root> | <bool_root> | <comparison>

<bool_root> ::= <bool> | (<or_expr>)

COMPARISON EXPRESSIONS

- Judo allows for !=, ==, <, <=, >, and >=
- The following grammar was used to define comparison expressions:

<comparison> ::= <or_expr> != <or_expr> | <or_expr> == <or_expr>
| <or_expr> < <or_expr> | <or_expr> <= <or_expr>
| <or_expr> > <or_expr> | <or_expr> >= <or_expr>

VALID BOOLEAN AND COMPARISON EXPRESSIONS

- Variables, integers, Booleans and comparisons can be in a Boolean expressions
- Here are a few valid expressions
  - T
  - F
  - T and F
  - T or F
  - not T
  - not (not T)
  - 1 == 1
  - 1 == x
  - 1 <= 99
  - 1 >= 10
  - 1 < 10
  - 1 > 10
  - 1 != 10
  - (1 != 10)
  - x or (1 != 10)
  - x and y and z
  - (x + y * 2) == z
  - (x and (z == 10)) and (y != 100 or l or 5 < x) or (T)

INVALID BOOLEAN AND COMPARISON EXPRESSIONS

- The Judo translator will catch invalid Boolean expressions such as invalid comparison operators and comparisons between different types
- Here are a few invalid expressions:
  - 1 ===1
  - 1 <== 1
  - 1 <== 1 1
  - 1 && 1
  - T and
  - T and ()
  - (1 + 1 == 2))
  - not 1
  - 1 = 10
  - (x and (z == 10)) and () or (T)

CONDITIONALS

- Judo includes if, if-else, if-elf-else statements
- If a conditional statement is true, the statement(s) to be run must be **indented**
- The following grammar was used to define comparison expressions:

```
<conditional> ::= <if_statement> |
                  <if_statement>
                  <elf_statement>
                  <else_statement>

<if_statement> ::= if <bool_expr>:
                       <statement>
```

```
<elf_statement> ::= elf <bool_expr>:
                        <statement>
                    <elf_statement>
                    | <empty>

<else_statement> ::= else:
                         <statement>
```

CONDITIONALS AND CONDITIONAL OPERATORS

- Example of using the if-elf-else conditional statements.
- Notice that each conditional statement is followed by a colon and that the "else if" is actually "elf"
- Judo Code outputs:

  "3 is greater than 2!"

```
let x = 3

if x < 2:
    out(x)
    outln(" is less than 2!")
elf x == 2:
    out(x)
    outln(" is equal to 2!")
else:
    out(x)
    outln(" is greater than 2!")
```

CONDITIONALS – “hallpass”

- hallpass is a reserved keyword that will allow for a sort of empty conditional block to exist where no operation takes place.
- hallpass is similar to Python’s pass keyword.
- Example Judo Code will output nothing.

```
let y = 50
if y == 50:
    hallpass
else:
    outln("Entered else statement")
```

LOOPS

- Judo includes for-loops and our version of while-loops
- The following grammar was used to define loop statements:

<loops> ::= for <var> in <range>[<integer>]:
<statement> |
loop <or_expr>:
<statement>
<range> ::= <integer> .. <integer>

LOOPS – FOR LOOPS

- The range in Judo's for-loops are beginning value inclusive, and end value exclusive. For example, the code on the right will print all values from 1 to 9 but not 10.

```
for i in 1..10:
    out(i)
    out(" ")
outln()
```

LOOPS – FOR LOOPS

- Also, for-loops allow for the increment value for the loop iterator variable to be set. If this value is not included in the for-loop statement, then the increment value is defaulted to 1.
- The increment value is written in square brackets [ ] directly following the upper value of the range.
- This code will print: 1 3 5 7 9

```
for i in 1..10[2]:
    out(i)
    out(" ")
outln()
```

LOOPS – LOOP

- loop is another type of loop that is similar to a while-loop in other languages.
- A conditional statement must be given following the loop keyword.
- The statement(s) indented under the loop statement will be run.

```
let a = 10
loop a > 0:
    out(a)
    out(" ")
    a = a - 1
outln()
```

PRINT STATEMENTS

- The following grammar was used to define print statements:

  out(<print_argument>) Note: No newline

  outln(<print_argument>) Note: Printed with trailing \n

  <print_argument> ::= <digit> | <integer> | <string_literal> | <var>

- Print statements allow for integers, strings, and variables to be printed.
- out() will not print a trailing newline
- outln() will print a trailing newline

PRINT STATEMENTS

- Examples of printing a string, an integer, a string literal, and an element of an array at a given index.
- Output:

  red apple
  1234
  abcdefghijklmnopqrstuvwxyz
  34

```
let x1 = "red apple"
let x2 = 1234
let a1 = [12,34,56,78]
outln(x1)
outln(x2)
outln("abcdefghijklmnopqrstuvwxyz")
outln(a1[1])
```

ARGUMENTS

- The variable argos contains the program arguments.
- To access the arguments, use argos[index]
- For example, the Judo code to the right sets x, y, and m to the values in the argument array at indices 0, 1 and 2 respectively.

```
let x = argos[0]
let y = argos[1]
let m = argos[2]
```

ARRAYS

- Arrays in Judo can contain booleans, strings, and integers.
- Array types must be homogeneous (arrays hold one type)
- The following grammar was used to define arrays:

<ray>::= [<int_list>] | [<string_list>] | [<bool_list>] | b{<expr>} | i{<expr>} | s{<expr>}

<int_list> ::= <integer>,<int_list> | <var>,<int_list> | <integer> | <var>

<string_list> ::= <string_literal>,<string_list> | <var>,<string_list> |
<string_literal> | <var>

<bool_list> ::= <bool>,<bool_list> | <var>,<bool_list> | <bool> | <var>

ARRAY ACCESSING AND ASSIGNING

- The following grammar was used to define accessing and assigning indices in arrays:

<ray_index> ::= <var>[<expr>]

<ray_index_assign> ::= <ray_index_access> = <value>

ARRAY - SIZED INITIALIZATION

- To initialize an array of a given length, you must specify its type (i.e., b for boolean, i for integer, or s for string) and its size.
- Example 1 instantiates a string array of length 3.
- Example 2 instantiates an int array of length n (n=15).

Example 1:

```
let array = s{3}
array[0] = "cat"
array[1] = "dog"
array[2] = "moose"
for i in array:
    out(i)
    out(" ")
outln()
```

Example 2:

```
let n = 15
let array1 = i{n}
for i in 0..n:
    array1[i] = n-i
for i in array1:
    out(i)
    out(" ")
outln()
```

Output from example 1:

```
cat dog moose
```

Output from example 2:

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

ARRAYS ITERATION

- These example demonstrate how to instantiate an array and iterate through its values.
- The iterator is the actual value at the current index.

```
let array = [10,6,1,9]
for i in array:
    out(i)
    out(" ")
outln()

let array2 = ["Apple", "Goldfish", "Quesadilla", "Burrito", "Milk"]
for k in array2:
    out(k)
    out(" ")
outln()
```

Output:

```
10 6 1 9
Apple Goldfish Quesadilla Burrito Milk
```

ARRAY ASSIGNMENT

- This example shows how to assign a value at an index.

```
let array2 = ["Apple", "Goldfish", "Quesadilla", "Burrito", "Milk"]
array2[3] = "asdf"
for k in array2:
    out(k)
    out(" ")
outln()
```

Output:

```
Apple Goldfish Quesadilla asdf Milk
```

GENERAL STATEMENTS

In general, there are three types of statements:

1. Value transfers (assignment, reassignment, array mutation, etc).
2. Control flow statements (conditionals and loops)
3. Print statements (out(...) and outln(...))

HERE ARE BIGGER PROGRAMS AND HOW THEY TRANSLATE INTO JAVA

- The following slides will show working programs where each program's purpose is listed in the comments section.
- Judo Code will be on the left and the equivalent translated Java code will be on the right.
- Many of the aforementioned features of our language are included in the programs.

Judo Code

```
? Project 2 - Design a language
? Group Members: Danilo Hidalgo and Christian Trejo
? File: Program 1
? Purpose: Calculate and print number of multiples x
?          and y in the range 1 to m.
? Arguments: Three integers x, y, and m

let x = argos[0]
let y = argos[1]
let m = argos[2]

let numMultiples = 0

for i in 1..m[1]:
    if i mod x == 0:
        numMultiples = numMultiples + 1

    if i mod y == 0:
        numMultiples = numMultiples + 1

outln(numMultiples)
```

Generated Java

```
// GENERATED: 2021-11-18T14:06:50.174327
public class TestProgram1 {
public static void main(String[] argos) {
    int x = Integer.parseInt(argos[0]);
    int y = Integer.parseInt(argos[1]);
    int m = Integer.parseInt(argos[2]);
    int numMultiples = 0;
    for(int i = 1; i < m; i += 1) {
        if (i%x == 0) {
        numMultiples = numMultiples + 1;
        }
        if (i%y == 0) {
        numMultiples = numMultiples + 1;
        }
        }
    System.out.println(numMultiples);
    }
}
```

Judo Code

```
? Project 2 - Design a language
? Group Members: Danilo Hidalgo and Christian Trejo
? File: Program2.txt
? Purpose: Print out "prime" if x is prime and
?          "not prime" if it is not.
? Arguments: Integer x

let x = argos[0]
let i = 2
let flag = F
loop i < x:
    if x mod i == 0:
        outln("not prime")
        flag = T
        i = x
    else:
        i = i + 1
if not flag:
    outln("prime")
```

Generated Java

```
// GENERATED: 2021-11-18T14:06:50.693897
public class TestProgram2 {
public static void main(String[] argos) {
    int x = Integer.parseInt(argos[0]);
    int i = 2;
    boolean  flag =  false ;
    while(i < x) {
        if (x%i == 0) {
        System.out.println("not prime");
        flag =  true ;
        i = x;
        }
        else {
            i = i + 1;
            }
        }
    if ( ! flag) {
    System.out.println("prime");
    }
    }
}
```

Judo Code

```
? Project 2 - Design a language
? Group Members: Danilo Hidalgo and Christian Trejo
? File: Program3.txt
? Purpose: Print the first n numbers in the Fibonacci sequence
? Arguments: Integer n
? Parsing Error:

let x = argos[0]              ? Number of desired Fibonacci number
let i = 0                     ? Count of nums printed
let current_num = 0           ? Current Fib number
let last_num = 0              ? Previous Fib number
let temp = 0                  ? temporary

loop i < x:
    if i == 0:
        outln(current_num)
        last_num = current_num
        current_num = 1
    else:
        outln(current_num)
        temp = current_num
        current_num = current_num + last_num
        last_num = temp
    i = i + 1
```

Generated Java

```
// GENERATED: 2021-11-18T15:17:06.106586
public class TestProgram3 {
public static void main(String[] argos) {
    int x = Integer.parseInt(argos[0]);
    int i = 0;
    int current_num = 0;
    int last_num = 0;
    int temp = 0;
    while(i < x) {
        if (i == 0) {
        System.out.println(current_num);
        last_num = current_num;
        current_num = 1;
        }
        else {
            System.out.println(current_num);
            temp = current_num;
            current_num = current_num + last_num;
            last_num = temp;
            }
        i = i + 1;
        }
    }
}
```

PARSING ERROR MESSAGES: PROGRAM 3

```
? Judo Code:
let bool1 = T
let bool2 = T or F and bool1
if bool1 or bool2 and F          ? Our language requires colons after if-statements
    out("This won't print")
```

Results in this error:

```
Exception in thread "main"
InvalidStatementError at line 3: `if bool1 or bool2 and F`
        at parser.Parser.parseBlock(Parser.java:426)
        at parser.Parser.parseFull(Parser.java:220)
        at Translator.main(Translator.java:30)
```

PARSING ERROR MESSAGES: PROGRAM 4

```
let one = 1
let two = 2
let three = 3
let four = 5
let intArr = [one, two, three, four, 5]
intArr[2] = "3"  ? fails because intArr is known to contain ints.
```

Results in this error:

```
Exception in thread "main"
TypeError: Expected expression `"3"` to be of type int
        at parser.Parser.validateByScalarType(Parser.java:276)
        at parser.Parser.handleRayIndexAssignment(Parser.java:814)
        at parser.Parser.parseBlock(Parser.java:424)
        at parser.Parser.parseFull(Parser.java:220)
        at Translator.main(Translator.java:30)
```

PARSING ERROR MESSAGES: PROGRAM 5

```
for i in 1 + 1 * 100..100 mod 100 + 100[1+2+3+4]:  ? This validates!?!
    let i = 10  ? This does not validate because i is already defined in this scope
    hallpass
```

Results in this error:

```
Exception in thread "main"
VariableError at line 2:Variable `i` is already defined in this scope.
        at parser.Parser.handleAssignment(Parser.java:455)
        at parser.Parser.parseBlock(Parser.java:376)
        at parser.Parser.parseBlock(Parser.java:408)
        at parser.Parser.parseFull(Parser.java:220)
        at Translator.main(Translator.java:30)
```

HOW TO RUN JUDO

- You can run the Judo translator using the provided pre-compiled JAR file in our latest release. To run a jar, use java -jar path/to/jar.jar
- On success, Judo will emit a compiled Java file and tell you where it put it. You can then run this file like a normal Java file.
- If you have Bash, you can use our convenience script which combines all these steps into one.
- More info on usage, including examples, can be seen at the release in Github.

CHALLENGES FACED WHEN WRITING THE TRANSLATOR AND HOW WE SOLVED THEM

- We implemented an algorithm that essentially uses DFS to find a parse tree that validates math and boolean expressions of arbitrary complexity. Taking the parse tree algorithm from class and putting it into Java was intellectually quite challenging, and what helped us was manually building parse trees on paper, analyzing how our brains were actually doing it, and what that meant in code.
- The second major challenge came when we incorporated a typing system, and soon realized that we would have to check variables in those arbitrary expressions. Our solution was to mix syntactic parsing with semantic checking by checking variables at a grammar level.

FUTURE LANGUAGE DESIGN CHOICES

- One of the first things we would do is implement a more robust type system. Our types in the project were just a Java Enum, but a real language should allow for arbitrarily compound/nested types, and this is challenging under our typing system.
- If we had more time, we would have implemented functions, and if we had *a lot* of time, it would be even better to have *first-class* functions, as it's a feature that is sorely missed in Java. Languages like Go and Rust show that first class functions are 100% possible in a strongly typed language.

CONCLUSION

Thank you for your time.

- To check out our project, go to: https://github.com/DaniloHP/csc372project2