Contents

1. Overview of J-Pie

2. Motivation

3. Lexical conventions
   3.1. Comments
   3.2. Identifiers (Names)
   3.3. Type Specifiers and Operators
   3.4. Keywords

4. Expressions
   4.1. Identifiers
   4.2. Unary Expressions
   4.3. Binary Expressions
   4.4. Relational Expressions

5. Statements
   5.1. Expression Statements
   5.2. Conditional Statements
   5.3. While
   5.4. For
   5.5. Break
   5.6. Return

6. Scope
   6.1. Lexical Scope
   6.2. Function Declarations

7. Example Code
1 Overview of J-Pie

J-Pie is a language inspired by Python, but without the constraint of indentation to indicate blocks. J-Pie retains the syntax that makes Python an easy and fun language, while enforcing curly braces to indicate blocks and semicolons to indicate statement endings to increase portability. In addition, J-Pie has the notion of a relation, which is a concept from logic programming. Specifically, our language will attempt to mimic this from the first-order-logic programming language, Prolog.

2 Motivation

The motivation for our language was to create a language that takes the best of both worlds from Python and Java. J-Pie is not bound by Python’s indentation rules or Java’s OOP rules. For many programmers who use Python, using a Python script on a different machine or deploying it into a production environment can be a nightmare when you receive the error “IndentationError: unindent does not match any outer indentation level”. After this, Python users have no idea if one indentation is broken or all of them. Having to go back into your script and delete until the previous line then press “enter”, then “tab” until your desired definition. We set out to create a version of Python that uses curly braces rather than indentation (like Java or C) to avoid this issue.

3 Lexical conventions

3.1 Comments
Block comments can be single-line or multi-line, as long as they are enclosed by two ## symbols. For example, ## this is a comment ##.

3.2 Identifiers (Names)
Identifiers follow the same conventions as Python. Valid identifiers are made of letters in lowercase (a to z) or uppercase (A to Z), digits (0 to 9), and/or an underscore _. Note that the first character must be a lowercase letter, not a digit or an underscore. For example, varName_1 is a valid identifier.

3.3 Type Specifiers and Operators
J-Pie supports the following types: int, boolean, float, String, List<>. There are several operators affiliated with each type. The following symbols are reserved as operators. Note that some operators are overloaded. For further explanation of how these operators are used, refer to Section 4 Expressions.
3.5 Keywords
J-Pie reuses several keywords from Python. There is a keyword corresponding to each control structure, namely `for`, `while`, `if`, `elif`, and `else`. Additionally `def` is reserved for function definitions, `return` is reserved for return statements, and `break` is reserved for terminating the loop containing it. `True` and `False` are reserved as boolean values, and `None` denotes a null value or no value at all. `print` and `len` are the names of two built-in functions.

4 Expressions
4.1 Identifiers
An identifier is a primary expression whose type is specified by its declaration: `int x` means that `x` is of type `int`.

4.2 Unary Expressions
The table below shows all unary expressions that J-Pie features.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Operators</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| `int`     | =, ==, !=, <, <=, >, >=, *, /, +, -, ^, ++, --, % | A regular integer type int | `int x = 6;`  
`int y = 4 ^ 7;` |
| `boolean` | =, ==, !=, and, or | Evaluates to `True` or `False` | `boolean rainy = true;`  
`boolean wet = false;`  
rainy and wet; `##false##` |
| `float`   | =, ==, !=, <, <=, >, >=, *, **, /, +, - | A double-precision float type | `0.3 + (1/5);` `##0.5##` |
| `String`  | =, ==, !=, + | Regular string type. Characters can be represented by strings of length 1. | `String a = “Colu”;`  
`String b = “mbia”;`  
a+b; `##“Columbia”##` |
| `List<>`  | in, not in, append | Can hold only homogeneous-type collection | `List<int> p = [1,2,3];`  
p.append(9); `##p == [1,2,3,9]##` |
### 4.3 Binary Expressions

The table below shows all binary expressions that J-Pie features.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>The logical negation operator ! changes the value of a binary expression to its opposite. If the value of an expression is True, the result of the ! expression is False. This operator is applicable only to Boolean expressions.</td>
</tr>
<tr>
<td>-</td>
<td>The negative of a numerical expression - turns the value of a numerical expression to its equivalent negative of the same type. This operator is applicable only on numeric expressions: int, float</td>
</tr>
<tr>
<td>++</td>
<td>The increment of a numerical expression ++ increments the value of a numerical expression by 1. This operator is applicable only on int</td>
</tr>
<tr>
<td>--</td>
<td>The decrement of a numerical expression -- decrements the value of a numerical expression by 1. This operator is applicable only on int</td>
</tr>
<tr>
<td>len</td>
<td>The len expression gives the size in length of a collection. When applied to a String the result is the total number of characters, when applied to an Array the result is the total number of elements in the Array.</td>
</tr>
<tr>
<td>=</td>
<td>The assignment expression, which groups right to left, takes two expressions and the value of the right expression is stored in the left expression. The type of the expression being stored must be the same as the type declaration of the expression on the left.</td>
</tr>
<tr>
<td>*</td>
<td>The binary * operator multiplies the expressions. This operator is applicable only on numeric expressions: int, float</td>
</tr>
<tr>
<td>/</td>
<td>The binary / operator divides the first expression by the right expression. This operator is applicable only on numeric expressions: int, float</td>
</tr>
</tbody>
</table>
| %          | The binary % operator gives the remainder of the division of the expressions. This operator is applicable only on numeric expressions:
### 4.4 Relational Expressions

The table below shows all relational expressions that J-Pie features.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; less than</td>
<td>The binary &lt; operator returns True if the expression on the left is smaller than the expression on the right. This operator is applicable only on numeric expressions.</td>
</tr>
<tr>
<td>&gt; greater than</td>
<td>The binary &gt; operator returns True if the expression on the left is larger than the expression on the right. This operator is applicable only on numeric expressions.</td>
</tr>
<tr>
<td>&lt;= less than or equal</td>
<td>The binary &lt;= operator returns True if the expression on the left is smaller than or equal to the expression on the right. This operator is applicable only on numeric expressions.</td>
</tr>
<tr>
<td>&gt;= great than or equal</td>
<td>The binary &gt;= operator returns True if the expression on the left is larger than or equal to the expression on the right. This operator is applicable only on numeric expressions.</td>
</tr>
<tr>
<td>== equals</td>
<td>The binary == operator returns True if the two expressions are equal.</td>
</tr>
<tr>
<td>!= not equals</td>
<td>The binary != operator returns True if the two expressions are not equal.</td>
</tr>
</tbody>
</table>
and | The binary and operator returns True if both expressions evaluate to True.
---|---
or | The binary or operator returns true if at least one of the expressions evaluate to True.

5 Statements

5.1 Expression Statement
Expression statements have the form: `expression;` Expression statements are typically assignments or function calls.

5.2 Conditional Statement
The three forms of the conditional statement are

```c
if (expression) {
    statement;
}

If (expression) {
    statement;
} else {
    statement;
}

if (expression) {
    statement;
} elif (expression) {
    statement;
} ...
else {
    statement;
}
```

If in any of the above cases the expression is evaluated to true, the first substatement is executed. In the second case and third case, the last substatement is executed if the previous conditions do
not evaluate to true. The third case chains if conditions when there are more than two expressions to evaluate, and executes the associated substatement under that condition.

5.3 While Statement
The **while** statement has the form

```plaintext
while (expression) {
    statement;
}
```

The substatement is executed continuously while the value of the expression evaluates to true. The evaluation takes place prior to the execution of each substatement inside of the loop.

5.4 For Statement
The **for** statement has the form

```plaintext
for (<element> in <Collection>) {
    statement;
}
```

The `<Collection>` represents a collection of data types and on each iteration of the loop the `<element>` variable will be assigned to the element of the `<Collection>` on that specific iteration. On each iteration, the substatement is executed.

5.5 Break Statement
The statement **break**; will force the termination of the smallest enclosing **while** or **for** loop statement.

5.6 Return Statement
A function is terminated and returned to its caller by the **return** statement, which has one of the following forms:

```plaintext
return;
return (expression);
```

In the first case returns the None value. The second case returns the value of the expression to the function caller.

6    Scope
6.1 Lexical scope

Variables created inside a curly bracket’s scope exclusively belong to the scope. The user cannot access the variable outside of the scope unless creating a new variable of the same name and/or a different value. Identifiers are permitted to be used only within their region, enclosed by curly brackets. Attempts to use an identifier outside of its region will result in an error of “undefined variable.”

- Local: a local variable can be used inside the scope it was initialized
- Global: a global variable can be used anywhere within the scope in which it was initialized. It can also be used inside “indented” scopes.

Example:

```plaintext
int x = 7;
int y = 9;
{
    int z = x + y;
}
print(z) ## will raise an error - z is undefined
```

6.2 Function declarations

```plaintext
def returntype function-name(datatype parameter-1, datatype parameter-2, ..., datatype parameter-n) {
    statement;
}
```

6.3 Mutability

Primitives are immutable - the only way to change the value of a primitive is by creating a new object of the same name. Data structures are mostly mutable:

- List - elements of a list can be changed by accessing the index
- Tuples - immutable
- Dictionary - keys of a dictionary are immutable; values can be changed by accessing a specific key.

7  Example Code

7.1 Defining and using a simple function

```plaintext
def int max(int a, int b) {
    if(a > b) {
        return a;
    } else {
        return b;
    }
}
```
return a;  ## both are the same, just return value a ##

```
int x = 1;
int y = 5;

int max_value = max(x, y);
print(max_value);
```

### 7.2 Using a while loop and printing

```
bool flag = true;
int i = 1;

while (flag) {
    print(i);
    i++;
    if (i == 10) {
        flag = false;
    }
}
```

### 7.3 String Concatenation

```
string word = “race”;
string other_word = “car”;

string combination = word + other_word;
```

### 7.4 Appending elements to a list and iterating over it

```
int list L = [1, 2, 3, 4];

L.append(5);
```
for (int i in L) {
    print(i);
}