CMPUT 275 Wi18 - INTRO TO TANGIBLE COMPUT II Combined LBL Wi18

Exercise 4: Code Generation using AST

The purpose of this exercise is to:

- Practice object oriented programming by reading the code for an expression-based calculator with variables, defined in `calculator.py`
- Understand how the Abstract Syntax Tree is created for an input string by reading the code in `exprparser.py`
- Understand how the Abstract Syntax Tree is used by a compiler to generate code

1) Abstract Syntax Tree (AST)

You should start with the simple calculator in `calculator.py` as a study in how to read object oriented programs. The calculator uses an instance of the expression parser class, `ExprParser`, to build an AST for the input string. This is done by calling its `parse` method. It then evaluates the AST to produce a numerical result with side effects of setting variables to values in the expression. This is done by calling the `evaluate_ast` method of the calculator. Read the code in `evaluate_ast` to find out how different node types in the AST are evaluated.

Note: there is nothing to submit for this part; the task is just to understand the code to do the second part.

2) Code Generation

In this exercise, you will walk over the AST and produce Python code which performs the same calculations when executed. The module `compiler.py` contains class definition for Compiler. The main code generator is a method called `compile`. You should **NOT** change this method. Most of the work is done in the method called `compile_ast`. This is where all new code you produce (apart from tests) should be added, and detailed descriptions of the task is in this method. In particular you need to implement the handling of 'set' and 'apply' nodes in the AST. You should support assignment (=), binary (+, -, *, /), and unary operations (neg, sqr) as well as function calls.

You can test your code using `comptest.py`

```
python3 comptest.py
```

and with doctests

```
python3 comptest.py --test
```

Examples:
Expression that contains binary and unary operations

```
python3 comptest.py
?(1 - 2) - 3 + 4 - - 5
Initial ast: ('apply', [('-', []), ('apply', [('+', []), ('apply', [('-', []), ('apply',
[('-', []), ('const', [(1, [])]), ('const', [(2, [])])]), ('const', [(3, [])])]), ('cons
t', [(4, [])])]), ('apply', [('neg', []), ('const', [(5, [])])])])

# code for:
# (1 - 2) - 3 + 4 - - 5
_result = ((((1 - 2) - 3) + 4) - (-5))
print(_result)

Running program ...
locals before []
5
locals after [('_result', 5)]
```

Expression that contains unary operations

```
python3 comptest.py
?-sqr(2)
Initial ast: ('apply', [('neg', []), ('apply', [('sqr', []), ('const', [(2, [])])])])

# code for:
# -sqr(2)
_result = (-(2 ** 2))
print(_result)

Running program ...
locals before []
-4
locals after [('_result', -4)]
```

Expression that contains a function call

```
python3 comptest.py
?max(1, 2)
Initial ast: ('apply', [('max', []), ('const', [(1, [])]), ('const', [(2, [])])])

# code for:
# max(1, 2)
_result = max(1, 2)
print(_result)

Running program ...
locals before []
2
locals after [('_result', 2)]
```

Expression that contains assignments

```
python3 comptest.py
?(x = 1) + (y = x + 1)
Initial ast: ('apply', [('+', []), ('set', [('x', []), ('const', [(1, [])])]), ('set', [
('y', []), ('apply', [('+', []), ('get', [('x', [])]), ('const', [(1, [])])])])])

# code for:
# (x = 1) + (y = x + 1)
_x_1 = 1
_y_1 = (_x_1 + 1)
_result = (_x_1 + _y_1)
# Update and cleanup x
x = _x_1
del(_x_1)
# Update and cleanup y
y = _y_1
del(_y_1)
print(_result)

Running program ...
locals before []
3
locals after [('_result', 3), ('x', 1), ('y', 2)]
```

Implementation strategy:

First get code generation working for expressions that do not involve any variable assignment operations (e.g. 2 + 3). Then worry about how to deal with assignment statements.

Summary of the task:

You have to:
1. Add code to `compile_ast` to make it run properly
2. Add at least five docstring tests to `tests`

Code Submission

Submit only the file `compiler.py`, do NOT zip it. You must create at least five more strings in the expression language to test on. These should be added as a docstring to the `tests` function in `compiler.py`. We will run these tests via

```
python3 compiler.py
```

We also use `comptest.py` to test your code.

Ex4.zip

Submission status

Attempt number	This is attempt 1 (1 attempts allowed).
Submission status	Submitted for grading

Grading status	Graded
Due date	Monday, 5 March 2018, 11:55 PM
Time remaining	Assignment was submitted 2 hours 45 mins early
Last modified	Monday, 5 March 2018, 9:09 PM
File submissions	compiler.py Export to portfolio
Submission comments	Comments (0)

Feedback

Grade	93.50 / 100.00
Graded on	Saturday, 17 March 2018, 4:05 PM
Graded by	Taher Jafferjee
Feedback comments	**Correctness** Your code passed 24 of 26 tests, and your grade on correctness is 92.31 **Style** Your mark on style is 100% **Weight Total** 93.50%