3.12 Test data for post-development

In this section, I will produce a comprehensive testing plan. Instead of simply testing input and output to the program as a whole (which would be cumbersome and unrevealing) I will instead make tests for each individual function in the program, including valid, invalid and borderline data, and as comprehensive a testing strategy as possible for that individual function. For the compiler and assembler, which are coded in Python, I will put this into effect by using Python’s unittest framework, which will make the tests versatile and repeatable. For the interpreter, programmed in C, I will use a similar framework called CUnit.

In each section below is a table. This will for each test give the test number and name, the method it will work on, the data to be put into the function as an argument, the testing criteria (typically this will be the expected output of the function, but it could be an exception that should have been raised or something else), and a general discussion of the test.

# Assembler

The tests are organised into 7 groups. The group designated A0 (with the A standing for assembler, starting at A001) are general program tests. Following this, A1, 2, 3, 4 and 5 correspond to tests involving their sections in the Decomposing the Problem Section. A6 tests the methods within Instruction objects, A7 tests Operand objects and then A8 holds tests involving the GUI and educational aspects.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test name/num | Function | Input data | | Expected result | Discussion |
| A0 (general tests) | | | | | |
|  |  |  | |  |  |
| A1 (performing text normalisation) | | | | | |
| A101 – normalise semantically empty text | normalise\_text | (space character) | | Output of "" | These are four things that the function should get rid of, so test each of them on their own |
| (newline) | |
| (tab character) | |
| ; Comment | |
| A102 – leave minimal text unchanged | MOV eax ebx | | Output same as input | The function should not be making any unnecessary changes |
| A103 – remove space between tokens | MOV eax ebx | | Output is the input with all duplicate whitespace removed and no whitespace at either side | This is likely the main job of the function, to make a line minimal |
| ADD (tab) ecx 5 | |
| A104 – remove empty lines | MOV eax [ebx]  ADD eax 5 | | Output is the same except with one less newline in the middle | This is another basic function of the normalisation function |
| A105 – remove all comments | CMP eax ebx ; On line  ; On own | | Both comments are removed and no whitespace is left except that between the tokens | This is the final individual piece of the function’s job, to remove comments. This checks it works both ways. |
| A106 – all mixed | section.data ; A section  i VAR ubyte (tab) 0  x VAR int (tab) -100  ; Another section  section.text  MOV eax i  MOV ebx x | | section.data  i VAR ubyte 0  x VAR int -100  section.text  MOV eax i  MOV ebx x | This tests that all of the individual parts work together. |
| A2 (split document into its sections) | | | | | |
| A201 – no content | split\_into\_sections | section.meta  section.data  section.text | | {"meta": "", "data": "", "text": ""} | This is the most basic test, jus checking to see if it can divide a simple document |
| A202 – with commands | section.meta  a=b  c=d  section.data  x VAR int 50  y VAR int 5  section.text  MOV eax x  ADD eax y  MOV out eax | | {"meta": "a=b  c=d", "data":  "x VAR int 50  Y VAR int 5",  "text": "MOV eax x  ADD eax y  MOV out eax"} | This is a more complex example, with data in between to put into the dictionary. |
| A203 – section missing |  | section.meta  section.data | | Throws an AssemblyError | All three sections must be present, so this checks that this is guaranteed |
| A3 (divide lines and contextualise) | | | | | |
| A301 – empty section dict | divide\_and\_contextualise | {"meta": "", "data": "", "text": ""} | | The default config dict and an empty list | This performs a baseline checks for errors and oddities |
| A302 – data in config dict | {"meta": "a=b\nc=d", "data": "", "text": ""} | | The default config dict plus the values "a": "b" and "c": "d", with an empty list | This does not check the second part of the function, dealing with the data and text sections, but just makes sure it correctly interprets the meta section |
| A303 – a single line in the data section | {"meta": "", "data": "a VAR int 5", "text": ""} | | The default config dict and a list with a single DataInstruction matching the given parameters | This makes sure the function can correctly process a basic data section entry and add it to the list |
| A304 – a single line in the text section | {"meta": "", "data": "", "text": "MOV eax ebx\nSUB eax 5"} | | The default config dict and a list with two TextInstruction objects corresponding to the given commands | This confirms that the processing of the text section works. More thorough testing will be done in tests A6\* and A7\*. |
| A305 – all combined | {"meta": "a=b\nc=d", "data": "a VAR int 5", "text": "MOV eax ebx"} | | The default config dict plus the values "a": "b" and "c": "d", with a list containing the expected DataInstruction and TextInstruction | This makes sure all the parts work together. |
| A310 – empty operand | interpret\_operand | (empty string) | | Raises a ValueError | By the time this function is called, the chance of an empty operand should have been removed so something went wrong |
| (tab character) | |
| A311 – register name | eax | | Returns a correct RegisterOperand object. | This is the standard entry format of one type of operand. |
| A312 – upper case register name | EAX | | Returns a correct RegisterOperand object | The function should be able to deal with alternative styles of input |
| A313 – integer | 5 | | Returns a correct ImmediateOperand object | This is the simplest type of immediate operand that should be understood |
| A314 – negative integer | -100 | | Returns a correct ImmediateOperand object | Negative numbers are permitted as well as positive |
| A315 – floating point | 7.5 | | Returns a correct ImmediateOperand object (including having the float data type) | Floating points are a valid type |
| A316 – non-register string | varname | | Returns an AddressOperand with a string value | At this point the variable names have not actually been turned into memory addresses but an Operand should still be produced |
| A317 – empty arithmetic statement | [] | | Raises a ValueError | This is an invalid sequence and should be treated as such |
| A318 – simple arithmetic operand ("a") | [eax] | | Raises the correct type of ArithmeticOperand | This is a valid type of arithmetic operand |
| A319 – multiplication operand ("a\*b") | [eax\*ebx] | |
| A320 – addition operand ("a+b") | [eax+ebx] | |
| A321 – addition and multiplication operand |  | [eax+ebx\*ecx] | |
| A322 – multiplication and addition operand | [eax\*ebx+ecx] | |
| A323 – Mixing immediate and register | [eax\*4] | | Immediate operands are a necessary part as well |
| A324 – illegal whitespace | [eax \* 4] | | Raises an AssemblyError | Whitespace here is not supposed to be allowed and this should be enforced |
| A4 (record labels and variables) | | | | | |
| A401 – return an empty memory table for no instructions | record\_labels\_  and\_variables | [] | | {} | This is a sample of the base case, making sure the function can correctly run with no input. |
| A402 – record the address for a single variable | [DataInstruction(0, "i", "5", "int")] | | {"i": 7} | This is the first simple, atomic job of the function |
| A403 – record the address for a single label | [TextInstruction(0, "MOV", "2B", RegisterOperand("eax"), RegisterOperand("ebx"), "start")] | | {"start": 0} | This is the second thing this function needs to process |
| A410 – correctly get the size of an empty var table | calculate\_var\_table\_size | {} | | 0 | This is the baseline once again |
| A411 – get correct size of each type | {"x": (0, "char")} | | 1 | This shows two things: that the function knows the size of each data type, and that it can process a single-valued table and give a sensible answer. |
| {"x": (0, "uchar")} | | 1 |
| {"x": (0, "short")} | | 2 |
| {"x": (0, "ushort")} | | 2 |
| {"x": (0, "int")} | | 4 |
| {"x": (0, "uint")} | | 4 |
| {"x": (0, "float")} | | 4 |
| A412 – correctly calculate the size of a complex table | {"x": (0, "char"), "y": (1, "float"), "z": (5, "ushort"), "temp": (7, "uchar")} | | 8 | This shows that the function can process multiple entries of multiple different types |
| A5 (convert each line to bytes) | | | | | |
| A501 – performs replacement correctly | place\_memory\_addresses | {"i": 50}, [TextInstruction(0, "MOV", "1B", AddressOperand("i"), RegisterOperand("eax"))] | | [TextInstruction(0, "MOV", "1B", AddressOperand(50), RegisterOperand("eax"))] | This tests the ability of the function to perform the correct replacement |
| A502 – skips data instructions | {"x": 100}, [DataInstruction(0, "x", "20", "int")] | | [DataInstruction(0, "x", "20", "int")] | Data instructions have no reason for any replacement to happen |
| A510 – empty dict gives only buffer | encode\_metadata | {} | | b"\x00\x00\x00\x00" | No metadata should leave no data except the buffer |
| A511 – single value correctly encoded | {"key": "value"} | | b"key=value&\x00\x00  \x00\x00" | This proves that the encoding works correctly, if only for a single value |
| A512 – several values correctly encoded | {"a": "b", "c": "d"} | | b"a=b&c=d&\x00\x00\x00  \x00" (the first two parts can be swapped) | This shows the loop is working correctly |
| A513 – non-string values handled | {"a": "b", "i": 5} | | b"a=b&i=5\x00\x00\x00  \x00" (the first two parts can be swapped) | This is necessary for the sake of robustness |
| A520 – encodes a single instruction correctly | encode\_instruction\_list | [TextInstruction(0, "MOV", "1B", RegisterOperand("eax"), ImmediateOperand("5"))], {} | | b"\x10\x12\xA0\x05" | The testing of this function is minimal even though it is important because the actual work is mostly done by the Instruction objects which are tested separately |
| A6 (Instruction objects) | | | | | |
| Test name/num | Description | | | Expected result | Discussion |
| A601 – Instruction is abstract | Create an Instruction object and run both get\_bytes\_length and get\_bytes on it. | | | Raise a NotImplementedError | It should only be possible to run these functions on a subclass |
| A610 – DataInstruction equality works | Create two distinct but identical objects and check for equality between them | | | True | Equality has been implemented, and must be tested |
| A611 – correctly calculates value size | Create one DataInstruction object for each of the values to the right and run \_calculate\_valsize on them. | | "-1000000" | 4 | This is an important function, which finds how many bytes will be taken up by the initial value of the variable |
| "-20000" | 2 |
| "-10" | 1 |
| "10" | 1 |
| "256" | 2 |
| "65536" | 4 |
| "3.67" | 4 |
| A612 – bytes length and length of bytes agree | Create a DataInstruction object and run get\_bytes\_length on it. Then run get\_bytes and check that the length of this value is the same as the function before said. | | |  | The get\_bytes\_length method is important, and used for calculating memory addresses. It must be trustable. |