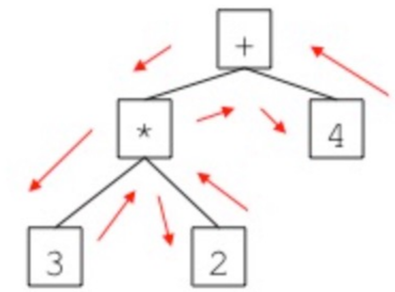
Processing ASTs: Tree Walking

- The recursive structure of trees gives rise to an elegant way of processing trees: *tree walking*.
- A tree walker typically starts at the root node and traverses the tree in a depth first manner.

Processing ASTs: Tree Walking

Consider the following:



3*2+4

```
$ python3
>>> from dumpast import dumpast
>>> ast = ('PLUS', ('MUL', ('INTEGER', 3), ('INTEGER', 2)), ('INTEGER', 4))
>>> dumpast(ast)

(PLUS
  |(MUL
  |   |(INTEGER 3)
  |   |(INTEGER 2))
  |(INTEGER 4))
>>>
```

Processing ASTs: Tree Walking

A simple tree walker for our expression tree

```
dispatch_dictionary = {
    'PLUS'    : add,
    'MUL'     : multiply,
    'INTEGER' : const
}
```

```
def walk(node):
    # first component of any tree node is its type
    t = node[0]

    # lookup the function for this node
    node_function = dispatch_dict[t]

    # now call this function on our node and capture the re
    val = node_function(node)

    return val
```

```
##############################################################
def const(node):
    # pattern match the constant node
    (INTEGER, val) = node

    # return the value as an integer value
    return int(val)

##############################################################
def add(node):
    # pattern match the tree node
    (PLUS, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the sum of the values of the children
    return left_val + right_val

##############################################################
def multiply(node):
    # pattern match the tree node
    (MUL, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the product of the values of the children
    return left_val * right_val
```

Processing ASTs: Tree Walking

A simple tree walker for our expression tree

```
$ python3
>>> from dumpast import dumpast
>>> ast = ('PLUS', ('MUL', ('INTEGER', 3), ('INTEGER', 2)), ('INTEGER', 4))
>>> dumpast(ast)

(PLUS
  |(MUL
  |  |(INTEGER 3)
  |  |(INTEGER 2))
  |(INTEGER 4))
>>>
>>> walk(ast)
10
>>>
```

We just interpreted the expression tree!!!

Processing ASTs: Tree Walking

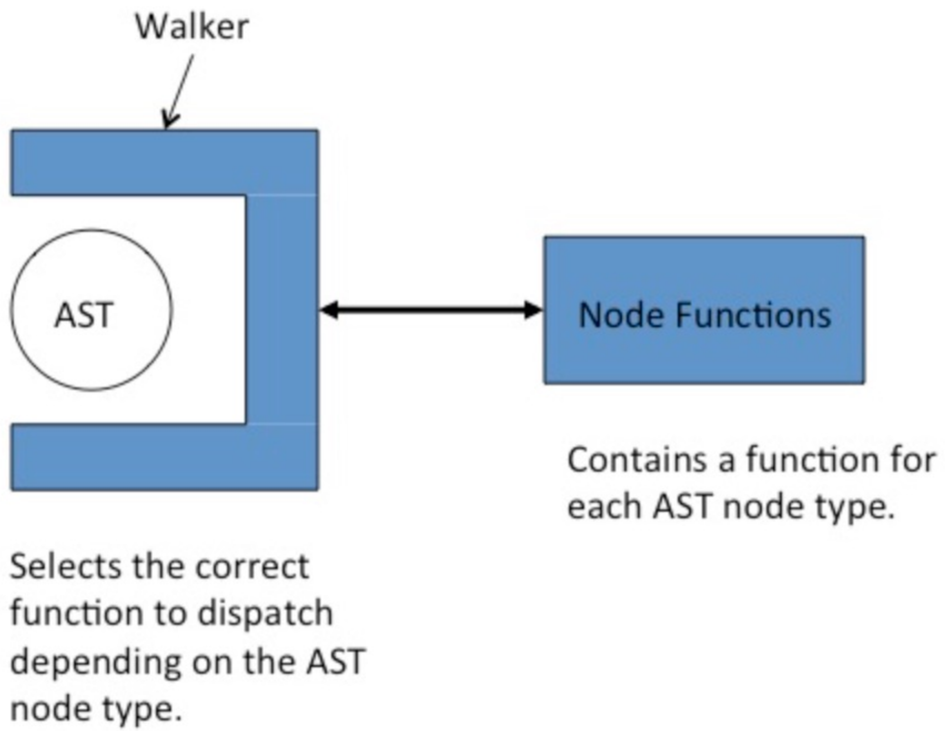
A simple tree walker for our expression tree

```
##########################################################
def const(node):
    # pattern match the constant node
    (INTEGER, val) = node

    # return the value as an integer value
    return int(val)

##########################################################
def add(node):
    # pattern match the tree node
    (PLUS, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the sum of the values of the children
    return left_val + right_val

##########################################################
def multiply(node):
    # pattern match the tree node
    (MUL, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the product of the values of the children
    return left_val * right_val
```

- Notice that this scheme mimics what we did in the syntax directed interpretation schema,
- But now we interpret an expression tree rather than the implicit tree constructed by the parser.

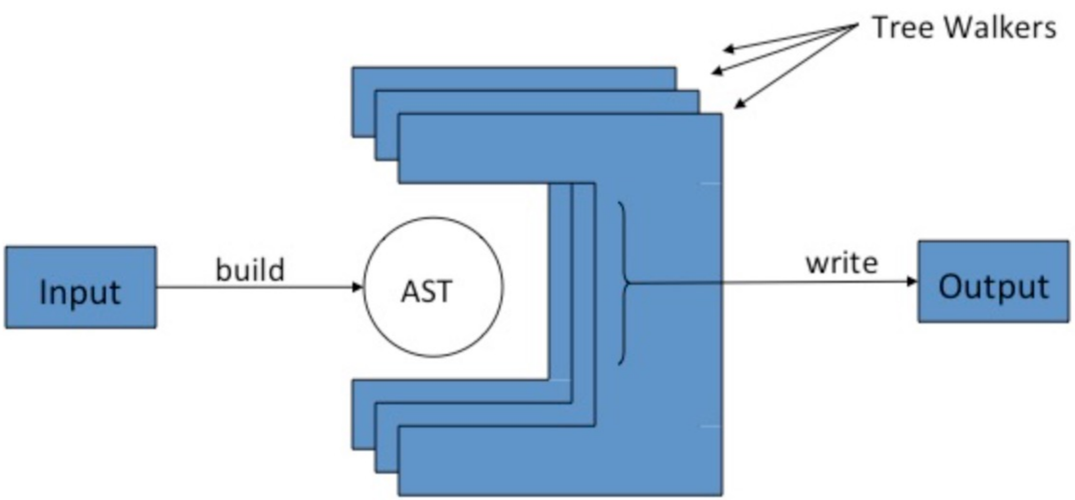
Tree Walkers are Plug'n Play

- Tree walkers exist completely separately from the AST.
- Tree walkers plug into the AST and process it using their node functions.

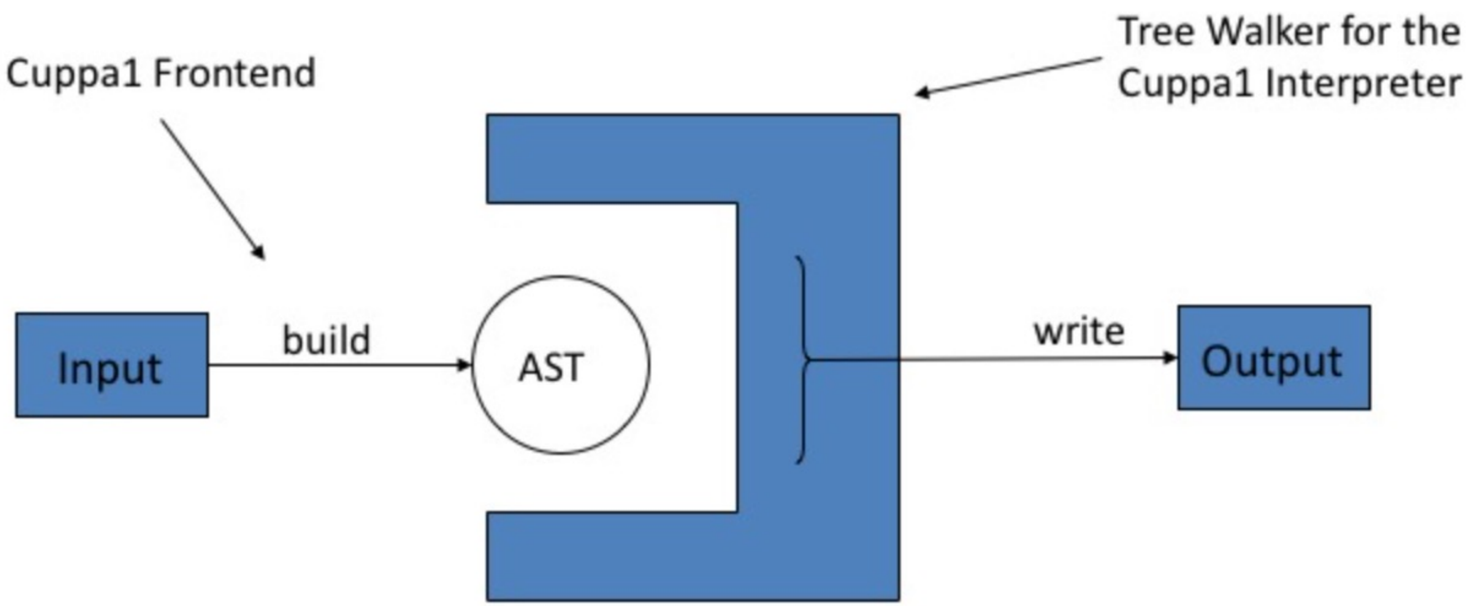


Tree Walkers are Plug'n Play

- There is nothing to prevent us from plugging in multiple walkers during the processing of an AST, each performing a distinct phase of the processing.



An Interpreter for Cuppa1



An Interpreter for Cuppa1

cuppa1_interp_walk.py

```
def walk(node):
    # node format: (TYPE, [child1[, child2[, ...]]])
    type = node[0]
    if type in dispatch:
        node_function = dispatch[type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + type)

# a dictionary to associate tree nodes with node functions
dispatch = {
    'STMTLIST' : stmtlist,
    'ASSIGN'   : assign_stmt,
    'GET'      : get_stmt,
    'PUT'      : put_stmt,
    'WHILE'    : while_stmt,
    'IF'       : if_stmt,
    'NIL'      : nil,
    'BLOCK'    : block_stmt,
    'INTEGER'  : integer_exp,
    'ID'       : id_exp,
    'PAREN'    : paren_exp,
    'PLUS'     : plus_exp,
    'MINUS'    : minus_exp,
    'MUL'      : mul_exp,
    'DIV'      : div_exp,
    'EQ'       : eq_exp,
    'LE'       : le_exp,
    'UMINUS'   : uminus_exp,
    'NOT'      : not_exp
}
```

An Interpreter for Cuppa1

cuppa1_interp_walk.py

```
def stmtlist(node):

    (STMTLIST, lst) = node

    for stmt in lst:
        walk(stmt)

    return None
```

```
def if_stmt(node):

    (IF, cond, then_stmt, else_stmt) = node

    if walk(cond) != 0:
        walk(then_stmt)
    else:
        walk(else_stmt)
    return None
```

```
def integer_exp(node):

    (INTEGER, value) = node

    return value
```

```
def assign_stmt(node):

    (ASSIGN, (ID, name), exp) = node

    value = walk(exp)
    state.symbol_table[name] = value

    return None
```

```
def plus_exp(node):

    (PLUS,c1,c2) = node

    v1 = walk(c1)
    v2 = walk(c2)

    return v1 + v2
```

```
def id_exp(node):

    (ID, name) = node

    return state.symbol_table.get(name, 0)
```

```
def while_stmt(node):

    (WHILE, cond, body) = node

    while walk(cond) != 0:
        walk(body)

    return None
```

Pattern matching on AST nodes!

An Interpreter for Cuppa1

cuppa1_state.py

cuppa1_interp.py

```
class State:
    def __init__(self):
        self.initialize()

    def initialize(self):
        # symbol table to hold variable-value association
        self.symbol_table = {}

state = State()
```

```
def interp(input_stream, dump=False):
    try:
        state.initialize()
        ast = parse(input_stream)
        if dump:
            dumpast(ast)
        else:
            walk(ast)
    except Exception as e:
        print("error: "+str(e))
    return None
```

```
if __name__ == "__main__":
    import sys
    import os

    ast_switch = False
    char_stream = ''

    if len(sys.argv) == 1: # no args - read stdin
        char_stream = sys.stdin.read()
    else:
        # if there is a '-d' switch use it
        ast_switch = sys.argv[1] == '-d'
        # last arg is the filename to open and read
        input_file = sys.argv[-1]
        if not os.path.isfile(input_file):
            print("unknown file {}".format(input_file))
            sys.exit(0)
        else:
            f = open(input_file, 'r')
            char_stream = f.read()
            f.close()

    interp(char_stream, dump=ast_switch)
```

Command line interface

Running the Interpreter

```
$ cat inc.txt
get x
x = x + 1
put x
$ python3 cuppa1_interp.py inc.txt
Value for x? 3
4
$
```

```
$ cat if.txt
get x; if (0 =< x) put 1 else put -1;
$ python cuppa1_interp.py if.txt
Value for x? 2
1
$ python cuppa1_interp.py if.txt
Value for x? -4
-1
$
```

```
~/.../chap05/cuppa1_interp$ cat fact.txt
// compute the factorial of x
get x;
y = 1;
while (1 =< x)
{
      y = y * x;
      x = x - 1;
}
put y;
~/.../chap05/cuppa1_interp$ python3 cuppa1_interp.py fact.txt
Value for x? 3
6
~/.../chap05/cuppa1_interp$
```

A Pretty Printer with a Twist

- Our pretty printer will do the following things:
  - It will read the Cuppa1 programs and construct an AST
  - It will compute whether a particular variable is used in the program
  - It will output a pretty printed version of the input script but <u>will flag assignment/get statements to variables which are not used in the program</u>

➔This cannot be accomplished in a syntax directed manner – therefore we need the AST

PrettyPrinting the Language

Listing 5.2: LL(1) grammar for the Cuppa1 language with precedence levels.

```
stmt_list : (stmt)*

stmt : ID = exp ;?
     | get ID ;?
     | put exp ;?
     | while \( exp \) stmt
     | if \( exp \) stmt (else stmt)?
     | \{ stmt_list \}

exp : exp_low
exp_low : exp_med ((== | =<) exp_med)*
exp_med : exp_high ((+ | -) exp_high)*
exp_high : primary ((\* | /) primary)*

primary : INTEGER
        | ID
        | \( exp \)
        | - primary
        | not primary

ID : <any valid variable name>
INTEGER : <any valid integer number>
```

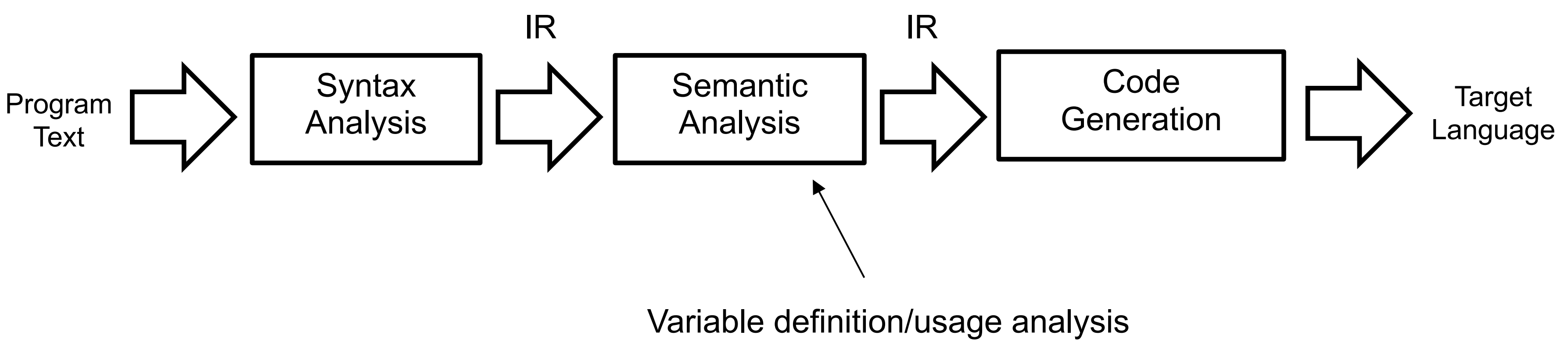
```
// list of integers
get x;
i = x;
while (1 <= x) {
      put x;
      x = x - 1;
}
```

```
get x
i = x  // -- var i unused --
while ( 1 <= x )
{
    put x
    x = x - 1
}
```

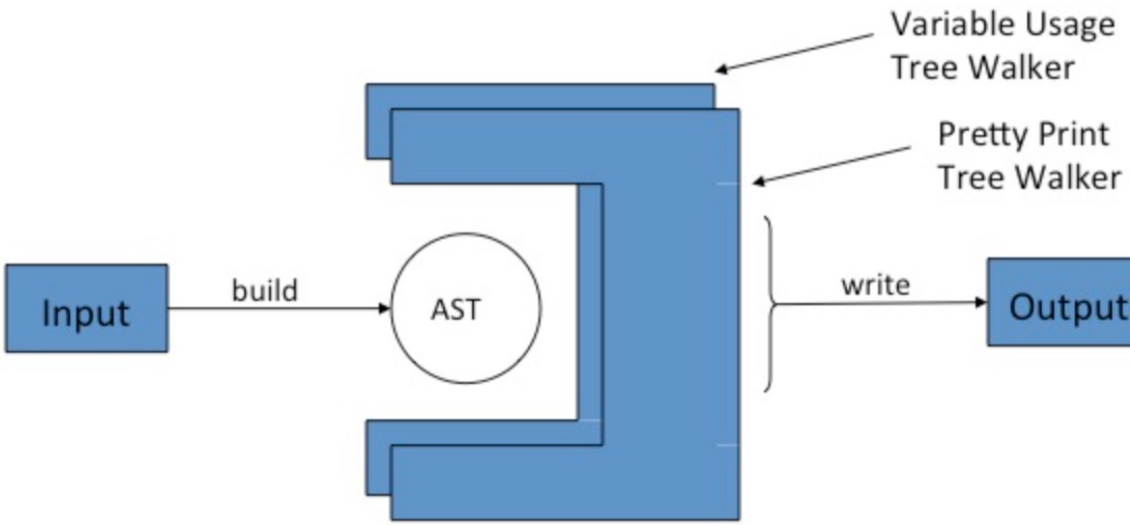
☞ We need an IR because usage will always occur after definition – cannot be handled by a syntax directed pretty printer.

The Pretty Printer is a Translator!

- The Pretty Printer with a Twist fits neatly into our translator class
  - Read input file and construct AST
  - Usage/Semantic Analysis
  - Generate output code, flagging unused assignments



Pretty Printer Architecture



Frontend + 2 Tree Walkers

PP1: Variable Usage

- The first pass of the pretty printer walks the AST and looks for variables in expressions
  - only those count as usage points.
- A peek at the tree walker for the first pass,
  cuppa1_pp1_walk.py
  shows that it literally just walks the tree doing nothing until it finds a variable in an expression.
- If it finds a variable in an expression then the node function for id_exp marks the variable in the symbol table as used,

PP1: Variable Usage

```
def walk(node):
    node_type = node[0]

    if node_type in dispatch_dict:
        node_function = dispatch_dict[node_type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + node_type)

# a dictionary to associate tree nodes with node functions
dispatch_dict = {
    'STMTLIST'    : stmtlist,
    'ASSIGN'      : assign_stmt,
    'GET'         : get_stmt,
    'PUT'         : put_stmt,
    'WHILE'       : while_stmt,
    'IF'          : if_stmt,
    'NIL'         : lambda node : None,
    'BLOCK'       : block_stmt,
    'INTEGER'     : lambda node : None,
    'ID'          : id_exp,
    'UMINUS'      : uminus_exp,
    'NOT'         : not_exp,
    'PAREN'       : paren_exp,
    'PLUS'        : binop_exp,
    'MINUS'       : binop_exp,
    'MUL'         : binop_exp,
    'DiV'         : binop_exp,
    'EQ'          : binop_exp,
    'LE'          : binop_exp
}
```

Just Walking the Tree!

PP1: Variable Usage

```
def assign_stmt(node):

    (ASSIGN, (ID, name), exp) = node

    state.symbol_table[name] = 'Defined'
    walk(exp)

    return None
```

```
def while_stmt(node):

    (WHILE, cond, body) = node

    walk(cond)
    walk(body)

    return None
```

```
def id_exp(node):

    (ID, name) = node

    # we found a use scenario of a variable
    state.symbol_table[name] = 'Used'

    return None
```

Just Walking the Tree!

```
def binop_exp(node):

    (OP, c1, c2) = node

    walk(c1)
    walk(c2)

    return None
```

PP1: Variable Usage

- According to the tree walker of our first phase a variable appearing in the symbol table has one of two states after the tree walker completes:
  - ‘Defined’ – a variable was defined in the program but never used
  - ‘Used’ – the value of a variable is being accessed, that is the variable is being used in an expression.
- We are interested in the first scenario…

PP1: Variable Usage

```
$ python3
### import our modules
>>> from cuppa1_state import state
>>> from cuppa1_fe import parse
>>> from cuppa1_pp1_walk import walk

### run the frontend and the walker
>>> state.initialize()
>>> ast = parse("get x")
>>> walk(ast)

### look at the symbol table
>>> state.symbol_table
'x': 'Defined'
>>>
```

Testing the tree walker

```
$ python3
### load our modules
>>> from cuppa1_state import state
>>> from cuppa1_fe import parse
>>> from cuppa1_pp1_walk import walk

### run the frontend and the walker
>>> state.initialize()
>>> ast = parse("get x; put x+1")
>>> walk(ast)

### look at the symbol table
>>> state.symbol_table
'x': 'Used'
>>>
```

PP2: Pretty Print Tree Walker

- The tree walker for the second pass walks the AST and compiles a formatted string that represents the pretty printed program.

```
def stmtlist(node):

    (STMTLIST, lst) = node

    code = ''
    for stmt in lst:
        code += walk(stmt)
    return code
```

Concatenate the string for each stmt into one long string.

PP2: Pretty Print Tree Walker

```python
def assign_stmt(node):

    (ASSIGN, (ID, name), exp) = node

    exp_code = walk(exp)
    code = indent() + name + ' = ' + exp_code
    if state.symbol_table[name] == 'Defined':
        code += ' // *** '+ name + ' is not used ***'
    code += '\n'
    return code
```

```python
def while_stmt(node):
    global indent_level

    (WHILE, cond, body) = node

    cond_code = walk(cond)

    indent_level += 1
    body_code = walk(body)
    indent_level -= 1

    code = indent() + 'while (' + cond_code + ')\n' + body_code

    return code
```

```python
def binop_exp(node):

    (OP, c1, c2) = node

    lcode = walk(c1)
    rcode = walk(c2)

    if OP == 'PLUS':
        code = lcode + ' + ' + rcode
    elif OP == 'MINUS':
        code = lcode + ' - ' + rcode
    elif OP == 'MUL':
        code = lcode + ' * ' + rcode
    elif OP == 'DIV':
        code = lcode + ' / ' + rcode
    elif OP == 'EQ':
        code = lcode + ' == ' + rcode
    elif OP == 'LE':
        code = lcode + ' =< ' + rcode
    else:
        raise ValueError("unknown OP")

    return code
```

Indent() and indent_level keep track of the code indentation for formatting purposes.

Top Level Function of PP

```
def pp(input_stream):

    try:
        state.initialize()
        init_indent_level()
        ast = parse(input_stream)
        pp1_walk(ast)
        code = pp2_walk(ast)
        print(code)
    except Exception as e:
        print("error: "+str(e))
```

Top level function

The Cuppa1 PP

Testing the pretty printer

```
$ python3 cuppa1_pp.py
get x;
^D

get x // *** x is not used ***

$ python3 cuppa1_pp.py
get x;
put x+1;
^D

get x
put x + 1

$
```

```
~/.../chap05/cuppa1_pp$ python3 cuppa1_pp.py
get x; while (1 =< x) { put x; x = x + - 1; i = x }


get x
while (1 =< x)
{
   put x
   x = x + -1
   i = x // *** i is not used ***
}

~/.../chap05/cuppa1_pp$
```

Assignment

- Reading: Chap 5