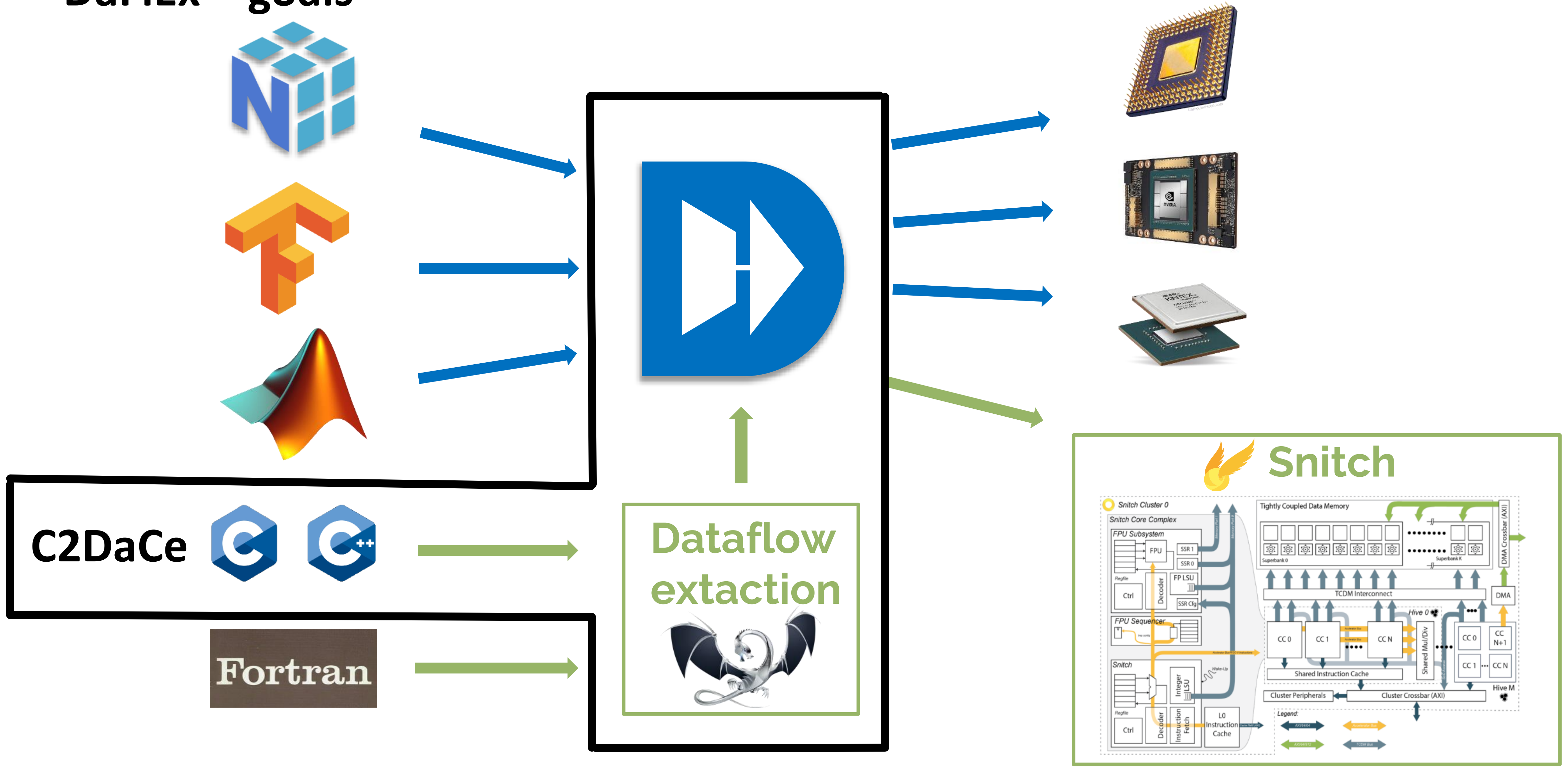
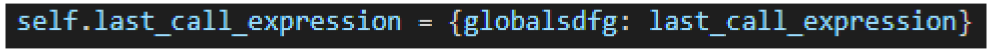
Alexandru Calotoiu

DaFlEx

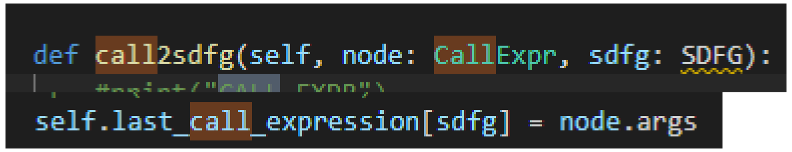
DaFlEx – goals



C2DaCe - from AST to SDFG

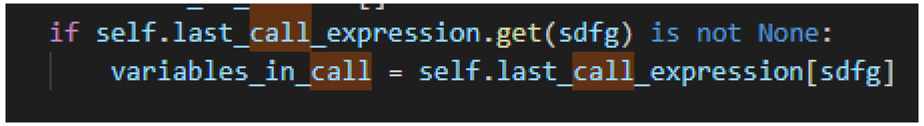


```
self.last_call_expression = {globalsdfg: last_call_expression}
```

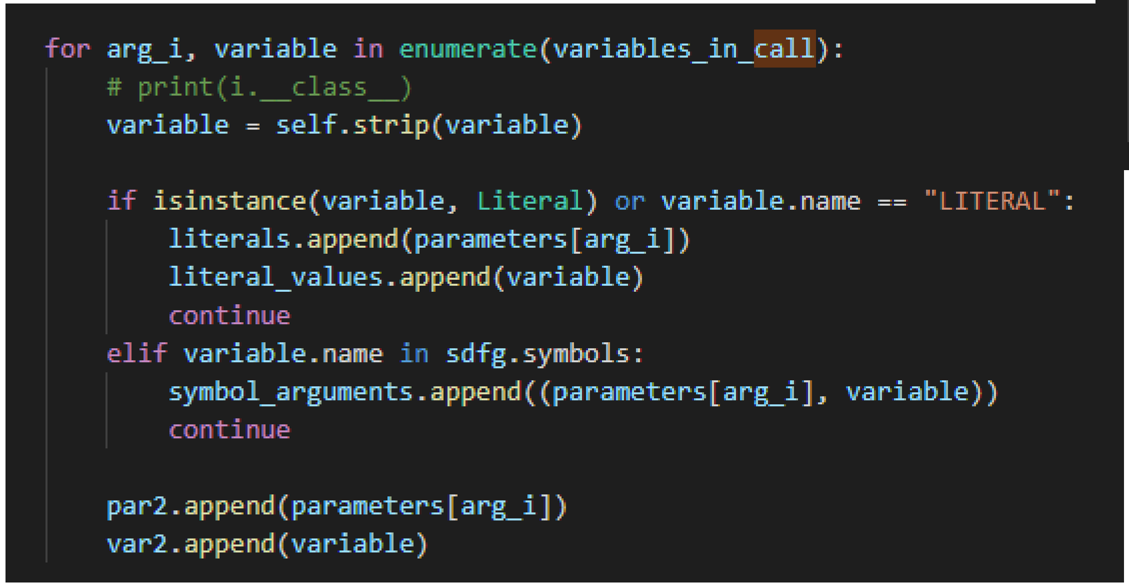


```
def call2sdfg(self, node: CallExpr, sdfg: SDFG):
    #print("CALL EXPR")
```

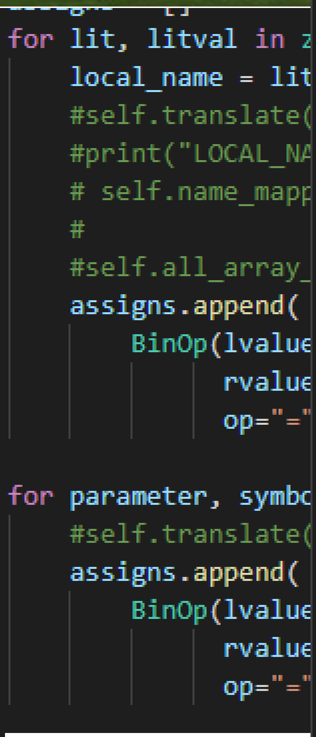
```
self.last_call_expression[sdfg] = node.args
```



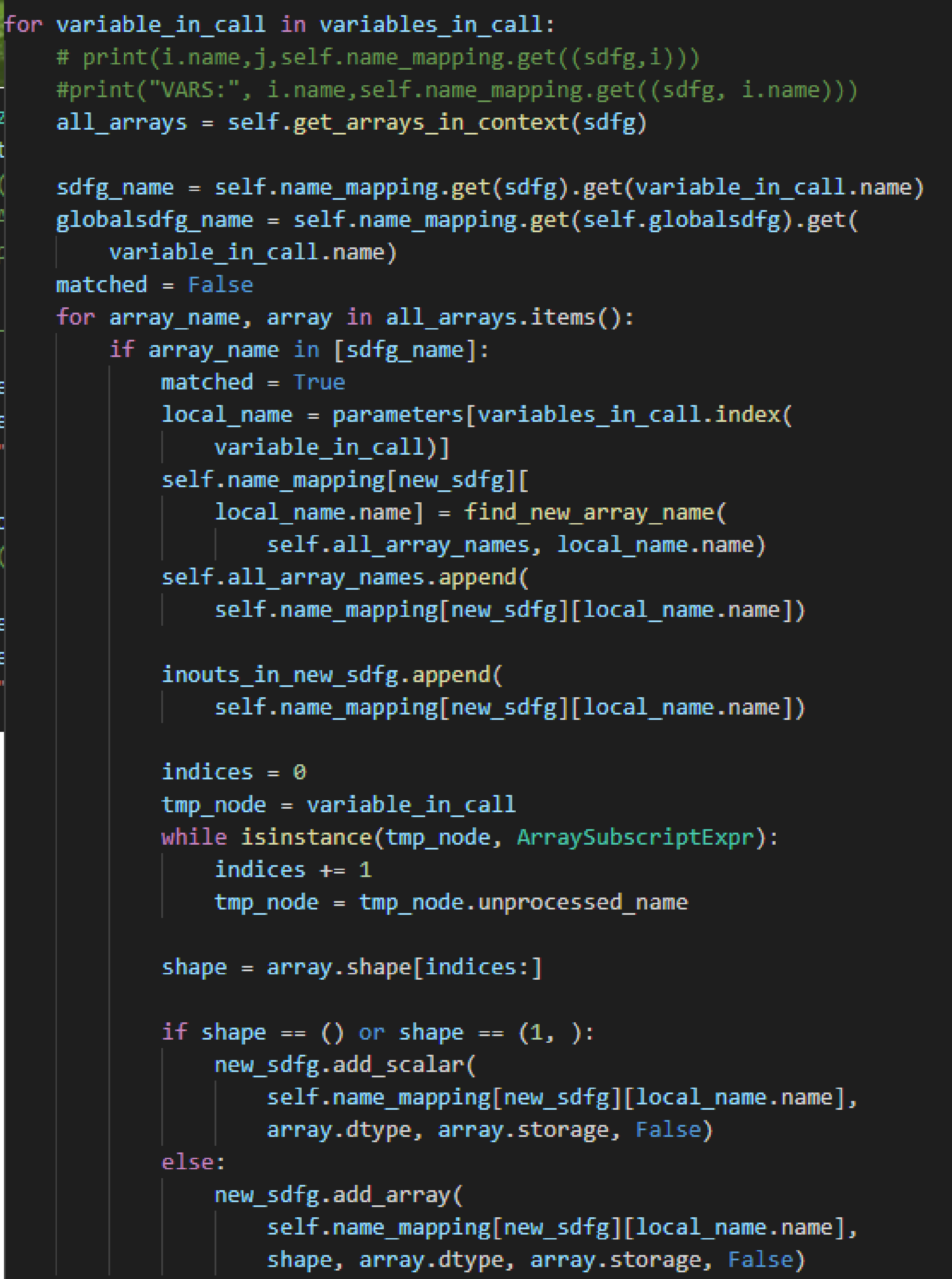
```
if self.last_call_expression.get(sdfg) is not None:
    variables_in_call = self.last_call_expression[sdfg]
```



```
for arg_i, variable in enumerate(variables_in_call):
    # print(i.__class__)
    variable = self.strip(variable)

    if isinstance(variable, Literal) or variable.name == "LITERAL":
        literals.append(parameters[arg_i])
        literal_values.append(variable)
        continue
    elif variable.name in sdfg.symbols:
        symbol_arguments.append((parameters[arg_i], variable))
        continue

    par2.append(parameters[arg_i])
    var2.append(variable)
```

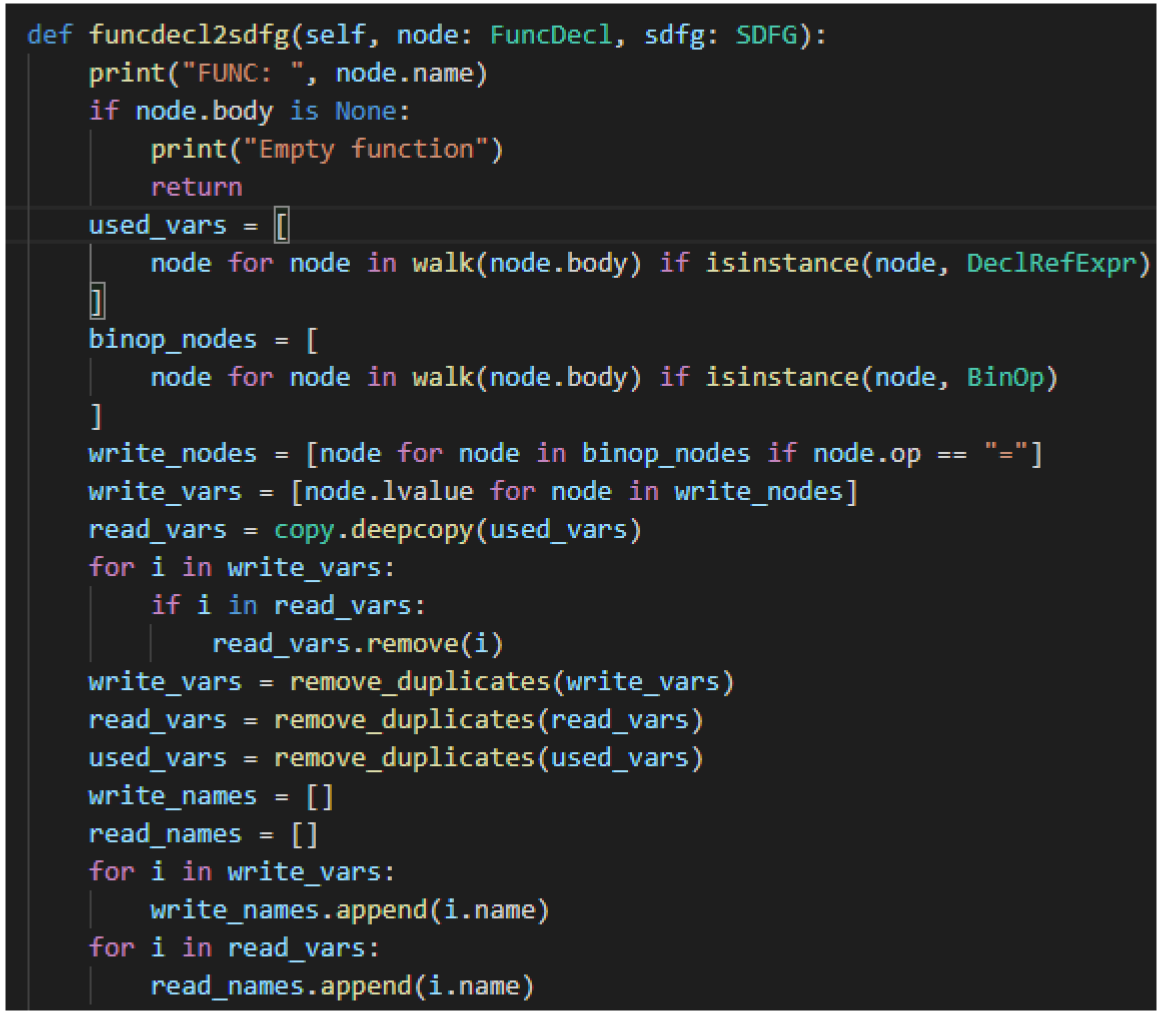


```
for lit, litval in z
    local_name = lit
    #self.translate(
    #print("LOCAL_NA
    # self.name_mapp
    #
    #self.all_array_
    assigns.append(
        BinOp(lvalue
              rvalue
              op="="

for parameter, symbo
    #self.translate(
    assigns.append(
        BinOp(lvalue
              rvalue
              op="="
```

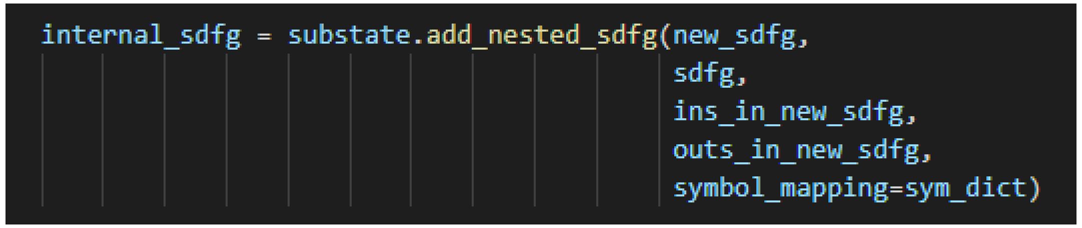


```
for variable_in_call in variables_in_call:
    # print(i.name,j,self.name_mapping.get((sdfg,i)))
    #print("VARS:", i.name,self.name_mapping.get((sdfg, i.name)))
    all_arrays = self.get_arrays_in_context(sdfg)

    sdfg_name = self.name_mapping.get(sdfg).get(variable_in_call.name)
    globalsdfg_name = self.name_mapping.get(self.globalsdfg).get(
        variable_in_call.name)
    matched = False
    for array_name, array in all_arrays.items():
        if array_name in [sdfg_name]:
            matched = True
            local_name = parameters[variables_in_call.index(
                variable_in_call)]
            self.name_mapping[new_sdfg][
                local_name.name] = find_new_array_name(
                    self.all_array_names, local_name.name)
            self.all_array_names.append(
                self.name_mapping[new_sdfg][local_name.name])

            inouts_in_new_sdfg.append(
                self.name_mapping[new_sdfg][local_name.name])

            indices = 0
            tmp_node = variable_in_call
            while isinstance(tmp_node, ArraySubscriptExpr):
                indices += 1
                tmp_node = tmp_node.unprocessed_name

            shape = array.shape[indices:]

            if shape == () or shape == (1, ):
                new_sdfg.add_scalar(
                    self.name_mapping[new_sdfg][local_name.name],
                    array.dtype, array.storage, False)
            else:
                new_sdfg.add_array(
                    self.name_mapping[new_sdfg][local_name.name],
                    shape, array.dtype, array.storage, False)
```

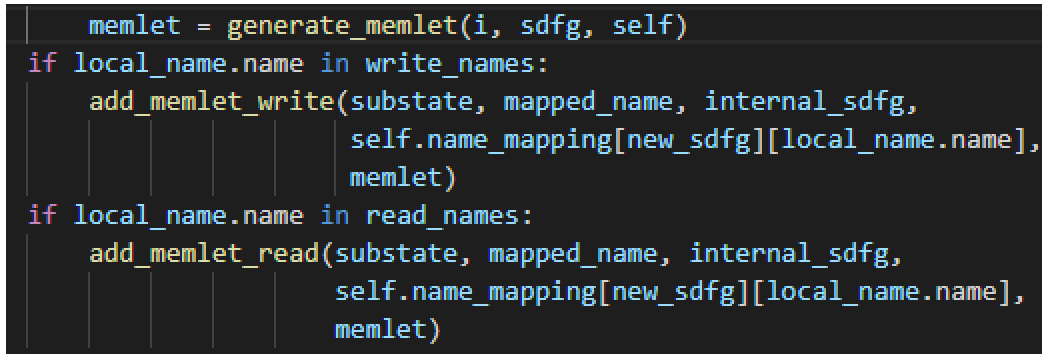
C2DaCe – Functions context change



```
def funcdecl2sdfg(self, node: FuncDecl, sdfg: SDFG):
    print("FUNC: ", node.name)
    if node.body is None:
        print("Empty function")
        return
    used_vars = [
        node for node in walk(node.body) if isinstance(node, DeclRefExpr)
    ]
    binop_nodes = [
        node for node in walk(node.body) if isinstance(node, BinOp)
    ]
    write_nodes = [node for node in binop_nodes if node.op == "="]
    write_vars = [node.lvalue for node in write_nodes]
    read_vars = copy.deepcopy(used_vars)
    for i in write_vars:
        if i in read_vars:
            read_vars.remove(i)
    write_vars = remove_duplicates(write_vars)
    read_vars = remove_duplicates(read_vars)
    used_vars = remove_duplicates(used_vars)
    write_names = []
    read_names = []
    for i in write_vars:
        write_names.append(i.name)
    for i in read_vars:
        read_names.append(i.name)
```



```
internal_sdfg = substate.add_nested_sdfg(new_sdfg,
                                         sdfg,
                                         ins_in_new_sdfg,
                                         outs_in_new_sdfg,
                                         symbol_mapping=sym_dict)
```



```
    memlet = generate_memlet(i, sdfg, self)
if local_name.name in write_names:
    add_memlet_write(substate, mapped_name, internal_sdfg,
                     self.name_mapping[new_sdfg][local_name.name],
                     memlet)
if local_name.name in read_names:
    add_memlet_read(substate, mapped_name, internal_sdfg,
                    self.name_mapping[new_sdfg][local_name.name],
                    memlet)
```

The most interesting bits!