CSC4180 Assignment 4 Report

Abstract

This assignment implements an instance of the last 2 stages front-end phase compilation for Oat v.1 Language. Semantic Analysis is responsible for filling the missing type and scope information, and check the structure of AST. Intermediate Code Generation (IR) is implemented by the Python tool llvmlite to generate LLVM IR.

Implementation

Semantic Analysis Part

The basic idea of Semantic Analysis is traversing AST of the program and handle each type of node separatively.

Take the Program node as the example:

```
def semantic_handler_program(node):
    symbol_table.push_scope()
    # insert built-in function names in global scope symbol table
    symbol_table.insert("array_of_string", DataType.INT_ARRAY)
    symbol_table.insert("string_of_array", DataType.STRING)
    symbol_table.insert("length_of_string", DataType.INT)
    symbol_table.insert("string_of_int", DataType.STRING)
    symbol_table.insert("string_cat", DataType.STRING)
    symbol_table.insert("print_string", DataType.VOID)
    symbol_table.insert("print_int", DataType.VOID)
    symbol_table.insert("print_bool", DataType.BOOL)
    # recursively do semantic analysis in left-to-right order for all children
nodes
    for child in node.children:
        semantic_analysis(child)
    symbol_table.pop_scope()
```

The `push_scope()` and `pop_scope()` method easily creates a scope. Since `symbol_table` is implemented as a stack, we always handle the current scope. For Program node, the current scope is the global scope, therefore we insert the global function into `symbol_table`, noticing that we are working on the current scope. The children of the Program node belongs to the scope, so we continue to handle them.

I will try my best to explain about the detailed implementation of each type of node.

The node can be classified into several sets. The first set is Declaration: GLOBAL_DECL,, FUNC_DECL, VAR_DECLVAR_DECL. The main works are: to unify the data type, to check the uniqueness of the variable, and to insert the variable if true.

The second set is IF, FOR_LOOP and WHILE_LOOP. We need to handle the scope problem. For `FOR` and `WHILE`, we just need to push and pop a new scope. For `IF-ELSE`, the problem is a little bit complex because `IF` and `ELSE` have different scopes but `ELSE` belongs to `IF` node instead of living independently. We just need to check whether the last child of `IF` is `ELSE` to solve the problem.

Code Generation Part

The general idea of IR code generation is similar to Semantic Analysis: handle each type of node one by one.

Still, the nodes can be classified into different sets. The first set is declarations. The works are: to create IR object, to fill the `ir_map` and to use builder to generate the code.

The second set is `for`,`while`,`if`. All of them should handle multiple blocks. Take the simplest `while` structure as the example:

```
def codegen_handler_while(node,builder=None):
    # blocks
    func=builder.function
    loop_cond = func.append_basic_block(name="loop.cond")
    loop_body = func.append_basic_block(name="loop.body")
    loop_end = func.append_basic_block(name="loop.end")

    builder.branch(loop_cond)

    # branch
    builder.position_at_end(loop_cond)
    compare_node=node.children[0]
    condition = codegen_handler_compare(compare_node,builder)
    builder.cbranch(condition, loop_body, loop_end)

    # build then block
    builder.position_at_end(loop_body)
    stmt_node=node.children[1]
    codegen(stmt_node,builder)
    builder.branch(loop_cond)

    builder.position_at_end(loop_end)
```

The WHILE node has a node for compare and a node for other statements. The first step is to build the blocks. For each block we use `branch` or `cbranch` method to jump to the position and build the code.

Problem

Both implementations are not complete. I wrote this program through watching the structure of the cases since I cannot cover all the situation by no instance. Therefore there must be problem where I leave it blank.

For example, the computation of expressions are not complete, it deals with only the simplest case when two integers plus or minus. Multiplication and other arithmetic for integers are easy to implement referring to the implemented part. However, if two strings are added, more considerations should be given to it.

Anyway, it helps to implement according to the visualized tree.