SPLCompiler

Team Members

11812102 Weijie Huang

11810517 Zhaoqi Xiao

11810206 Tao Sun

Design and Implementation

In this project, our compiler will generate a particular intermediate representation (IR) for a given source program. The IR can be further optimized for better runtime performance.

We firstly generate the parse tree and then generate TAC from the generated parse tree by traversing the tree in **post-order**, then convert the tree nodes according to some particular patterns. We use syntax-directed translation to accomplish this task. At the implementation level, we have implemented a set of translate_X functions for each nonterminal X, and invoke these functions recursively, either directly or indirectly.

Basic expression

We use the following functions to generate IR for basic expression.

```
void translate_ExtDefList(tree_node*);
void translate_ExtDef(tree_node*);
void translate_ExtDecList(tree_node*);
void translate_Specifier(tree_node*);
void translate_StructSpecifier(tree_node*);
void translate_Exp(tree_node*, string);
......
```

The implementation detail is similar as the translation schemes given in the document.

Conditional Expressions

We use the following functions to generate IR for conditional expression.

```
void translate_cond_Exp(tree_node*, string, string);
```

The implementation detail is similar as the translation schemes given in the document.

Function Invocations

We use the following functions to generate IR for conditional expression.

```
void translate_Args(tree_node*, vector<string>&);
```

The implementation detail is similar as the translation schemes given in the document.

Derived Data Types

For struct and array we need to add more translation schemes.

For declaration, we use DEC statement with the size of the declared type.

For assignment, we need to modify the translation schemes for basic expression

```
Exp1 ASSIGN Exp2

variable = symtab_lookup(Exp1.ID)
tp = new_place()
code1 = translate_Exp(Exp2, tp)
code2 = [variable.name := tp]
code3 = [place := variable.name]
return code1 + code2 + code3
```

The variable should be replaced with a base address and the offset.

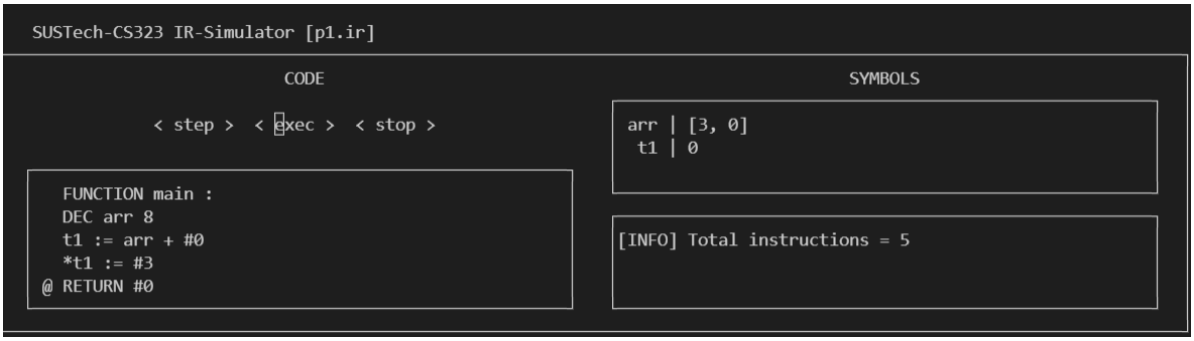
Also, we need to rewrite two rules: Exp -> Exp LB Exp RB and Exp -> Exp DOT ID, both of which will be translated into memory offset calculation.

Problem and Solution

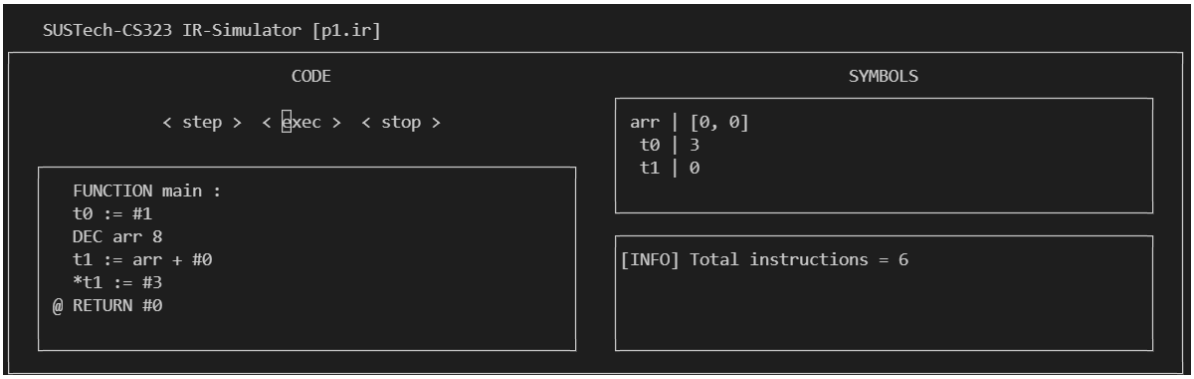
The IR simulator works well with basic test cases. However, when dealing with derived data types, I met some problems

Problem

IR 1 which assigns arr[0] to 3

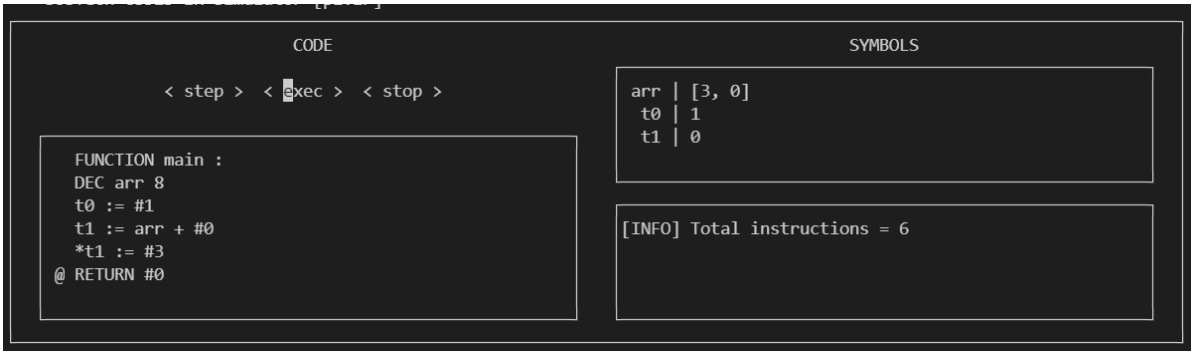


It works well. However, if we initialize a variable before `DEC arr 8`, it will be confused.



We will assign t0 to 3 while the array is unchanged.

If we change the location where we define to, it will work normally.



The problem is similar as buffer overflow.

Solution

We are stuck at the conventional thinking in using cpp. We should use &arr instead of arr when get the address and offset. After modifying the code. The problem is solved.

Summary

We have learned how to generated TAC from the parse tree. We have learned a lot through the process of solving problems encountered during this project.