Project 3 Intermediate Code Generator

11713020 张佳晨

Overview

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Methodology

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.
The Three–Address Code Specification for this project is as below,

Table 5: Three-address-code specification

Instruction	Description
`LABEL x :`	define a label `x`
`FUNCTION f :`	define a function `f`
`x := y`	assign value of `y` to `x`
`x := y + z`	arithmetic addition
`x := y - z`	arithmetic subtraction
`x := y * z`	arithmetic multiplication
`x := y / z`	arithmetic division
`x := &y`	assign address of `y` to `x`
`x := *y`	assign value stored in address `y` to `x`
`*x := y`	copy value `y` to address `x`
`GOTO x`	unconditional jump to label `x`
`IF x [relop] y GOTO z`	if the condition (binary boolean) is true, jump to label `z`
`RETURN x`	exit the current function and return value `x`
`DEC x [size]`	allocate space pointed by `x`, `size` must be a multiple of 4
`PARAM x`	declare a function parameter
`ARG x`	pass argument `x`
`x := CALL f`	call a function, assign the return value to `x`
`READ x`	read `x` from console
`WRITE x`	print the value of `x` to console

And we use the following strategies to convert the CFG into the intermediate code.

For `struct` type or `array` type instructions, the `Exp` is divided into two parts: normal part like before and addressing part (which is used to locate the address of data for computing).

The `struct` type is regard as a special `array` type. In detail element size in `array` are same while may be different for `struct` members, which has great impact on the member/element addressing. So I use one list (vector) to store type of the element and record the size information into the type data structure. Thus, once the indexing of member/element of the struct/array is known, the generator could compute the address shifting by the size info record in the type data structure.

Data Structure

Three address code requires linear or pointer like storage. Besides the data storing in the specific data structure, I need to implement an easy way to convert (or generate) into the three–address code specifications.

Table 5: Three-address-code specification

Instruction	Description
`LABEL x :`	define a label `x`
`FUNCTION f :`	define a function `f`
`x := y`	assign value of `y` to `x`
`x := y + z`	arithmetic addition
`x := y - z`	arithmetic subtraction
`x := y * z`	arithmetic multiplication
`x := y / z`	arithmetic division
`x := &y`	assign address of `y` to `x`
`x := *y`	assign value stored in address `y` to `x`
`*x := y`	copy value `y` to address `x`
`GOTO x`	unconditional jump to label `x`
`IF x [relop] y GOTO z`	if the condition (binary boolean) is true, jump to label `z`
`RETURN x`	exit the current function and return value `x`
`DEC x [size]`	allocate space pointed by `x`, `size` must be a multiple of 4
`PARAM x`	declare a function parameter
`ARG x`	pass argument `x`
`x := CALL f`	call a function, assign the return value to `x`
`READ x`	read `x` from console
`WRITE x`	print the value of `x` to console

Thus, I define 16 different `TAC class` to represent 16 Instructions (4 arithmetic instruction are merged together).

Discussion

I use the semantic tree generated in project2 but found some problems. For example, I have not handled the expression like `if (a == b || c == d)` as there is no priority for `EQ` operator and `OR` operator. Syntax error like this has been fixed in this project.

In project2, as the tree traversal, I use recursive call to do DFS. But in project 3, I need to do some iterative traversal to ensure functionality.

Reference

1. 编译工程9：中间代码生成
2. Compiler – Intermediate Code Generation