Step 6: Semantic Routines (III) - Due Friday, April 15th, 11:59pm

Introduction

Being able to handle function and/or procedure calls is an essential feature of any programming language available these days. At this point you have implemented the semantic routines for control structures and expression handling in your compiler. This step goes further with the implementation of the semantic routines extending them to handle function declarations and calls. At the end of this step your compiler should be able to handle function calls made in programs written in the LITTLE language including the ability of performing recursive function calls.

Overview

Your goal in this step is to augment the compiler to handle functions declarations and function calls. In the end of this step you should be able to generate an executable from a source program that makes calls to declared functions. Your compiler should also handle recursion.

In the previous step, you have created an intermediate representation for a single function 'main" and implemented a code generator that generates "tiny" instructions which could be run on top of tiny simulator. In this step you will generalize the intermediate representation to allow multiple functions and add support for function calls.

In addition to that, your compiler should be able to handle local variables. The previous step had only global variables. Function call creates an activation record and passes parameters and gets return values through the stack. Local variable are also stored on the stack.

Implementation details

You will add code for the function declaration semantic routines. Every function will have its own scope that is also nested to the program main scope. Therefore, the symbols declared in the main program prior to the function declaration list should be visible inside the functions. The function parameters are local symbols to the function first (main) scope and should be visible from inside that function.

Make a global list of function IRs. The intermediate code for each function should be created in a different list. During the code generation step all the list of IRs should be traversed.

In order to support recursion the variables declared local to a function should be referenced as offsets of a base address. In that way when a function calls itself recursively, the local variables will be in different memory positions in each activation record.

You will also have to follow a procedure to make function calls. More detailed descriptions of how to make function calls and store local variables can be found here. The document describes how a function call stack should be set up when a function call happens and how parameters, return value and local variables can be accessed from the stack.

Also note that temp variables are also local to a function so if a temp variable needs to be stored it will be

stored on the stack. The current step does not have such case but in step 7 when you implement register allocation, temp variables can sometimes be spilled and will be stored on the stack. Naming scheme for temp variables in IR nodes for sample outputs are slightly changed in this step. Since temp variables are local to a function, temp numbering is reset to 1 for each new function. Also a prefix "$" is used to make it not conflict with any variable name. The prefix also implies that temp variables are local variables.

Testcases

Test case with non recursive functions.

- fma.micro and the IR list and tiny code fma.output

Test cases with recursive functions.

- factorial2.micro and the IR list and tiny code factorial2.output
- fibonacci2.micro and the IR list and tiny code fibonacci2.output

Tiny Simulator

Continue using the Tiny simulator (tinyR with 1000 registers) that you have used for step 4 and 5.

Submission

Rules for submission is the same as in previous steps. This time the project name is "step6".

Make sure you follow the guideline for creating a Makefile and have correct directory structure for your submission files. You can find details from step1 description page.

References

- **Charles Fischer, Richard LeBlanc, "Crafting a Compiler with C", The Benjamin/Commings Publishing Company, Inc, 1991.**
  Translating Procedures and Functions - Chapter 13