Computer Engineering 175
Phase V: Storage Allocation

"This goodly frame, the earth, seems to me a sterile promontory ..."
Shakespeare, *Hamlet*, Act II

1 Overview

In this assignment, you will extend your compiler to generate code for a 32-bit Intel processor running the Linux operating system. This assignment is worth 10% of your project grade. Your program is due at 11:59 pm, Sunday, February 27th.

2 Storage Allocation

Any variable declared outside of a function has static storage. Such variables and all functions are global symbols. Any variable declared within a function has automatic (i.e., stack-based) storage. For this assignment, the only valid specifiers are `int` and `char`; no structures will be defined or used. The type `int` is a signed type and requires four bytes of storage. The type `char` is a signed type and requires one byte of storage.

The types "pointer to T" and "callback returning T" require four bytes of storage regardless of the type T. The type "array of T" is stored as a consecutive sequence of objects of type T. The total number of bytes of storage is therefore equal to the length of the array times the size of the type T. The first object in an array has index zero, with lower indices having lower addresses than higher indices.

3 Expressions

3.1 Overview

For this assignment, your compiler will see only function calls in which all arguments are either integer literals or scalar variables of type `int`, and the function designator will always be an identifier of type "function returning T."

3.2 Semantic Rules

3.2.1 Postfix expressions

postfix-expression → *postfix-expression* `(` *expression-list* `)`
| *postfix-expression* `(` `)`

For function calls, arguments are passed by value. A function may therefore change the values of its parameters without affecting the values of the arguments. The order of evaluation of arguments is unspecified. Recursive calls to any function are permitted. For this assignment only, each argument will either be an integer literal or a scalar variable of type `int`, and the function designator will always be an identifier of type "function returning T."

4 Function Definitions and Statements

4.1 Overview

A function definition contains a sequence of statements, which are executed in sequence. For this assignment, the statements will be only simple assignment statements (Sec. 4.2.1) or simple function call expressions (Sec. 3.2.1).

4.2 Semantic Rules

4.2.1 Assignment statements

assignment → *expression* = *expression*

The value of the right operand replaces that of the object referred to by the lvalue of the left operand. For this assignment only, the right operand will always be an integer literal, and the left operand will always be a scalar variable of type `int`.

4.2.2 Parameter lists

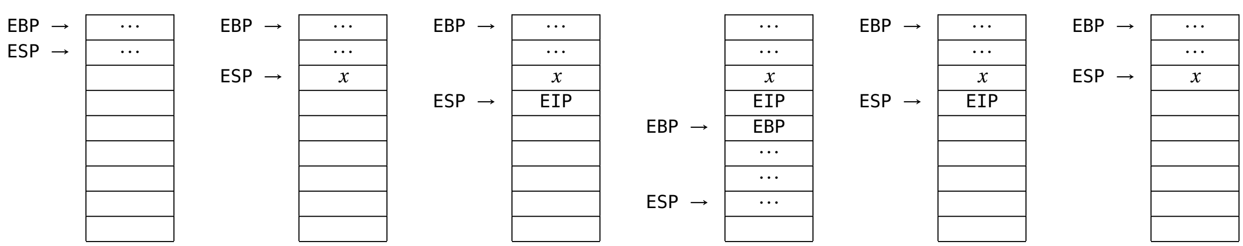
parameter-list → *parameter*
| *parameter* , *parameter-list*

parameter → *specifier pointers* **id**

For this assignment only, all parameters will be scalar variables of type `int`.

5 Stack Frame

Arguments are pushed on the stack from right to left. Therefore, the first argument is on the top of the stack and has the the lowest memory address. The `CALL` instruction pushes the return address on the stack before transferring control. The first task the callee usually does is to use the `ENTER` instruction (or equivalent instructions) to push the old base pointer on the stack, copy the stack pointer to the base pointer, and allocate space by adjusting the stack pointer. Therefore, the first argument is found at eight bytes from the base pointer:



(a) initial stack (b) after `PUSH` *x* (c) after `CALL` *f* (d) after `ENTER 12` (e) after `LEAVE` (f) after `RET`

The callee saves the `EBP`, `ESI`, `EDI`, and `EBX` registers. Other registers are caller-saved. Any return value is stored in the `EAX` register. The callee then uses the `LEAVE` instruction (or equivalent instructions) to deallocate its space and restore the old base pointer. The `RET` instruction is then used to restore the old instruction pointer, but ***not*** to pop the arguments from the stack. The caller is responsible for popping the arguments by adding the appropriate number of bytes to the stack pointer.

6 Assignment

You will write a code generator for Simple C by augmenting your parser, using the given rules as a guide. Your compiler will only be given ***legal programs as input***. Your compiler should write valid Intel assembly code to the ***standard output***. In the previous assignment, error messages were written to the standard error. Therefore, you do not need to change or remove the semantic checks already in place. Your generated assembly code can be assembled and linked using the native compiler along with any additional C files:

```
$ scc < file.c > file.s 2> /dev/null
$ gcc -m32 file.s [additional-source-files]
$ ./a.out
```

Your assembly code will be inspected to check that you are allocating a realistic amount of memory for the stack frames.

7 Hints

You can use the native compiler to determine how to perform most operations. Using the `-S` flag will generate assembly code rather than object code from a source program. You should be aware that the native compiler sometimes uses sophisticated instructions and addressing modes for faster code. However, the correctness of your code is what is important, not its speed, and using simpler instructions may result in code that is easier to understand and debug.