my-zhang add question #4 46d5c9f a minute ago

1 contributor

101 lines (72 sloc) 2.39 KB Raw Blame History

# Assignment 6

## Exercise

Let's consider the following context-free grammar. We are creating a calculator handling identifiers.

```
S -> I; S | ε
I -> ident := E | E | ε
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ident | const | (E)
```

## Question 1

> Propose a simple data structure to allow the compiler to represent and handle internally an assembly code.

I make assembly code stored in a list, which each assembly instruction is a tuple of the format (opcode, oprand1, oprand2).

## Question 2

> Where can you store the variables?

I used a hashmap to track the position of each variable that will be placed on the stack.

For exmaple, if `"a:=6; b:=9; a+b;"`, then the hashmap will be like `{a: -4, b: -8}`, and thus `a` and `b` will locate at `-4(%ebp)` and `-8(%ebp)`.

## Question 3

Implemented in `p3.py`, with lib files in dir `ply`.

**X86-64 Mac** and **gcc** platform required.

Simple example:

```
$ echo "a:=1; a+2;" | python p3.py > a.s
$ gcc a.s
$ ./a.out
```

```
Ans: 3
```

The content of generated assembly file.

```
.globl _main
_main:
pushq %rbp
movq %rsp, %rbp
subq $256, %rsp
pushq $1
popq -8(%rbp)
pushq -8(%rbp)
pushq $2
popq %rbx
popq %rax
cltd
addq %rbx, %rax
pushq %rax
popq %rax
leaq L_.str(%rip), %rdi
movl %eax, %esi
movb $0, %al
callq _printf
addq $256, %rsp
popq %rbp
retq
.section __TEXT,__cstring,cstring_literals
L_.str:
.asciz "Ans: %d\n"
```

A complicated case:

```
$ echo "a:=1; b:=2; (a+b)*(3-4); a+6/3-b;" | python p3.py > a.s
$ gcc a.s
$ ./a.out
Ans: -3
Ans: 1
```

## Question 4

> The assembly code is directly executable? Explain how to obtain an executable?

Yes, as illustrated in question #3.

Here're some experiences and efforts I made to make it excutable on my Mac.

Firstly, I was trying to generate X86-32 code which looks more common and familiar. `gcc` have an option `-m32` to generate 32-bits cross-platform code. It worked fine before introducing `printf`.

However when `printf` introduced, the code will raise error during the linking phase on my macbook, since the `ld` doesn't know where to find a 32-bits library that implements `printf`. I thought about copying a 32-bits printf library somewhere and add it to `ld`'s search path. Since this idea may require some tricky hacks, I decided to turn to X86-64 code.

The differences between X86-32 and X86-64 code are like `pushl` and `pushq`, `%eax` and `%rax`, `addl` and `addq` and etc.