Compilers Principals - Chapter5&6

Zhixin Zhang, 3210106357

Problems：5.1(a,b),6.3,7(a,b)

5.1(a,b)

Improve the hash table implementation of Program 5.2:

a. Double the size of the array when the average bucket length grows larger than 2 (so table is now a pointer to a dynamically allocated array). To double an array, allocate a bigger one and rehash the contents of the old array; then discard the old array.

b. Allow for more than one table to be in use by making the table a param- eter to insert and lookup.

a.

```
#define SIZE 109
#define MAX_LOAD 2
struct bucket **table; unsigned int num_entries = 0, size = SIZE;
void insert(string key, void *binding)
{
  if(table == NULL)
  {
    table = (struct bucket**) malloc (size * sizeof(struct bucket *));
    memset(table, 0, size * sizeof(struct buckt * ));
  }
  int index = hash(key) % size;
  table[index] = Bucket(key, binding, table[index]);
  num_entries ++;

  if(num_entries > MAX_LOAD * size)
  {
    unsigned int new_size = size << 1;
    struct bucket **new_table =
      (struct bucket**) malloc (new_size * sizeof(struct bucket *));
    memset(new_table, 0, new_size * sizeof(struct buckt * ));

    for(unsigned int i = 0; i < size; ++i)
    {
      while(table[i])
      {
        struct bucket* t = table[i];
        table[i] = table[i]->next;
        int new_index = hash(t->key) % new_size;
        new_table[index] = Bucket(t->key, t->binding, new_table[new_index]);
        free(t);
      }
    }
    free(table);
    table = new_table;
  }
}
void pop(string key)
```

```
{
  int index = hash(key) % SIZE;
  table[index] = table[index]->next;
  num_entries --;
}
```

b.

```
typedef struct {
  struct bucket **_table;
  unsigned int size;
} Table;
void insert(string key, void *binding, Table* table)
{
  if(table->size == 0)
  {
    table->size = SIZE;
    table._table = (struct bucket**) malloc (SIZE * sizeof(struct bucket *));
  }
  int index = hash(key) % table->size;
  table->_table[index] = Bucket(key, binding, table->_table[index]);
}
void *lookup(string key, Table* table)
{
  int index = hash(key) % table->size;
  struct bucket* b;
  for(b=table->_table[index]; b; b=b->next)
  {
    if(0 == strcmp(b->key, key)) return b->binding;
  }
  return NULL;
}
```

6.3

For each of the variables a, b, c, d, e in this C program, say whether the variable should be kept in memory or a register, and why.

```
int f(int a, int b)
{
  int c[3], d, e;
  d = a+1;
  e = g(c, &b);
  return e + c[1] + b;
}
```

a:in register. It is the parameters of function f.

b:in memory. Passed by reference, the variable is accessed by a procedure.

c:in memory. It is an array.

d:in register. It is the reusult of expression 'a+1', and there is no use after call the function 'g(c, &b)'.

e:in register. It is the function result of f.

6.7(a,b)

A **display** is a data structure that may be used as an alternatives to static links for maintaining access to nonlocal variables. It is an array of frame pointers, indexed by static nesting depth. Element D_i of the display always points to the most recently called function whose static nesting depth is i.

The bookkeeping performed by a function f, whose static nesting depth is i, looks like:

```
Copy D_i to save location in statck frame
Copy frame pointer to D_i
  ... body of f ...
Copy save location back to D_i
```

In Program 6.3, function prettyprint is at depth 1, write and show are at depth 2, and so on.

a. Show the sequence of machine instructions required to fetch the variable output into a register at line 14 of Program 6.3, using static links.

b. Show the machine instructions required if a display were used instead.

```
1     type tree = {key: string, left: tree, right: tree}
3     function prettyprint(tree: tree) : string =
4       let
5         var output := ""
7         function write(s: string) =
8           output := concat(output,s)
10        function show(n:int, t: tree) =
11          let function indent(s: string) =
12            (for i := 1 to n
13              do write(" ");
14            output := concat(output,s); write("\n"))
15          in if t=nil then indent(".")
16            else (indent(t.key);
17              show(n+1,t.left);
18              show(n+1,t.right))
19          end
21      in show(0,tree); output
22    end
```

a. use static links.

```
MOV R1, FP            # get the frame pointer to the show
MOV R1, [R1 - K]      # fetch show's static link
MOV R1, [R1 - K]      # get the frame pointer to the prettyprint
MOV R2, [R1 + OFFSET] # fetchs output
```

b. use display.

```
# get the D2 (the frame pointer of show), variable 'output' not exists
MOV R1, [DISPLAY + 1 * POINTER_SIZE]  # get D1 (the frame pointer of prettyprint)
MOV R2, [R1 + OFFSET]                 # fetch the variable output.
```