Compiler Principle

Prof. Dongming LU

Mar. 25th, 2024

Content

5 SEMANTIC ANALYSIS

The task of semantic analysis

1. **Connect variable definitions to their uses ;**

2. **Check if each expression has a correct type ;**

3. **Translates the abstract syntax into a simpler representation suitable for generating machine code.**

5.1 Symbol Table

Maintenance of symbol tables mapping identifiers to their types and locations

Environment definition

- An environment is **a set of *bindings***
  - ✓ denoted by the ↦ arrow

```
1. function f( a:int ,b:int, c:int) =
2.    {  print_int (a+c);
3.          let var j:= a+b
4.             var a:= “hello”
5.          in print(a);  print_int(j)
6.          end;
7.         print_int(b)
8.     }
```

line 1:
σ1 equal to σ 0 + {a ↦ int, b ↦ int, c ↦ int}

line 3:
the table σ 2 = σ 1 + {j ↦ int}

line 4:
the table σ 3 = σ 2 + {a ↦ string}

Scope: where the identifiers are visible

Environment definition

- **An environment is *a set of bindings***
  - ✓ **denoted by the $\mapsto$ arrow**

```
1. function f( a:int ,b:int, c:int) =
2.    {  print_int (a+c);
3.         let var j:= a+b
4.            var a:= "hello"
5.         in print(a);  print_int(j)
6.         end;
7.        print_int(b)
8.     }
```

line 1:
$\sigma 1$ equal to $\sigma 0 + \{a \mapsto int, b \mapsto int, c \mapsto int\}$

line 3:
the table $\sigma 2 = \sigma 1 + \{j \mapsto int\}$

line 4:
the table $\sigma 3 = \sigma 2 + \{a \mapsto string\}$

$X + Y$ for tables is not the same as $Y + X$

Environment management

A functional style

- To keep σ_1 in pristine condition while creating create σ_2 and σ_3

An imperative style

- Modify σ 1 until it becomes σ 2.
- While σ 2 exists, we cannot look things up in σ 1.
- When done with σ 2, can undo the modification to get σ 1 back again.

Either the functional or imperative style of environment management can be used.

MULTIPLE SYMBOL TABLES

- There can be **several active environments** at once;
- Each module, or class has a **symbol table σ** of its own.

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
 }
class D {
    static int d = E.a + N.a;
 }
```

Java

$\sigma 1 = \{ a \mapsto \mathit{int} \}$

$\sigma 2 = \{ E \mapsto \sigma 1 \}$

$\sigma 3 = \{ b \mapsto \mathit{int} , a \mapsto \mathit{int} \}$

$\sigma 4 = \{ N \mapsto \sigma 3 \}$

$\sigma 5 = \{ d \mapsto \mathit{int} \}$

$\sigma 6 = \{ D \mapsto \sigma 5 \}$

$\sigma 7 = \sigma 2 + \sigma 4 + \sigma 6$

In Java, **forward reference** is allowed so *E*, *N*, and *D* are all compiled in the environment σ_7; for this program the result is still $\{M \mapsto \sigma_7\}$.

MULTIPLE SYMBOL TABLES

- There can be several active environments at once;
- Each module, or class has a symbol table σ of its own.

```
structure M =
 struct
  structure E =
 struct
    val a = 5;
  end
  structure N =
 struct
    val b = 10
    val a = E.a + b
  end
  structure D =
 struct
```

ML

$\sigma 1 = \{ a \mapsto \mathit{int} \}$

$\sigma 2 = \{ E \mapsto \sigma 1 \}$

$\sigma 3 = \{ b \mapsto \mathit{int} , a \mapsto \mathit{int} \}$

$\sigma 4 = \{ N \mapsto \sigma 3 \}$

$\sigma 5 = \{ d \mapsto \mathit{int} \}$

$\sigma 6 = \{ D \mapsto \sigma 5 \}$

$\sigma 7 = \sigma 2 + \sigma 4 + \sigma 6$

The *N* is compiled using environment $\sigma_0 + \sigma_2$. *D* is compiled using $\sigma_0 + \sigma_2 + \sigma_4$, the result of the analysis is $\{M \mapsto \sigma_7\}$.

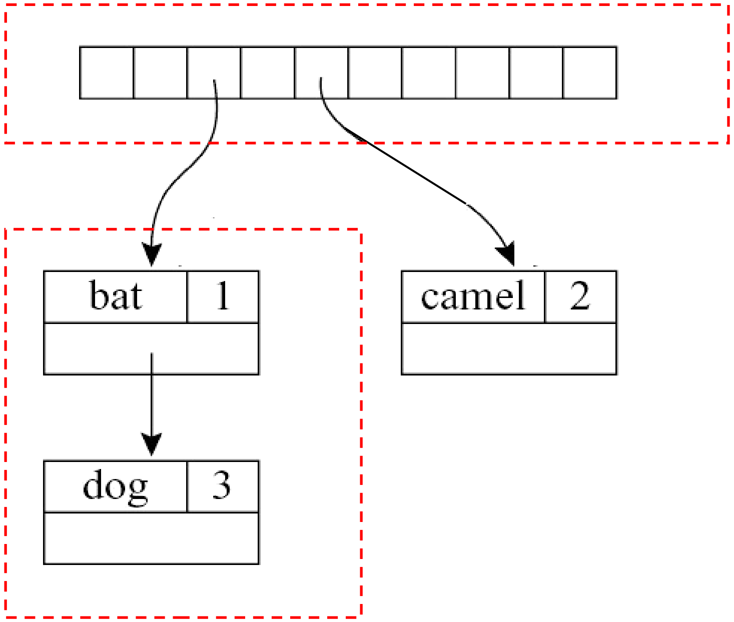
EFFICIENT IMPERATIVE

- Using hash tables to implement the Imperative environments efficiently.

M1 = { bat ↦ 1,camel ↦ 2,dog ↦3 }

Table

Bucket list



EFFICIENT IMPERATIVE

```
struct bucket { string key; void *binding; struct bucket
  *next; };

#define SIZE 109

struct  bucket *table[SIZE];

unsigned int hash(char *s0)
 { unsigned int h=0;  char *s;
   for (s=s0; *s; s++)
         h=h*65599 + *s;
         return h;
}
struct bucket *Bucket (string key, void *binding, struct
  bucket *next) {
   struct bucket *b=check_malloc(sizeof(*b));
   b->key = key; b->binding = binding; b->next = next;
```

$$\boxed{h = (\alpha^{n-1}c_1+\alpha^{n-2}c_2 +.....+ \alpha\, c_{n-1}+ c_n)}$$

EFFICIENT IMPERATIVE

```
void insert(string key, void *binding) {
    int index=hash(key)%SIZE ;
    table[index]=Bucket( key, binding,
  table[index]); }

void *lookup(string key) {
     int index=hash(key)%SIZE
     struct bucket *b;
    for (b = table[index]; b; b=b->next)
          if (0==strcmp(b->key,key)) return b-
  >binding;
    return NULL; }

 void pop( string key) {
     int index=hash(key)%SIZE
     table[index]=table[index].next;  }
```

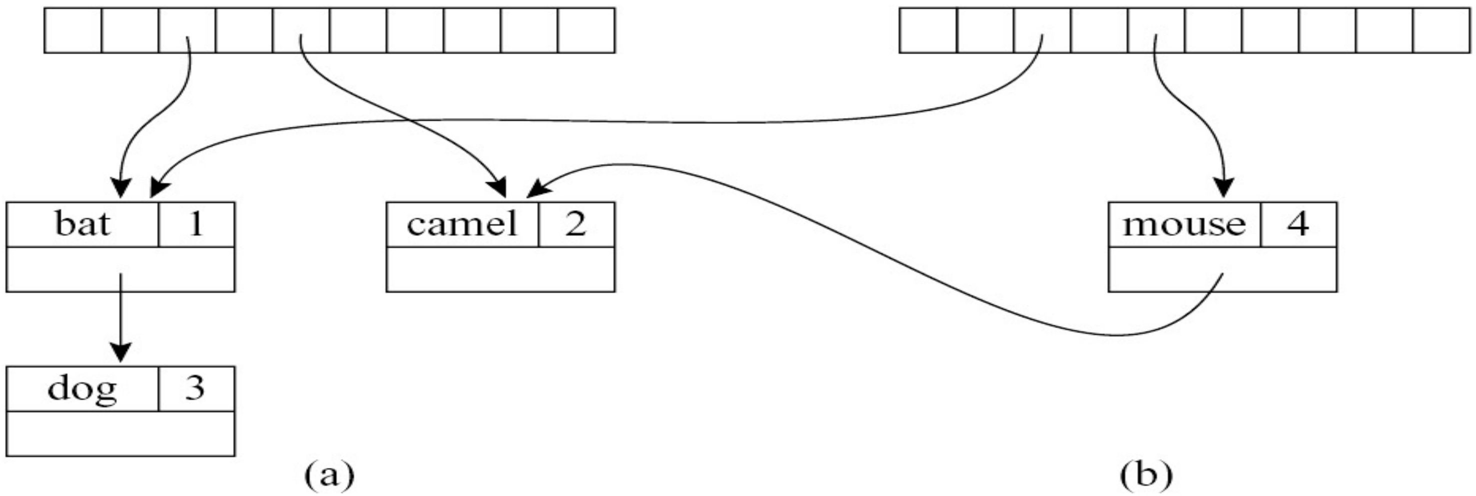
EFFICIENT IMPERATIVE

Consider $\sigma + \{a \mapsto \tau_2\}$ when σ contains $a \mapsto \tau_1$ already.

- **The insert function leaves $a \mapsto \tau_1$ in the bucket and puts $a \mapsto \tau_2$ earlier in the list.**
- **When pop(a) is done at the end of a's scope, σ is restored.**
  (insertion and pop work in a stack-like fashion.)

EFFICIENT FUNCTIONAL

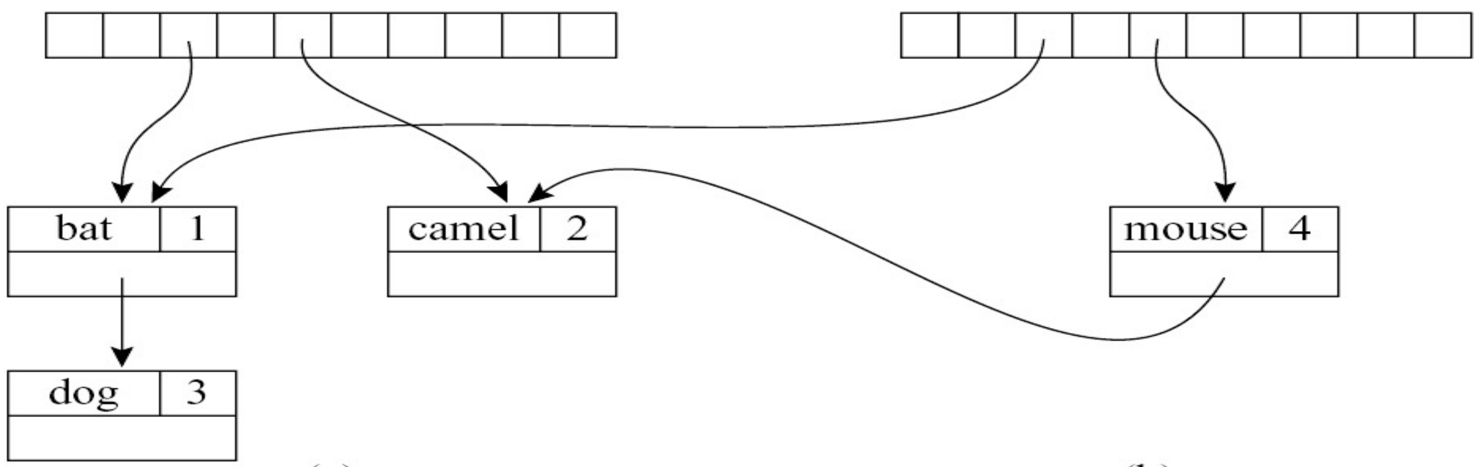
M1 = { bat ↦ 1,camel ↦ 2,dog ↦3 }
add the binding mouse ↦4



EFFICIENT FUNCTIONAL

- **Compute $\sigma' = \sigma + \{a \mapsto \tau\}$ in such a way that still have σ available to look up identifiers.**

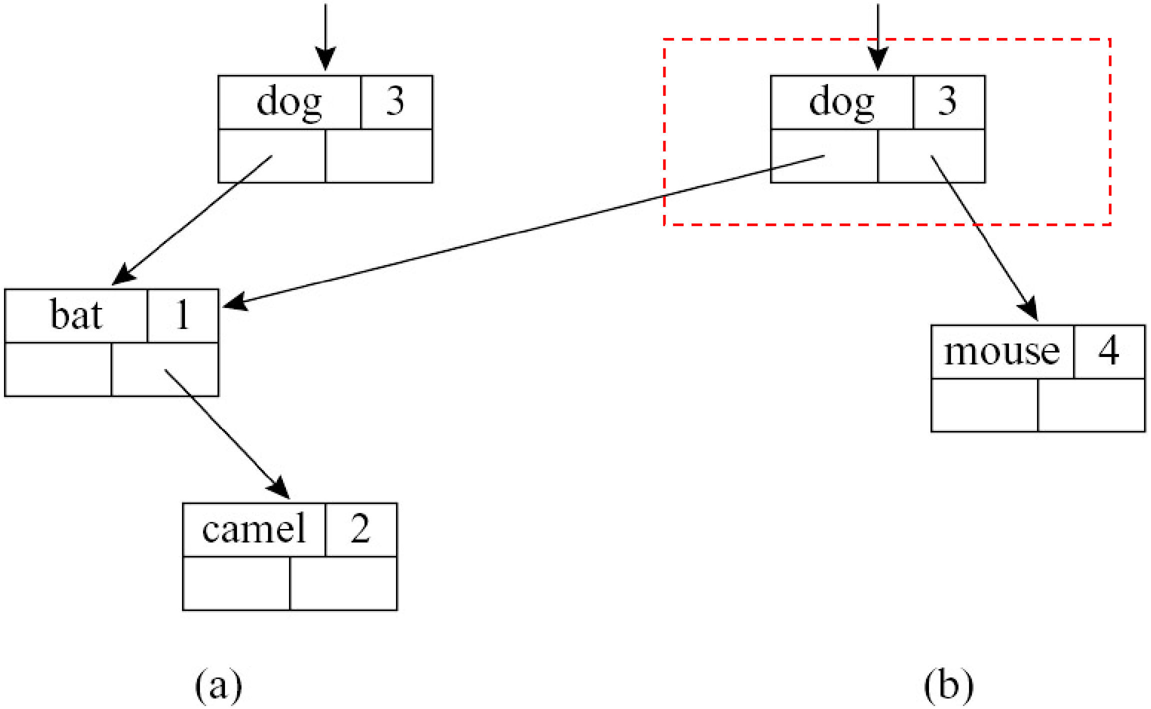
- **Create a new table by computing the "sum" of an existing table and a new binding.**



(a) (b)

EFFICIENT FUNCTIONAL

- **Add a new node at depth *d* of the tree, we must create *d* new nodes —— don't need to copy the whole tree.**



SYMBOLS IN THE Tiger COMPILER

- Convert each string to a symbol to avoid unnecessary string comparisons

$$h = (\alpha^{n-1}c_1+\alpha^{n-2}c_2 + \ldots + \alpha\, c_{n-1}+c_n)$$

- The Symbol module has these important properties:
  - ✓ Comparing symbols for equality is fast (just pointer or integer comparison).
  - ✓ Comparing two symbols for "greater-than" (in some arbitrary ordering) is fast (in case we want to make binary search trees).
  - ✓ Extracting an integer hash key is fast (in case we want to make a hash table mapping symbols to something else).

SYMBOLS IN THE Tiger COMPILER

- The interface of symbols and symbol tables

```
typedef struct S_symbol_  *S_symbol;
S_symbol S_symbol (string);
string  S_name(S_symbol);

typedef struct TAB_table_ *S_table;
S_table S_empty( void);
void S_enter( S_table t, S_symbol sym, void *value);
void *S_look( S_table t, S_symbol sym);

void S_beginScope( S_table  t);
void S_endScope( S_table t);
```

SYMBOLS IN THE Tiger COMPILER

```
static S_symbol mksymbol (string name , S_symbol
 next)
{
    S_symbol s = checked_malloc(sizeof(*s));
    s->name = name;  s->next = next;
    return s;
}

S_symbol S_symbol (string name) {
   int index = hash(name)%SIZE;
   S_symbol syms = hashtable[index], sym;

   for ( sym = syms; sym; sym = sym->next)
       if (0 == strcmp(sym->name, name)) return sym;

   sym = mksymbol(name,syms);
   hashtable[index] = sym;
   return sym;
}
```

SYMBOLS IN THE Tiger COMPILER

```
string  S_name (S_symbol sym)  {
                                                   return sym->name;
                                                   }



S_table S_empty(void) {return TAB_empty();}

void S_enter(S_table t, S_symbol sym, void *value)
  {TAB_enter(t, sym, value);}

void *S_look(S_table t, S_symbol sym) {return
  TAB_look(t, sym);}
```

SYMBOLS IN THE Tiger COMPILER

Use **destructive-update** environment:

- **S_beginScope:** remembers the current state of the table
- **S_endScope:** restores the table to where it was at the most recent beginScope that has not already been ended.

```
static struct S_symbol_   marksym =
  {  "<mark>", 0 };

void S_beginScope ( S_table  t)
   { S_enter(t, &marksym, NULL);   }

void S_endScope( S_table t)   {
    S_symbol s;
    do  s= TAB_pop(t); while (s !=
  &marksym);
```

SYMBOLS IN THE Tiger COMPILER

Auxiliary stack

- **Showing in what order the symbols were "pushed" into the symbol table.**
- **As each symbol is popped, the head binding in its bucket is removed.**

A global variable TOP

- **showing the most recent Symbol bound in the table.**
- **"pushing": copy TOP into the prevtop field of the Binder.**
- **"stack" is threaded through the binders.**

The end of Chapter 5(1)