Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

6 Symbol tables **1**
6.1 Introduction : 1
6.2 Symbol table design and interface 2
6.3 Implementing symbol tables 3
6.4 Block-structure, scoping, binding, name-space organization 9
6.5 Symbol tables as attributes in an AG 14

Chapter 6
Symbol tables

What is it about?

Learning Targets of this Chapter

1. symbol table data structure
2. design and implementation choices
3. how to deal with scopes
4. connection to attribute grammars

Contents

6.1 Introduction : 1
6.2 Symbol table design and interface 2
6.3 Implementing symbol tables 3
6.4 Block-structure, scoping, binding, name-space organization 9
6.5 Symbol tables as attributes in an AG 14

6.1 Introduction :

Symbol tables, in general

- **central** data structure
- "data base" or repository associating properties with "names" (identifiers, symbols)[1]
- **declarations**
  - constants
  - type declarationss
  - variable declarations
  - procedure declarations
  - class declarations
  - ...
- *declaring* occurrences vs. *use* occurrences of names (e.g. variables)

Does my compiler need a symbol table?

- goal: associate attributes (properties) to syntactic elements (names/symbols)
- storing once calculated: (costs memory) ↔ recalculating on demand (costs time)
- most often: **storing** preferred
- but: can't one store it in the nodes of the *AST*?

[1] Remember the (general) notion of "attribute".

  - remember: attribute grammar
  - however, fancy attribute grammars with many rules and complex synthesized/inherited attribute (whose evaluation traverses up and down and across the tree):
    * might be intransparent
    * storing info *in* the tree: might not be efficient

⇒ central repository (= **symbol table**) better

So: do I need a symbol table?

In theory, alternatives exists; in practice, yes, symbol tables is the way to go; most compilers do use symbol tables.

Further side remarks

Often (and in our course), the symbol table is set up once, containing all the symbols that occur in a given program, and then the semantics analyses (type checking etc) update the table accordinging. Implicit in that is that the symbol table is *static*. There are also some languages, which allow "manipulation" of symbol tables at *run time* (Racket is one (formally PLT scheme)).

In the slides, a point was made that basically every compiler has a symbol table (or even more than one). You find statements in the internet that symbol tables are not needed or even to be avoided. For instance "no symbol tables in Go" claims that there are no symbol tables in Go (and in functional languages). It's not clear how reliable that information is, because here's a link `https://golang.org/pkg/debug/gosym/` to the official go implementation, referring to symbol tables .

6.2 Symbol table design and interface

Symbol table as abstract data type

- separate **interface** from implementation
- ST: "nothing else" than a lookup-table or *dictionary*,
- associating "keys" with "values"
- here: keys = names (id's, symbols), values the attribute(s)

Schematic interface: two core functions (+ more)

- *insert*: add new binding
- *lookup*: retrieve

besides the core functionality:

- structure of (different?) *name spaces* in the implemented language, *scoping* rules
- typically: not one single "flat" namespace ⇒ typically not one big *flat* look-up table

⇒ influence on the design/interface of the ST (and indirectly the choice of implementation)
- necessary to "delete" or "hide" information (*delete*)

A symbol table is, typically, not just a "flat" dictionary, and that is the case neither conceptually nor the way it's implemented. *Scoping* typically is something that often complicates the design of the symbol table.

Two main philosophies

traditional table(s)

- central repository, separate from AST
- interface
  - $lookup(name)$,
  - $insert(name, decl)$,
  - $delete(name)$
- last 2: update ST for declarations *and* when entering/exiting *blocks*

decls. in the AST nodes

- do look-up ⇒ tree-*search*
- insert/delete: implicit, depending on relative positioning in the tree
- look-up:
  - efficiency?
  - however: optimizations exist, e.g. "redundant" extra table (similar to the traditional ST)

Here, for concreteness, *declarations* are the attributes stored in the ST. In general, it is not the only possible stored attribute. Also, there may be more than one ST.

6.3 Implementing symbol tables

Data structures to implement a symbol table

- different ways to implement *dictionaries* (or look-up tables etc.)
  - simple (association) lists
  - trees
    * balanced (AVL, B, red-black, binary-search trees)
  - association list
  - **hash** tables, often method of choice
  - functional vs. imperative implementation
- careful choice influences efficiency
- influenced also by the language being implemented,

- in particular, by its **scoping** rules (or the structure of the name space in general) etc.[2]

Nested block / lexical scope

for instance: *C*

```
{ int i; ... ; double d;
  void p(...);
  {
     int i;
     ...
  }
  int j;
  ...
```

more later

Blocks in other languages

TEX

```
\def\x{a}
{
  \def\x{b}
  \x
}
\x
\bye
```

LATEX

```
\documentclass{article}
\newcommand{\x}{a}
\begin{document}
\x
{\renewcommand{\x}{b}
  \x
}
\x
\end{document}
```

But: static vs. dynamic binding (see later)

LATEX and TEX are chosen for easy trying out the result oneself (assuming that most people have access to LATEX and by implication, TEX). TEX is the underlying "core" on which LATEX is put on top. There are other formats in top of TEX (`texi` is another one; `texi` is involved, for instance, type setting the pdf version of the Compila language specification)

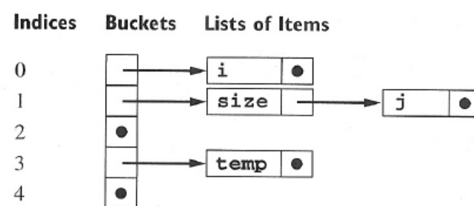
[2] Also the language used for implementation (and the availability of libraries therein) may play a role (but remember "bootstrapping")

Hash tables

- classical and common implementation for STs
- “hash table”:
  - generic term itself, different general forms of HTs exists
  - e.g. *separate chaining* vs. *open addressing*

There exists alternative terminology (cf. INF2220), under which separate chaining is also known as *open hashing*. The *open addressing* methods are also called *closed hashing*. It’s confusing, but that’s how it is, and it’s just words.

Separate chaining



Code snippet

```
{
  int temp;
  int j;
  real i;
  void size (....) {
    {
        ....
    }
  }
}
```

Block structures in programming languages

- almost no language has one global namespace (at least not for variables)
- pretty old concept, seriously started with ALGOL60

Block

- “region” in the program code
- delimited often by `{` and `}` or `BEGIN` and `END` or similar
- organizes the **scope** of declarations (i.e., the name space)
- can be **nested**

Block-structured scopes (in C)

```
int i, j;

int f(int size)
{ char i, temp;
   ...
  { double j;
     ..
  }
   ...
  { char * j;
     ...
  }
}
```

Nested procedures in Pascal

```
program Ex;
var i,j : integer

function f(size : integer) : integer;
var i, temp : char;
   procedure g;
   var j : real;
   begin
      ...
   end;
   procedure h;
   var j : ^char;
   begin
      ...
   end;

begin (* f's body *)
 ...
end;
begin  (* main program *)
   ...
end.
```

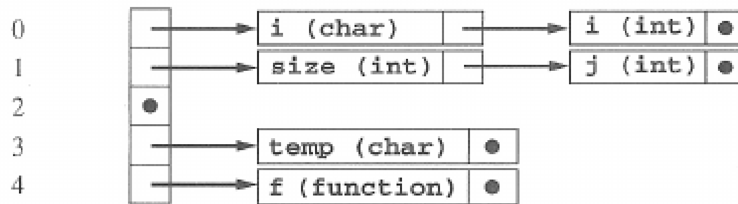
Block-strucured via stack-organized separate chaining

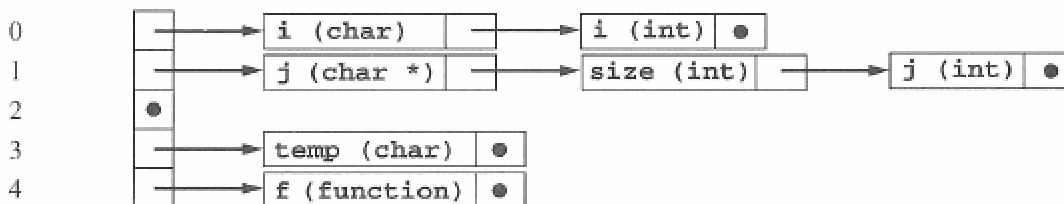
C code snippet

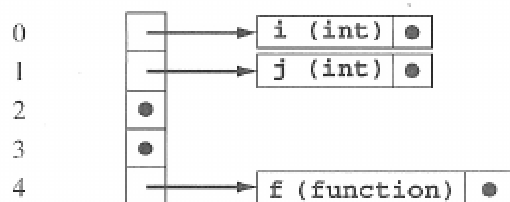
```
int i, j;

int f(int size)
{ char i, temp;
   ...
  { double j;
     ..
```

```
  }
  ...
  { char * j;
    ...
  }
}
```

"Evolution" of the hash table

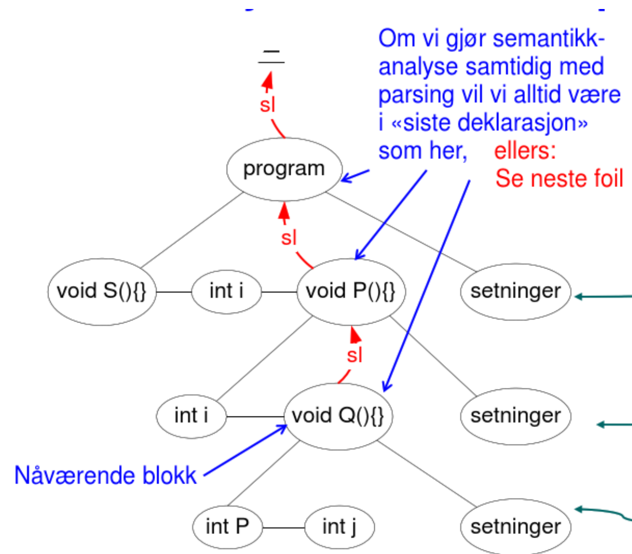






Using the syntax tree for lookup following (static links)

```
lookup (string n) {
   k = current, surrounding block
   do              // search for n in  decl for block k;
      k = k.sl     // one nesting level  up
   until found or k == none
}
```



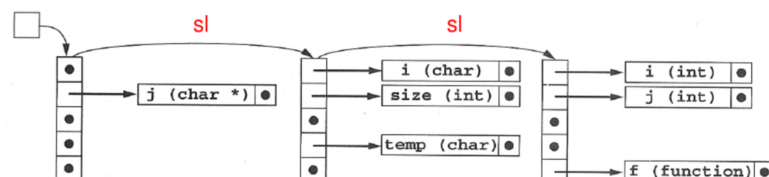
The notion of *static link* will be discussed later, there in connection with the so-called run-time system and the run-time *stack*. There we go into more details, but the idea is the same as here: find a way to "locate" the relevant scope. If they are nested, connect them via come "parent pointer", and that pointer is known as static links (again, different names exists for that, unfortunately).

Alternative representation:

- arrangement different from 1 table with stack-organized external chaining
- each *block* with its **own** hash table.[3]
- standard hashing within each block
- **static links** to link the block levels

⇒ "tree-of-hashtables"

- AKA: *sheaf-of-tables* or *chained symbol tables* representation



[3]One may say: one *symbol table* per block, as this form of organization can generally be done for symbol tables data structures (where hash tables is just one of many possible implementing data structures).

6.4 Block-structure, scoping, binding, name-space organization

Block-structured scoping with chained symbol tables

- remember the *interface*
- look-up: following the static link (as seen)[4]
- **Enter** a block
  - create new (empty) symbol table
  - set static link from there to the "old" (= previously current) one
  - set the current block to the newly created one
- at **exit**
  - move the *current block* one level up
  - note: no *deletion* of bindings, just made *inaccessible*

Lexical scoping & beyond

- block-structured lexical scoping: **central** in programming languages (ever since ALGOL60 ...)
- but: other scoping mechanism exists (and exist side-by-side)
- example: C^{++}
  - member functions *declared* inside a class
  - *defined* outside
- still: method supposed to be able to access names defined in the *scope of the class* definition (i.e., other members, e.g. using `this`)

C^{++}class and member function

```
class A {
  ... int f(); ... // member function
}

A::f() {}     // def. of f ``in'' A
```

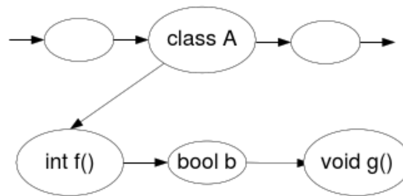
Java analogon

```
class A {
    int f() {...};
    boolean b;
    void h() {...};
}
```

[4]The notion of static links will be encountered later again when dealing with *run-time* environments (and for analogous purposes: identfying scopes in "block-stuctured" languages).

Scope resolution in C^{++}

- class *name* introduces a **name for the scope**[5] (not only in C^{++})
- scope resolution operator `::`
- allows to explicitly refer to a "scope"'
- to implement
  - such flexibility,
  - also for *remote access* like `a.f()`
- declarations must be kept separately for each block (e.g. one hash table per class, record, etc., appropriately chained up)



Same-level declarations

Same level

```
typedef int i
int i;
```

- often forbidden (e.g. in C)
- *insert*: requires check (= *lookup*) first

Sequential vs. "collateral" declarations

1. Sequential in C

```
int i = 1;
void f(void)
  { int i = 2, j = i+1,
    ...
  }
```

2. Collateral in ocaml/ML/Lisp

```
let i = 1;;
let i = 2 and y = i+1;;

print_int(y);;
```

I think the name "collateral" infortunate. A better word would be *simultaneous*.

[5]Besides that, class names themselves are subject to scoping themselves, of course ...

Recursive declarations/definitions

- for instance for functions/procedures
- also classes and their members

Direct recursion

```
int gcd(int n, int m) {
  if   (m == 0) return n;
  else return gcd(m,n % m);
}
```

Indirect recursion/mutual recursive def's

```
void f(void) {
     ... g() ... }
void g(void) {
  ... f() ...}
```

Before treating the body, parser must add `gcd` into the symbol table (similar for the other example).

Mutual recursive defintions

```
void g(void);  /* function prototype decl. */

void f(void) {
     ... g() ... }
void g(void) {
  ... f() ...}
```

- different solutions possible
- Pascal: *forward declarations*
- or: treat all function definitions (within a block or similar) as mutually recursive
- or: special grouping syntax

Example syntax-es for mutual recursion

ocaml

```
let rec  f (x:int): int  =
     g(x+1)
and g(x:int) : int =
     f(x+1);;
```

Go

```
func f(x int) (int) {
        return g(x) +1
}

func g(x int) (int) {
        return f(x) -1
}
```

Static vs dynamic scope

- concentration so far on:
  - lexical scoping/block structure, static binding
  - some minor complications/adaptations (recursion, duplicate declarations, ...)
- **big** variation: **dynamic** binding / **dynamic scope**
- for variables: *static* binding/ *lexical scoping* the norm
- however: cf. late-bound methods in OO

Static scoping in C

Code snippet

```
#include <stdio.h>

int i = 1;
void f(void) {
  printf("%d\n",i);
}


void main(void) {
  int  i = 2;
  f();
  return 0;
}
```

- which value of `i` is printed then?

Dynamic binding example

```
1  void Y () {
2    int i;
3    void P() {
4      int i;
5      ...;
6      Q();
7    }
8    void Q(){
9      ...;
10     i = 5;  // which i is meant?
11   }
12   ...;
13
```

```
14    P();
15    ...;
16  }
```

for dynamic binding: the one from line 4

Static or dynamic?

TeX

```
\def\astring{a1}
\def\x{\astring}
\x
{
  \def\astring{a2}
  \x
}
\x
\bye
```

LaTeX

```
\documentclass{article}
\newcommand{\astring}{a1}
\newcommand{\x}{\astring}
\begin{document}
\x
{
  \renewcommand{\astring}{a2}
  \x
}
\x
\end{document}
```

emacs lisp (not Scheme)

```
(setq astring "a1")  ;; ``assignment''
(defun x() astring)  ;; define ``variable x''
(x)                  ;;  read value
(let ((astring "a2"))
      (x))
```

Again, it's very easy to check by invoking TeX or LaTeX, or firing off emacs and evaluate the lisp snippet in a buffer, for instance.

Static binding is not about "value"

- the "static" in static binding is about
  - binding to the declaration / memory location,
  - not about the *value*

- nested functions used in the example (Go)
- `g` declared inside `f`

```
package main
import ("fmt")

var f = func ()    {
  var x = 0
  var g = func() {fmt.Printf(" x = %v", x)}
  x = x + 1
    {
      var x = 40                    // local variable
      g()
      fmt.Printf(" x = %v", x)}
}
func main() {
  f()
}
```

Static binding can be come tricky

```
package main
import ("fmt")

var f = func () (func (int) int)  {
        var x = 40                    // local variable
        var g = func (y int) int { // nested function
                return x + 1
        }
        x = x+1                       // update x
        return g                      // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf(" r = %v", r)
}
```

- example uses *higher-order* functions

6.5 Symbol tables as attributes in an AG

Nested lets in ocaml

```
let x = 2 and y = 3 in
  (let x = x+2 and y =
       (let z = 4 in x+y+z)
   in print_int (x+y))
```

- simple grammar (using , for “collateral” = simultaneous declarations)

$$
\begin{array}{rcl}
S & \rightarrow & exp \\
exp & \rightarrow & (\, exp\,) \mid exp + exp \mid \textbf{id} \mid num \mid \textbf{let}\ dec\text{-}list\ \textbf{in}\ exp \\
dec\text{-}list & \rightarrow & dec\text{-}list\ ,\ decl \mid decl \\
decl & \rightarrow & \textbf{id} = exp
\end{array}
$$

1. no identical names in the same let-block
2. used names must be declared
3. most-closely nested binding counts
4. sequential (non-simultaneous) declaration ($\neq$ ocaml/ML/Haskell ...)

```
let x = 2, x = 3 in x + 1        (* no, duplicate  *)

let x = 2  in x+y                (* no, y unbound *)

let x = 2 in (let x = 3 in x)    (* decl. with 3 counts *)

let x = 2, y = x+1               (* one after the other *)
in  (let x = x+y,
         y = x+y
     in y)
```

Goal

Design an *attribute grammar* (using a *symbol table*) specifying those rules. Focus on: error attribute.

Attributes and ST interface

symbol	attributes	kind
exp	`symtab`	inherited
	`nestlevel`	inherited
	err	synthesis
dec-list, *decl*	`intab`	inherited
	`outtab`	synthesized
	`nestlevel`	inherited
id	`name`	injected by scanner

Symbol table functions

- `insert(tab,name,lev)`: returns a new table
- `isin(tab,name)`: boolean check
- `lookup(tab,name)`: gives back *level*
- `emptytable`: you have to start somewhere
- `errtab`: error from declaration (but not stored as attribute)

As for the information stored and especially for the look-up function: Realistically, more info would be stored, as well, for instance types etc.

Attribute grammar (1): expressions

Grammar Rule	Semantic Rules
$S \to exp$	$exp.symtab$ = $emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \to exp_2$ **+** exp_3	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \to$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \to$ **id**	$exp.err$ = **not** $isin(exp.symtab,$**id**$.name)$ } 2
$exp \to$ **num**	$exp.err$ = **false**
$exp_1 \to$ **let** $dec\text{-}list$ **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ } 3 $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$

- note: expression in let's can introduce scope themselves!
- interpretation of nesting level: expressions vs. declarations[6]

Attribute grammar (2): declarations

Grammar Rule	Semantic Rules
$dec\text{-}list_1 \to dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ } 4 $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \to decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ } 4 $dec\text{-}list.outtab = decl.outtab$
$decl \to$ **id** **=** exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab$ = **if** $(decl.intab = errtab)$ **or** $exp.err$ **then** $errtab$ **else if** $(lookup(decl.intab,$**id**$.name) =$ $decl.nestlevel)$ } 1 **then** $errtab$ **else** $insert(decl.intab,$**id**$.name, decl.nestlevel)$

[6] I would not have recommended doing it like that (though it works)

Final remarks concerning symbol tables

- *strings* as symbols i.e., as keys in the ST: might be improved
- name spaces can get complex in modern languages,
- more than one "hierarchy"
  - lexical blocks
  - inheritance or similar
  - (nested) modules
- not all bindings (of course) can be solved at compile time: *dynamic binding*
- can e.g. variables and types have same name (and still be distinguished)
- *overloading* (see next slide)

Final remarks: name resolution via overloading

- corresponds to "in abuse of notation" in textbooks
- disambiguation not by name, but differently especially by "argument types" etc.
- variants :
  - method or function overloading
  - operator overloading
  - user defined?

```
i + j    // integer addition
r + s    // real-addition

void f(int i)
void f(int i,int j)
void f(double r)
```

Bibliography

Bibliography

[1] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[2] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.