INF5110 – Compiler Construction

Symbol tables

Spring 2016

Outline

1. Symbol tables

Introduction
Symbol table design an interface
Implementing symbol tables
Block-structure, scoping, binding, name-space organization
Symbol tables as attributes in an AG

1. Symbol tables
   Introduction
   Symbol table design an interface
   Implementing symbol tables
   Block-structure, scoping, binding, name-space organization
   Symbol tables as attributes in an AG

Symbol tables, in general

- central data structure
- "data base" or repository associating properties with "names" (identifiers, symbols)
- 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) $\leftrightarrow$ recalculating on demand (costs time)
- most often: storing prefered
- but: can't one store it in the nodes of the *AST*?
  - 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

$\Rightarrow$ central repository (= symbol table) better.

So: do I need a symbol table?

In theory, alternatives exists; in practice, yes, symbol tables needed; most compilers do use symbol tables.

Outline

1. Symbol tables
    - Introduction
    - Symbol table design an interface
    - Implementing symbol tables
    - Block-structure, scoping, binding, name-space organization
    - Symbol tables as attributes in an AG

Symbol table as abstract date type

- separate interface from implementation
- ST: basically 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[1]
- ⇒ influence on the design/interface of the ST (and indirectly the choice of implementation)
- necessary to "delete" or "hide" information (*delete*)

[1] Neither conceptually nor the way it's implemented.

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*

declarations in the AST nodes

- to look-up ⇒ tree- *search*
- insert/delete: implicit, depending on relative positioning in the tree
- look-up:
  - potential lack of 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. It's not the only possible attribute stored. There may also be more than one ST.

Outline

1. Symbol tables
    - Introduction
    - Symbol table design an interface
    - Implementing symbol tables
    - Block-structure, scoping, binding, name-space organization
    - Symbol tables as attributes in an AG

- different ways to implement *dictionaries* (or look-up tables etc)
    - simple (association) lists
    - trees
        - balanced (AVL, B, red-black, binary-search trees)
    - hash tables, often method of choice
    - functional vs. imperative implementation
- careful choice influences efficiency
- influenced also by the language being implemented,
- in particular, its scoping rules (or the structure of the name space in general) etc.[2]

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

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

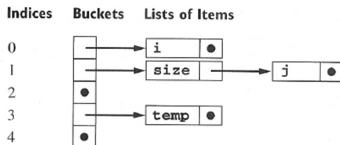
```
\documentclass{article}
\newcommand{\x}{a}
\begin{document}
\x
{\renewcommand{\x}{b}
  \x
}
\end{document}
```

But: static vs. dynamic binding (see later)

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*[3]

Separate chaining



Code snippet

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

[3]There is alternative terminology (cf. INF2220), under which separate chaining is also known as *open hashing*. The *open addressing* methods are also called *closed hashing*. That's how it is.

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`
- used to organize the scope of declarations (i.e., the name space)
- nested blocks

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.
```

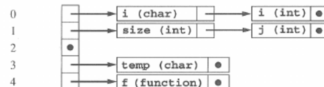
Block-strucured via stack-organized separate chaining

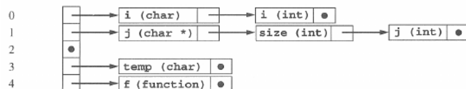
C code snippet

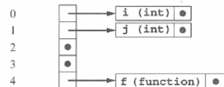
```
int i, j;

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

"Evolution" of the hash table

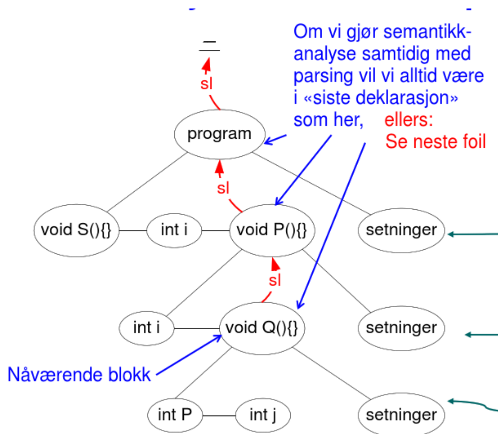






Using the syntax tree for lookup

```
lookup (string n) {
    k = naavaerende blokk
    do  // let etter n i decl til blokk k;
    k = k.sl
    until funnet eller k == none
}
```

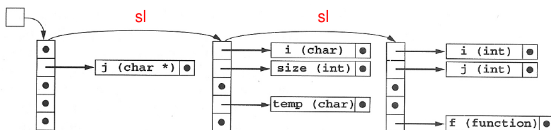


Alternative representation:

- arrangement different from 1 table with stack-organized external chaining
- each *block* with one own hash table.[4]
- standard hashing within each block
- static links to link the block levels

⇒ "tree-of-hashtables"

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



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

Outline

1. Symbol tables
   - Introduction
   - Symbol table design an interface
   - Implementing symbol tables
   - Block-structure, scoping, binding, name-space organization
   - Symbol tables as attributes in an AG

- remember the *interface*
- look-up: following the static link (as seen)[5]
- 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*

[5]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).

- 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

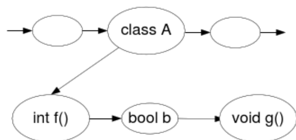
```
class A {
  ... int f(); ... // member function
}

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

Java analogon

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

- class *name* introduces a name for the scope[6] (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 separatly for each block (e.g. one hash table per class, record, etc., appropriately chained up)



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

Same-level declarations

Same level

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

- often forbidden (for instance in C)
- *insert*: requires check (= *lookup*) first

Sequential vs. “collaterals declarations

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

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

print_int(y);;
```

- 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);
}
```

- before treating the body, parser must add gcd into the symbol table.

Indirect recursion/mutual recursive def's

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

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

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

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:
  - 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 }
```

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 ($\neq$ Scheme)

```
(setq astring "a1")
(defun x() astring)
(x)
(let ((astring "a2"))
      (x))
```

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

Outline

1. Symbol tables
   - Introduction
   - Symbol table design an interface
   - Implementing symbol tables
   - Block-structure, scoping, binding, name-space organization
   - 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 (x+y))
```

- simple grammar (using , for "collateral" declarations)

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

Informal rules governing declarations

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)

```
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*[a]
- `emptytable`: you have to start somewhere
- `errtab`: error from declaration (but not stored as attribute)

[a] Realistically, more info would be stored as well (types etc)

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

- note: expression in let's can introduce scope themselves!
- interpretation of nesting level: expressions vs. declarations[7]

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

Attribute grammar (2): declarations

Grammar rule	Semantic rules
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$
$decl \rightarrow$ **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**$.name) =$ $decl.nestlevel$) **then** $errtab$ **else** $insert(decl.intab,$**id**$.name, decl.nestlevel)$

Annotations: 4 (second to fourth rules of first production), 4 (second production), 1 (else-if lookup clause)

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 context, but differently by "contexts", "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)
```