ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 4b

Symbol Table

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Symbol Table

Πίνακας Συμβόλων

- A data structure that holds information about identified *names* (type, scope, etc.)
  - Linear List
  - Hash table
- Operations
  - *Search*: The **symbol table is searched** every time a new name is encountered in the source text
  - *Change*: The **symbol table is changed** if a new name or new information about an existing name is discovered
- Evaluation
  - What is the time required to insert *n* entries and make *e* inquiries

Symbol-table Entries

- An entry is a name followed by some attributes
- Entries are entered in the table in various times
  - Keywords can be entered in advance, or not
- Role of an entry can become clear at a later point
- The same name may have several meanings
  - `int x; struct x { float y, z };`

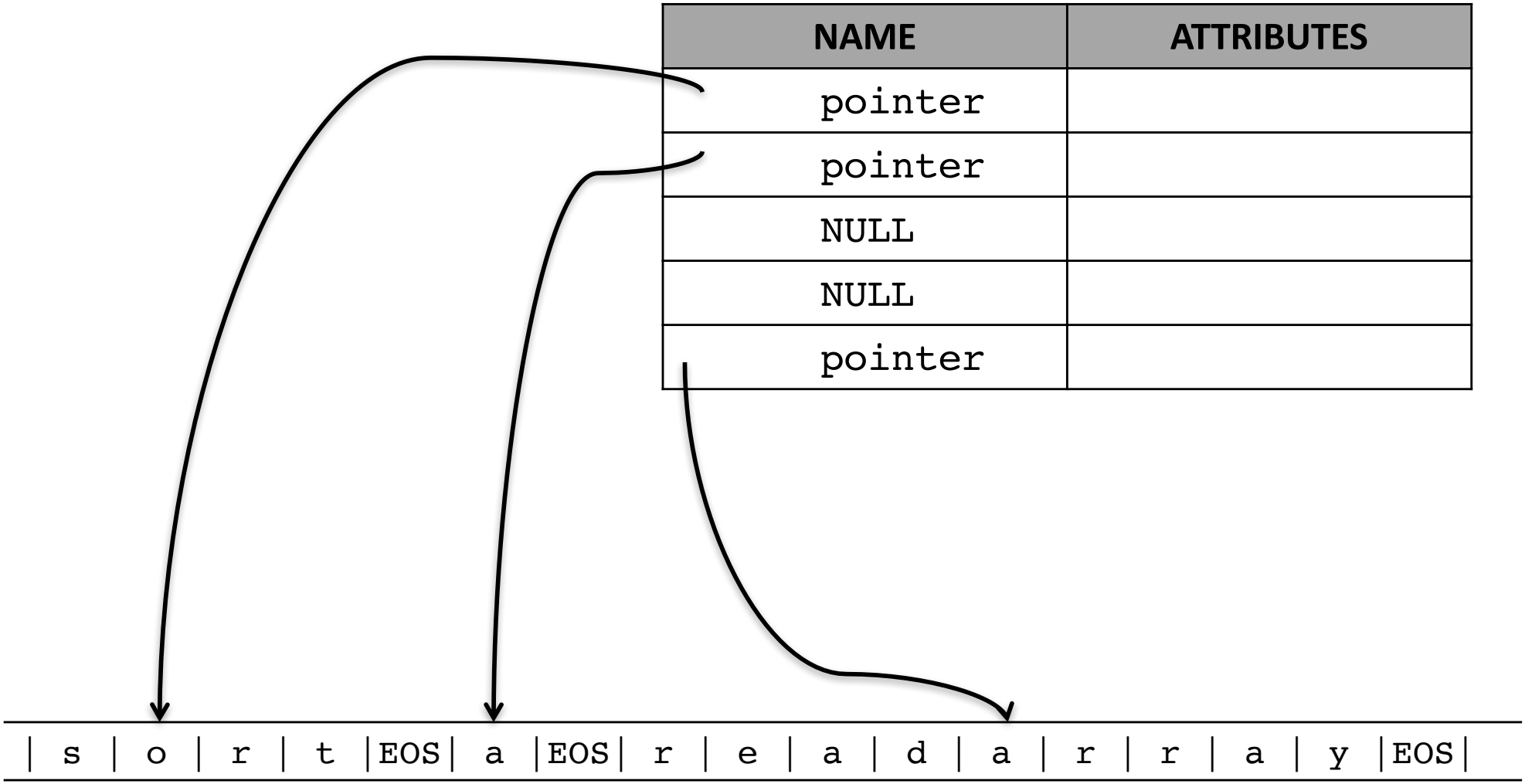
Characters in name

Fixed length

NAME	ATTRIBUTES
s|o|r|t| | | | | |	
a| | | | | | | | |	
r|e|a|d|a|r|r|a|y| |	
i| | | | | | | | |	
z|a|d|d|r| | | | |	

Characters in name

Dynamic Length

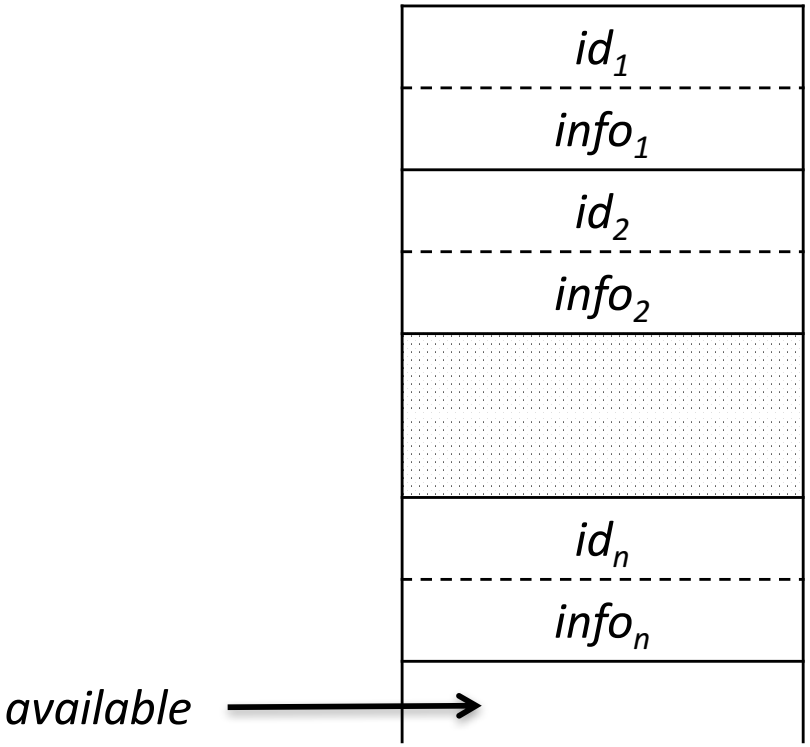


Storage Allocation Information

- Symbol entries may include storage-allocation details
- Variables are placed at certain memory locations at run-time
  - Compilers that produce assembly code, should delegate this to the assembler
  - Compilers that produce machine code, should take care of this at compile-time
- Variables are dynamically generated on the stack and heap of the process
  - Compilers should not be aware of the exact stack/heap locations
  - Compilers should emit the proper code to handle stack/heap allocations/deallocations

Symbol Table Implementation

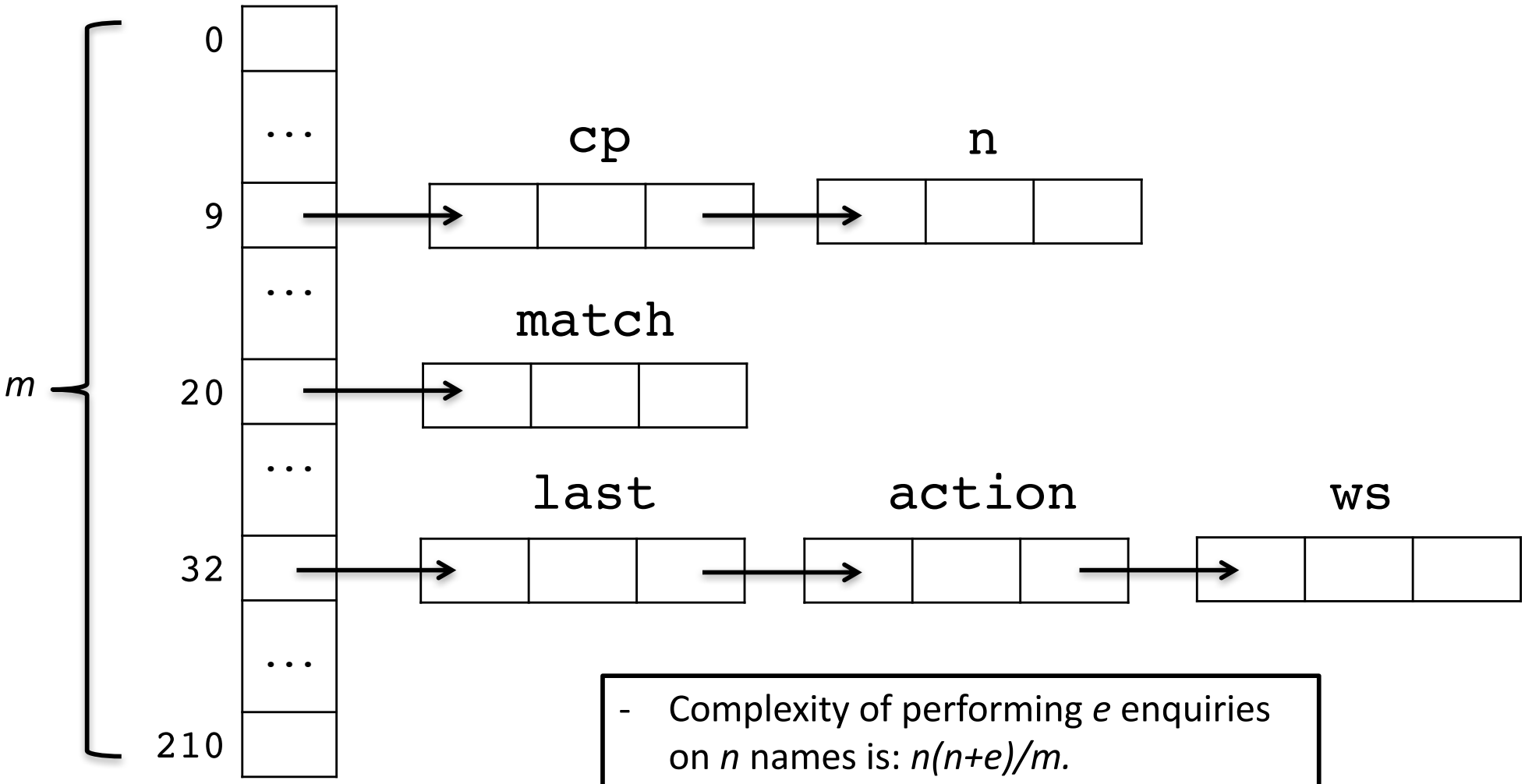
Linear list of records



- Entering a name needs work proportional to *n*, since we need to make sure the name is not already there (linear complexity).
- To find a name, on the average we search *n/2* names (linear complexity).
- The total work for inserting n names and carry out *e* enquiries is: *cn(n+e).*

Symbol Table Implementation

Hash Table



- Complexity of performing *e* enquiries on *n* names is: *n(n+e)/m.*
- We control *m*, if *m* ~ *n* then the complexity becomes linear ~*(n+e).*

Hash Table insertion

- Determine if there is an entry for string *s* in the symbol table
  - `h(s)` → `[0,(m-1)]`
  - If *s* is in the symbol table, then it is on the list numbered *h(s)*
  - If *s* is not in the symbol table, it is entered by creating a record for *s* that is linked at the front of the list numbered *h(s)*
- Rule of thumb: the average list is *n/m* records long If there are *n* names stored in a hash table of size *m*
  - If *m~n*, then the time to access the table is essentially constant

Other properties

- The space taken is
  - *m* words for the hash table
  - *cn* words for table entries, where *c* is the number of words per table entry
- Choice of *m*
  - It depends on the compiler
  - Some compilers take pre-compiled code, where the number of different names may be significant

Computing *h()*

- Determine a positive integer *h* from the characters $c_1, c_2, \ldots, c_k$ in string *s*
  - Most programming languages support a number representation of characters
- Convert *h* to [0, m-1]
  - Simply dividing by *m* and taking the remainder is a reasonable policy (works better if *m* is prime)

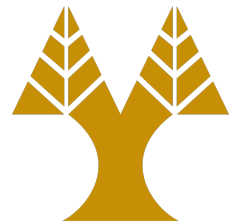
hashpjw()

```
#define PRIME 211
#define EOS '\0'
int hashpjw(char *s) {

    char *p;
    unsigned h = 0, g;

    for (p = s; *p != EOS; p = p + 1) {
        h = (h << 4) + (*p);
        if (g = h & 0xf0000000) {
            h = h ^ (g >> 24);
            h = h ^ g;
        }  /* if */
    } /* for */
} /* hasphjw */
```

Representing Scope Information



- Symbol table holds *names*
- When a *name* is occurred, while the source is scanned, the appropriate record should be fetched (or processed)
- The symbol table may host several entries with the same name
  - Consider the name *i* in C, which is used frequently as an integer counter

Scope

- Scope and name, both determine the meaning of a variable

```
void foo(int N) {
    int i = 0;
    for (i = 0; i < N; i++) { . . . }
    ...
}

int main(int argc, char *argv[]) {
    int i = 0;
    for (i = 0; i < N; i++) { foo(i); }
    ...
    if () { int i; ...  }
}
```

Local scope for function foo()

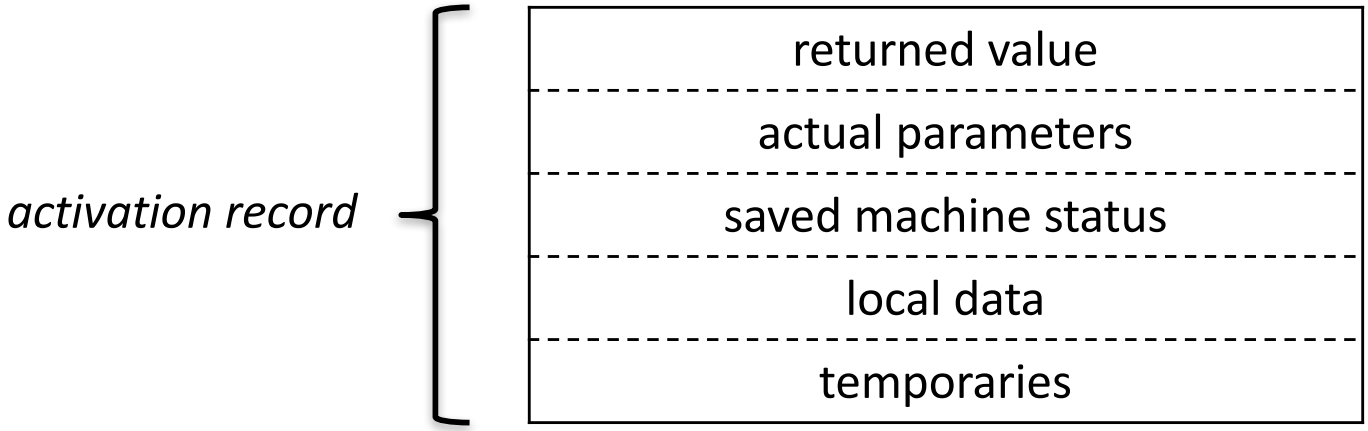
Local scope for function main()

Function parameter

Block scope

Scope and Symbol Tables

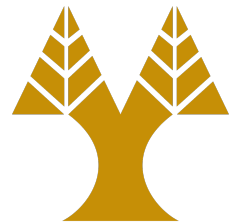
- Use several symbol tables *per scope*
- Equivalent at compile-time of *activation records* (or *frames*) at run-time



Symbol table per scope

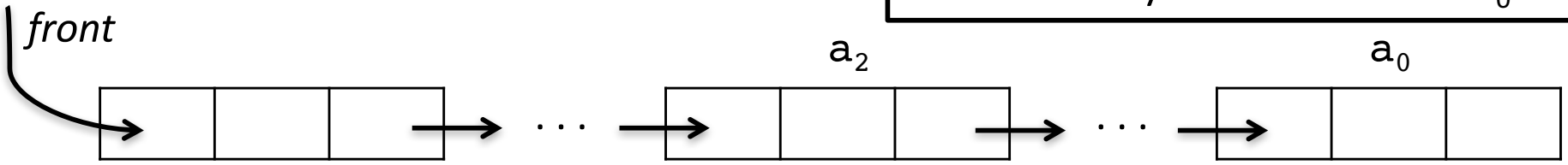
- Nested scope can be implemented using nested symbol tables
- Each symbol entry, includes a procedure (or block) number
  - i.e., symbol *x* declared in *id: 13,* which is *printf()*
  - Functions/blocks can be tracked in syntax analysis
- Operations for nested symbol tables
  - *lookup*: find the most recently created entry
  - *insert*: make a new entry
  - *delete*: remove the most recently created entry

Implementation of nested symbol tables



- Linear list

Entry of *a* declared in block B_2, nested in B_0, appears nearer the front of the list than the entry for *a* declared in B_0.



- Hash tables
  - Use a stack to keep track of the lists containing entries to be deleted