1 Abstract machines

1.1 Components

A computer is composed of at least a processor, a main memory, a mass storage device and peripherals for I/O. The different components are connected by a bus

1.1.1 CPU

obtains instruction from memory and executes them. Is composed of a control part, an operative part (ALU), registers and some times of a floating arithmetic part (FPU)

1.1.2 Registers

They are invisible, like address and data registers, or visibile like the program counter and status register

1.1.3 execution of instructions

read the instruction (or fetch), decode it and execute the decoded instruction. Precisely the fetch fase follows this logic: copy the PC into the AR, transfer the data referenced in the AR from the RAM to the DR, save the DR in an invisible register and increment the PC

1.1.4 Main memory

In the Von Neumann model, data and instruction are in the same memory. The access to the memory works like this: loads the address in the AR, in case of writing, load the data in the DR, then signal through the bus which operation to execute and, in case of read, move the data in the DR

1.2 Types of machines

1.2.1 physical machine

A physical machine is designed for executing programs in its own language and uses an intrepreted algorithm to understand and execute it. The execution of a program is a cycle composed of fetch,decode,load,execute and save.

1.2.2 abstract machine

The algorithm that executes the program does not need to be implemented in the hardware but is instead implemented through software.

NB: $\mathcal{M}_{\mathcal{L}}$ is an abstract machine that uses the language $\mathcal{L}$

To execute a program in $\mathcal{L}$, $\mathcal{M}_{\mathcal{L}}$ must execute elementary operations, control the seguence of execution, transfer data to/from memory and organize it.

$\mathcal{M}_\mathcal{L}$ understands only one language, in this case $\mathcal{L}$, but $\mathcal{L}$ can be understood by many abstract machines.
The implementation of a language can be via hardware, software and firmware

1.3 Implementation in software

The software runs on a host machine $\mathcal{MO}_{\mathcal{LO}}$ and can take two approaches: by interpretation and by compiler

NB:
A program can be written as a function

$$P^\mathcal{L} : \mathcal{D} \to \mathcal{D}$$
$$P^\mathcal{L}(i) = o$$

In this case P is a program written in $\mathcal{L}$ and gets i as input and o as output

1.3.1 Interpretive implementation

Uses an interpreter, or a program written in $\mathcal{LO}$ that executes $\mathcal{MO}_{\mathcal{LO}}$, that translates program from $\mathcal{L}$ to $\mathcal{LO}$ instruction by instruction. Implementation by interpretation is less efficient but more flexible and portable, plus the debugging is easier, thanks to interpretation at run time.

Definition
An interpreter for a language $\mathcal{L}$ writte in $\mathcal{LO}$ implements an abstract machine $\mathcal{ML}$ an a host $\mathcal{MO}_{\mathcal{LO}}$. HIs function is

$$\mathcal{I}^{\mathcal{LO}}_\mathcal{L} : (\mathcal{PR}^\mathcal{L} \times \mathcal{D}) \to \mathcal{D}$$

where $\mathcal{PR}^\mathcal{L}$ is the set of programs $\mathcal{P}^\mathcal{L}$ written in $\mathcal{L}$. So, for all $i \in \mathcal{D}$, $\mathcal{I}^{\mathcal{LO}}_\mathcal{L}(\mathcal{P}^\mathcal{L}, i) = \mathcal{P}^\mathcal{L}(i)$, which means for any input i, the interpreter applied to $\mathcal{P}^\mathcal{L}$ and i returns the same result as $\mathcal{P}^\mathcal{L}$ applied to i. Remember that $\mathcal{P}_\mathcal{L}(i)$ is calculated by $\mathcal{M}_\mathcal{L}$ and $\mathcal{I}^{\mathcal{LO}}_\mathcal{L}(\mathcal{P}^\mathcal{L}, i)$ is calculated by $\mathcal{MO}_{\mathcal{LO}}$

1.3.2 Compiler implementation

Uses a compiler that translates an entire program from $\mathcal{L}$ to $\mathcal{LO}$ before execution. The compiler doesn't need to be written in $\mathcal{LO}$ anc can be executed on an abstract machine $\mathcal{MA}$ different from $\mathcal{MO}_{\mathcal{LO}}$. Implementation by interpretation is more efficient but more complex due to the distance between $\mathcal{L}$ and $\mathcal{LI}$

Definition
A compiler from language $\mathcal{L}$ to $\mathcal{LO}$, written in $\mathcal{LA}$, implements M_L on $\mathcal{MO}_{\mathcal{OL}}$ and transforms a program $\mathcal{P}^\mathcal{L} \in \mathcal{PR}^\mathcal{L}$ into $\mathcal{P}^{\mathcal{LO}} \in \mathcal{PR}^{\mathcal{LO}}$. With the function

$$\mathcal{C}^{\mathcal{LA}}_{\mathcal{L},\mathcal{LO}} : \mathcal{PR}^\mathcal{L} \to \mathcal{PR}^{\mathcal{LO}}$$

So, for all $i \in \mathcal{D}$, $\mathcal{P}^{\mathcal{LO}}_\mathcal{C} = \mathcal{C}^{\mathcal{LA}}_{\mathcal{L},\mathcal{LO}}(\mathcal{P}^\mathcal{L}) \Rightarrow \mathcal{P}^{\mathcal{LO}}_\mathcal{C}(i) = \mathcal{P}^\mathcal{L}(i)$, where $\mathcal{C}^{\mathcal{LA}}_{\mathcal{L},\mathcal{LO}}(\mathcal{P}^\mathcal{L})$ is calculated by $\mathcal{MA}$

1.3.3 Hybrid implementation

It uses a compiler to translate the program in an intermediate language $\mathcal{LI}$ an than is interpreted by a $\mathcal{MO}_{\mathcal{LO}}$-program written in $\mathcal{LI}$. Depending on the differences between $\mathcal{LI}$ and $\mathcal{LO}$, the implementation can be mainly compilative or interpreted

1.4 Implementation in firmware

This is an intermediat implementation between an abstract machine and a CPU. It uses microprogrammable CPUs, abstract machines not implemented in hardware but as microinterpreter, microinstructions and microprograms. This is more flexible than a pure hardware implementation

1.5 Linking and loading

The compiler doesn't generate completely executable code and memory locations are relative, not absolute. It is the linker/loader that replaces this with absolute addresses and links precompiled libraries

2 Environments

An environment is the collection of associations between names and objects in run-time at a specific point in the program.
The declaration is the mechanism that creates an association between name and object. The same name can denote different objects at different points in the program. When it is in the same point it's called Aliasing.

2.1 Blocks

An environment is structured in different blocks, or sections of the program. This allowes the creation of local declarations. Blocks can be nested one inside the other, and the declaration are accesible in the nested blocks, or hidden outside the block.

2.2 Subdivisions of the environment

There are 3 types of subdivisions: a local environment (where are stored local variables and formal parameters), nonlocal environment (with all the inherited associations) and the global environment (where are the associations common to all blocks)

2.3 Fundamental events

Creation of an object
Creation of an association for the object
Reference to an object via association
deactivation of an association
Reactivation of an association
destruction of an association
Destruction of an object

2.4 Scopes

2.4.1 Static scope

A non local name is resolved in the block that includes. In this case all information is included in the program text, associations can be derived at compile-time, uses the principle of indipendece and it's easier to implement and more efficient

2.4.2 Dynamic scoping

A non local name is resolved in the vlock that has been most recently activated. In this case information is derived during execution, is harder to implement, less efficient and can result in hard to read programs

2.4.3 Determining the environment

The environment is determined by scoping rules, and IDK THE REST IS CONFUSING

3 Memory management

3.1 Memory allocation

It can be static (allocated at compile time), dynamic (allocated at execution time) in stack (LIFO) or heap(objects allocated and deallocated at any time)

3.1.1 Static

An object absolute address in mantained throughout the execution of a program. Usually we are talking about global variables, local variables of subprograms, constants and tables determined at run time

3.1.2 Stack

In the stack gets stored the information of every new subprogram that enters run time. In a similar way, each block has his activation record (composed of pointers, local variables and intermediate results) saved in the stack.
To understand what is the active block, we use the activation record, which is a pointer that points to the activation record of the active block. It also uses the dynamic link to point to the previous record on the stack

3.1.3 heap

The freedom of operation of this region of memory permits to explicitly allocate memory at runtime, to create objects of varying sizes and with a life time that is not LIFO. But the heap needs a good management to avoid fragmentation of data.
The heap can be divided in blocks of fixed size and, at the start of the program, this blocks are linked in a free list. Alternatively, the blocks can be of variable size, with a single block a the begging.

The first case case can cause internal fragmentation, when a block size in bigger than the space that needs to be allocated. In the case of variable block size, this can cause external fragmentation, namely when between two blocks is empty space that cannot be allocated because of it size. To manage this two situations we can use different approaches, like first fit (the first block available is allocated) or best fit (it is allocated the smallest block large enough). In case of internal fragmentation, the excess space is used to create a new block.

PAGINA 26/62

4 abstraction

The objective of abstraction is to identify the important properties of what we want to describe

4.1 Parameter passing

A parameter can be transfered by value or by reference (or variable). In the first case the variable is local (on the stack) and after exiting from the block, it gets destroyed. This type of transfer is expensive on the memory because it requires the copy of the data. Instead, in the referenced parameter, it's passed an address or a pointer, after exiting the block only the link is destroyed and the trasmission is two-way.

In alternative to call-by-value there is read-only, which doesn't copy the data, but makes it impossible to modify it. Also there is the call by value/result, where the parameter passed is used also as output

4.2 High order functions

This types of functions are those used as arguments to procedures or as a returning functions as results of procedures

19/59

5 Data structures and types

Types of data can be used on different levels: in projects to organize the information, in a program to identify data and in implementation to act certain optimizations

5.1 Type systems

The type system of a language is composed of predefined types, mechanism to define new ones and control mechanisms. It's defined as type safe if the system is safe from type errors at runtime

5.2 Scalar types

The booleans are scalar types with true and false as values and or,and, not and conditions as operation. We have characters, integers and reals with mathematical operations and the void type

5.3 Classification

ordinal: each value has a successor and a predecessor
scalar: have a direct representation in the implementation
Structured: arrays, sets, pointers and records (structs)

NB:variant records are basicallly nested structs
Es:

```
struct student {char name[6];
int number;
bool finished;
union {
int last_year;
int : variant records};
}
```

5.4 Arrays

The information about the form of an array is maintained by the compiler. If the form can vary during runtime, the compiler uses a dope vector which contains the pointer to the beginning of the array, the number of dimensions, the lower bound and the space needed for each dimension. The dope vector is stored in the fixed part of the activation record

5.5 Equivalence and compatibility

Two types S and T are equivalent if every object of type T is also of type S. Instead, T is compatible with S when a object of type T can be used when S is expected.

It's also defined a structural equivalence when a type is equal to another with the same structure.

If T is compatible with S we can implement a conversion mechanism which can be of two type. We are talking of implicit conversion (or coercion) when the conversion is managed by the abstract machine. The other case is explicit conversion (or cast) and it's applied when the conversion is mentioned in the code. The casting is allowed only when the language knows how to do the conversion

5.6 Polymorphism

When a single value has multiple types is called polymorphism. It can be ad hoc (also called overloading), parametric or subtype.

Overloading happens when the same symbol has different meanings. In this case it is resolved at compile type.

Parametric polymortphism is when a value has an infinite number of possible types, obtained by instantiation of a general type schema. In a nutshell, when a function doesn't need a defined type (<typename T>)

Subtype polymorphism is very similar to parametric polymorphism, the difference is that instead of using a general type, it accepts all subtypes of a given one.

5.7 dangling references

A dangling reference is when a pointer points to a no longer valid object. The avoid this behaviour, various techniques can be implemented. One of which are tombstones (data used to validate the object), or locks and keys (it saves both object and pointer with a number equal only for the reference)

5.8 Garbage collection

It's when the system recovers allocated memory that no longer in use.
One way to do it is the mark and sweep method, in which all objects in the heap are marked as unused, then visit all the linked structures and marking them as used, and in the end recover all unused items. This method can be very heavy on the system performance

6 Data abstraction

6.1 Abstraction

Components (everything in the code), interface, specify (functionality of a component observable through the interface), implementation (structures inside the component).

Other terms are abstraction of control (when the implementetion of procedures is hidden), data abstraction (hiding decisions about representation and implementation of data structures), encapsulation (limiting the user to what components it can see/use).

The entire point of abstraction is to separate the interface from implementation.

7 Lambda calculus

5