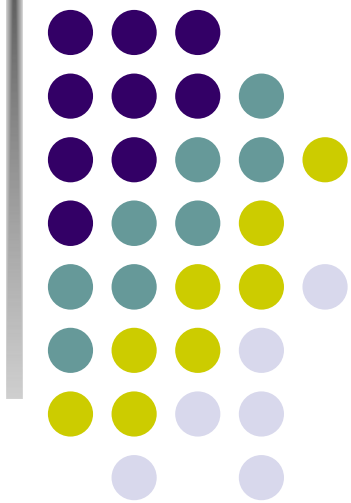
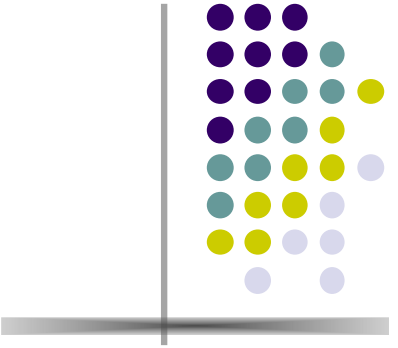
Semantics & Intermediate Representation

Functions (Subroutines)

(Context Management)

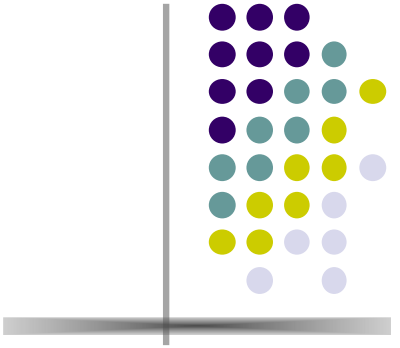


Modular Programming



- *One of the most important features in any programming language within the Imperative Paradigm (and also in OO) is the possibility to "break" a large task into specialized subroutines (functions) that can be called from many different places in a program.*
- *This feature allow us to avoid writing over and over the same chunck of code in different parts of the program.*
- *If necessary, it allows us to execute the same piece of code for different "input" values (arguments).*
- *This module/subroutine/function can even return a "result" generated by the execution of the code it contains.*

Modular Programming



- ***In the theory of Programming Paradigms there are a lot of characteristics expected in a "good modular programming" like:***
  - ***Coupling***
  - ***Cohesion***
  - ***Information Hiding***
  - ***etc, etc..***

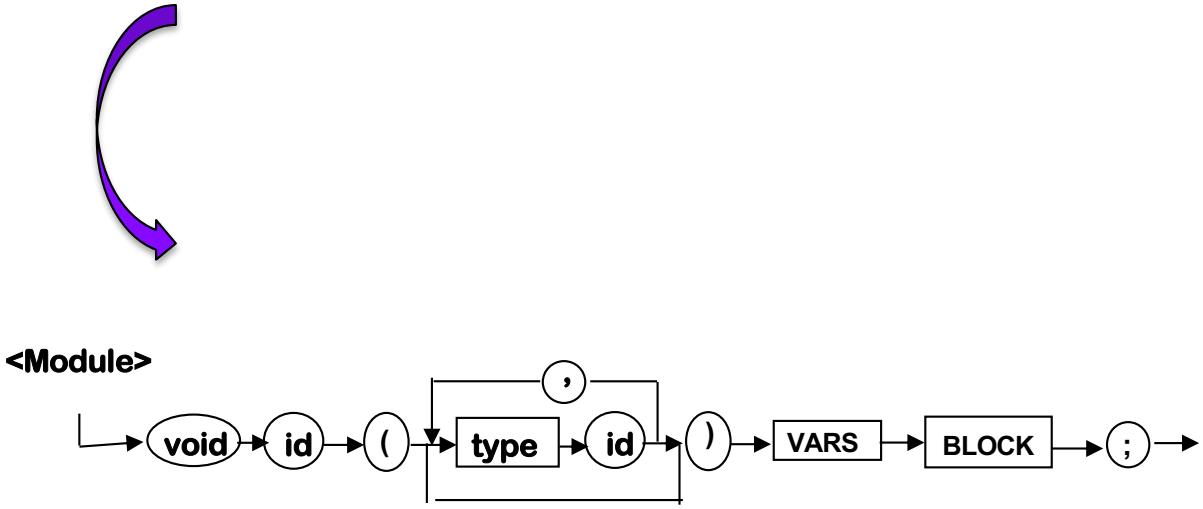
- ***You should remember them from your previous courses (it's NOT part of this course).***

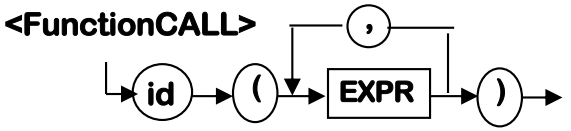
Modules

- *These elements must be analyzed from 2 different perspectives:*
  - *Function Definition / Declaration*
  - *Functions Calling*

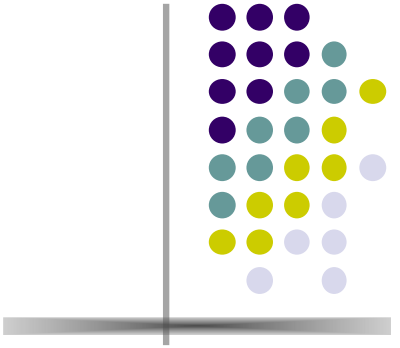
Function Calling

Function/Module Definition



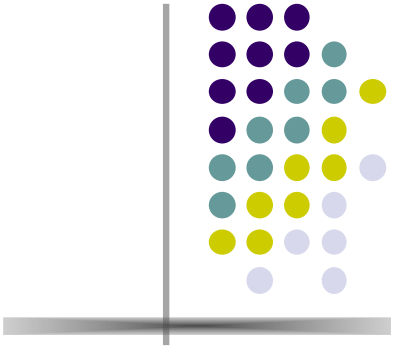


Modules Semantics



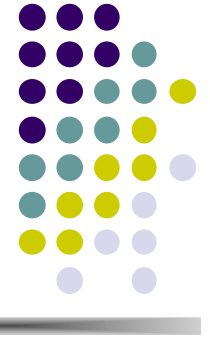
- *Every Function has a UNIQUE name.*
- *Each function declares parameters and local variables (these follow the same rules as global variables).*
- *If the function returns a result, a RETURN Statement must exist within the function body (block).*
- *When a function is called, its "signature" must be respected: name, number of arguments and types. If its a VOID functions, the call is a statement; if it is a Non-Void function, the call is part of an expression.*

Module Execution



- When a function is called, the current context must be saved (memory and returning address).
- A new Memory segment must be created to allocate arguments, local variables and temporal records.
- When the function ends, the current context (local) must be deleted, the return value must be sent (if any), the previous memory must be awakened, and the IP takes the returning address.

Intermediate Representation for a VOID Function Declaration

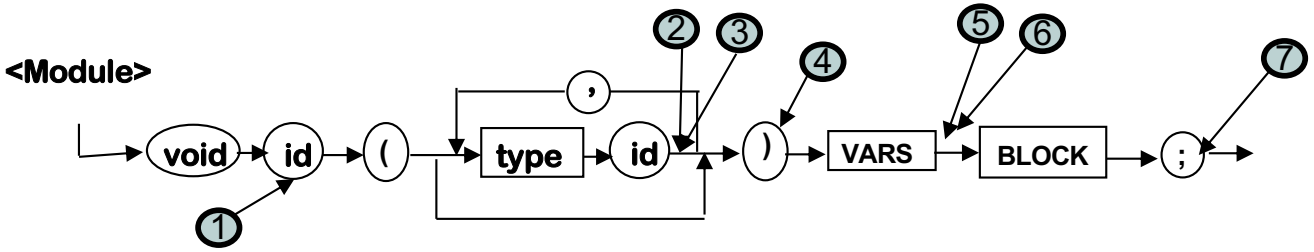


- ***MODULES-related Operation CODES.***
  - **GOSUB , FUNCTION-Name , InitialAddress**
  - **ERA , SIZE**
  - **PARAMETER , argument , Parameter#**
  - **ENDFUNC**

- ***GOSUB***
  - ***Unconditional JUMP that changes the InstructionPointer to a specific line of code (DESTINATION),*** *(in **RUN-Time**, Save the current IP (Instruction-Pointer). Update IP with initial-address. Transfer the Control-Flow to that address and continue.)*
- ***ERA***
  - ***Indicates the size of the Local-Memory to be created in Run-Time****. (in **RUN-Time,** Save the current Memory pointer (in case it is an Activation Record).. Creates Memory (Activation-Record) to store arguments, local variables and temp-vars according to the size specified (--LocalMemory--).*
- ***Parameter***
  - ***Indicates that the argument sent must be copied into parámater#-- in Run-Time***
- ***ENDFunc***
  - ***Indicates the END. (In Run-Time,*** *Update the current memory (prior to the call). Erase LocalMemory (Activation Record). Update IP (prior to the call). Transfer the Control-Flow to that address.)*

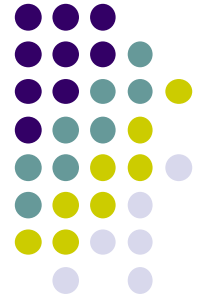
Intermediate Representation for a VOID Function Declaration

- ***Function Definiton***

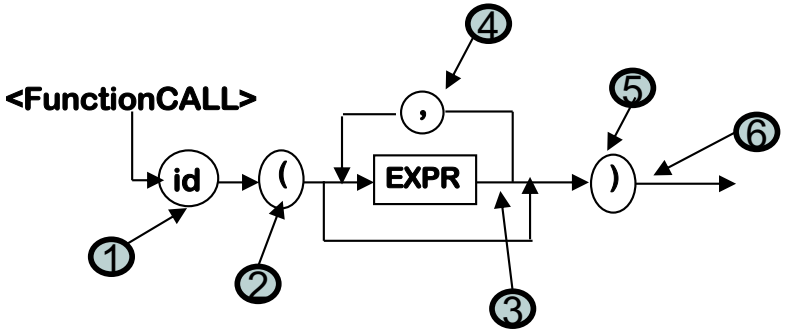


1.- Insert Function name into the DirFunc table (and its type, if any), verify semantics.
2.- Insert every parameter into the current (local) VarTable.
3.- Insert the type to every parameter uploaded into the VarTable.
At the same time into the **ParameterTable** (to create the Function's signature)..
4.- Insert into DirFunc the number of parameters defined. ****to calculate the workspace required for execution***
5.- Insert into DirFunc the number of local variables defined. ****to calculate the workspace required for execution***
6.- Insert into DirFunc the current quadruple counter (CONT), ****to establish where the function starts***
7.- Release the current VarTable (local).
Generate an action to end the function (**ENDFunc**).
Insert into DirFunc the number of temporal vars used. ****to calculate the workspace required for execution***

Intermediate Representation for a VOID Function Calling



- ***Function Calling***



1.- Verify that the function exists into the DirFunc.
2.- Generate action **ERA size** (*Activation Record expansion –NEW—size).*
Start the parameter counter (k) in 1.
Add a pointer to the first parameter type in the ParameterTable.
3.- Argument= PilaO.Pop() ArgumentType= PTypes.Pop().
Verify ArgumentType against current Parameter (#k) in ParameterTable.
Generate action **PARAMETER, Argument, Argument#**k
4.- K = K + 1, move to next parameter.
5.- Verify that the last parameter points to null *(coherence in number of parameters).*
6.- Generate action **GOSUB**, **procedure-name, , initial-address**.