*REMIND*

* *Output of lexical analysis is tokenization.*
* (def f(x y)  
   (+ x y))

F 🡪 ( def N (V V) E)  
E 🡪 (+ V V)

…

F.value = E.value(v, v)

**DATA TYPES**

Scope

The range of statements where a variable is visible, i.e. can be accessed

Local variables are visible in the program unit where they are declared

Non-local variables are visible in a program unit but not declared there

Scope rules determine how names are associated with variables

Static Scope

Also called Lexical Scope

* The scope of variable can be determined statically (prior to execution), just by looking at the code

Based on structure of program text

To connect a name to a variable, one (you or the compiler) must find the declaration

Search process:

* Search declarations, first locally,
* Then in increasingly larger outer enclosing scopes
* Until declaration for the name is found

Enclosing static scopes (to a specific scope) are called its static ancestors

* The nearest static ancestor is called a static parent

Shadowed Variables

Variables are shadowed (hidden) in part of the code where there is a more immediate ancestor (closer in scope) with the same name

C++ and Ada allow access to hidden variables using longer names

* In Ada: <unit> .<name>
* In C++: <class\_name> ::<name>

Common Lisp

* Packages variables accessible via longer name or package imported

Java

* this.<name>
* <class\_name>.this.<name>
* super.<name> // shadowed field in superclass

Scope Blocks

Static scopes can be created inside program units

Started with ALGOL 60!

C and C++:

for (int index; …) {

int x; …

}

Scope block

* statements in between {…}, and
* condition expression in preceding or following (…)

Advantage of declaring the variable anywhere in the block: flexibility, writability  
Disadvantage: readability

Declaration Order

Where should be variables declared

* For some PLs, must appear at the beginning of functions
* Some others, e.g., C++, Java, etc. can appear anywhere

What scope

* Only after declaration till the end of the block, e.g., in C++
* Anywhere in the block, e.g., in C# and JavaScript
  + C# still requires declaration
  + JavaScript will result in “undefined”

Chart, radar chart

Description automatically generated

-> E’s ancestors are B and main

Diagram

Description automatically generated

Dynamic scoping: I don’t wanna decide what the scope is at static time. I run the code, I am at an expression and I have a variable. Then I look at my history and say is this variable declared up to now? If it is declared up to that point, then I use that declaration.

In dynamic scoping, you cannot just look at your code and say “the scope of this variable is this”.

Dynamic Scope

Uses calling sequences of program units, not their textual layout

* i.e. temporal versus spatial

References are connected to declarations by searching back through the chain of subprogram calls in execution

* i.e. search through the dynamic stack for the most recent variable with the name

Scope Example

Graphical user interface, text, application

Description automatically generated

For static scoping:

* First look at the scope SUB2 for x, it doesn’t exist there
* I cannot look at the SUB1 bc it is not visible to me (SUB2)
* So my scope says it is MAIN’s x

Static scoping 🡪 Compile Time  
Dynamic scoping 🡪 Runtime

If MAIN calls SUB2, So it would be MAIN’s x in dynamic scoping too. In dynamic scoping it all depends on called sequences.

If this code is C, then we have a problem. C cannot do dynamic scoping this way. declaration of x in SUB1 is local variable. After return, local variables are gone. So if we don’t call SUB2 in SUB1 but we call SUB2 after call SUB1 in MAIN, now lifetime of x in SUB1 is gone so SUB2 will use declaration of x in MAIN.

Static Scope Example

Graphical user interface

Description automatically generated with medium confidence

Dynamic Scope Example

Graphical user interface, application

Description automatically generated

Static Scope in Nested Loops

Table

Description automatically generated with medium confidence

Scope Evaluation

Advantage of dynamic scoping

* convenience (no need to pass variables, our calling sequence will define that)
  + not having to pass the variables might give us speed up
* flexibility

Disadvantage

* poor readability (same name different non-local vars)
* reliability (accessibility from multiple subprograms)
* longer access times for non-local variables (going through stack)

Scope and Lifetime

Scope and lifetime are sometimes closely related, but are different concepts

* Scope of a variable determines the area or a region of code where a variable is available to use.
* Lifetime of a variable is defined by the time for which a variable occupies some valid space in the system's memory.

Consider a static variable in a C or C++ function

* Scope is static and local to the function
* Lifetime extends over the entire execution of the program

Referencing Environments

Defined for a statement or an expression. It means all the names (not necessarily just variable names but other names like functions, types) visible in that statement or visible in that expression.

Referencing environment (RE) of a statement

* all names that are visible in the statement

*In statically-scoped language RE is*

* *Local variables, and*
* *All of the visible variables in all of the enclosing scopes in terms of variables but this is extended to other names*

*In dynamically-scoped languages RE is*

* *Local variables, and*
* *All visible variables in all active subprograms*
* Active means
  + execution of the subprogram has begun and
  + is not yet terminated

RE is gonna be your list of things that you can have access to at that given statement or expression.

Named Constants

A variable that is bound to a value only once

Improved readability

* E.g., “pi” instead of “3.14….”

Parameterize a program

* E.g., parameterizing length of an array

C++ allows dynamic binding of values to named constants

* “const int result = results \* width + 1;”

C# allows static “const” as well as dynamic “readonly”

Initialization

The binding of a variable to a value at the time it is bound to storage

Static

* Binding and initialization occur before run time
* Initial value must be specified as a literal or an expression whose only nonliterals are named constants
* When you load things to memory, or your program is loaded to run, then your values must be defined. In that case you can either have simply literals or expressions that use non-literals as name constants and nothing else. You cannot use variables for example.
* Advantage is you can do semantic check if the initialization is right

Dynamic

* Initialization is dynamic, it happens during runtime
* You can also do semantic check but it is expensive. You may need a assignment check if that assignment operator or statement is type compatible or not.

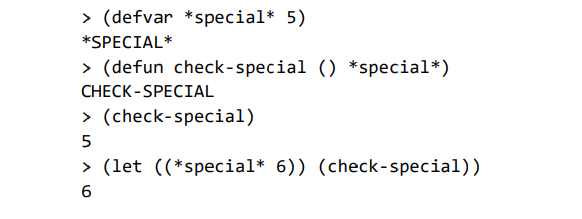
*Example*:

Consider the C preprocessor:

* #define ADD(x)x + a

*Example*:

LISP calls dynamically scoped variables as special variables



*Example*:

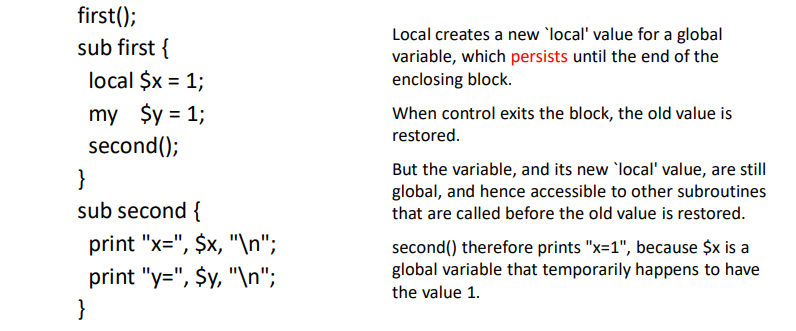
Perl

“my” for lexical scoping

* A “my” variable has a block of code as its scope
* A block is often declared with braces {}, but as far as Perl is concerned, a file is a block
* A variable declared with “my” does not belong to any package, it 'belongs' only to its block

“local” for dynamic scoping

*Example*:



Type

A type is a collection of computable values that share some structural property

A type is also a set of operations on the values

Examples:

* Integers, Strings, int 🡪 bool, (int 🡪 int) 🡪 bool, …

Not type examples:

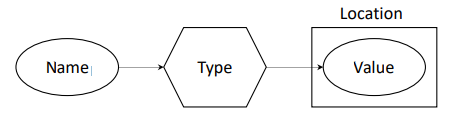
* {3, true}, , Even integers (differs from language to language)

Distinction between sets that are types and sets that are not types is language-dependent

Variables Revisited

Recall that variables have six attributes

* Name
* Type
* Location or reference (l-value)
* Value (r-value)
* *Scope - where variable accessible and modifiable – static vs. dynamic*
* *Lifetime - interval of time in which location bound to variable*



An identifier (or name) is used to identify the entities, e.g., variable names, function names, etc., in a programming language

A *descriptor* is the collection of the attributes of a variable

* Implementation: an area of the memory where attributes of a variable is stored
  + All static attributes: descriptors are required only at compile time. Built by compiler and stored usually in symbol table and used during compilation.
  + If dynamic: Part or all off the attributes need to be maintained during execution. The descriptor is used by the run-time system.
* Used for type checking and building the code for allocation and de-allocation

Uses for Types

Program organization and documentation

* Separate types for separate concepts
  + Represent concepts from problem domain
* Indicate intended use of declared identifiers
  + Types can be checked, unlike program comments

Identify and prevent errors

* Compile-time or run-time checking can prevent meaningless computations such as 3+true-“Bill”

Support optimization

* Example: short integers require fewer bits
* Access record component by known offset

*Example*: z = x / y; (Java)

Text, letter

Description automatically generated

Operations on Typed Values

Often a type has operations defined on values of this type

* Integers: + - / \* < > … Booleans: V …

Set of values is usually finite due to internal binary representation inside computer

* 32-bit integers in C: –2147483648 to 2147483647
* Addition and subtraction may overflow the finite range, so sometimes a + (b + c) (a + b) + c
* Exceptions: unbounded fractions in Smalltalk, unbounded Integer type in Haskell
* Floating point problems

Type Errors

Machine data carries no type information

* 01000000010110000000000000000000 means…
  + Floating point value 3.375?
  + 32-bit integer 1,079,508,992?
  + Two 16-bit integers 16472 and 0?
  + Four ASCII characters @ X NUL NUL?

A type error is any error that arises because an operation is attempted on a value of a data type for which this operation is undefined

* Historical note: in Fortran and Algol, all of the types were built in. If needed a §çtype “color,” could use integers, but what does it mean to multiply two colors?

Java Types

Diagram

Description automatically generated

C Types

Timeline

Description automatically generated with medium confidence

Simple Data Types

No internal structure, directly supported by hardware, machine supported:

* e.g., integer, double, character, and boolean

Often directly supported in hardware

* machine dependency

Most predefined types are simple types

* Exceptions: String in Java.

Some simple types are not predefined

* Enumerated types
* Subrange types

Floating Point

Diagram

Description automatically generated

Enumerated Types

Ordered set, whose elements are named and listed explicitly.

Examples:

* enum Color\_Type {Red, Green, Blue}; ( C )
* type Color\_Type is (Red, Green, Blue); ( Ada )
* datatype Color\_Type = Red | Green | Blue; ( ML )

Operations:

* Graphical user interface, application

  Description automatically generatedSuccessor and predecessor

Graphical user interface, application

Description automatically generated

Evaluation of Enumeration Types

Efficiency – e.g., compiler can select and use a compact efficient representation (e.g., small integers)

Readability -- e.g. no need to code a color as a number

Maintainability – e.g., adding a new color doesn’t require updating hard-coded constants.

Reliability -- e.g. compiler can check operations and ranges of value.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Type constructors: Defining New Types

We are dealing with sets of values. So new types are generated basically using set operations.

To define new type, you need type constructors.

We look two components for types:

* Values (finite set)
* Operations

If you want to create a new type, simplest way is defining your types as set of values and new types will be just new sets. How can I generate a new set from an existing set of sets? You use set operations.

Type constructors as set operations:

* Cartesian product
* Union
* Subset
* Functions (Arrays)

Some type constructors do not correspond to set operations(e.g., pointers)

Some set operators do not have corresponding type constructors (e.g., intersection)

Cartesian Product

Ordered pairs of elements from U and V (For example integers and characters)

U x V = {(u, v) | uU and vV}

Operations:

* projection (only meaningful operation)

p1 : U x V → U; p2 : U x V → V

p1((u,v))=u; p2((u,v))=v

*Example*:

struct in C :

struct IntCharReal {

int i;

char c;

double r;

}

int × char × double 🡪 you have 3 entries (int, char, double), this is a new type

I can have 28x216x264 = 288 different values.

I can only have projection operations (what’s your first entry, second entry, third entry?):

* int x char x double 🡪 int

🡪 char

🡪 double

structs are cartesian products.

This new struct type has only projection operation.

Our new type is 🡪 (int, char, double)

Compatibility:

* Can you use values interchangeably?
  + Can you put 2 types’ values in the same place?
* Are the operations same?

The Same Type?

struct IntCharReal{

int i;

char c;

double r;

}

struct IntCharReal{

int j;

char ch;

double d;

}

First one semantically says:

* values 🡪 int x char x double
* projections (individual components):
  + .i
  + .c
  + .r

Second one semantically says:

* values 🡪 int x char x double
* projections:
  + .j
  + .ch
  + .d

They have same values but their operators are different. So they are not same type.

If C uses [0], [1], and [2] to get components, we can say semantically these are same. They have both same operations and both have same values. So they would be same type.

The Same Type?

struct IntCharReal{

int i;

char c;

double r;

}

struct IntCharReal{

char c;

int i;

double r;

}

Values for first:

* int x char x double
* For example: (0, ‘a’, 0.0)

Values for second:

* char x int x double
* For example: (‘a’, 0, 0.0)

They are not the same type.

Projections seem to be getting the same thing but they are not getting the same order. C gets the same thing (for instance when you say .c) as far as the projections are concerned but values are not the same.

Also they don’t use the same structure in memory, their orders are different in memory (int-char-double vs. char-int-double).

Graphical user interface, text, application, chat or text message

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Union

U V = { x | xU or xV}

data items with different types are stored in overlapping region, reduce memory allocation

Only one type of value is valid at one time

E.g.,

union IntOrReal {

int i;

double r;

}

New type cannot be both at the same time, it can be either one of those.

I can ask for the type, that is my operation.

C Style Unions

C style unions allow the same bit pattern to be interpreted in multiple ways

E.g., it's common to use a union to pack/unpack a fixed length binary blob into its constituent fields

Useful in some systems programming contexts

But it's a very low level feature and was excluded in most languages

Undiscriminated Union in C

union IntOrReal {

int i;

double r;

}

union IntOrReal x;

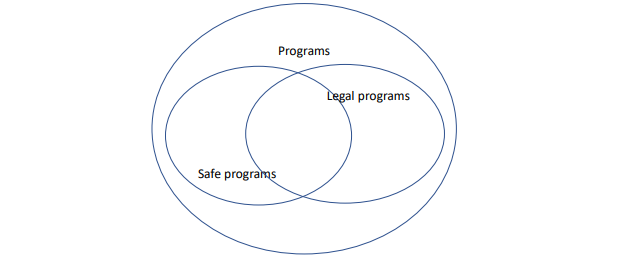
x.i = 1;

printf(“%f\n”, x.r);

Can be unsafe

You have projection operator but it is not the right thing as a projection operator in a cartesian product. Because cartesian product will pick one of the entries from the tuples. Here we say either this entry or that entry.

Safe vs. Legal



Any decision that we have with any semantic that we implement in our language will have the question of legality of that semantic and safety of that semantic.

Language will generate all programs (bigger circle).

Abstract syntax will let you have portion of the program to be legal.

For example unions are legal but not safe. Because of the semantic implementation, idea can be misused. Like casting, casting can be used in an unsafe manner easily.

Type Equivalence

How to decide if two types are the same?

Structural Equivalence

* Types are sets of values
* Two types are equivalent if they contain the same values

Name Equivalence

* Same type names
* Two types are same if their names are same

Which one is more flexible?

* Structural equivalence

Two types are same if:

* They have same set of values
  + This might also mean that bit patterns that you have for those things are the same, somehow you handle them and you do the conversions
* They have same operators

Structural Equivalence of Type

Two type expressions are structural equivalent if they are

* same basic type, or
* formed by applying the same constructor to two structurally equivalent types, or
* after type declaration type n=T the type name n is structurally equivalent to T

🡪 two type expressions are structural equivalent if and only if they are identical

Example:

* integer is equivalent to integer
* pointer(char) is equivalent to pointer(char)

Modification is needed for structural equivalence: When array are passed as parameter, we may not wish to include the array bounds as part of the type

* If there is a type having 10 entries for someone’s array, and having 20 entries for my array shouldn’t matter. They are still the same.

Structural Equivalence

Diagram, schematic

Description automatically generated

in terms of constructor

They are structurally equivalent.

For C, from the perspective of accessing the entries, first three are having problems bc for example RecB and RecC will access their entries in different ways. C will not use structural equivalency for records.

Graphical user interface, text, application

Description automatically generated

C uses name compatibility for structs.

For C, if you put a casting in front of a (b = (casting)a;) then you can do that.

Graphical user interface, text, application, email

Description automatically generated

For pointer, C uses structural equivalency. &a is taking the address of a and it is generating a pointer. *Structural equivalency says RecA and RecB should be the same.* Since they are structs, their names are different so they are not equivalent.

So first we use structural equivalency on the pointer operation and then inside when we check recursively the component types of that thing, you encounter a struct and then you have to go back to name equivalency.

Graphical user interface, text, application

Description automatically generated

OK but not safe. Compiler says “Okay, you are on your own. You know what you are doing so I am not gonna check equivalency anymore bc you told me that they are equivalent.”.

As a programmer, you have to go back and make sure that safety-wise this casting is possible. This is in legal but not safe circle.

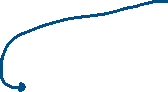
Graphical user interface, text

Description automatically generated with medium confidence

Structural Equivalence Algorithm

Text

Description automatically generated



function that takes s1 and generates s2.

Equivalence Algorithm

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Structural Equivalence

Can be complicated when there are names, anonymous types, and recursive types

Simpler, and more strict rules: name equivalence

There are 2 things:

* Static
  + Type checking is done at static time (compile time)
  + We did:
    - Lexer
    - Parser 🡪 Concrete syntax
      * 🡪 Semantic 🡪 Type (Everytime I do type equivalency check (e.g., anytime I do an assignment, expression, I pass a value to a function…))

I need to run an algorithm here

That algorithm is part of the system

Lexer takes time, parser takes time, syntax takes time, semantic… takes time.

Type checking also takes time in compile time.

* Dynamic (Real time – runtime)
  + While you are executing your statement, statement will require equivalency check. When there is an equivalency check, your algorithm is gonna run there. For example everytime you have an assignment, there is an algorithm that is being called in the background to do the checking.
  + This algorithm needs to be very efficient bc it takes cycles. Assignment needs to be done in 1 cycle but we may have 100 cycles due to this checking.

Name Equivalence

A picture containing text

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Type Conversion

Use code to designate conversion?

* No: automatic/implicit conversion (compiler)
* Yes: manual/explicit conversion (user)

Data representation (bit patterns) changed?

* No, just the type
* Yes

Example: Java

Implicit conversion:

* Representation change (type promotion, e.g., int to double)
* No representation change (upcasting)

Explicit conversion:

* Representation change (double x = 1.5; int y = (int)x)
* No representation change (downcasting)

Diagram

Description automatically generatedUpcasting and Downcasting

Upcasting and downcasting allow building of complicated programs using simple syntax, and gives great advantages, like Polymorphism or grouping different objects

upcasting: allowing an object of a subclass type to be treated as an object of any superclass type.

* Upcasting is done automatically, while downcasting must be manually done by the programmer

Graphical user interface

Description automatically generated with low confidence

Type System

***Type Constructors:***

* Build new data types upon simple data types

***Type Checking:*** The translator checks if data types are used correctly.

* **Type Inference:** Infer the type of an expression, whose data type is not given explicitly.
  + e.g., x/y
* **Type Equivalence:** Compare two types, decide if they are the same. Are their operations and values same?
  + e.g., x/y and z
* **Type Compatibility:** Can we use a value of type A in a place that expects type B? Nontrivial with user-defined types and anonymous types

Type Checking vs. Type Inference

Standard type checking

int f(int x) { return x+1; }; 🡪 x is defined, 1 is literal

int g(int y) { return f(y+1)\*2; };

* Look at the body of each function and use declared types of identifiers to check agreement

Type inference

X

X

int f(int x) { return x+1; }; 🡪 f(x) { return x+1; };

X

X

int g(int y) { return f(y+1)\*2; }; 🡪 g(y) {return f(y+1)\*2; };

* Look at the code without type information and figure out what types could have been declared

ML is designed to make type inference tractable

Motivation

Types and type checking

* Type systems have improved steadily since Algol 60
* Important for modularity, compilation, reliability

Type inference (not many systems or languages)

* Widely regarded as important language innovation
* ML type inference is an illustrative example of a flow-insensitive static analysis algorithm

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Summary

Types are important in modern languages, they carry a lot of semantic information

* Organize and document the program, prevent errors, provide important information to compiler

Type inference

* Determine best type for an expression, based on known information about symbols in the expression

Polymorphism

* Single algorithm (function) can have many types

Overloading

* Symbol with multiple meanings, resolved when program is compiled