Contents

Specific Themes

Here are some specific themes that are addressed repeatedly in the text:

Computability: Some problems cannot be solved by computer. The undecidability of the halting problem implies that programming language compilers and interpreters cannot do everything that we might wish they could do.

Static analysis: There is a difference between compile time and run time. At compile time, the program is known but the input is not. At run time, the program and the input are both available to the run-time system. Although a program designer or implementer would like to find errors at compile time, many will not surface until run time. Methods that detect program errors at compile time are usually conservative, which means that when they say a program does not have a certain kind of error this statement is correct. However, compile-time error-detection methods will usually say that some programs contain errors even if errors may not actually occur when the program is run.

Expressiveness versus efficiency: There are many situations in which it would be convenient to have a programming language implementation do something automatically.

Computability

PARTIAL FUNCTIONS AND COMPUTABILITY

From a mathematical point of view, a program defines a function. In practice, there is a lot more to a program than the function it computes. However, as a starting point in the study of programming languages, it is useful to understand some basic facts about computable functions.

The fact that not all functions are computable has important ramifications for programming language tools and implementations. Some kinds of programming constructs,

however useful they might be, cannot be added to real programming languages because they cannot be implemented on real computers.

Expressions, Errors, and Nontermination

In mathematics, an expression may have a defined value or it may not. There is nothing to try to do when we see the expression 3/0; a mathematician would just say that this operation is undefined, and that would be the end of the discussion.

In computation, there are two different reasons why an expression might not have a value:

Error termination: Evaluation of the expression cannot proceed because of a conflict between operator and operand.

Nontermination: Evaluation of the expression proceeds indefinitely.

An example of the first kind is division by zero. There is nothing to compute in this case, except possibly to stop the computation in a way that indicates that it could not proceed any further. This may halt execution of the entire program, abort one thread of a concurrent program, or raise an exception if the programming language provides exceptions. The second case is different: There is a specific computation to perform, but the computation may not terminate and therefore may not yield a value.

Partial Functions

A partial function is a function that is defined on some arguments and undefined on others. This is ordinarily what is meant by function in programming, as a function declared in a program may return a result or may not if some loop sequence of recursive calls does not terminate. some loop

In words, a partial function is single valued, but need not be defined on all elements of its domain.

Computability

Computability theory gives us a precise characterization of the functions that are computable in principle. Computability theory gives us a precise characterization of the functions that are computable in principle. The The class

The reason why we say "computable in principle" instead of "computable in practice" is that some computable functions might take an extremely long time to compute. If a function call will not return for an amount of time equal to the length of the entire history of the universe, then in practice we will not be able to wait for the computation to finish.

Nonetheless, computability in principle is an important benchmark for programming languages.

Computable Functions

Intuitively, a function is computable if there is some program that computes it.

One problem with this intuitive definition of computable is that a program has to be written out in some programming language, and we need to have some implementation to execute the program. It might very well be that, in one programming language, there is a program to compute some mathematical function and in another language there is not.

In the 1930s, 1930s, Alonzo Church of Princeton University proposed an important principle, called Church's thesis. Church's thesis, states that the same class of functions on the integers can be computed by any general computing device. This is the class of partial recursive functions, sometimes called the class of computable functions. There is a mathematical definition of this class of functions that does not refer to programming languages, a second definition that uses a kind of idealized computing device called a Turing machine, and a third (equivalent) definition that uses lambda calculus a Turing machine consists of an infinite tape, a tape read-write head, and a finite-state controller. The tape is divided into contiguous cells, each containing a single symbol. In each computation step, the machine reads a tape symbol and the finite-state controller decides whether to write a different symbol on the current tape square and then whether to move the read-write head one square left or right. Church cited in formulating this thesis was the proof that Turing machines and lambda calculus are equivalent. The fact that all standard programming languages express precisely the class of partial recursive functions is often summarized by the statement that all programming languages are Turing complete.

Noncomputable Functions

It is useful to know that some specific functions are not computable. An important example is commonly referred to as the halting problem.

The undecidability of the halting problem is the fact that the function fhalt is not computable. The undecidability of the halting problem is an important fact to keep in mind in designing programming language implementations and optimizations. It implies that many useful operations on programs cannot be implemented, even in principle.

Applications

Programming language compilers can often detect errors in programs. However, the undecidability of the halting problem implies that some properties of programs cannot be determined in advance. The simplest example is halting itself.

SUMMARY

The following main concepts from this short overview should be remembered:

Partiality: Recursively defined functions may be partial functions. They are not always total functions. A function may be partial because a basic operation is not defined on some argument or because a computation does not terminate.

Computability: Some functions are computable and others are not. Programming languages can be used to define computable functions; we cannot write programs for functions that are not computable in principle.

Turing completeness: All standard general-purpose programming languages give us the same class of computable functions.

Undecidability: Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable.

Fundamentals

COMPILERS AND SYNTAX

A program is a description of a dynamic process. The text of a program itself is called its syntax; the things a program does comprise its semantics. The function of a programming language implementation is to transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur.

Structure of a Simple Compiler

Programming languages that are convenient for people to use are built around concepts and abstractions that may not correspond directly to features of the underlying machine. For this reason, a program must be translated into the basic instruction set of the machine before it can be executed. This can be done by a compiler, which translates the entire program into machine code before the program is run, or an interpreter, which combines translation and program execution. This can be done by a compiler, which translates the entire program into machine code before the program is run, or an interpreter, which combines translation and program execution.

Lexical

Lexical Analysis

The input symbols are scanned and grouped into meaningful units called tokens. For example, := x+1, five tokens: the identifier temp, the assignment "symbol" :=, the variable x, the addition symbol+, and the number 1. Lexical analysis can distinguish numbers from identifiers. However, because lexical analysis is based on a single left-to-right (and top-to-bottom)scan, lexical analysis does not distinguish between identifiers that are names of variables and identifiers that are names of constants. Because variables and constants are declared differently, variables and constants are distinguished in the semantic analysis phase.

Syntax Analysis

tokens are grouped into syntactic units such as expressions, statements, and declarations that must conform to the grammatical rules of the programming language. action performed during this phase, parsing, The purpose of parsing is to produce a data structure called a parse tree, which represents the syntactic structure of the program in a way that is convenient for subsequent phases of the compiler. If a program does not meet the syntactic requirements to be a well formed program, then the parsing phase will produce an error message and terminate the compiler.

Semantic Analysis

In this phase of a compiler, rules rules and procedures that depend on the context surrounding an expression are applied.

Intermediate Code Generation

Although it might be possible to generate a target program from the results of syntactic and semantic analysis, it is difficult to generate efficient code in one phase. Therefore, many compilers first produce an intermediate form of code and then optimize this code to produce a more efficient target program.

It is important to use an intermediate representation that is easy to produce and easy to translate into the target language. The intermediate representation can be some form of generic low-level code that has properties common to several computers.

Code Optimization

There are a variety of techniques that may be used to improve the efficiency of a program. These techniques are

usually applied to the intermediate representation.

The following list describes some standard optimizations:

Common Subexpression Elimination: If a program calculates the same value more than once and the compiler can detect this, then it may be possible to transform the program so that the value is calculated only once and stored for subsequent use.

Copy Propagation: If a program contains an assignment such as x=y, then it may be possible to change subsequent statements to refer to y instead of to x and to eliminate the assignment.

Dead-Code Elimination: If some sequence of instructions can never be reached, then it can be eliminated from the program.

Loop Optimizations: There are several techniques that can be applied to remove instructions from loops. For example, if some expression appears inside a loop but has the same value on each pass through the loop, then the expression can be moved outside the loop.

In-Lining Function Calls: If a program calls function f, it is possible to substitute the code for f into the place where f is called. This makes the target program more efficient, as the instructions associated with calling a function can be eliminated, but it also increases the size of the program. The most important consequence of in-lining function calls is usually that they allow other optimizations to be performed by removing jumps from the code.

Code Generation

The final phase of a standard compiler is to convert the intermediate code into a target machine code. This involves choosing a memory location, a register, or both, for each variable that appears in the program. There are a variety of register allocation algorithms that try to reuse registers efficiently. This is important because many machines have a fixed number of registers, and operations on registers are more efficient than transferring data into and out of memory.

Grammars

Grammars provide a convenient method for defining infinite sets of expressions. In addition, the structure imposed by a grammar gives us a systematic way of processing expressions.

A grammar consists of a start symbol, a set of nonterminals, a set of terminals, and a set of productions.

BNF.

The main ideas are illustrated by example. A simple language of numeric expressions is defined by the following grammar: e ::= n | e+e | e-e n ::= d | nd d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

e is the start symbol, e, n, and d are nonterminals, and 0, and 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, and are the terminals. The language defined by this grammar consists of all the sequences of terminals that we can produce by starting with the start symbol e and by replacing nonterminals according to the preceding productions.

Derivations

A sequence of replacement steps resulting in a string of terminals is called a derivation.

Here are two derivations in this grammar, the first given in full and the second with a few missing steps that can be filled in by the reader

e $\rightarrow$ n

$\rightarrow$ n $\rightarrow$ nd $\rightarrow$ dd $\rightarrow$ 2d $\rightarrow$ 25 e $\rightarrow$ e e $\rightarrow$ e e+e $\rightarrow$ ... $\rightarrow$ n-n+n $\rightarrow$ ... ... 10-15+12

Parse Trees and Ambiguity

It is often convenient to represent a derivation by a tree. This tree, called the parse tree of a derivation, or derivation tree, is constructed with the start symbol as the root of the tree.

The parse tree for the derivation of $10 - 15+12$ in the preceding subsection has some useful structure. Specifically, because the first step yields e-e, the parse tree has the form

which is another parse tree for the same expression. An important fact about parse trees is that each corresponds to a unique parenthesization of the expression.

A grammar is ambiguous if some expression has more than one parse tree. If every expression has at most one parse tree, the grammar is unambiguous.

Parsing and Precedence

Parsing is the process of constructing parse trees for sequences of symbols. Suppose we define a language L by writing out a grammar G. Then, given a sequence of symbols s, we would like to determine if s is in the language L. If so, then we would like to compile or interpret s, and for this purpose we would like to find a parse tree for s. An algorithm that decides whether s is in L, and constructs a parse tree if it is, is called a parsing algorithm for G.

Programming in Lambda Calculus

Functions of Several Arguments

Declarations

Recursion and Fixed Points

DENOTATIONAL SEMANTICS

In computer science, the phrase denotational semantics refers to a specific style of mathematical semantics for imperative programs. The term denotational semantics suggests that a meaning or denotation is associated with each program or program phrase

The denotation of a program is a mathematical object, typically a function, as opposed to an algorithm or a sequence of instructions to execute. In denotational semantics, the meaning of a simple program like

x := 0; y:=0; while x $\leq$ z do y:=y+x;x:=x+1

is a mathematical function from states to states, in which a state is a mathematical function representing the values in memory at some point in the execution of a program.

Associating mathematical functions with programs is good for some purposes and not so good for others. In many situations, we consider a program correct if we get the correct output for any possible input. This form of correctness depends on only the denotational semantics of a program, the mathematical function from input to output associated with the program.

Forms of denotational semantics are commonly used for reasoning about program optimization and static analysis methods.

Compositionality

An important principle of denotational semantics is that the meaning of a program is determined from its text compositionally. This means that the meaning of a program must be defined from the meanings of its parts, not something else, such as the text of its parts or the meanings of related programs obtained by syntactic operations.

Object Language and Metalanguage

One source of confusion in talking (or writing) about the interpretation of syntactic expressions is that everything we write is actually syntactic. When we study a programming language, we need to distinguish the programming language we study from the language we use to describe this language and its meaning.

States and Commands

The meaning of a program is a function from states to states. In a more realistic programming language, with procedures or pointers, it is necessary to model the fact that two variable names may refer to the same location. However, in the simple language of while programs, we assume that each variable is given a different location. State of mathematical representations of machine states be

State = Variables $\rightarrow$ Values

In words, a state is a function from variables to values.

The meaning of a program is an element of the mathematical set Command of commands, defined by Command = State $\rightarrow$ State

In words, a command is a function from states to states. Unlike states themselves, which are total functions, a command may be a partial function. The reason we need partial functions is that a program might not terminate on an initial state.

Perspective and Nonstandard Semantics

FUNCTIONAL AND IMPERATIVE LANGUAGES

Imperative and Declarative Sentences

In an imperative sentence, the subject of the sentence is implicit. For example, the subject of the sentence Pick up that fish is (implicitly) the person to whom the command is addressed. A declarative sentence expresses a fact and may consist of a subject and a verb or subject, verb, and object.

In many programming languages, the basic constructs are imperative statements. For example, an assignment statement such as x:=5

is a command to the computer (the implied subject of the utterance) to store the value5 in a certain location. Programming languages also contain declarative constructs such as the function declaration

function f(int x) {return x+1;} that states a fact. One reading of this as a declarative sentence is that the subject is the name f and the sentence about f is "f is a function whose return value is 1 greater than its argument." In programming, the distinction between imperative and declarative constructs rests on the distinction between changing an existing value and declaring a new value. example,

{ int x=1; /* declares new x */ *x = x+1;* / assignment to existing x / { *int y = x+1;* / declares new y / { *int x = y+1;* / declares new x */ }}}

Here, only the second line is an imperative statement. This is an imperative command that changes the state of the machine by storing the value 2 in the location associated with variable x. The other lines contain declarations of new variables.

Functional versus Imperative Programs

The phrase functional language is used to refer to programming languages in which most computation is done by evaluation of expressions that contain functions. Two examples are Lisp and ML. Both of these languages contain declarative and imperative constructs. However, it is possible to write a substantial program in either language without using any imperative constructs.

Some people use the phrase functional language to refer to languages that do not have expressions with side effects or any other form of imperative construct. However, we will use the more emphatic phrase pure functional language for declarative languages that are designed around flexible constructs for defining and using functions.

As a consequence, pure functional languages have a useful optimization property: If expression e occurs several places within a specific scope, this expression needs to be evaluated only once.

Referential Transparency

In some of the academic literature on programming languages, including some textbooks on programming language semantics, the concept that is used to distinguish declarative from imperative languages is called referential transparency. the concept that is used to distinguish declarative from imperative languages is called referential transparency.

it is traditional to say that a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program. This is a property of pure functional languages. a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program.

Functional Programming and Concurrency

An appealing aspect of pure functional languages and of programs written in the pure functional subset of larger languages is that programs can be executed concurrently. This is a consequence of the declarative language test mentioned at the beginning of this subsection.

Functional Programming: We can evaluate f(e1,...,en) by evaluating e1,...,en in parallel because values of these expressions are independent.

Imperative Programming: For an expression such as f(g(x), h(x)), the function g might change the value of x. Hence the arguments of functions in imperative

languages must be evaluated in a fixed, sequential order. This order ing restricts the use of concurrency.

CHAPTER SUMMARY

A standard compiler transforms an input program, written in a source language, into an output program, written in a target language. This process is organized into a series of six phases, The first three phases, lexical analysis, syntax analysis, and semantic analysis, organize the input symbols into meaningful tokens, construct a parse tree, and determine context-dependent properties of the program such as type agreement of operators and operands. The last three phases, intermediate code generation, optimization, and target code generation, are aimed at producing efficient target code through language transformations and optimizations. generation, are aimed at producing

Lambda calculus provides a notation and symbolic evaluation mechanism that is useful for studying some properties of programming languages.

Lambda expressions are symbolically evaluated by use of -reduction, with the function argument substituted in place of the formal parameter. This process resembles macro

expansion and function in lining, two transformations that are commonly done by compilers. Although lambda calculus is a very simple system, it is theoretically possible to write every computable function in the lambda calculus.

Qué es y qué puede hacer un lenguaje de programación

Un lenguaje de programación es un lenguaje formal dise˜nado para realizar procesos que pueden ser llevados a cabo por máquinas Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina o para expresar algoritmos con precisión

Sintaxis y semántica

Los lenguajes

son sistemas que se sirven de una **forma** para comunicar un **significado.** Lo que tiene que ver con la forma recibe el nombre de

sintaxis

lo que tiene que ver con el significado recibe el nombre de

semántica.

la forma

son los programas

el significado

es lo que los programas hacen,

Un lenguaje de programación

se describe con su **sintaxis** (qué es lo que se puede escribir legalmente en ese lenguaje) y su **semántica** (qué efectos tiene en la máquina lo que se escribe en ese lenguaje).

Alcance de los lenguajes de programación

algoritmos,

conjunto de instrucciones bien definidas, ordenadas y finitas que permite realizar algo de forma inambigua mediante pasos.

funciones computables

una función es computable si existe un algoritmo que puede hacer el trabajo de la función, es decir, dada una entrada del dominio de la función puede devolver la salida correspondiente.

tesis de Church-Turing1

las funciones computables son exactamente las funciones que se pueden calcular utilizando un dispositivo de cálculo mecánico dada una cantidad ilimitada de tiempo y espacio de almacenamiento. De manera equivalente, esta tesis establece que cualquier función que tiene un algoritmo es computable.

función no computable famosas

Halting problem o calcular la Complejidad de Kolmogorov.

Sintaxis a través de gramáticas

objetivo con respecto a la sintaxis

es describir de forma compacta e inambigua el conjunto de los programas válidos

gramáticas independientes de contexto.

El estándar para gramáticas independientes de contexto de lenguajes de programación es EBNF. Sin embargo, las gramáticas independientes de contexto no son suficientemente expresivas para describir adecuadamente la mayor parte de lenguajes de programación. Por ejemplo es difícil expresar la obligación de que una variable sea declarada antes de ser usada, o bien describir la asignación múltiple de variables,

Semántica operacional vs. lambda cálculo

La semántica operacional

describe formalmente cómo se llevan a cabo cada uno de los pasos de un cálculo en un sistema computacional. Para eso se suele trabajar sobre un modelo simplificado de la máquina. Cuando describimos la semántica de un programa mediante semántica operacional, describimos cómo un programa válido se interpreta como secuencias de pasos computacionales. Estas secuencias son el significado del programa.

Cómo funcionan los lenguajes de programación

Estructura de un compilador

Mitchell 4.1.1.

Estructuras de datos de bajo nivel

Variables

Una variable está formada por una ubicación en la memoria y un identificador asociado a esa ubicación. Esa ubicación contiene un valor, El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa.

Los compiladores

reemplazan los nombres simbólicos de las variables con la real ubicación de los datos. Diferentes identificadores del código pueden referirse a una misma ubicación en memoria, lo cual se conoce como aliasing.

aliasing.

La ubicación de una variable se llama su L-valor

L-valor

el valor almacenado en esta ubicación se llama el R-valor de la variable.

R-valor

Estructura en bloques

Código estructurado vs. código spaghetti

programas que resulten incomprensibles porque al que lee le cuesta entender la estructura de control del programa. A este tipo de código se le llama código spaghetti,

código spaghetti,

bloque.

Un bloque es una región del texto del programa con inicio y fin explícitos e inambiguos. Esta región del texto permite organizar de forma explícita la lógica del programa. además, posibilita sucesivas abstracciones sobre el flujo de ejecución.

En primer lugar,
los bloques nos permiten hacer declaraciones de variables locales.

variable local

Una variable declarada dentro de un bloque se dice que es una variable local para ese bloque. Las variables que no están declaradas en un bloque, sino en algún otro bloque que lo contiene, se dice que es una variable global para el bloque más interno.

variable global

Estructura de bloque

propiedades:

Las nuevas variables se pueden declarar en varios puntos de un programa.

nuevas variables
Cada declaración es visible dentro de una determinada región de texto del programa, llamada bloque.

bloque.
si dos bloques contienen expresiones o declaraciones en común, entonces un bloque debe estar enteramente contenida dentro del otro.

Cuando un programa inicia la ejecución de las instrucciones contenidas en un bloque en tiempo de ejecución, se asigna memoria a las variables declaradas en ese bloque.

memoria
Cuando un programa sale de un bloque, parte o toda la memoria asignada a las variables declaradas en ese bloque se libera.

identificador de variable
Un identificador de variable que no está declarado en el bloque actual se considera global a ese bloque, y su referencia es a la entidad con el mismo identificador nombre que se encuentra en el bloque más cercano que contiene al bloque actual.

cómo se manejan en memoria tres clases de variables:

variables locales
se almacenan en la pila de ejecución, en el activation record asociado al bloque.

parámetros de función
también se almacenan en el activation record asociada con el bloque.

variables globales
que se declaran en algún bloque que contiene al bloque actual y por lo tanto hay que acceder a ellos desde un activation record que se colocó en la pila de ejecución antes del bloque actual.

Activation records

El contador de programa

es la dirección donde se

encuentra la instrucción de programa que se está ejecuntando actualmente.

El puntero de entorno

nos sirve para saber cuáles son los valores que se asignan a las variables que se están usando en una parte determinada del código.

El stack

cuando el programa entra en un nuevo bloque, se agrega a la pila una estructura de datos que se llama **activation record** (stack frame), que contiene el espacio para las variables locales declaradas en el bloque, Entonces, **el puntero de entorno** apunta al nuevo activation record. Cuando el programa sale del bloque, se retira el activation record de la pila y el puntero de entorno se restablece a su ubicación anterior, El **activation record** que se apila más recientemente es el primero en ser desapilado,

control link,

contiene el que será el puntero de entorno cuando se desapile el activation record actual,

dirección de retorno,

es donde se va a guardar el resultado de la ejecución de la función, si es que lo hay.

Detalle de ejecución de un activation record

bloques in line

Un bloque in line es un bloque que no es el cuerpo de una función o procedimiento.

un activation record también puede contener espacio para resultados intermedios, Estos son valores que no reciben un identificador de variable explícito en el código, pero que se guardan temporalmente para facilitar algún cálculo.

los valores de estas subexpresiones pueden tener que ser evaluados y almacenados en algún lugar antes de multiplicarse. # Scope, Functions, and Storage Management

important topics in this chapter

parameter passing, access to global variables, storage optimization with a function call called **tail call.**

BLOCK-STRUCTURED LANGUAGES

Most modern programming languages provide some form of block.

block
is a region of program text, identified by begin and end markers, that may contain declarations local to this region.

A variable declared within a block is local to that block. A variable declared in an enclosing block is global to the block.

Storage management mechanisms associated with block structure allow functions to be called recursively.

Block-structured languages

New variables may be declared at various points in a program.

Each declaration is visible within a certain region of program text,

Blocks may be nested, but cannot partially overlap.

When a program begins executing the instructions contained in a block

memory is allocated for the variables

When a program exits a block,

the memory allocated to variables declared in that block will be deallocated.

An identifier that is not declared in the current block is considered global

local variables,
are stored on the stack in the activation record associated with the block

parameters
are also stored in the activation record associated with the block

global variables,
are declared in some enclosing block and therefore must be accessed from an activation record that was placed before activation of the current block.

Simplified Machine Model

Reference Implementation.
is an implementation of a language that is designed to define the behavior of the language.

IN-LINE BLOCKS

is a block that is not the body of a function or procedure.

Activation Records and Local Variables

When a running program enters an in-line block, space must be allocated for variables thataredeclaredintheblock.Wedothisbyallocatingasetofmemorylocationscalled **activation record** on the run-time stack.

The number of locations that need to be allocated at run time depends on the number of variables declared in the block and their types.

Intermediate Results

an activation record may also contain space for intermediate results. These are values that are not given names in the code, but that may need to be saved temporarily.

Scope and Lifetime

It is important to distinguish the scope of a declaration from the lifetime of a location:

Scope:

a region of text in which a declaration is visible.

Lifetime:

the duration, during a run of a program, during which a location is allocated as the result of a specific declaration.

Blocks and Activation Records for ML

In ML code that has sequences of declarations, we treat each declaration as a separate block.

When an ML expression contains declarations as part of the let-in-end construct, we consider the declarations to be part of the same block.

Global Variables and Control Links

operations that push and pop activation records from the run-time stack store a pointer to the top of the preceding activation record. The pointer to the top of the previous activation record is called **control link,** is the link that is followed when control returns to the instructions in the preceding block.

When a new activation record is added to the stack,

the control link of the new activation record is set to the previous value of the environment pointer, and the pointer is updated to point to the new activation record. When an activationrecordispoppedoffthestack,theenvironmentpointerisresetbyfollowing the control link from the activation record.

FUNCTIONS AND PROCEDURES

Most block-structured languages have procedures or functions that include parameters, local variables, and a body consisting of an arbitrary expression or sequence of statements.

The difference between a **procedure** and a **function** is that a function has a return value but a procedure does not. a procedure call is a statement and not an expression.

Activation Records for Functions

The **activation record** of a function or procedure block must include space for parameters and return values. a procedure may be called from different call sites, it is necessary to save the return address, **For functions,** the activation record must also contain the location that the calling routine expects to have filled with the return value of the function.

The activation record associated with a function

must contain space for the following information:

control link,

access link,

return address,

return-result address,

actual parameters of

local variables

temporary storage

Parameter Passing

The parameter names used in a function declaration are called **formal parameters.** When a function is called, expressions called **actual parameters** are used to compute the parameter values for that call.

The way that actual parameters are evaluated and passed to the function depends on the programming language The main distinctions between different parameter-passing mechanisms are

the time that the actual parameter is evaluated

the location used to store the parameter value.

Most current programming languages evaluate the actual parameters before executing the function body,

Among mechanisms that evaluate the actual parameter before executing the function body, the most common are

Pass-by-reference:

Pass-by-value:

The difference between pass-by-value and pass-by-reference is important to the programmer in several ways:

Side Effects.

Assignments inside the function body may have different effects under pass-by-value and pass-by-reference.

Aliasing.

Aliasing occurs when two names refer to the same object or location.

Efficiency.

Pass-by-value may be inefficient for large structures if the value of the large structure must be copied.

There are two ways of explaining the semantics of call-by-reference and call-byvalue. One is to draw pictures of computer memory and the run-time program stack, showing whether the stack contains a copy of the actual parameter or a reference to it. The other explanation proceeds by translating code into a language that distinguishes between Land R-values.

Semantics of Pass-by-Value

the actual parameter is evaluated. The value of the actual parameter is then stored in a new location allocated for the function parameter.

Semantics of Pass-by-Reference

the actual parameter must have an L-value. The L-value of the actual parameter is then bound to the formal parameter.

Global Variables (First-Order Case)

If an identifier **x** appears in the body of a function, but **x** is not declared inside the function, then the value of **x** depends on some declaration outside the function.

There are two main rules for finding the declaration of a global identifier:

Static Scope:

A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text.

Dynamic Scope:

A global identifier refers to the identifier associated with the most recent activation record.

static scope uses the static relationship between blocks in the program text. **dynamic scope** uses the actual sequence of calls that are executed in the dynamic execution of the program.

Access Links are Used to Maintain Static Scope

The **access link** of an activation record points to the activation record of the closest enclosing block in the program.

Tail Recursion (First-Order Case)

a useful compiler optimization For tail recursive functions, it is possible to reuse an activation record for a recursive call to the function. # Control in Sequential Languages

STRUCTURED CONTROL

Spaghetti Code

it is easy to write programs with incomprehensible control structure.

Structured Control

In the 1960s, programmers began to understand that unstructured jumps could make it difficult to understand a program. This led to the development of some constructs that structure jumps:

if . . . then . . . else . . . end while . . . do . . . end for . . . { . . . } case . . .

These are now adopted in virtually all modern languages.

In modern programming style, we group code in logical blocks,

avoid explicit jumps and cannot jump into the middle of a block

Paradigma funcional

Expresiones imperativas vs. expresiones funcionales

En muchos lenguajes las construcciones básicas son imperativas, Por ejemplo, esta instrucción de asignación:

x: = 5

es una orden que el programador le da a la computadora para guardar el valor 5 en un lugar determinado. Los lenguajes de programación también contienen construcciones declarativas, como la declaración de la función

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

La distinción entre construcciones imperativas y declarativas se basa en que las imperativas cambian un valor y las declarativas declaran un nuevo valor.

La forma más sencilla de entender la diferencia entre declarar una nueva variable y cambiar el valor de una variable ya existente es cambiando el nombre de la variable. **Las variables ligadas,** pueden cambiar de nombre sin cambiar el significado de la expresión.

La asignación imperativa puede introducir **efectos secundarios** porque puede destruir el valor anterior de una variable, **En programación funcional** la asignación imperativa se conoce como **actualización destructiva.** una operación computacional es **declarativa** si, cada vez que se invoca con los mismos argumentos, devuelve los mismos resultados independientemente de cualquier otro estado de computación. Una operación declarativa es:

independiente

sin estado y

determinística

Una consecuencia de estas propiedades es la **transparencia referencial.** Una expresión es transparente referencialmente si se puede sustituir por su valor sin cambiar la semántica del programa. Esto hace que todas las componentes declarativas, se puedan usar como valores en un programa,

Veamos un ejemplo de dos funciones, **una referencialmente transparente** y **otra referencialmente opaca:**

globalValue = 0;

integer function rq(integer x) begin

globalValue = globalValue + 1; return x + globalValue;

end

integer function rt(integer x) begin

return x + 1;

end

La función **rt** es referencialmente transparente, significa que **rt(x) = rt(y)** si x = y. Sin embargo, no podemos decir lo mismo de rq porque usa y modifica una variable global.

la transparencia referencial − nos permite razonar sobre nuestro código de forma que podemos construir programas más robustos,

Propiedades valiosas de los lenguajes funcionales

Los lenguajes funcionales, realizan la mayor parte del cómputo mediante expresiones con declaraciones de funciones.

En algunos casos se usa el término "lenguaje funcional" para referirse a lenguajes que no tienen expresiones con efectos secundarios Para evitar ambig¨uedades entre estos y lenguajes como **Lisp o ML,** llamaremos a estos últimos **"lenguajes funcionales puros".** Los lenguajes funcionales puros pueden pasar el siguiente test:

Dentro del alcance de las declaraciones x1, ..., xn, todas las ocurrencias de una expresión e que contenga sólo las variables x1, ..., xn tendrán el mismo valor.

Como consecuencia los lenguajes funcionales puros tienen una propiedad muy útil: **si la expresión e ocurre en varios lugares dentro de un alcance específico, entonces la expresión sólo necesita evaluarse una vez.** Otra propiedad interesante de los programas declarativos es que son **composicionales.** Un programa declarativo consiste de componentes que pueden ser escritos, comprobados, y probados correctos independientemente de otros componentes Otra propiedad interesante es que **razonar**

sobre programas declarativos es sencillo. Es más fácil razonar sobre programas escritos en el modelo declarativo que sobre programas escritos en modelos más expresivos. Estas dos propiedades son importantes tanto para programar en grande como en peque˜no.

Problemas naturalmente no declarativos

La forma más elegante y natural de representar algunos problemas es mediante **estado explícito.** En general, todo programa que realice algún tipo de Input-Output lo hará de forma más natural mediante estado explícito.

También hay tipos de problemas que se pueden programar más eficiente si se hace de forma imperativa. En esos casos podemos llegar a convertir un problema **intratable** con programación declarativa en un problema **tratable.** Este es el caso de un programa que realiza modificaciones incrementales de estructuras de datos grandes, Por esta razón muchos lenguajes funcionales proveen algún tipo de construcción para poder expresar instrucciones imperativas. estas construcciones, se conocen como **mónadas.** Las mónadas son una construcción de un lenguaje que permite crear un alcance aislado del resto del programa. Dentro de ese alcance, se permiten ciertas operaciones con efectos secundarios, Está garantizado que estos efectos secundarios no van a afectar a la parte del programa que queda fuera del alcance de la mónada.

Concurrencia declarativa

Una consecuencia muy valiosa de los programas escritos de forma funcional pura, **sin expresiones con efectos secundarios,** es que se pueden ejecutar de forma concurrente con otros programas con la garantía de que su semántica permanecerá inalterada. paralelizar programas funcionales puros es trivial. En cambio, para paralelizar programas imperativos, es necesario detectar primero las posibles regiones del programa donde puede haber condiciones de carrera, aunque paralelizar programas funcionales sea trivial, es difícil aprovechar este potencial en algunos puntos de un programa tiene sentido introducir paralelismo y en otros no. El paralelismo tiene un peque˜no coste en la creación de nuevos procesos, # Type Systems and Type Inference

6.1 TYPES IN PROGRAMMING

type

is a collection of computational entities that share some common property. Some examples of types are the type int of integers, the type int→int of functions from integers to integers,

there is no such thing as an untyped programming language.

6.1.1 Program Organization and Documentation

Using types to organize a program

makes it easier for someone to read, understand, and maintain the program. Types therefore serve an important purpose in documenting the design and intent of the program.

An important advantage of type information,

in comparison with comments written by a programmer, is that types may be checked by the programming language compiler. In contrast, many programs contain incorrect comments,

6.1.2 Type Errors

A **type error** occurs when a computational entity, such as a function or a data value, is used in a manner that is inconsistent with the concept it represents.

Hardware Errors.

a machine instruction that results in a hardware error.

If x is an integer variable with value 256, then executing x() will cause the machine to jump to location 256 and begin executing the instructions stored at that place in memory. If location 256 contains data that do not represent a valid machine instruction, this will cause a hardware interrupt.

Unintended Semantics.

compiled code does not contain the same information as the program source code does.

int add(3, 4.5)

is a type error, as int add is an integer operation and is applied here to a floating-point number. Most hardware would perform this operation. int add is intended to perform addition, but the result of int add(3, 4.5) is not the arithmetic sum of the two operands.

6.1.3 Types and Optimization

Type information in programs can be used for many kinds of optimizations.

Some operations can be computed more efficiently if the type of the operand is known at compile time.

6.2 TYPE SAFETY AND TYPE CHECKING

6.2.1 Type Safety

A programming language is **type safe** if no program is allowed to violate its type distinctions. a function has a different

type from an integer. Therefore, any language that allows integers to be used as functions is not type safe.

form of type error

Type Casts.

Type casts allow a value of one type to be used as another type.

Pointer Arithmetic.

an assignment like x = *(p+i) may store a value of one type into a variable of another type and therefore may cause a type error.

Explicit Deallocation and Dangling Pointers.

the location reached through a pointer may be deallocated (freed) by the programmer. This creates a **dangling pointer,** If p is a pointer to an integer, then after we deallocate the memory referenced by p, the program can allocate new memory to store another type of value. This new memory may be reachable through the old pointer p,

6.2.2 Compile-Time and Run-Time Checking

Run-Time Checking.

the compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct type. **An advantage** of run-time type checking is that it catches type errors. **A disadvantage** is the run-time cost associated with making these checks.

Compile-Time Checking.

reject programs that do not pass the compile-time type checks. **An advantage** is that it catches errors earlier than run-time checking does: compiletime checking can make it possible to produce more efficient code.

Conservativity of Compile-Time Checking.

A property of compile-time type checking is that the compiler must be conservative. This mean that compile-time type checking will find all statements and expressions that produce run-time type errors, but also may flag statements or expressions as errors even if they do not produce run-time errors.

Combining Compile-Time and Run-Time Checking.

Most programming languages actually use some combination of compile-time and run-time type checking.

6.3 TYPE INFERENCE

Type inference is the process of determining the types of expressions based on the known types of some symbols that appear in them.

difference between type inference and compile-time type checking

is really a matter of degree. A **type-checking** algorithm goes through the program to check that the types declared by the programmer agree with the language requirements. In **type inference,** the idea is that some information is not specified, and some form of logical inference is required for determining the types of identifiers from the way they are used.

6.3.2 Type-Inference Algorithm

The ML type-inference algorithm uses the following three steps:

1. A assign a type to the expression and each subexpression. For any

compound expression or variable, use a type variable. For **known operations or constants,** such as + or 3, use the type that is known for this symbol.

2. Generates a set of constraints on types, using the parse tree of the expression.
3. Solve these constraints by means of unification, which is a substitution-based algorithm for solving systems of equations.

6.4 POLYMORPHISM AND OVERLOADING

Polymorphism,

which literally means "having multiple forms," refers to constructs that can take on different types as needed.

forms of polymorphism

parametric polymorphism,
in which a function may be applied to any arguments whose types match a type expression involving type variables;

ad hoc polymorphism,
another term for overloading, in which two or more implementations with different types are referred to by the same name;

subtype polymorphism,
in which the subtype relation between types allows an expression to have many possible types.

6.4.1 Parametric Polymorphism

main characteristic
the set of types associated with a function or other value is given by a type expression that contains type variables.

In parametric polymorphism,
a function may have infinitely many types, as there are infinitely many ways of replacing type variables with actual types.

explicit parametric polymorphism,
the program text contains type variables that determine the way that a function or other value may be treated polymorphically.

implicit parametric polymorphism
programs that declare and use polymorphic functions do not need to contain types

6.4.3 Overloading

A symbol is **overloaded** if it has two (or more) meanings, distinguished by type, and resolved at compile time. # Control de la ejecución

Pasaje de parámetros

Los nombres de los parámetros cuando se declara una función se llaman parámetros formales. Cuando se llama a una función, se usan los llamados parámetros reales, que son los valores efectivos de los parámetros formales en una llamada concreta a la función.

Los distintos mecanismos de pasaje de parámetros se diferencian por el momento en el que se evalúa el parámetro real (evaluación estricta, en el momento de pasar el parámetro, o perezosa, cuando se necesita) y por la ubicación de memoria donde se almacena el valor

del parámetro (la misma ubicación del bloque que hace la llamada a función o una nueva ubicación específica de la función). La mayoría de los lenguajes de programación actuales evalúan los parámetros reales de forma estricta, no perezosa, Entre los mecanismos los más comunes son el pasaje por referencia y el pasaje por valor. se diferencian entre ellos por la ubicación de memoria donde se almacena el valor del parámetro. El pasaje por referencia por por referencia pasa el L-valor, es decir, la ubicación en memoria, del parámetro real, mientras que el pasaje por valor pasa el R-valor, es decir, el valor que hay en la ubicación de memoria. es importante por varias razones:

Efectos secundarios Las asignaciones de valor en el interior del cuerpo de la función pueden tener diferentes efectos.

Aliasing el aliasing ocurre cuando dos identificadores de variable se refieren al mismo objeto o ubicación.

Eficiencia Pasar por valor puede ser ineficaz para grandes estructuras de datos, si hay que copiar ese valor. Pasar por referencia puede ser menos eficiente que pasar por valor peque˜nas estructuras que se adapten bien en la pila, porque cuando los parámetros se pasan por referencia, debemos resolver la referencia de un puntero para obtener su valor.

dos formas de describir la semántica del pasaje de parámetros: con diagramas de la memoria y la pila que muestren si la pila contiene una copia del parámetro real o una referencia. Otra es traducir el código a un lenguaje que distinga entre R-valores y L-valores,

Semántica de pasaje por valor (call-by-value)

pasaje por valor es la estrategia de evaluación más común en los lenguajes de programación. Es una forma estricta de pasaje de parámetros, el argumento se evalúa, y el valor que se obtiene (su R-valor) se liga a una variable local de la función, en general copiando el valor a una nueva ubicación de memoria en el activation record de la llamada a función. El valor que se encuentra en el activation record que llama a la función no es modificado en ningún momento.

Semántica de pasaje por referencia call-by-reference

En pasaje por referencia se liga el L-valor del parámetro real al parámetro formal. En este caso se tiene un único valor referenciado (o apuntado) desde dos puntos diferentes, el programa principal y la función a la que se le pasa el argumento, por lo que cualquier acción sobre el parámetro se realiza sobre la misma posición de memoria (aliasing). Esto significa que la función que es llamada puede modificar la variable con efecto en el bloque que llama a la función.

Sutilezas entre pasaje por valor y pasaje por referencia

En lenguajes puramente funcionales típicamente no hay diferencia semántica entre pasaje por valor y pasaje por referencia, porque sus estructuras de datos son inmutables, por lo que es imposible que una función pueda modificar sus argumentos. normalmente se describen como de pasaje por valor, a pesar de que las implementaciones utilizan normalmente pasaje por referencia internamente porque es más eficiente. En algunos casos de pasaje por valor el valor que se pasa es una referencia a una ubicación de memoria. sintácticamente parece llamada por valor puede terminar teniendo los efectos de un pasaje por referencia. Lenguajes como C y ML utilizan esta técnica.

varias razones por las que se puede pasar una referencia. Una razón puede ser que el lenguaje no permita el pasaje por valor de estructuras de datos complejas, y se pase una referencia para preservar esa estructura. También se usa de forma sistemática en lenguajes con orientación a objetos, lo que se suele llamar call-by-sharing, call-by-object o call-by-object-sharing. **call-by-sharing, el pasaje de parámetros es por valor, pero se pasa una referencia a un objeto. Por lo tanto, se hace una copia local del argumento que se pasa, pero esa copia es una referencia a un objeto que es visible tanto desde la función llamada como desde la función que llama.** La semántica difiere del pasaje por referencia en que las asignaciones a los argumentos dentro de la función no son visibles a la función que llama, si se pasa una variable, no es posible simular una asignación en esa variable en el alcance de la función que llama. Por lo tanto, no se pueden hacer cambios a la referencia que se ha pasado por valor. a través de esa referencia de la que la función llamada ha hecho una copia, ambas funciones, la llamada y la que llama, tienen acceso al mismo objeto, no se hace ninguna copia. Si

ese objeto es mutable, los cambios que se hagan al objeto dentro de la función llamada son visibles para la función que llama, a diferencia del pasaje por valor. el término call-by-sharing no se usa no se usa mucho, se usan diferentes términos en los diferentes lenguajes y documentaciones.

Semántica de pasaje por valor-resultado

Es un tipo de pasaje de parámetros con evaluación estricta (no perezosa) poco usado en los lenguajes de programación actuales. Se basa en que dentro de la función se trabaja como si los argumentos hubieran sido pasados por valor, pero al acabar la función los valores que tengan los argumentos serán copiados a la ubicación de memoria en la que se ubicaba el valor copiado inicialmente.

Pasaje de parámetros no estricto (perezoso)

En la evaluación perezosa, los argumentos de una función no se evalúan a menos que se utilicen efectivamente en la evaluación del cuerpo de la función.

Pasaje por nombre call-by-name

En el pasaje call-by-name, los argumentos de una función no se evalúan antes de la llamada a la función, sino que se sustituyen sus nombres directamente en el cuerpo de la función y luego se dejan para ser evaluados cada vez que aparecen en la función. Si un argumento no se utiliza en el cuerpo de la función, el argumento no se evalúa; si se utiliza varias veces, se re-evalúa cada vez que aparece. En algunas ocasiones, call-by-name tiene ventajas sobre pasaje por valor, por ejemplo, si el argumento no siempre se usa, porque nos ahorramos la evaluación del argumento. Si el argumento es un cálculo no termina, la ventaja es enorme.

Pasaje por necesidad call-by-need

El pasaje por necesidad call-by-need es una versión memoizada de pasaje por nombre donde, si se evalúa el argumento de la función, ese valor se almacena para usos posteriores. En un entorno de ”puro”(sin efectos secundarios), esto produce los mismos esto produce los mismos resultados que el pasaje por nombre; cuando el argumento de la función se utiliza dos o más veces, el pasaje por necesidad es casi siempre más rápido.

Alcance estático vs. alcance dinámico

Si un identificador x aparece en el cuerpo de una función, pero x no se declara dentro de la función, entonces el valor de x depende de alguna declaración fuera de la función. la ubicación de x está fuera del registro de activación para la función y es una variable libre o variable global respecto a esa función. Debido a que x ha sido declarada en otro bloque, Hay dos políticas principales para buscar la declaración adecuada de un identificador de variable libre:

alcance estático

---

léxico)

Un identificador global se refiere al identificador con ese nombre que se encuentre en el bloque contenedor más cercano en el texto del programa.

alcance dinámico Un identificador global se refiere al identificador que se encuentre en el registro de activación más reciente en la pila de ejecución.

la búsqueda de una declaración usando la política de alcance estático utiliza la relación estática e inmutable entre bloques en el texto del programa. Por el contrario, el alcance dinámico utiliza la secuencia efectiva de las llamadas que se ejecutan en la ejecución del programa, que es dinámica y puede cambiar.

Naturalidad y overhead del alcance estático

El alcance estático es la semántica más intuitiva los lenguajes de programación proveen los mecanismos necesarios para poder mantener esta semántica. Hay diferentes opciones para hacerlo. La opción más sencilla es prohibir los contextos en los que el alcance estático daría un resultado distinto al alcance dinámico.

Sin embargo, en la mayoría de lenguajes se usan access links para mantener la semántica de alcance estático. **Un access link es una dirección de memoria que se guarda en el activation record de una función y que apunta al activation record del bloque más cercano que lo contiene en el texto**

del programa. Para los bloques anidados in-line esta dirección de memoria es redundante con el control link, ya que el bloque más cercano que lo contiene siempre se corresponde con el activation record que se ha apilado inmediatamente antes. Para las funciones, el bloque más cercano que la contiene viene determinado por el lugar donde

se declara la función. Debido a que el punto de declaración a menudo es diferente del punto en el que se llama a una función, el access link generalmente apunta a un registro de activación diferente al enlace de control.

Recursión a la cola

optimización del compilador que se llama eliminación de la recursión a la cola. Para las funciones recursivas a la cola, que describimos a continuación, se puede reutilizar un activation record para una llamada recursiva a la función. Esto reduce la cantidad de espacio de la pila que usa una función recursiva, y

evita llegar a problemas por límites de hardware como el llamado stack overflow, en el que la ejecución de un programa requiere más espacio del que hay disponible en la pila.

Una función f es recursiva de cola si todas las llamadas recursivas en el cuerpo de f son llamadas a la cola a f.

La ventaja de la recursión de cola es que podemos utilizar el mismo activation record para todas las llamadas recursivas.

Alto orden

Funciones de primera clase

Un lenguaje tiene funciones de primera clase si las funciones pueden ser

declaradas dentro de cualquier alcance,

pasadas como argumentos a otras funciones y,

devueltas como resultado de funciones.

En un lenguaje con funciones de primera clase y con alcance estático, un valor de función se representa generalmente por una clausura, un par formado por un puntero al código del cuerpo de una función y otro puntero a un activation record, con una semántica prácticamente equivalente a un access link.

Pasar Funciones a otras Funciones

Vamos a ver que cuando una función f se pasa a una función g, es posible que tengamos que pasar también la clausura de f, que contiene un puntero a su activation record. Cuando f se llama dentro del cuerpo de g, el puntero de entorno de la clausura se utiliza para determinar cuál es el access link que tiene que guardarse en el activation record de la llamada a f. La necesidad de clausuras en este contexto se conoce como el problema

de downward funarg, porque sucede cuando se pasan funciones como argumentos hacia abajo en alcances anidados.