**CHAPTER 1**

Why Study Programming Languages?

1. To aid in learning new languages.

2. To aid in the development of new languages.

3. To aid in compiler/interpreter design.

4. To increase programming language vocabulary.

5. To improve algorithmic skills.

6. To aid in the selection of an appropriate language for an application.

7. To encourage “good” languages to thrive and “bad” languages to die.

What may be considered “good” in some contexts is not “good” in others.

Characteristics of a “Good” Programming Language

1. Readability – measure of how well a program written in the language is understandable to another reader. This is a measure of maintainability.

a. Overall simplicity

1) Small number of features or components

a) Lisp

2) Only one method to accomplish any task; lack of redundancy

a) count++ and count = count + 1

3) No overloading

a) Using a plus sign between two ints vs two floats. These have to be handled explicitly. This is more of a problem with set union vs addition.

b. Orthogonality – relatively small number of language primitives which can be combined to form larger constructs.

1) Example:

a) Algol 68

c. Strong set of control statements

1) For example, using “while” statements vs “goto” statements

2) Examples:

a) Pascal

b) Ada

d. Rich set of data types and data structures

1) Examples:

a) PL1

e. Rules for forming identifiers names

1) Examples of unlimited length

a) C++

b) Ada

f. Use of reserved words vs keywords

1) Keywords – words that are generally set aside, but can be overwritten

a) FORTRAN – this was done so that the new versions could be updated without crashing the program or without changing the code.

2) Reserved words – the words are completely reserved for the definition of that the language provides (easier on compiler) (better for readability)

a) C++

2. Writeability – the measure of the ease in constructing programs in a language

a. Dependent on the application area

b. Methods of comments

1) Some languages require a symbol at the beginning of the line

a) FORTRAN

2) Some languages allow line comments or paragraph comments

a) C++

c. Free format or fixed format lines

d. Conform to standard mathematical conventions

1) APL (has a bar with a dot above and below for division)

e. Keywords vs reserved words

1) You don't have to avoid any strings to write a program

f. Large set of built in operators

1) APL – contains an operator that multiplies two matricies together

3. Reliability – the characteristic that a program written in the language will perform to specification under all conditions

a. Type checking – determining if the correct types are presented to operations

1) Good example

a) Ada – a strongly typed languages

2) Bad example

a) C – a poorly typed language that allows you to get by with almost anything

b. Exception handling – an exception is a rare event which must be dealt with in a special fashion and is usually a runtime error

1) Ada is very good with exception handling

c. Lack of aliasing (allows multiple names to be associated with the same storage location)

1) Bad example

a) FORTRAN has aliasing built in

4. Cost

a. Cost to learn the language

1) Easy to learning

a) Pascal

b) BASIC

2) Not easy to learn

a) Ada – written by

b) PL1

b. Cost to write a program

1) You must have a compiler

2) The language has certain benefits to certain applications

c. Cost to compile a program

1) Some languages are designed to compile very quickly (quick and dirty). This removes certain features, such as overloading.

d. Cost to execute a program

1) We use code optimization to reduce the cost

2) We also may use optimizing compilers

3) FORTRAN can optimize, but Pascal cannot

e. Cost for lack of reliability

1) Can cost your reputation

2) Can cost production time

f. Cost of maintenance

5. Portability – ease of moving a program from one platform to another without major modification

a. Knowing standards is very important, because some compilers offer functionality that may not run on other compilers.

b. Pascal is very portable

6. Well-defined – good description of the syntax and semantics of the language

a. Good Examples

1) Ada

2) SNOBOL

b. It makes it easier to learn, write compilers for it, and it is likely to be more portable

Syntax – the description of the structure of sentences in a language.

Semantics – the description of the meaning of the sentences in a language.

7. Support of Abstration – the ability to create abstract data types

a. Examples

1) Ada

2) C++

3) Java

8. Naturalness for the application (“good fit” for the problem)

a. Examples

1) SNOBOL is good in string manipulation vs Pascal, which is bad

2) COBOL is used in a lot of traditional banking systems, so why change legacy code?

Language Categories

1. Imperative – a sequence of statements which are commands

a. They are not object oriented

b. They are not function

c. Examples

1) BASIC

2. Functional

a. These get their results though function, and then pass that result to another function, etc.

b. Examples:  
 1) LISP

2) Scheme

c. Especially handy with AI programs

3. Login-oriented

a. Especially handy with AI programs

b. Examples

1) PROLOG

4. Object-oriented

a. Examples

1) C++ – that is, unless you don't use objects, which makes it more like C, and therefore is essentially imperative (this is called **subsetting**)

2) Smalltalk – cannot be imperative, since everything is an object

Languages may also be categorized as general purpose or special purpose

Examples:

1. General purpose

a. C++

b. BASIC

2. Special purpose

a. Simula – a programming language designed specifically for similations

b. There is not as large of an audience for these languages since these are for specific languages.

Language Design Tradeoffs

1. Reliability vs cost of execution

a. Do you want it to be fast or to not crash?

b. Examples

1) Ada good for reliability

2) FORTRAN good for execution speed

2. Readability vs writeability

**CHAPTER 2**

Universal language – a language that can do anything, theoretically.

Although we already have languages that could handle anything, there are certain languages that are better for certain tasks.

History

1. Konrad Zuses' Plankalkul

a. Konrad worked on this programming language in 1945.

b. It was based on calculus

c. It had bits, integers, floats, arrays, records.

d. It existed, but nobody was aware of it; and there

e. Example:

1) | A + 1 => A

V | 4 5

S | 1.n 1.n

2) What we are used to: A[5] = A[4] +1

2. Pseudo-codes – not high level nor low level, but somewhere in between

a. Speed Code – John Backus

b. Short Code – John Mauchley (late 40's or early 50's)

c. These implemented high level-like concepts.

FORTRAN (FORmula TRANslation) | Fortran

a. Remained an acronym until 1990, when a new version came out that changed it so much that it no longer resembled the previous version. This more resembled C++.

b. The environment in which FORTRAN was developed.

1) Computers were slow and unreliable

2) Strong need for scientific calculations

3) Computers were extremely expensive and labor costs were low.

4) Most programming was in machine code or assembly language code.

5) John Backus was involved in the development.

6) Developed for IBM 701 (only one processor).

a) The problem was that the keyboard did not have brackets for arrays, and you could not use A(3) because it was used for functions. So, there had to be some means of interpreting the correct meaning for arrays.

b) The number of array dimensions had to be limited to 7, since the hardware would only allow 7 nested loops at maximum.

c) The hardware would only allow you to keep 6-character variable names

c. Three reasons for FORTRAN's success

1) Good for scientific calculations

a) Logarithms

b) Trigonometric

2) Good for creating optimizing compilers

3) Pushed by IBM

a) They gave out FORTRAN “free” compilers when the mainframe was bought

b) They also gave compilers to academics to market the language in the future

d. FORTRAN facts

1) Designed to use keywords, not reserved words.

a) They wanted to add to the language, but allow old programs to work the same way.

2) Comment lines are formed by \* or C in column 1

3) Fixed format → semifixed → free

a) Fixed

i. Col 1-5 – label

ii. Col 6 – continuation field (continuation of previous line)

iii. Col 7 – statement field

b) Semifixed – allowed you to put the statement at or after column 7

c) Free – removed columns

4) Variable names limited to 6 characters

5) Used implicit typing – if a variable began with I-N, it was an integer. Otherwise, it was a real. That means that the original FORTRAN only had integers and reals.

e. Evolution of FORTRAN

1) FORTRAN 0 (1954) – was not actually released

a) DO loop – post-test loop

b) Variables were 2 characters

c) Implicit typing (I for integer, everything else was real)

2) FORTRAN I (1955-1957)

a) 18 worker years (number of workers for a unit of time)

b) Variables were 6 characters

c) Implicit typing (I for integer, everything else was real)

3) FORTRAN II (1958)

a) Independent compilation of subprograms was integrated

b) Implicit typing (I for integer, everything else was real)

4) FORTRAN IV (1960-62)

a) Type declaration (explicit typing) was added

b) Logical IF was added

5) WATFIV – the fifth version of FORTRAN that was developed by the University of Waterloo, but never officially released by IBM

6) FORTRAN 77 (1978)

a) Added character string functionality

b) Added IF/ELSE structure

c) Added other control statements like WHILE loops

d) Added complex number data type

7) Fortran 90 (1990) – big change

a) Allowed recursion

b) Added a pointer data type

c) Added a CASE statements

d) Added dynamically allocated arrays

e) Removed the fixed format

8) Fortran 95 (1997)

a) Added a FORALL feature

I. for all elements in an array

II. great for parallelization

III. This made it great for processing images (represented in a 2D array)

9) Fortran 2003 (2004)

a) Added derived types

I. This could see and communicate with the C programming language

II. Added OOP structures

b) Added more parallelizing

c) Added pointers to subprograms

10) Fortran 2008 (2010)

a) Added a concurrency construct and more parallelizing

b) Added local blocks

11) There are other dialects of FORTRAN that existed, such as WATFIV

FORTRAN was geared more toward scientific calculations, but there were many people attempting to use it for simple business calculations. For this reason, COBOL was developed.

COBOL – Common Business Oriented Language

1. Developed in 1959 by the Department of Defense (DOD) under the leadership of Admiral Grace Hopper.

a. “Mathematical programs should be written in mathematical notation, data processing program should be written in English statements” - Admiral Grace Hopper

2. Design Goals

a. Use English as much as possible

b. Make it easy to use, even at the expense of power/efficiency/execution

c. The design should not be constrained by the difficulty in writing the compiler

d. Do it quickly

3. They planned on starting with a Short Range Committee and then handing it off to a Long Range Committee

a. The Short Range Committee produced a preliminary report with these decisions:

1) Separate the data description from the executable statements

2) Use long names, word connectors, and noise words for readability

3) Records – essential for many business applications

4) Formatting

a) Output was easily formatted for the business applications

b. The Long Range Committee was supposed to analyze and improve over a long period of time

4. There are no high level concepts, and the language is fairly strict

ALGOL – Algorithmic Language

1. In the late 1950's, there was an international team led by Peter Naur.

2. They developed IAL (International Algorithmic Language), but later renamed it as ALGOL

3. Release years

a. 1958

b. 1960

c. 1968

4. Design goals

a. Close to mathematical standards

b. Be useful for expressing algorithms

c. Should be able to be compiled

d. Should not be constrained by any one architecture (machine independent)

1) No input and output statements were allowed in the language specification since I/O is the feature that is most dependent on the architecture.

2) This meant that people had to design their own I/O according to their own system.

5. Contributions of ALGOL to the programming language landscape

a. Formal concept of a data type

b. Introduced the compound statement

c. Unlimited identifier length

d. Unlimited number of array dimensions

e. Allowed lower bounds of array subscripts

f. Recursion

g. Different methods of parameter passing

1) By value

2) By name

h. Assignment operator was :=

I. Formal method developed for describing the syntax

1) Bachus-Naur format was a formal description of the syntax of the language developed by Bachus and Peter Naur

6. ALGOL had several advantages which should have led to success:

a. International group

b. Machine independence

c. Clear description of syntax

7. Unfortunately, some of the design goals led to less than desirable features (characteristics):

a. Needed to be too flexible

b. Lack of built in I/O

c. No support by a government or a large company

8. There was a dialect of IAL called JOVIAL (Jule's Own Version of the International Algorithmic Language) that the government actually used in some of their applications

a. The dialect would run under the original language compiler, but not vice versa.

BASIC – Beginner's All-Purpose Symbolic Instruction Code

1. Came about in 1964 through John Kemeny and Thomas Kurtz at Dartmouth University

2. They wanted to encourage students to do their homework by adding terminals in each dorm room that were connected to the mainframe.

2. Design goals

a. Easy to learn (small set of instructions)

b. Pleasant/User friendly

c. Fast turnaround time

d. Provide private access (to keep everyone's projects separated from other individuals)

e. Consider the user's time more important that computer time

4. Features

a. First language widely used on remote access terminals

b. It was easy to implement (design the translator)

1) It ended up being an interpreted languages

c. It was easy to learn due to the small size of the language. It was not designed to do a large number of things.

d. Since it was so small, BASIC was shipped out with the PC's in their early days.

APL – Array Processing Language/A Programming Language

1. Originally came from a book written by Ken Iverson, describing a method for expressing algorithms as arrays; it was not intended to be a programming language.

2. Characteristics

a. There are no English words

b. Use standard mathematical symbols when possible

c. Needs a different keyboards. Removed the lower case and used the shifted alphabetic keys for more symbols

d. Interpreted

e. Large set of built in functions

f. Assignment operator A ← A + 1

g. Right associative (works right to left)

h. No precedence of operators

i. Certain symbols must be created by striking one key, backspace, and strike another key (called an “overstrike symbol”).

1)

j. Types are implicitly created when values are assigned.

3. Examples:

a. LIST ← 17 20 13 5

1) Stores a list of integers (separated by spaces)

b. p (Row) LIST: outputs 4 for the number of items in the list

c. SUM ← +/ LIST: outputs 55 and returns 55 for use with some other result

d. [] (box) is used for input

1) LIST ← []

e. Using the ceiling operator followed by | finds the largest value (vice versa for floor operator)

f. Index generator is over the letter I (looks like a vertical tilde)

1) | 5 : outputs 1 2 3 4 5

g. ? (roll generator)

1) ? 6 : outputs a random number between 1-6

h. LIST ← 3 5 11

X ← 5 2 11

Y ← LIST + X : outputs 8 7 12

i. Character strings are in single quotes

j. The “take” function is the up-pointing arrow

k. NAME ← 'JONES'

3 (up arrow) NAME : outputs JON

l. Relational operators return 0/1 for False/True

1) LIST ← 17 20 13 5

LIST > 15 : outputs 1 1 0 0

m. To create a 2D array

1) TABLE ← 2 3p(row) 4 8 5 6 0 1 : outputs an array of [4 8 5]

[6 0 1]

n. Array slicing – extracting a contiguous subset of an array

1) TABLE [;3] : outputs the 3rd column

2) TABLE [2;] : outputs the 2nd row

4. Monadic vs Dyadic

a. Monadic – a function that requires only one operand

b. Dyadic – a function that requires two operands

5. To define a function (function/definition mode):

a. We start a line with the del operator (upside down triangle) followed by the name of the function (this is called a **header line**).

1) No parameters (niladic) (void) – (upside-down triangle) name

2) One parameter (monadic) – (upside-down triangle) name parm1

3) Two parameters (dyadic) - (upside-down triangle) parm2 name parm1

b. We know that we are in definition mode, the interpreter displays each line number until another del operator is entered.

c. Return values may be specified by something like: del AVG ← name parm1

6. All work done can be saved in a “workspace,” and then can be later recalled. However, all variables associated with certain values in memory will be remembered.

7. tryapl.org allows us to find multiple resources for and try APL.

Environments That Influenced Languages

1. Batch environment

a. Earliest operating environment

b. Errors were costly

c. FORTRAN and COBOL had been developed during this time

d. Long turnaround time – the time it takes from submitting your program to the results

e. Operating on external files of data

f. Lack of timing constraints since multiple people are not running at the same time

2. Interactive environment

a. Programmer interacts directly with display/keyboard devices.

b. Shorter turnaround time

c. Errors may be corrected and handled quickly

d. Different types of I/O (interactive vs files)

e. Need for timing constraints

f. Examples

1) The language, C

3. Single user (PC)

a. Performance is less of an issue

b. No time sharing of a processor

c. Ease of use is important

d. Examples:

1) BASIC

2) Java

4. Embedded environment

a. Reliability is an issue

b. Timing constraints

c. Concurrency may be an issue

d. Exception handling

e. Strong type checking

f. Examples:

1) Ada

5. Distributed environment

a. Timing constraints

b. File sharing

c. Memory management

Other Influences on Language Development

1. Standardization

a. Proprietary – Done by the owner of the language

b. Consensus – Agreement by users

c. When to standardize?

1) 3 philosophies:

a) Early – at first release (Ada)

b) Release for input from user groups before setting a standard

I. Pascal

II. C

c) Late – after years of use when realizing dialects

I. FORTRAN

II. LISP

2. Computer architecture – The architecture available at the present time

3. Programming methodologies

a. Imperative

b. Object oriented

c. Functional

d. Scientific

4. Methods of language implementation

a. Language Implementation – design of the language translation

1) Compilation

a) Definition - the process of translating a source program in a high level language into an equivalent object program in a target language (could be machine code, assembly, or another high level language).

b) Terms to know:

I. Token – the smallest recognizable units of a language (constructed from the language alphabet (includes special symbols) )

c) May be divided into two phases:

I. Analysis – language dependent, machine independent (front end of compiler)

i) Lexical analysis

a. Performed by the scanner (scans across the lines of the language)

b. Input is the source code in the high level language

c. Action – Recognize the tokens or **lexenes** of the program and convert to token numbers.

d. Output – Stream of token numbers

ii) Syntactic analysis

a. Performed by the parser

b. Input – Stream of token numbers from the scanner

c. Action – Recognize valid syntactic structures which can be passed to the next step.

d. Output – recognize syntactic structures

e. Note: An error in syntax is detected here, but either:

i. An error is generated and compilation ends.

ii. An error is generated and error recovery is implemented to attempt to find other errors.

iii) Semantic Analysis

a. Performed by the semantic analyzer

b. Input – recognized syntactic structures from the parser

c. Action – Translate the structures to their meaning using intermediate code (triples, quads, or indirect triples).

i. Quads are better for optimization but use more memory.

ii. Triples use less memory but are not as good for optimization.

Iii. Indirect triples are the best of both worlds.

d. Output – intermediate code

II. Synthesis – language independent, machine dependent (back end of compiler)

i) Intermediate code optimization (optional)

a. Input – intermediate code from the semantic analyzer

b. Action – recognize changes to the intermediate code to either:

a. Reduce the number of quads

b. Reduce the number of temporaries

c. Output – optimized intermediate code

ii) Code Generation

a. Input – intermediate code

b. Action – translate the intermediate code to target language.

c. Output – object program in the target language.

iii) Optional Object Code Optimization

a. Input – generated code

b. Action – Make use of machine features to optimize the object code.

c. Output – optimized object program

III. Source program → Scanner –(token stream)→ Parser –(recognized structures)→ Semantic analyzer –(intermediate code)→ Intermediate code optimizer –(intermediate code)→ Code generation –(object code)→ Object code optimizer → Object Program in target language

2) Pure interpretation

a) Interpretation – translation method where a statement of the source (line) program is translated and executed directly and no object program is created.

b) Source → Interpreter → Results

3) Hybrid of compilation and interpretation

a) Perform the analysis phase as a compiler does.

b) Interpret the intermediate code as an interpreter.

b. Language features – often determine the best way to implement the language (compiler or interpreter).

1) \*Binding Time – The most important language features

a) Binding – the act of selecting a value form a set of possible values for a language attribute.

b) Binding time – the time of a programming language/program when an attribute is bound.

I. Five possible binding times in the life of a programming language

i) Language creation/design/definition

a. Example: + means integer addition

ii) Language implementation (creating the translator)

a. Using a full word to represent an integer

iii) Language translation

a. Programmer specifies INTEGER x (type to variable)

b. Translator (compiler) chooses

1. Example: Relative position of the object (constant, variable, literal) ) within storage for the program.

iv) Load time – when the translated object program is brought into memory by a loader

a. Example: Actual memory address is bound to a variable

v) Execution time

a. Example: When a value is bound to a variable

II. When do you bind?

i) Early binding

a. Most binding occurs before or during translation

b. These languages are generally compiled

c. Execution efficient

ii) Late binding

a. Most binding is delayed to execution

b. These languages are generally interpreted

c. Flexible, but not execution efficient

Characteristics of a Programming Language

1. Syntax – description of the structure of sentences in a language

a. Issues

1) Readability

a) Characteristics

I. Self documenting – understandable without further explanation

II. Understandable

b) Enhanced by:

I. Noise words (COBOL)

II. Reserved words

III. Unrestricted identifier length

IV Free format (whitespace indentions)

V. Data type declaration

VI. Embedded (inline) comments

VII. Compound/block statements

2) Writability

a) Enhanced by:

I. Keywords

II. Implicit typing instead of required data type declaration

III. Simple structures

IV. Small set of language features

3) Character Set (ASCII or EBCDIC?)

4) Identifier naming rules (identifier is a string used to name a program element)

a) Program elements which are named include:

I. Variables

II. Constants

III. Subprograms

IV. Exceptions

V. Coroutines

VI. Tasks

VII. Modules, classes, objects, units, packages

VIII. Labels, program

b) Rules

I. What must it begin with? (a-z, special characters, etc)

II. What can they contain?

III. Is it case sensitive?

IV. What is its length?

i) Fixed length max

ii) Limited but only recognize difference in the first *n* characters

iii) Unlimited

5) Operator Symbols

a) + or PLUS or ADD

6) Keywords vs Reserved Words

a) Keywords

I. Writability

II. Upward compatibility

b) Reserved Words

I. Readability

II. Ease of translation

c) Pascal has both

7) Method of commenting

a) Special comment line (line begins with a certain character) (easiest for compiler)

b) Beginning and ending symbol for a comment paragraph (hardest for compiler)

c) Beginning symbol to end of line (next easiest for compiler)

8) Use of blanks

a) Three different approaches

I. Ignore blanks

II. Use as separators

III. Use as an operator

9) Delimiters (grouping symbols)

10) Format of statements

a) Three approaches

I. Fixed format (specified columns)

II. Semi-fixed format

III. Free format (most modern programming languages)

11) Assignment statement

a) Symbol used

b) Does it return a value

12) Expressions – how are they formed (infix, prefix, postfix)

13) Statements

a) Single format or multiple format

b) Simple or nested

14) Program structure

a) Separate subprogram definition, separately compiled and linked at load time (FORTRAN)

b) Separate data definition section (COBOL)

c) Nested subprogram definition (Pascal, Ada (came from ALGOL) ) creates a nonlocal reference environment, so other functions cannot access nested functions.

d) Separate interface definition (Ada)

e) Separate creation/definition of a data type (Java)

f) No subprograms (no separation of statements) (SNOBOL)

15) Case sensitivity

2. Semantics – description of the meaning of the structures of a language

Programming Language Theory

1. Language – set of strings that are formed from some alphabet.

2. Lexeme – smallest recognizable unit of a language.

3. A language may be defined by:

a. Language generator – grammar

b. Language recognizer – automaton

\*4. Chomsky Hierarchy – method of categorizing the complexity of languages

a. Definitions

1) Let A,B be an element of nonterminals N

a,b be an element of terminals (tokens) T

o, T, alpha, beta be an element of (N(onterminals) AND T(okens/erminals))\* (o, T, alpha, beta are strings of tokens and nonterminals)

2) → – “may be replaced by”

3) | – “or”

|  |  |  |  |
| --- | --- | --- | --- |
| Language | Recognizer (Automaton) | Generator (grammar) | Grammar Rules (form) |
| Regular Language | Finite state automaton | Regular grammar | A → a  A → bB  (start symbol may be replaced with empty string) |
| Context Free Language | Pushdown Automaton | Context Free Grammar | A → alpha |
| Context Sensitive Language | Linear Bounded Automaton | Context Sensitive Grammar | o alpha T → o beta T  length(alpha) <= length(beta) |
| Recursively Enumerable Language | Turing Machine | Unrestricted grammar | alpha → beta  alpha != empty string ( {} e ) |

b. Regular

1) A finite state automaton is able to handle problems such as coin dispensers, but are not powerful enough to handle programming languages.

2) Good for a scanner

c. Context free

1) The meaning of a word is not dependent upon its context. This would be favorable for a programming language.

2) Good for a parser

d. Complexity and power increase as we move down the table.

e. Example

1) Grammar = 4 tuple = (N,T,P,S)

where

N = finite set of nonterminals

T = finite set of terminals (tokens)

P = finite set of production rules

S = start symbol S is an element of N

P = { A → aB N = {A,B,C}

{ B → bC T = {a,b,c}

{ B → c

{ C → c S = A

a) We start with A because it is not represented anywhere on the right, which would make it a wasted symbol if not used.

b) We begin at the start symbol, replacing the resulting set of terminals/nonterminals with the next set of terminals/nonterminals according to the rules.

f. Derivation – process formed by beginning with a start symbol replacing one rule at a time until only tokens/terminals remain.

1) Leftmost derivation – replacing the leftmost nonterminal.

g. Sentential form – one step in a derivation.

h. Sentence – a sentential form with only tokens.

5. Parse Tree – graphical representation of a derivation (root is the start symbol, leaves are the terminals)

6. Metalanguage – language used to describe another language

a. Backus-Naur Form (BNF) (it used to be “Normal” before Peter Naur) – a metalanguage useful for describing context-free languages.

1) ::= – “is defined as” or “may be replaced by”

a) The symbol later became →

2) | – “or” used to separate alternatives

3) {…} – “zero or more times”

4) [...] – “optional”

5) <...> – encloses nonterminals

6) Examples:

a) BNF for a Pascal assignment statement

I. <assignment statement> ::= <identifier> := <expression>

b) BNF for Pascal identifiers

I. <identifier> ::= A | B | C | D | E | F| … Z {A | B | … Z | 0 | 1 | … 9 | }07

c) <expression> :: = <term> + <expression | <term> - <expression> | <term>

<term> ::= <factor> \* <term> | <factor> / <term> | <factor>

<factor> ::= <identifier> | <integer> | (<expression>)

I. An example of a recursive expression

II. <term> + <expression> – this is known as **right recursion**, since the expression is on the right which is easier for the parser.

III. The first rule shows that addition and subtraction have the same priority.

IV. The second rule shows that multiplication and division have the same priority, but has a higher priority than addition and subtraction.

V. The third rule shows that identifiers, integers, and expressions within parentheses have higher priority than the previous two rules.

VI. The three rules result in **indirect recursion** if the expression is used for the factor.

VII. Each rule is dependent on the one below it due to the definitions.

d) Assuming the following, is A := B \* C + D a valid assignemtn statement?

<assignment statement> :: = <identifier> := <expression>

<expression> :: = <term> + <expression | <term> - <expression> | <term>

<term> ::= <factor> \* <term> | <factor> / <term> | <factor>

<factor> ::= <identifier> | <integer> | (<expression>)

Grammar N = { assignment statement, expression, term, factor, identifier, integer}

T = { := , +, -, \*, /, (, ), A..Z, 0..0, \_ }

P =

S = Assignment statement

We can construct a tree starting at <assignment statement>

/ | \

<identifier> := <expression>

/ / | \

A <term> + <expression>

/ | \ \

<factor> \* <term> <term>

/ \ \

<identifier> <factor> <factor>

/ \ \

B <identifier> <identifier>

\ \

C D

OR we can do it this way (don't apply two rules in one step):

<assignment statement>

<identifier> : = <expression>

A := <expression>

A := <term> + <expression>

A := <factor> + <term> + <expression>

A := <identifier> \* <term> + <expression>

A := B \* <term> + <expression>

A := B \* <factor> + <expression>

A := B \* <identifier> + <expression>

A := B \* C + <expression>

A := B \* C + <term>

A := B \* C + <factor>

A := B \* C + <identifier>

A := B \* C + D **Done here!**

**In both cases, the answer is YES!**

7) Ambiguous grammar – a grammar in which one sentence has two or more distinct parse trees.

a) Example:

I. <expression> ::= <expression> + <expression> |

<expression> - <expression> |

<expression> \* <expression> |

<expression> / <expression>

i) We can create two differently ordered sets of expressions

a. <id> ::= A | B | C

1. A \* B + C

8) BNF does not describe the semantics, but only the syntax.

b. Attribute grammar – extension of BNF used to convey syntax as well as static semantics.

1) Static semantics – the semantic information which can be determined at compile time.

2) Dynamic semantics – semantic information which cannot be determined until runtime.

3) Additions to BNF to create an attribute grammar (pg 137)

a) For each nonterminal X we add a set of attributes A(X) = S(X) + I(X) where S(X) is a set of synthesized attributes (passed up the parse tree from the child to the parent) and I(X) is a set of inherited attributes (passed down the parse tree from parent to child).

b) Associated with each grammar rule (used for natural language processors):

I. Semantic functions

II. Predicate functions

7. Semantics – description of the meaning of the structures of a programming language.

a. Methods for describing semantics of a programming language.

1) Operational models – define how a program operates on a virtual machine.

a) Most well known is VDL (Vienna Definition Language)

b) Pros:

I. This is good for **user** (programmer) or **implementer** (designer of translator)

2) Applicative models (most widely used) – denotational semantics

a) Uses recursive function theory (define functions to explain language semantics.

b) Pros:

I. Good for language designers

c) Cons:

I. Gets large

II. Not user friendly

3) Axiomatic models – defining semantics in terms of axioms (rules)

a) Pros:

I. Great for proving correctness

b) Cons:

I. Sometimes is too mathematical, and people shy away from it

II. Not good for users or implementers

c) Example: Axiomatic Semantics

Name Entities in a Programming Language

1. Variables – a storage location whose value may be changed.

2. Constants – Storage location whose value is fixed during the life of the program.

3. Subprograms – series of actions associated with a name.

4. Label – name used to reference a location in a program.

5. Data types

6. Data object – container (in memory) to hold a value

a. Categories

1) Variable

2) Constants

3) Literal – storage location whose name is its value representation

b. Two important concepts associated with data objects:

1) Scope – that part of a program where an entity is known and can be used

\*a) Two approaches to evaluating scope:

\*I. Static scoping – associate, with the reference to an identifier, the definition of the identifier in the closest textually containing block of code.

i) Facts:

a. This may be determined at compile time.

b. This requires declarations instead of implicit typing.

c. This allows the execution to be more efficient.

ii) Two methods for implementing:

\*a. Static pointer

1. Replace the reference to an identifier with an ordered pair (I,J) where I = # of pointer traversals and J = offset (bytes) into activation record where the identifier is defined.

2. Each activation record contains a pointer to the activation record of a block which contains it.

3. Variables may be local to a block in which it is declared, and global if defined in parent blocks.

4. When initially calling the main program, each block that is called is added on top of the previously called block (a stack) and the variables used but not defined in the called block that have been defined in the parent block will point to the location of the variable in the parent block.

5. More efficient with many calls/returns and few references.

\*b. Display method

1. Replace the reference to an identifier with an ordered par (I,J) where I = index within a display stack of pointers to activation records.

2. This method uses 3 stacks:

a) Display stack

b) Hidden stack

c) Stack of activation records

3. Faster when you have many pointers/references and few calls

\*II. Dynamic scoping – associate, with the reference to an identifier, the definition of the identifier in the most recently invoked, not yet terminated, block.

i) Facts:

a. This cannot be determined until run time.

b. This allows for implicit typing.

c. This requires overhead during runtime, hence a slower running program.

ii) Two methods for implementing:

a. Look back through the calling chain

1. If there is not a local variable, it looks to the calling function for the variable.

b. CRET – current referencing environment table

1. Uses a table with 3 fields:

a) Identifier

b) Active flag (true or false)

c) Pointer to the activation record where defined (null if not active)

2. Each pointer is called the **current referencing environment pointer**

3. When pointing to a local variable, if the variable exists in the calling function, we push that pointer onto the hidden stack so that our reference points to the local variable.

4. We may regain our pointers to the calling function by popping the pointers off of the hidden stack, and pointing to the popped pointer.

5. We use a special marker to separate the pointers on the hidden stack every time that a function call is added to the active stack.

4. Hole in the scope – when the pointer to a global variable is hidden from the local scope.

6. **Overhead of calls and returns** – management of resources between subroutine calls and returns.

2) Extent on Lifetime – that part of a program where a data object actually has storage associated with it (time from allocation to deallocation).

Binding

1. Recall:

a. Binding – the act of “fixing” or “setting” the value of an attribute associated with a programming language entity from a set of possible values.

2. Two categories of binding

a. Static (early) binding -determined at compile time

1) Allows for type checking

2) Very efficient

3) Usually done in compiled languages

b. Dynamic binding – determined at runtime

1) Usually interpreted languages

2) Not as efficient

Extent (Lifetime)

1. Time between allocation and deallocation of memory for a data object (container for value).

2. Categories

a. Static

1) Storage is allocated when the program begins and deallocated when the program

terminates.

2) For global variables or any others which need to be history sensitive.

3) This is very efficient in execution speed, but reduces flexibility as storage may not be used and may not be efficient with storage space.

4) Making everything static will remove the ability to create recursive programs.

b. Stack-Dynamic-variables

1) Storage is bound/allocated when the declaration of the variable is elaborated (when the code where the declaration appears is executed, the storage is bound) (Java)

2) Managed from the runtime stack

3) Explicit Heap-Dynamic variables

a) Storage managed from a runtime heap where the program explicitly requests allocation/deallocation of memory.

b) The language has two special functions:

I. NEW/DISPOSE

II. NEW/FREE

III. ALLOC/FREE

IV. NEW/DELETE

4) Implicit Heap Dynamic Variables

a) Storage is bound to a variable when a value is assigned to it.

b) The most flexible, but error detection is harder and overhead is greater

Referencing Environment

1. Definition – all the identifiers which can be used within a currently executing block (subprogram).

2. 3 Categories

a. Global – an identifier defined in the main program

1) If an identifier is used in the main program, then it is considered global, not local.

b. Local – identifiers defined within the current executing block that's local.

c. Nonlocal – identifier which can be used but is not defined locally (in the current block) and is not defined globally. This is the case if the block above a scope contains an identifier.

3. We first look for an identifier locally, then up a block (non-locally) until we then reach and check globally.

4. Two problems related to scope/extent:

a. Assume PTRA and PTRB are pointers to the same data type:

1) If we did:

New(PTRA);

PTRB := PTRA;

DISPOSE(PTRA);

a) Then the scope exceeds EXTENT (**dangling reference**). This means that the memory is considered to be unallocated memory in the heap, but PTRB still points to that address.

I. If the data at that address changed, then PTRB would return that data.

2) If we did:

New(PTRA);

PTRA := NIL;

a) Then the EXTENT exceeds SCOPE (**inaccessible storage**). This means that the memory is removed from the **memory available heap**, but it is not referenced.

END CHAPTER 5

Homework, Due Tuesday, 10/7

pg. 163

6, 7d, 8, 10, 11

Midterm – Tuesday, October 21 Definitions

Give example of Language

Many discussions

Parse Tree

Draw or describe the grammar (prove/show if ambiguous)

Static Scoping

CRET

**CHAPTER 6**

Data Types

1. Definition – set of possible values which can be stored in a variable of that type.

2. It is important to know the maximum and minimum integers that can be stored.

3. We may consider data types on 3 levels:

a. Specification

1) Attributes – long int, short int, etc.

2) Values it can take on

3) Operations on that data type (postfix/prefix addition, etc.)

b. Implementation

1) Storage for that type (halfword, fullword, etc.)

2) Algorithms for each operation

c. Syntactic representation

1) Declarations

a) char, int, string, etc.

2) Value representation

a) 1.25 or .125e1

3) Operator symbols

a) +, -, \*, etc.

4. Classification by complexity

a. Elementary data types – data container for this type holds one single value (single character, integer, etc.)

1) Examples:

a) Integers

b) Reals

c) Character

d) Boolean

e) Enumerations

2) Specification

a) Attribute – possibly use a descriptor to specify the type

b) Values – range of values (max-min)

c) Operations – express as a mathematical function

I. Operation name : argument type 1 X argument type 2 → result type

a. Examples:

1. + : integer X integer → integer

2. + : integer X real → real

3. + : real X integer → real

II. **Arity of an operation** – the number of input operands needed for the operation.

a. Binary or dyadic – needs 2 operands

b. Unary or monadic – needs only one operand

III. How operations are to work is sometimes obscured by:

a. Operation may be undefined for certain inputs

1. Two integers that are two large

2. Dividing by 0

b. Operation may have implicit arguments

c. Operation may have side effects

d. Operation may be history sensitive

3) Implementation

a) Storage (how it is represented in hardware)

I. At compile time – build a symbol table and determine displacement

II. At runtime – need a runtime descriptor

i) Create an identifier that points to storage, where the type and the value is stored.

a. Takes less memory, but more time (more execution efficient.

ii) Create an identifier that holds the type and points to storage, where the value is stored.

a. Takes more memory, but less time (more execution efficient)

b) Operations

I. Directly on hardware, using hardware primitives

II. Write a software procedure to implement.

III. Insert extra in-line code for small operations.

b. Structured or Composite type – data container holds multiple values.

1) Examples:

a) Arrays

b) Vectors

c) Matrices

d) Records

e) Structs

c. Hybrid

1) Examples:

a) Pointers

5. Declaration

a. A statement which “fixes” a data type to a variable. This communicates what the data object or container will hold.

b. Type of declaration

1) Explicit – special statement in the language

a) Example – X: INTEGER (Pascal)

2) Implicit

a) Default by spelling

I. Examples

i) FORTRAN – any variable beginning with I-N is integer. All others are reals.

b) By the assigning of a value to a variable

I. Examples

i) A ← 17 (APL)

c. Purposes of having explicit declarations or implicit by spelling

1) Help the compiler determine storage management

2) Give the lifetime (extent) of the variable (when to allocate/deallocate)

3) Resolve overloaded operators

a) This is extremely important for a large number of operators, such as + with ints, floats, doubles, etc., and a mixture of them.

4) Allows for static type checking

d. Type checking – determining if the operands for an operation are compatible with the operation.

1) Categories

a) Dynamic type checking – performed at runtime

I. Pros

I) Flexible

II. Cons

i) Slows execution

ii) Requires a type tag at runtime (extra storage)

iii) Forces operation to be implemented through software routines

III. Examples – APL, LISP, SNOBOL

b) Static type checking – performed at compile time, making use of a symbol table

I. Pros

i) Makes program more efficient

ii) All paths are checked at translation)

iii) No type tags are required

II. Cons

i) Inflexible

e. Strongly typed language

1) A language in which mismatched types are statically detected at compile time. This feature contributes to reliability.

2) If a mismatch occurs:

a) In a strongly typed language, issue an error message.

b) In other languages, an attempt to change one type to match another may be made.

f. Coercion – the changing of mismatched types (type conversion)

1) Types

a) Implicit – the language makes the decision

b) Explicit – programmer specifies how to coerce

2) The decision as to which type to coerce to may be classified as:

a) Widening – changing from a subset to a superset

b) Narrowing – changing from a superset to a subset

3) Coercion Philosophies

a) No implicit coercion

b) Coerce only in cases of widening

c) Coerce as much as possible, even when it may not make sense (SNOBOL)

6. Scalar or elementary data types (data objects which contain only one value)

a. Numeric

1) Integer

a) Operations

I. Add

II. Subtract

III. Multiply

IV. Divide

V. Mod

VI. Relational

i) integer X integer → integers

ii) integer X integer → boolean

b) Implementation of storage:

I. No runtime descriptor (static typing languages that require declaration)

I) Can make use of hardware primitives (most efficient)

II. Separate runtime descriptor

i) Dynamic typing languages

ii) Takes extra storage, but can still use the hardware primitives

III. Runtime descriptor sharing storage with the value

i) Saves storage (from II.) but cannot use hardware primitives without software support

2) Subrange (of integers)

a) Finite, sequential subset of the integers (1..10)

b) Advantages over just integer type:

I. Makes for better storage management

II. Aids in better type checking

3) Floating point (real)

a) We are concerned with:

I. Expression of the value (173.59, or 1.7359E2)

II. Range of value

III. Operators

i) +, -, \*, /

ii) Problem with relational operator and real numbers. Binary floating point may be directly represented in decimal floating point format, but not vice versa.

a. Example: .110 = .00011001100110011...2

1. This error grows even more if we attempt to add .1 to .1

IV. Implementation

4) Fixed point type

a) A real number with a fixed number of decimal places

b) May be implemented using hardware primitives (packed) and software simulation or entirely software simulated.

5) Complex number

a) Example: a + bi where i = sqrt(-1)

b) Generally implemented as 2 consecutive real number locations (FORTRAN)

c) Operations

I. +, -, \*, /, conjugate, absolute value

6) Rational number

a) Represented as 2 consecutive integers for numerator and denominator

b. Enumerations

1) Finite, ordered list of distinct values

2) Example: colors = (red, blue, green)

3) Generally implemented as a correspondence to integers

4) Operations

a) Relational, predecessor pred() (previous value), successor succ() (next value)

5) Not considered numeric because it is not used in arithmetic

c. Boolean or logical

1) Values: True or False

2) May be implemented differently internally

a) 1 = true, 0 = false

b) One bit

c) One byte

d) One word

e) Nonzero = true, zero = false

3) Representation in syntax

a) True = .TRUE. , False = .FALSE. (FORTRAN)

b) True = t , False = nil (LISP)

4) Operations

a) AND (FORTRAN = .AND.)

b) OR (FORTRAN = .OR.)

c) NOT (FORTRAN = .NOT.)

d) NAND

e) NOR

f) XOR

g) EQU

h) NEQ

d. Character

1) Represented using the hardware character type (one byte per character)

2) Generally in ASCII or EBCDIC

3) Relational (depends on the coding sequence for character)

4) Collating sequence – ordering of characters according to their numeric representation (different between codes such as ASCII and EBCDIC)

5) Some languages have:

a) ORD (ordinal), which returns the numeric representation of the character.

I. ORD('A')

b) CHR, which returns the character representation of the number

I. CHR(65) = A

II. CHR(193)

7. Composite or structured data types

1) Character strings

a) Sequence of characters (FORTRAN 77)

b) Three approaches (pg 185 ?)

I. Fixed declared length

i) There must always be exactly that amount of characters stored, not up to a number.

ii) Great for compiling without having to manage memory

II. Max declared length

i) These have a maximum number of characters, but you may vary the amount of stored characters less than the maximum

ii) Great for compiling without having to manage memory as long as we keep up with the length.

III. No declared length

i) Everything must be managed at runtime

c) Operations

I. Concatenation

II. Relational

III. Substring selection

IV. Pattern matching (SNOBOL is great at this)

2) Vectors or one dimensional arrays (probably the most common structured type)

a) A homogeneous data structure which has components referenced by a subscript or index.

b) Issues:

I. Number of components

i) Explicitly stated

ii) Implied by assignment

II. Index range

i) Specified upper bound, lower bound implied ( a[5] )

ii) Specified upper bound and lower bound ( [3..8] )

iii) Implied upper bound, one less than the number of terms (C++)

III. Type of the index

I) Enumerated types

ii) Integers

iii) Characters

iv) Anything ordinal

IV. Operations

i) Subscripting?

a. A(3) instead of A[3] if there were no [ ] brackets

b. Using A<3> (SNOBOL)

ii) Entire structure operations of:

a. Asssignment?

1. Can we assign 6 values to an array in one statemtent?

b. Comparison?

1. Can we compare A < B where A = (2,4,7) and B = (2,4,5)?

c. Input?

1. Can we read with one READ function?

d. Output?

1. Can we output with one statement?

e. Arithmetic

V. Storage

I) Sequential for homogeneous, fixed declared size

ii) Linked for homogeneous, variable size

iii) Linked for heterogeneous

c) Implemented using a runtime descriptor

|  |
| --- |
| **VECTOR** |
| Name |
| Type of component |
| Upperbound of index |
| Lower bound of index |
| Type of index |
| Address of vector |

I. Every time an element is accessed in an array, the address is calculated: address A[i] = start address + (i – Lower Bound) \* element size

3) Multi-dimensional arrays (2 or more) may be viewed as a vector of vectors

a) Issues:

I. Storage (row-major or column-major)

i) Row-major – the first index becomes the outer loop

a. Caclulation of the address of A[i,j] given start address (*a*), element size (E), lower bound 1, upper bound 1, lower bound 2, upper bound 2

1. address A[i,j] = start address + [(# row to skip)(size of address) + (# in last row to skip)] \* element size = *a* + [(UB1 - i)(UB2 – LB2 + 1) + (UB2 – j)] \* E

b. In general for A[I1, i2, …, in] stored in row major:

1. address of A[I1, i2, …, in] = start address + [(# rows to skip)(# in each row) + (# in last row to skip)] \* element size = *a* + SUMi=1n[(ii – Lbi) \*

ii) Column-major – the last index becomes the outer loop

b) Slice – substructure of an array that is also an array

i) FORTRAN contained this and was an ancestor of PL1, but then PL1 implemented this and FORTRAN then followed PL1's implementation.

c) Associative array (table)

i) Uses a subscript, which is not a predefined ordinal type or enumeration

ii) Used in:

a. SNOBOL

1. T = TABLE() //declaring a table T

2. T[THURSDAY] = RAINY

T[WEDNESDAY] = WINDY

3. This table can stored in a 2D array where TUESDAY and WEDNESDAY are stored in column 1, and RAINY and WINDY are stored in column 2.

4. We can also sort by the columns.

b. PERL

4) Record – heterogeneous data structured

a) Issues

I. Number of components (fields)

II. Name for each field

III. Type for each field

IV. Method of selecting a field

i) Most languages use the . (dot) operator

V. Method of storage (sequential with offset saved for each field in a runtime descriptor)

i) Example:

TYPE

PART = RECORD

NUMBER: INTEGER;

NAME: PACKED ARRAY[1..20] OF CHAR;

QUANTITY: INTEGER;

PRICE: REAL

END;

VAR

X,Y: PART;

ii) If we want to access these fields, we will have to use an offset to access them. We store each descriptor name and their offset.

VI. Entire structure operations

i) Assignment X := Y

ii) Comparison

iii) Input

iv) Output

VII. Problems with having two records having the exact same structure, but different field names.

i) Some languages allow compatibility, and others do not

VIII. Problems with two record types with exact same field names and structure, but different type names

i) Some languages allow compatibility, and others do not

5) Variant record – a record which has a fixed portion of fields and a varying portion of fields dependent on the value stored within one field (**tag**  or **discriminant**).

a) Problem with attempting to access a field which does not exist at the current time according to the value stored in the tag or discriminant.

b) There is also a problem with requiring storage be reserved the largest size of the variant record.

6) List – ordered sequence of data objects

a) LISP is a classic example of list processing

b) Operations

I. First (Head) (CAR in LISP)

II. Last (Tail) (CDR in LISP)

c) How are lists different than vectors?

I. Vectors normally have fixed length; lists rarely do.

II. Vectors are normally homogeneous; lists rarely are or are not required to be.

III. Vectors are normally explicitly declared; lists rarely are.

IV. Vectors appear in both compiled and interpreted languages; lists are normally in interpreted languages.

7) Sets – unordered collection of objects (without repeats)

a) Operations

I. Membership

II. Set union

III. Set intersection

IV. Set difference

V. Set equivalence

VI. Assignment

8) Special Consideration of structured data types

a) Number of components

I. Fixed

i) Easier for a compiler to managed

ii) Determined at compile time for efficiency

II. Variable

i) Determined at run time

b) Type of components

I. Homogeneous – all the same type (array of Booleans, Structs, Ints, etc.)

II. Heterogeneous

c) Selection method (how to specify a component)

I. How to get variables from a struct (cat.meow)

II. How to get an index of an array ( cat[1] or cat(1) )

d) Organization of the components in memory

I. Is array data stored in row-major or column-major order in memory?

e) Entire structure operations or only component operations

I. Can we print an entire array without having to loop through?

II. Can we print an entire struct without accessing specific items?

f) Operations for inserting or deleting components

I. Can we insert into the middle of an array?

g) Operations for creating or destroying the entire structure

h) Required declarations

I. If so:

i) Type of structure

ii) Number of components (fields)

iii) Type of each field (if not homogeneous)

i.) Problems with structured types:

I. Does a component actually exist?

II. Does the type of the component match?

8. Abstract Data type

a. Set of data objects

b. Set of abstract operations

c. Method of encapsulating and hiding

d. Programs may be designed from two perspectives:

1) Functional decomposition (using imperative languages)

2) Data decomposition (using object oriented programming and abstract data types)

Arithmetic Expressions

1. Issues

a. Unary or binary or other (# of operands)

b. Infix, prefix, or postfix

c. Order of evaluation (for infix)

1) Left associative or right associative (derived term is on left or right)

2) Precedence of operators (+ and – tend to have the same precedence, while \* and / tend to have higher precedence than + and -)

3) Parentheses to change precedence

4) Order of evaluation of operands

a) Methods

I. Left to right

II. Right to left

III. Simple first then function calls

IV. Functions first then simple variables

b) Problems with **side effects** (changes to variables not passed as parameters)

I. For, A + fun(x) \* A, assume A is initially 1, fun(x) returns 3 and has a side effect of incrementing A by one.

I) Left to right 1 + 3 \* 2 = 7

ii) Right to left 2 + 3 \* 1 = 5

iii) Simple first then function calls 1 + 3 \* 1 = 4

iv) Functions first then simple variables 2 + 3 \* 2 = 8

c) Solutions

I. Disallow side effects

II. Only allow changes to parameters

III. Have a prescribed order from the language (do not leave up to the compiler designer)

5) Overloaded operators – operator symbol has multiple meanings

a) Must be resolved by the compiler

b) Problems

I. Lack of context-free implementation (is context-sensitive)

I) Context sensitivity is harder

II. Lack of readability

III. May make errors more difficult to find (also has to do with readability)

IV. Makes the compiler design more difficult

6) Type conversions for mismatched types (coercion)

a) Widening – going from subset to superset (int to float)

b) Narrowing – going from superset to subset (float to int)

c) If automatically performed by the language (implicit) then reliability is reduced.

I. If a float is implicitly casted to an int, then an important fractional part can be lost.

7) Relational expressions which return Boolean values and Boolean expressions (involving Boolean operators: AND, OR, NOT, etc) which also return Boolean values also have issues with precedence, associativity, and order of evaluation of operands as well.

2. Assignment statement – mechanism for changing the binding of a value to a variable

a. Examples:

1) A := B (Ada?)

2) A = B

3) MOVE B TO A (COBOL)

4) A ← B (APL)

5) (setq A B) (LISP)

b. Views of how an assignment statement works

1) Separate statement that does not return a value (cannot be embedded in some other statement) or doesn't have multiple destinations

2) Statement returns a value which can be embedded

a) Problems with error detection

b) Examples: Javascript

3) Multiple target or destination, multiple source

a) Examples: Ruby, Perl, Lua

I. ($x, $y, $z) = (1, 3, 5)

4) Conditional targets

a) Perl

I. ($flag ? $count1 : $count2) = 0;

5) As compound operators

a) Introduced by Algol 68

b) Supported by most C-based languages (Python, C, C++, Java, Javascript)

c) Example: +=, -=, \*=, etc.

6) Note: Assignment also has an issue with mismatched types

a) Solutions

I. No mismatched types accepted

I) Ada

II. Only allow widening

I) Java

2) C#

III. Change whenever possible

IV. For interpreted languages (like LISP), delay the decision until runtime

Chapter 8

Statement Level Control Statements – affect the sequence of execution of statements.

1. Control structure – control statement and the collection of the statements which it controls

2. Three categories of statement control level control statements

a. Composition – sequence of statements which are treated as one statement (such as the scope braces in C++)

1) Key issues

a) Syntax enclosing the composition

I. Indention

b) Where can it be used?

I. Is there a limit on levels of nesting?

b. Alternation (Selection) (such as IF) – selecting between two or more execution paths

1) IF statement issues

a) Do we use words or symbols?

I. SNOBOL and APL do not use words.

II. Most other languages use words.

b) Are the paths selected using Boolean expressions or arithmetic?

I. FORTRAN uses arithmetic(?)

i) IF (N + 5) 200, 300, 400

ii) This calculates N+5 and then goes to line 200 if the result is 0, 300 if the result is negative, and 400 if the result is positive.

\*c) Dangling Else problem

I. Example: IF (A < B) THEN IF (C > D) THEN N = N + 1 ELSE N = 0

I) We have to determine which IF the ELSE is paired with.

II. To solve this, we may:

I) Agree by convention (of the language) to resolve by associating the else to the nearest IF statement.

ii) Force the use of some composition notation

iii) Have an ending symbol for each structure

iv) Require all IF statements to have an ELSE clause

2) Multi-way selection (CASE, switch)

a) Form and type of expression which controls the selection

b) How are the different paths denoted?

I. Can we put multiple paths together?

c) Once one path has been selected and executed, is the structure automatically vacated?

d) How are the CASE values specified?

I. Is it a constant; can it be an expression?

e) Is there a default case?

f) How to implement?

I. Use conditional branching on the hardware level

II. Use a jump table

III. Use a hash table

IV. Use an array

c. Repetition (Iteration or loops)

1) Simple iteration

a) COBOL example

I. PERFORM

…

5 TIMES

2) Repetition controlled by a counter

a) FORTRAN contained a DO loop

b) Pascal contained a FOR loop

c) Issues

I. Type and scope of the control variable

II. Can the loop variable be changed in the body of the loop or only by the control statement?

III. Pretest or Posttest loop

IV. Do you evaluate the loop parameters once or on each iteration

3) Repetition controlled by a logical condition

a) Pretest – WHILE

b) Posttest – REPEAT/UNTIL or DO WHILE

4) Repetition based on a data structure

a) forevery or foreach

5) Indefinite iteration – not controlled by a counter, logical expression, or data structure

pg 383, #1, 3, 4

Due Tuesday, Nov 25

**CHAPTER 9**

Subprograms

1. Characteristics

a. Single entry – start at the same place every time

b. Caller is suspended during the execution of the call-ee

c. Return to the caller

1) Void returning function – return to the next instruction without returning a value.

a) Pascal (procedures) and Ada ( … subroutines)?

2) Value returning function – where the return goes back to the last call and returns the value.

2. Facts

a. The definition of a subprogram gives the interface and the actions

b. The call to a subprogram is a request for execution.

c. The header of a subprogram gives the name, the type of subprogram, the name and type of parameters, and possibly the return type.

d. Parameter profile – Gives the number, order, type and name of formal parameters.

e. Formal parameter – parameter listed in the header line.

f. Actual parameter – parameter which appears in the subprogram call.

g. Correspondence of formal to actual parameters:

1) Positional passing (first with first, second with second, etc.)

2) Keyword passing

a) For the following example:

Function Reader(var x: integer, y: real)

…

Reader(y = B, x = A)

b) Python and Ada

3) Combination of (1) and (2)

h. Two categories of subprograms

1) Non-value returning subprograms

a) Examples of naming conventions

I. Procedure (Pascal)

II. Subroutine (Ada?)

III. Void function (C++)

b) Returns to the next statement after the point of call

2) Value returning subprogram

a) Generally called a function

b) Return is the point of call, replacing the call with the value it returns.

c) The call is not a separate statement

i. There is an issue with how to handle local variables of a subprogram

1) Statically

a) Efficient (execution wise)

b) Allow for history sensitivity

c) Restricts the ability to handle recursion

2) Dynamically

a) Has overhead at call and return

b) Could conserve storage

c) Prevents history sensitivity

d) Allows for recursion

j. Can subprograms be nested in definition?

k. Parameter passing methods

1) Pass by value

2) Pass by reference

3) By value-result

a) Copies the passed parameter to the matching local variable, allows changes to the local variables, and then passes back the values from each variable to the matching passed parameter.

4) By name

a) Replace the name of the actual with the formal.

b) This has to be handled by a chunk of code called the thunk.

5) By text

a) Does the same as by name, but does it in the environment of the call-ee

6) By result

7) By assignment (Python, Ruby)

l. Types of the actual parameters type checked against the types of the formal parameters.

m. Can subprograms be passed as parameters?

1) Problems

a) Knowing the referencing environment (what variables can the subprogram use?)

2) Solutions

a) Shallow binding – use the environment of the call statement that enacts the passed subprogram.

b) Deep binding – use the environment of the called subprogram.

c) Ad hoc binding – use the environment of the call statement that passed the subprogram (pg 418).

n. Does the language support overloading of subprogram names?

1) This requires that the parameter profile must differ.

2) This decreases readability, but writeability is increased.

o. Does the language support generic subprograms?

1) Not an actual subprogram, but is a pattern (template) which may be used to generate a subprogram based on certain parameters.

p. Does a function allow for side effects?

1) Any change to a variable that affects the scope of the call-ee, or global variables.

q. What type will a function allow to be returned?

1) Some allow only elementary types, others allow more complex types such as an array.

r. How many types values may a function return?

1) Is it only one, or a list of values.

s. Does the language support closures?

1) Closure – a subprogram along with the referencing environment of where it was defined.

3. Special categories

a. Coroutines – special subprogram with multiple entry points, history sensitivity, and a resume instead of a call.

b. Scheduled subprograms

1) Could be after or before another event

2) When a Boolean expression is true

3) When a clock time is reached

4) According to a priority

c. Concurrent subprograms – two or more subprograms which can execute at the same

d. Exception handler – subprogram which is called when a certain exception is encountered.

1) An exception should be a rare event.

2) Encountered at runtime.

4. Implementation of Subprograms

a. Two methods

1) Simple Subprogram Copy Rule

a) The effect of a subprogram call is the same as if a copy of the subprogram body replaces the call (with adjustments for parameters).

b) This results in 5 implicit assumptions:

I. No recursion

II. There must be an explicit call

III. The subprogram must execute entirely (copy the entire body).

IV. There is an immediate transfer of control to the body at the point of call.

V. There can only be one subprogram copied at a time.

c) This does away with (not in matching order to (b) ):

I. Recursion

II. Exception handling

III. Concurrency

IV. Coroutines

V. Scheduled subprograms  
 2) At call to a subprogram (referred to as **overhead**):

a) Save the status of the caller (CIP – Current Instruction Pointer; and CEP – Current Environment Pointer)

b) Compute and pass parameters

c) Transfer control to the callee by changing the CIP and the CEP.

d) At return to the caller (referred to as **overhead**):

I. Prepare the return value

II. Send any other output parameter values

III. Reset the CIP from the saved caller's value

IV. Reset the CEP from the saved caller's value

V. Transfer control to the caller (through the CIP)

**CHAPTER 11**

Abstraction

1. Definition – viewing an entity including only those attributes which are significant at the time.

2. Two types

a. Process abstraction (oldest type in programming) – manner in which to specify a series of steps and associate a name with those steps or actions

b. Data abstraction (appeared first with Simula 67) – creating an abstract data type

1) Data structure

2) Operations to manipulate

3) Enclosure for encapsulating the abstract data type

a) Information hiding – manner of separating the representation and implementation of an abstract data type.

I. Advantages of information hiding and using the ADT

i) Increased reliability

ii) Reduces the amount of code that the client must be concerned with

iii) Makes name conflicts less likely

iv) Changes to the implementation of the ADT need not require the client to change.

II. Disadvantages of ADT

i) Overhead

ii) Extra subprograms may need to be created which are only used within the ADT and not by the client

iii) Extra subprograms may need to be created that allows the client to get indirect access to the hidden data.

4) Issues concerning ADT within a language

a) Syntactic unit for enclosing

I. Special for just ADT

II. General enclosing method

b) Do you provide built-in operation on the ADT (assignment, comparison)

c) Can they be parameterized?

d) What access controls are provided?

e) Are the concepts of private, public and limited private supported?

**CHAPTER 12**

Object-Oriented Programming

1. Roots are in SIMULA 67

2. Was fully developed in Smalltalk 80

3. These type of languages provide:

a. Support for ADT

b. Support of Inheritance

c. Dynamic binding of method calls to methods

4.Terms

a. Class – structure used to implement an ADT

b. Object – an instance of a class

c. Derived or subclass – class defined using inheritance

d. Parent class – class used to create the derived class

e. Method – operation on an object (instance of a class)

f. Message – call to a method of a class

g. Message protocol/interface – entire collection of possible messages