

**CPS 400**  
**Organization of Programming**  
**Languages**

**Fall 1991**

**Prof. Betty H.C. Cheng**

**Department of Computer Science**

**Michigan State University**

**East Lansing, MI 48824-1027**

Copyright © September 1991 by Betty H.C. Cheng

# Overview

# Module 1: Languages and Language Processors

## Purpose:

- to explain the importance of study of programming languages,
- to give the characteristics of a good language, and
- to provide an overview of language processors.

## Concepts:

- Characteristics of a good language
- Attributes of a programming language
  - Data
  - Primitive operations
  - Sequence control
  - Data control
  - Storage management
  - Operating Environment
- Compilation and interpretation

# Module 2: Language Syntax

## Purpose:

- To introduce the syntactic elements
- the formal specification of syntax of a language

## Concepts:

- General syntactic criteria
- Syntactic elements of a language
- Formal definition of syntax  
(BNF notation)

# Module 3: Data Types and Representation

## Purpose:

- To introduce simple and structured data types
- their declaration and
- storage structures

## Concepts:

- Data Types
  - Integer, real, boolean, and character
  - Pointers
  - Array, record, set, list, and string
- User versus system defined data types
- Internal representation of data types
- Abstract data types

# Module 4: Operations on Data Types

## Purpose:

- To introduce the operation on data types and
- syntactic representation

## Concepts:

- Arithmetic and logic operations
- Operations on structured data types
- Assignment statements
- Operator precedence hierarchies and sequence control with expressions
- Prefix, infix, and postfix expressions and representations

# Module 5: Sequence Control

## Purpose:

- To introduce aspects of control of the order of execution
- of program statements

## Concepts:

- Sequence control between statements
  - implicit statement sequence control – parallel execution
  - labels and GOTO, conditional and case statements
  - iteration statements
- Subprogram sequence control
  - simple subprogram
  - recursive subprogram
  - interrupts
  - coroutines

# Module 6: Data Control

## Purpose:

- To introduce the basic concepts of data control,
- referencing environments and
- scope rules

## Concepts:

- Basic data control concepts
  - Names,
  - identifiers,
  - identifier associations, and
  - data control primitives
- Referencing environments
  - global
  - local
  - nonlocal
- Scope rules
  - static
  - dynamic
- Binding and binding time

# Module 7: Storage Management

## Purpose:

- To introduce the run-time elements requiring
- storage and
- storage management

## Concepts:

- Run-time elements
- Storage management phases
  - initial allocation
  - recovery
  - compaction
- Storage management
  - static
  - stack-based
  - heap

# Module 8: Introduction to Translation

## Purpose:

To introduce the stages in translation

## Concepts:

- Stages in translation
  - Analysis of the source program
  - Synthesis of the object program
  
- Lexical analysis
  - FSA's and scanning
  - Tokens and the symbol table
  
- Simple Parsing
  - Top-down parsing of arithmetic expressions
  - Simple precedence parsing of arithmetic expressions
  
- Discussion of code generation
  - Intermediate form such as quads
  - Machine dependent output and optimization

# Module 9: Case Studies in Common Computer Languages

## Purpose:

To contrast the features and implementation of at least two languages

## Concepts:

- Simple Procedural languages
  - FORTRAN
  - BASIC
  - COBOL
  
- Block structured language supporting recursion
  - ALGOL
  - Pascal
  - PL/I
  
- Functional languages
  - LISP
  - APL
  
- Other important languages
  - ADA (concurrency)
  - PROLOG (logic, declarative)
  - Smalltalk and C++ (object-oriented)

# Module 10: Specialized Architectures

Purpose:

To examine how hardware can be specially designed to support certain programming language features

# Motivation

# Programming Languages

## Language:

must help us write good programs, where a program is good if it is easy to read, easy to understand, and easy to modify. (Sethi89)

A programming language is a language intended to be used by a person to express a process by which a computer can solve a problem.

## 4 key components of definition:

- **Computer:** machine that will carry out the process described by program
- **Person:** programmer who serves as the source of the communication
- **Process:** the activity being described by the program
- **Problem:** actual system or environment where the problem arises

## Languages to satisfy 4 components:

- **Imperative** is geared towards easy machine translation.
- **Logical (Declarative):** geared towards people and the logical thinking process
- **Functional:** focused on the function or operation being performed
- **Object-Oriented:** geared towards the problem domain.

# Intro

# Introduction

## Why study programming languages?

- Organize computations
- Different problems have different needs
- Efficient programming
- Language provides
  - computation model
  - data types and operations
  - abstraction facilities
  - checking and enforcement

## Brief History

- Von Neumann Machine
- Assembly and machine language
  - tedious to write programs
  - difficult to understand and debug
- Higher level language
  - language closer to problem description
  - programs portable – exchange software
  - programs easier to understand

## What makes a good language?

- Clarity, simplicity, and unity of language concept
- Clarity of program syntax
- Naturalness for application
- Support for abstraction
- Ease of program verification
- Programming environment
- Portability of programs
- Cost of use:
  - execution
  - translation
  - creation, testing, and use
  - maintenance

# Language Processors

# Language Processors

## Aspects of Computer Structure

- **Data:** provide simple data structures
- **Primitive Operations:**  
simple operations to manipulate data
- **Data control:**  
mechanisms control data supplied to execution of operation
- **Storage Management:**  
mechanisms control allocation of storage for
  - programs
  - data
- **Operating environment:**  
mechanisms for communication with external environment (programs and data)

# Hardware Computer Organization

## Main Memory

- programs
- data

## Interpreter

- decodes each machine language instruction
- calls designated *primitive operation* with operands as input

## Primitives

- manipulate data in main memory and in high speed registers
- transmit programs or data between memory or *external operating environment*

# Data

## Data Storage

- **Main memory:**
  - linear sequence of bits
  - subdivided into fixed-length words or bytes
- **High-speed registers:**
  - word or address-length bit sequences
  - may have special subfields – directly accessible
- **External Files:**
  - subdivided into records
  - each record is sequence of bits or bytes

## Built-in data types

- Manipulated directly by hardware primitive operations
- Common :
  - integers,
  - single-precision reals,
  - fixed-length characters
  - fixed-length bit strings

# Operations

- Built-in primitive operations
- Paired 1-1 with operation codes (machine language)
- Examples:
  - Primitives for arithmetic: (+,-)
  - Primitives for testing properties ( $\geq$ ,  $\neq$ )
  - Primitives for accessing and modifying various parts of a data item (retrieve or store a character in a word)
  - Primitives for sequence control (GOTOs, returns)

# Sequence Control

## Program Address Register (PAR)

- Next instruction to be executed
- Register *always* contains memory address of next instruction
- Certain primitive operations can modify PAR (JUMP)

## Interpreter

- Uses PAR and guides sequence of operations
- Executes cyclic algorithm:
  - Gets address of next instruction from PAR
  - increments value of register by 1
  - fetches designated instruction from memory
  - decodes instruction into an operation code and set of operand designators
  - fetches the designated operands
  - calls designated operation with operands as arguments

# Data Control

## Computer must incorporate

- A means of designating operands
- Mechanisms for retrieving operands from operand designator
- Store results of primitive operations

## How to achieve operations

- associate integer *addresses* with memory locations
- provide operations for getting contents of a location with address

# Storage Management

## Simple

- No storage Management built into hardware
- Data in fixed place in memory (throughout execution)

## More Sophisticated

Facilities for *paging* (dynamic program relocation) in hardware

# Operating Environment

## Consists of:

- Peripheral storage
- Input-output devices
  - represent “outside world” to computer
  - all communication via operating environment
- Hardware distinctions between classes in environment
  - high-speed storage (magnetic drums, extended core)
  - medium-speed storage (magnetic disks)
  - low-speed storage (tapes)
  - input-output devices  
(readers, printers, terminals, console)

# Dynamic Operation

## Initial State

Initial contents of memory, registers and external storage

## State Transition

- Program execution of 1 statement:

$old\_state \rightarrow new\_state$

- Modifies contents of storage areas

## Final State

Final contents of storage areas

## Program Execution

Sequence of state transitions

# Hardware and Firmware

## Computer

- Integrated set of algorithms and data structures
- store programs
- execute programs

## Hardware

Machine language that defined by computer

- Can realize any precisely defined algorithm or data structure
- More efficient and flexible to have low-level machine language

## Firmware

- Alternative to hardware implementation of computer
- Simulated by *microprogram* (emulation)
  - run on *microprogrammable hardware computer*
  - machine language: low-level set of *microinstructions*
  - Simple transfers of data

# Translators and Software-Simulated Computers

## Goal:

- Low-level machine language (speed, flexibility, cost)
- High-level programming language

## Translation (compilation)

- Translate programs (high-level language - *source*) into machine language (*object*)
- Different types of translators
  - **compiler:** high-level → assembly
  - **assembler:** assembly → machine
  - **loader:** machine (relocatable) → machine
  - **link editor:**
    - 1 or more machine (relocatable) → single machine code
  - **preprocessor:** extended high-level → high-level

## Software simulation (interpretation)

- set of programs in machine language of host
- represent algorithms for execution of high-level computer
- Main simulator use interpreter algorithm
  - decode (high-level) and execute each statement
  - produce specified output from program

# Terms

## Actual Computer

- Hardware realization
- Physical devices realize data structures and algorithms

## Virtual Computer

- **Firmware:** microprogram a hardware computer
- **Software:** program language represent data and algorithms
- **Combination of software and firmware:**  
appropriate combination of translation and simulation

## Syntax of Programming Language

Form of components making up a program

## Semantics

Logical meaning of statements

# Binding and Binding Time

## Binding

establish a relationship between a program element and property

## Binding Time

Time during program processing when choice is made

## Classes of Binding Times

- **Language definition time**
  - kinds of statements
  - data structure types
  
- **Language implementation time**
  - representations of numbers
  - operations
  
- **Translation time** (compile time)
  - bindings chosen by programmer  
(names, types, statements)
  - bindings chosen by translator  
(storage locations, source program → particular object program)
  
- **Execution time** (run):
  - entry to subprogram or block (parameters)
  - arbitrary points during execution (assignments)

# Binding

## Types of Bindings

- **Early Binding:**
  - during translation
  - efficient
- **Late Binding:**
  - during execution
  - flexibility

# Syntax and Translation

# Syntax and Translation

## Purpose of Syntax

- Readability
- Writeability
- Ease of Translation  
Simpler syntax  $\leftrightarrow$  fewer translation rules
- Lack of Ambiguity

## Syntactic Elements of Language

- Characters
- Identifiers: string of letters and digits
- Operator Symbols
- Key and Reserved Words: words in statements
- Comments
- Blanks
- Delimiters and Brackets: mark beginning and end
- Free- and Fixed-Field Formats: special meaning to positions
- Expressions: basic statement building unit
- Statements: basic program building unit

- Overall Program-Subprogram Structure:
  - Separate subprograms (FORTRAN) : compiled separately
  - Nested procedures (Pascal) :
    - nonlocal referencing environment – efficient compilation

# Stages in Translation

## Analysis of Source Program

- **Lexical:** divide input into units

input → tokens

- **Syntactic:** (parsing)

tokens → (expressions, statements, subprograms)

- **Semantic:**

- symbol-table: contains identifiers
- implicit information (defaults)
- error detection
- compile-time processing:
  - macro inclusion
  - (conditional) substitutions during translation
- compile-time operation

- **Synthesis of Object Program**

- Optimization: make intermediate code efficient
- Code generation: assembly language or machine code
- Linking and loading:
  - combine pieces from translation into one final code

# Formal Definition of Syntax

## Grammar:

set of definitions (rules)

- Formal Grammar: well-defined
- Context-free grammar:
  - Tokens (terminals): (T) atomic symbols
  - Nonterminals: (NT) variables representing constructs
  - Productions: rules describe constructs

Production:  $NT ::= T^*NT^*$

- Start symbol
- BNF grammar: (Backus Naur Form)
  - $\langle \rangle$  enclose nonterminals,  
 $\langle \text{empty} \rangle$ : empty string
  - expression: sequence of terms separated by + or –.
  - term: sequence of factors separated by  $\times$  or **div**
  - factor: parenthesized expression, variable or constant
  - Production rules

$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle + \langle \text{term} \rangle$		$\langle \text{expression} \rangle - \langle \text{term} \rangle$	
$\langle \text{term} \rangle ::= \langle \text{term} \rangle \times \langle \text{factor} \rangle$		$\langle \text{term} \rangle \mathbf{div} \langle \text{factor} \rangle$	
$\langle \text{factor} \rangle ::= (\langle \text{expression} \rangle)$		$\langle \text{variable} \rangle$	

# Data Objects, Variables and Constants

# Data Objects, Variables and Constants

- Programmer-defined: declarations
- System-defined: run-time stacks, activation records generated as needed
- Data Value:
  - single number, character, or pointer
  - *pattern of bits* in storage
- Data Object:
  - container for data values
  - *block* of storage
  - attributes: logical organization, types of data values
  - elementary: single data value used as unit
  - bindings:
    1. *data values*
    2. *names*
    3. *data objects* –one component (pointer)
    4. *storage location* (storage mgmt)
- Variables and Constants:
  - **variable**: data object named and defined by programmer
  - **constant**: data object permanently bound to value
    - *literal*: name of constant is its representation (3,4)
    - *programmer-defined constant*: programmer selects name of constant

# Data Types

## Definition:

- *class* of data objects
- set of operations for *manipulating* and *creating*
- *primitive data types* defined by language
- *abstract data types* (ADTs) programmer defined

# Specification:

- **Attributes** specific to data object
  - invariant
  - type, name and values
  - *descriptor*: store attributes for execution
  - other attributes used for storage representation
- **Permissible values**
  - *ordered set* of values (least, greatest)
  - data object has only 1 value from set
- **Operations** for manipulation
  - form of *mathematical function*

# Problems with Data Type Definitions

- Operations undefined for inputs:  
subset of domain has undefined results

- Implicit arguments:

- Side effects: results not explicitly specified

```
function abs(x): integer
```

```
x:= x+1;
```

```
if x < 0 then abs:= -x
```

```
else abs:= x
```

```
endif
```

- Self-modification: (random number generator)

# Implementation:

- **Storage representation:**

- Uses storage provided by underlying hardware (usually)
- Otherwise, software simulated – less efficient execution
- How *attributes* are treated:
  - storage determined by attributes at compile time  
more efficient (FORTRAN, Pascal) (HW determined)
  - attributes stored in *descriptors* - run time use more flexible (LISP, SNOBOL) (SW simulated)

- **Algorithms** for operations: 3 ways to implement

1. *Hardware*: if storage representation for data object is provided by hardware (integer arithmetic)
2. *Procedure or function*: subprogram carry out operation
3. *In-line code*: represent sequence of HW operations

- defined in terms of *primitive data types*

# Declarations

## Type Declaration

- Program statement for translator
- number and type of data objects
- value of constants
- give lifetime of object

## Operation Declaration

- Information to language translator
- Number, order and data types of operands and results
- Declarations unnecessary for primitive operations
- Why use?:
  1. Choice of storage representations
  2. storage mgmt
  3. specify operation type for *overloaded* operators
  4. type checking

# Type Checking and Conversion

## Type Checking:

each argument of operation has correct type according to declarations

- **Static type checking:**
  - done at compile time
  - **advantages:**
    1. check all paths of execution – no errors
    2. efficient and storage is limited
  - **disadvantages:**
    1. constraints on language (declarations)
    2. compilation restrictions
  
- **Dynamic type checking**
  - done at run-time (before execution of operation)
  - store tag with data (check tag later)
  - **advantages:**
    1. flexibility (variable type depends on context)
    2. programmer not concerned about data types
  - **disadvantages:**
    1. difficult to debug
    2. more storage for tags
    3. implemented in software - slow execution

# Type Conversion

- conversion: explicit command to change type (ROUND)
- coercion: implicit conversion  
(Pascal:  $\text{int} \rightarrow \text{real}$  in a real operation)

# Assignment and Initialization

## Assignment

- change the *binding* of a value to a data object
- *side effect*

## Types of Assignments

- copy value: (Pascal)  
A := B (value of variable B assigned to variable A)
- copy pointer: (SNOBOL)  
A = B (ptr to value of variable B assigned to variable A)

## Initialization

- Declaration of variable allocates storage
- Storage may contain garbage information (cause errors)
- Must initialize!

# Numeric Data Types

- Integer:
  - Specification: type declaration
  - Operations: (binary, unary, relational, assignments)
  - Implementation: HW-defined (see book for specific types)
- Subranges:
  - Specification: subset of original type (*subtype*)
  - Implementation:
    - smaller storage requirements
    - better type checking
- Floating-point real numbers
- Misc. numeric data types

# Enumeration

- Specification:
  - ordered list of *distinct* values
  - programmer specifies literal names used for values
  - and ordering on values
- Implementation:
  - ordering gained through assignment of integer values
  - operations are implemented in HW

# Miscellaneous Types

- Boolean
  - logical operators (relational)
  - implemented with single bit of storage (0 or 1)
- characters
  - single character (letter, digit)
  - Operations: relational, assignment, or special class
  - supported by underlying HW and OS – because related to I/O

# Structured Data Types

# Structured Data Types

## Data Structure:

A data object constructed as an aggregate of other data objects.

## Component:

elements within a data structure

## Specification:

- number of components:
  - fixed-sized: arrays, records
  - variable-sized: sets, files, strings
  
- type of components
  - *homogeneous*: all elements of same type
  - *heterogeneous*: elements of different types
  
- name of components (selection purposes) and maximum size

## Specification of Structured Data Type cont'd

- organization:
  - linear: sequential (files, 1-dimen. arrays lists)
  - multi-dimensional: (records, 2-dimen. arrays, matrix)
  
- operations on data structures
  - selection: access one component of structure for processing
  - entire data structure operations: addition arrays, set union
  - modification of structure: changing storage requirement
    1. insertion/deletion of components
    2. creation/destruction of data structures

# Data Structure Implementation

- Storage Management
  - Sequential: contiguous block of storage has descriptor and components (fixed-size, homogeneous variable-sized structures)
  - Linked: several blocks in different parts of storage use pointers (for variable-size structures)
- Implementations of Operations on Data Structures
  - Sequential: use offset and size of each component
  - Linked: use link pointer field to get next element
- Storage Mgmt and Data Structures
  - Lifetime of data structure begins with declaration
  - ends with completion of routine that created it
  - **garbage**: all access paths to object are destroyed ties up storage
  - **dangling references**: access path to non-existent object modify storage allocated for other purposes

# Specific Structured Data Types

## Records

- a data structure made up of different types of components
- type and name of each subcomponent
- whole-data operations

## Characters

- Specification and Syntax:
  - fixed length (truncation or padding – Std Pascal)
  - variable length to a declared bound (UCSD Pascal)
  - unbounded length (SNOBOL4)
- Operations:
  - concatenation
  - relational operators (<) lexicographic
  - substring selection (by position)  
    str(1:5)
  - I/O formatting
  - substring selection by pattern matching (SNOBOL4)
- Implementations: see Figure

# More Complex Structured Data Types

## Variable-sized Data Structures

- use pointers (data object contains address of another data object)
- link together data objects

## Pointer Types

- reference data objects of one type (Pascal: declaration)
- reference different types (SNOBOL:context)
- implementation:
  - *absolute address*:
    - pro: efficient access
    - con: storage mgmt  
(move data, recover unused data)
  - *relative address*:
    - offset in some heap of storage  
(extra cost, base to offset)
    - one area or multiple areas
  - (Note: pointer storage mgmt adds to complexity of implementation – not in FORTRAN)

## Sets:

- unordered set of distinct values
- operations:
  - member
  - insertion/deletion
  - union, intersection, difference
- implementation:
  - *bit string*:
    - for small number of possible objects (restricted)
    - operations are Boolean  
(OR  $\equiv$  union, AND  $\equiv$  intersection)
  - *hash coded*:
    - efficient member/insert
    - inefficient union, intersection, difference
    - substantial amount of storage  
(double the number of expected values)
    - hash addressing by:
      - rehash
      - sequential scan
      - bucketing

# Files and I/O

## Files:

lifetime may be longer than program

- Sequential files:
  - fixed size data structures
  - not pointer-based since pointers are meaningless in a file
  - operators:
    - open O/S (read/write access)
    - read buffer variable
    - write
    - end-of-file test
    - close O/S
  - implementation: determined by O/S
- Direct Access Files:
  - similar to viewing file as a *vector* of memory
- Indexed Sequential Files:
  - random access then access data sequentially from that point

A *syntactically* correct program may:

- make no sense *semantically* (context-sensitive).
- not compile, even if semantically correct

# Examples of Bindings

$X := X + 10$

1. set of possible types for variable X:
  - set fixed by language definition (FORTRAN)
  - variable set (user-defined – ADA, Pascal)
  - choice by programmer (default in FORTRAN)
2. type of variable X:
  - often fixed at compile time (C, or conversion in Pascal)
  - sometimes bound during program execution (LISP)
3. set of possible values for X:
  - determined at implementation time
  - example: range of integers on PC or CRAY)
4. value of variable X:
  - bind a value to variable (initialization)
  - change binding – assignment
5. representation of constant 10:
  - characters “1” “0” represent ten
  - decimal, binary, octal
  - determined at language definition time
6. properties of operator ‘+’:
  - type of operation
  - set defined by language (real, int, complex)
  - choice defined at translation
  - value defined at execution (LISP)

# Subprograms and Programmer-Defined Data Types

# Subprograms and Programmer-Defined Data Types

## Abstraction

- address large, complex problems/programs
- modularize operations
- information hiding (black box concept)

## Subprograms

- implement modules
- procedures and functions
- generalize with module
- specialize with parameters

## Type Definitions

- gives name and structure of data type
- does NOT define permissible operations for type

## Abstract Data Types

- Programmer-defined data type
- Specify properties of data object
- Specify operations permitted
- Implementation hidden from user

# Sequence Control

# Sequence Control

## Definition:

Control the order of execution of statements.

## Sequence control within expressions

- Syntax:
  - *Prefix:*
    - Ordinary: Op, (Opnd1, ..., Opndn)
    - Cambridge: (Op Opnd1 ... Opndn)
    - Polish: Op Opnd1 ... Opndn
  - Postfix (suffix or reverse Polish )
  - Infix (binary)
    - precedence rules
    - parenthesis
    - easier to read
    - not good for operators greater than binary
  
- operator precedences determine evaluation order
  
- expression trees (root - top level operator),  
    each subtree represents expression
  
- execution: (machine code, tree structure (interpreter), pre-, postfix form)

## Sequence control between statements

- Composition (sequence of statements)
- Conditional (one or more possible branches – execute only one)
- Iteration (increment counter and/or condition holds)
- GOTOs and labels (taboo!)

# Subprograms

## Copy-rule view of subprograms:

- subprogram call  $\equiv$  in-line insertion copy of subprogram
- Properties:
  1. no recursion
  2. explicit CALL statements
  3. subprogram execute to its entirety
  4. immediate transfer of control at CALL
  5. only one program executing at a time
- Examples: FORTRAN and COBOL

## Implementation of Subprogram Invocation

- Program Definition and code (invariant)
- Activation
  - Activation Record
  - Current Instruction Pointer (CIP)
  - Current Environment Pointer (CIP)

## Recursive Routines

- Types of recursion
  - Tail recursion
  - Mutual recursion
- Nested activation records (AR)
- use *central* stack to manage AR

## Exceptions

- Conditions (*exceptions*) trigger an event
  - errors (OVERFLOW, OUT\_OF\_RANGE)
  - eof, headings
  - program monitoring
- *Exception handler* subprogram to process exception
  - declarations (local variables)
  - executable statements

## Scheduled subprograms

- scheduled events (after a condition is satisfied)
- specific time

## Coroutines

- subprogram gains control from another subroutine
- not execute entire program
- resume at point of execution in next invocation

## Concurrency

- assume have multiprocessor
- **tasks**: subprogram with special synchronization
- parent task start child task – run in parallel
- parent task cannot terminate until all child tasks finish
- each task execute at its own speed
- **real-time processing**: programs interact with input-output devices or other time-constrained devices.

## Data Structures and sequence control

- systematic processing of data structures
- recursive
- generators:  $\text{input} \times \text{program} \rightarrow \text{result}$
- iterators (CLU)

# Exceptions and Concurrency

## Exceptions:

Condition that requires some immediate action on the part of program

- Error: (arithmetic overflow, index out of range)
- Abnormal or rare occurrence: EOF
- Modification of specified storage location (debugging)
- exception handlers: procedures invoked implicitly by exception
- raising the exception: implicit invocation of an exception
- Propagation of Exception: If current unit does not contain handler, then terminate unit and propagate the exception to the parent routine.

# Synchronization

- synchronous conditions: conditions regarded as exceptions, arise at predictable places in program
- asynchronous conditions: may occur at any time
  - user-generated interrupt
  - device-ready signal
- achieve synchronization:
  - Interrupts: Task A sends interrupt to task B, execute “interrupt handler”, when complete, return control to B
  - Semaphores: data object:
    - integer counter
    - queue of tasks
    - 2 primitive operations, `SIGNAL(P)`, `WAIT(P)`, where P is counter
  - Rendezvous: 2 tasks synchronize actions for brief amount of time.

# Data Control

# Data Control

## Names:

- Identifiers (variables, parameters, constants, labels, types)
- Operations (subprograms, exceptions, primitives)

## Association:

a binding of an identifier to a data object

## Referencing Environments

- Local: declarations at beginning of program (subprogram)
- Nonlocal: applicable associations not declared locally
- global: associations from the execution of main program
- predefined: default associations (language definition)

## Aliases:

a data object available through more than one name in a single referencing environment

```
program main;  
var a,b: integer;  
    procedure first(x,y:integer);  
        y := x + a;  
    end;  
first(a,b);  
end.
```

## Dynamic Scope

- association(s) visible at time of execution
- includes subprograms activated in current environment

### Static scope

- determined by declarations
- Aid in translation:
  - relate variable to declaration
  - relate constant name to value
  - type names to declarations
  - formal parameters
  - subprogram names to their bodies

### Block structure

- Nesting of program structures (and declarations)
- Deeper levels declarations invisible (abstraction)
- Outer levels declaration available to inner subprograms

### Explicit common environments

- COMMON blocks (FORTRAN)
- variables in top level environment

# Shared Data: Parameters and Parameter Transmission

- Second major method for sharing data objects
- Most useful when a subprogram is to be given different data each time.

Argument Data sent to a subprogram

Result Data returned from a subprogram

## Parameters

is one way of transmitting arguments and results

Formal

- Particular kind of local data object within subprogram
- Subprogram lists names and declarations in header
- Parameter name is *simple identifier*
- Declaration gives type and other attributes

Example:

Actual

- Data object shared with subprogram explicitly transmitted from caller
- may be local data belonging to caller
- nonlocal data object visible to caller
- result returned by function invoked by caller and immediately transmitted to called subprogram
- Represented at point of call by an expression:  
*actual-parameter expression:*
- Expression same form as any other expression in language
- Expression is *evaluated* at time of call before subprogram is entered.

Examples:

Data Control-4

# Parameters cont'd

## Establishing Correspondence

- Positional:  
Pairing actual and formal parameters based on respective positions
- Correspondence by explicit name:  
Pair explicitly the formal and the actual parameters

## Transmitting Parameters

- A *data object* has the value of the evaluated actual-parameter expression. And it will act as the actual parameter.
- Data object somewhere in memory.
- Beginning of execution of subprogram, need to initialize pointers to actual parameters. (2 methods)
  1. Ptr to actual-parameter data object is copied into formal param locn. OR
  2. Entire actual-parameter data object is copied into formal param. locn.

## Two steps for transmission:

1. In calling subprogram, each actual-parameter expression is evaluated to give a pointer to actual-param data object.
  - a list of pointers stored in common storage area accessible to subprogram being called (registers)
  - Control transferred to subprogram
  - activation record for subprogram created,
  - return point established, etc.
2. In called subprogram, as it begins execution
  - list of ptrs to actual params is accessed
  - formal params are initialized before stmts in body are executed

During execution of subprogram references to formal parameter names are treated as ordinary local variables

# Types of Parameter Transmission

- IN OUT
  - Transmission by reference (location)
  - Transmission by value-result:
  
- IN-only
  - Transmission by value:
  - Transmission by constant value:
  
- OUT-only A parameter *transmitted by result*  
Properties:

Using *explicit function values* is another way to return values from a subprogram

# Implementation of Parameter Transmission

- Type specified for parameters
  - Data object is of type specified
  - Data object is of type pointer
- Type not specified:  
formal parameter is local ptr variable (not type specific)

# Actions associated with transmission

- Point of *call*
  - evaluate actual-param expression
  - list of ptrs to actual-params data objects set up
  - takes place in the *referencing* environment
  - then control transferred to called program
- Point of *entry* and *exit* (after transferred control)
  - *prologue* completes actions associated with transmission
  - either copying entire contents of actual param into formal param OR
  - copy ptr to actual param into formal param.
  - BEFORE subprogram, terminates, *epilogue* for subprogram copies result and value-result values into actual parameters.
  - Function values copied into register or temporary storage
  - After subprogram terminates, activation record and contents lost.

## Main actions of compiler for transmission

- Generate executable code:
  - for transmission of parameter,
  - for return of results, and
  - for each reference to formal-param name
- Static type checking

# Subprograms as Parameters

- Subprogram transmitted as an actual parameter to another subprog.
- actual-parameter expression is the *name* of the subprog.

## Two Problems with subprograms as parameters:

### 1. Static Type Checking:

- insure that calls include proper number and type of actual parameters for subprogram being called.
- Compiler needs full specification for formal params  $R$  that include type proc or func and number and types of params.

### 2. Nonlocal references (free variables):

- *free variable*: no local binding within the subprogram definition
- can't use "normal" nonlocal referencing rules
- a nonlocal reference (reference to a free variable) should mean the same thing during the execution of the subprog passed as a param as it would *if the subprog were invoked at the point where it appears as an actual param in the param list.*)

# Unevaluated Parameters: Transmission by Name

- leave actual params unevaluated until point of use
- implement by: treating actual params as simple parameterless subprograms (*thunks*).
- Allows uniform handling of subprogram params.
- Whenever formal param corresponding to by-name param is referenced in subprogram, the thunk compiled for that parameter is executed resulting in evaluation of the actual param in proper referencing environment
- same as by-reference except if have params containing more than one reference to variable

```
procedure R(var I,J: integer)
  begin
  I := I + 1;
  J := J + 1;
  write(I,j);
  end;
```

And R is called from P with:

```
M := 2;
R(M,C[M]);
```

# Tasks and Shared Data

Tasks, coroutines and exception handlers all have a problem with *mutual exclusion*.  
Operations on a set of shared data.

## Potential Solutions:

- **Critical Region:**  
sequence of prog. stmts within a task where the task is operating on a shared data object. Only one task execute region at a time.
- **Monitors:**
  - Shared data object with set of operations (ADT).
  - only one operation is executed at a time.
- **Message Passing:**
  - only access to shared data. - may need buffer to hold messages
  - or use rendezvous (Ada) to exchange info

## Storage Management

# Storage Management

## Run-time elements needing storage

- code segments
- system programs
- user-defined data structures
- return points
- referencing environments
- system info (tables, reference counts)
- temporaries (expression evaluations, parameters, I/O buffers)

## Allocating and Freeing Storage

- subprogram call (allocate space for activation record, local ref. env., and data ) and return (free)
- NEW and DISPOSE operations
- insertion and deletion of components of data structures (LISP)

## Storage management phases

- Initial Allocation:
  - free: available for dynamic allocation during execution
  - allocated: store code, declarations
  - maintenance of free storage
  
- Recovery:
  - recover used storage for reuse
  - reposition stack-pointer (Pascal) OR
  - garbage collection (LISP)
  
- Compaction and Reuse:
  - free storage scattered
  - compact into chunk (block) of storage to make usable

# Static Storage Management

- Allocation is fixed throughout execution (FORTRAN)
- Subprograms compiled separately
- Setup activation records,
- Space for code segments
- No storage management – efficient!

# Dynamic Storage Management

## Stack-based:

- free storage at beginning of execution (sequential)
- allocation occurs at one end
- must be freed in reverse order (LIFO)
- stack-pointer controls mgmt

## Heap Storage Management: (LISP)

- block of storage where pieces are allocated and freed in *unstructured* manner.
- Use: storage needs to be allocated and freed at random points
- Two types: depending fixed or variable-sized elements
- Fixed sized elements
- Variable-sized elements

# Heap: fixed-size elements

- Divided into equal-sized blocks
- Initially in contiguous block of memory
- Link elements together into *free list*
- Recovery: (of space)
  - *Explicit Return*
    - explicit command to free storage
    - create problem with garbage and dangling references
  - *Reference Counts*:
    - no dangling references and no garbage
    - keep track of number of references
    - when count reaches 0, return to memory
    - need more storage (count bit)
    - more execution time (less efficient)
  - *Garbage Collection (no garbage)*
    - no dangling references (allow garbage)
    - when no free space left, collect garbage
    - 2-step process (not done often)
      1. mark garbage bits (on/off)
      2. link together storage if garbage bit on

# Heap: Variable-sized elements:

## Allocation:

- *Heap pointer*: indicates starting point of free storage
- When reach end, reuse storage

## Reuse:

- **Free-list**: if need N-word block, but have N+M word length block, split block and return remaining M-word block to free list.
  - first-fit
  - best-fit
- **Recovery**
  - partial compaction (if cannot shift blocks, merge adjacent ones)
  - full compaction (eliminate fragmentation, free-list)

# Prolog

# Prolog

## Definition:

- Computer programming language
- solve problems involving *objects* and
- *relationships* between objects.

## Programming in Prolog involves:

- declaring *facts* about objects and their relationships
- defining some *rules* about objects and relationships
- asking *questions* about objects and relationships

# Components of Prolog

- Facts:
- Questions:
- Variables:
  - Uninstantiated (free)
  - Instantiated (bound)
- Conjunctions:
- Rules:
  - Variable stands for the same object within a rule
  - Define relationships of object
- Backtracking:

# Prolog Syntax

Prolog program built from *terms*.

- **Constants:** Name objects or relationships
  - atoms (begin with lower case letter)
  - integers
- **Variable:**
  - begin with upper case letters or ‘\_’
  - corresponds to the use of pronouns in English
  - anonymous variables
- **Structure:**
  - Single object consisting of a collection of other objects (components).
  - Specify *functor* and its *components*.

# Logic

## Propositional Logic

Represent propositions about the world – describe objects.

## Predicate Logic (Calculus)

- Use *terms* to represent objects
- Terms: made up of *constants, variables, compound term*.
- Predicates represent relationships between objects
- Atomic Proposition: predicate symbol, with ordered set of terms as its arguments.

- Compound Propositions: Using atomic propositions and logical connectives

Connective	Syntax	Semantics
Negation	$\sim a$	“not a”
Conjunction	$a \& b$	“a and b”
Disjunction	$a \# b$	“a or b”
Implication	$a \rightarrow b$	“a implies b”
Equivalence	$a \equiv b$	“a is equivalent to b”
$\forall v.P$	<b>all</b> (v,P)	“P is true for whatever v stands for”
$\exists v.P$	<b>exists</b> (v,P)	“there is something that v can stand for such that P is true”

- Can rewrite  $\leftrightarrow$ ,  $\rightarrow$  in terms of **and**, **or**.

# Clausal Form

## Universal form for expressing information

Follow a set of steps in rewriting the other symbols in terms of **and**, **or**.

1. Remove Implications
2. Move negation inwards
3. Skolemize
4. Move universal quantifiers outwards
5. Distribute **and** over **or**.
6. Put into clausal form

# Unification Algorithm

Input: set of expressions  $W$ ,

Output: set  $\sigma$  of unifiers.

1. Set  $k = 0$ ,  $W_k = W$ , and  $\sigma_k = \emptyset$ ,  
where  $W$  is a set of expressions.
2. If  $W_k$  is a singleton, stop;  
 $\sigma_k$  is a most general unifier for  $W$ .  
Otherwise, find the disagreement set  $D_k$  of  $W_k$ .
3. If there exist elements  $v_k$  and  $t_k$  in  $D_k$  such that  $v_k$  is a variable that does not occur  
in  $t_k$ , go to (3).  
Otherwise, stop;  $W$  is not unifiable.
4. Let  $\sigma_{k+1} = \sigma_k^{v_k t_k}$  and  
 $W_{k+1} = W_k^{v_k t_k}$ .
5. Set  $k = k + 1$  and go to (2).

# Deduction Process

- Inference Engine:

- Core of logic programming
- Derive new facts from facts and rules
- Base on given goal
- Two operations used: resolution and unification

- Resolution:

- Given rule of form

$$\begin{array}{l} f \text{ if } a_1, a_2, \dots, a_n \\ g \text{ if } f, b_1, b_2, \dots, b_m \end{array}$$

- Conclude:

$$g \text{ if } a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m$$

- Possible Combinations:

1. New rule from 2 rules
2. New rule from rule and a fact
3. New fact from a rule and a fact

- Unification:

- Derive a new rule from given rule
- Through binding of variables
- Possible Combinations:
  1. Rule unified with fact
  2. Reduction using a fact

# Differences Between Prolog and Pure Logic

- Search order  
corresponds to physical order of rules and facts
- Cut (!)
- Assertions (`asserta` and `assertz`)
- Input/Output
- Data structures

# Good Applications for Prolog

- Search problems
- Parsing problems
- Pattern Matching

# Database Query Languages

- Relation:  
table of relationships
  - tuple: (row) represents objects
  - attribute: (column) represents attributes
- Database made up of one or more relations
- Query Language:
  - Manipulates data
  - Store data in tables
  - Retrieve data from tables
- Tools for organizing data facilitate:
  - efficient data entry
  - efficient retrieval
  - formatting reports

# SQL Query Language:

- Creation of relation:

```
CREATE STUDENT( NAME      CHAR(20),
                CLASS     CHAR(2),
                SEX       CHAR(1),
                MAJOR     CHAR(10),
                AGE       INTEGER)
```

- Insertion:

```
INSERT INTO STUDENT (NAME,CLASS,SEX,MAJOR,AGE)
VALUES ('Jane', 'SR', 'F','CPS',21)
```

- Retrievals:

- Single field retrieval:

```
SELECT NAME
      FROM STUDENT
      WHERE AGE > 20
```

- Multiple fields retrieval:

```
SELECT NAME,AGE
      FROM STUDENT
      WHERE AGE > 20
```

- Multiple Relations retrieval:

```
SELECT STUDENT.NAME,STUDENT.AGE
      FROM STUDENT,ATHLETE
      WHERE STUDENT.NAME = ATHLETE.NAME
```

# Views in SQL

- View in SQL conceptually the same as relation
- View consists of tuples and attributes
- View not found in storage of computer
  - Derived from existing relations and definition of view
  - Example:

```
CREATE VIEW STUDENT_ATHLETE
AS   SELECT STUDENT.NAME, STUDENT.AGE, ATHLETE.SPORT
      FROM STUDENT, ATHLETE
      WHERE STUDENT.NAME = ATHLETE.NAME
```

- Retrievals based on view:

```
SELECT NAME, AGE
      FROM STUDENT_ATHLETE
      WHERE AGE > 20
```

- All changes to relations are propagated to views.

# SQL as a Logic Language:

- Most logic model capabilities can be captured with SQL's INSERT, VIEW, SELECT operations
  - Assertion of facts (INSERT)
  - Rules (VIEW)
  - Goals (SELECT)

# Unification Algorithm

Input: set of expressions  $W$ ,

Output: set  $\sigma$  of unifiers.

1. Set  $k = 0$ ,  $W_k = W$ , and  $\sigma_k = \emptyset$ ,  
where  $W$  is a set of expressions.
2. If  $W_k$  is a singleton, stop;  
 $\sigma_k$  is a most general unifier for  $W$ .  
Otherwise, find the disagreement set  $D_k$  of  $W_k$ .
3. If there exist elements  $v_k$  and  $t_k$  in  $D_k$  such that  $v_k$  is a variable that does not occur  
in  $t_k$ , go to (3).  
Otherwise, stop;  $W$  is not unifiable.
4. Let  $\sigma_{k+1} = \sigma_k^{v_k t_k}$  and  
 $W_{k+1} = W_k^{v_k t_k}$ .
5. Set  $k = k + 1$  and go to (2).

**LISP**

# Functional Programming

## Three main characteristics:

- The value of an expression depend only on the values of its subexpressions, if any
- *Implicit storage management*  
(allocate as needed and inaccessible storage is deallocated)
- *Functions are first-class values.*  
Functions have same status as any other values :
  - value of expression
  - passed as an argument
  - put in a data structure

# Basic Structures and Functions

- Cons is the list constructor
  - Lists are built out of cells
  - Cells contain pointers to head and tail of list
- car points to the head of the list
- cdr points to the tail of the list

## Notions of equality

- equal
- eq

# Data Types

- lists and atoms
- each data object carries a run-time descriptor giving its type and other attributes
- if data object has components, use pointer to components
- atom: basic elementary type of data object
  - complex data object represented by a location in memory which contains the type descriptor for atom and pointer to *property list*
  - Property list: contains various properties associated with the atom:
    - print name (character string representing the atom for input and output)
    - may include function named by the atom
    - various other bindings
  - Every atom also appears as a component in a central system-defined table called the object list (ob-list)
    - organized as a hash-coded table
    - allows efficient lookup of a print name (character string)
    - and retrieval of pointer to atom with corresponding print name
- Lists:
  - singly linked structures
  - Each list element contains pointer to a data item and a pointer to the following list element

# Allocation and Deallocation

- short-lived:
  
- longer-lifetime:

## Two approaches to deallocation of cells

- **Lazy approach:**  
Wait until memory runs out, work comes to halt when collecting garbage (takes time to examine a cell or manipulate a pointer)
  
- **Eager approach:**  
Each time a cell is reached, check whether cell will be needed after operation. If not, deallocate cell – place on *free list*. (Can also use *reference counts*.)

## Implementation of Deallocation

- *Mark-sweep*
  1. **Mark phase:** mark all cells that can be reached by following pointers.
  2. **Sweep phase:** Sweep through memory, looking for unmarked cells. (Starts at one end of memory and looks at every cell.)

# Referencing Environments

## Association List

- A-list represents a referencing environment.
- LISP list, each of whose elements is a pointer to a word representing an identifier (atom) and its current association
- Association pairs are list words whose CAR points to the atom and whose CDR points to the value (list, atom, or other data object)

# Subprograms

- activated when function is called
- APPLY:  
evaluates arguments and adds to current referencing environment (“most recent association”), then calls EVAL
- EVAL:
  - looks up variables in referencing environment
  - traverses the list structure representing the body of the function
  - calls APPLY to evaluate internal function calls

# Parameter Transmission

- can be specified by programmer
- Value: (default)
- Name: why use?
  - variable number of arguments
  - where evaluation of one or more of arguments is undesirable (COND, DEFUN)
  - macros

# LISP

- Running LISP
  - At the UNIX prompt: `cl`
  - If have memory problems, run `cl` from `buster` (using X-Window)
  - Exiting LISP
    - At `< cl >` prompt: `(exit)`
- Creating and Running Programs
  - Use EMACS/MOCKLISP
  - Use any text editor (EMACS, VI, XEDIT)
    - at `< cl >` prompt, enter `(LOAD "filename")`
  - Creating a sample run `(dribble "filename")`
    - `(dribble)`
- LISP as an interactive environment
  - Functional Paradigm
  - Running a program - Calling a function

# LISP Notation

- LISP Notation

- Prefix
- Motivational Examples

(+ 1 2)

(+ 2 (- 4 1))

(> 6 4)

(first '(a b c))

- Atoms, Lists, List Elements and Expressions

- ATOMS:

- Simplest type of LISP Objects
  - “indivisible”
  - SYMBOLS
  - Two Types : Symbolic, Numeric
- EX) BOB, 1, NAME, X, +
- Two special atoms : NIL, T

- LISTS:

- non-atomic, a sequence of items
- EX) (THIS IS AN EXAMPLE OF A LIST)

\*\* 7 elements in the list

(+ 1 2)

\*\* 3 elements

- lists can also have lists as their elements
- (NAME (BOB BOURDEAU) OCCUPATIONS (TEACHER CONSULTANT))
- \*\* 4 elements

- lists are a flexible general purpose data structure
- (DEPARTMENTS (COMPUTER-SCIENCE 20) (MATHEMATICS 30))
- \*\* a sequence of records
- (10 (3 2 5) (25 (15 13 20) 31))
- \*\* a binary search tree

– EXPRESSIONS:

- atoms and lists are often referred to as expressions, meaning that their “values” are of interest v

• Bindings

– associates an atom with a value (not really assignment)

(SETF atom value)

– Example:

```
(SETF name 'Bob)
```

```
(SETF fullname '(Bob Bourdeau))
```

```
(SETF x 12)
```

```
(SETF formula '(+ x 2))
```

# Evaluation Rules

- Atoms:

- numeric --> themselves
- symbolic --> binding (error if none exists)
- Examples:

```
1 --> 1
x --> 12
name --> Bob
```

- Two Special Atoms:

```
NIL --> NIL == ()      (logical False in LISP)
T   --> T
```

Note: NIL is both an atom AND a list

- Lists:

- Interpreted as function calls in prefix-format
- First element = functor
- Remaining Elements = parameters
  1. Evaluate each parameter recursively
  2. Apply functor to list of actual parameters
- Example:

```
(+ 1 2) --> 3
(+ 1 (- 3 4)) --> 0
(+ 1 x) --> 13
```

- Stopping Evaluation:

- The QUOTE function (and its macro)
  - simply returns its argument
- Example:

```
(QUOTE ROBERT-BOURDEAU) == ROBERT-BOURDEAU  
formula == (+ x 2)
```

# Standard Functions and Predicates

- NOTE: these are all non-destructive

- List Manipulation Primitives

- CAR/FIRST     (`car -list-`)  
returns first element of `-list-`
- CDR/REST     (`cdr -list-`)  
returns a copy of `-list-` with the first element removed
- CONS     (`cons -newhead- -list` )  
returns a copy of `-list-` with `-newhead-` added to left
- Example:

```
(car '(1 2 3))
```

```
(car (car '((APPLE GRAPE) PEAR) ) )
```

```
(car '(car ((APPLE GRAPE) PEAR) ) )
```

```
(cdr '(a b))
```

```
(cdr '(a))
```

```
(cons 'a '(b c))
```

```
(cons 'x NIL)
```

```
(cons 'x 'y)     --- beware
```

- How would you write a statement to return the following:

1. 3rd member of (A B C D E)
2. 2nd member of (A (B (C D) E) F)
3. The list (a (b) (c d))

- Other List Functions

```
(MEMBER -el- -list-)
```

- Arithmetic and Relational Functions +, -, \*, /  
>, <, >=, <=, =

How would the following be written as an s-expression?

2+3\*4

2/3

10+5/3\*2

- Predicates  
- return NIL or T only

(numberp -exp- )

(zerop -exp- )

(NULL -exp- )

(listp -exp- )

(equal -exp1- -exp2- )

SAME VALUE : most general, most expensive

(eq -exp1- -exp2- )

SAME INTERNAL POINTER: most picky, least expensive

(eql -exp1- -exp2- )

same as EQ except numbers test as in EQUAL: compromise

- Logical Connectives

(AND -exp1- -exp2- .... -expN- )

(OR )

(NOT -exp-)

# Function Definition

- Syntax:

```
(DEFUN functor  -parameter list-  
                -expression 1-  
                -expression 2-  
                ....  
                -expression n-  
)
```

returns the value of the last evaluated expression

- Examples:

```
(defun double (x)  
  (* x 2)  
)
```

```
<cl> (double 4)
```

```
(defun second (l)  
  (car (cdr l))  
)
```

```
<cl> (second '(a b c))
```

```
(defun dot-prod-3 (v1 v2)  
  (+
```

```
        (* (car v1) (car v2))
        (* (cadr v1) (cadr v2))
        (* (caddr v1) (caddr v2))
    )
)
```

```
<cl> (dot-prod-3 '(1 2 3) '(4 3 2))
```

```
(defun Stupid (x)
  (* 2 x)
  (+ 4 (* 6 (- 1 2) x))
  (setf g x)
  12
)
```

# Debugging and More Functions

- Debugging

- (trace -functor- )
- (notrace -functor- )

## 11. The COND control Structure

- similar to 'switch' in C, and 'case' in Pascal
- Syntax:

```
(COND ( guard1  expressions )
      ( guard2  expressions )
      .....
      ( guardN  expressions )
      )
```

- Semantics:
  - guards are tested from top to bottom
  - first non-NIL guard ==> evaluate the subsequent expressions
  - return the value of the last expression evaluated

- Examples:

```
(cond ( (equal x 12) (* x 2))
      ( (listp x) (car x))
      ( T (car x) (cdr x) 14)
      )
```

## More Complicated Functions

- Example1

```
(defun head-is-a-number-or-atom (l)
  (cond ((numberp (car l)) t)
        ((atom (car l)) t)
        (t NIL))
)
```

- Example2:

```
(defun power (x n)
  (cond ((= n 0) 1)
        (< n 0) NIL)
        (t (* x (power x (- n 1)))))
)
```

# FORTRAN

# FORTRAN

## History

- Developed in the mid 50's
- Main goal: efficiency
- First high-level language to be widely used
- FORTRAN program consists of a MAIN program and a set of subprograms
- Each statement begins with a *statement label* field. (first 5 columns)

# Data Types

- Numerical: Integer, Real, double precision real, complex (pair of real numbers stored in 2-words)
- Logical: Boolean (.TRUE., .FALSE., .NOT., .AND., .OR.)
- Variables and Constants:
  - Type of variable must be declared
  - If no explicit, then use naming convention
  -
- Arrays:
  - Subscript ranges declared explicitly
  - Upper bound of 7 dimensions
  - Use *EQUIVALENCE* to refer to the same storage area.
- Type Checking
  - Performed at compile time
  - Incomplete! Why?

## Greatest Weakness of FORTRAN

restricted data structuring facilities (arrays and character strings of fixed length).

# Subprograms

- Subprograms are the only abstraction available in FORTRAN
- 3 types of subprograms
  - Function: return single value as result
  - Subroutine: results in parameters or COMMON variables
  - Statement Functions: value computed in a single arithmetic or logical expression. local to the subprogram that it is defined within.

# Sequence Control

- Expressions: associativity is from left to right or use parenthesis
- statement sequence control:
  - statement labels, GOTOs (3 types)
  - conditionals:
    1. IF (expr)  $n_{neg}, n_{zero}, n_{pos}$
    2. IF (expr) *statement*
    3. IF (test expr) THEN  
sequence of statements  
ELSEIF  
...  
END IF
  - iteration:  
DO *stmt-num int-var=init-val,term-val,increment-val*
- Subprogram sequence control:
  - Only the basic call and return
  - No recursion

# Data Control

- Local referencing environments:  
local environment is retained between calls because activation record is allocated statically.
- Common Environments:
  - sets of variables and arrays
  - stored in COMMON blocks
  - only the name is global identifier
  - only the block of storage is being shared, not the identifiers themselves:

```
COMMON /BLK/X,Y,K(25)
```

```
COMMON /BLK/U,V,I(5),M(4,5)
```

- Parameters and results:
  - Parameter transmission is by reference (value-result)
  - Results of function subprograms == assignment to name of subprogram

# Compiler Overview

# Compilers

- Lexical Analysis
- Parsing
- Semantics
- Optimization
- Code Generation

# Symbol Table

- Lexical analyzer: puts in character string or *lexeme* (sequence of input characters comprising a single token) which forms an identifier.
- Later phases add attributes to identifier (type, usage, position in storage).
- Code generation phase use info to generate proper code to store and access this variable

# Lexical Analysis

## Scanner and Lexical Analyzer

- Read input characters
- Produce a sequence of tokens (for syntax analysis)
- Strip out comments and white space (newline, tabs, blanks)

## Why bother with Lexical Analysis

- **Simplify** either the Lexical or Parsing phases  
(handling of white space, designing a new language)
- Compiler **efficiency** is improved.  
A separate lexical analyzer permits construction of a specialized and potentially more efficient processor for the task. A large amount of time spent reading source program and partitioning into tokens. Specialized buffering techniques for reading input characters and processing tokens can speed up performance of compiler.
- Compiler **portability** is enhanced.  
Input alphabet peculiarities and device-specific information can be restricted to lexical analyzer.

# Parsing

Process of determining if a string of tokens can be generated by a grammar

- Bottom-up
  - Begin with leaves in constructing parse trees
  - “Reduce” a string  $w$  to start symbol of grammar
  
- Top-Down
  - Start with root
  - Repeat the following steps:
    1. At node  $n$ , labeled with nonterm  $A$ , select one of the productions for  $A$ , construct children at  $n$  for symbols on right side of production.
    2. Find next node at which a subtree is to be constructed.

# Top-Down Parsing

- Left-most derivation for input string
- parse tree in preorder (start with root)
  
- Recursive-Descent Parsing:
  - Execute a set of recursive procedures to process input.
  - A procedure is associated with each nonterminal of a grammar
  - May involve backtracking (repeat scans of input)
  - Left-recursive grammar can cause infinite loop, why?
  
- Predictive Parsers:
  - Special case of recursive-descent
  - Lookahead symbol unambiguously determines procedure selected for each nonterminal.
  - Eliminate left recursion
  - Left factor resulting grammar
  - Obtain grammar that needs no backtracking
  - Use **FIRST** sets:  
let  $\alpha$  be the right side of a production for nonterminal A  
 $\text{FIRST}(\alpha)$ : set of tokens that appear as the first symbols of one or more string generated from  $\alpha$ .
  - The lookahead symbol unambiguously determines the procedure for each nonterminal
  - Use  $\in$ -productions as default when no other productions apply
  - Implementation of Predictive Parsers:  
Use Transition Diagrams:
    - Eliminate left recursion
    - Left factor grammar
    - For each production  $A \rightarrow X_1 X_2 \dots X_n$ , create path from initial state to final state, label edges with  $X_1, X_2, \dots, X_n$ .

# Nonrecursive Predictive Parsing

- Algorithm for nonrecursive predicting parsing
- FIRST and FOLLOW sets
- Construction of Predictive Parsing Tables (see algorithm)
- LL(1) Grammars
  - No multiply-defined entries in parsing table
  - Second 'L' stands for leftmost derivation
  - (1) stands for 1 symbol lookahead
  - Properties:
    - No ambiguous or left-recursive grammar is LL(1)
    - Grammar G is LL(1) iff  $A \rightarrow \alpha \mid \beta$  are two distinct productions of G, the following conditions hold
      1. For no terminal  $a$  do both  $\alpha$  and  $\beta$  derive string beginning with  $a$ .
      2. At most one of  $\alpha$  and  $\beta$  can derive the empty string.
      3. if  $\beta \xRightarrow{*} \epsilon$ , then  $\alpha$  does not derive any string beginning with a terminal in FOLLOW(A).

- Error Recovery
  - panic mode  
(discard symbols until get good token)
  - phrase level  
(perform local correction, replace portion of input string)
  - error productions  
(augment grammar to acct for error cases)
  - global correction  
(algs to choose minimal sequence of changes to obtain globally least-cost correction).

# Bottom-up Parsing

- **Shift-reducing** parsing

- Construct parse tree beginning at leaves
- Working up towards to root (top)
- “Reduce” a string  $w$  to start symbol of grammar
- At each *reduction* step, replace particular substring matching the right side of production by symbol on left of that production.
- If substring chosen correctly at each step, rightmost derivation is traced out in reverse

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence `abbcd` can be reduced to  $S$  by:

`abbcd`

`aAbcd`

`aAde`

`aABe`

$S$

- **Viable Prefixes:**

Set of prefixes of right sentential forms that can appear on the stack of shift-reduce parser.

- **Conflicts during Shift-Reduce Parsing:**

Given contents of stack and next input symbol, not know what action to take. (e.g. ambiguous grammar can never be LR.)

# Shift-Reducing

- Handles: A “Handle” of a string is a substring that matches the right side of a production.
- Whose redux to the nonterminal on the left side of produx represents one step along the reverse of a rightmost derivation.
- Formally, a *handle* of a right-sentential form  $\gamma$  is a production  $A \rightarrow \beta$  and a position of  $\gamma$  where string  $\beta$  may be found an replaced by  $a$  to produce the previous right-sentential form in a rightmost derivation of  $\gamma$ .

If  $S \xrightarrow{*}_{rm} \alpha Aw \xrightarrow{*}_{rm} \alpha\beta w$

then  $A \rightarrow \beta$  in position following  $\alpha$  is a handle of  $\alpha\beta w$ .

(1)  $E \rightarrow E + E$

(2)  $E \rightarrow E * E$

(3)  $E \rightarrow (E)$

(4)  $E \rightarrow id$

and the rightmost derivation

$$\begin{aligned}
 E &\Rightarrow_{rm} \underline{E + E} \\
 &\Rightarrow_{rm} E + \underline{E * E} \\
 &\Rightarrow_{rm} E + E * \underline{id_3} \\
 &\Rightarrow_{rm} E + \underline{id_2} * id_3 \\
 &\Rightarrow_{rm} \underline{id_1} + id_2 * id_3
 \end{aligned}$$

- Handle Pruning

- Start with a string of terminals  $w$  to be parsed
- If  $w$  is a sentence of the grammar, then  $w = \gamma_n$ , where  $\gamma_n$  is the  $n$ th right-sentential form of some as yet unknown rightmost derivation
$$S = \gamma_0 \xrightarrow{rm} \gamma_1 \xrightarrow{rm} \gamma_2 \dots \xrightarrow{rm} \gamma_{n-1} \xrightarrow{rm} \gamma_n = w.$$
- Locate the handle  $\beta_n$  in  $\gamma_n$  and replace  $\beta_n$  by the left side of some production  $A_n \rightarrow \beta_n$  to obtain the  $(n-1)$ st right-sentential form  $\gamma_{n-1}$ . (Repeat)

- Stack Implementation:

- Two problems:
  1. locate substring to be reduced
  2. determine what production to be choose (if more than 1)
- Use **stack** to hold grammar symbols
- Use input buffer to hold string  $w$  to be parsed
- Shift zero or more input symbols onto stack until a handle  $\beta$  is on top of stack
- Parser reduces  $\beta$  to left side of appropriate production.
- Parser repeats cycle until:
  - detect an error or stack
  - start symbol on top of stack

# Operator-Precedence Parsing

- Properties of grammar:

- No production right side is  $\in$  OR
- has two adjacent nonterminals

- CounterExample:

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Corrected:

$E \rightarrow E+E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid id$

- May effectively ignore grammar

- Use the nonterminals on stack only as placeholders for attributes associated with nonterminals

- Disadvantages:

- Hard to handle tokens that have 2 different precedences (-)
- Loose relationship between grammar and parser – can't always be sure parser accepts exactly the desired language.
- Only a small class of grammars can be parsed using operator-precedence parsers.

- Three disjoint **precedence relations**:

$\langle \cdot, \doteq, \cdot \rangle$

- Guide the selection of handles
- Table of Semantics:

Relation	Meaning
$a < \cdot b$	$a$ “yields precedence to” $b$
$a \doteq b$	$a$ “has same precedence as” $b$
$a \cdot > b$	$a$ “takes precedence over” $b$

- Two ways determining precedence relations between 2 terminals
  1. Intuition-based (i.e. \* greater than +)
  2. Construct an **unambiguous** grammar for language

# Using Operator-Precedence Relations

- Delimit the handle of right-sentential form,
  - $< \cdot$  marks the left
  - $\dot{=}$  appearing in the interior
  - $\cdot >$  marks the right
- Because no adjacent nonterms. appear on right sides of productions implies that no right-sentential form will have 2 adjacent nonterms.
- Write right-sentential form as
$$\beta_0 a_1 \beta_1 \dots a_n \beta_n,$$
  - where  $\beta_i$  is either  $\in$  or single nonterm.
  - each  $a_i$  is single terminal
- Suppose between  $a_i$  and  $a_{i+1}$  exactly one of  $(< \cdot, \dot{=}, \cdot >)$  holds.
- Use \$ to mark each end of string
- Define  $\$ < \cdot b$  and  $b \cdot > \$$  for all terminals  $b$ .
- Remove nonterms from string and place correct relation between each pair of terminals.

# LR Parsing

- Efficient, bottom-up syntax analysis
- Parse large class of context-free grammars
- LR(k):
  - “L” : left-to-right derivation scanning of input
  - “R” : rightmost derivation
  - (k) : number of lookahead symbols (k = 1, default)

## Why use?

- Can be constructed to recognize almost all programming language constructs (if can write context-free grammars)
- Most general **nonbacktracking** shift-reduce method (implemented as efficiently)
- Class of grammars parsed is a proper *superset* of class parsed with predictive parsers.
- Can detect syntactic error as early as possible in left-to-right scan.

## Drawback:

Lots of work to construct LR parser by hand for grammar.  
Need parser-generators (e.g. Yacc)

- LR Grammars:  
A left-to-right shift-reduce parser be able to recognize handles when they are on top of stack.
  
- Constructing SLR Parsing Tables:
  - Simple LR: simplest/weakest
  - Construct (from grammar) a deterministic finite automaton to recognize viable prefixes.
  - Group items together into sets  $\rightarrow$  states of SLR parser.  
The items correspond to states of an NFA.
  -
  
- Constructing LALR Parsing Tables
  - Lookahead-LR parsing
  - Tables smaller than canonical LR tables
  - Same number of states as SLR ( several hundred for Pascal)

# Semantics

## Two General Classes of Instructions

- Executable:
  - Some activity to occur at execution time
  - During compilation, source instruction has to be converted into machine language
  
- Non-Executable:
  - Information for the translator
  - Header information, type declarations, etc.
  - - “Used up” by compiler,
    - may change symbol table
    - Won’t produce machine language

# Executable Instructions

In order to work with executable instructions:

- Must have internal form to represent them
- Why not go directly to machine language? (or machine language)
  - Makes optimization much more difficult
  - Compiler then only good for one machine

## Use Intermediate Language

- Only need to write a code generator.  
takes intermediate lang. and produces appropriate machine-specific code.
- The program can be optimized
- What type of intermediate language?

Use 4-tuples :

operation operand1 operand2 result

or

result operation operand1 operand2

# Example

In our example:

```
< mult,sum,val,IR1 >  
< div,result,cost,IR2 >  
< add,IR1,IR2,IR3 >  
< store,IR3,,result >
```

What are we able to go back and forth between symbol table index and the variable name?

Name is stored!

Index is easy to use because it points out all info you have about that variable.

Name is better to use for outputting (printed or intermediate code).

# Expressions:

## Major purpose:

Evaluate the expression, or calculate single value that is the expression.

## Possible results:

integer value

real value

bit pattern

‘‘true’’

‘‘false’’

- Usually involves one (unary) or two (binary) operands and an operator.
- Have to know the **meaning** of the **operators**.
- Often, meaning is dependent on the **context**.  
(what type are the two operands?)
  - Simple case:  
both operands are of same type, meaning is defined
  - More Complex:
    - convert operand?
    - assume operand is more important? (use as is)
    - Decisions have to be cleared defined in order to write semantics
    - Give in **Language Definitions**

# Assignment:

## Generally:

$\text{var} \leftarrow \text{expr}$

- Semantic routine must first check if type of expr and type of variable agree.
- If so, do a store operation
- If not, is assignment permitted?
  - No: give error message
  - Yes: is conversion necessary?
    - No: store
    - Yes: convert, store

# If-Statement

Given a block IF, what do we want to occur at machine level?

```
IF (lexpr) THEN block1 ELSE block2 ENDIF
```

1. evaluate *lexpr*
2. if *lexpr* is false, skip over block1 code
3. < block1 code >
4. skip over block2 code
5. < block2 code >

What about in terms of 4-tuples? (generic machine)

## If-Statement continued

The order in which we encounter things is very important!

WRT example, must evaluate and test logical expr *prior* to working with the set of statements in block1.

Write a grammar to handle this:

IFSTAT  $\rightarrow$  IFHEAD IFTHEN IFELSE ENDIF

IFHEAD  $\rightarrow$  IF (LEXPR)

IFTHEN  $\rightarrow$  THEN ELBLOCK

IFELSE  $\rightarrow$  ELSE ELBLOCK

LEXPR  $\rightarrow$  ?

EBLOCK  $\rightarrow$  ?

# Looping Constructs

- **Top-tested** while loop
- **Bottom-tested** repeat loop
- **Counted** loop

## While Loop

Consider what you want in the machine code version:

```
loc1      code to
          evaluate
          expr

          branch to loc2 if lexpr is false

          loop
          body

loc2      branch to loc1
          continue
```

# Loops continued

As 4-tuples:

Notice that it is crucial that you recognize the top of the loop **before** evaluating the logical expression:

## Grammar

WSTAT → WHEAD WEXPR EBLOCK ENDWHILE

WHEAD → WHILE

WEXPR → LEXPR

# Optimization

- Object:  
Make program more efficient
- Efficient:  
Reduce execution time and memory used (often conflict!)
  - Memory: both code **and** data
  - Execution Time: Number of instructions **and** type of instructions

Many, many, many approaches to optimization – we'll look at general techniques.

## IMPORTANT:

Optimized code **must not** change the results!

# Optimization Continued

## Optimization Goals:

1. User program run-time reduced
2. User program memory reduced, if possible
3. No change to user program (logic or function)
4. Minimal increase in compile-time costs

## Optimization techniques

1. **Folding** (with constant propagation)  
do math at compile- time whenever possible
2. **Common subexpression removal:**  
eliminate unnecessary operations where possible
3. **Invariant code movement:**  
move operations to less-frequently executed places
4. **Operator strength reduction:**  
change math operators to less expensive ones ( $\times$  to  $+$ )
5. **Elimination of dead definitions:**  
delete unnecessary operations
6. **Test condition modification:**  
reorganize tests for efficiency (early exit)

# Types of Optimizations

## Machine-independent

- Based on the properties of program being translated
- Depends only on sequence of instructions
- How:
  - Analyze entire program
  - Find equivalent version of program that is more efficient
  - Examples:
    - Identify *null* and *identity* operations
    - Identify places where *commutativity* and *associativity* can be used to improve code.

## Machine-dependent:

- Make use of special features of hardware to produce more efficient code.
- How:
  - Compiler-writer must know alot about hardware,
  - Be able to recognize places to use the features
  - Examples:
    - increment instruction to replace tuples for  $A = A + 1$ .
    - parallelism
    - use of vectors

# General techniques to Improve Code

- Reduce number of memory operations
- Reduce number of comparisons
- Move invariant code outside of loops
- Eliminate unused code.
- Compute some values at compile time:  $1 + B + 6$
- Eliminate common subexpressions
- Unroll loops – straight line code is faster, but takes up more memory.
- Reduce the strength of operations (mult. to add)

# Program Topology

Abstract representation of flow of control in program

- Base blocks
- Strongly connected regions

Basic block:

- Linear sequence of intermediate code
- Have one entry and one exit (first and last instructions)

Represent program as flow graph

- Each node represent basic block
- Think about it in terms of intermediate language:
  - labels: start a new block
  - jumps: terminate a block
  - Subroutines call counted as a jump (registers destroyed by subroutine)

# Intra-block Optimization

## Two techniques that we'll study

- Folding with constant propagation
- Common subexpression elimination

## Folding with Constant Propagation:

- Folding: computing expression values at compile time instead of run time
- Constant Propagation: carry known values forward

## Simple Example:

$$\begin{aligned}A &= 4.0 \times 5.0 \\ B &= A + 1.0\end{aligned}$$

1. Compute  $4.0 \times 5.0$  at compile time
2. Convert first instruction into  $A = 20.0$ .
3. Add  $20.0 + 1.0$  at compile time
4. Convert second line into  $B = 21.0$ .

# Method for Constant Propagation

- Operate only on unsubscripted variables and intermediate results
- Restrict math operations to  $+$ ,  $-$ ,  $\times$ ,  $/$
- Maintain table of “variables” and known values
- For Math tuples:  
If both operands are constants or known from the table,  
do math operation and put the intermediate result in table with value
- Assign tuples:
  - If opnd is a constant or know from table, update table
  - If opnd unknown, delete an entry from table if present
  - Regardless, output tuple (maybe iwth constant instead of operand)

# Example

```
input :    < I$001    IADD    K$0001    K$0010  >
          < A      ASSIGN  I$001    -----  >
          < B      ASSIGN  A        -----  >
          < I$002    IMUL   C        A        >
          < B      ASSIGN  I$002    -----  >
```

```
table:    1) I$001    11
          2) A        11
          3) B        11
```

```
output:   <A      ASSIGN  K$0011    -----  >
          <B      ASSIGN  K$0011    -----  >
          < I$002    IMUL   C        K$0011  >
          < B      ASSIGN  I$002    -----  >
```

Corresponding Source:

$$\begin{aligned} A &= A + 10 \\ B &= A \\ B &= C \times A \end{aligned}$$

# Summary

- Folding with constant propagation will operate only **within** a basic block
- Operates only on variables are not arrays, and on intermediate results.
- Different processing needed for short and long integer constants
- Table cleared at end of each block

## Maintaining Table

- **Add** an entry when there is a first defn of variable and value is known
- **Modify** an entry when variable already appears in table and new value is known
- **Delete** an entry for a user variable if a defn occurs, but value is not known.

## Why only within blocks?

```
        I = 0
10      I = I + 1
        .
        .
        .
        IF (I .LT. 10) GOTO 10
```

If ignore block boundaries, get strange code

```
        I = 0
10      I = I + 1
        .
        .
        .
        IF (1 .LT. 10) GOTO 10
```

### Why only unsubscripted variables?

- Usually, all array subscripts are variables themselves
- Very seldom do you know which element is being used
- Too much overhead and too little reward

### Example:

A(I) = 0

.

.

.

B = 2 \* A(J)

- Very rarely, would you know both *I* and *J* values
- Not gain much by keeping track of array elements
- Worse, If you **don't** know *I*, **all** separate A(something) definitions you have would have to be deleted.

# Common Subexpression Removal

- Some instructions are redundant
- Should be eliminated to make program more efficient
- Less memory and less time

## Which are redundant?

- Those operations which have already been done before
  - Identical operations – same operands, same action
  - To reach second you, always have to go through first operation. (branch point)
- Definition allows us to call something redundant even if not in same block

## How do redundant instructions arise?

- User ignorance
- Natural expression involves redundancy
- Expansion of subscripts
- Movement due to optimization

### Examples:

Natural Expression:

$$\begin{aligned} X &= (-B + \text{SQRT}(B ** 2 - 4 * A * C)) / (2 * A) \\ Y &= (-B - \text{SQRT}(B ** 2 - 4 * A * C)) / (2 * A) \end{aligned}$$

Optimal:

$$\begin{aligned} D &= 2 * A \\ F &= \text{SQRT}(—) \\ X &= (-B + F) / D \\ Y &= (-B - F) / D \end{aligned}$$

# Data Dependencies

- Find inter-block dependencies using **dependency tree**
- Tree:
  - Nodes represent results of operation
  - Leaf nodes must represent fetches, since there is no previous result
  - Assign level numbers based on length of path from leaf nodes  
leaf nodes = level 0.

## Example

$$X = (A * B) + (C * D)$$

$$Y = X + (C * D)$$

$$Z = E + F$$

$$X = 5$$

Tuples:

< I\$001	IMUL	A	B	>
< I\$002	IMUL	C	D	>
< I\$003	IADD	I\$001	I\$002	>
< X	ASSIGN	I\$003	----	>
< I\$004	IMUL	C	D	>
< I\$005	IADD	X	I\$004	>
< Y	ASSIGN	I\$005	----	>
< I\$006	IADD	E	F	>
< Z	ASSIGN	I\$006	----	>
< X	ASSIGN	K\$005	----	>

Tree:

### Determining Level numbers:

- Level number of a result is equal to  $1 + \max$  of its operands
- Level number of a user variable must be greater than any previous uses or definitions of that variable

You can eliminate the second of two instructions which are identical and have the same level numbers

### Procedure:

- Make sure operands are ordered (either in parser or now)
- Assign level numbers
- Sort instructions  
sort on level numbers (topological sort)
- Eliminate redundant ones

### Notice:

- constants have level 0
- variables level 0 initially,  
then have level  $n$ , where  $n$  is greater than every previous use of definition
- first instruction in block must have level 1 for its result
- instruction with one or more IRs as operands must be at level 2 or greater
- redundant instructions will have:
  - same operator
  - same operand
  - $\langle$  same operand level numbers  $\rangle$
  - same result level number