Programming Language Concepts (CS383) Final Review Manuscripts

Hongjian Wang Shanghai Jiao Tong University
sishenhadesi@163.com

***Abstract*—The Programming Language (CS383) course introduces the concepts that serve as a basis for programming languages. It aims to provide the students with a basic understanding and appreciation of the various essential programming language constructs, programming paradigms, evaluation criteria and language implementation issues. The course covers concepts from imperative, object-oriented, functional, logic programming and scripting languages. These concepts are illustrated by examples from varieties of languages such as Pascal, C, Java, Smalltalk, Scheme, ML, Haskell, Prolog, Perl etc. As the final examination is coming, I make notes on all the topics has been given and bring out this review manuscripts. As this manuscripts is finished, my understanding on this course is definitely deepened.**

***Index Terms*—Programming Language, Principles, Paradigms, evaluation criteria**

I. Overview, Inductive Definition

A. Overview

1) Principles: Four important properties for both language designers and programmers.

- Syntax: precise description of all grammatically correct programs
- Names: name entities, bound in running program to scope visibility type lifetime
- Types: a collection of values and a collection of operations on them
- Semantics: meaning of a program. Several categories on this: operational/static/dynamic ˜ .

2) Paradigms: A pattern of problem-solving thought that underlies a particular genre of programs and languages.

- Imperative: follows the Von Neumann-Eckert model (data and instruction share memory), a sequence of commands. Ex: Cobol, Fortan, Ada, Perl
- Object-oriented: collection of objects interact by passing messages. Core: Inheritance, Polymorphism. Ex: Smalltalk, Java, C++, C#, python
- Functional: models a computation as a collection of mathematics functions. Core: functional composition, recursion, no state change. Ex: Lisp, scheme, haskell, ML
- Logic: declares what outcome should accomplish (declarative). Programs are constraints, achieve all solution, nondeterministic. Ex: Prolog, CLP

I have attended the Programming Language Concepts course of Prof. Kenny Q. Zhu in the fall semester of 2012. The teacher assistant of this course is Mr. Tianwan Zhao.

3) Sepcial Topics:

- Concurrency
- Event handling (optional)
- Correctness (optional)

4) A brief history: How and when PL evolve, which communities have developed and used them? AI, CS education, information systems, networks, WWW.

5) On Language Design: Design constraints: computer architecture, technical setting, standards, legacy systems.

Successful PL characteristics:

- simplicity & readability: small instruction set, simple syntax
- clarity & binding: binding is association between object and property (early/late binding)
- reliability: behaviour consistent on different platform, type error checking, memory leaks prevented
- support: accessible (public domain), good tutorials, wide community users, better IDE
- abstraction: user-defined type, procedural
- orthogonality: primitive operations are small and mutually independent efficient. A tradeoff with efficiency
- implementation: different scenarios has differrent requirement: embedded systems, web applications, database systems, AI applications.

6) Compilers and Virtual Machines: compiler, interpreter, hybrid compilation/interpretation

B. Inductive Definition

1) Judgement: A judgement is an assertion about a syntactic object, which states one or more syntactic objects have a property or have a relation among one another. The property or relation itself is called *judgement form*, which can be instaniated as *instance* of judgement form.

2) Inference Rules: An inductive definition of a judgement consists of a collection of rules with following form:

$$\frac{J_1, \ldots, J_k}{J}$$

$J_1, \ldots, J_k$ is called premises, and J is called conclusion. A rule without premises is called an *axiom*, otherwise *proper rule*.

3) Inductive Definition: Eg: definition of judgement *t tree*.

$$\frac{}{empty\ tree} \qquad \frac{t_1\ tree \quad t_2\ tree}{node(t_1;\ t_2)\ tree}$$

4) Derivation: To show an inductively defined judgement holds. A derivation is an evidence of the validity of the defined judgement. Usually, it's the finite composition of rules starting from axioms and ending at at that judgement. And it shapes as a tree structure, like the *parse tree*.

Eg: Derivation of $node(node(empty, empty), empty)$ *tree*).

Two types: forward chaining (bottom-up); backward chaining (top-down). A deductive systems has two parts: definition of a judgement form; a collection of inference rules.

5) Rule Induction: Reason about rules under an inductive definition (or within a deductive system).

Principle of rule induction: To show property P holds for a judgement J, it is enough to show that P is closed under or respects the rules defining J. Eg: if $P(J)$ holds whenever $P(J_1), \ldots, P(J_k)$ hold; the former is inductive conclusion and the latter are inductive hypothesis.

It's a generalized version of mathematical induction, and step 1 is basis, step 2 is induction step.

II. Proof by induction

A. Proof Principles

Given

$$\frac{J_1, \ldots, J_n}{J}\,[name]$$

if $J_1, \ldots, J_n$ have property P, then so does J.

B. Natural numbers

Induction on the structure of natural numbers. Prove problems: (1) addtion, (2) even/odd number

C. List

Judgement form of **l list**:

$$\frac{}{nil\ list}nil \quad \frac{n\ nat \quad l\ list}{cons(n, l)\ list}cons$$

Problems to be proved: cons((S Z), cons(Z, nil)) is a list; list-len; list-append; list-reverse; [lemma]if *len l n*, and *append l* n_1 *l'*, then *len l'* (*S n*).

D. Proof structure

Theorem: If X then A
Proof: By induction on the structure of derivation of J (assuming with rules: Foo-1, Foo-2, Bar)

Case Foo-1: 1. ...
2. X (by assumption)
3. ...
n. A (...)

Case Foo-2: similar to Foo-1
Case Bar: similar as above

Rules to prove by

Clearly state Induction Hypothesis and the proof methodology. There should be one case for each rule. Two column format with logical steps left and reasoning right. Number the steps for reference. Always state where you use the I.H. Omit similar proof. Break down a proof into appropriate lemmas. Define the judgement first clearly, before using it.

III. Syntax

The *syntax* of a PL is a precise description of all its grammatically correct programs. There are three levels:

- lexical syntax: all the basic symbols of the PL
- concrete syntax: rules for writing programs (statements and expressions)
- abstract syntax: internal representation of the program, favoring content over form.

A. Grammers

A *metalanguage* is a language used to define other languages. A *grammer* is a metalanguage used to define the syntax of a language. Our interest: using grammers to define the syntax of a PL.

1) Backus-Naur Form (BNF): It's stylized version of a context-free grammer, also called Backus Normal Form, which is used to define most major language syntax. BNF consists a set of *productions P, terminal symbols T, nonterminal symbols N, start symbol* $S \in N$. P has the form $A \to \omega$, where $A \in N$ and $\omega \in (N \cup T)^*$.

2) Derivations: The grammer is actually also inductive definition. There are two kinds of derivation: leftmost and rightmost derivation. Notations for derivations:

- $\Rightarrow^*$ means a finite number of steps of derivation.
- $3234 \in L(G)$ means 3234 is a member of language defined by grammar G.
- $L(G) = \{\omega \in T^* | Integer \Rightarrow^* \omega\}$ means L(G) is the set of all strings ω that can be derived as Integer.

3) Parse Trees: Parse tree is a graphical representation of a derivation. Internal node $\longrightarrow$ a step; child node $\longrightarrow$ right-hand side of a production; leaf node $\longrightarrow$ a symbol of the derived string (L to R). Eg: Integer, Arithmetic Expression grammar.

4) Associativity and Precedence: A grammar can be used to define these two concepts among operators. Eg: Arithmetic Expression with associativity and precedence:

$$\begin{aligned} Expr &\to Expr + Term | Expr - Term | Term \\ Term &\to Term * Factor | Term / Factor | Term\%Factor | Factor \\ Factor &\to Primary * * Factor | Primary \\ Primary &\to 0 | \ldots | 9 | (expr) \end{aligned}$$

The associativity and precedence relationships are shown by the structure of the parse tree: highest precedence at the bottom, and the left-associativity recurses on the left at each level.

5) Ambiguous Grammars: A grammar is *ambiguous* if one of its strings has two or more dfferent parse trees. Instead of using a large grammer, such as C/C++, Java, we can write a smaller ambiguous grammer, and give seperate precedence and associativity. Eg—Dangling else problem: solution differs. In Algol 60, C/C++, the **else** associate with closest **if**; in Algol 68, Modula, Ada use explicit delimiter to end every condition; in Java rewrite the grammer to limit what can appear in a conditional statement.

B. Extended BNF

In BNF, use recursion for iteration; use non-terminals for grouping. In EBNF, use additional meta-characters: {} for series of zero or more; [] for an optional list, picking one or none; () for a list, must pick one.

We can always rewrite an EBNF as a BNF. While EBNF is no more powerful than BNF, its rules are often simpler and clearer.

C. Syntax of Clite

1) Lexical Syntax: Input is a stream of ASCII chars; output is a stream of tokens or basic symbols, such as *Identifiers, Literals, Keywords, Operators, Punctuation*.

1) Whitespace: any space, tab, end line, character inside comment. No token contains whitespace.
2) Comments not defined in grammer
3) Identifier sequence of letter and digits (mostly should be distinct from keywords). Redefining identifiers can be dangerous. Old language generally has case-insensitive identifiers, for they are less complicated.

2) Concrete Syntax: It's based on a parse of its tokens, the **Ifstatement** is ambiguous, and thus need addtional associativity rule.

Note: $<, >, <=, >=, ==, ! =$ are non-associative. In C series PL, they are associative, which is a questionable design. In this case, $2 < 3 < 2$ is true, because $2 < 3$ evaluates to 1 first, then compare 1 with 2.

D. Compiler and Interpreter

Compiler consists of

1) lexer: from ASCII characters to Tokens
2) parser: based on BNF, from Tokens to abstract syntax tree
3) semantic analysis: check identifiers are declared, type checking, insert implied conversion operators
4) code optimization: evaluate constant at compile time, reorder code, eliminate common subexpressions and unnecessary code.
5) code generation: output machine code. doing instruction selection, register management, and peephole optimization.

Interpreter replaces last 2 phases of a compiler. Input can be mixed (intermediate code) such as Java, Perl, Python, Haskell, Scheme; or pure (ASCII characters) most Basics, shell commands.

E. Linking Syntax and Semantics

The shape of a parse tree reveals the meaning of the program. However, output a parse tree is inefficient. So we want better tree, which removes the separator/terminal symbols, trivial root nonterminals, and replace remaining nonterminals with leaf terminals. In a word, it removes "syntactic sugar" and keeps only essential elements of a language.

IV. Names

A. Names and Denotable objects

Name is a sequence of characters used to represent or denote an object. Name is not that object (entity); name is just a "character string" and can denote different objects at different times, while the object can have mulitple names — "aliasing". **Syntactic issues for names**

- lexical rules
- reserved words and keywords: can't be used as identifiers. Keywords of major construct (if, while); predefined routines.
- case sensitivity

Purpose of names: provide data abstraction (memory addresses abstraction) and control abstraction (procedure name). Denotable objects are objects whose names are defined by user, or by the programming language.

1) Binding: It is an association between a name and the denotable object it represents. Binding can happen in design of language, programming writing, compile time, run time. There are two kinds of binding: static binding and dynamic binding. The *lifetime* of a name refers to the time interval during which the name remains bound.

Variables in C-like language: L-value denote a variable's address; R-value denote a variable's value.

B. Environment and Blocks

The set of association between names and denotable objects which exist at runtime at a specific point in the program and at a specific time during execution, is called *envrionment*, also known as "symbol table".

A block is a textual region of the program, identified by a start sign and end sign, which can contain declaration local to that region. Blocks can be nested but not overlapped. The block in which a name is defined or declared is called its defining block.

Visibility A name is *visible* in the block where it's declared and all the blocks nested inside, unless the same name is redeclared inside a inner block. A redeclared name in an inner block effectively hide the outer declaratoin. Mechanism to access hidden names is provided in some lanuages (C++/Java), using **this**.

Type of environments for a block: local environment, non-local environment, global environment. Operations on environment: entering, exit.

C. Scope

The **scope** of a name is the collection of statements which can access the name binding. In statics scoping (lexical scoping), which is most modern languages using, a name is bound to a collection of statements according to its position in the source program. In dynamic scoping, the valid association for a name X, at any point P, is the most recent association created for X which is still active when control flow arrives at P.

Symbol tables (aka. environments) is a data structure kept by a translator that allows it to keep track of each declared name and its binding. Assume that name is unique within its local environment. The data structure can be any implementation of a dictionary, where the name is the key.

Handling non-local references. Each time a block is entered, push its dictionary onto the stack; while exit, pop the dictionary off the stack. For each name declared, generate an appropriate binding and enter the name-binding into the dictionary on the top of stack. Given a name reference, searching the dictionary on stack top first: found, return; otherwise repeat on the next dictionary down in the stack; if can't find, report error.

D. Static Scoping

For static scoping, the scope for a name is its defining block and all nested blocks. Pro: better understanding by programmer, compiler perform correctness checks. Con: compiler don't know which name is bound to which address.

E. Dynamic Scoping

In dynamic scoping, a name is bound to its most recent declaration based on the program's call history. Used by early Lisp, APL, Snobol, Perl. Symbol table for each scope built at compile time, but managed at run time.

Disadvantages: compromises the ability to statically type check reference to non-local variable. Variables in function visible to other functinos that call this one (less reliable). Access to non-local variable follows chains of dynamic links (more time-consuming).

F. Problems with static scopes

Different interpretations of static scope rules depending on where made declaration, what is the visibility of local variables. Eg: PASCAL additional rule, declaration only at start of block, scope extends from beginning to end of a block independent of declare position, names be declared before use (this is problematic, it forbid recursive type).

G. Overloading

Overloading uses the number or type of parameters to distinguish among identical function names or operators.

H. Lifetime

Lifetime of a variable is the time interval during which the varible has been allocated a block of memory. Earliest languages used static allocation at compile time. Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit. Modern languages are based on this idea but may break *scope equals lifetime* rule. Eg: lifetime beyond scope. Pascal: single compilation unit, variable keeps its value. C: multiple compilation unit, global compilation scope static.

V. Types

A. Basics

A *type* is a collection of values and operations on those values. Computer types have a finite number of values due to fixed size allocation. Exception: smalltalk has unbounded fractions, haskell has unbounded integers. Purpose of types in PL is to provide ways of effectively modeling a problem solution.

B. Type Errors

A type error is any error that arises because an operationis attempted on a data type for which it is undefined. They are common in assembly language. High level languages reduce the number of type errors by providing a type system as the basis for detecting type errors.

C. Static and Dynamic Typing

A type system imposes constraints can't be expressed syntactically in EBNF. **Statically typed** compile time fixed type: C/C++, Java, ML, Haskell. **Dynamically typed** run time fiexed type: Perl, Python, JS, prolog, scheme. **Strongly typed** its type system allows all type errors in a program to be deteced at either time. Strongly T can be either Dynamic T or Statially T. Eg: *Union type* the hole in most type system.

D. Basic Types

Finite size in types is problematic: the overflow cause $a + (b + c) \neq (a + b) + c$. Type conversion is implicit or explicit change of type of a value to a different one. Generally, implicit conversion (type coercion) is widening conversion; explicit conversion (type casting) is narrowing.

Characters set: older languages use 7-bit ASCII encoding, like (C). Java, etc use Unicode.

E. NonBasic Types

Eg: Enumeration, Pointers, Linkded List, array and list, strings, structures (analogous to a tuple in mathematics, usded first in Cobol, PL, common to C-like language, omitted from java), union (C logically multiple views of same storage).

Pitfalls of Pointers: bane of reliable software development; error-prone; buffer overflow, memory leaks.

Other languages such as Java, Haskell, ML and Prolog completely remove pointers from vocabulary of the language.

Equivalence between arrays and pointers: if either e1 or e2 is type **ref T** e1[e2] = *(e1 + e2).

F. Recursion Data Types

Mainly in Haskell.

G. Functions as Types

They are first-class citizens in PL, can be assigned a value and passed as an argument to a function.

H. Type Equivalence

Nowhere does the Pascal Report define *identical type*. Two types of equivalence is defined: name equivalence and structrual equivalence.

I. Subtypes

A subtype is a type that has certain constraints placed on its values or operations. Eg: Ada subtypes [subtype one_to_ten is Integer range 1..10]; Java implements subtypes using class hierarchy.

J. Polymorphism and Generics

A function is *polymorphic* if it can be applied to any one of several related types and achieve the same result. **It enables code reuse.** Two types of polymorphism:

1) Ad-hoc polymorphism: one function applies to arguments of different types — multiple implementation of the same function. also known as **overloading**. Java overloaded methods with different number or type of parameters.
2) Parametric polymorphism: functions or data types written genericaly so they can handle values identicaly without regard to their types. also known as **generics**. type binding delayed from code implementation to compile time. Eg: Ada.

K. Programmer-Defined Types

Defer further until OOP.

VI. Type Systems

A. A case study

1) Type checking: The detection of type errors, either at compile time or run time is called type checking. Type error can't be prevented by EBNF. Type system identify type errors before they occur.

2) Design a Type System: A set of rules V in highly-stylized English: return True or False, based on abstract syntax, is a mathematical function. Facilitates type checking. Implementation throws an exception if invalid.

Type Rule

1) *All referenced variables must be declared.* Type map is a set of ordered pairs, which can be implemented as a hash table.
2) *All declared variables must have unique names.*
3) *A program is valid if its declarations are valid and its block body is valid with respect to the type map.*
4) *Validity of a statement: a skip is always valid; an assignment is valid if: target variables declared, source expression valid, target variable have the same type as source expression, or their type match with proper implicit type casting; a conditional is valid if: test expression valid with type bool, thenbranch and elsebranch statements valid; a loop is valid if: test expression valid with type bool, statement body is valid; a block is valid if its statement are valid.*
5) *Validity of an Expression: a value is always valid; variable is valid if in the type map; a binary is valid if: expressin term1 and term2 valid, arithmetic operator both exp are numeric, op is relational both exp have same type, op is && or —— both exp are bool.*
6) *The type of an Expression e is: if e is value, the value type; if e variable, the variable type; e is binary, the type of result; e is unary op, bool.*

3) Implicit Type Conversion: Implicit widening conversions implementeb by transform the abstract syntax tree.

B. Formalizing Type Systems

The type system of a language consists of a set of inductive definitions with judgement form:

$$G \vdash e : t$$

G is the typing context, which is a set of hypothesis of the form $x : t$. x is the variable name, t is the type of x. The form known as *hypothetical judgment*.

Properties of $L\{NUM - STR\}$:

Lemma 1 (Unicity of Typing) for every typing context G and expression e, there exists at most one t such that $G \vdash e : t$.

Lemma 2 (Inversion for Typing) Supposing $G \vdash e : t$. If $e = plus(e_1; e_2)$, then $t = num$, $G \vdash e_1 : num$, and $G \vdash e_2 : num$.

1) Typing Inference: Criticism of Typed Languages: (1) type overly constrain functions and data.Polymorphism makes typed constructs useful in more contexts; (2) types cluster programs and slow down programmer productivity. Type inference automatically deduct type of an expression in a PL, which is a type reconstruction if some but not all type annotations exist. Type inference available in Haskell, ML, O'caml, VB, C#, Perl 6, etc.

Steps in type inference:

1) Add type schemes: type scheme contains type variables that may be filled in during type inference.

2) Generate constraints: walk over the program, keep track of type equatinos according to the normal typing rules.
3) Solve Constraints: a solution to a system of type constraints is a substitution S, a function from type variables to types. Substitution can be composed and generate a specific one. S is the **principal** (most general) solution of a constraint q: if $S \models q$, if $T \models q$ then $T \leq S$. **Unification**: an algorithm provides the principal solution to a set of constraints.
4) Generate Types: Apply S to e. S is principal, S(e) characterizes all reconstructions.

2) Type safety: Type safety = preservation + progress
A language is type safe if it satisfies the following:

- Type Safety: If $e : t$ and $e| \rightarrow e_0$ then $e_0 : t$. If $e : t$, then either e is value, or there exists e_0 such that $e| \rightarrow e_0$.
- Preservation: If $e : t$ and $e| \rightarrow e_0$, then $e_0 : t$.
- Progress: If $e : t$, then either e is value, or there exists e_0 such that $e| \rightarrow e_0$.

We prove these after Lamabda calculus.

VII. Semantics

A. Motivation

To provide an authoritative definition of the meaning of all language constructs for: programmer, compiler writer, standards developer. A PL is complete only when its syntax, type system, and semantics are well-defined. Semantics is a precise definition of the meaning of a syntactically and type-wise correct program. Ideas of meaning: operational semantics, axiomatic semantics, denotational semantics.

B. Expression Semantics

- Operators + associativity + precedence: Eg. infix uses associativity and precedence to disambiguate.
- Evaluation orders: Eg. short circuit evaluation (also known as *lazy evaluation*, disadvantage is commutative law is broken.
- Pure vs. Impure: a change to any non-local variable or I/O is called side effect, which makes a language impure and should be avoided if possible.

C. Program State

The state of a program is the bindings of all active variables and their current values. ENVIRONMENT → pairing of active variable name and memory locations. MEMORY → pairing of active memory locations and values.

$$State = environment \times memory$$

The current statement to be executed in a program is interpreted relative to the current state. The individual steps that occur during a program run can be viewed as a series of state transformations.

D. Assignment Semantics

Semantics of assignment: evaluate the source expression, replace the value of target variable. It is fundamental to imperative and object-oriented programming. Issues:

- multiple assignment: ambiguity between assignment operator and equality operation.
- assignment statement vs. expression: In most imperative languages, assignment is a statement; can't appear in an expression. In c-like language, assignment is an expression. In java, all conditions must have boolean type. In functional languages, the whole program is an expression.
- copy vs. reference semantics: $a = b$ make a, b have same value (refer to same object). Copy generate independent object, reference generate connected object.

E. Control Flow Semantics

To be complete, an imperaive language needs:

- statement sequencing: in the absence of a branch
- conditional statement: ifstatement
- looping statement: whilestatement

Goto controversy — Dijkstra's letter "GOTO considered Harmful". Eg: Ada and java excludes "goto" from their vocabulary.

F. Input/Output Semantics

The sources and destinations of input and output are called files: keyboard, display monitor, disk, memory. Binding: open, close. Access: sequential vs. random. Stream vs. fixed length records. Character vs. binary. Format.

G. Exception Handling Semantics

Purpose of exception handling: simplify programming, make application more robust (application continue to perate under all conceivable error situations). An exception is an error condition occurring from an operation that cannot be resolved by the operation itself.

Two important questions in exception handling: how is handler associated with an exception? should handler resume executing the code that throws exception?

C++ exception handling — try ...throw ...catch ... Java exception type hierarchy.

Eg: using assert. In java there used to be assert class, and now it is an keyword (or something like that).

VIII. Semantic Interpretation

A. State Transformation and Partial Functions

Give definition of denotational semantics, semantic domain, partial functions. Historic problem, valid program had different meanings on different machines, because of lack of precision in defining meaning.

B. Semantics of Clite

State — represents the set of all program states. A meaning function M is a mapping. The meaning of a program is set of Ms. *I*: interpretation.

Meaning Rule

1) whole program: meaning of body with initial state.
2) skip: an identity function on the state
3) assignment
4) conditional
5) loop
6) block: a sequence of statements
7) expression

Expression with side effects, eg: $f(x)+x$'s result is non-deterministic depending on the order of evaluating "+". Solution to side effects: ban them (not practical); a new semantic function $M : expression \times state \rightarrow value \times \underline{state}$.

C. Semantics with Dynamic Typing

Scripting: Perl, Python, PHP
Object-oriented: Smalltalk, Ruby
Functional: Scheme, ML, Haskell
Logic: Prolog

Perl vs. Python Perl use implicit conversion with distinct operators, python use explicit conversions.

D. Formal treatment of semantics

Overriding Union $X\overline{\cup}Y$ is the result of replacing in X all pairs $\langle x, v\rangle$ whose first member matches a pair $\langle x, w\rangle$ in Y by $\langle x, w\rangle$ and then adding to X any remaining paires in Y. Formally

$$X\overline{\cup}Y = (X - (X \otimes Y)) \cup Y.$$

With these formal definition, we can have (refer to textbook 217)

$$\begin{aligned} M(Statement\ s, State\ state) = & \\ & M((Skip)\ s, state) \\ & M((Assignment)\ s, state) \\ & \ldots \end{aligned}$$

Limitations of above semantics:

- difficult to apply to transfer of control, throw and catch, functoin calls, object creation
- Transfer of control is handled by *continuation* with program instruction counter.
- the functional style of semantic interpretation gives deterministic outcome, not applicable to concurrent programs

IX. Functions

A. Basic Terminology

Value–returning functions and Non–value–returning functions.

B. Parameters

An **argument** is an expression that appears in a function call. A **parameter** is an identifier that appears in a function declaration. Parameter–Argument matching usually by number and by position (exceptions: perl — parameters aren't declared in function header, they are available in an array; Ada, python — parameters and arguments can be linked by name).

C. Parameter Passing Mechanisms

1) By Value: compute the value of the argument at the time of call and assign that value to the parameter. can't modify argument in function.
2) By reference: computer the address of the argument at the time of call and assign it to the parameter.
3) By value-result: pass by value at the time of call and copy the result back to the argument at the end of call. Ada's in out parameter is value-result.
4) By result: copy the result back to the argument at the end of the call. reference and value–result are the same, except when aliasing occurs. That is, when the same variable is both passed and globally referenced from the called function, or the same variable is passed for two different parameters.
5) By name: textually substitute the argument for every instance of its corresponding parameter in the function body, originated with Algol 60, droped by Algol's successors — Pascal, Ada, Modula. Exemplifies **late binding**, associated with **lazy evaluation** in functional languages.

D. Activation Records

A block of information is associated with each function call, which includes: parameters and local variables, return address, temporary variables, return value, stack link, dynamic link. Also known as *Stack Frame*.

E. Recursive Functions

A function that can call itself, either directly or indirectly.

F. Run Time Stack

A stack of activation records. Each new call pushes an activation record, and each completing call pops the topmost one. So, the topmost record is the most recent call, and the stack has all active calls at any run-time moment.

G. Function Declaration and Call in Clite

Concrete Syntax

$$\begin{aligned} Program &\rightarrow \{Type\ Identifier\ FunctionOrGlobal\} \\ FunctionOrGlobal &\rightarrow (Parameters\{Declarations\ Statements\}|Global \\ Parameters &\rightarrow [Parameter\{, Parameter\}] \\ Global &\rightarrow \{, Identifier\}; \\ & \ldots \end{aligned}$$

Abstract Syntax The syntax tree.

H. Completing the Clite Type System

Additional type rule

1) Every function and global id must unique.
2) Every function's parameters and locals must have mutually unique ids.
3) Every statement in the body of each functions must be valid with respect to the function's locals, parameters, and visible globals.
4) A non-void function must have a return statement, whose expression must be the same type as the function.
5) A void function cannot have a Return.
6) Every Call Statement must identify a void function, and every Call Expression must identify a non-void function.
7) Every Call must have the same number of arguments as the number of parameters in the function it identifies. Each argument must have the same type as its corresponding parameter, reading from left to right.
8) Every Call to a non-void function has the type of that function. The Expression in which the Call appears must be valid according to Clite Type Rule.

Additional meaning rule

1) The meaning of a Call c to Function f has the following steps: make an activation record and add f's parameters and locals to it, evaluate c's args and assign those values to f's corresponding params, add a result variable identical with f's name and type if f is non-void, push the activation record onto the run-time stack, interpret f's body, pop activation record from the stack, if f is non-void return the value of the result variable to the Expression where c appears.
2) The meaning of a Return is the result of assigning value of the Expression to the result variable.
3) The meaning of Block is the aggregated meaning of its statement when applied to the current state, up to and including the point where the first Return is encountered.

I. Formal Treatment of Types and Semantics

Refer to Page 35 of 11_functions.pdf

Function *typing* creates type maps for each individual function f in a program with globals G and functions F.

Given Memory(μ), Environment(σ), and State(σ), we have $\sigma_f = \gamma_f \times \mu \times \alpha$, where α addresses the top of the run–time stack.

X. Memory Management

Memory management is the process of binding values to memory locations. A process is a program in execution. All memory used by a process must reside in the process's address space.

A. Memory Organization

Major areas of memory: static area (storage requirements known advance and remain constant, allocated at compile time); run–time stack (local variables that get allocated each time a function is called, center of control for function call and return, a function's stack frame exists as long as the function is active); heap (dynamically allocated objects and data structures, the least organized and most dynamic storage area, easily fragmented, C/C++ leave managing heap manul work, Java/Perl do heap management).

Structure of runtime memory: static begins at the low address, heap begins at the high address, and they grow to each other. When met, then memory leakage happens.

Definition of *Stack overflow, stack underflow, heap overflow*.

B. Implementation of Dynamic Arrays

The procedure to generate a dynamic arrays is: creates heap blocks, create the address and length recored in stack.

The procedure to make array reference: just calculate the address of target reference, check whether it is out of range, return the value.

The procedure of array assignments: calculate address, check validity, assign value.

C. Garbage Collection

Garbage is a block of heap memory that can not be assessed by the program. It occurs when either an allocated block of heap memory has no reference to it, or a reference exists to a block of memory that is no longer allocated.

Why do GC? (1) programs consume storage freely, (2) mismanaging causes heavy bad consequence, (3) explicit memory management breaks the high-level programming abstraction.

GC is not a language feature, but a pragmatic concern for automatic and efficient heap management. Cooperative languages: lisp, scheme, prolog, smalltalk, uncooperative languages: c/c++ (with third–party GC libs). Recent GC in OOL modula-3 and Java; in functional language ML and Haskell.

The standards for a perfect GC: no visible impact on the program execution,works with any program and its data structure, collects garbage cells quickly, has excellent spatial locality of reference, manage the heap efficiently.

GC algorithms A strategy that reclaims unused heap blocks for later use by the program.

1) Reference counting: Occurs whenever a heap block is allocated, but doesn't detect all garbage.

Heap is a chain of nodes (free_list), each node has a reference count. But not all garbage is collected. If fails to return to the $free_list$ any garbage that occurs in the form of an isolated circular chain.

Advantages: dynamically occurs, the overhead is spread over time, easy to implement, can coexist with manual memory management, spatial locality of reference is good, can re-use freed cells immediately. Disadvantages: failure to detect inaccessible circular structure hence is imcomplete, space overhead by appending RC, performance overhead created by the book-keeping done during pointer assignment or when a heap block is allocated/deallocated.

2) Mark–and–swap: Occurs only on heap overflow, detects all garbage, but makes two passes on the heap.

Each node in the $free_list$ has a mark bit (MB) initially 0. Called only when the heap overflow occurs: pass I mark all nodes that are accessible from the stack by setting their MB=1; pass II sweep through the entire heap and return all unmarked nodes to the $free_list$.

Pros: handles cycles correctly, very little space overhead. Cons: normal execution must be suspended, may touch all virtual memory pages, heap may fragment.

3) Copy collection: Faster than mark-sweep, but reduces the size of the heap space.

Heap partitioned into two halves; only one is active. When heap overflow occurs, the accessible nodes are packed, orphans are returned to the $free_list$ and the two halves reverse roles.

Cheney's Algorithm

Pros: very low cell allocation overhead, compacting. Cons: twice the memeory footprint

4) GC summary: Modern algorithms are more elaborate, most are hybrids/refinements of the above three. Eg, generational garbage collection, nodes die young, divide the heap into generations, and GC younger generations more often, doesn't reclaim all free space – may need mark & sweep or copy collection occasionally. Java/.NET gc a few recent generations only. In Java and functional languages GC are built–in. C/C++ default gc to the programmer.

XI. Lambda Calculus

A. Overview of functions and Lambda Calculus

Originally, Alonzo Church proposed the lambda calculus as a logic in 1932. In mid-1960's, Peter Landin formulated complex language into a tiny core calculus — lambda calculus.

A calculus is any system of calculation, including basic values, types, legal operations, axioms, etc. Other calculus: Pi-calculus (message passing concurrent languages), Object-calculus (OOL).

Essentially every full-scale PL has some notino of function, the lambda calculus is a language composed entirely of functions.

B. Untyped(pure) Lambda Calculus

Syntax:
e is a lambda expression, or lambda term.

$$\begin{aligned} e ::= \; & x && \text{(a variable)} \\ | \; & \backslash x.e && \text{(a function)} \\ | \; & e \; e && \text{(function application)} \end{aligned}$$

Note: "\" will be written "λ" in a nice font, the above is inductive definition. Expressions which are functions are also called lambda abstractions.

Two notation conventions: applications associate to the left, the body of a lambda extends as far as possible to the right.

Some terminology: the scope of x in term e; free variable; variable is bound.

Free variables

$$\begin{aligned} free(x) &= x \\ free(e_1 \; e_2) &= free(e_1) \cup free(e_2) \\ free(\backslash x.e) &= free(e) - \{x\} \end{aligned}$$

Substitution and closed terms $e[v/x]$ is the term in which all free occurrences of x in e are replaced wth v. This replacement operation is called *substitution*. A term with no free variables is called *closed term*. (A single variable x is not closed!)

The principle of "bound variable names don't matter" When you declare some functions, the parameter names can vary only if it doesn't conflict with the free variables of the expression.

In order to avoid variable clashes, it is very convenient to *alpha–convert* expressions so that bound variables don't get in the way. We define this form of substitution as $e\,[[z/x]]$ so it is a total function when z is not in $Vars(\backslash x.e)$. Terminology: *alpha–equivelant* when they are same after alpha–converting some of their bound variables.

Operational Semantics Single–step evaluation (judgement form): $e \rightarrow e'$. Primary rule (beta reduction): $(\backslash x.e_1) \; e_2 \rightarrow e_1\,[e_2/x]$. A term of the form in the left of the previous rule is called *redex* (reducible expression).

Evaluation strategies Each strategy defines which redex in an expression gets reduced on the next step of evaluation.

- Full beta-reduction: any redex
- Normal order: leftmost, outermost redex first
- Call–by–name: similar to normal order except NO reduction inside lambda abstractions
- Call–by–value: only outermost redex, whose right-hand-side must be a value.

The lambda calculus can be used to do anything, which can support the useful, high–level operations and language features.

Let expressions It is useful to bind intermediate results of computations to variables. And *let expression* can be implemented with lambda calculus.

Booleans

$$tru = \backslash t.\backslash f.t \quad fls = \backslash t.\backslash f.f$$

$$test = \backslash x.\backslash then.\backslash else.x\ then\ else$$

$$and = \backslash b.\backslash c.\ b\ c\ fls$$

Pairs

$$pair = \backslash f.\backslash s.\backslash b.\ b\ f\ s$$

$$fst = \backslash p.\ p\ tru$$

$$snd = \backslash p.\ p\ fls$$

And we can go on to encode numbers, arithetic expressions, lists, trees, and any datatypes in lambda calculus. The general trick is that values will be functions, construct these functions so that they return the appropriate information when called by an operation.

C. Simply-typed Lambda Calculus

Construct a similar system of PL that combines the pure lambda–calculus with the basic types such as bool and num. Adds new syntax on types. One special syntax to attention:

$$v ::= \backslash x : t.e \quad (Values : abstractionvalue)$$

Refer to slides 14_lambda.pdf, page 34 Evaluations rules (three): E-App1, E-App2, E-AppAbs. Typing rules (three): T-Var, T-Abs, T-App.

Progress Theorem Suppose e is a closed and well-typed term. Then either e is a value or there is some e' which $e \to e'$.

Preservation Theorem If $G \vdash e : t$ and $e \to e'$, then $G \vdash e' : t$.

Substitution Lemma If $G, x : t' \vdash e : t$, and $G \vdash v : t'$, then $G \vdash e[v/x] : t$.

XII. Functional Programming

A. Overview

They emerged in the 1960's with Lisp. Functional programming mirrors mathematical functions: domain = input, range = output. Variables are mathematical symbols: not associated with memory location. Pure functional PL is state–free. Referential transparency: a function's result depends only upon the values of its parameters.

B. Scheme

A derivation of Lisp, and our subset omits assignments, simulates looping via recursion, simulates blocks via functional composition. Scheme is Turing complete, but Scheme programs have a different flavor.

Expressions Cambridge prefix notation for all Scheme expressions. Comments begins with ";". Expression evaluation: replace names of symbols by their current bindings, evaluate lists as function calls in Cambridge prefix, constants evaluate to themselves.

Lists A list is a series of expression enclosed in parentheses. Lists represent both functions and data. Empty list is "()". It is actually a pair: (head, tail).

List transforming functions: car, cdr, cons, null?, equal?, append, list.

Elementary Values: Numbers (integers, floats, rationals), symbols, characters, functions, strings.

Control flow: conditional (if), case selection (case).

Defining functions: (define (name arguments) function–body)

Let expressions Allows simplification of function definitions by defining intermediate expressions.

C. Haskell

A more modern functional language. Many similarities with Lisp and Scheme. Key distinctions: lazy evaluation, extensive type system, cleaner syntax, notation closer to mathematics, infinite lists.

Expressions Infix or prefix, both do.

Lists and list comprehensions A list comprehension can be defined using a generator. Eg: moreevens = [2*x — x ¡- [0..10]]

Infinite lists Generators may include additional conditions, list comprehension can also be infinite.

List transforming functions: head, tail, :, ++, null, type, ==.

Elementary types and values: Numbers (Integer, Float), Booleans (Bool), Characters (Char), Strings.

Control flow conditional, guarded command.

Defing functions (1) define its prototype on first line, (2) define its parameters and body on the remaining lines. (Type inference). The functions in Haskell are polymorphic.

Tuples A tuple is a collection of values of different types. Its values are surrounded by parens and separated by commas. Functions on tuples: fst, snd.

Functions as arguments A function that applies another function to every member of a list, returning another list. Eg: map.

D. Examples

Many Haskell examples, such as Semantics of Clite, Symbolic differentiation, eight queens. Refer to slides 15_functional.pdf Page 40 for eight queens.

XIII. Imperative Programming

Oldest and most well–developed paradigm. It mirrors computer architecture.

A. What makes a language imperative

In a von Neumann machine memroy holds instructions, data, intellectual heart, and others. The program just follows the instructions.

B. Two ideas of imperative programming

Procedural abstraction: allows programmer to be concerned mainly with a function interface, ignoring the details of how it is computed.

Stepwise refinement (functional decomposition): utilizes procedural abstraction to develop an algorithm starting with a general form and ending with an implementation.

C. Expressions and assignment

Assignment statement is fundamental. There are two types of semantics: copy semantics — expression is evaluated to a value, which is copied to the target; used by imperative languages. reference semantics — expression is evaluated to an object, whose pointer is copied to the target; used by object–oriented languages.

D. Library Support for data structure

There exist vast libraries of functions for most imperative languages. Partilly accounts for the longevity of languages like Fortan, Cobol and C.

E. Turing completeness

Any system of rules that is capable of simulating a single–tape Turing Machine. Imperative languages are turing complete. The regular languages, Pushdown automata, and context–free grammar are not turing complete.

F. C

C was originally designed for and implemented on the UNIX OS on DEC PDP–11, by Dennis Ritchie. C is not tied to any particular hardware or system.

General Characteristics Relatively low level language, macro facility, conditional compilation, assignments are expression. Lacks: iterators, generics, exception handling, overloading.

G. Ada

Developed in late 1970's by Department of Defence. DoD spending billions of dollars on software, over 450 languages in use. They want to standardize on one language. Hence Ada is created by the Higher Order Language Working Group.

Ada83 has the problem of too large in size of lanuage and compiler. It get renewed interest with development of Spart Ada and NYU GNAT (Ada) compiler.

General Characteristics It influences Algol and Pascal. A large PL, case insensitive. Unlike C, array indexing errors are trapped. It is type safe, use generics, provides exception handling.

H. Perl

A widely used scripting language, dynamically typed, encourage a variety of styles, supports regular expression pattern matching. Larry Wall created Perl when he was trying to produce some reports from a Usenet–news–like hierarch of files for a bug reporting system.

Scripting Languages Serve as "glue" for applications. Take output from one application and reformat into desired input format for a different application. Most time is spent in the underlying applications. Also used for web applicationos.

General characteristics Dynamically typed, default conversion from one type to another (vs. Python). Result is distinct operators. Types: numbers, strings, regular expressions. Dynamic arrays: indexed and associative. Many different ways of saying the same thing, much of the syntax is optional, Eg, (). Strength: great support for regular expressions. Weakness: manay irregularities in Perl.

Arrays Indexed arrays @a = (2,3,34), $a[1] == 3. Associative arrays %d = ("bob" =>"2312", "allen" =>"12312"). Scalar variables start with a $. Indexed arrays with @. Hash arrayes with %. Otherwise: bare word syntax error.

Use strict forces declaration of variables. local: dynamic scoping; my: static scoping; NB: only 1 $_.

Strings double quotes: special chars interpreted. Single quotes: special chars uninterpreted.

Reading from input *while*(<>){...} is same as *while*($_ =< *STDIN* >){...}, where <> read a line, returns undef at end of file, and undef interpreted as false.

More Perl program demos Ex: Mailing Grades.

Regular expressions Operators: m/// — match, m optional; s/// — substitute; split — array returning function. Modifiers: i — case insensitive; m — treat string as multiple lines; s — treat string as a single line; x — extend with whitespace and comments.

Most characteristics match themselves. Meta characteristics:
\ — predefined sets: \d — digits; \w — letters plus digits; \s — whitespace
| — or
( ) — grouping
[ ] — set
{ } — quantifiers, eg: a{m,n} matches at least m *a*s and at most n *a*s ˆ — not
$ — if last
* + ? — Kleene star
. — single character wildcard

XIV. Object–Oriented Programming

A. Abstract Data Types

Imperative programming paradigm: algorithms + data structures = programs. Produce a program by functional decomposition.

Procedural abstraction Concerned mainly with interface, ignore details of how.

Data abstraction (abstract data type), extend procedural abstraction to include data; extend imperative notion of type by providing encapsulation of data/functions, seperation of interface from implementation.

Encapsulation is a mechanism which allows logically related constants, types, variables, methods, and so on, to be grouped into a new entity.

Goal of data abstraction Package data type and functions into a module so that functions provide public interface and defines type.

B. The object Model

Problems remained: automatic initialization and finalization, no simple way to extend a data abstraction.

Object decomposition rather than function decomposition.

Class is a type declaration which encapsulates constants, variables, and functions for manipulating these variables. It is a mechanism for defining an ADT.

Concepts in OOP: constructor, destructor, client of a class, class methods, instance methods, OO program — collection of objects which communicate by sending messages.

Visibility: public, protected, private.

Inheritance: class hierarchy — subclass, parent or super class, is–a relationship, has–a relationship — aggregation. In single inheritance the class hierarchy forms a tree. Rooted in a most general class: object. Inheritance supports code reuse.

Some languages are single inheritance, such as Smalltalk and Java. There do have multiple inheritance, which allows a class to be a subclass of zero, one, or more classes. Class hierarchy is a directed graph. Advantage: facilitates code reuse. Disadvantage: more complicated semantics.

Object oriented language A language is OO if it supports: an encapsulation mechanism with information hiding for defining abstract data types, virtual methods, and inheritance.

Polymorphism has many forms. In OO language polymorphism refers to the late binding of a call to one of several different implementations of a method in an inheritance hierarchy.

A subclass method is substitutable for a parent class method if the subclass's method performs the same general function.

Templates or Generics A template defines a family of classes parameterized by one or more types. It is a kind of class generator, and can restrict a collections class to holding a particular kind of object.

Abstract classes An *abstract class* is one that is either declared to be abstract or has one or more abstract methods. An *abstract method* is a method that contains no code beyond its signature.

Any subclass of an abstract class that does not provide an implementation of an inherited abstract method is itself abstract. Because abstract classes have methods that cannot be executed, client programs cannot initialize an object that is a member of an abstract class. This restriction ensures that a call will not be made to an abstract method.

Interfaces An *interface* encapsulates a collection of constants and abstract method sigatures. An interface may not include either variables, constructors, or non–abstract methods.

Difference between interface and abstract classes: in interface all methods must be abstract, only constants; in abstract class some methods can be implemented, variables can be declared.

Difference between interface and multiple inheritace: An interface does not have a constructor, but an abstract class does. Some like to think of an interface as an alternative to multiple inheritance. Strictly speaking, however, an interface is not quite the same since it doesn't provide a means of reusing code. An interface is similar to multiple inheritance in the sense that an interface is a type. A class that implements multiple interfaces appears to be many different types, one for each interface.

Virtual Method Table (VMT) How the appropriate virtual method is called at run time. At compile time the actual run time class of any object may be unknown. Each class has its own VMT, with each instance of the class having a reference (or pointer) to the VMT. A simple implementation of the VMT would be a hash table, using the method name as the key and the run time address of the method invoked as the value.

For statically typed languages, the VMT is kept as an array. The method being invoked is converted to an index into the VMT at compile time.

Run time type identification Run Time Type Identification (RTTI) is the ability of the language to identify at run time the actual type or class of an object. All dynamically typed languages have this ability, whereas most statically typed imperative languages, such as C, lack this ability. At the machine level, recall that the data is basically untyped.

Reflection is a mechanism whereby a program can discover and use teh methods of any of its objects and classes. It is essential for programming tools that allow plugins and for JavaBeans components.

In Java the *Class* class provides the following information about an object: the superclass, the names and types of all fields, the names and signatures of all methods, the signature of all constructors, the interface that class implements.

C. Smalltalk

The original object-oriented language, developed in 1970s at Xerox PARC (OS, IDE, mouse based GUI, smalltalk system).

General characteristics Simple language, most of the class libraries written in Smalltalk; everything is an

object, even control structures; excluding lexical productoins, grammers has 21 productions; the value of every variable is an object, every object is an instance of some class; a method is triggered by sending a message to an object: the object responds by evaluating the method of the same name, otherwise this message is sent to the parent object; all methods return a value. Precedence: Unary messages, binary messages, keyword messages, in the absence of parentheses, code is evaluated from left to right. By default, Smalltalk use infinite precision.

Example: Polynomials refers to slides 18_object.pdf Page 55.

XV. Logic Programming

A. Preliminary Concepts

Computation vs. Deduction

Compute:
An expression + rules (program) → result.

Deduce:
A conjecture + rules (axioms/inference rules) → a proof.

Logic Programming unites these two: if we fix a strategy for proof search, then deduction can be considered a computation. Stratege == algorithm in logic.

Proof system Recall Curry–Howard Isomorphism: logic == type system. Deduction based on inference rule.

Proof search search for the right rule to apply at each step. Two strategies: backward reasoning (goal–directed) — from conjecture (goal) to axioms; forward reasoning — from axioms to conjecture.

Answer substitution LP also computes values. Search not only constructs a proof, but also searches for a value R that makes the goal hold.

Backtracking when a goal matches the conclusion of more than one rule: we reach a *choice point*. At this point, we pick a rule and attempt the proof. If fail, we go back to the most recent choice point, pick another rule.

Subgoal order When a rule contains multiple premises, we need to determine which premise (or subgoal) to attempt first. The order of subgoal evaluation has a significant impact on the computation: complete in a few steps, non-terminating.

B. Horn Clauses

An inference rule can be represented as a *horn clause*. A horn clause has a head h, which is a *predicate*, and a body, which is a list of *predicates* $p_1, \ldots, p_n$.

Any horn clause

$$h \leftarrow p_1, p_2, \ldots, p_n$$

can be written as a predicate:

$$p_1 \wedge p_2 \wedge \cdots \wedge p_n \supset h$$

or equivalenty:

$$\neg(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \vee h$$

But not every predicate can be written as a Horn clause.

Resolution and unification If h is the head of a Horn clause, and it matches one of the terms of another Horn clause, then that term can be replaced by h's terms.

During resolution, assignment of variables to values is called *instantiation*. *Unification* is a pattern matching process that determines what particular instantiations can be made to variables during a series of resolution.

C. Logic Programming in Prolog

In logic programming the program declares the goals of the computation, not the method for achieving them.

Logic programming has applications in AI and databases: NLP, automated reasoning and theorem proving, expert system, DB searching as in SQL. Prolog 1970s with two distinguishing features — nondeterminism and backtracking.

D. Prolog Program Elements

Prolog programs are made from terms: variables [begins with capital letter], constants [either integers or atoms], structures [predicates with arguments]. The arity of a structure is the number of its arguments.

Facts, Rules, and Programs A Prolog *fact* is a Horn clause without a right–hand side. A Prolog *rule* is a Horn clause with a right–hand side, eg:

$$term : -term_1, term_2, \cdots, term_n.$$

A Prolog *program* is a collection of facts and rules.

Queries — searching for success. A query is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest. Eg:

$$? - speaks(who, russian).$$

Unification — anwser the query Prolog considers every fact and rule whose head is *speaks*; resolution and unification locate all the successes.

Lists A *list* is a series of terms separated by commas and enclosed in brackets. Eg: empty list []; a "don't care" entry is denoted by _; function *append* joins two lists, function *member* tests for membership.

Pattern matching governs tests for equality. Don't care entries _ mark parts of a list that aren't important to the rule.

More list functions *prefix*, *surffix*.

E. Practical Aspects of Prolog

Tracing To see dynamics of a function call, the **trace** function can be used. Note: the argument to **trace** must include the function's arity.

The Cut is an operator (!) inserted on the right–hand side of a rule. *Semantics:* the cut forces the subgoals to its left not to be retried if the right–hand side succeeds once, i.e. no backtrace to the left of !. It helps limit search space and improves performance.

The *IS* operator instantiates a temporary variable.

Other operators Prolog provides the operators

```
+ - * / ^ = < > >= <= \=
```

with their usual interpretations. The **not** operator is implemented as goal failure.

The *assert* function can update the facts and rules of a program dynamically.

F. Prolog Examples

Eg: Solving word puzzles, NLP. Refers to slides 20_logic.pdf page 38.

XVI. Concurrent Programming

Concurrency occurs at many levels. Traditionally studied in the context of OS, such as client–sever application such as web browsing.

A. Concurrency Concepts

Some definitions Multiprogramming several programs loaded into memory and executed in an interleaved manner. Scheduler: switches from one program or thread to another. Time-sharing: allow muliple users to communicate with computer simultaneously. Process: an execution context, including registers, activation stack, next instruction to be executed. A concurrent program is a program designed to have two or more execution contexts. Also called multithreaded, since more than one execution context can be active simultaneously. Parallel program 2 or more threads simultaneously active. Distributed program designed so that different pieces are on computers connected by a network. Concurrency and Single threaded.

States of a thread Created, runnable, running, blocked, terminated.

Inter–threaded communication needs to occur: thread requires exclusive access to some resource; thread needs to exchange data with another thread. They can communicate via shared variables, message passing, parameters.

A race condition occurs when the resulting value of a variable depends on the execution order of two or more threads.

A deadlock occurs when a thread is waiting for an event that will never happen.

B. Synchronization Strategies

A semaphore is an integer variable and an associated thread queue.

A monitor encapsulates a shared resource together with access functions, also known as *conditional variable*.

C. Synchronization in Java

Java threads Each subclass of *Thread* class must provides a run method, which normally contains a loop. A class implement the *Runnable* interface can also be a thread.

D. Concurrency in Other languages

C# Monitors with method Monitor.Enter(0) and Monitor.Exit(0).

C++ No generic support, ACE project provides wrapper class for semaphores.

High–Performance Fortan Specify number of processors, distribution of data, FORALL (concurrent loop).

Ada Rendezvous: two tasks can only communicate if both are ready. Protected objects are more efficient than rendezvous. Asynchronous communication, either entry or delay clause can be activated.

Acknowledgment

I would like to thank Prof. Kenny Q. Zhu for the one semester's teaching and Mr. Tianwan Zhao for his tutorial and assignment judgement.

References

[1] Allen B. Tucker, Robert E. Noonan, *Programming Languages Principles and Paradigms*, 2nd edition McGraw Hill, 2009.

[2] *CS383 Programming Language Concepts* course website, http://www.cs.sjtu.edu.cn/~kzhu/cs383/, Fall semester of 2012.

Hongjian Wang I am just one normal undergraduate majoring computer science at Shanghai Jiao Tong University. I have a huge interesting in computer programming language. Currently, I am skilled programmer in C/C++, Python, and C#; I can write Tcl/Tk and Java too; with help of documents, I also write Haskell and Perl code; besides, I have sufficient programming experience on web development, including PHP + mySQL, Python Django, and CSS + JS or AJAX in the front end; one more thing, I prefer LaTeX than any other text processor. Programming language is the most powerful object I have ever seen. I believe that coding is an attitude of problem solving.

You can get in touch with me with sishenhadesi@163.com, and my personal website is http://grid.sjtu.edu.cn/~hongjianwang (welcome to visit).