**Design Of**

**Programming Languages**

Prof. M. S. Fallah

Fall 2015

9231064

[Important calculation models 3](#_Toc434063637)

[1. Functional Model 3](#_Toc434063638)

[2. Logic Model 3](#_Toc434063639)

[3. Imperative Model 4](#_Toc434063640)

[Summary 4](#_Toc434063641)

[Syntax 5](#_Toc434063642)

[Grammar 5](#_Toc434063643)

[Types of Grammars 6](#_Toc434063644)

[Wren Language 8](#_Toc434063645)

[Ambiguity 8](#_Toc434063646)

[static semantic 9](#_Toc434063647)

[Context constraints 9](#_Toc434063648)

[Semantic Errors 10](#_Toc434063649)

[Abstract Syntax 10](#_Toc434063650)

[Attribute Grammars 12](#_Toc434063651)

[Semantics 13](#_Toc434063652)

[1.Operational Semantics 13](#_Toc434063653)

[2. Denotational Semantics 15](#_Toc434063654)

[A While Language 16](#_Toc434063655)

[Nonstandard Semantics 18](#_Toc434063656)

[Abstract interpretation 18](#_Toc434063657)

[IMPERATIVE and declarative 19](#_Toc434063658)

[Functional language 19](#_Toc434063659)

[Declarative Language Test 19](#_Toc434063660)

[The World Of Expressions and The World Of Statements 19](#_Toc434063661)

[Church-Rosen Property (Confluence) 20](#_Toc434063662)

[Referential Transparency 20](#_Toc434063663)

[Lambda-Calculus 21](#_Toc434063664)

[Untyped λ-calculus: 21](#_Toc434063665)

[scope 21](#_Toc434063666)

[Operational Semantics 21](#_Toc434063667)

[Evaluate Strategies 22](#_Toc434063668)

[1. Full beta-reduction 22](#_Toc434063669)

[2. Normal order 22](#_Toc434063670)

[3. Call by name (non-strict or lazy) 22](#_Toc434063671)

[4. Call by value (strict) 22](#_Toc434063672)

[Programming in λ 23](#_Toc434063673)

[Church Boolean 23](#_Toc434063674)

[Church Numerals 24](#_Toc434063675)

[Recursion 25](#_Toc434063676)

[Fixed Point 25](#_Toc434063677)

[Call-by-name y combinator (or fixed point combinator) 25](#_Toc434063678)

[Call-by-value Z combinator 25](#_Toc434063679)

[some Programming languages 26](#_Toc434063680)

[Lisp 26](#_Toc434063681)

[Historical Lisp structure 26](#_Toc434063682)

Session 1

#### Definitions

* A **computational model** is collection of values and operations.
* A **computation** is the application of a sequence of operations to values to yield another value.
* A **programme** is a specification of a computation.
* A **programming language** is a notation for writing programmes.

# Important calculation models

## 1. Functional Model

**Values**: functions

**Operations**: function operations

Example:­

Sd(xs) = sqrt (v)

Where

n= length(xs)

v= fold(plus , map(sqr , xs) ) /n - sqr(fold(plus , xs) / n)

\*Pure functional lang: Haskell

\*functional langs: ML – Lisp …

\*This language is "Declarative".

\*Functional and Logical languages are Declarative.

## 2. Logic Model

**Values**: facts, definitions of relations

**Operations**: logical inferences

Example:

1. Human(Socrates)
2. Human(Penelope)
3. Mortal(x) if Human(x)
4. ­­­­­­­­­­­­­¬mortal(y) Assumption
5. x=y (3),(4) and unification and Modus Tollens
6. ¬human(y) (3),(4) and unification and Modus Tollens
7. y=Socrates (1),(5),(6) and unification
8. y=Penelope (1),(5),(6) and unification
9. Contradiction(¬human(Socrates) and human(Socrates))

\*"Prolog" is a logical language or framework.

## 3. Imperative Model

**Values**: states

**Operations**: state transitions

Example:­

constant pi = 3.14

input (radius)

circumference := 2 \* pi \* radius

output(circumference)

\*languages: Pascal - C - C++ - Java – Assembly

## Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Computational model | Value | Operation | example |
| Imperative | state | State transition | C |
| Functional | functional | Function application | Haskell |
| logic | Facts and relations | rules | Prolog |

Session 2

# Syntax

A language: **Syntax**, **Semantics**, **Pragmatics**.

\*The amount of expressiveness of a language.

\*Chomsky, The linguist said the following

## Grammar

A grammar (∑, N, P, S) consists of four parts;

1- ∑ : terminal symbols or alphabet

2- N : nonterminal symbols or syntactic alphabet

3- P : productions or rules

4- S : the start symbol

#### **BNF** (Backus-Naur Form):

<declaration>::=**var**<variable list>**:**<type>**;**

\*BNF is called a metalanguage, because it defines languages.

Example

var x,y : int ;

#### Definition

**Vocabulary**: terminals and nonterminals.

A production **α::=** **β**

**\*α** must have at least one nonterminal.

### Types of Grammars

#### Type 0 (Unrestricted grammars):

At least one nonterminal occurs on the left side of a rule.

Example

**a**<thing>**b** ::= **b**<another thing>

#### Type 1 (Context-sensitive grammars):

The right side contains no fewer symbols than the left.

Example

<thing>**b** ::= **b**<thing>

rules would be like this:

α <B> Ɣ ::= α β Ɣ

#### Type 2 (Context-free grammars):

The left side is a single nonterminal.

Example

<A> ::= β

rules would be like this:

<expression> ::= <expression> **a** <term>

\*BNF is a rule for specifying Type 2 languages and programming languages are defined by it.

#### Type 3 (Regular grammars):

The left side is a single nonterminal.

Example

<A> ::= β

rules would be like this:

<A> ::= **a**

**or**

<A> ::= **a** <B>

\*These languages would be accepted by Finite automata.

Example

A grammar for binary numbers

<binary number> ::= **0**

<binary number> ::= **1**

<binary number> ::= **0** <binary number>

<binary number> ::= **1** <binary number>

**or**

<binary number> ::= **0**|**1**|**0**<binary number>|**1**<binary number>

Example

A grammar for a natural language

<sentence> ::= <noun phrase><verb phrase> **.**

<noun phrase> ::= <determiner><noun>|<determiner><noun><prepositional phrase>

<verb phrase>::=<verb>|<verb><noun phrase>|<verb><noun phrase><prepositional phrase>

<prepositional phrase>::=<preposition><noun phrase>

<noun>::=**boy | girl | cat | telescope**

<determiner>::=**a | the**

<verb>::=**say | go | shop | saw**

*<preposition>::=* ***by | with***

\*The latter language is Ambiguous.

Example

A context-sensitive grammar

<sentence> ::= **a b c** |**a** <thing> **b c**

<thing> **b** ::= **b** <thing>

<thing> **c** ::= <thing> **b c c**

**a** <other> ::= **a a** | **a a**  <thing>

**b** <other> ::= <other> **b**

The language would be like this:

{an bn cn | n ϵ Z+}

#### Definition

A grammar is **Ambiguous** if some phrases in the language generated by the grammar has two or more distinct derivation trees.

Session 3

### Wren Language

Syntaxes are either:

**Lexical syntax** 🡪 Lexical Analysis (scanning)

or

**Phrase-Structure syntax** 🡪 Syntactic Analysis (Parsing)

### Ambiguity

Having more than one derivation tree for a statement in a language.

Example

if exp1 then if exp2 then cmd1 else cmd2

Session 4

## static semantic

Example

**Program** illegal **is**

**var** a : **boolean**;

**begin**

a := 5

**end**

There are two ideas about this problem:

It's a **syntactic** problem.

It's a **static semantic** problem.

so we are going to need a **static analyser:**

Static semantics cannot be analysed by context free machines.

We do not use a Type 1 (context sensitive) machine because it is much harder to have a context sensitive compiler.

### Context constraints

1. All identifiers that appear in a block must be declared in that block
2. No identifier may be declared more than once in a block.
3. An identifier occur in a read command must be an integer variable.

.

.

.

### Semantic Errors

Example

1. An attempt is made to divide by zero.
2. A variable that has not been initialised has been accessed.
3. A read command is executed when the input file is empty.
4. type mismatch.

### Abstract Syntax

Is a way of fixing the redundancy in a concrete syntax.

\***Concrete** is the opposite of Abstract.

Example

5\*a-(b+1)

Derivation tree:

#### AST : Abstract Syntax Tree

<expression> 🡪 … 🡪 operations

numerals

identifiers

boolean constants

Syntactic Categories: Expression , Numeral , Identifier

Now we can define an abstract syntax by removing **unit rules** (rules without terminals) unless they have a basic component.

Session 5

### Attribute Grammars

(Sebesta book P.134)

Concepts:

* Attribute
* Attribute computation functions (semantic functions)
* Predicate functions (conditions)

To Avoid Static semantic problems, attribute grammars are used.

A(X) : The set of attributes associated with symbol X.

**A(X) = S(X) U I(X)**

S(X) : synthesised I(X) : Inherited

X0 ::= X1 … Xn (a production)

S(X0) = f ( A(X1) , … , A(Xn) )

i(Xj) = f ( A(X0) , … , A(Xn) ) 🡪 i(Xj) = f ( A(X0) , … , A(Xj-1) )

P ( A(X0) , … , A(Xn)) or is a predicate over

A Fully attributed tree

**Intrinsic attributes** are some kind of synthetic Attributes, which are given to leaves of a tree.

Example

<proc\_def> ::= **Procedure** <proc\_name>[1] <proc\_body> **end** <proc\_name>[2]

Predicate : <proc\_name> [1].string == <proc\_name> [2].string

Example

<assign> ::= <var> = <expr>

<expr> ::= <var> + <var> | <var>

<var> ::= A | B | C

**actual\_type**: A synthesised attribute associated with the nonterminals <var> and <expr>

**expected\_type**: An inherited Attribute associated with the nonterminal <expr>

so the attribute grammar would be like this:

<assign> ::= <var> = <expr>

<expr>.expected\_type = <var>.actual\_type

<expr> ::= <var>[1] + <var>[2]

<expr>.actual\_type = if(<var>[1].actual\_type = int

and<var>[2].actual\_type = int)

then int else real endif

predicate : <expr>.actual\_type == <expr>.expected\_type

<expr> ::= <var>

<expr>.actual\_type=<var>.actual\_type

predicate : <expr>.actual\_type == <expr>.expected\_type

<var> ::= A | B | C

<var>.actual\_type = look\_up( <var>.string )

Session 6

# Semantics

1. Operational Semantics
2. Denotational Semantics (معنا شناسی دلالتی) 🡪John Michel's book (The Bible of Denotational Semantics :)
3. Axiomatic Semantics

## 1.Operational Semantics

Example

t ::= true | false | if t then t else t | 0 | succ t | pred t | iszero t

Inductive Definitions (judgment of **t term** ):

\_\_\_\_\_\_\_\_\_\_ (Axiom)

true term

\_\_\_\_\_\_\_\_\_\_ (Axiom)

false term

t1 term t2 term t3 term (proper machine)

if t1 then t2 else t3 term

v ::= true | false values

Evaluation: (Small-Step)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

if true then t2 else t3  🡪t2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

if false then t2 else t3  🡪t3

if t1 🡪t1'\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

if t1 then t2 else t3  🡪 if t1' then t2 else t3

#### Theorem - Determinacy of one-step evaluation

if t🡪t' and t🡪t", then t'=t"

#### Definition

A term **t** is in **normal form** if no evaluation rule applies to it.

#### Theorem

In our language (the above language) If t is normal form, then t is a value.

#### Definition

The **Multistep evaluation** relation 🡪\* is the reflexive and transitive closure of🡪.

if t🡪t' , then t🡪\*t'

t🡪\*t

if t🡪\*t' and t'🡪\*t" , the t🡪\*t"

#### Theorem

For every term t, there is some normal form t' such that t🡪\*t'.

v ::= true | false | nv

nv ::= 0 | succ nv

t1 🡪t1'\_\_\_\_\_\_

succ t1 🡪succ t1'

\_\_\_\_\_\_\_\_\_\_\_

pred 0 🡪0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

pred(succ nv1) 🡪nv1

t1 🡪t1'\_\_\_\_\_\_\_

pred t1 🡪pred t1'

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

iszero 0 🡪true

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

iszero (succ nv1) 🡪false

t1 🡪t1'\_\_\_\_\_\_\_\_\_\_

iszero t1 🡪iszero t1'

#### Definition

A closed term is **stuck** if it is normal form but not a value.

\*further studies: middle weight Java and its Operational semantics.

Session 7

## 2. Denotational Semantics

Christopher Strachey and Dana Scott had introduced it back in 1960s.

Denotational Semantics consists of **:**

**Object language (programme)**

and **Meta language (mathematics)**.

Example

Object language: x :=0; y := 0; while x<=z do (y := y + x; x:= x+1)

Meta language: F(z)=1+2+3+…+z

**Compositionality** is a feature of this kind of semantics meaning that if elements of two phrases are the same then the phrases are the same.

Example

B≡B' , P≡P' , Q≡Q' 🡪 if B then P else Q ≡ if B' then P' else Q'

Example

Denotational semantics for binary numbers:

e::=n|e+e|e-e

n::=b|nb

b::=0|1

\*[[e]] = The parse tree of e

E[[e]] is the meaning of e

E[[0]]=0

0 is from the meta language(the mathematical language).

E[[1]]=1

E[[nb]]=E[[n]]\*2+E[[b]]

E[[e1+e2]]=E[[e1]]+E[[e2]]

Some arithmetic expressions:

e::= v| n| e+e |e-e

n::= d|nd

d::=0|1|…|9

v::=x|y|z|…

A **programme** is a function from states to states. P: states 🡪 states

A **State** is a function from variables to values. S: Variables 🡪 values

Now we define E[[e]](S)

E[[x]](S) = S(x)

E[[0]](S) = 0

…

E[[9]](S) = 9

E[[nd]](S)=E[[n]](S)\*10+E[[d]](S)

E[[e1+e2]](S)=E[[e1]](S)+E[[e2]](S)

E : Parse tree 🡪( states 🡪N )

E([[e]],S)

E: parse tree 🡪 N states parse tree🡪(states 🡪 N)

E[[e]](S)

C : Parse tree 🡪( state 🡪 state )

\*both E and C give us a function from states in return.

### A While Language

P::= x:=e|P;P| if e then P else P | while e do P

State = Variables 🡪 Values

Command = States 🡪 States

modify(s,x,a) = λv ϵ variables [if v=x then a else s(v)] \*s(v) is the value of v in the state s

C[[P]](s) is supposed to be a meaning function

C[[x:=e]](s)=modify(s, x, E[[e]](s))

C[[P1;P2]](s)=C[[P2]] (C[[P1]](s) )

C[[if e then P1 else P2 ]](s) = if E[[e]](s) then C[[P1]](s) else C[[p2]](s)

C[[while e do P]](s) = if not E[[e]](s) then s else C[[while e do p]]( C[[P]](s) )

Session 8

Example

if x>y then x:=y else y:=x

s0(x) =1 , s0(y)=2

s1(x) =1 , s1(y)=1

C[[if x>y then x:=y else y:=x]](s0) = if E[[x>y]](s0) then C[[x:=y]](s0) else C[[y:=x]](s0)

= C[[y:=x]](s0)

=modify (s0, y, E[[x]](s0))=modify(s0,y,1)=s1

Example

P= x:=0; y:=0; while x<=z do (y:=y+x; x:=x+1)

s0(z) =2

s1=modify(s0,x,0)

s2=modify(s1,y,0)

C[[P]](s0)

= C[[while x<=z do (y:=y+x; x:=x+1)]](s2)

= if not E[[x<=z]](s2) then s2 else C[[while x<=z do (y:=y+x; x:=x+1)]](C[[y:=y+x; x:=x+1]](s2))

= C[[while x<=z do (y:=y+x; x:=x+1)]](s3)

= …

=s' which s'(x)=3, s'(y)=3, s'(z)=2

C is a **partial function** meaning that it is undefined for some programmes.

Example

C[[while x=x do x:=x]] (s) = ?

or

C[[while x=y do x:=y]](s) = s if s(x) ≠ s(y)

undefined otherwise

## Nonstandard Semantics

They are used in **programme analysis** (Data flow analysis, … , Abstract interpretation).

### Abstract interpretation

Example

To check programmes to make sure that every variable is initialised before it is used.

\*Methods of programme analysis can only be **conservative** so they wouldn't have a false positive which means they're sound, because of the halting problem being unsolvable.

error error state

variables 🡪{ init , uninit }

states ={ { error } U { variables 🡪{ init , uninit } } }

C[[P]](s)

E[[e]](s) =err if e contains any variable y with s(y) = uninit

E[[e]](s) = OK otherwise

for example

C[[x:=e]](s) = if E[[e]](s)= OK then modify(s, x, init) else error

C[[P1;P2]](s)= if C[[P1]](s) = error then error else C[[P2]]( C[[P1]](s) )

s1\*s2 = λv ϵ Variables if s1(v)=s2(v)=init then init else uninit

C[[if e then P1 else P2]](s) = if E[[e]](s)=err or C[[P1]](s)=error or C[[P2]](s)=error

then error else C[[P1]](s) \* C[[P2]](s)

C[[if 0=1 then x:=0 else x:=1; y:=2]](s0) = modify( s0, x, init )

Session 9

Michel's book chapter 4

# IMPERATIVE and declarative

There are four kinds of sentences in natural languages:

* Imperative
* Declarative
* Interrogative
* Exclamatory

Programming languages are one of the first two.

## Functional language

A programming language in which most computation is done by evaluation of expressions that contain functions. Like *Lisp, Haskell* and *ML* languages.

## Declarative Language Test

Within the scope of specific declaration of x1,…,xn , all occurrences of an expression *e* containing only variables x1,…,xn have the same value.

## The World Of Expressions and The World Of Statements

Example

z:=(z\*a\*y+b)\*(z\*a\*y+c)

* t:=z\*a\*y

z:=(t+b)\*(t+c)

this is OK in the world of expressions

y:=(z\*a\*y+b); z:=(z\*a\*y+c);

* t:=z\*a\*y

y:=t+b z:=t+c

but this is not OK in the world of statements

### Church-Rosen Property (Confluence)

No matter what the order of decoration, as long as we obey the structure of the tree, we will always get the same.

\*a language with this property is called **confluential.**

\* It doesn't matter which leaf we start from.

Example

(2ax+b)(2ax+c)

a=3 , b=2 ,c=-1, x=2

In this tree, to decorate (calculate) every node, we need to decorate its children first.

Example

**a+2\*F(b)**

*function F(x: integer) L integer*

*begin*

*F:=x\*x;*

*end*

this id pure functional

*function F(x: integer) L integer*

*begin*

*a:=a+1;*

*F:=x\*x;*

*end*

but this is not pure functional

### Referential Transparency

Example

*"I saw Walter get into his car."*

*"I saw Walter get into his Ferrari."*

This sentence is referentially transparent.

*"He was called William Rufus because of his red beard."*

*"He was called William IV because of his red beard."*

This sentence is **not** referentially transparent.

Session 10

# Lambda-Calculus

(Pierce's book – season 5)

**Core calculus** is the basic language which other languages have been built on it.

**Lambda-calculus (λ-calculus)** is the core of functional languages. (Introduced by Alonso Church)

**Pi-calculus (π-calculus)** is the core of concurrent languages.

**Object-calculus** is the core of object-oriented languages.

### Untyped λ-calculus:

t ::= x | λx.t | tt

rules:

- Application associates to left.

s u t = ((s u) t)

- The bodies of abstractions are taken to extend as far to the right as possible.

λx. λy.xyx= λx.( λy.(xy)x)

### scope

binding: an element can be bound or free and free elements are variables.

for every x , x>y. "y" is a variable and "for every" is a binder

in λ-calculus **λ** is the **binder**.

Example

in λz.λx.λy.x(yz) there is no variable because all of the elements have a binder.

this term is closed.

Closed terms are also called combinatory.

The most famous closed term is the identity function: id=λx.x

### Operational Semantics

(λx.t)t' 🡪 [xt']t (beta-reduction)

or

[t'/x] t

Example

(λx.x)y 🡪 y

(λx.x(λx.x))(ur)🡪(ur)(λx.x)

redex = reducible expression : (λx.t)t'

## Evaluate Strategies

Consider this term:

(λx.x)((λx.x)(λz.(λx.x)z))

or id ( id (λz.id z))

### 1. Full beta-reduction

id ( id (λz.id z)) id ( id (λz.z))

id (λz.z))

λz.z

### 2. Normal order

Starting from the outmost.

id ( id (λz.id z)) id (λz.id z)

λz.id z

λz.z

### 3. Call by name (non-strict or lazy)

Starting from the outmost.

No reductions inside abstractions.

id ( id (λz.id z)) id (λz.id z)

λz.id z

### 4. Call by value (strict)

Starting from the outmost.

No reductions inside abstractions.

Reduction can only be applied when the argument is a value (a λ abstraction).

id ( id (λz.id z)) id (λz.id z)

λz.id z

Session 11

## Programming in λ

+ : R\*R 🡪 R

+(2,3)=5

**Currying:**

+ : R 🡪 RR a function that gives back another function

(+(2))(3)=5 +(2) is a function that gets 3 as an argument.

### Church Boolean

tru = λt.λf.t tru v w =v

fls = λt.λf.f fls v w =w

we need to have a *test* like this:

test b v w = v b is tru

w b is fls

so the definition of *test* would be like:

test = λl.λm.λn.lmn

now if we give true v w to *test*:

test true v w 🡪(λm.λn.tru m n) v w

🡪(λn.tru v n) w

🡪(tru v w)

🡪(λt.λf.t) v w

🡪(λf.v) w

🡪v

and = λb.λc. b c fls

and tru tru = tru tru fls = tru

or= λb.λc. b tru c

or fls tru = fls tru tru= tru

neg=λb. b fls tru

pair = λf.λs.λb. b f s

fst=λp.p tru

scd=λp. p fls

fst(pair v w)=fst(λb. b v w) = (λp. p true) (λb. b v w)= tru v w = v

### Church Numerals

C0 = λs.λz.z

C1 = λs.λz.sz

C2 = λs.λz.s(sz)

C3 = λs.λz.s(s(sz))

scc = λn.λs.λz.s(nsz) successor

scc Cn = λs.λz.s(s(s(s…(sz)))…)) = Cn+1

­plus = λm.λn.λs.λz.ms(nsz)

times = λm.λn.m(plus n)C0­

iszero = λm.m ( λx.fls ) tru

zz= pair C­0 C0

ss= λp.pair ( snd p ) ( plus C1 (snd p) )

prd = λm. fst (m ss zz) predecessor

Session 12

## Recursion

### Fixed Point

f: A🡪A

Its Fixed point is  *xϵA : f(x)=x*

\*if x is a fixed point in f then, f(f(f(…..f(x)…)=x

Factorial is:

*f(0)=1*

*f(n+1) = (n+1) \* f(n)*

or

or

Let's define the functional F(f) = f':

The only fixed point of F is the factorial function.

### Call-by-name y combinator (or fixed point combinator)

Now we need a combinator fix: F 🡪the fixed point of F

**y = λh.( λx.h(xx) ) ( λx.h(xx) )** this is introduced by Church

Example

yF = (λx.F(xx)) (λx.F(xx)) = F( (λx.F(xx)) ( λx.F(xx)) ) = F (yF)

so yF is the fixed point of F

Now for the factorial:

fct = λf.λn. if n=0 then 1 else times n f(n-1)

factorial = y fct meaning the fixed point of fct

Example

y fct 2 = fct ( y fct ) 2

= ( λf.λn. if n=0 then 1 else times n f(n-1) ) (y fct ) 2

= ( λn. if n=0 then 1 else n \* ( y fct ) (n-1) )2

= if 2=0 then 1 else 2 \* ( y fct ) (2-1)

= 2 \* ( y fct ) (1)

= …

### Call-by-value Z combinator

Z= λh. ( λx.h (λy.xxy) )( λx.h (λy.xxy) ) this is introduced by Gordon Plotkin

which ZF = F(ZF) (to prove this we need to accept λx.mx=m which is called Etha-equivalence.)

Session 13

# some Programming languages

## Lisp

Abbreviation of "List Processor"

Developed in MIT at late 50s (by John McCarthy's team)

Motivating application: for symbolic computations and exploratory programming.

Example

Integ x^2 dx > x^3/+C

2x^2+x^3 > x^2 (2+x)

Some products:

* emacs
* gtk

Some developments and branches:

* Maclisp (MIT 1960s)
* Scheme (MIT 1970s)
* Common Lisp

Lisp project:

* Motivating application
* Abstract machine (IBM 704)

\*concrete 🡨🡪 abstract : concrete programme are less portable and abstract ones are less efficient.

* Theoretical foundation

An Article to read:

Recursive functions of symbolic expressions and their computation by machine.

CACM, 3(4), 184-195 (1960)

### Historical Lisp structure

#### Prefix

(+ 1 2 3 4) 🡪 1+2+3+4

#### Atom

<atom> ::= <symbol> | <number>

<smbl> ::= <char> | <smbl> <char> | <smbl> <digit>

<num> ::= <digit> | <num> <digit>

#### S-expressions and Lists

dotted pair : a.a

<sexp> ::=<atom> | (<sexp>.<sexp>)

#### Functions and special forms

cons, car , cdr , eq , atom

cond, lambda, define, quote, eval

+, - , \*

Until the above section Lisp is pure functional

here are some functions that make Lisp inpure.

rplaca, rplacd, set, setq

\* T true

nil false

Examples

(quote cons) 🡪 Makes an atom "cons"

(cons a b) 🡪 A pair containing the values of **a** and **b**

(cons (quote a)(quote b)) 🡪 A pair containing the atom "a" and "b"

'(+ 1 2) or (quote (+ 1 2)) 🡪 the list (+ 1 2)

(+ 1 2) 🡪 3

Examples

A function to find something in a list

*(define find (lambda(x y)*

*( cond ( ( equal y nil ) nil )*

*( ( equal x (car y) ) x )*

*( true ( find x (cdr y ) ) )*

*)*

*))*

now to use it we can say:

*( find 'apple '( pear peach apple banana fig ) )*