DESIGN OF PROGRAMMING LANGUAGES

Dr. M. S. Fallah
Amirkabir University of Technology

Autumn 2015
9231064

Caution:

This document is not complete and might be missing some parts.
For a complete study, **use the reference books** as well.

IN THE NAME OF GOD

IMPORTANT CALCULATION MODELS 5

1. Functional Model 5
2. Logic Model 5
3. Imperative Model 6
Summary 6

SYNTAX 7

Grammar 7
Types of Grammars 8
Wren Language 10
Ambiguity 10
static semantic 11
Context constraints 11
Semantic Errors 12
Abstract Syntax 12
Attribute Grammars 14

SEMANTICS 15

1.Operational Semantics 15
2. Denotational Semantics 17
A While Language 18
Nonstandard Denotational Semantics 20
Abstract interpretation 20

IMPERATIVE AND DECLARATIVE 21

Functional language 21
Declarative Language Test 21
The World Of Expressions and The World Of Statements 21
Church-Rosser Property (Confluence) 22
Referential Transparency 22

LAMBDA-CALCULUS 23

Untyped λ-calculus: 23
scope 23
Operational Semantics 23
Evaluate Strategies 24
1. Full beta-reduction 24
2. Normal order 24
3. Call by name (non-strict or lazy) 24
4. Call by value (strict) 24

Programming in λ 25
Church Boolean 25
Church Numerals 26
Recursion 27
Fixed Point 27
Call-by-name fixed point combinator (Y combinator) 27
Call-by-value fixed point combinator (Z combinator) 27

SOME PROGRAMMING LANGUAGES 28

Lisp 28
Historical Lisp structure 28
Scheme 30
The Lisp Abstract Machine 30
Cons cells (doted pairs) 30
Some functions 31
An Example 32
Function Expressions 32
Recursion, The History 32
Higher-Order Functions 32
Garbage Collection 33
Side Effects 34
The Algol Family and ML 35
Algol 60 35
Algol 68 36
Pascal 36
Modula 36
C 36
ML 36
ML 37
Read-eval-print 37
Data-types 38
Data-type declaration 40
Reference types 41

TYPE SYSTEMS AND TYPE INFERENCE 42

Some Reasons to Have Types 42
Type Errors 42
Types and Optimisation 42
Type Safety And Type Checking 43
Compile-Time and Run-Time Checking 43
An Example of Having Types 44
Safety 45
Type Inference (Type Reconstruction) 46
The Type Inference Algorithm 46
Polymorphism 48
Subtype Polymorphism 48

Parametric Polymorphism 48
Ad hoc Polymorphism (Overloading) 49
Type Declaration and Type Equality 50
Transparent type declaration 50
Opaque type declaration 50

SCOPE, FUNCTIONS AND STORAGE MANAGEMENT 51

Block-Structured Languages 51
In-Line Blocks 51
Activation Records and Local Variables 51
Global Variables and Control Links 53
Functions and Procedures 53
Activation records for functions 54
Parameter Passing 55
Global Variables 57
Higher-Order Functions 58
First-Class Functions 58
Passing Functions to Functions 58

FINAL EXAM 59

Problem 1 59
(a) 59
(b) 59
Problem 2 59
Problem 3 60
Problem 4 60
Problem 5 61
Problem 6 62
(a) 62
(b) 62

Session 1

IMPORTANT CALCULATION MODELS

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.

1. Functional Model

Values: functions

Operations: function applications

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	-
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	Function	Functional	Haskell
LOGIC	Facts And Relations	Inference	Prolog

Session 2

SYNTAX

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

* **Pragmatics** is the amount of expressiveness of a language.

*Chomsky, the linguist said the following definition of grammar.

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 ;

John Backus Peter Naur

Definition

Vocabulary: terminals and nonterminals.

A production **α::= β**

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

Types of Grammars

Type 0 (Unrestricted grammars) - Turing Machine

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

Example

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

Type 1 (Context-sensitive grammars) – Linear Bounded Automata

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) – Push-Down Automata

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) – Finite Automata

The left side is a single nonterminal and the right side is either a single terminal or a single terminal followed by 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:

$\{a^n b^n c^n \mid n \in Z^+\}$

Definition

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

Session 3

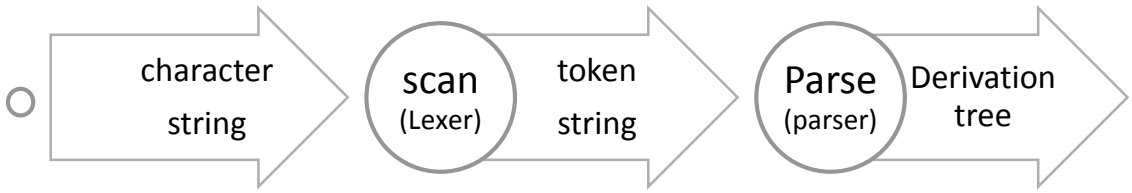
Wren Language

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

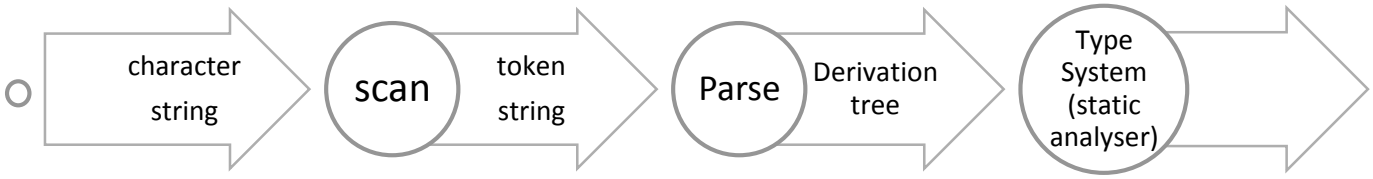
```
Program illegal is
        var a : boolean;
begin
        a := 5
end
```

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:**

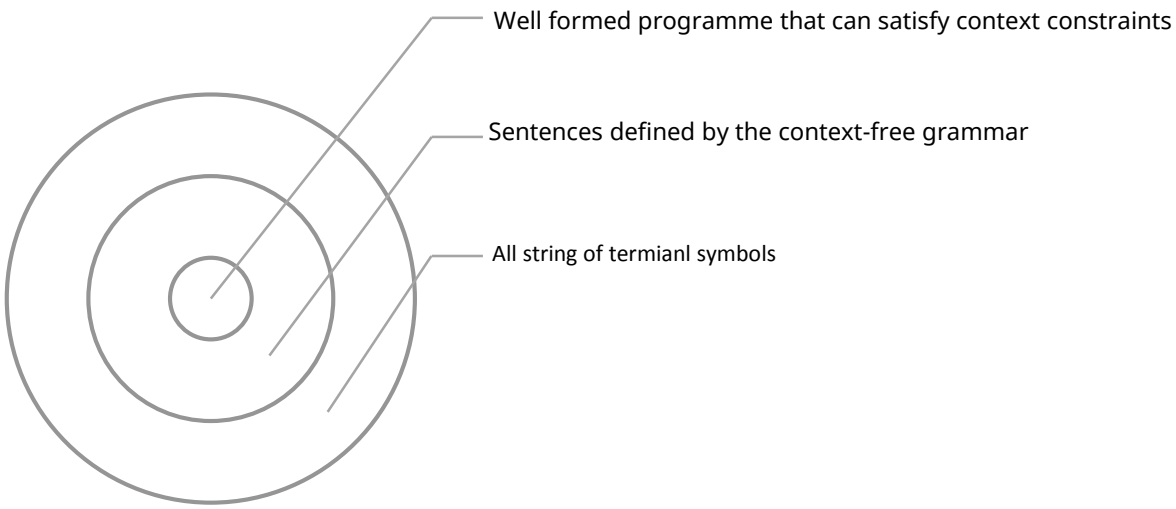


Static semantics cannot be analysed by context free machines.
We do not use a Type 1 (context sensitive) machine because it is much harder to have a context sensitive compiler.

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 that occurs 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

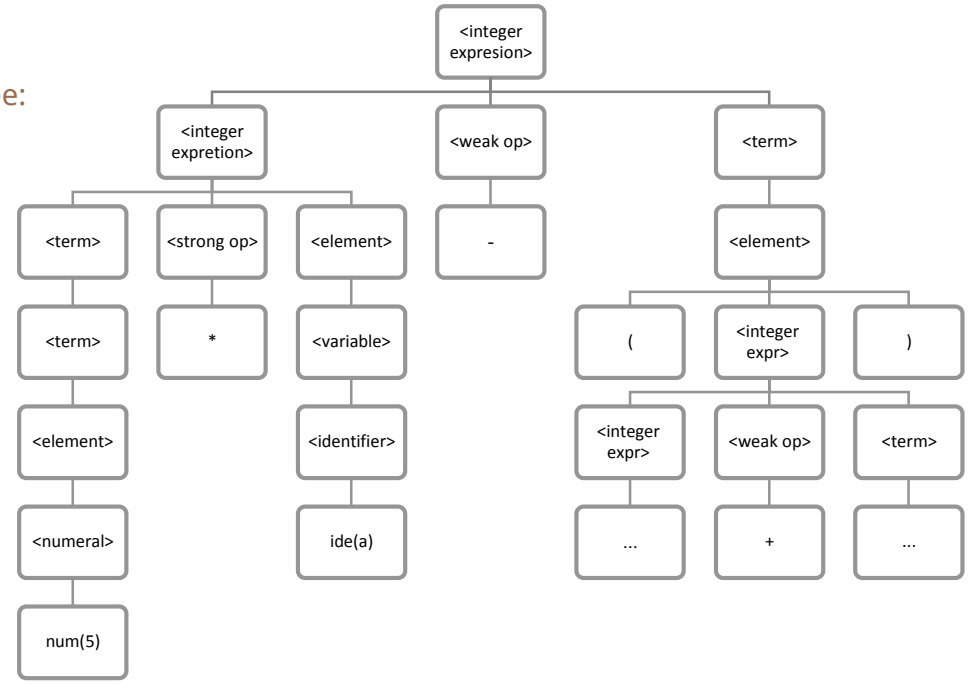
Is a way of fixing the redundancy in a concrete syntax.

*__Concrete__ is the opposite of Abstract.

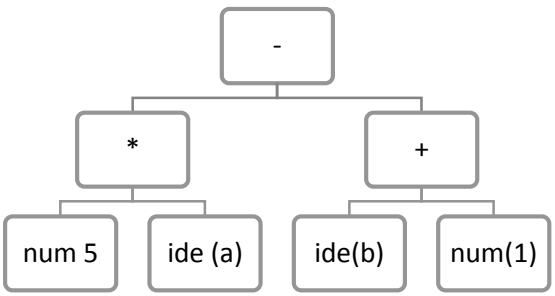
Example

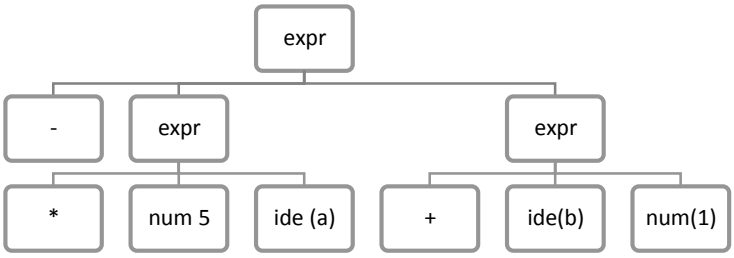
5*a-(b+1)

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's 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

$X_0 ::= X_1 \dots X_n$ (a production)

$S(X_0) = f(A(X_1) , \dots , A(X_n))$

$i(X_j) = f(A(X_0) , \dots , A(X_n)) \rightarrow i(X_j) = f(A(X_0) , \dots , A(X_{j-1}))$

$P(A(X_0) , \dots , A(X_n))$ or $P(\cup_{i=0}^{n} A(X_i))$ is a predicate over $\cup_{i=0}^{n} A(X_i)$

A Fully attributed tree is a tree that all of the associated attributes of its symbols are valid and computed.

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

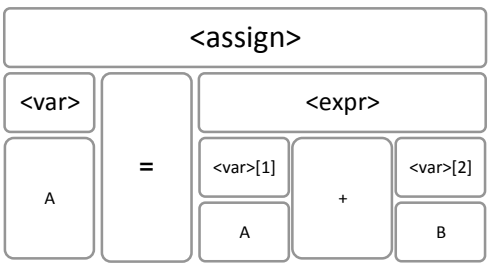
<expr> ::= <var>

<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 →John Mitchell'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**):

$$\frac{}{\text{true term}}$$ (Axiom)

$$\frac{}{\text{false term}}$$ (Axiom)

$$\frac{\text{t1 term t2 term t3 term}}{\text{if t1 then t2 else t3 term}}$$ (proper machine)

v ::= true | false values

Evaluation: (Small-Step)

$$\frac{}{\text{if true then t2 else t3}\rightarrow\text{t2}}$$

$$\frac{}{\text{if false then t2 else t3}\ \rightarrow\text{t3}}$$

$$\frac{\text{if t1}\rightarrow\text{t1}'}{\text{if t1 then t2 else t3}\ \rightarrow \text{if t1}' \text{ then t2 else t3}}$$

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 are some normal form t' such that t→*t'.

v ::= true | false | nv
nv ::= 0 | succ nv

$$\frac{t1 \rightarrow t1'}{\text{succ } t1 \rightarrow \text{succ } t1'}$$

$$\frac{}{\text{pred } 0 \rightarrow 0}$$

$$\frac{}{\text{pred}(\text{succ } nv1) \rightarrow nv1}$$

$$\frac{t1 \rightarrow t1'}{\text{pred } t1 \rightarrow \text{pred } t1'}$$

$$\frac{}{\text{iszero } 0 \rightarrow \text{true}}$$

$$\frac{}{\text{iszero } (\text{succ } nv1) \rightarrow \text{false}}$$

$$\frac{t1 \rightarrow t1'}{\text{iszero } t1 \rightarrow \text{iszero } t1'}$$

Definition

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

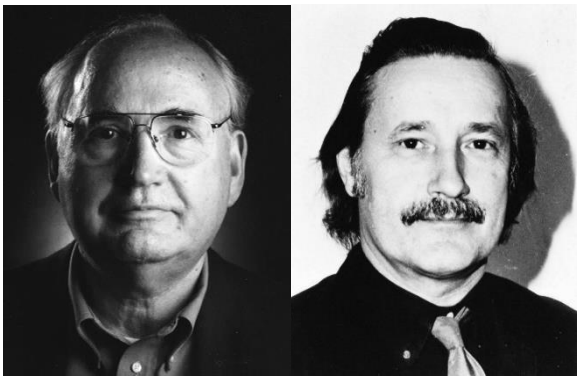
*further studies: middle weight Java and its Operational semantics.

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)**.



Dana Scott Christopher Strachey

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]]=𝕆
𝕆 is from the meta language (the mathematical language).
E[[1]]=𝟙
E[[nb]]=E[[n]]*𝟚+E[[b]]
$E[[e_1+e_2]]=E[[e_1]]+E[[e_2]]$

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) = 𝕆
...
E[[9]](S) = 𝟡
E[[nd]](S)=E[[n]](S)*10+E[[d]](S)
E[[e_1+e_2]](S)=E[[e_1]](S)+E[[e_2]](S)

E : Parse tree →(states →ℕ)

E([[e]],S)
E: parse tree → $\mathbb{N}^{\text{states}}$ 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 a meaning function
C[[x:=e]](s)=modify(s, x, E[[e]](s))
C[[P_1;P_2]](s)=C[[P_2]] (C[[P_1]](s))
C[[if e then P_1 else P_2]](s) = if E[[e]](s) then C[[P_1]](s) else C[[p_2]](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

$s_0(x) = 1$, $s_0(y)=2$
$s_1(x) = 1$, $s_1(y)=1$

$C[[\text{if } x>y \text{ then } x:=y \text{ else } y:=x]](s_0) = \text{if } E[[x>y]](s_0) \text{ then } C[[x:=y]](s_0) \text{ else } C[[y:=x]](s_0)$
$= C[[y:=x]](s_0)$
$= \text{modify}(s_0, y, E[[x]](s_0)) = \text{modify}(s_0,y,1) = s_1$

Example

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

$s_0(z) = 2$
$s_1 = \text{modify}(s_0,x,0)$
$s_2 = \text{modify}(s_1,y,0)$

$C[[P]](s_0)$
$= C[[\text{while } x<=z \text{ do } (y:=y+x; x:=x+1)]](s_2)$
$= \text{if not } E[[x<=z]](s_2) \text{ then } s_2 \text{ else } C[[\text{while } x<=z \text{ do } (y:=y+x; x:=x+1)]](C[[y:=y+x; x:=x+1]](s_2))$
$= C[[\text{while } x<=z \text{ do } (y:=y+x; x:=x+1)]](s_3)$
$= \ldots$
$= s'$ which s'(x)=3, s'(y)=3, s'(z)=2

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

Example

$C[[\text{while } x=x \text{ do } x:=x]] (s) = ?$
or
$C[[\text{while } x=y \text{ do } x:=y]](s) = s$ if $s(x) \neq s(y)$
undefined otherwise

Nonstandard Denotational 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 $\rightarrow$ { init , uninit }
states ={ { error } U { variables $\rightarrow$ { 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))

$s_1 * s_2 = \lambda v \in$ Variables if $s_1(v)=s_2(v)=$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]](s_0) = modify(s_0, x, init)

Session 9

IMPERATIVE AND DECLARATIVE

Mitchell's book chapter 4

There are four kinds of sentences in natural languages:

- Imperative
- Declarative
- Interrogative
- Exclamatory

Programming languages are one of the first two.

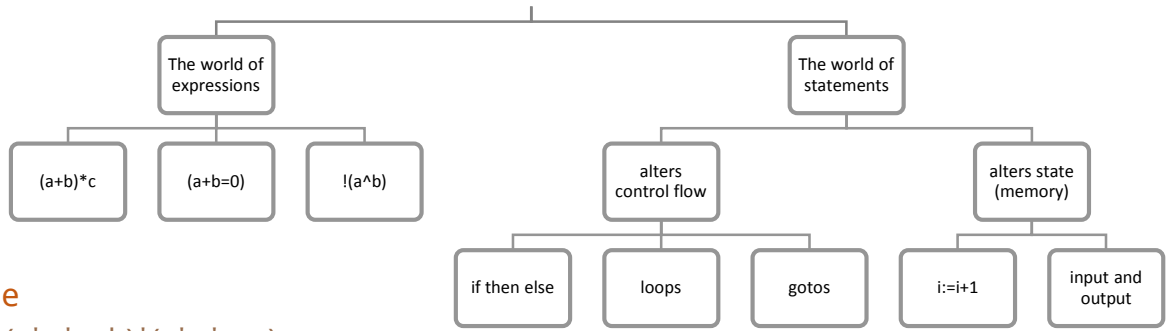
Functional language

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 $x_1,...,x_n$, all occurrences of an expression *e* containing only variables $x_1,...,x_n$ 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

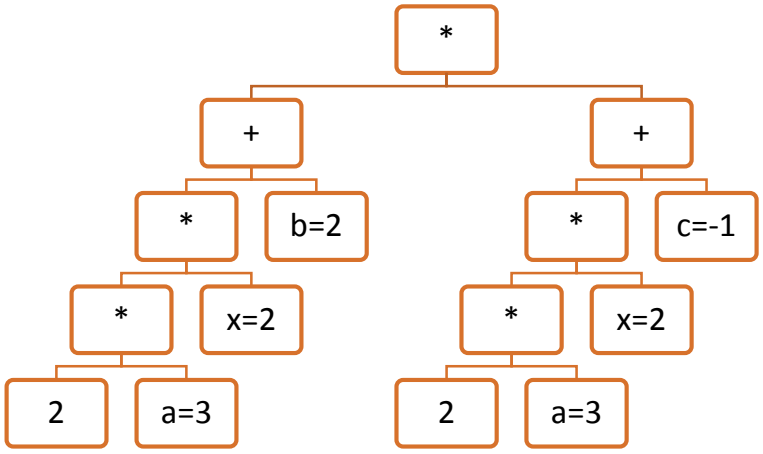
Church-Rosser Property (Confluence)

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.

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

variable abstraction application

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 a closed term or a cluster.

Closed terms are also called combinatory.

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

Operational Semantics

(λx.t)t' → [x↦t']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 an 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 → R^R 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

$C_0 = \lambda s.\lambda z.z$
$C_1 = \lambda s.\lambda z.sz$
$C_2 = \lambda s.\lambda z.s(sz)$
$C_3 = \lambda s.\lambda z.s(s(sz))$

scc = $\lambda n.\lambda s.\lambda z.s(nsz)$ successor
scc $C_n = \lambda s.\lambda z.s(s(s(s...(sz)))...)) = C_{n+1}$
plus = $\lambda m.\lambda n.\lambda s.\lambda z.ms(nsz)$
times = $\lambda m.\lambda n.m(\text{plus } n)C_0$
iszero = $\lambda m.m\ (\ \lambda x.\text{fls}\ )\ \text{tru}$

zz= pair $C_0\ C_0$
ss= $\lambda p.\text{pair}\ (\ \text{snd p}\ )\ (\ \text{plus}\ C_1\ (\text{snd p})\ )$
prd = $\lambda m.$ fst (m ss zz) predecessor

Session 12

Recursion

Fixed Point

f: A → A

Its Fixed point is $x \epsilon A : f(x)=x$

*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

$$f(n) = \begin{cases} 1 & if\ n = 0 \\ n \times f(n-1) & otherwise \end{cases}$$

or

$$f: n \mapsto \begin{cases} 1 & if\ n = 0 \\ n \times f(n-1) & otherwise \end{cases}$$

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

$$f': n \mapsto \begin{cases} 1 & if\ n = 0 \\ n \times f(n-1) & otherwise \end{cases}$$

The only fixed point of F is the factorial function.

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

* Omega = (λx. x x)(λx. x x) → diverge

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 fixed point combinator (Z combinator)

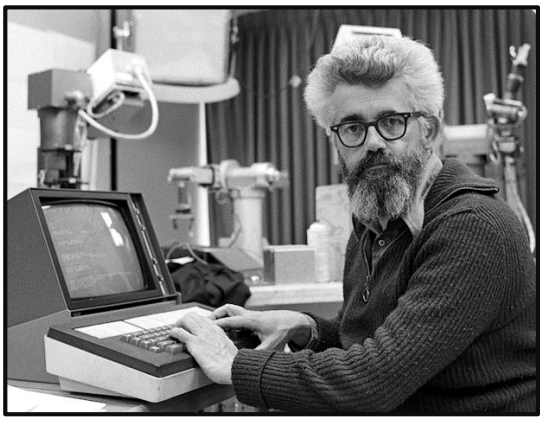
Z= λh. (λx.h (λy.xxy)) (λx.h (λy.xxy)) this is introduced by Gordon Plotkin

which ZF = F(ZF) (to prove this we need to accept λx.mx=m when m is not bind which is called Etha-equivalence.)

Session 13

SOME PROGRAMMING LANGUAGES

Lisp



Abbreviation of "List Processor"
or maybe "Lots of Infernal Stupid Parentheses"
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

+, - , *

The following functions make Lisp impure but without them it's pure functional.

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
```

Example

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 ) )
```

Session 14

Scheme

A newer version of Lisp

It has some differences from the old one.

The Lisp Abstract Machine

- A Lisp expression to be evaluated.
- A continuation (which is a function to determine where the output of a function goes).
- An association list (A-list) or run-time stack.
- A heap which is a set of cons cells (pairs stored in memory).

Example

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 ) ) :`

find x y

→ A-list : (x,y) : x=apple y= *pear peach apple banana fig*

→ equal y nil → nil →continuation → equal x (car y) → nil →continuation

→ find (x (cdr y))

→ A-list : (x,y) : x=apple y= *peach apple banana fig*

→ ....

Cons cells (doted pairs)

Address and **decrement** register in IBM 704 was the basic idea.

Every cons cell consists of an address and decrement .

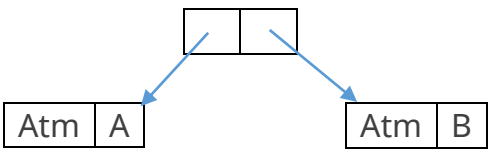
car: content of address register

cdr: content of decrement register

Atoms would be like this:

Atm	content

so (A.B) is like



Some functions

(atom v) : checks if v is an atom.
(eq A B) : checks if two cons cell are at the same block of memory.
(cons x y) : creates a cons cell of pointers to x and y and returns the pointer to the new cell.
(car x) : returns the content of address register of x.
(cdr x) : returns the content of decrement register of x.

Example

```
(( lambda (x) (cons x x) ) (cons 'A 'B))
```

The list (A B C) means (A.(B.(C.nil)))
So (A B) ≠ (A.B)
The first one is a list but the second one is a pair.

Example

```
> (set 'apostles '(Matthew Mark Peter))
   (Matthew Mark Peter)
> (cons 'John apostles)
   (John Matthew Mark Peter)
> (cons 'a nil)
   (a)
> (cons 'a 'b)
   (a.b)  // this is a cons cell
> (cons 'a '(b))
   (a b)   // this is a list which is (a.(b.nil))
> (cons '(ab) '(c (de)) )
   ( (ab) c (d e) )
> (cons '(c (de)) '(ab) )
   ( (c(de)) a b )
> (car '(abc))
   a
> (car apostle)
   John
> (cdr '(abc))
   (b c)
> (cdr apostle)
   (Matthew Mark Peter)
```

Caveat:

c d e is (c.(d.(e.nil)))
c (d e) is (c.((d.(e.nil)).nil))

quote command delays the calculation but **eval** does the opposite.

Session 15

An Example

To substitute expression x for all occurrence of y in expression z and evaluate the resulting expression. **[y↦x]z**

```
(define substitute (lambda ( exp1 var exp2)
      (cond ( (atom exp2) (cond (( eq exp2 var ) exp1 ) (true exp2) ))
            ( true (cons (substitute exp1 var (car exp2) )
                  (substitute  exp1 var (cdr exp2) ))))))
(define substitute_and_eval (lambda (x y z) (eval (substitute x y z))
> substitute '(t+1) 'y '(y^2-1)
'((t+1)^2-1)
```

*__eval__ function gives us the ability to run a code that is generated within the process.

Function Expressions

Recursive calls in Lisp was a great innovation.

An **anonymous function** is like this:

(lambda (<parameters>) <function body>)

Like in *(lambda (x) (+(square x) y))*

Now *((lambda (x) (+(square x) y)) 4)* gives us 16 + y

Recursion, The History

(f x) = (cond ((eq x 0)0) (true (+x (f(- x 1))))))

Now let's see how they've got here:

1. (lambda (x) (cond ((eq x 0)0) (true (+x (f(- x 1)))))))
2. (label f (lambda (x) (cond ((eq x 0)0) (true (+x (f(- x 1))))))))
3. (define f (lambda (x) (cond ((eq x 0)0) (true (+x (f(- x 1))))))))
4. (defun f(x) (cond ((eq x 0)0) (true (+x (f(- x 1)))))))

Higher-Order Functions

The functions introduced until now were all **First-order** functions.

A function that gets a first-order function as an input is called **Second-Order** function.

A function that gets a second-order function as an input is called **Third-Order** function.

Example

```
(fog)(x)  = f(g(x))
(define compose (lambda (f g) (lambda (x) ( f(g x) ) )) )
```

Example

A function to apply another function to all members of a list

```
(define maplist(lambda(f x)
      (cond((eq x nil)nil)(true(cons(f(car x)(maplist f(cdr x))))))))
> (maplist square '(1 2 3))
 (1 4 9)
```

Session 16

Garbage Collection

At a given point in the execution of a programme *P*, a memory location *M* is **garbage** if no completed execution of *P* from this point can access location *M*.
It was invented by **John McCarthy** in the late 60s.
Some languages are garbage-collected, like Lisp and some are not which user has to dealocate memory manually, like the old versions of C.

Example

```
(car(cons e1 e2)) // e2 is a garbage.
```

Mark_and_sweep garbage collection algorithm

1- Set all tag bits to 0.
2- Start from each location used directly in the programme. Follow all links, changing the tag bit of each cell visited to 1.
3- Place all cells with tags still equal to 0 on free list.

Google it for more info.

A comparison

A programme to find x in a list and return the elements that come after it.

in Lisp:

```
(define select (lambda (x lst)
     ( cond
          ((equal lst nil) nil)
          ((equal x (car lst)) (cdr lst))
          (true (select x (cdr lst)))
)))
```

in C:

```
typedef struct cell cell;
struct  cell {
     cell *car, *cdr; };
cell *select(cell *x, cell *lst) {
     cell *ptr;
     for(ptr=lst; ptr!=0;) {
          if (ptr->car == x) return (ptr->cdr);
          else ptr = ptr->cdr;
     }
}
cell *select_gtarbage_collected (cell *x, cell *lst){
cell *ptr, *previous;
     for(ptr=lst; ptr!=0){
          if(ptr->car==x) return (ptr->cdr);
          else previous=ptr;
          ptr=ptr->cdr;
          free(previous);
     }
}
```

Side Effects

```
(rplaca x y)//replace car of x with y
(rplacd x y)//replace cdr of x with y
```

Example

```
(lambda(x)(cons(car x)(cons(rplaca x c)(car x))))(cons a b)
                  a                        c
```

Example

to replace the third element of list (A B C D) with x

pure lisp : `(cons(car x)(cons(cadr x)(cons y(cdddrx))))`

impure lisp : `(rplaca(cddr x)y)`

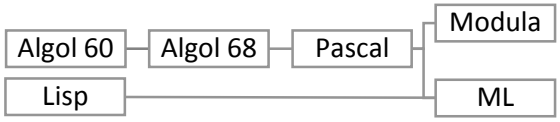
Session 17

The Algol Family and ML

Mitchell's book - Chapter 5

Pascal and C are some of the descendants of Algol.

ML is a combination of Pascal and Lisp.



Algol 60

New features:

- ;
- Blocks begin ... end
- Recursive functions
- Fewer constraints (i.e. A[i+1])
- Procedures with procedure arguments
- A primitive static type system
- BNF

Example

```
real procedure average (A,n);
      Real array A; integer n;
      Begin
            Real sum; sum:=0;
            For i:=1 step 1 until n do
                  sum:= sum+A[i];
            average := sum/n
      end;
```

Problems are:

- There are no array bounds
- Type discipline problems
- Parameter passing (pass_by_value and pass_by_name)

Example

```
begin integer i;
      integer procedure sum(i, j);
            integer i,j;
                  comment parameters pass by name;
            begin integer sm; sm:=0;
                  for i:=1 step 1 until 100 do sm:=sm+j;
                  sum:=sm
            end;
      print(sum (i , i*10))
end;
```

Algol 68

New features:

- Regular, systematic type system:

  Modes: 1. premetives: int,real,char,bool,string,complex,bit,byte,semaphore,format,file
  2. compound: array, structure, set, pointer
- Memory management by heap
- Pass by reference

Nonetheless, it was not successful :(

Pascal

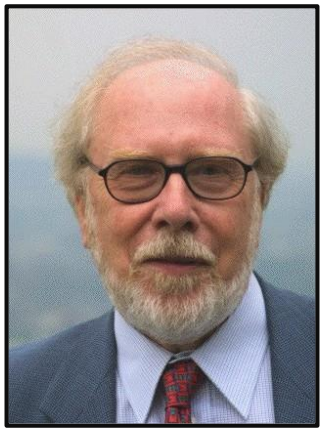
Niklaus Wirth originally designed Pascal for education.

New features:

- Record data structure (record is like struct in C)
- Subrange type (A[1..10])
- ...

Example

```
procedure DoSmthn (i,k:integer);
procedure DoSmthn (procedure p(i:integer),i,k:integer);
procedure notAllowed (procedure p(procedure d()));
array <index type> of <entry type>
procedure p(a:array[1..10] of integer)
```



Niklaus Wirth

Modula

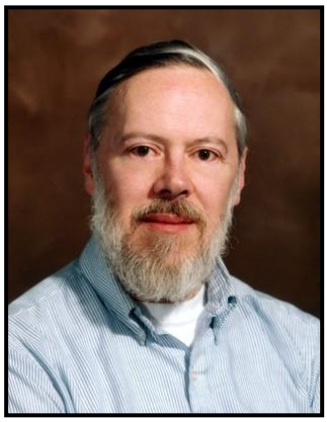
Niklaus Wirth had introduced this too.

C

The most important innovation of C is **pointer arithmetic**.

"C is quirky, flawed and an enormous success."

-Dennis Ritchie



Dennis Ritchie

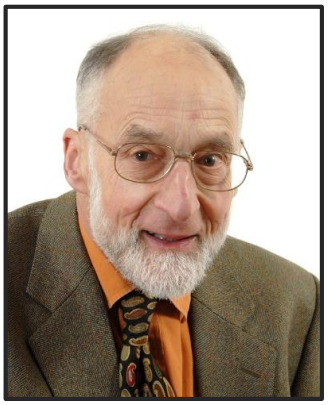
ML

...

Session 18

ML

A mostly functional language.
Is the first Language to have a **strong type system** and yet it's easy to use.
It's created from **Meta-Language.**
It was a foundation for the project LCF (Logic for Computable Functions) by Robin Milner so it has a **Mathematical foundation**.
Robin Milner won a Turing Award for introducing a safe language with **Exception** in 1991.
Curry-Howard isomorphism : **Logic = Programming Languages**
Proposition ↔ type
Proof ↔ programme
There are some Languages that are used to proof , like *coq* and *LF*.
OCAML is one of the descendants of ML. (supported by INRIA)



Robin Milner

Read-eval-print

```
> <expression>;
val it=<print_value> : <type>
```

1. Parse
2. Type check
3. Compile
4. Execute

```
>(5+3)-2;
val it=6 : int

>if true then 3 else false
Error: stdIn.1.1.20 ……. //Type error(branches must have the same type)
```

Declaration

```
> val <identifier> = <expression>;
val <identifier>=<print_value> : <type>

>val x=(5+3)-2;
val x=6 : int
```

*these declared variables can't have anything Assigned to them after declaration.

Function declaration

```
> fun <identifier>(<arguments>)=<expression>;Or
val <identifier>=fn <ar_type> → <result_type>
>fun f(x)=x+5;
Or
> val f=fn x => x+5;
val f=fn:int→int
```

creating types

int * int is a pair constructed from two int base types.

Data-types

unit

Value: ():unit //It has only one value. Like void in C.

*The evaluation of x=2 returns a unit.

bool

Values: true : bool false : bool

if a then b else c

false	false value
not	logical not
or / and	logical or / and (non short circuit (always evaluates both arguments))
orelse / andalso	logical or / and (short circuit)
true	true value
bool	type name

int

Values: 0,1,2,...,-1,-2, ...

Operations: +,-,*,div : int*int →int

string

Values: "some example for string": string

sub	accessing n-th character
chr	ascii to character
#"z"	character "z"
ord	character to ascii
char	character type name
substring	extract a substring
print	simple print (on strings)
^	string concatenation
= <>	string equality & inequality
size	string size
"..."	strings (with no interpolation of variables)
string	type name

real

Values: 1.0, 2.0 , 3.14 ...

* 4+5.1 is an error.

tuple

> (3,4,true)

val it=(3,4,true) :int*int*bool

> #2 (3,4)

Val it=4 :int

Record

> {first_name="Leonardo" , last_name="Da Vinci"};

val it= ... : {first_nam : string , last_name : string}

> #first_name (...);

val it="Leonardo":string

Session 19

list

>[1,2,3,4];
val it=[1,2,3,4]: int list;
>[fn x=>x+1, fn x=>x+2];
val it =[fn, fn] : (int→int) list
*All elements must have a same type.
Operations:
>3::nil;
val it=[3]:int list
>4::5::it;
val it=[4,5,3]:int list
>1::2;
Error: operator and operand don't agree

::	adding an element at the beginning (list cons) (return the new list (no side-effect))
find	find an element
app	for each element do something
exists	is the predicate true for an element
all	is the predicate true for every element
filter	keep elements (matching)
@	list concatenation
[a, b, c]	list constructor
length	list size
rev	reverse
a list	type name

Patterns

val <pattern>=<exp>;
<pattern>::=<id>|<tuple>|<cons>|<record>|<constr>
<tuple>::=(<pattern>,...,<pattern>)
<cons>::=<pattern>::<pattern>
<record>::={<id>=<pattern>,...,<id>=<pattern>}
<constr>::=<id>(<pattern>,...,<pattern>)

Example

>val t=(1,2,3);
val t=(1,2,3) : int*int*int
>val (x,y,z)=t;
val x=1 : int
val y=2 : int
val z=3 : int

>fun f(<pattern>)=<exp>;

>fun f(<pattern>)=<exp$_1$>| ... | f(<pattern>)=<exp$_n$>;//tries to match the arguments of *f* to the first expression, and if they could not be matched, it goes to the second and so on.

>fun length(nil)=0 | length(x::xs)=1+length(xs); //polymorphism

val length = fn : 'a list→int //'*a* is variable. ML can define a generic type like this.

>fun f(x,(y,z)) = y;

val f = fn : 'a * ('b * 'c) → 'b

>fun g(x::y::z) =x::z;

val g = fn : 'a list →'a list

>fun h ({a=x, b=y, c=z}) = {d=y , e=z};

val h=fn : {a: 'a, b: 'b, c: 'c} → {d: 'b , e; 'c}

>fun f(x,0) =x | f(0,y)= y |f(x,y)=x+y;

val f = fn : int*int →int

Data-type declaration

datatype <type_name> = <constructor_clause> |...|<constructor_clause>

<constructor_clause> ::=<constructor> | <constructor> of <arg_types>

Enumerated data-type

>datatype colour = Red | Blue | Green;

Tagged union data-type (Disjoint union)

*tagged union (⊎) is a kind of union of sets that every element will be tagged by its original set's name so there might be duplicate elements but with different tags.

$$A \uplus B = (\{A\} \times A) \cup (\{B\} \times B)$$

>datatype student = BS of name |MS of name*school | PHD of name*faculty;

>fun name(BS(n))=n | name (MS(n,s))=n | name (PHD (n,f))=n;

val name = fn : student →name

>val t=Bs('Ali');

Session 20

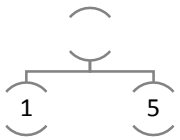
Recursive data-type

The set of binary trees with integer labels at the leaves.

```
>datatype tree=leaf of int | node of tree*tree
```

Example

```
node(leaf(1), leaf(5))
```




```
>fun inTree(x, leaf(y))=x=y | inTree(x, node(y, z))=inTree(x,y) orelse inTree(x,z) ;
```

*study recursive types in Pierce's book.

Reference types

L_Values and R_Values:

```
x: int;
y: int;
x:= y+3;
```

The location of x(L_Value) and the content of y(R_Value) is important in the last statement.

So **L_Values** are **references** and **R_Values** are **contents**.

```
>ref v    //creates a reference cell containing value V
>!r       //returns the value contained in reference cell r
>r:=v     //places a value V in reference cell r
```

Example

```
>val x=ref 0;
val x=ref 0:int ref
>x:=3*(!x)+5;
val it=():unit
>!x;
val it=5 : int
```

```
>val y= ref "apple";
val y=ref "apple" : string ref
>y:="something";
val it=():unit
```

*strings are different than other types like int so "y" is a reference to the address of "apple".

So in **ML** we **cannot** access the **address**.

Session 21

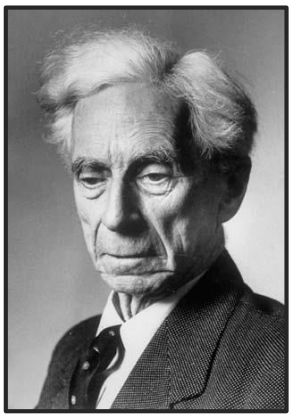
TYPE SYSTEMS AND TYPE INFERENCE

Mitchell's book - Ch6

Untyped λ-calculus has the same problem as **Russel's paradox**.

$$S = \{x | x \notin x\} \Rightarrow S \in S\ ?$$

For example, there can be values that are neither True nor False.


Bertrand Russell

Some Reasons to Have Types

1. Naming and organizing concepts.
2. Making sure that bit sequences in computer memory are interpreted consistently. (**Type errors**)
3. Providing information to the compiler about data manipulated by the programme. (**Optimisation**)

Having types is somehow a way to guarantee the safety of a language.

A Safe Language

A language that would protect its own abstraction.

A Type-Safe Language

A language that would protect its type distinction.

Type Errors

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

Hardware error

a machine instruction that results in a hardware error.

Example

x() is a hardware error if x is not a function.
float_add(3,4.5) is a hardware error. it makes a hardware interrupt.

Unintended semantics

Example

int_add(3,4.5) // This one won't make any hardware interrupts. It will consider the integer represented by the bit sequence of 4.5 and will add it to 3.

Types and Optimisation

Example

```
student={name: string, number: int}
undergrad={ name: string, number: int, year; int}
r.name //the compiler would replace it with the address r+1
```

Type Safety And Type Checking

Compile-Time and Run-Time Checking

Compile-Time Checking

Checks for type errors while compiling and returns errors.
ML and C are this type. (Java is mostly Compile time as well)

Run-Time Checking

When compiling creates a code to check the type on the run-time.
(car x) → 1.check that x is a cons cell
2.if OK ....

Conservativity of Compile-Time Checking

Example

```
if true then 2 else 2+true
//Although it would never go to the second statement, the compiler would give an error.
```

So Languages like this are **sound** and **conservative**.

Session 22

An Example of Having Types

Consider the following language.

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

In this language, terms would give a value or they would get stuck (like pred false).
So we need types.

Types

Bool - Nat

Typing relation

t:T

Now to define types:
T :: Bool | Nat

(*)

$$\frac{}{\text{True} : \text{Bool}} \qquad \frac{t1 : \text{Bool} \;\; t2 : T \;\; t3 : T}{\text{if } t1 \text{ then } t2 \text{ else } t3 : T}$$

$$\frac{}{\text{False} : \text{Bool}} \qquad \frac{t1 : \text{Nat}}{\text{succ } t1 : \text{Nat}}$$

$$\frac{}{0 : \text{Nat}} \qquad \frac{t1 : \text{Nat}}{\text{pred } t1 : \text{Nat}}$$

$$\frac{t1 : \text{Nat}}{\text{isZero } t1 : \text{Bool}}$$

Example:

$$\frac{\dfrac{\overline{0 : \text{Nat}}}{\text{isZero } 0 : \text{Bool}} \quad \dfrac{}{0 : \text{Nat}} \quad \dfrac{\overline{0 : \text{Nat}}}{\text{pred } 0 : \text{Nat}}}{\text{if isZero } 0 \text{ then } 0 \text{ else pred } 0 : \text{Nat}}$$

Definition-

The **typing relation** for arithmetic expressions is the smallest binary relation between terms and types, satisfying all instances of the rules in (*).

Definition-

A term *t* is **typable** (or well-typed) if there is some *T* such that *t:T.*

Theorem (Uniqueness of types)

Each term *t* has at most one type. (This is true in some languages like the one above)

Safety

Safety= Progress + Preservation

*The goal is to avoid terms that would get stuck.

Progress

A well-typed term is not stuck. i.e. either it is a value or it can take a step according to the evaluation rules.

$$t : T \Rightarrow t = v \ \vee \ \exists t'. t \to t'$$

Preservation

If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

$$t : T \wedge t \to t' \Rightarrow t' : T'$$

Theorem

The language of arithmetic expressions is **type-safe**, that is, progress and preservation hold in the language. (**type soundness**)

*A programme can be well-typed and a language can be type-safe.
*Type checkers are sound but not complete.
*A programme is ill-typed if the type checker cannot ...

Session 23

Type Inference (Type Reconstruction)

Deducing the type of an expression.

It was introduced by Robin Milner in ML for the first time.

Example

```
>fun f(x)=x;
val f=fn: 'a→'a
>fun f(x)=x+1;
val f=fn: int→int
```

Example

```
>fun f(g, h)=g(h(0));
val f=fn: ('a→'b)*(int→'a)→'b
```

The Type Inference Algorithm

1. Assign a type to the expression and each subexpressions.
2. Generate a set of constraints on types, using the parse tree of the expression.
3. Solve the constraints by means of unification.

Now we should find the most general unifier.

Currying

+: (int*int)→int

+: int→(int→int)

Example

(+2)3=5

Function Application

If the type of *f* is *a*, the type of *e* is *b*, and the type of *fe* is *c*, then we must have $a=b \rightarrow c$.

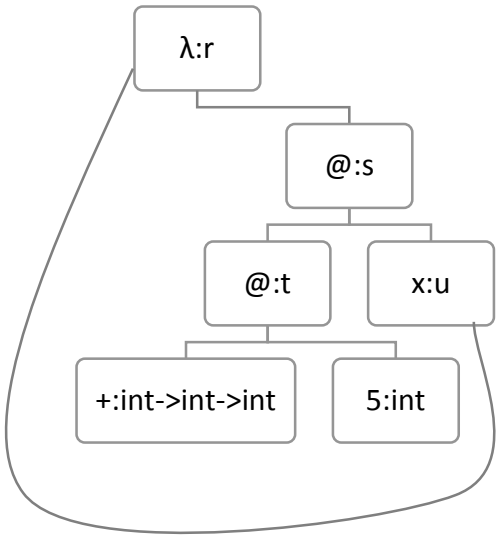
Lambda Abstraction

If the type of *x* is *a* and the type of *b* is *e*, then the type of $\lambda x.e$ must be equal $a \rightarrow b$.

Example 1

```
>fun g(x)=5+x;
val f=fn: int→int
```

parse tree:



Step 1:

λ: r ...

Step 2:

int→(int→int)=int→t

r=u→s

t=u→s

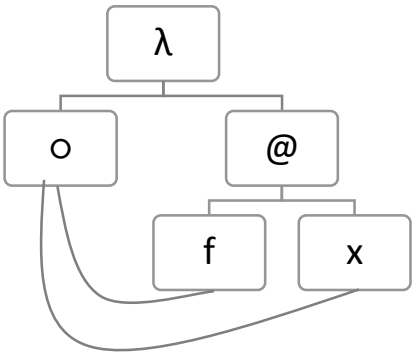
Step 3:

int→int=u→s➔u=s=int

result : r=int→int

Example 2 - A Polymorphic Function Definition

```
>fun apply(f,x)=f(x);
val apply=fn:('a→'b)*'a→'b
```



Step 1:

λ : t
○: r*s
@: u
f: r
x: s

Step 2:

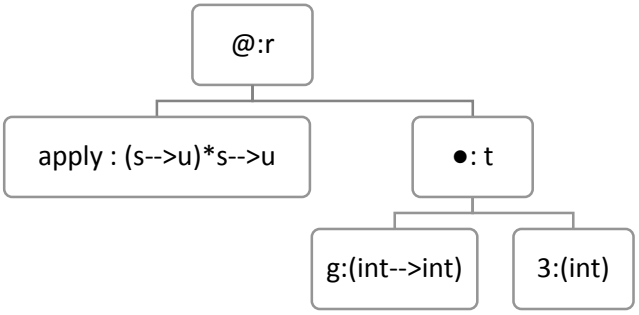
r=s→u
t=(r*s)→u

Step 3:

result: t=(s→u)*s→u

Example 3 - Application of a Polymorphic Function

```
>apply(g, 3);
```

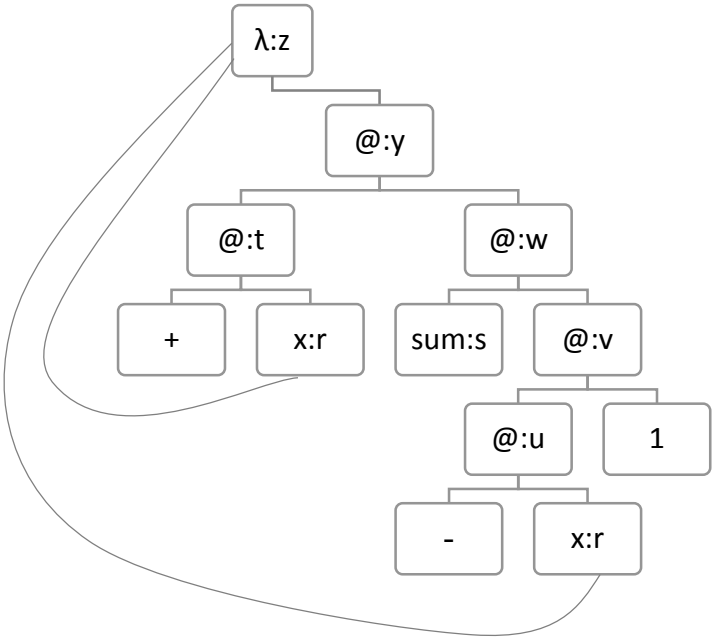


Step 3:

t=(int→int)*int
(s→u)*s→u=t→r
(s→u)*s→u=(int→int)*int
u=r
⇨ s=int, u=int , **r=int**

Example 4 - A Recursive Function

```
>fun sum(x)=x+sum(x-1)
```



Step 2:

int→(int→int)=r→u
u=int→v
s=v→w
int→(int→int)=r→t
t=w→y
z=r→y

Step 3:

r=int, u=int→int
v=int
s=int→int
t=int→int
w=int , y=int
z=int→int

*u=int→u is called a recursive type.

Session 24

Example 5

```
>fun append (nil, l)=l | append(x::xs, l)=x:: append(xs, l);
val append=fun: 'a list * 'a list → 'a list
```

result of the first inference - append : 'a list * 'b → 'b
result of the second inference - append : 'a list * 'b → 'a list
so *'b* should be *'a list*.

Example 6

```
>fun reverse(nil)=nil | reverse(x::lst)=reverse(lst);
val reverse=fun: 'a list→'b list
```

Polymorphism

- **Parametric Polymorphism**, in which a function may be applied to any arguments whose types match a type expression involving type variables;
- **Ad hoc Polymorphism (Overloading)**, another term for overloading, in which two or more implementations with different types are referred to by the same name;
- **Subtype Polymorphism**, in which the subtype relation between types allows an expression to have many possible types.

Subtype Polymorphism

x is the subtype of y and we can use it safely as y when:
x={int a; string b;} y={int a;}
Subclassing in OO is Subtyping.

Parametric Polymorphism

Strachey in Linguistics and Gerard in Logics had introduced it.
Functional languages use this type.
When the types in an expression are variable.

A sort function in ML using parametric polymorphism:

```
sort: ('a*'a→bool)*'a list→'a list
```

there are two kinds of parametric polymorphism:

- Explicit (C++)
- Implicit (ML)

Explicit Parametric Polymorphism (C++ Templates)

C++ makes **copies** of the code of this function for every call with a specific type. This happens in **linking time**.

```
template <typename T>
void swap(T &x, T &y){T tmp=x; x=y; y=tmp;}
```

Implicit Parametric Polymorphism (ML)

ML uses a different approach from C++.

Swap in ML

```
>fun swap(x,y)=let val tmp=x in x:=!y;u:=!tmp end;
val swap=fun: 'a ref*'a ref→unit
```

Sort in ML

```
>fun insert(less, x, nil)=[x]
     |insert(less, x, y::ys)=if less(x, y)
                             then x::y::ys
                             else y::insert(less, x, ys);
val insert=fun:('a*'a→bool)*'a*'a list→'a list
>fun sort(less, nil)=nil
     |sort(less, x::xs)=insert(less, x, sort(less, xs));
val sort=fun:('a*'a→bool)*'a list→'a list
```

Ad hoc Polymorphism (Overloading)

A symbol is overloaded if it has two or more meanings distinguished by type and resolved at compile time.
Like + on int or real or complex.

Session 25

Type Declaration and Type Equality

There are two types of type declaration.

- **Transparent type declaration**: an alternative name is given to a type.
- **Opaque type declaration**: a new type is introduced into the programme.

Transparent type declaration

In ML

```
type <identifier> = <type expression>
```

Example

```
>type Celsius=real;
>type Fahrenheit=real;
>fun toCelsius(x:Fahrenheit)=((x-32.0)*0.555556):Celsius;
val toCelsius=fun:Fahrenheit→Celsius
```

In C++

```
typedef char byte;
typedef byte tenBytes[10];
```

These are treated like in ML but the following is not.

```
typedef struct {int m;} A;
typedef struct {int m;} B;
A x;
B y;
x=y; → incompatible types in assignment.
```

Opaque type declaration

Example

A polymorphic datatype:

```
datatype 'a tree=LEAF of 'a | NODE of ('a tree*'a tree);
```

Session 26

SCOPE, FUNCTIONS AND STORAGE MANAGEMENT

Mitchell's book, Ch7

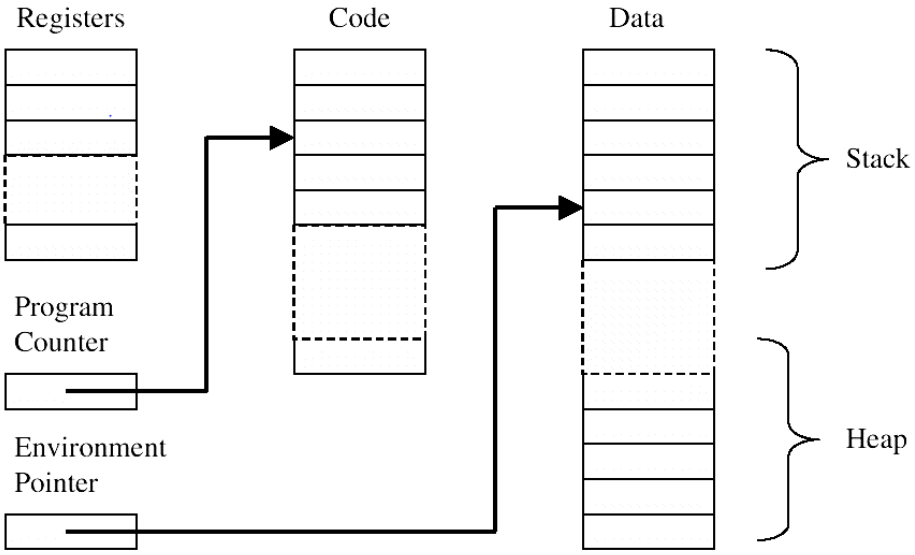
Block-Structured Languages

A **Block** is a region of programme text identified by *begin* and *end* markers.

Example

```
{
int x;
x=2;
}
```

Simplified Machine Model



In-Line Blocks

C: {...}

Pascal: begin ... end

ML: let ... in ... end

*Fortran in 60s and 70s wasn't block-structured.

*C and C++ don't support all of the features of block-structured languages.

Activation Records and Local Variables

Local variables which are stored on the stack in the **activation record** associated with the block.

Parameters to function or procedure blocks.

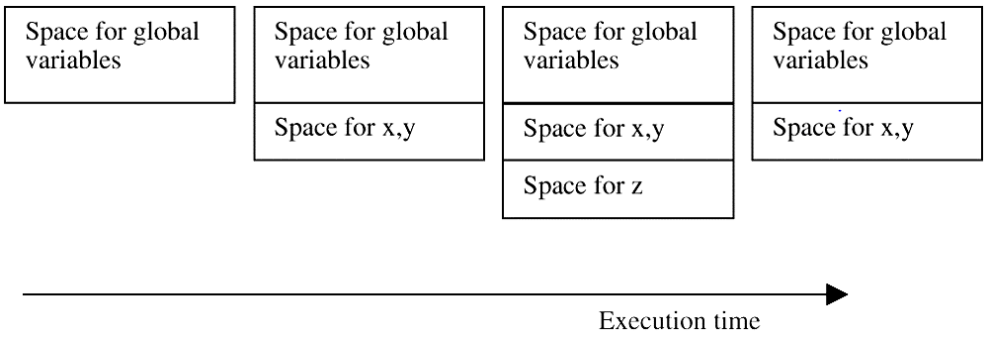
Global variables.

Stack discipline: pushing activation records into the stack and popping them out.

Example

```
{
int x=0;
int y=x+1;
 {
 int z=(x+y)*(x-y);
 };
};
```

the stack is like this:



Scope and lifetime

Scope: a region of text in which a declaration is visible.

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

Example

The difference is shown in this example.

```
{
int x=0;
…
 {
 int x=2;
 …
 };
};
```

the life time of the first x is the whole block but it's scope doesn't contain the inner block. The inner declaration of x hides the outer one so the inner block is called a **hole**.

Blocks and Activation Records for ML

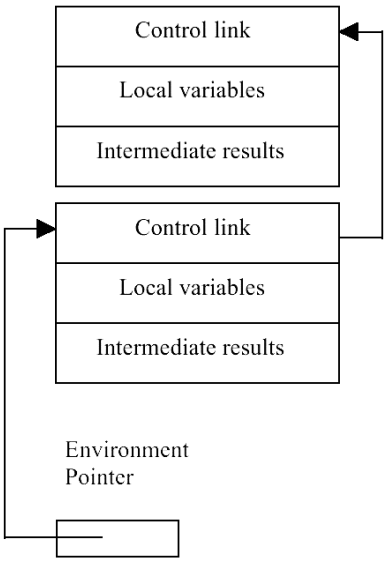
```
fun f(x) = x+1;
fun g(y) = f(y) +2;
g(3);
```

the declarations of the functions are considered separate blocks and have separate activation records. but the following code is all considered one block.

```
let    fun g(y)=y+3
    fun h(z)=g(g(z))
in
    h(3)
end;
```

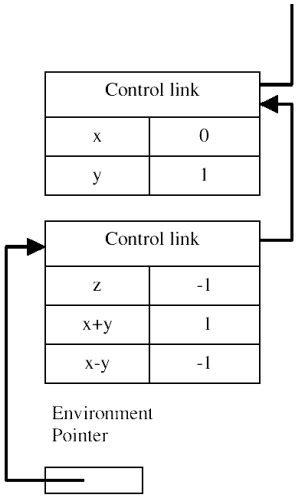
Global Variables and Control Links

control link is for retrieving the global variables.



Example

```
{
  int x=0;
  int y=x+1;
   {
      int z=(x+y)*(x-y);
   };
};
```



Functions and Procedures

in Algol like languages:

```
Procedure P(<parameters>)
begin
     <local variables>
     <body>
end
```

In C family:

```
<type> f(<parameters>)
{
     <local variables>
     <body>
}
```

*Procedures do not have return values so they must have side effects;

procedure are statement

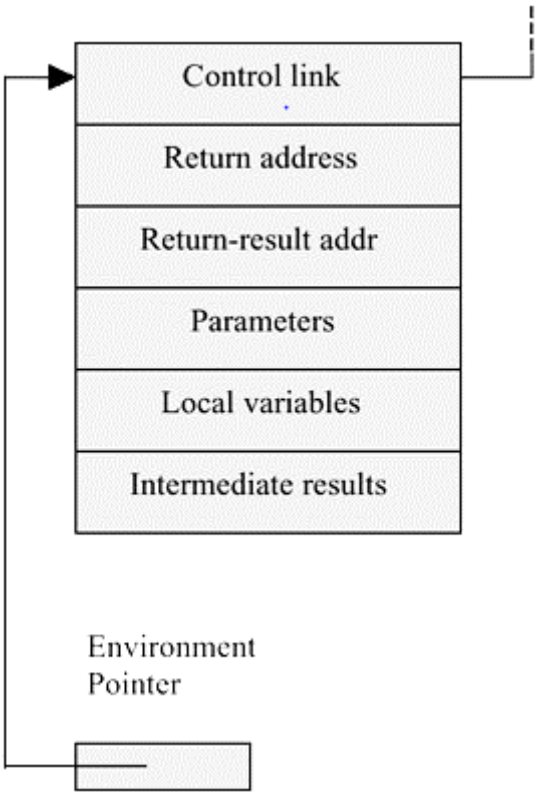
Function are Expression

Activation records for functions

- Parameters
- return values
- return address

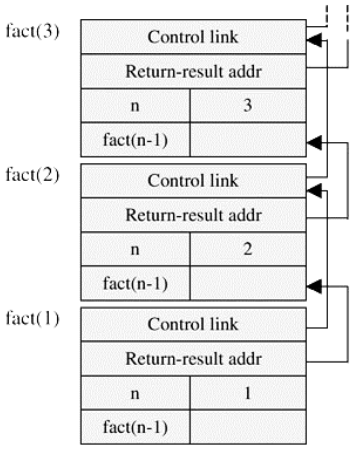
The activation record of a function must contain space for the following information:

- control link, pointing to the previous activation record on the stack.
- access link
- return address, giving the address of the first instruction to execute when the function terminates.
- return-result address, the location in which to store the function return value.
- actual parameters of the function
- local variable declared within the function
- temporary storage or intermediate result computed with the function executes



Example

```
fun fact(n)= if n<=1 then 1 else n*fact(n-1);
fact(3)
```



Session 27

Parameter Passing

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

Example

```
proc p (int x, int y) {
    if (x > y) then ... else ... ;
    ...
    x:= y*2 + 3;
    ...
}
p (z, 4*z+1);
```

In this example, x and y are formal parameters but z and 4*z+1 are actual parameters.

The main distinctions between different parameter-passing mechanisms are:

- the time that the actual parameter is evaluated
- the location used to store the parameter value.

The most common mechanisms to evaluate the actual parameter before executing the function body are:

- Pass-by-reference: pass the L-value (address) of the actual parameter
- Pass-by-value: pass the R-value (contents of address) of the actual parameter

The difference between pass-by-value and pass-by-reference

- **Side effects**: Assignments inside the function body may have different effects under pass-by-value and pass-by-reference.
- **Aliasing**: Aliasing occurs when two names refer to the same object or location. Aliasing may occur when two parameters are passed by reference or one parameter passed by reference has the same location as the global variable of the procedure.
- **Efficiency**: Pass-by-value may be inefficient for large structures if the value of the large structure must be copied. Pass-by-reference may be less efficient than pass-by-value for small structures that would fit directly on stack, because when parameters are passed by reference we must dereference a pointer to get their value.

Semantics of Pass-by-Value

```
function f (x) = { x := x+1; return x };
      .... f(y) ...;
```

If the parameter is passed by value and `y` is an integer variable, then this code has the same meaning as the following ML code:

```
fun f (z : int) = let x = ref z in x := !x+1; !x end;
      y = ref 10 ... f(!y)...;
```

*in x=ref z , x is the address to a cell containing the value z.

Semantics of Pass-by-Reference

```
function f (x) = { x := x+1; return x };
.... f(y) . . .;
```

If the parameter is passed by reference and y is an integer variable, then this code has the same meaning as the following ML code:

```
fun f (x : int ref) = ( x := !x+1; !x );
     y = ref 10 ... f(y) ...;
```

An Example in Algol-like Notation

```
fun f(pass-by-ref x : int, pass-by-value y : int)
    begin
       x:= 2;
       y:= 1;
       if x = 1 then return 1 else return 2;
    end;
var z : int;
z := 0;
print f(z,z);
```

Translating the preceding pseudo-Algol example into ML gives us

```
fun f(x : int ref, y : int) =
    let val yy = ref y in
        x := 2;
        yy := 1;
        if (!x = 1) then 1 else 2
    end;
val z = ref 0;
f(z,!z);
```

This code, which treats L-and R-values explicitly, shows that for pass-by-reference we pass an L-value, the integer reference z. For pass-by-value, we pass an R-value, the contents !z of z. The pass-by-value is assigned a new temporary location.

With y passed by value as written, z is assigned the value 2. If y is instead passed by reference, then x and y are **aliases** and z is assigned the value 1.

Another Example

Here is a function that tests whether its two parameters are aliases:

```
function (y,z){
   y:= 0;
   z:=0;
   y:= 1;
   if z = 1 then y:=0; return 1 else y :=0; return 0
}
```

Session 28

Global Variables

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

- **Static Scope**: A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text.
- **Dynamic Scope**: A global identifier refers to the identifier associated with the most recent activation record.

Dynamically Scoped	Statically Scoped
Older Lisps	Newer Lisps, Scheme
TeX/LaTeX document languages	Algol and Pascal
Exceptions in many languages	C
Macros	ML
	Other current languages

Example

```
int x=1;
function g(z) = x+z;
function f(y) = {
     int x = y+1;
     return g(y*x)
};
f(3);
```

if this is statistically scoped, in *g(12)*, x would be considered 1.
if it's dynamically scoped, in *g(12)*, x would use the changed value, which is 4.

Access Links Are Used To Maintain Static Scope

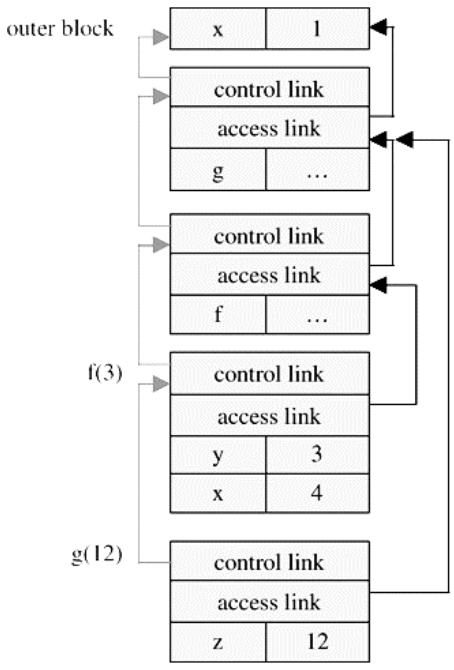
The **access link** of an activation record points to the activation record of the closest enclosing block in the program.

In In-line blocks, access link and control link are the same.

Example

in the previous example, the activation record would look like this.

*the block of function g goes to the end of the code.



Higher-Order Functions

First-Class Functions

A language has first-class functions if functions can be

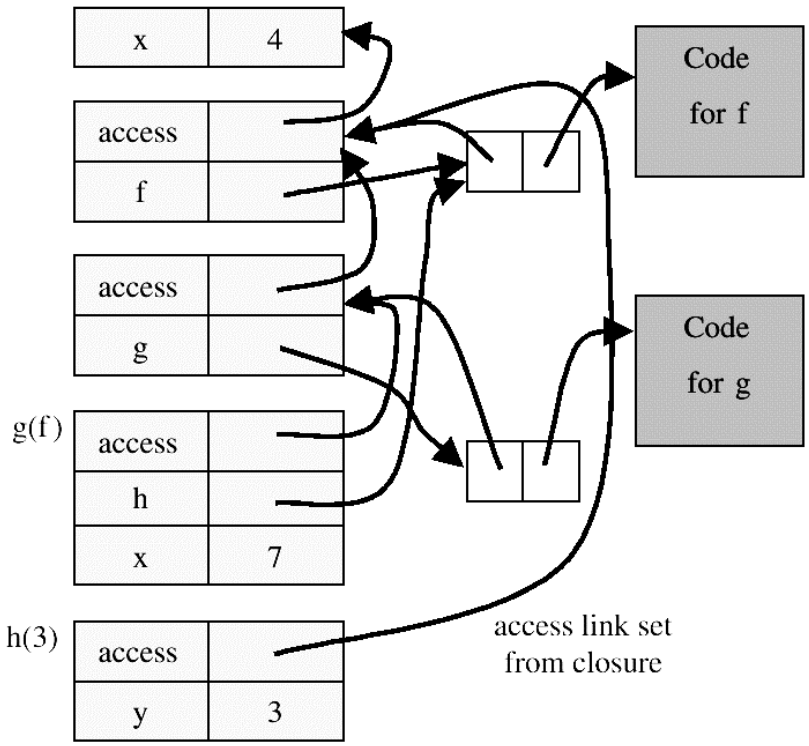
- declared within any scope,
- passed as arguments to other functions, and
- returned as results of functions.

In a language with first-class functions and static scope, a **function value** is generally represented by a **closure**, which is a pair consisting of a pointer to function code and a pointer to an activation record.

Passing Functions to Functions

Example

```
val x = 4;
fun f(y) = x*y;
fun g(h) = let val x=7 in h(3) + x;
g(f);
```



FINAL EXAM

Autumn1394

Problem 1

(a)

Find context free grammar for password consists of at least four character and they must have at least one digit. Our character set consist of {1, 2, .., 9, A, B, ..., Z, a, b, ..., z} and our digit set consist of {1, 2, 3, .., 9}.

```
<Password> ::= <Character><Character><Character><Digit> |
               <Digit><Character><Character><Character> |
               <Character><Digit><Character><Character> |
               <Character><Character><Digit><Character> |
               <Character><Password> |
               <Password><Character>
<Character> ::= <Digit> | A | B | .... | Z | a | b | ...  | z
<Digit> ::= 1 | 2 | 3 | ... | 9
```

(b)

Find context free grammar for context free grammars consists of rules separated by space and non of the rules consists of | in their definition. In this context free grammar our terminal consists of { 1, 2, 3, ..., ε} and out non-terminal consists of {A, B, ..., Z}.

```
<Grammar> ::= <Rule> | <Rule>_<Grammar>
<Rule> ::= <L-Side> = <R-Side>
<L-Side> ::= <Non-Terminal>
<R-Side> ::= <Non-Terminal> | <Terminal> | <Terminal><R-Side> | <Non-Terminal><R-Side>
<Non-Terminal> ::= A | B | ... | Z
<Terminal> ::= 1 | 2 | 3 | ... | 9 | ε
```

Problem 2

Following code written in semi-C language. Set parameter passing type for f, g and h function in order to have 29, 34 and 43 at the end of execution.

```
f(x) {return g(2*x);}
g(x) {let y = 1 in {h(y); return x + y + x;}}
h(x) {x = x + 5; return 0}
main() {printf(f(7));}
```

For having 29 at the end we use following pattern for functions parameter passing:

f	g	h
Call by value	Call by value	Call by value

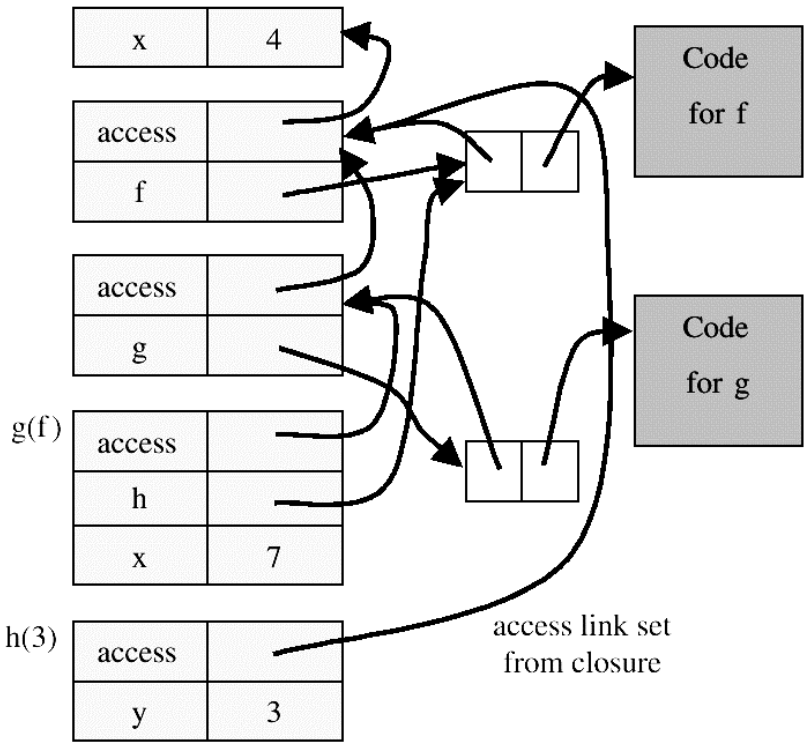
For having 34 at the end we use following pattern for functions parameter passing:

f	g	h
Call by value	Call by value	Call by refrence

Problem 3

Draw the activation recodes and stack for following code:

```
val x = 4;
fun f(y) = x*y;
fun g(h) = let val x=7 in h(3) + x;
g(f);
```

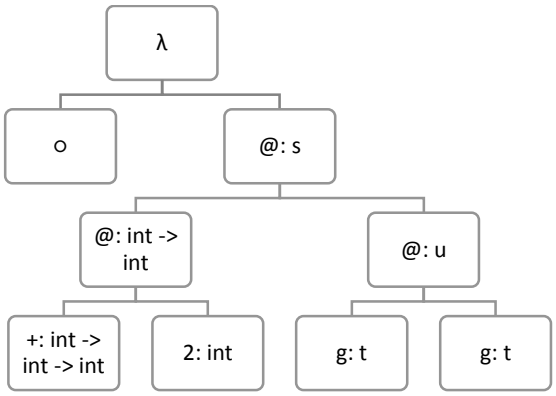


Problem 4

Express types of polymorphism and draw parse tree in order to find type of following expression:

```
f(g) = g(g) + 2;
```

- **Parametric Polymorphism**, in which a function may be applied to any arguments whose types match a type expression involving type variables;
- **Ad hoc Polymorphism (Overloading)**, another term for overloading, in which two or more implementations with different types are referred to by the same name;
- **Subtype Polymorphism**, in which the subtype relation between types allows an expression to have many possible types.



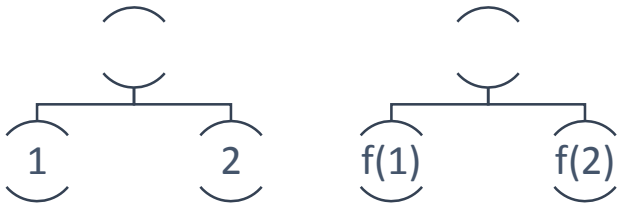
t: t -> u
int -> int: u -> s => u: int, s: int
=> t: t -> int => Type Error :(

Problem 5

The binary tree datatype

```
datatype 'a tree = LEAF of 'a |
  NODE of 'a tree * 'a tree;
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).
Write a function maptree that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if f is a function that can be applied to the leaves of tree t and t is the tree on the left, then maptree f t should result in the tree on the right:




```
fun reduce(f) = let
      fun apply(f, NODE(x, y)) = NODE(apply(f, x), apply(f, y))
      | apply(f, LEAF(x)) = LEAF(f(x));
   in
      fn (x) => apply(f, x)
   end;
```

Problem 6

This problem asks you to show that the ML types ’a → (’b → ’c) and (’a ∗ ’b) → ’c are essentially equivalent.

(a)

Define higher-order ML functions
Curry: ((’a ∗ ’b) → ’c)→ (’a → (’b→ ’c)) and
UnCurry: (’a → (’b → ’c))→ ((’a ∗ ’b) → ’c)

```
fun Curry(f) = fn (x) => fn (y) => f(x, y);
fun UnCurry(f) = fn (x, y) => f(x)(y);
```

(b)

For all functions f:(’a∗’b)→’c and g:’a→(’b→’c), the following two equalities should hold (if you wrote the right functions):
UnCurry(Curry(f)) = f Curry(UnCurry(g)) = g

شب تاریک و بیم موج و گردابی چنین هایل
کجا دانند حال ما سبـــــــکباران ساحل‌ها
...

پاییز۱۳۹۴
۹۲۲۱۰۶۴