1) John c. Mitchell, "Concepts in Programming Languages", Cambridge University press, 2002

2) Banjamin pierce, "Types and Programming Languages", MIT press, 2002

3) Keneth slonnege and Barry L. Kurtz, "Formal Syntax and semantics of programming Languages", Addison Wesley, 1995

Lisp, Algol, ML, C, simula, C++, Java

OCAML ← ML

- Programming Languages are the medium of expression in the art of Computer Programming.

- To write programs quickly vs. harder to design, easier to optimize vs. to make programming Cumbersome. Correctness vs. Complicated type.

Specific Themes:

- Computability (قابل محاسبه بودن)

Year: ------ Month: ------ Day: ------ Subject: ------

- static analysis (ایستا) (تحلیل استاتیک) → (تمام کارهایی که می‌خواهیم قبل از اجرای برنامه انجام دهیم (در Compile time))

Compile time Run time

↓ Conservative (محافظه‌کارانه)

مثلاً ما نمی‌توانیم یک متغیر را در زمان کامپایل بدانیم مقدار مورد نظر هست یا نه، در حالی که در زمان اجرا به راحتی می‌توان فهمید.

if (x=4)

در کل استاتیک دانش محافظه‌کارانه‌ای است یعنی هرگونه اشکال احتمالی را گوشزد می‌کند. در استاتیک فقط یک بار

syntax اشکالی

اشکال‌ها را رفع می‌کند ولی در زمان اجرا به تعداد اجراهای انجام شده عیب‌یابی و رفع اشکال صورت می‌گیرد.

- Expressiveness (بیانگری) (وضوح) vs. efficiency (بهره‌وری) (بازده) (سودمندی)

هر چه زبان ساده‌تر باشد، مفید بودن آن کمتر می‌شود (سطح بالاتر) (کارایی)

History:

1958 - 1960 : Algol, Cobol, Fortran, Lisp

1970 : methods for analyzing data

1980-1990 : Abstract data type

21st Century : - greater diversity (تنوع) computing devices
- Correctness
- Security

Definition:

- A Computational (محاسباتی) model is a collection of value & operations.

مجموعه‌ای از مقادیر و عملگرها است.

Value ← 2 + 3 → operation

معمولاً برنامه نویسی براساس یک نوع مدل محاسباتی است.

محاسبه

- A Computation is the application of a sequence of operations to a value to yield another value.

- A program is a specification of a computation.

توصیف

مثلا در Java یا ... داریم برنامه می‌نویسیم یعنی با آن مدل محاسباتی که آن محاسبه مورد نظر انجام می‌شود

- A Programming language is a ~~collection~~ notation for writing programs.

البته programming lang. فراتر از notation است. در واقع تعریف بالا خیلی کامل نیست.

در واقع programming lang. یک محدودیتی است روی specification ای که می‌خواهیم فراهم کنیم و امکان پیاده کردن آن به اجرا و محاسبه برسد را فراهم می‌کند.

پیچیده‌ترین و مهم‌ترین جزء زبان ← اجزای زبان چگونه کنار هم قرار می‌گیرند.

Language = syntax + semantics + Computational model + Pragmatics

این دوتا اهمیت زیادی دارند ولی آن دوتای دیگر را بعضی‌ها حذف می‌کنند و بعضی‌ها اعتقاد دارند.

وابسته به هر برنامه و محاسبه متناظر آن (ارتباط دهنده)

- یک معیار است که به برنامه‌ها معنی می‌دهد.

← نشان دهنده درجه موفقیت آن زبان است (میزان expressive بودن)

Computational Models:

سه نوع مدل محاسباتی به وجود آمده:

1) Functional 2) Logic 3) Imprative

(1) and (2): Declarative (خبری)

(3): (امری)

Functional:

value = Functional , operations = Function application

زبان‌های Lisp , ML و Functional هستند (ولی کاملاً Functional نیست)

(در اینها assignment نیست)

مثال: این رابطه در زبان Functional به شکل زیر می‌شود:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N} x_i^2}{N} - \left(\frac{\sum_{i=1}^{N} x_i}{N}\right)^2}$$

```
sd(xs) = sqrt(v)
    where
    n = length(xs)
    v = fold(Plus, map(Sqr, xs)) / n
        - sqr(fold(plus, xs) / n)
```

علامت "=" در بالا به معنی definition است. (assignment نیست)

که می‌توان def = فرض کرد

√ Logic :

human (Sucrates)

human (prolope)

mortal(x) if human(x)

¬ mortal(y)

1a: human (Sucrates) fact

1b: human (prolope) fact

2: mortal(x) if human(x) rule

3: ¬ mortal(y) assemption

4a: x=y 2,3, unification

4b: ¬ human(y) modes tollens

5a: y = Sucrates 1,4, unification

5b: y = prolope 1,4, unification

6: Contradiction

Value = fact and relation

Operation = inference rule

√ Imperative : Value = state , operation = transition (assignment)

$S_0 \xrightarrow{O_0} S_1 \xrightarrow{O_1} \cdots \xrightarrow{O_{n-2}} S_{n-1} \xrightarrow{O_{n-1}} S_n$

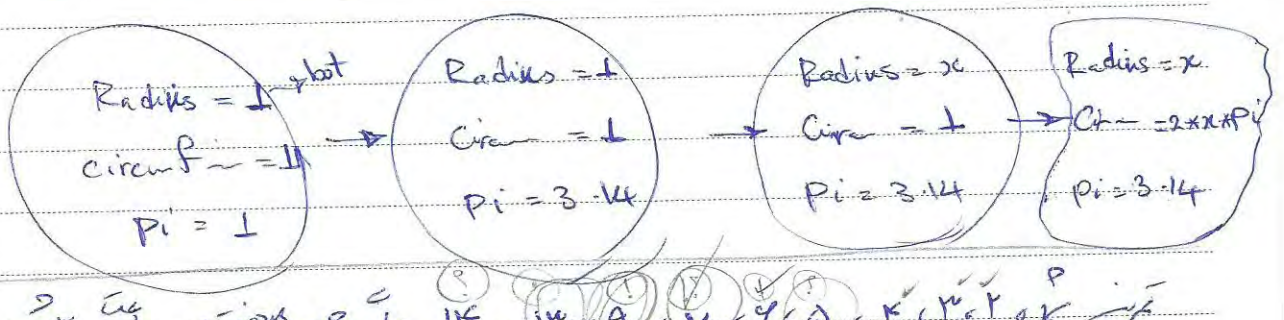
PL2 9031806 .pdf

Year: 2 Month: Day: Subject:

Constant pi = 3.14;

input (Radius); circumference := 2*pi* Radius;

output (circumference);



Syntax:

Grammar Symbol

valid phrase

Definition: A grammar (Σ, N, P, S) consists of four parts

1. A finite set Σ of terminal symbols, the alphabet of the language that are assembled to make up sentences in the language.

2. A finite set of nonterminal symbols, or syntactic categories, each of which represents some collection of subphrases of the sentences.

Year: ------ Month: ------ Day: ------ Subject: ------

3. A finite set p of production rule or productions that describe how each nonterminals is defined in terms of terminal symbols and nonterminal.

4. A distinguish nonterminal S, the start symbol, the specification the principle category being defined.

::= → is defined tobe

<declaration> → nonterminal

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

terminal, nonterminal

var x, y : int;

BNF (Backus – Naur Form)

vocabulary ، nonterminal symbol, terminal symbol

Type 3 , Type 2 , Type 1 , Type 0 .

√ Type 0 (Unrestricted grammar)

nonterminal ، production rule

MAHAN

$\alpha ::= \beta$

sequence of terminal & nonterminal

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

Type 0 grammar ⟷ (معادل هستند) Turing

هر زبان بدون قیدی، توسط ماشین تورینگ پذیرفته می‌شود.

✓ Type 1 (Context-Sensitive grammars) حساس به متن

قید می‌گذاریم به سمت راست production rules باید بزرگتر از symbol های سمت چپ باشد.

<thing> b ::= b <thing>

<thing> c ::= <another thing> d

Type 1 ⟷ LBA (Linear Bounded Automata)

✓ Type 2 (Context-free grammar) مستقل از متن

$\langle A \rangle ::= \alpha$ طرف چپ تنها یک nonterminal باشد.

Type 2 ⟷ PDA (Push-Down Automata)

برای برنامه‌ها برای Syntax استفاده می‌کنیم، مستقل از متن است. (BNF Grammar)

اگر در زبان Context-Sensitive بیاییم، می‌توانیم static semantics چک کنیم! (!؟) (ادامه)

✓ Type 3 (Regular grammars)

محدودترین زبان

$\langle A \rangle ::= a$

$\langle A \rangle ::= a \langle B \rangle$

<binary numerial> ::= 0

<binary numerial> ::= 1

<binary num> ::= 0 <binary num>

<bin num> ::= 1 <bin num>

regular B.N.F Grammars

Type 3 ⟷ FSA
(Finite state Automata)

Context-free grammars:

<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 | song | feather

<determiner> ::= a | the

<verb> ::= saw | sang | surprised | touched

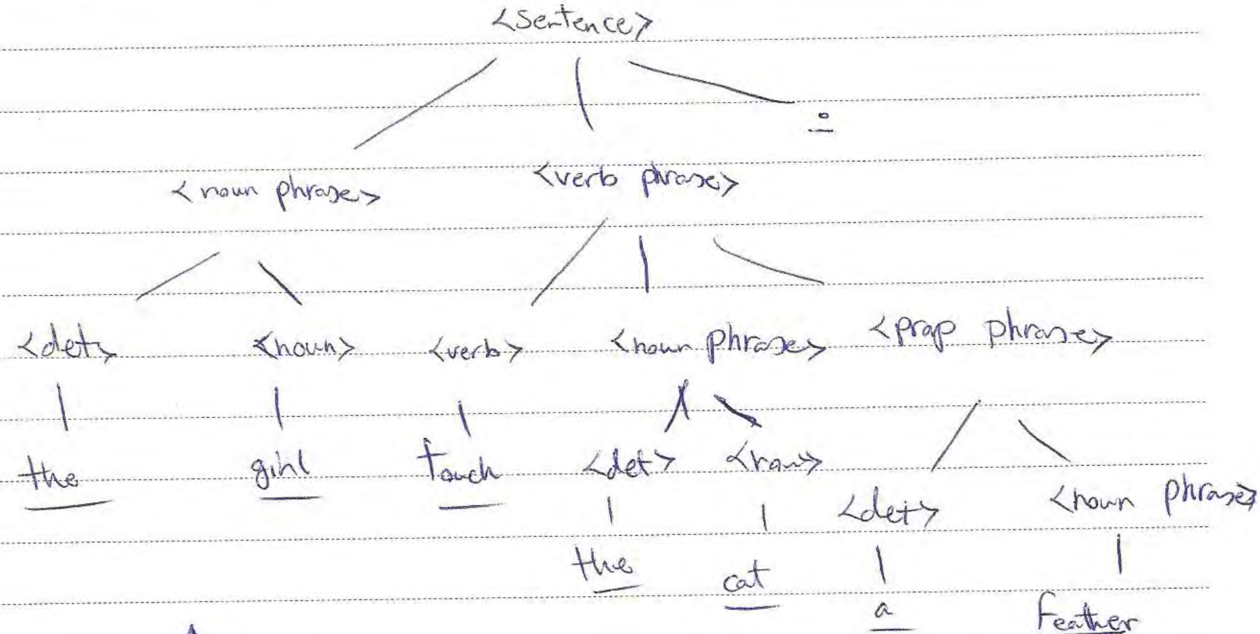
<preposition> ::= by | with

a boy surprised.

the girl with the cat sang feather.

MAHAN

Semantic grammar



a derivation tree for a sentence!

مثلاً برای جمله بالا دو تا درخت اشتقاق به دست می‌آید. چون دو تا معنی از آن برداشت می‌شود.

در اینجا یعنی آن گربه‌ای که پر داشت، دختر آن را دست زد. چون از سطح‌های پایین درخت شروع به معنی می‌کنیم.

یک مثال ابتدایی

ambiguity ← اگر در زبان ساخته شده، بیش از یک درخت اشتقاق داشته باشد ← ابهام

ولی در زبان برنامه‌نویسی نباید ابهام داشته باشد.

در زبان طبیعی ∞ جمله داشتیم چون loop (cycle) داریم! اگر در گراف وابستگی‌ها دور باشد، جمله‌ها نامحدود می‌شوند.

آن زبان می‌تواند ∞ valid phrase داشته باشد.

Definition - A grammar is ambiguous if some phrase

in the language generated by the grammar has more than one derivation tree. ... ...

Context – sensitive grammars:

<sentence> ::= abc | a <thing> bc
<thing> b ::= b <thing>
<thing> c ::= <other> bcc
a <other> ::= aa | aa <thing>
b <other> ::= <other> b

<sentence> ⇒ a <thing> bc ⇒ ab <thing> c ⇒ ab <other> bcc ⇒

a <other> bbcc ⇒ aabbcc

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

iwren :

Control structures → if, while

typed → boolean, integer ⇒

terminal symbol → reserved words, +, −, ÷, ٠, ١, ٢, ..., ٩

MAHAN

(empty string)

Syntax برای پیاده‌سازی مفسر شامل دو بخش است:

Two parts of Syntax
- → lexical syntax → token (لغوی) (lexer)
- → phrase-structure syntax → tree (parser)

token کوچکترین جزء زبان هستند. lexer (تحلیلگر لغوی) (lexer، tokenize می‌کند)

مثلاً token های wren به شرح زیر است:
<token> ::= <identifier> | <numeral> | <reserved word> | <relation> | <weak op> | <strong op> | (|) | := | , | ; | :

lexer متن text را به دنباله‌ای از token ها با یک tag برای هر token تبدیل می‌کند

(اولین گام برای مفسرین یا مترجم‌ها). lexical syntax کمکی است به token ها را تعریف می‌کند

بخش اصلی syntax، phrase-structure syntax است که phrase ها تعبیر را می‌دهد.

parser: syntactic analyzer
parser رشته token ها را می‌گیرد و درخت را می‌سازد.

شکل 1.8 کتاب formal syntax، BNF زبان wren است.

<program> ::= program <identifier> is <block>

<block> ::= <decleration sequence> begin <command seq> end

<declaration seq> ::= ε | <declaration> <declaration seq>

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

Year: ------ Month: ------ Day: -------

Subject: ----------

<type> ::= integer | boolean

<variable list> ::= <variable> | <variable> , <variable list>

<command seq> ::= <command> | <command> ; <command seq>

<command> ::= <variable> := <expression> | skip | read <variable> | write <integer expr> | while <boolean expr> do <command seq> end while | if <boolean expr> then <command seq> end if | if <boolean expr> then <command seq> else <command seq> end if

<variable> ::= <identifier>

<relation> ::= <= | => | < | > | =

<weak op> ::= + | -

<strong op> ::= * | /

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

<letter> ::= a | b | ... | z

<numeral> ::= <digit> | <digit> <numeral>

<digit> ::= 0 | 1 | ... | 9

MAHAN

۷) زبان wren ، ابهام ندارد.

مسئله در Pascal : <command> ::= if <boolean expr> then <command>

| if <boolean expr> then <command> else <command>

این کار مبنای ابهام است :

if expr1 then if $expr_2$ then cmd1 else cmd2

ولی pascal هم قانون گذاشته است که همیشه else به نزدیکترین if برمی‌گردد.

ولی در wren چون end if دارد ، پس ابهام را رفع کرده است.

۷) اگر می‌خواهیم عملیاتی strong تر باشد (strong operation) ، طوری لایه‌بندی درخت می‌کنیم که در سطح‌های پایین‌تر باشند (اونی که پایین‌تره ، اولویتش بالاتره) مثلاً در شرایط برابری درجه هم باید لایه‌بندی ابهام نداشته باشد

2+3+5 را در درخت به دو حالت که هر دو ابهام می‌دهد. (در اینجا ابهام هست ولی مهم نیست)

2+(3+5) و (2+3)+5

Context Constraints in Wren:

با اندازه‌ای از تست دو مرحله Syntax (lexical و phrase-structure) ، هنوز نمی‌توان گفت برنامه معتبر است.

(مثال)

آیا این illegal بودن برنامه زیر ، به مفهوم syntactic است یا semantic ؟ Program is illegal

var a := boolean ;

begin

a:=5

end

{ به عده‌ای می‌گویند semantic است چون از لحاظ معنایی مشکل دارد

" syntactic است چون همین طور باید تعریف کنیم ، در runtime متوجه نشویم

۸) اگر می‌خواستیم زبان را با گرامرهای BNF جوری تعریف کنیم ، این مشکل را می‌شد در گرامر حل کرد. (چون باید گرامر بتواند خطاهای ...)

Year: ------ Month: ------ Day: ------ Subject: ------

ولی به دلایلی که بعداً خواهیم گفت، از الگوریتمهای ... به متن استفاده می‌کنیم. این کار را چون می‌توانیم به صورت استاتیک بفهمیم پس (اشکال)

static semantics می‌گوییم، در حالی که در runtime بفهمیم را بهش semantics می‌گوییم!

این قسمت در اصل syntactic هست اما چون نمی‌توانستیم از روش‌های مستقل از متن استفاده کنیم پس به آن static semantic می‌گوییم

می‌کنیم. می‌توان به این قسمت Context Condition گفت. این قوانین را برای جلوگیری از چنین اشکالاتی می‌آوریم (مثل ۱.۱۱ کتاب)

زبان‌هایی که Context-sensitive هستند را می‌توان به صورت غیر از BNF نوشت.

این نقطه خیلی واضح clear نیست

Compile | Runtime

lexical analysis, syntactic analysis (parser)

Type checking (static semantics)

Context Constraints

می‌توان خیلی از کارهای static semantics را در parser چک کرد. ولی دیگر این standard BNF نخواهد بود.

در زمان اجرا چون به بعضی چیزها دسترسی داریم برنامه دقیق‌تر عمل می‌کنیم.

Semantic Errors:

یک اجرا از برنامه تمام می‌شود ولی نتیجه specify شده را نیاورده باشد. (خیلی به programmer مربوط است)

یک سری requirement خواسته شده بود که در برنامه specify نکردیم، در این حالت Error نمی‌شود پیدا کرد.

بعضی‌ها اعتقاد دارند که برنامه اگر terminate نشد Error دارد.

- An attempt is to divid by zero.

یا مثلاً null pointer که در جاوا در طول اجرا exception می‌دهد که بعضی‌ها compile time

کشف شوند. یا مثلاً دوباره start کردن یک thread.

این خطاها در Runtime است پس چطور می‌توان در Compiletime چک کرد!؟ این‌ها را تشخیص بدن

در بین Compile time و Runtime است! static checking بسیار بسیار سخت‌تر از Runtime checking است.

بعضی‌ها را Runtime checking است.

- A variable that hasn't been initialized!

- An iteration Command doesn't terminate!

در مورد سینتکس زبان اولیه که بررسی می‌کنیم:

exp::= num | lit | id | LP exp RP | exp ADD exp | exp CAT exp | exp MUL exp |

VB exp VB | LET id BE exp IN exp

num ::= Num[n] (n is a natural number)

lit::= LIT[s] (s is a string)

id::= ID[s]

تعیین کنیم عبارت زیر چیست:

Num[1] ADD Num[2] MUL Num[3]

در اینجا ambiguity موجود است. در اینجا ۲ نوع درخت اشتقاق می‌توان کشید.

تعریف exp بصورت flat (تک سطحی) است که باعث این ابهام شده است. ممکن است بگویند priority دارد که

بنابراین اولویت‌بندی در syntax آورده نشده. پس به ناچار «ابهام» خواهیم داشت.

Subject (Formal Syntax & semantics ...)
Date

non-Terminal

fct ::= num | lit | id | LP exp RP

trm ::= fct | fct MUL fct | VB fct VB

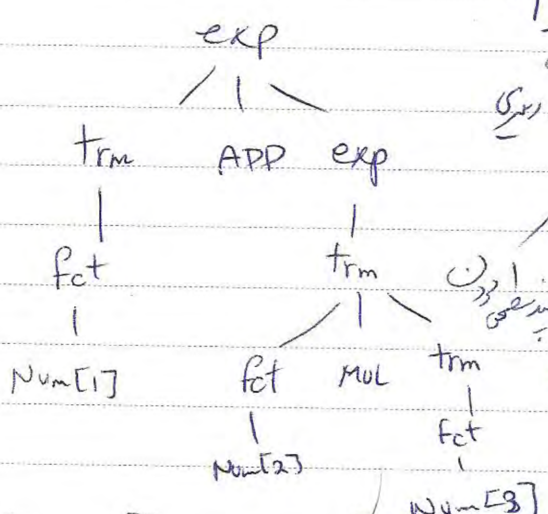
exp ::= trm CAT exp | trm | trm ADD exp

prg ::= exp | LET id BE exp IN prg

از گرامر بالا fct بیشترین اولویت را دارد! و prg کمترین اولویت.
چون در trm از fct استفاده شده

ترتیب اولویت: prg < exp < trm < fct

حال بر اساس گرامر فوق ها عبارت مقابل را رسم می کنیم:



با این گرامر که درخت فوق هیچ درخت دیگری
قابل دریافت نیست!
(ابهام برطرف شد، با روش چند سطحی اولویت)

تمرین:

Page 8 : 1, 3, 5, 6, 7, 9
Page 16 : 2, 3, 5, 6, 9, 10, 11
Page 20 : nothing
Page 29 : 2, 3

Attribute gr

این ها مربوط به کتاب syntax ، interp
برای هم همه اینها

sebesta - chapter 3 - 19 & 20 context free, context ... 3, 2 ; 29p ; 11, 10, 9, 3 ; 16p ; 9, 7, 6 ← 8p تمرین های تیپیره
check

Subject

Date

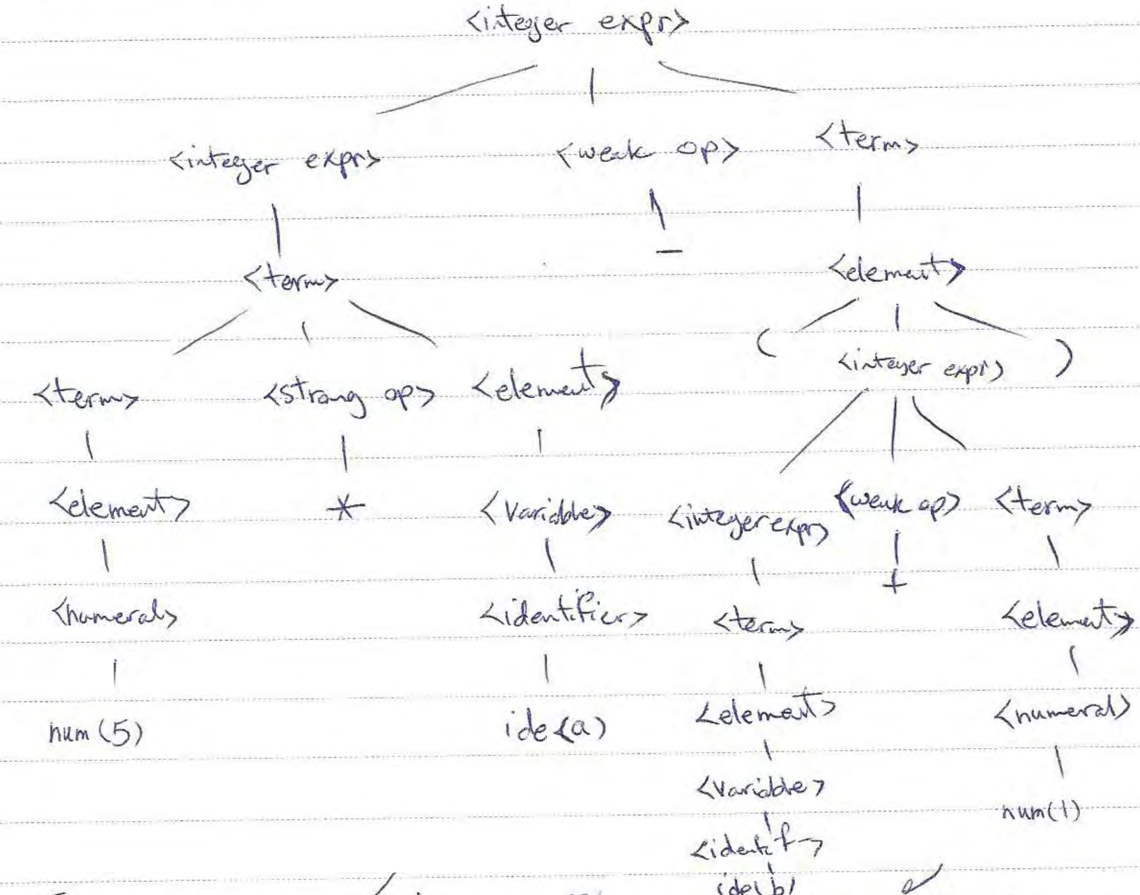
Abstract Syntax:

(Abstract ≠ Concrete)

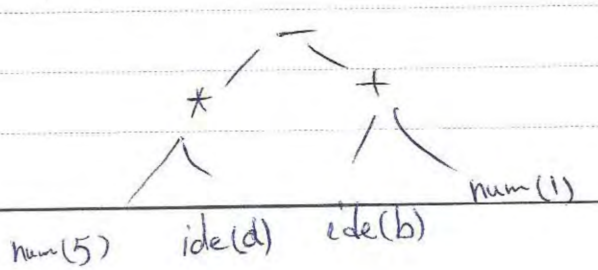
چیزی که تا الان کار کردیم Concrete syntax (عینی) بوده است. یعنی: physical text ها را داشتیم.

Abstract یعنی مفهوم خارجی ندارد. مفاهیم ریاضی abstract هستند.

$5 * a - (b+1)$



درخت بالا و گره‌هایش همه لازم هستند ولی درست و بدیهی هستند، چیزی که لازم داریم و کافیست این است ↓

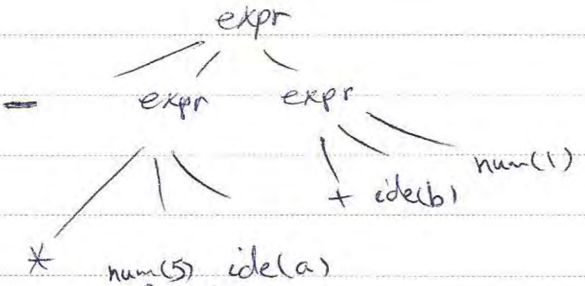


12 = 13

Subject
Date

{ حل تمرین ← شنبه‌ها
{ دوشنبه‌ها ← OCAML

یا به شکل :



در واقع ما یک درخت abstract می‌سازیم . مثلاً Compiler برای تبدیل نیاز به درخت پارس نیاز به جزئیات درخت

اول ندارد . در واقع بخش‌هایی از آن مدنظر که مورد استفاده است . (هدف abstraction)

اگر یک درخت اشتقاق به ما نشان دهد چند عبارت باشد abstraction است . مثلاً حتی در Wren یک Concrete است

اگر عبارتی را بنویسیم space بزنیم یا space نزنیم درخت اشتقاق ما نشان خواهد داشت لیکن برای لغت

abstraction است . اما هنوز در آن abstraction می‌سازیم operator ها را که جای بهتری است در

سمت ریشه root نگه داریم و زوائد را حذف کنیم و عناصر مهم را نگه داریم . (یک قاعده مشترک است)

با این یک درخت ترتیب syntax tree ایجاد می‌کنیم ممکن است چندین Abstract syntax باشد .

تعدادی operation و تعدادی argument داریم .

Abstract non-terminals , Abstract productions

در زبان Wren :

expression → numeral, boolean constant (true, false), identifier, operations

Subject Abstract Nonterminal
Date

Expression, Numeral, Identifier

<expr> ::= <integer expr> | <boolean expr> |

<integer expr> ::= <term> | <integer expr> + <term> | <integer expr> − <term>

<term> ::= <element> | <term> * <element> | <term> / <element>

<element> ::= <numeral> | <identifier> | (<integer expr>) | − <element>

⋮

<comparison> ::= <integer expr> <= <integer expr> |

⋮

| <integer expr> <> <integer expr>

<expr> ⇒ <integer expr> → <term> ⇒ <element> ⇒ <numeral>

این non-terminal ها را نیازی نداریم.

Unit Rule (قواعد واحد) ها را می‌توانیم برداریم مثلاً ... در Basic Comparison داشته باشیم.

مثل numeral ها، identifier ها و

essential non-terminals مثل + ، — و ...

Subject

Date

بگذار پذیرش قواعد واحد :

<integer expr> + <term>

<integer expr> − <term>

<term> * <element>

<term> / <element>

<numeral>

<identifier>

−<element>

⋮

<integer expr> <= <integer expr>

⋮

<integer expr> <> <integer expr>

مثال : مجموعه non Terminal های Abstract را به تعداد دلخواه تعریف می‌کنیم را می‌آوریم

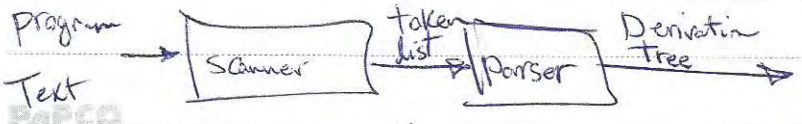
Expression ::= Numeral | Identifier | true | false | Expression operator Expression
 | − Expression

Operator ::= + | − | * | / | V (or) | Λ (and) | < | <= | = | >= | <>

این Abstract Syntax ها Syntax نیست ↑

می‌توان Semantics را برای هر گرامری برد به همین Abstract Syntax رجعت شده و تعریف کرد ولی ابهام را از بین می‌برد

Scan : character* → Token* Pars : Token* → Derivation tree



Parse o Scan
(fog)

منظور از parse tree همان Derivation tree است که نمایش از Concrete Syntax می‌باشد.

نمایش از Abstract Syntax است. (البته parse درخت Concrete باشد یا [illegible])

parse یک به یک نیست اما abstract یک به یک خواهد بود (یک به یک خواهیم)

a+b−c
(a+b−c) → یک درخت اشتقاق مشترک

این مفهوم با مفهوم ابهام ambiguity متفاوت است. (اشتباه نشود در سوال درخت ابهام)

Abstract Syntax ذهنیت Concrete Syntax ابزار (physical). ولی در خروجی Parser فقط یک درخت خواهیم داشت.

پس جامع‌ترین Concrete Syntax گرامری در مورد Semantics باید.

Attribute Grammar:

جداول اعتقاد داریم به ازای بعضی از انواع گرامر BNF به همراه جزئیات اضافی، گرامر context-sensitive را handle کرد. (static semantics)

گرامری است که در قالب Context-free نوشته می‌شود ولی بعضی attribute ها به آن اضافه می‌شود.

این ایده اولین بار توسط Knuth مطرح شد. عناصری که اضافه می‌شود

1) Attributes → عنصر اول که اضافه می‌کنیم. به دو دسته‌اند: ترمینال‌ها و نان‌ترمینال‌ها

2) Attribute Computation functions → ها Rule وصف می‌شوند (Semantic function) و مشخص می‌کنند که attribute ها چطور استخراج می‌شوند.

Subject

Date

3) Predicate Functions → هر یک یک مقدار صحیح یا غلط را از این متغیرهای دیگر برمی‌گردانند و کارشان این است که static semantic rules را بیان می‌کنند.

grammar symbol $X \rightarrow A(X)$ of attributes

هر سمبل به یک مجموعه از ویژگی‌ها مربوط.

$$A(X) = S(X) \cup I(X)$$

$$S(X) \cap I(X) = \emptyset$$

[illegible]

Synthesized — سنتز شده

Inherited — به ارث برده شده

در parse tree، اگر X ظاهر شود، ویژگی‌های synthesized با استفاده از بچه‌ها محاسبه می‌شود.

ویژگی‌های Inherited با استفاده از پدر یا برادر خواهر محاسبه می‌شود.

$$X_0 ::= X_1 \dots X_n$$

↓

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

$$I(X_j) = f(A(X_0), \dots, A(X_n))$$

داریم ↓

$$I(X_j) = f(A(X_0), A(X_1), \dots, A(X_{j-1}))$$

- A predicate function is in the form of boolean expression on the union of the attribute sets $\{A(X_0), A(X_1), \dots, A(X_n)\}$ and a set of literal attributes value.

Subject
Date

در ابتدا درخت parse مقداردهی اولیه و کم کم مقادیرش محاسبه می‌شود و درختی که تمام مقادیرش محاسبه شده باشد،

درخت fully attributed گفته می‌شود.

intrinsic attribute → مقدار برگ‌ها را که مقادیرشان از جدول سمبل‌ها از آن محاسبه می‌شود

به آن برگ‌ها ویژگی ذاتی گفته می‌شود. از جنس synthesized هستند.

مثال) <proc_def> ::= Procedure <proc_name>[1] <proc_body> End

<proc_name>[2]; ← ADA (زبان)

Predicate : <proc_name>[1].string == <proc_name>[2].string

دقیقاً رشته کاراکتری است که ظاهر می‌شود.

در هنگام ساختن درخت، Compiler باید این شرط را چک کند. ← این یک attribute است.

پس هر <proc_name> ویژگی string به آن چسبیده است. synthesized attribute

هستند برای <proc_name>.

مثال)

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

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

<var> ::= A | B | C

static semantic

- Variables are int or real

- اگر جنس <var> + <var> ها یکسان نبود، حاصل real است.

Subject
Date

actual_type : A synthesized attribute associated with <var> & <expr>.

این ذخیره می‌کند تایپ واقعی یک متغیر یا یک عبارت را.

اگر <var> باشد، اسمی هم intrinsic attribute حساب می‌شود ولی

<expr> باید حساب بشود.

expected_type: An inherited attribute for <expr>

آن چیزی که انتظار داریم را ذخیره می‌کند.

به ازای BNF Syntax های rule یک Semantic Rule نوشت:

<assign> ::= <var> = <expr> —semantic rule→ <expr>.expected_type ← <var>.actual_type

(I)

<expr> ::= <var>[2] + <var>[3]   <expr>.actual_type ← if (var[2].actual_type = int)

(II)

and (var[3].actual_type = int)
then int
else real
end if

باید تایپ‌ها برابر باشند

predicate : <expr>.actual_type == <expr>.expected_type

<expr> ::= <var>   (III) <expr>.actual_type ← <var>.actual_type

predicate: <expr>.actual_type == <expr>.expected_type

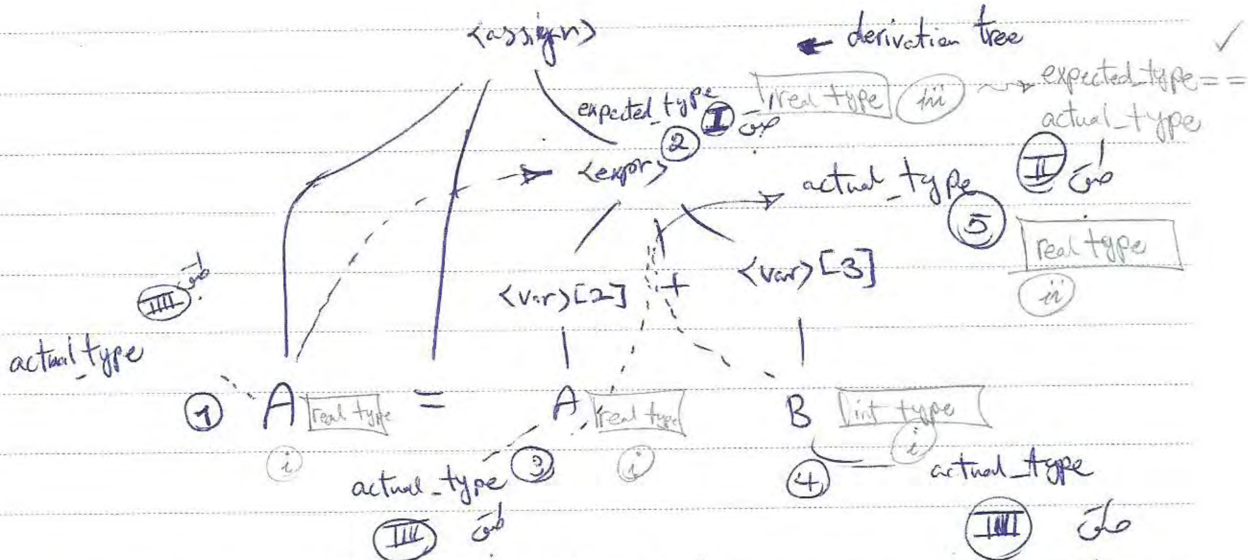
<var> ::= A|B|C   <var>.actual_type ← look_up(<var.string>)

(IIII)

که به صورت جدولی type های A,B,C را از آن می‌خواند.

Subject
Date

A=A+B



یعنی درخت را بر اساس BNF شکل می‌دهد و از برگ‌ها به سمت سطوح بالا می‌شود و بالا می‌رود ↑

نکته اول ← طراحان کامپایلر حتی در زمان این موقع attribute را لحاظ می‌کنند.

نکته دوم ← در جاهایی که می‌خواهند type systems را implement کنند شبیه همین عمل attribute grammar هست.

داشتن یک مجموعه Syntax directed در طراحی type system ها بسیار مهم است.

تا اینجا: F

... — lexical Analysis — Syntactic Analysis — Semantic Analysis — ...

Subject Programming Languages
Date

Semantics:

(semantics: معناشناخت / Semantic: معنایی)

به سه دسته کلی می‌توان دسته بندی کرد:

1) Operational semantics (معناشناخت عملکردی) → نزدیک ترین به اجرا / دستور

یک abstract machine مثل Turing ارائه می‌شود که گام به گام مرور می‌کند

توسط یک سری از state های میانی به final state میرسد، عملا در Operational semantics

symbol ها هستند به چشم ماشین Abstract Machine است.

2) Denotational semantics (معناشناخت دلالتی) →

این برنامه دال بر فلان شی است (ممکن است یک شیء ریاضی باشد یا ...)

مثل اینکه این برنامه برنامه‌ای است که ورودی را با 2 جمع میکند و نشان میدهد. (دیدگاهی که معرف برنامه‌ها به چشم توابع ریاضی نگاه میکنیم.)

3) Axiomatic Semantics (معناشناخت اصل موضوعی) → تعریف رفتاری از احکام میدهد

ارائه یک سری rule ها که رفتار برنامه را تعریف می‌کند.

Operational Semantic مخصوص زبان‌های imperative نیست. denotational و axiomatic اکثرا کاربرد دارند.

برای اثبات درستی (Correctness) است. (اثبات خواص برنامه‌ها) که به آنها formal semantics

دارد هر چند اگر خواننده باشیم ولی باید برنامه‌ها را بفهمیم که معنی برنامه‌ها چیست. زبان‌هایی مثل Java، Semantics های نوشته‌شده

دارد ولی هیچوقت همه آن را مطالعه نمی‌شود.

Subject

Date

با خواندن (مطالعه) semantics دیگران semantics یک زبان را به طور کامل نوشت و ...

مثال: $t ::= true \mid false \mid if\ t\ then\ t\ else\ t \mid 0 \mid succ\ \{t\} \mid pred\ t \mid iszero\ t$

تعریف inductive (استقرائی) (بازگشتی) Base_Case و یک تعریف بازگشتی که به Base_Case برسد

$f^0 = f$

$f^{n+1} = f \circ f^n$

در تعریف مفاهیم زبانی از rule استفاده می‌کنیم.

نمونه‌ای از Rule:

$$\frac{}{f^0 = f} \quad \text{و} \quad \frac{f^n = g}{f^{n+1} = f \circ g}$$

Conclusion ←

Axiom (فرض (صورت) ندارد) — Proper Rule

$$\frac{a\ int}{succ(a)\ int}\ (**) \qquad \frac{}{zero\ int}\ (*)$$

proper Rule — Axiom

Definition of Natural numbers

$$\cfrac{\cfrac{\cfrac{}{zero\ int}\ (*)}{succ(zero)\ int}\ (**)}{succ(succ(zero))\ int}\ (**)$$

یک derivation tree برای succ(succ(zero)) ←

ما در زبان judgment داریم. judgment آن چیزی است که حتماً proof (اثبات) می‌خواهد.

در زبان judgment تعریف می‌کنیم مثل $f = g$ یا $t \to t'$ یا $a\ int$ یا $b > a$

در تعریف یک judgment یک Rule داریم که یک تعداد judgment بالا و یک judgment پایین خط داریم.

$$\frac{J_1\ J_2 \cdots J_k}{J}$$

PAPCO

Subject

Date

اگر بخواهیم operational تعریف کنیم ، term ها را به تدریج تبدیل می‌کنیم تا به value برسیم به علامت

قواعد تعریف شده یک انتقال term را در یک "abstract machine" انجام می‌دهد.

$v ::= true \mid false \mid nv$

$nv ::= 0 \mid succ\ nv$

ادامه مثال ۲: تعریف کردن value ها برای بیان semantics ضروری است.

حال می‌خواهیم judgment transition را تعریف کنیم:

1) $\dfrac{}{pred\ 0 \rightarrow 0}$ ← برای گام کوتاه (زیر محاسبات) این فلش نشان داده می‌شود.

2) $\dfrac{t_1 \rightarrow t_1'}{succ\ t_1 \rightarrow succ\ t_1'}$

3) $\dfrac{t_1 \rightarrow t_1'}{pred\ t_1 \rightarrow pred\ t_1'}$

4) $\dfrac{}{pred\ succ\ nv_1 \rightarrow nv_1}$

5) $\dfrac{}{iszero\ 0 \rightarrow true}$

6) $\dfrac{t_1 \rightarrow t_1'}{iszero\ t_1 \rightarrow iszero\ t_1'}$

7) $\dfrac{}{iszero\ succ(nv_1) \rightarrow false}$

8) $\dfrac{}{if\ true\ then\ t_2\ else\ t_3 \rightarrow t_2}$

9) $\dfrac{}{if\ false\ then\ t_2\ else\ t_3 \rightarrow t_3}$

10) $\dfrac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ then\ t_2\ else\ t_3}$

Subject
Date

(اوایل فصل سوم کتاب types & pm — (pierce))

حساب کردن عبارت مقابل با استفاده از rule های گفته شده: Pred (succ (pred 0))

$$\dfrac{\dfrac{\dfrac{}{\text{pred } 0 \rightarrow 0}\ (1)}{\text{succ}(\text{pred } 0) \rightarrow \text{succ } 0}\ (2)}{\text{pred}(\text{succ}(\text{pred } 0)) \rightarrow \text{pred}(\text{succ } 0)}\ (3)$$

$$\dfrac{}{\text{pred}(\text{succ } 0) \rightarrow 0}\ (4)$$

عوض کردن تک تک مقادیر و term ها را مرحله به مرحله و افقی ها را به ترتیب می‌نویسیم.

$\Longrightarrow$ pred (succ (pred 0)) → pred (succ 0) → 0

می‌توان با تعریف کردن evaluation semantics یک‌ضرب به عبارت نهایی دسترسی پیدا کرد.

مثال بالا نمونه‌ای از یک معناشناخت operational است.

دو سوال مطرح می‌شود:

آیا ممکن است یک term به بیش از یک value، evaluate شود؟

آیا term های این زبان حتماً به value می‌رسند؟

Denotational semantics:

برای بیان معنای یک شیء، به ترکیب شیء دیگر دلالت می‌کند که معانی آن شیء قابل فهم برای ما باشد.

(denote: represent) meaning (connote → معنی غیرمستقیم)

بطور mathematical تعریف می‌شود.

Subject
Date

- The denotation of a program is a mathematical object.

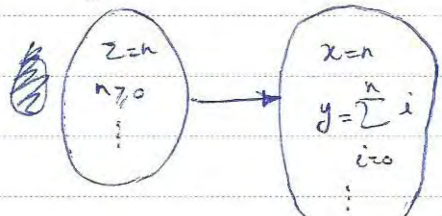
معنای یک برنامه یک شیء ریاضی است.

$x := 0;\ y := 0;\ \text{while } x \leq Z \text{ do } (y := y + x;\ x := x + 1)$

program : states → states

program یک تابع است از state ها به state های دیگر

state یک تابع است از name متغیرها به مقادیر



- denotational semantics must be Compositional.

ترکیب پذیر

if B then P else Q

denotes: $B \doteq B'$

$P = P' \Rightarrow$ if B then P else Q = if B' then P' else Q'

$Q = Q'$

از روی معنای بچه‌ها معنای parent را بفهمیم

: Meta language vs. Object language

زبانی که برای آن می‌خواهیم معناشناخت را بدهیم ← Object language

Meta language ← زبانی که به کمک آن می‌خواهیم معناشناخت را ارائه دهیم

(mathematics مثل)

3+6−4 → 5

عدد پنج به عنوان مدلول

Meta lang. ① یک عبارت در object language نوشتن نسبت برای که چه است

Denotational semantics of binary numbers:

$e ::= n \mid e+e \mid e-e$

$n ::= b \mid nb$

$b ::= 0 \mid 1$

$[[e]] \triangleq$ The parse tree of expression "e"

$[[e_1 + e_2]] \triangleq$ (tree: e with children $[[e_1]]$, $+$, $[[e_2]]$)

$E[[e]] \triangleq$ The meaning of e according to the parse tree $[[e]]$

$E[[0]] = 0 \rightarrow$ عدد صفر   منظور از علامت 0 ، عدد صفر است!

$E[[1]] = 1 \rightarrow$ عدد یک

$E[[nb]] = E[[n]] \circledast 2 \oplus E[[b]]$ (ضرب)

ضرب در Meta lang است

به معنای ضرب است

$E[[e_1 + e_2]] = E[[e_1]] \boxplus E[[e_2]]$

$E[[e_1 - e_2]] = E[[e_1]] \sqsubset\!\sqsupset E[[e_2]]$

[illegible] (⇒): Sound است اگر هرچه در operational semantics می‌گوییم، denotational semantics بگوید.

[illegible] (⇐): Complete است اگر هرچه در denotational semantics می‌گوییم، operational semantics هم بگوید.

Subject
Date

$e ::= v \mid n \mid e+e \mid e-e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid \ldots \mid 9$

$v ::= x \mid y \mid z \mid \ldots$

$E[\![x]\!](s) = S(x)$

$E[\![0]\!](s) = \ldots$

$E[\![nd]\!](s) = E[\![n]\!](s) \ldots E[\![d]\!](s)$

$E[\![e_1 + e_2]\!](s) = E[\![e_1]\!](s) + E[\![e_2]\!](s)$

$E : \text{parse trees} \times \text{states} \to \mathbb{N}$

$E : \text{Tree} \to (\text{State} \to \mathbb{N})$

Subject
Date

$$E : \text{Parse Trees} \to \{ f : \text{state} \to \mathbb{N} \}$$

یعنی تابع meaning، تابع برمی‌گرداند.

While Programs:

چون Turing Complete هستند می‌توانند هر مسائل را حل کنند پس برای مطالعه مناسب هستند.

$$p ::= x := e \mid p ; p \mid \text{if } e \text{ then } p \text{ else } p \mid \text{while } e \text{ do } p$$

e می‌تواند بولین یا arithmetic باشد. (با تعریف‌های معنی قبلی است)

$x := 0 ; \; y := 0 ; \; \text{while } x \leq z \text{ do } (y := y + x ; x := x + 1)$

معنای برنامه: تعریف است که z را نگاشت می‌دهد به یک عدد. مثلاً اگر z=10 مجموع تا 10 را حساب می‌کند.

توابع به دلیل وجود متغیرها، stateها هم هستند.

state: variable → value

Command: states → states

Command تابعی است از توابع به توابع (که هر کدام از توابع از متغیرها به مقادیر می‌روند)

$$\text{modify}(s, x, a) = \lambda v \in \text{Variable} . (\text{if } v = x \text{ then } a \text{ else } s(x))$$

$$\begin{cases} f : \mathbb{R} \to \mathbb{R} \\ f(x) = x + 1 \end{cases} \Longleftarrow \lambda x \in \mathbb{R} . \, x + 1$$

Subject
Date

lambda

مشترک زبان‌های functional، λ است.

(Core)

$[\![P]\!]$

C : Tree → state → state

(تابع) meaning

$$C[\![x := e]\!](s) = \text{modify}\,(s, x, E[\![e]\!](s))$$

تعریف است از یک state به state دیگر به طوری که مقدار e در s به حالت و بقیه متغیرها فرق نمی‌کنند.

$$C[\![P_1 ; P_2]\!](s) = C[\![P_2]\!]\,(C[\![P_1]\!](s))$$

$$C[\![\text{if } e \text{ then } P_1 \text{ else } P_2]\!] = \text{if } E[\![e]\!](s) \text{ then } C[\![P_1]\!](s) \text{ else } C[\![P_2]\!](s)$$

$$C[\![\text{while } e \text{ do } p]\!](s) = \text{if not } E[\![e]\!](s) \text{ then } s$$
$$\text{else } C[\![\text{while } e \text{ do } p]\!](C[\![p]\!](s))$$

مهم‌ترین چالش زبان (خود زبان)

مشکلات while

داشتن حلقه نامتناهی به ما این امکان را می‌دهد که Turing complete شود. بنابراین برخی برنامه‌ها Terminate نشوند (diverge) (in general partial)

تابع‌های معنایی

این تعاریف به تازگی گفته شده directed Syntax است.

یعنی به ازای هر Syntax، یک Semantics داریم.

$$C[\![\text{while } x = x \text{ do } x := x]\!] : \text{states} \to \text{states}$$

$$(s) = \bot \quad \text{(always diverging)}$$

این تابع همیشه Partial است.

Totally partial

Subject
Date

$C[\![\text{while } x=y \text{ do } x:=y]\!]$: States → States

$$C[\![\text{while } x=y \text{ do } x:=y]\!](s) = \begin{cases} s & s(x) \neq s(y) \\ \perp & \text{otherwise} \end{cases}$$

undefined

$x:=0 ; y:=0 ;$ while $x \leq z$ do $(y:=y+x ; x:=x+1)$

$s_0(z) = 1$

$s_1 = C[\![x:=0]\!](s_0)$

$s_2 = C[\![y:=0]\!](s_1)$ $\quad s_2 : \{(x,0),(y,0),(z,1)\}$

$C[\![\text{while } x \leq z \text{ do } (y:=y+x ; x:=x+1)]\!](s_2) = C[\![\text{while } x \leq z \text{ do } (y:=y+x ; x:=x+1)]\!](s_2)$

$=$ if not $E[\![x \leq z]\!](s_2)$ then s_2

else $C[\![\text{while } x \leq z \text{ do } (y:=y+x ; x:=x+1)]\!]\big(C[\![y:=y+x ; x:=x+1]\!](s_2)\big)$

$s_3 = \{(y,0),(x,1),(z,1)\}$

$= $ if not $E[\![x \leq z]\!](s_3)$ then else ... $\big(C[\![\ldots]\!](s_3)\big)$

$s_4 = \{(y,1),(x,2),(z,1)\}$

$=$ if not $E[\![x \leq z]\!](s_4)$ then ———

$= s_4$

پس این برنامه s_0 را به s_4 می‌برد.

این یک نمونه از denotational semantics بود!

Subject
Date

(Mitchell از کتاب)

① اختلاف رویکرد functional و imperative:

اختلاف expression ها و statement:

* side effect یعنی یک بخش از یک عبارت (phrase) اثر آن می‌ماند و تمام نمی‌شود بلکه باقی می‌ماند تا در محاسبات بعدی بیاید.

(مثل استفاده از متغیرها در زبان‌های امری)

این مسئله باعث می‌شود برنامه‌های parallel را سخت کند و این عیب محسوب می‌شود و Programming را سخت‌تر کرد.

* side effect در expression ها رخ نمی‌دهد.

* statement در زبان‌های functional نیست (مثل Haskell)

در زبان طبیعی چهار نوع جمله داریم:

Sentences
- imperative (امری)
- declarative (خبری)
- interogative (پرسشی)
- exclomatory (تعجبی) → wow!

* در زبان برنامه‌سازی دو نوع خبری و امری داریم. (x را برابر صفر بگذار ↓ statement ، x صفر است ↓ expression)

Subject
Date

$x := 5$ imperative → چنین فرمی را imperative گویند. (تغییر دهنده state هستند) (امری است)

Function f (int x) { return x+1; } declarative → (امر نیست، یک جمله است.)

این کتاب declarative و imperative را به گونه‌ای نگاه می‌کند که طراحی زبان باشد.

Functional Language:

* زبانی است که برنامه‌ها در آن specify شده به عنوان دنبال‌هایی از Functional هستند.
* زبانی است که در آن اغلب محاسبات با ارزیابی عبارت‌های شامل توابع انجام می‌شود. (evaluation)
* ML، Lisp، scheme نمونه زبان‌هایی هستند Functional ولی assignment دارند (قابلیت تعریف متغیرهای محلی Function را دارند)
* بعضی زبان‌ها pure functional هستند مثل Haskell، lambda که هیچ کدام این را ندارند.

Functional هستند. عبارتی دارای side effect وجود ندارد. ارزیابی عبارت‌ها state را تغییر نمی‌دهند. فی‌سبات تأثیری در محاسبات بعدی ندارد.
تعریف pure functional

Test of being pure functional:

✓ Declaration language Test: within the scope of specific declarations

Subject
Date

of $x_1, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, \ldots, x_n$ have the same value.

* مهم این طوری است که یک عبارت در همه جا یکسان است (ارزیابی می‌شود). پس C و Java، declarative نیست.
ارزیابی یک عبارت تأثیری در بقیه نداشته باشد.

* دو دنیا در Programming Language ها تصور می‌شود: 1) دنیای گزاره‌ها statement 2) دنیای عبارت‌ها expression

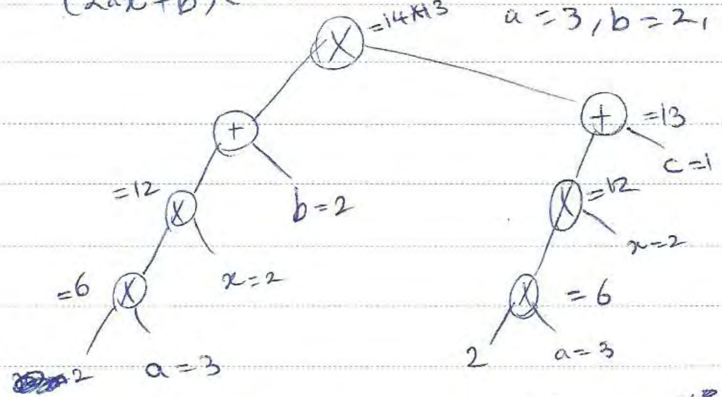
- The world of expression
  - $(a+b) \times c$
  - $(a+b) = 0$
  - $\neg(a \wedge b)$
- The world of statement
  - alters Control flow
    - if_then_else
    - loops
    - goto s
    - procedure invocation
  - alters state
    - $i := i+1$
    - input / output

در دنیای گزاره‌ها، ترتیب مهم است؛ برخلاف دنیای عبارت‌ها. (Order is relevant)

" " ، حق نداریم مثلاً یک عبارت مشترک را حذف کنیم چون به جایش نداریم.
(Common expression cannot be eliminated.)

در درخت‌ها می‌توان برنامه‌نویسی موازی (Parallelism) داشت

$(2ax+b)(2ax+c)$ $a=3, b=2, c=-1, x=2$



decoration ←

Parallelism : به معنای این است که مثلاً هر یک از گره‌ها را یک پردازنده در processor های مختلف در دو سمت Pram تقسیم می‌شود.

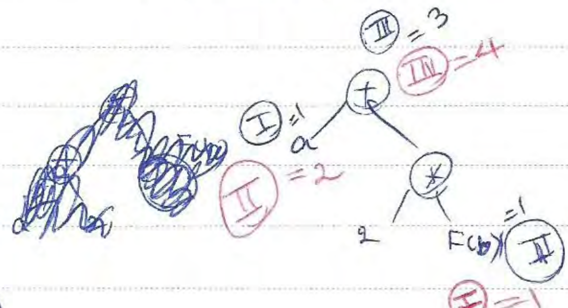
No matter what the order of decoration, as long as we obey the structure of the tree, we will always get the same answer.

* Church-Rosser property (confluence)

```
Function F(x: integer) : integer
    begin
        a := a+1;
        F := x*x;
    end
```

$a=1, b=1$

$a + 2 * F(b)$



پس برنامه‌نویسی موازی امکان‌پذیر نیست ↑

(order ها اهمیت دارد)

Subject
Date

۷) درصورتیکه meaning یک عبارت به context ارتباطی نداشته باشد، می‌توان آن عبارت را برداشت و عبارت معادل دیگری گذاشت. (Referential Transparency) (اصل شفافیت ارجاعی)

I saw ~~other~~ walter get into his car.

I saw walter get into his Ferrari.

(در صورتیکه والتر فقط فراری داشته باشد، قاعده برقرار است.)

meaning مستقل از context است!

He was called William Rufus because of his red hair.
(Rufus: Reddish)

He was called William II because of his red hair.

این referential transparency نیست چون مستقل از context نیست.

λ - Calculus:

زبانی کاملی است که هر محاسبه‌ای را بتوان انجام می‌دهد. اولین نمونه یک Language pure functional است.

این زبان در اوایل قرن بیستم و هم‌زمان با Turing و ... بوده است. (1920)

۶) Church - Turing Thesis: همه تصورات از محاسبه یکی است و ماشین Turing، λ Church و ... معادل هستند.

غیرقابل حل بودن یک مسئله به معنی این است که نمی‌توان برای آن λ-term نوشت. (یا ماشین تورینگ طراحی کرد)

Subject
Date

lambda term

(1) Syntax of λ-calculus: $t ::= x \mid \lambda x.t \mid t\,t$

lambda variable — lambda abstraction — lambda application

λ-calculus مبنای طراحی زبان‌های مختلف اعم از functional و imperative است. چون طراح این زبان Church
است، Core language می‌گویند. چرا که concept و چارچوب design همه زبان‌ها از این زبان است.

دیگر زبان‌هایی که همروند برنامه می‌نویسند، حسابان π-calculus است. ← Milner, ...

object-oriented هم نیاز به حسابان object-calculus است. ← Abadi & Cardelli

(1) اصطلاح model language به زبانی گفته می‌شود که ترین زبان core و زبان واقعی است و دارای امکانات
زبان خاص است. (feature های کمتری دارد.)

Untyped λ-calculus:

حسابان λ-calculus مؤثرترین ایده در زمینه programming language است.

$$\lambda x.t \overset{\text{از نظر ریاضی}}{\equiv} f(x)=t$$

از نظر برنامه‌نویسی:

$\lambda n.$ if $n=0$ then 1 else $n*(n-1)$

(در نحو λ، if، then، else، * را نداریم!) پس از اینکه نداریم این را برای این کارها به راحتی نداریم تولید کرد ابهام بود؟

$s, t, u, \dots \rightarrow \lambda$ term

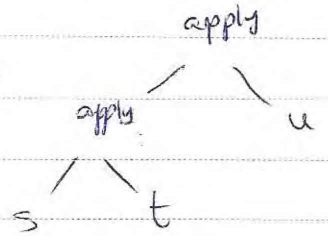
Subject

Date

به صورت پیش‌فرض از چپ associate می‌شود.

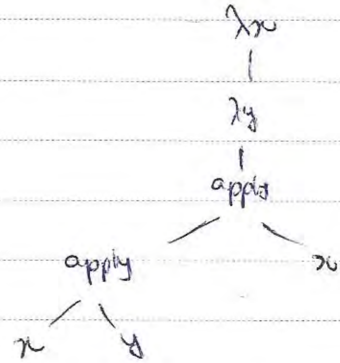
st application / $\lambda x.t$ abstraction / $stu = (st)u$

abstract syntax tree for stu $\Longrightarrow$



$\lambda x . \lambda y . xyx = \lambda x . (\lambda y . ((xy)x))$. از راست associate می‌شود

abstract syntax for above



Scope :

- An accurance of the variable x is "bound" when it occurs in the body of an abstraction $\lambda x.t$. (λx is "binder")

اگر وضعیت بالا نباشد، x را "free" است.

$\lambda x . x \longrightarrow x$ is bound

$\lambda z . \lambda x . \lambda y . x(yz) \longrightarrow x$ is bound

y " "

z " "

Subject

Date → this x is bound

$(\lambda x.\ x)\ x$

↳ this x is Free

اگر یک term متغیر Free نداشته باشد ← closed (combinator)

$id = \lambda x.\ x$ به ساده‌ترین‌ها:

↓

identity

Operational Semantics:

تنها حرکتی که در λ، application (tt) است، فقط همین را داریم:

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]\ t_{12} \qquad (\beta\text{-reduction})$$

به عبارتی، $[t_2/x]$ را هم می‌توان نوشت.

redex (reducible expression) ← جایگزین کردن هر t_2 به جای هر free occurrence در t_{12} می‌شود.

$$(\lambda x.\ ((\lambda x.\ x)\ x))(\lambda y.\ y) \longrightarrow (\lambda x.\ x)(\lambda y.\ y)$$

هر بخش‌بندی redex ها ممکن است.

Evaluation Strategies:

استراتژی‌های ارزیابی

1) Full beta-reduction → هر redex را بخواهی reduction را انجام می‌دهیم پیش می‌رویم

$(\lambda x.\ x)((\lambda x.\ x)(\lambda z.\ (\lambda x.\ x)\ z))$ (هر کدام را دوست داری انتخاب کن)

$id\ (id\ (\lambda z.\ id\ z))$

سه redex داریم

Subject
Date

ولی مثلا اگر :

$id\,(id(\lambda z.\,id\,z)) \to id\,(id\,(\lambda z.\,z)) \to$

$id(\lambda z.z) \to \lambda z.z \not\to$

یا

$id\,(id\,(\lambda z.\,id\,z)) \to id\,(\lambda z.\,id\,z) \to \lambda z.\,id\,z \to \lambda z.z \not\to$

که این دیگر redex نیست که دیده شود.

2) Normal Order:

بیرونی ترین چپ ترین redex باید ابتدا کاسته شود. (یعنی کوتاه ترین ... در مثال بالا 4)

3) Call-by-name:

مخصوص
در زبان های برنامه سازی استفاده می شود.

اجازه reduction درون abstraction را نمی دهد.

$id(id(\lambda z.\,id\,z)) \to id(\lambda z.\,id\,z) \to \lambda z.\,id\,z \not\to$

(lazy Semantics!) هر چیزی به تعداد دفعات occurrence آن name حساب میشود.

(پیش از حساب تابع)
↳ فقط مقدار آن را نمیدهیم بعدا محاسبه میشود.

Call-by-need شبیه Call-by-name است که بعد از محاسبه، ذخیره میکند و دوباره آن را حساب نمیکند.

4) Call-by-Value:

تا حدی شبیه است ولی باید پیش از اینکه redex را apply کنیم، آرگومانش value باشد.

Subject (Pierce فصل پنجم)
Date

اکثر زبان‌ها از این استفاده می‌کنند.

$id\,(id\,(\lambda z.\, id\, z)) \rightarrow id\,(\lambda z.\, id\, z) \rightarrow \lambda z.\, id\, z \nrightarrow$

وقتی به Value رسیدیم، هیچ ساده‌سازی انجام نمی‌دهیم.
(abstraction)

strict → Call_by_Value
non-strict → Call_by_name

هر برنامه‌ای را که بتوان نوشت، می‌توان آن برنامه را با λ-Calculus نوشت.

Multiple arguments:

$\lambda(x,y).\,s$ ⇐ که چنین چیزی نداریم

به جای آن از higher-order function استفاده می‌کنیم؛ یعنی به ترتیب متغیرها را می‌گیریم و یک تابع برمی‌گردانیم.

$f(x,y) = (f(x))(y)$

$f(x)$ یک تابع second order است چون یک عدد می‌گیرد و یک تابع برمی‌گرداند.

$\lambda(x,y).\,s = \lambda x.\,\lambda y.\,s$ (به ترتیب به متغیرها apply می‌کنیم)

$(\lambda x.\lambda y.\,s)\,t\,r \rightarrow ([x \mapsto t](\lambda y.\,s))\,r \rightarrow [y \mapsto r][x \mapsto t]\,s$

به این کار، Curring گفته می‌شود.

Boolean و ... را نداریم که بر حسب سازه‌های λ می‌سازیم.

47
Subject:
Year. Month. Date. ( )

Church Boolean:

tru = $\lambda t.\lambda f.t$

fls = $\lambda t.\lambda f.f$

این روش‌ها را باید درست حفظ باشید:

test = $\lambda l.\lambda m.\lambda n.lmn$

test b v w

b: boolean

if b → then v → else w

آنطور طبیعی

test tru v w = $(\lambda l.\lambda m.\lambda n.lmn)$ tru v w $\longrightarrow (\lambda m.\lambda n.\text{tru}\,mn)$ v w

$\longrightarrow (\lambda n.\text{tru}\ v\ n)\ w \longrightarrow \text{tru}\ v\ w = (\lambda t.\lambda f.t)\ v\ w \longrightarrow$

$(\lambda f.v)\ w \longrightarrow v$

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

b: boolean

if (b = true) then c

if (b = false) then fls

or = $\lambda b.\lambda c.b$ tru c

neg = $\lambda b.b$ fls tru

Pair :

ما علاقه ای به ساختار درونی نداریم، بلکه به جزء اول و دوم آن توجه داریم (مانند Blackbox)

ما برای ساختن pair می‌خواهیم، مولفه اول و دوم را از هم تفکیک کنیم.

~~pair = λf. λs. λb~~

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

$fst = \lambda p.\ p\ true$

$snd = \lambda p.\ p\ false$

$fst(pair\ t_1\ t_2) \longrightarrow (\lambda p. p\ tru)((\lambda f. \lambda s. \lambda b.\ b\ f\ s)\ t_1\ t_2)$

$\xrightarrow{*} (\lambda p. p\ tru)(\lambda b. b\ \ t_1\ t_2) \rightarrow (\lambda b. b\ \ t_1\ t_2)\ tru \rightarrow$

$tru\ \ t_1\ t_2 \rightarrow t_1$

Church Numeral :

عددها را با zero و عملیات succ نمایش می‌دهند.

$c_0 = \lambda s. \lambda z. z$ ⟶ صفر در λ-Calculus خالص

$c_1 = \lambda s. \lambda z. s z$ ⟶ $Succ(z)$

$c_2 = \lambda s. \lambda z. s(s z)$

$c_3 = \lambda s. \lambda z. s(s(s z))$

$scc = \lambda n. \lambda s. \lambda z. s(n s z)$

Subject:

Year. Month. Date. ( )

د) plus = λm. λn. λs. λz. m s (n s z)

plus c_m c_n = f_{m+n}

د)

times = λm. λn. m (plus n) c_0

⋮

Turing Complete است، زیرا recursion دارد.

Recursion:

factorial(n) = if n=0 then 1 else n * factorial(n-1)

$f: \mathbb{N} \to \mathbb{N}$

$f(0) = 1$ تعریف بازگشتی:

$f(n+1) = (n+1) \times f(n)$

⇒

تعریف به صورت کلی:

$$f(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times f(n') & \text{if } n = n'+1 \end{cases}$$

⇒

تعریف به صورت دیگر:

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

Subject:
Year. Month. Date.

$F: f_g \longmapsto f_{g'}$

$g': n \longmapsto \begin{cases} 1 & \text{if } n=0 \\ n \times g(n') & \text{if } n = n'+1 \end{cases}$

$F(g) = g$

F is a functional

Year. Month. Date.

(combinator) operator

... fixed point ... recursion ...

Fix F = the fixed point of F

↳ λ-term

$$F(\text{Fix}\ F) = \text{Fix}\ F \qquad \text{Fix} = ?$$

$$\text{Omega} = (\lambda x.\ x\ x)(\lambda x.\ x\ x) \to (\lambda x.\ x\ x)(\lambda x.\ x\ x) \to \dots$$

... normal form ... (λ-abstraction) ... λ-term ... (*)

... diverge ...

$$Y = \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

$$Y\ g = (\lambda x.\ g(x\ x))(\lambda x.\ g(x\ x))$$

$$= g(\lambda x.\ g(x\ x))(\lambda x.\ g(x\ x)) = g(Y\ g)$$

... g ... fixed point ... Yg

fix = Y ... call-by-name fixed-point combinator

$$Z = \text{fix} = \lambda f.\ (\lambda x.\ f(\lambda y.\ x\ x\ y))(\lambda x.\ f(\lambda y.\ x\ x\ y))$$

call-by-value fixed-point combinator

$$Zg = g(Zg) \to \eta\text{-reduction}$$

Subject:
Year. Month. Date. ( )

تعریف فاکتوریل در λ :

$F = \lambda g.\lambda n.$ if $n=0$ then 1 else $n * g(n-1)$

Factorial = fix F

همچنین فیبوناچی قابل تعریف در λ است

$fix(\lambda g.\lambda n.$ if $n=0$ then 1 else $n*g(n-1))\, C_5 = C_{120}$

$fix(\lambda g.\lambda n.$ if $n=0$ then 1 if $n=1$ then 1 else plus$(g\ n-1)(g\ n-2))$
||
Fibonacci فیبوناچی

(چرا Y در call-by-value مسئله دارد؟) چون در [illegible] Yg را fix می‌کند (در call by value ...)

$Yg = g(Yg) = g(g(Yg)) = \dots$

(ربط) $Zg = g(Zg)$ را ثابت کنید

تمرین: فصل پنجم کتاب Pierce :

5.2.2
5.2.3
5.2.4
5.2.7
5.2.8
5.2.10
5.2.11

(جمعه هفته آینده)

η-reduction $\rightarrow \lambda x.Mx \underset{\eta}{\rightarrow} M$

$Zg \rightarrow g(\lambda y.(\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy))y)$

$\rightarrow g(\lambda y.\underset{M}{Zg}\ y) \rightarrow g(Zg)$

121 Shadab

Subject:
Year. Month. Date. ( )

$Y = \lambda f.(\lambda x.\ f(x\,x))(\lambda x.\ f(x\,x)) \rightarrow Yg = g(Yg)$

$g = \lambda f.\lambda n.$ if $n=0$ then 1 else $n * f(n-1)$
↳ Functional

$Yg \rightarrow$ نقطه ثابت g است

g روی Yg، apply میکند

factorial(2) $= Yg(2) = g(Yg)(2) = (\lambda n.$ if $n=0$ then 1 else $(n * (Yg)(n-1)))\ 2$

$=$ if $2=0$ then 1 else $2 * (Yg)(1) = 2 * (Yg)(1) =$

apply ?

$2 * g(Yg)(1) = \dots = 2 * 1 * (Yg)(0) = \dots = 2 * 1 * 1$

این هم recursion مورد نظر است

Definition - The set of free variables of a term t, written FV(t), is defined as follows: (تعریف بازگشتی)

$FV(x) = \{x\}$

$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$ ← خارج کردن

$FV(t_1\,t_2) = FV(t_1) \cup FV(t_2)$

این تعریف syntax directed است چون به syntax ترم ها بستگی دارد (...)

Substitution:

$[x \mapsto s]t$

s را بجای همه‌ی x ها در t جایگذاری کن.

تعریف استقرایی است. (inductive)

I) $[x \mapsto s]x = s$

II) $[x \mapsto s]y = y \quad (\text{if } x \neq y)$

III) $[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$

IV) $[x \mapsto s](t_1 t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

تا حدودی قواعد بالا درست به نظر می‌رسند.

مثال:

$$[x \mapsto \underbrace{(\lambda z.zw)}_{s}](\underbrace{\lambda y.x}_{t_1})$$

$$= \lambda y.[x \mapsto (\lambda z.zw)]x = \lambda y.\lambda z.zw$$

اما این یک مثال مشکل‌ساز است.

$$[x \mapsto y](\lambda x.x) = \lambda x.y$$

ولی قطعاً این تعریف یک اشکال دارد.

در نتیجه اگر y را بگذاریم، تابع عوض می‌شود!!

پس برای درست کردن قواعد نسبت به قاعده III را اصلاح می‌کنیم:

$$\text{III)}\ [x \mapsto s](\lambda y.t_1) = \begin{cases} \lambda y.t_1 & \text{if } y = x \\ \lambda y.[x \mapsto s]t_1 & \text{if } y \neq x \end{cases}$$

حال یک مثال می‌زنیم،

$$[x \mapsto z](\lambda z.x) = \lambda z.z$$

این هم درست نیست که معنی عوض شد. یعنی با جایگزینی به تابع id رسیدیم.

متغیر z ، capture شده است (variable capturing) (یعنی متغیرهای free ، bound شده‌اند)

روش Capture Avoiding Substitution (جایگزینی) را به صورت زیر تعریف می‌کنیم:

$$\text{III)}\ \ldots = \begin{cases} \ldots & \\ \ldots & \text{if } y \neq x \text{ and } y \notin FV(s) \end{cases}$$

در substitution نیاز به α-renaming (α-Conversion) داریم یعنی

یک متغیر fresh مثل y کنار بگذاریم (متغیر) bound یک λ-term را با متغیر fresh (rename کنیم)

$$[x \mapsto z](\lambda w.x) = \lambda w.z$$

پس قاعده III می‌شود:

(*) III) $[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$ if $y \neq x$ and $y \notin FV(s)$

و همچنین α-renaming داریم که مشخص شود...

Subject:
Year. Month. Date. ( )

Operational Semantics (for λ-Calculus):

به judgment تعریف می‌کنیم (یک Transition)

$t \longrightarrow t'$

1) $\dfrac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2}$

2) $\dfrac{t_2 \longrightarrow t_2'}{v_1 t_2 \longrightarrow v_1 t_2'}$

3) $\dfrac{}{(\lambda x.t_{12})v_2 \longrightarrow [x \mapsto v_2]t_{12}}$ → Axiom

$(v ::= \lambda x.t)$ → λ-abstraction

value

این بیان می‌کند ما داریم Call-by-Value می‌کنیم چون در Axiom فقط value ها مجاز است

این semantics، ترم ها را ارزیابی می‌کنند.

[تمرین] (تحویل هفته بعد) (pierce ـ ت)

5.3.6
5.3.7
5.3.8

(mitchell ـ ت)

4.4
4.5
4.6
4.8
4.9
4.11 4.13

Subject:

Year. Month. Date. ( )

Non_Standard Semantics: (Abstract Interpretation)

(Analysis)

initialize_before_use

$state = \{error\} \cup (Variables \rightarrow \{init, uninit\})$

$E[\![e]\!](s) =$ error if e Contains any Variable with $s(y) = uninit$

$state = O.K.$ otherwise

$C[\![x := e]\!](s) =$ if $E[\![e]\!](s) = O.K$

then modify(s, x, init)

else error

$s_0 = \lambda v.\ uninit$

$v \in Variable$

Subject:

Year. Month. Date.

$c[\![x := a]\!](s_0) = modify(s_0, x, init)$

$c[\![p_1 ; p_2]\!](s) = if\ c[\![p_1]\!](s) = error\ then\ error$

$else\ c[\![p_2]\!](c[\![p_1]\!](s))$

$s_1 \uplus s_2 = \lambda v.\ if\ s_1(v) = s_2(v) = init\ then\ init\ else\ uninit$

$c[\![if\ e\ then\ p_1\ else\ p_2]\!](s) = if\ E[\![e]\!](s) = error\ or$

$c[\![p_1]\!] = error\ or$

$c[\![p_2]\!] = error$

$then\ error$

$else\ c[\![p_1]\!](s) \uplus c[\![p_2]\!](s)$

$c[\![if\ a=1\ then\ x := a\ else\ x := 1 ; y := 2]\!](s_0) = modify(s_0, x, init)$

Subject: Mitchell (فصل چهارم کتاب Mitchell پایان یافته است) Mitchell از جلسه بعد
Year. Month. Date. ( )

این پیچ Sound است یعنی هر چیزی که ردیابی کردیم درست است. ولی Compelete نیست

یعنی ممکن است برنامه اشکال داشته باشد (مثلاً حلقه بی‌پایان داشته باشد) ولی ما آن را اشکال‌دار ندانیم

Lisp:

(فصل دوم کتاب Mitchell)

(طراحی زبان) lisp چه مشکلی را برطرف کرده است؟ Lisp مخفف چیست؟ برای اینکه طراحی زبان را بررسی کنیم ...

John McCarty ← طراح زبان Lisp

LISt processor

بیشتر به خاطر simplicity و flexity اش مطرح است.

به خاطر همین مناسب است ← exploratory programming

emacs و gtk بر اساس LISP ساخته شده است.

LISP انواع مختلفی دارد از جمله Maclisp و scheme

Recursive Functions of symbolic expressions and their computation by machine ← در این مقاله ایده اصلی LISP بوده است

LISP برای Symbolic computation طراحی شده است.

Subject: Programming Languages
Year 1392 Month. 07 Date. 30

درمورد LISP سه جنبه اصلی را بررسی می‌کنیم:

I – motivating application

انگیزه اصلی طرح زبان
↓
در کدوم موارد موثر است.

- Advise taker ~ زودره → Common sense reasoning

(سوال می‌پرسی، ماشین جواب می‌دهد)

- Symbolic expression

$\int x \, dx = \frac{x^2}{2} + c$ (به جای عدد، کلمه)

→ Symbolic Computation انجام

II – Abstract Machine

این مدل مبتنی بر IBM 704 برداشته شده. (بر پایه ماشین خاص)

اگر بر پایه خیلی ماشین خاص باشد، portability را از دست می‌دهد

اگر خیلی abstract و general باشد، efficiency را از دست می‌دهد.

III – Theoritical Foundation:

از نظر تئوری ثابت شده Turing Complete است، هر Partial Function را می‌توان پیاده کرد.

بسیار مبتنی بر λ است. (نه چندان زیاد!)

Overview:

در اینجا LISP 1.5 مطرح می‌شود (دهه ۶۰).

prefix notation:

(+ 1 2 3 4 5) → 1+2+3+4+5

علم برنامه‌نویسی می‌خواهد

Subject:
Year. Month. Date. ( )

(* (+ 2 3) (+ 4 5)) → (2+3) * (4+5)

(f x y) → f(x, y)

Atoms:

✓ تمام برنامه‌ها lisp فقط بر اساس اتم‌ها و سلول‌ها بنا می‌شود؟

integer , floating point , symbolic
2 2.1 duck

(اتم‌ها غیر قابل تجزیه‌اند)

<atom> ::= <smbl> | <num>

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

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

✓ اتم خاص "nil" به معنی لیست‌های خالی. (رزرو شده است)

S-Expression:

✓ ابتدایی‌ترین (Basic) data structure ، زوج‌های نقطه‌دار (dotted pairs) هستند.

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

→ زوجی از sexp ها

Subject:
Year. Month. Date. ( )

Functions & special forms:

Functions:

Cons, car, cdr, eq, atom → چک کردن اتم بودن

concatenate
consolidate

car → قسمت اول را برمی‌گرداند

cdr → (یعنی بغیر از قسمت اول) قسمت دوم را برمی‌گرداند

eq → چک کردن تساوی

Content of address register   Content of decrement register

(مانند pointer هستند و DS ها مثل Linkedlist هستند)

Special form:

Cond, lambda, define, quote, eval → evaluate می‌کند

شرط، تعریف تابع، برای مشخص کردن اتم ها

اینها آرگومان ها را ارزیابی (evaluate) نمی‌کنند سپس محاسبه می‌کنند.
(به همه آرگومان ها)

* شروطی وارد که برای محاسبه شرط همه آرگومان ها ارزیابی نشوند. به این دلیل به اینها special form می‌گویند

eval مشابه یک تابع ... اتفاق می‌افتد

+, -, * عملگرهای ریاضی

تا اینجا pure LISP بود یعنی functional است. همه expression هستند

side effect ندارند. اما LISP دارای انواعی از side effect ها است.

پس LISP purely functional نیست

Shadab 121

Subject:
Year. Month. Date. ( )

rplaca , replacd , set , setq

↓

replace car

دستور assignment است.

(کنش Impure نیز Lisp)

* (cond (p_1 e_1) (p_2 e_2) ... (p_n e_n))

≡ if p_1 then e_1 else if p_2 then e_2 ...

Special form ها ابتدا p_1 را ارزیابی می‌کند اگر TRUE بود e_1 را ارزیابی می‌کند و تمام وگرنه ادامه می‌دهد...

T for TRUE
nil for FALSE

(quote cons) → به اسم نماد "cons" ساخته می‌شود (به عنوان) → cons

(cons a b) → یک جفت نقطه‌دار را ایجاد می‌کند که car به a اشاره می‌کند و cdr به b اشاره می‌کند

(cons (quote A) (quote B)) → جفت نقطه‌دار "B","A"

(quote (+ 1 2)) → (+ 1 2)

'(+ 1 2) → (+ 1 2)

نماد ' quote برگردانده می‌شود.

مثال) یک تابع بنویسید که یک لیست را جستجو می‌کند و دو ورودی x و y می‌گیرد و می‌بیند آیا x در لیست y هست یا نه.

```
(define find (lambda (x y)
```

find: نام تابع — (x y): دو ورودی تابع

```
(cond ((equal y nil) nil)
```

cond: شرط — اگر y برابر nil باشد جواب nil است (اگر لیست خالی است، False)

```
((equal x (car y)) x)
```

اگر x عنصر اول لیست باشد، x را برمی‌گرداند.

```
(true (find x (cdr y)))
)))
```

(else) در غیر این صورت x را در بقیه لیست می‌گردد.

```
(find 'Apple '(pear peach Apple Fis Banana))
```

این تابع "Apple" را برمی‌گرداند.

* Historical LISP (1960) و Dynamicly Scope است.

در زبان‌ها اجرا معلوم می‌شود متغیرها از کجا می‌آیند (آخرین Activation Record اجرا)

در staticly scope، شناسه متغیر مشخص شده و آن مهم است. (شامل آن متغیر است مهم است)

Subject:
Year 392 Month 08 Date 05

نوآوری های LISP :

① ترکیب statement و expression و در یک نوآوری است declaration.

② Conditional Expression (if-else) طرح شد. (برای اولین بار) (جایگزین jump می گردد)

(Cond ((< 2 1) 2) ((< 1 2) 1))) ⇒ 1

(Cond ((< 2 1) 2) ((< 3 2) 3))) ⇒ undefined

(Cond (true 0) (diverge 1)) ⇒ 0

از این بیرون می آییم.

(Cond (diverge 1) (true 1)) ⇒ undefined

expression : entity ... side effect ... value

statement : زبانی که باعث تغییر state می شود.

$x := y + 3$

declaration : new identifier را تعریف می کنیم

expression : if f(2) = 2 or f(3) = 3 then 4 else 4

و می شود if-then-else ... statement ... (بیشتر به گونه ...)

اولین بار ... expression ... است

if (condition) goto 112 ← LISP به طور non-strict عمل می‌کند

The Lisp Abstract Machine:

LISP یک Abstract Machine دارد که شامل اجزای زیر است (چهار جزء دارد)

اجزای Lisp Abstract machine ←

- A Lisp expression to be evaluated.

- A Continuation, which is a Function ← همینه که ادامه برنامه‌های بعدی
  representing to Remaining program
  to evaluate when done with the current expression.
  (که بعد از آن برنامه را ادامه بدهیم)

- An association list (A-list) ← مقادیر متغیرها در یک لیست نگهداری می‌شود
  or Runtime stack.
  مستقیم در یک stack را نگه‌داری می‌کنیم، value ها معلوم می‌کند به چه چیزهایی هستند

تعریف semantics و برنامه تعریف Abstract Machine است.

- A heap which is a set of cons cells.
  یک لیست car و cdr است

بر این چهار جزء Abstract Machine و state های Abstract Machine گفته می‌شود.

مثلاً برنامه‌ای به دنبال مثل نوشتیم را بررسی می‌کنیم:

Find (x, y)

(scheme اینطور بود!)

A-list

| x | y |

x → apple

y → (peer peach ... banana)

x و y آدرس‌هایی به مکانی در heap است

ادامه دارد ←

Subject:
Year 1392 Month 08 Date 05 (5)

equal y nil → nil → (Continuation) → equal x (car y) → nil → (Continuation) → Find (x (cdr y)) → Changing A-list

| x | y |

x → (apple)

y → (peach ... banana)

جالب است بدانید این مفهوم "Contextual Semantics" (معناشناخت زمینه‌ای) است.

خاطر نشان می‌شود [illegible] در جزئیات [illegible] دیده می‌شوند.

Cons Cells:

بنیادی‌ترین ساختار داده‌ای در Lisp، Cons cell است. هر Cons cell دو بخش دارد:

Cons cell

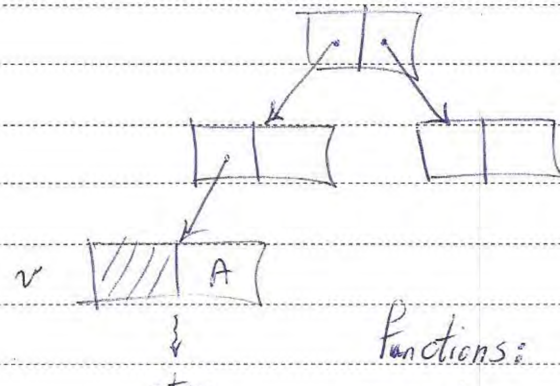
address (car) | decreament (cdr)

در Lisp هیچ محدودیتی در [illegible] نیست!

car و cdr نام‌های [illegible] هستند [illegible]

Subject:
Year. 1392 Month. 08 Date. 05

در Lisp چیزی به جز conscell نداریم. و در هر چیزی به atom می‌رسیم با یک عنصر



atom ها دارای pattern های مشخصی‌اند

تشخیص atom نصفش [illegible]

Functions:

1. atom v → درست است اگر اتم باشد

2. eq x y → آیا این دو درست به [illegible] ← اشاره‌گرها
در غیر این صورت اگر هر دو atom
با هم برابر هستند یا نه؟

3. (Cons x y)

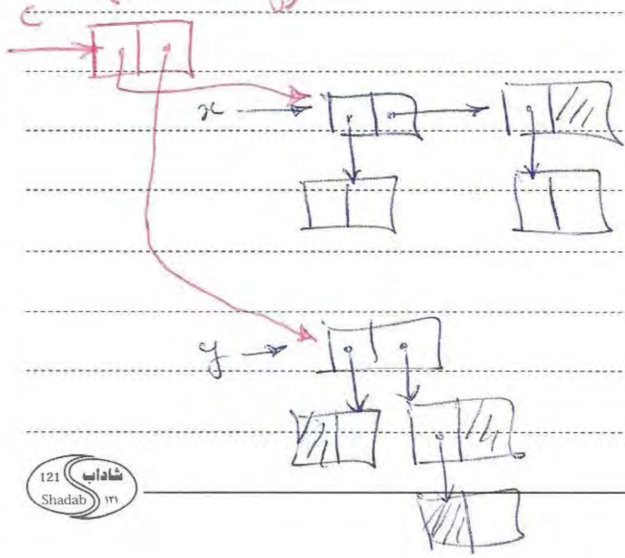
(نماد = را برای اعداد استفاده می‌کنند)

(equal ترکیبی است و معمول‌تر است و در صورتی که atom باشند به eq می‌رسد)

✓ Lisp با تعریف هوش مصنوعی بود.

(Cons x y)

* نحوه کار Cons:

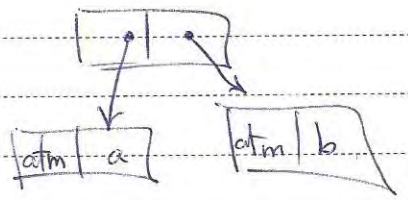


121 شاداب Shadab

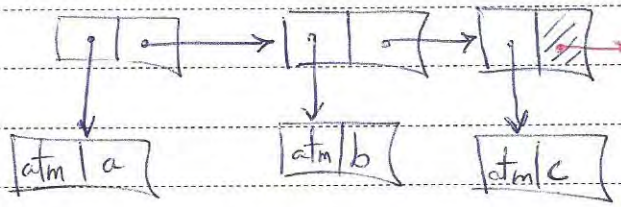
4. Car و Cdr

▷ (Cons 'a 'b)

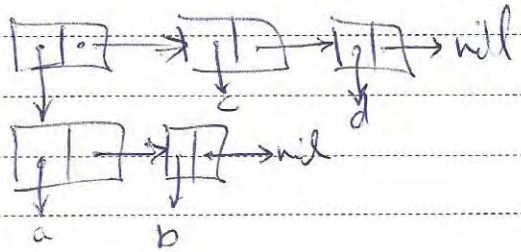
(a.b)



▷ '(a b c)



(a . (b . (c . nil)))



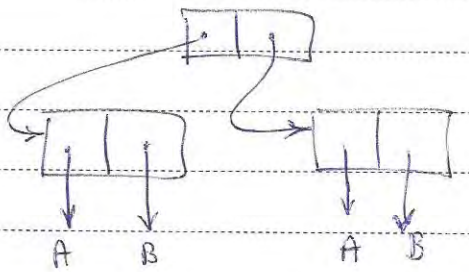
((a b) c d)

((((a . (b . nil)) . c) . d . nil)

Subject:

Year. 1397 Month. 08 Date. 05

(Cons (Cons 'A 'B) (Cons 'A 'B))



((A . B) . (A . B))

((A B) (A B)) → B, A

(set 'apost '(math mark peter))

(math mark peter)

(Cons 'John 'apost)

(John math mark peter)

(Cons 'a nil)

(a)

(Cons '(a b) '(c (d e)))

((a b) c (d e))

Subject:
Year 1392 Month 08 Date 05

> (car 'apost)

John ← خروجی

> (cdr '(a b c))

(b c) ← خروجی

> (cdr apost)

(matt mark peter) ← خروجی

Exercise: 3.1, 3.2, 3.4, 3.6, 3.8 (صفحه 87 کتاب Mitchell)

program as data در Lisp و eval آن (special form)

- To substitute expression x for all occurances of y in expression z and then evaluate the resulting expression.

```
(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)))))))
```

اگر exp2 یک اتم باشد

Subject:
Year. 1392 Month 08 Date. 27

define substitute-and-eval (lambda (x y z) (eval (substitute x y z))))

↓

"program as a data"

(lambda (<parameters>) <function body>) :کد تعریف سینتکس

((lambda (x) (+ (square x) y))

formal parameter

کد apply کردن تابع

((lambda (x) (+ (square x) y) 4) => 16+y

actual parameter

global variable
(not bound variable)
(free variable)

binder
local variable
bounded variable

recursion در زبان‌های قبل از LISP ممکن نبوده است.

Recursion:

McCarthy استفاده از label را پیشنهاد داد

(label f (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))

بعدها به جای label از define استفاده کردند

(define f (lambda ....)

بعدها هم به defun تغییر کرد:

(defun f (x) (...)

Subject: Programming Language
Year. 1392 Month 08 Date. 07

λ-calculus به صورت کلی آن، نمی‌تواند در پیاده‌سازی اجرا شود.

در زبان Functional، در پیاده‌سازی، تغییر state داریم ولی در ظاهر برنامه دیده نمی‌شود.

ولی در زبان‌های Imperative، تغییر state‌ها observable است.

Higher order Functions:

تابع first-order، ورودی و خروجی آن نمی‌تواند تابع باشد.

تابع Higher-order، ورودی و خروجی آن می‌تواند تابع باشد (ورودی یا خروجی تابع است) (توابع بالاتر)

مثل تابع MAX با اشتراک دو تابع

(second-order هستند، یعنی ترتیب first-order هستند)

Third-order Function
تابعی که ورودی آن توابعی هستند که second-order باشند، پس تابع را ...

مثال: $(f \circ g)(x) = f(g(x))$

(define compose (lambda (f g) (lambda (x) (f (g (x))))

(anonymous) یک تابع بی‌نام

(Compose (lambda (x) (+ x x)) (lambda (x) (* x x)))

$\Rightarrow$ lambda (x) (+ (* x x) (* x x))

مثال) یک تابع بگیرد و یک لیست، آن تابع را به تک تک اعضای لیست apply کند.

```
(define maplist (lambda (f x))
    (cond ((eq x nil) nil)
        (true (cons (f (car x)) (maplist f (cdr x))))))
```

```
(maplist square '(1 2 3 4 5)) ⟹ (1 4 9 16 25)
```

Garbage Collection :

زبان‌هایی هستند که حافظه را allocate می‌کنند. باید ابزاری داشته باشیم که deallocate کند

هر حافظه‌ای که استفاده نشود. البته زمانی مشخص است که ما در برنامه استفاده‌ای نخواهیم کرد.

✓ در بعضی زبان‌ها برنامه‌نویس باید به طور دستی garbage collection را انجام دهد. در بعضی زبان‌ها خود زبان به صورت اتوماتیک برای

مثال C garbage collection ندارد ولی LISP دارد. (البته در زبان‌هایی که ... portable نیستند؛
(که در interpreter انجام می‌شود)

✓ 5٪ سربار (overhead) اجرا در زبان اضافه می‌کند.

آزمایش garbage collector

✓ At a given point in the execution of a program p, a memory location m is garbage if no completed execution of p from this point can access location m.

اگر محتویات m را عوض کنیم هیچ تأثیری ندارد.

Subject:

Year. Month. Date. ( )

garbage collected language : garbage collection

(Car (Cons e1 e2))

↓

garbage : evaluate

((lambda (x) (car (Cons x x))) '(A B))

↓

garbage

garbage Collection :

I. mark and sweep :

tag

tag

garbage tag

II. reference Counting :

Subject:
Year. 1392 Month 08 Date 07 ( )

سوال: یک لیست و یک عنصر دیگر از list را به عنوان ورودی بگیرد و لیست را برگرداند.

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

اگر بخواهیم همین برنامه را به زبان C بنویسیم:

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

Subject: برنامه‌سازی

Year. Month. Date: ( )

در برنامه بعد، اتم x را از list حذف می‌کند و پوینتر آن لیست جدید را برمی‌گرداند. دنباله لیست را داریم.

```
cell * struct ( cell * a , cell * list ) {

  cell * ptr , *previous ;

  for ( ptr = lst ; ptr != 0 ; )

        if ( ptr -> car == x ) return ( ptr -> cdr ) ;

        else { previous = ptr ; }

        ptr = ptr -> cdr ;

        free ( previous ) ;

     }

}
```

این برنامه مناسب نیست!

{ ۹۲ / ۸ / ۲۶ → اول بهمن ترم }

تاریخ lisp

* این در دستورهای side effect هست:

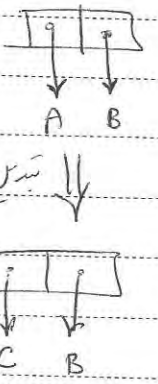
(rplaca x y) : replace the address field of cons cell x with y

(rplacd x y) : replace the decrement field of cons cell x with y.

Subject:

Year. Month. Date. ( )

(rplaca (cons 'A 'B) 'c)



assignment

(Functional (declarative) LISP)

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

a

c

side effect

expression

rplaca

list

A → B → C → D → ...

LISP list

Subject:
Year. 1397 Month. 08 Date 04 (8)

PURE LISP

(Cons (Car x) (Cons (Cadr x) (Cons y (cdddr x))))

A B C D

Car(cdr x) ← syntactic sugar

cdr(cdr(cdr x)) ← syntactic sugar

Impure LISP

(rplaca (cddr x) y)

(cdr(cdr x)) ← Syntactic Sugar

side effect

Correctness

pure lisp

Manageable

impure lisp

parallelism

impure lisp

Subject:
Year. 1392 Month. 08 Date. 12 S

Algol:

زبان های Algol-like به ML، C، Pascal و ... اطلاق می شود.

از جمله ویژگی های این زبان ها: type (در زبان برنامه نویسی این امکان هست که) operation های معتبر را statically مشخص

و هم جلوگیری کنیم و از طرفی مدیریت حافظه (۴ بایتی، ۸ بایتی و ... بیت متغیر) را می توان انجام دهد.

* type نوع داده ها را مشخص می کند. type system ویژگی Safety را داشته باشد تا Correctness را تضمین می کند

پس طراحی type ها از جمله جلوگیری از جمع ۴ بایت با ۲ بایت اعشاری و اشکال در محاسبات، تشخیص Correctness هست.

(int) (float)

زبان باید خاصیت Safety داشته باشد: یعنی زبان باید بطور ما را از غلط ... برآورده سازد.

زبان از abstraction های خودش محافظت کند، Safety است.
(بدون توجه به جزئیات)
تعریف آنچه ... برآورده شد.

Java، ML، Lisp ← safe است.

C ← safe نیست (مثلا در C، pointer می تواند از محدوده (boundary) آرایه هم بیرون بزند).

(خیلی در C، با pointer ها دسترسی دارد)

برای پیاده سازی زبان safe دو روش: 1) static 2) dynamic

Subject:
Year. Month. Date. ( )

Java ⟷ statically ، safe است. یعنی قبل از اجرا خطاهای انجام می‌دهد.

امنیت بیشتر

Lisp ← dynamicly ، safe است. در طول اجرا این خطاها را انجام می‌دهد:

* بهترین طراحی این است که type system هایی را بسازیم که statically باشد یا safe باشد.

یک تعدادی rule است که برای عبارات type تعیین می‌کند.

well-typed (typable) ← term هایی هستند که type آنها به دست می‌آید
⟵ تایپ دار و خوش تایپ

ill-typed ← term هایی هستند که برای آنها type نمی‌توان در نظر گرفت.

* type checker ← تابعی است که بر اساس type system ، type داشتن term را می‌دهد.

ویژگی‌ها:

1. عبارت هایی type دار همیشه محاسبه می‌شوند. (تا به یک مقدار Value برسد)

(مثلاً هر زبانی که type دار شود)

↓

Strong normalization property

2. عبارت type دار، نوع آن preserve می‌شود. نوع داده‌ها در طول می‌ماند

↓

Preservation حفظ می‌شود

3. عبارتی type دار یا value هستند یا می‌توانند به اجرا ادامه دهند.

↓

progress

Subject: Programming Languages
Year. ۱۳۹۲ Month. ۰۸ Date. ۱۲

زبان هایی که Preservation و Progress را داشته باشند، type safety تعریف می‌کنیم.

با type safety زبان، safety بصورت statically را فراهم کند.

(اگر بخواهیم در runtime ، اینجا دوباره Thread را اجرا کنیم، exception میدهد.)

[البته در Java ، statically چک میشود ولی این در semantics است dynamic]

۱) هر برنامه well-typed ، security مورد نظر ما را برآورده کند. (به صورت built-in هستند)

بین این زبان‌ها JIF (Java Information Flow) است. هر برنامه توسط JIF

نوشته شود، اثبات می‌شود که Information Security دارد)

مسئله مورد نظر در این درس، correctness و security است!

برای حل یا اولین این از یک type system استفاده می‌کنید:

۲) type ها به طور کلی اهمیت دارند:

– naming and organizing concepts

– making sure that bit sequences in computer memory are interpreted consistently

Subject: PL
Year 1392 Month 08 Date 14

- providing information to the compiler about data manipulated by the program

- Provable correctness (verifiability)

می‌توانیم requirement ها را در program بنویسیم و hint های لازم را برای type چک‌ها بفرستیم.

statically توان اثبات Correctness را.

زبان ها چون type دارند، برخی زبان ها untyped language هستند یعنی در Runtime ، type دارند

یعنی Dynamic type دارند. (type system ندارند، static ، type ها را بررسی)

documentation هر چیزی را بعدا نوشتن وقت هم بر است چون ممکن است وقت نداشته باشیم ولی

type استفاده و خیلی از خطاهای ذهنی را پیدا می‌کنیم.

وقتی می‌گوییم type error اتفاق می‌افتد یعنی داده (Entity) هایی استفاده می‌شود که با ساختار جور نیست.

مثلا اگر یک int value را به function ای بدهیم که آن را نمی‌خواهد یا operation را روی operand های نادرست

بعضی از این خطاها منجر به خطاهای سخت‌افزاری Hardware Errors می‌شوند مثل x() که x int است.

به خاطر اینکه ممکن است وقتی سخت‌افزاری رخ دهد دستورالعملی نیست که بتوانیم آن را اجرا کنیم.

یک record مثل student = {name : string, number : int} (struct) که به صورت static تعریف

کامپایلر می‌فهمد که چه مقدار فضا برای این نوع داده در نظر بگیرد

Subject: PL
Year. 1392 Month. 08 Date. 14

مفهوم type safety :

یک روش است برای برنامه‌نویس safety به منظور حفظ کردن تصورات غلط از آنچه که از abstraction های برنامه‌نویسی حاصل شده

type safety ویژگی زبان است نه برنامه.

A Programming Language is type-safe if no program is allowed to violate its type distinctions.

مثلاً زبان به function ها int value اجازه بدهد (حالت‌های اشاره‌گر) type safe نیست مثل C.

جدول مقایسه انواع زبان‌ها براساس safety :

Safety	Example language	Explanation
Not safe	C and C++	Type Casts, Pointer arithmetic
Almost safe	Pascal	Explicit deallocation, dangling Pointer
safe	Lisp, ML, Smalltalk, Java	Complete type checking

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

v ::= true | false | nv

nv ::= 0 | succ nv

Subject

Date 1392.08.14

اگر t را evaluate کنیم (قدم به قدم) به یک value می‌رسیم یا نمی‌رسیم. اگر به term ای برسیم که نتوانیم (ریاضی‌ش) به اصطلاح stuck را به کار می‌بریم (یعنی گیر کرده و جلوتر نمی‌رود)

خاصیت normalization در یک زبان یعنی همه term ها به value می‌رسند.

مثلاً pred false و stuck می‌شود. این term ها meaningless هستند.

قرار است type system به طور اتوماتیک این مشکلات را بگوید و بگوید کجاست و راه‌حل‌های آن را بگوید. و اگر خوب باشد

"a term t has type T" ≡ "t belongs to T" ≡ "t is an element of T"

$t : T$ → Typing Relation

Typing statement

در type system داریم:

$T ::= Bool \mid Nat$ → Based Type هستند

$$\frac{}{true : Bool} \text{ (Axiom)}$$

→ یعنی true از type Bool است.

$$\frac{}{false : Bool}$$

$$\frac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

→ T_2, T_3 باید type یکسان داشته باشند.

$$\frac{}{0 : Nat}$$

$$\frac{t : Nat}{succ\ t : Nat}$$

$$\frac{t : Nat}{pred\ t : Nat}$$

$$\frac{t : Nat}{iszero\ t : Bool}$$

به هرکدام از قوانین بالا Typing Rule گفته می‌شود.

این Type system که در بالا دیدیم Syntax directed است.

هر زبان سعی می‌کند براساس type system، type برنامه را به دست آورد.

مثلا برای pred false به هیچ عنوان نمی‌توان type اش را مشخص کرد.

هر term ای که typable باشد، stuck نخواهد بود.

if iszero 0 then 0 else pred 0

$$\Rightarrow \frac{\dfrac{\dfrac{}{0 : Nat}}{iszero\ 0 : Bool} \quad \dfrac{}{0 : Nat} \quad \dfrac{\dfrac{}{0 : Nat}}{pred\ 0 : Nat}}{if\ iszero\ 0\ then\ 0\ else\ pred\ 0 : Nat}$$

PL

Subject 1392.08.19/
Date

~~Type safety (Type soundness)~~

A term is well-typed ⇒ the term does not go wrong
⇍ ↓ (get stuck)

progress: A well-typed term is not stuck.

preservation: If a well-typed term takes a step of evaluation,

then the resulting term is also well-typed.

تمام این قسمت کامل نیست (در پی آمده است)

Simply typed lambda calculus:

نمی‌توان به دلیل Turing Complete بودن زبان Type System ساده ارائه کرد.

به دلیل اضافه شدن type به lambda، امکان recursive را از دست می‌دهیم

به همین دلیل بعضی از زبان‌ها به λ اضافه می‌کنند تا recursive درست کنند در λ

عناصر primitive هستند مثل Fixpoint که زبان را درست می‌کند.

plotkin's PCF → ...
↙ Programming ↘ Computing

rule هایی که در Conclusion (صحیح) عناصر Bad را تعریف می‌کنند elementory نامیده می‌شوند.

rule هایی که در صورت اعداد دارند و در صحیح مورد Type تعریف می‌کنند eliminatory نامیده می‌شوند.

- Type safety (Type Soundness)

- A term is well-typed ⟹ The term does not go wrong (get stuck) (در هنگام اجرا)

⟸̸

Type-sys. هایی که ویژگی بالا را داشته باشند، sound است. توجه کنیم که عکس عبارت بالا درست نیست. اگر این شرط محقق می‌شد می‌توانستیم بگوییم که Type-sys. ما Complete است. از دغدغه‌های مهم طراحان زبان این است که مجموعه‌ای از برعکس شرط بالا بگیرند. در این حالت می‌گویند که Type-sys. ها conservative یا restrictive است. بنابراین طراح سعی می‌کند که زبان در درجه اول Sound باشد و کمتر restrictive باشد.

- یک Term در شکل normal است، اگر دیگر (ساده) تر evaluate نشود. پس یا می‌تواند یک Value باشد و یا get Stuck باشد. برای اینکه به حالت stuck نرسیم می‌گوییم باید ۲ خاصیت برقرار باشد:

1, progress: a well-typed term is not stuck.

یعنی یا یک value باشد، و یا یک term‌ای باشد که می‌تواند evaluate شود.

2, Preservation: اگر یک term، well-typed باشد، در صورتی که evaluate شود باز هم Well-typed است.

If a well-typed term takes a step of evaluation, then the resulting term is also well-typed. (یعنی اگر ترمی تایپ دارد، بعد از eval هم تایپ داشته باشد)

- این دو خاصیت برای هر زبانی باید اثبات شوند. و این اثبات‌ها در دسته‌ی اثبات‌های استقرایی هستند. البته در این درس به این اثبات‌ها مراجعه نمی‌کنیم.

IRAN

Subject
Date

$\lambda x.t$: ⟶ (از جنس تابع است)

$T_1 \to T_2$

$T ::= Bool \mid T \to T$

مثلاً می‌توانیم type هایی نظیر را داشته باشیم:

$Bool \to Bool$

$Bool \to Bool \to Bool$ یک بولین می‌گیرد و یک تابع می‌دهد که آن تابع هم بولین می‌گیرد و بولین می‌دهد.

arrow (⟶) Right associative است.

$(Bool \to Bool) \to Bool \to Bool$

یک تابع بولین به بولین می‌گیرد و یک تابع بولین به بولین می‌دهد.

arrow را Type Constructor اطلاق می‌کنیم. مثل struct در زبان C

در این زبان ما یک type داریم و چون type constructor داریم

$\lambda x.t$ ⟶ باید بدانیم x از جنس چیست

پس دو روش داریم: ۱) Type anotatation یعنی صراحتاً type ها را تعریف کنیم. Explicitly

(یا چیزی که کامپایلر خودش پیدا کند)

مثلاً $\lambda x : T . t$

یک تابع داریم که یک x از جنس T می‌گیرد و عبارت t را ارزیابی می‌کند

89

Subject

Date 1392.08.19

۱) اگر در این روش محل سیستم تایپ آن زبان، explicitly_typed نامیده می‌شود.

۲) زبان را به تنهایی تکمیل نماید به وسیله عبارت t نوع آن را استخراج کند. به این زبان‌ها implicitly_typed نامیده می‌شوند.

$\lambda x.\ x \wedge TRUE$

x is Bool

به این فرآیند type-inferenced یا type reconstruction نامیده می‌شود.

اینگونه زبان‌ها که Polymorphism مناسب است. (مثل ML)

این قاعده (rule) typing در lambda calculus چنین است:

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

اگر x از نوع T_1 باشد، t_2 از نوع T_2 است → Hypothetical judgment

$(\alpha \to \beta) \to \gamma$

مجموعه شروط است

(این عبارت) $\lambda x : T_1 . t_2$ از نوع $T_1 \to T_2$ است.

این قانون اصلی در typing lambda calculus است.

typing context

و در حالت کلی:

$$(I)\quad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

دارای متغیر آزاد؟ مثل ... هم boundary ها را مشخص می‌کند.

Subject PL
Date 1392-08-19

ضمناً در زبان C آن‌هایی که declare می‌کنیم به آن Γ است به اصطلاح Typing Context یا Type enviroment نیز می‌گویند.

دو rule زیر نیز مطرحند:

(II)
$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(reflexivity)}$$

یا به شکل:

$$\frac{}{\Gamma, x : T \vdash x : T}$$

می‌توان نوشت.

و rule زیر:

(III)
$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

این rule ها نیز syntax directed هستند.

یعنی (I) برای abstraction ها، (II) برای variable ها و (III) برای application ها است.

مثال:

$\lambda x : Bool .\ \lambda y : Bool .\ and\ x\ y$

می‌توان نشان داد از جنس Bool است.

Subject PL
Date 1392.08.19

انتظار داریم به صورت باشد : Bool → Bool → Bool

⋮

درواقع می‌خواهیم :

$x : Bool, y : Bool \vdash and\ x\ y : Bool$

$x : Bool \vdash \lambda y : Bool .\ and\ x\ y = ?\ Bool \to Bool$

$\vdash \lambda x : Bool .\ \lambda y : Bool .\ and\ x\ y : ?\ Bool \to Bool \to Bool$

طبق قانون (I) می‌توان "?" را به دست آورد.

(درشکل بالا فرض شده تابع and بین ورودی ، given است وگرنه باید آن هم اثبات شود)

(کتاب pierce فصل ۸ و ۹)

(فصل ۴ و ۵) Mitchell

The Algol Family and ML :

از ویژگی‌های زبان Algol که در زبان‌های دیگر استفاده می‌شود :

- Colon - Seprated statements
- block structure
- static type
- function and procedure

Subject: PL

Date: 1392.08.21

Algol 60 → A general-purpose language

scientific computation [illegible]

- simple statement-oriented syntax (imperative [illegible])
- begin ... end ({...} in C)
- recursive functions (stack storage allocation)
- fewer ad hoc restrictions ⇝ [illegible] محدودیت‌ها [illegible] index [illegible]
- procedures with procedure parameters → مثل high-order function ها

```
real procedure Average (A, n);
real array A ; integer n;        ⇝ explicitly typed است Algol
begin                              آرایه محدود معلوم نیست
    real sum; sum := 0;
    for i:=1 step 1 until n do
        sum := sum + A[i];
    average := sum / n        → اسم procedure که می‌آید یعنی آن چیزی برگرداند
end;
```

Subject
Date

مشکلات و نقاط ضعف Algol :

- shortcoming in its type discipline → نقاط ضعف و کاستی در type discipline

اگر پارامترها رویه (procedure) باشند، آن رویه type پارامترهایش مشخص نبود.

- The type of a procedure parameter to a procedure doesn't include the type of parameters.
- array bounds

- Parameter passing mechanism
  - pass-by-name → copy rule داشت، یعنی expression را به جای ...

در procedure تا هر بار ورودی محاسبه می‌شد. ← ممکن است برنامه نویس را به خطا بیندازد (side effect)
ایجاد نتایج مختلف اشتباه

  - Pass-by-value → هزینه کپی آرایه‌ها زیاد بود.
چون باید کپی شود، value کپی می‌شد.

Subject
Date

Jensen's Device مثال

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

دقت: به جای پارامترهای رسمی باید مقادیر ورودی را قرار می‌دهیم.

$$\Rightarrow \sum_{i=1}^{100} i*10 = 10\,\frac{100(100+1)}{2} \neq 100*10i$$

($100*10i$ ↓ pass by value)

بعد از Algol 60 ، Algol 68 ایجاد شد. سعی شد از اصطلاحات جدیدی استفاده شود. مثلاً به جای
(Terminology)
type از mode استفاده می‌کرد، البته بعداً در نسخه‌های بعدی type را اضافه کردند.

Subject

Date

دو نوع mode داریم: primitive , compound

compound	primitive
↓	↓
array	integer
structure	⋮
pointer	
⋮	

قابلیت ایجاد type های پیچیده را داشت.

an array of pointers to procedures

به همین دلیل پیچیدگی، مثل دیگر زبان‌ها پیچیده و کند بود!

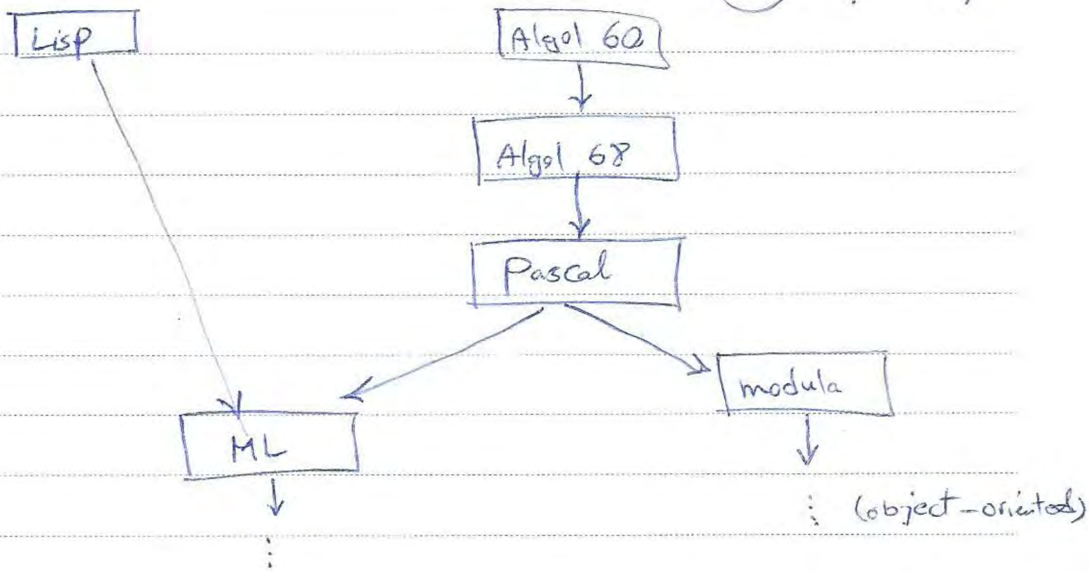
نکات مثبت Algol 68:

- type system
- memory management < stack (local variables) / Heap (other variables) global
- allocation را در اختیار برنامه‌نویس قرار داد، deallocation با garbage collector
- Pass-by-Value
  Pass-by-reference

در Pascal، type system را ساده‌تر کرد و (نسبتاً سریع‌تر) (اجازه second order function نمی‌داد)

Subject (Java, OCAML)
Date

شجره نامه زبان های Algol:



C:

با ورود زبان C ، pascal کنار رفت چون بسیاری از بخش های سیستم عامل Unix را نوشتند

Unix با زبان C نوشته شده و system callها به زبان C هستند. سیستم عامل بود.

ML:

این زبان در type system ها پیشرو است.

ML را بیشتر Function-Oriented و mostly functional دانسته‌اند.

descipline typing همچنین ویژگی imperative این زبان بسیار قوی است.

ویژگی های Lisp-like در این زبان مشهود است. (مثل higher order funcs)

این زبان نسبتا full featured است و تفاوتش با بقیه در functional بودن است.

PAPCO

Date 1392.08.25

برای proof و Analyze زبان علمی است. مثل correctness و provable بودن

بحث داغ این روزها است.

در پروژه LCF آقای milner تئوری scott را ثابت کند.
(Logic for Computable Functions)

ML → Meta Language

تجربه‌ی LCF ساخت ML است و بخاطر اینکه ابزاری بوده است Meta language است.

Curry — Howard Isomorphism:

logic های متفاوت با زبان ها معادل اند.
(programming language)

$(\lambda x : p . x) : p \rightarrow p$

این در منطق درست و اثبات شده — برنامه‌ی زبان (اعتبار) منطق

قابلیت programming Language این است که هر کاری را می‌خواهید انجام دهید. برای برنامه‌ها کار بزرگ شده است.

در اینجا Standard ML 97 کمک می‌گیریم:
(SML)

Expressions:

ورودی - <expression>;

خروجی → val it = <print_value> : <type>

Subject

Date

- (5+3) - 2;

→ val it = 6 : int

تا اینجا typing را خودکار با inference انجام می‌دهد

- if true then 1 else 5;

→ val it = 1 : int

هر عبارتی را parse و typecheck , compile , execute می‌کند.

- if true then 3 else False;

→ error (type checking)

Declarations

- val <identifier> = <expression>;

→ val <identifier> = <print_values> : type

- val x = 7+2; → نمی‌توان بعدها این x را تغییر داد.

→ val x = 9 : int

زبان های مثل C ، بعداً می‌توان مقدار declare متغیر داد ولی در ML امکان‌پذیر نیست.

Subject

Date

برای منظور assignment به معنای تغییرپذیر از reference و pointer استفاده می‌کنند.

- val y = x+3;

→ val y = 12 : int;

- val f = fn x => x+5; → function declaration

→ val f = fn : int -> int ⤳ چون نمی‌تواند هر نوع رشته‌ها را نشان دهد به جای آن به بیان type اکتفا می‌شد.

- fun <identifier> <arguments> = <expression>;

- fun f(x) = x+2;

val f = fn : int -> int

Basic types:

عوض وقتی سمبل type در آن معرفی می‌شود به introduction آن، همه type‌های تعریف شده مقادیر (جدید ...) با

elimination (از داده‌ی یک نوع خاص چگونه می‌توان هر نوع یا انواع دیگر را ساخت)

unit ⤳ فقط یک مقدار دارد

( ) : unit ← unit تنها مقدار دارای type unit است

Subject
Date 1392.08.26

void در C همان unit در ML است. (type تک مقدار)

در زبان ML نوع void داریم یعنی type بی‌مقدار.

اساساً side effect ها، type تک مقدار می‌گیرند.

true : bool — introduction form (تعاریف type ثابت) — elimination form (تعاریف type چطور به دست می‌آید؟)

false : bool

if e_1 then e_2 else e_3

andalso → and ثابت و ...   ⇒ از type bool می‌توان bool ساخت.

orelse

not

— fun equiv(x,y) = (x andalso y) orelse((not x) andalso(not y))

→ val equiv = fn : bool * bool → bool

یک روش برنامه‌نویسی: مقدار اولین عملوند را بررسی می‌کند و ...

— equiv (true , false) ;   در ابتدا x ... false نیست ... ولی اگر true نباشد، y را ارزیابی می‌کند. (در orelse اگر true باشد کار تمام است.)

→ val it = false : bool

Subject

Date

Basic type ها، type هایی هستند که آنها از پیش تعریف شده‌اند ← bool

Compound type ، 〃 〃 〃 شده‌اند ← bool*bool

bool*bool → bool

بطور کلی، type های Basic را ترکیب می‌کنند

0, 1, 2, ..., −1, −2, ... : int

+, −, /, * : int → int (overloading)

دارد یعنی برای سه تعریف مختلف شده یعنی هم برای int است هم float.

"william" : string

"Baris" : string

\- "chelsey" ^ " " ^ "clinton" ;
↳ Concatnation

→ val it = "chelsey clinton" : string

1.0, 2.0, 3.1415 : real

\- 3 + 4

→ val it = 7 : int

\- 3.0 + 4.0 ;

→ val it = 7.0 : real

Subject
Date

real(3) → real ، int تبدیل

floor( ) , ceil( ) → int ، real تبدیل
round( ) , trunc( )

type های ترکیبی، یعنی compound ها:

Tuple →

Tuple تعمیم یافته pair است.

- (3, 4);

→ val it = (3,4) : int * int

↓

product type (نوع ضربی)

- (4,5, true);

→ val it = (4,5, true) : int * int * bool

↑ introduction
↓ elimination

- #2 (3, 4); ← دومین عضو چیست (از چپ)

→ val it = 4 : int

Record type:

Record مثل tuple است ولی name دارد.

- { First_name = "Donald", Last_name = "Knuth" };

→ val it ... : { first_name : string, last_name : string }

type هم مثل name دارد.

- #first_name ({ first_name = "Donald", last_name = "Knuth" });

→ val it = "Donald" : string

List:

- [1, 2, 3, 4];

→ val it = [1,2,3,4] : int list

همه عضوها باید از یک type باشند.

- [fn x ⇒ x+1, fn x ⇒ x+2];

→ val it = [f, f] : (int → int) list

- 3 :: nil

→ val it = [3] : int list

↑ introduction
↓ elimination

Subject PL
Date 1392.08.26

```
- 4 :: 5 :: it

→ val it = [4, 5, 3] : int list
```

Value Declaration: p

الگوی pattern match : Pattern Matching

A pattern is an expression containing variables (x, y, ...) and constants (true, false, 1, 2, ...) combined by certain forms such as tupling, record, expressions, and a form of operation called a constructor.

<pattern> ::= <id> | <tuple> | <cons> | <records> | <constr>

<tuple> ::= (<pattern>, ... , <pattern>)

<cons> ::= <pattern> :: <pattern>

<records> ::= { <id> = <pattern>, ... , <id> = <pattern> }

<constr> ::= <id>(<pattern>, ... , <pattern>)

```
- val <pattern> = <exp>;
- val t = (1, 2, 3);
→ val it = (1, 2, 3) : int * int * int
```

- val (x, y, z) = t;

→ val x = 1 : int
val y = 2 : int
val z = 3 : int

- val t = [1, 2, 3, 4];

- val x::y = t

→ val x = 1 : int
val y = [2, 3, 4] : int list

- val xx::yy::zz = t

→ val xx = 1 : int
val yy = 2 : int
val zz = [3, 4] : int list

- fun f(<pattern>) = <expr> | f(<pattern>) = <exp2> | ... |
f(<pattern>) = <expn>

- fun f(x,y) = x+y;

Subject PL
Date 1392.09.03

```
- fun length(nil) = 0 | length(x::xs) = 1 + length(xs)

→ val length = fn : 'a list → int
```

type variable (در بحث polymorphism)

ML مجهز به type inference است (قبلاً گفته شده است)

مبتنی بر الگوریتم

یعنی تابع length هر type ای بدهیم کار می‌کند.

'a list → int ⟶ یک لیست از هر type ای می‌گیرد، و int برمی‌گرداند

'a → آلفا خوانده

در ML، Universal Polymorphism است.

```
- fun f(x, (y,z)) = y;

→ val f = fn : 'a * ('b * 'c) → 'b

- 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;
```

به کمک pattern ها می‌توان type declaration داشت.

Subject

Date 1392.09.03

Type declaration :

یک تعداد <constructor_clause>

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

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

از یک سری type ها یک type جدید می‌سازیم.

Example – An enumerated data type

– datatype color = Red | Blue | Green ;

این value ها type color هستند.

چون تعریف type را می‌سازند به آنها Constructor گفته می‌شود.

Example – A Tagged union data type

A = {1, 2}

B = {2, 3, 4}

$A \cup B = \{1, 2, 3, 4\}$

$A \underset{T}{\cup} B = \{(L,1), (L,2), (R,2), (R,3), (R,4)\}$

A ← عدد مربوط به مجموعه چپ

Subject PL
Date 1392.09.03

: Tagged Union ، Joined Union هم گفته می‌شود.

- datatype student = BS of name | MS of name * school | PhD of name * Faculty

تعدادی از دانشجویان لیسانس اند و فقط در ساختارشان name است.

- fun name(BS(n)) = n | name(MS(n,s)) = n | name(PhD(n,f)) = n

(BS(n) → pattern, n → expression)

→ val name = fn : student → name

(student → type, name → type)

Example - A Recursive type

درس Recursive Function داشتیم ولی اینجا Recursive type است. infinite datatype را اینجا داریم است

The set of trees with integer labels, at leaves.

datatype tree = LEAF of int | NODE of tree * tree ;

یک درخت است که برگ‌هایش دو تا درخت (زیر درخت) مقدار دارد. (constructor است که داده‌ای از نوع tree type می‌سازد (خودش یک type است))

☑ می‌خواهیم تابعی بنویسیم که یک عدد داده شود در یک درخت داده شده جستجو کند؟

Subject
Date

عبارت بولین است یا TRUE یا FALSE

- fun inTree (x, LEAF(y)) = x = y | inTree(x, NODE(y, z)) =

  inTree (x,y) orelse inTree (x,z);

→ val inTree = fn : int * Tree → bool

L-Values vs. R-Values:

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

مقدار y مهم است در واقع مقدار y مهم است نه آدرس آن که

در کجای حافظه است.

اگر متغیر سمت راست ظاهر شود ← مقدار (محتوا) مهم است.

y ، R-Value است.

اگر متغیر سمت چپ ظاهر شود ← آدرس مهم است.

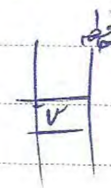
x ، L-Value است.

در ML به طور صریح این دو مفهوم R-Value و L-Value متفاوت است و تفکیک کرده است.

1) ref v : creates a <u>reference cell</u> containing value v.

یعنی یک سلول که assign کردن در حافظه

(mutable است)



Subject
Date

2) !r : returns the value contained in reference cell r.
دادن محتوای آدرس r reference cell

3) r := v : places value v in reference cell r.

- val x = ref 0 ;
تکه‌ای از حافظه را جدا کردیم، اسمش را x گذاشتیم. آن x به 0 اشاره می‌کند.
(محتوایش صفر است)
x، آدرس است و تایپش reference است.

→ val x = ref 0 : int ref   x [0]

- x := 3 * (!x) + 5 ; → این expression نیست
statement است، تایپ ندارد.
(assignment)

→ val it = () : unit   x [5]

ما assignment را در این زبان مرجع داشتند، تایپ ندارد. (unit است)

- !x ;

→ val it = 5 : int

هیچ راهی برای به دست آوردن آدرس نداریم. (نمی‌خواهیم هیچ کار!)

اگر بنویسیم x := 3*x+5 ، Error می‌دهد. چون پردازش روی آدرس نداریم.

- val y = ref "Apple" ;   y ["Apple"]

→ val y = ref "Apple" : string ref. y را می‌توانیم دست بزنیم، mutable assignable است

I

- y := "Fried green tomatos"

→ val it = () : unit

- !y;

→ val it = "Fried green tomatos" : string

در مورد رشته ها (string ها)، ML طور دیگری نسبت به اینها عمل می‌کند.

x : آدرس — در حالت مقدار می‌دهد.

y : آدرس — در حالت pointer می‌دهد.

فصل سوم تمام شد ↑

فصل ششم کتاب Mitchell : Type

مباحث مرتبط به type گفته شد، در اینجا type inference مطرح می‌شود. (به طور خاص در ML)

Type Inference (Type Reconstruction):

Inference : استنتاج

برای Type checking، باید guideline هایی به compiler بدهیم. (با Type declaration)

که به type context تبدیل می‌شد. — به طور یا گرافیک می‌کند

می‌خواهیم خواصی را به type system، زور بدهیم (enforce)، که ممکن است باعث شود declaration ها کمتر نوشته شود.

موضوع PL
تاریخ ۱۳۹۲/۸/۵

در این بحث می‌خواهیم که چطور زبان بفهمد تایپ آن چیست. در برخی زبان‌ها بعضی از declaration ها را خود کاربر توسط خودش انجام داده،

Type Inference می‌شود. در Type checking ، Type برنامه را بدست می‌آوریم و از قبل چیزی نداریم.

در Type Inference ، سعی می‌کنیم Type برنامه را تشخیص بدهیم. اگر تشخیص ندادیم type error داریم.

Type Variable (متغیر تایپی) متغیری است که تغییر بعداً جای آن، type معلوم می‌شود. ← که در جایی که type inference ... چیست می‌نامیم.

Type Expression (عبارت تایپی) ← از متغیر تایپی در حاصل می‌شود. type inference باقی کارها را حساب می‌کنیم تا به type مشخص برسد. سپس تایپ آن

مثال:
```
val f = fn : 'a list * 'a list -> bool
```

← ارتباطی بین Type Inference و Polymorphism وجود دارد.

```
- fun f2(g,h) = g(h(0));
→ val f2 = fn : ('a -> 'b) * (int -> 'a) -> 'b
```

g(h(0))
↓
تایپ h ، از int است به چیزی
h: int -> 'a
تایپ g (از چیزی که h برمی‌گرداند) به یک چیز دیگر
g: 'a -> 'b

قواعد Type checking در درون Type Inference وجود دارد.

شرط برای Type Inference وجود دارد:

1. assigning type variables to expression and subexpression.

2. Deriving a number of equations on type variables based on the parse tree of the expression and typing rules (constraints)
(constraint-based typing)

3_ Solving the system of equations by substitution (unification)

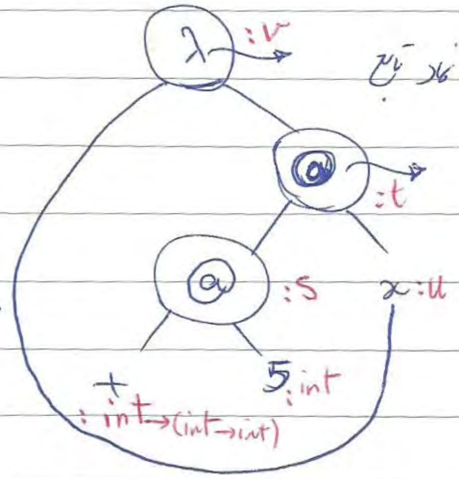
- fun $g(x) = x + 5$ → $int \to (int \to int)$

→ val g = fn : int → int

طبق همین سه قدم الگوریتم، Type را می‌دهد.

قدم صفرم

Parse tree :



این خط متغیر bound شده توسط λ را ←
abstraction را مشخص می‌کنیم.

type هایی را که می‌دانیم، روی درخت می‌نویسیم. ← قدم اول

قدم دوم

قواعدی که قبلاً در سیستم را می‌نویسیم:

قاعده Abstraction

$$\frac{\Gamma, x:T_1 \vdash e:T_2}{\Gamma \vdash \lambda x:T_1.e : T_1 \to T_2}$$

قاعده Application

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$$

در ML:

$f: a \to b$, $e : a$ $\Rightarrow fe : b$ (App)

$x : a$, $e : b$ $\Rightarrow \lambda x.e : a \to b$ (Abst)

موضوع ....

تاریخ 1392/09/05

حال معادله ها را می نویسیم:

(1) $int \to (int \to int) = int \to s$

(2) $s = u \to t$ ← طبق قاعده App

(3) $v = u \to t$ ← طبق قاعده Abst

گام سوم: حل کردن معادلات.

$\xrightarrow{(1)}$ $s = int \to int$

$\xrightarrow{(1),(2)}$ $u = int,\ t = int$

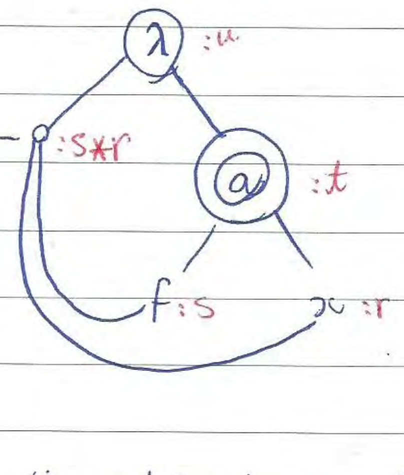
$\xrightarrow{(1),(2),(3)}$ $v = int \to int$

اعمال را می نویسیم:

- g(false)

→ Type Error!

- fun apply(f, x) = f(x);



برای اینکه بنویسیم تابع ما هم نوع رو bind می کند

$s = r \to t$

$u = (s * r) \to t$

این معادله یک کار جواب دارد و فقط به جای s آن را قرار می دهیم

$u = (r \to t) * r \to t$

→ val apply = fn : ('a → 'b) * 'a → 'b

- apply(g, 0) ⇝ در چنین جاها Type Inference داریم.

← چون چند شکلی

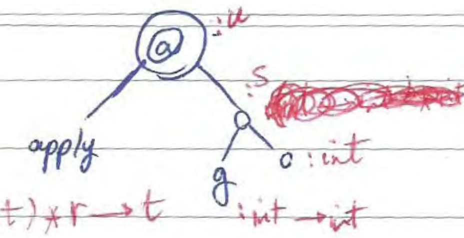
→ val it = 5 : int

M-doka

(mitchell فصل سیستم نوع)

موضوع ...
تاریخ 1392/09/05

III



$(r \to t) * r \to t = S \to u$

$S = (int \to int) * int$

$S = (r \to t) * r \Rightarrow r \to t = int \to int$

$\Rightarrow r = int \;,\; t = int$

$\Rightarrow u = t \Rightarrow u = int$

سعی بر این بوده به فرآیند Type checking را به Type Inference تبدیل شود؛ در درون Type Inference،

checking انجام شود که بررسی مجدد انجام شود.

در بسیاری از موارد Type Inference ، undecidable است.

در ML نیازی به Type anotation نیست.

CBT (Constraint-Based Typing) : همان مرحله گفته شده در بالا است که معمولاً به آن
Type Inference گفته می‌شود. (مرحله دوم آن به دست آوردن معادله (یا Constraint) است)

سه حالت ممکن است پس از Type Inference رخ دهد:

1) اساساً به هیچ type ای نمی‌رسد یعنی عبارت Type دار نیست.

2) موفق می‌شود Type پیدا می‌کند (بدون هیچ متغیر)

3) موفق می‌شود Type پیدا می‌کند ولی بعضی از متغیرها در آن می‌ماند. ← از این Polymorphism و Type Inference.

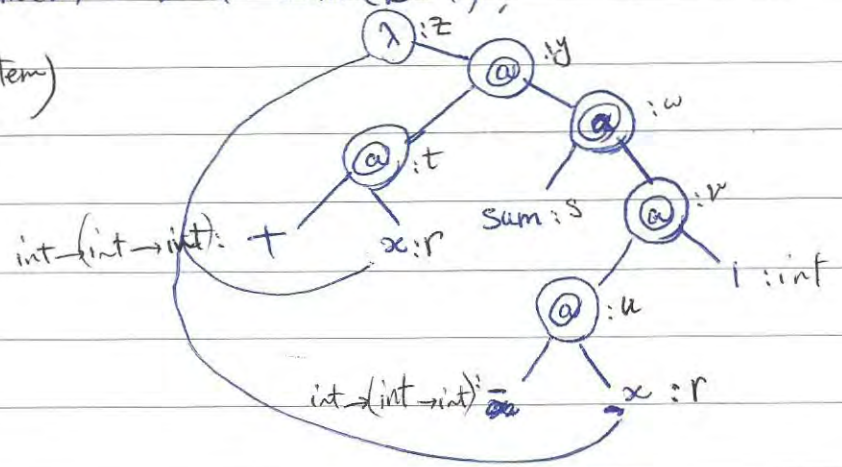
موضوع .....

تاریخ / /

Example - A recursive function

- fun sum(x) = x + sum(x-1);

(ML Runtime system)



(1) $int \to (int \to int) = r \to u$

(2) $int \to int \to int = r \to t$

(3) $u = int \to v$

(4) $s = v \to w$

(5) $t = w \to y$

(6) $z = r \to y$

(7) $z = s$

recursive بودن ← رفع

$r = int$, $u = int \to int$, $t = int \to int$, $v = int$, $s = int \to w$,

$t = w \to y \Rightarrow \begin{matrix} w = int \\ y = int \end{matrix}$, $z = r \to y \to z \equiv int \to int$, $z = s$, $s = int \to int$

→ val sum = fn : int → int

IV

موضوع: ...

تاریخ 1392/09/15

- fun append (nil, l) = l

  | append (x::xs, l) = x :: append (xs, l);

Type Inference هر دو را چطور انجام می‌دهند

برای اولی پس از اولی شده: append: 'a list * 'b → 'b

برای دومی پس از مراحل شده: append: 'c list * 'd → 'c list

چون آن لیستی که برمی‌گرداند، دومی‌اش x است پس از جنس "c" است

حال این دو باید با هم برابر باشند:

'a list * 'b → 'b = 'c list * 'd → 'c list

⇒ append : 'a list * 'a list → 'a list

- fun reverse (nil) = nil

  | reverse (x::lst) = lst   (nil هر جنسی دارد)

از اولی: reverse = 'a list → 'b list

از دومی: reverse = 'c list → 'd

⇒ 'a list → 'b list = 'c list → 'd ⇒

در خروجی ML این را می‌دهد: reverse: 'a list → 'b list

برنامه‌نویس در دیدن این Type می‌فهمد برنامه‌اش مشکل دارد. (خطای semantic اینجاست.) چون جنس لیست خروجی با جنس لیست ورودی متفاوت است. و منظور برنامه‌نویس احتمالاً می‌خواسته برگرداندن لیست را بنویسد که از جنس لیست ورودی باشد.

M-doka

ولی چون ما نمی‌دانیم آیا منظور برنامه‌نویس پردازش نوشتن است یا نه، پس نمی‌توان قطعی گفت که غلط است.

نقطه ضعف:
پس با به دست آوردن Type، احتمال اینکه زیاد خطایی نداریم، ولی ممکن است خطای semantic داشته باشد.

که می‌توان Type recursion درون تابع باشد. آوردن Type system مثل ML بسیار پیچیده می‌شود.

گاهی اوقات ممکن است معادلات به دست آمده، معادلات سختی باشند.

Polymorphism:

morph به معنی شکل است و polymorphism به معنای "چندشکلی" است. می‌خواهیم عبارت‌هایی داشته باشیم که چندین شکل داشته باشند. در زبان برنامه‌نویسی، توابعی داریم که منطق یکسانی دارند ولی داده‌ها متفاوتی می‌توانند داشته باشند.

مثل sort کردن، insert کردن، addition، ... صرف‌نظر از داده‌ها، منطق یکسانی دارند.

Abstraction
به جای اینکه برای هر داده، نوع توابع مختلف بنویسیم از polymorphism استفاده می‌کنیم. برنامه‌نویس باید این اصل را رعایت کند ولی این کار باید پشتیبانی زبان برنامه‌نویسی را داشته باشد.

مثلاً append را در نظر بگیریم و بعداً معرفی تابع بنویسیم:

```
- append([2,3], [0,1]);
→ val it = [2,3,0,1] : int list
- append([[1,2],[2,3]], [[0,5],[1,2]]);
→ val it = [[1,2],[2,3],[0,5],[1,2]] : int list list
```

موضوع ....
تاریخ / /

انواع Polymorphism :

- Parametric polymorphism :

عبارت‌ها (تابع) Type شان دارای متغیر است. وقتی می‌خواهیم آن تابع را apply کنیم به عبارت، باید دو کار انجام دهیم

اول Type Application انجام می‌شود یعنی روی Type ، apply می‌کنیم. (Type تابع مشخص می‌شود)

دوم Application حالت عادی انجام می‌شود.

- Ad hoc Polymorphism (overloading) :

عبارت‌های overloaded داریم یعنی implementation‌های مختلفی از آن عبارت داریم (مثل sum برای float, int)

در ML این نوع Polymorphism وجود ندارد.

- subtype Polymorphism :

اجازه می‌دهد یک عبارت دارای چندین Type باشد. عبارت با Type خودش ، Type هم supertype‌ها

خودش را هم دارد. مثلاً یک object از type ، subclass هم type superclass ها خودش را دارد.

این مشکل پیش می‌آید که دارا بودن Type خود subType که مخصوص به Type supertype آن!

* همه این موارد برای خودشان تئوری دارند.

Parametric polymorphism :

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

تعریف مقایسه کردن باید انجام شود

M-doka

تابعی که هر permutation ای از ورودی‌ها ایجاد کند، تابع Sort است. (اگه که اول یک بعد دوم ...!؟)

(قبلاً) تعیین متغیر type ای در مشخص تابع (صریح) ← Explicit Parametric Polymorphism
معلوم است. (در متن برنامه وجود داشته باشد) ⟹ C++

Implicit Parametric Polymorphism ← بدون اینکه متغیر type ای در برنامه صریح باشد ⟹ ML

مثال در C++ :

```
void swap (int& x, int& y){
    int tmp = x;
    x = y;
    y = tmp;
}
```

اگر بخواهیم تابع swap را برای هر نوعی متغیر بنویسیم:

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

این تابع سه متغیر (T، x و y) است.

(اول باید Type Application شود. سپس Application عادی)

این نوع Explicit parametric polymorphism است.

دقیقاً متغیر type ای در برنامه دیده می‌شود.

⊘ داشتن type template مهم است. چون کامپایلر می‌خواهد Activation Record تخصیص دهم. باید بدانم چقدر فضا تخصیص دهم.

⊘ در C++، در Link Time، Type Application اتفاق می‌افتد.

در subtype : (supsumtion) در درون خودش دارد

$$\frac{e : \tau \quad \tau <: \tau'}{e : \tau'}$$

$\tau <: \tau' \rightarrow$ τ، subtype τ' است.

e دارای تایپ τ است. τ، subtype τ' است. ⇐ e دارای تایپ τ' است.

τ' | name: | scope: |

$\tau <: \tau'$ ⇐ τ | name: | scope: | Prof.name: |

(τ' supsumes τ!)

مفهوم subclass هم بسیار شبیه به Subtyping است.

سومین نوع polymorphism، توضیحش در بالا نوشته شد ↑.

Type Application یعنی عبارت شامل متغیر تایپی است، که یک تایپ را روی آن apply می‌کنیم.

که حاصل عبارتی است با تایپ مشخص و بدون متغیر تایپی.

انواع زبان‌ها از نظر Type Application:

زبان Impredicative زبانی است که اجازه Type Application را روی Type داریم.

زبان Predicative زبانی است که اجازه Type Application را روی Type نداریم.

(Polymorphism نداریم.)

(تنها روی Typeهای ساده ممکن است قابل apply باشد)

(توضیحی در مورد اولین نوع polymorphism)

در زبان C++، Type Application در Link Time بررسی می‌شود.

یعنی برنامه شامل moduleهای مختلفی هستند که هر کدام جدا هستند. کدهای برنامه جداگانه کامپایل می‌شوند. Linker کدهای به دست آمده را مرتبط می‌کند.

در C++، با هر تایپی که به کلاس‌ها پاس داده شود، در Link Time کپی‌هایی از آن تابع جدا ساخته می‌شود.

می‌رسیم. چون هیچ Type اعلام نشده است (در ۲ نقطه) پس تایپ‌های متفاوتی به دست polymorphic تابع داشت. پس کامپایلر کد عمومی آن را درست می‌کند.

روش ML با C++ متفاوت است. کد را درست می‌کند (به جای قطعات متفاوت). از pointer استفاده می‌کند.

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

x := !y ⇐ این operation مربوط به reference هاست.

تنها چیزی که اینجا مهم است این است که assign داریم.

به این روش که ML انجام می‌دهد برای polymorphism در ورودی داده‌ها تمرکز دارد و با اشاره‌گرها کار می‌کند uniform data representation گفته می‌شود.

پس چون با pointer ها کار می‌کند برایش فرقی ندارد به چه نوعی اشاره کند، هر type ای را دارد.

خوبی این روش این است که تایپ‌های متفاوتی می‌گیرد. عیب آن این است که باید اشاره‌گر کمکی تولید و بررسی کند.

کد تولیدی در روش سریع‌تر ← C++ ، کد کمتر در روش کندتر ← ML

دیگری به Polymorphism parametric و Abstract تعمیم یافته برای اولین بار Gerrard در system F گفته می‌شود.

(نوع دیگر از نوع دوم Polymorphism) (overloading)

در ML:

+(plus): int * int → int

+(plus): real * real → real

با اینکه علامت‌ها معانی دارند، مفهوم یکسانی دارند ولی الگوریتمی که این دو دارند، کاملا متفاوت است.

کامپایلر بین این دو باید تمایز بدهد، برای تبدیل به کد ماشین، می‌فهمد که + برای int است یا real. سپس کد مربوط

(با inference)

به int (یا real) را به کار می‌برد. (overload)

Type declarations and Type equality:

زبان باید قادر باشد به type ها مساوی را بفهمد که در روش در assignment ها است. در این تشخیص داده شود،

(Type checker این نیاز دارد.)

Error می‌دهد.

اسم یک نوع جدید type declaration ها نیز باید به درستی مشخص شود Type equality هم نیز به درستی مشخص.

Type declaration
- Transparent ⇒ فقط یک alternative name می‌سازد.
- Opaque ⇒ type جدید درست می‌کند. با type های قبلی فرق می‌کند.

زبان باید این دو نوع را تشخیص دهد.

PAPCO

Subject:
Year. Month. Date. ( )

: Transparent

Type <identifier> = <type_expression>

مثال: در ML:

```
type Celsius = real;
type Farenhite = real;
```

```
- fun toCelcius(x) = ((x-32.0) * 0.55556);
→ val toCelcius = fn : real → real
```

```
- fun toCelcius(x: Farenhit) = ((x-32.0) *0.55556) : Celcius;
→ val toCelcius = fn : Farenhit → celcius
```

Type Equality
- Name type equality → فقط وقتی می‌گوییم متساوی‌اند که اسم هر دو یکی باشد
- Structured type equality → باید مقدار و شکل و ساختار یکسان باشند تا بگوییم متساوی است.

Type checking این تساوی را بررسی می‌کند.

در C++:

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

یعنی:

char equal byte

char[10] equal byte[10] equal ten_byte[10]

[illegible] name [illegible]

III 125

Subject:

Year 1392 Month. 09 Date. 11 (T)

و اما سوم :

```c
typedef struct {int m;} A;

typedef struct {int m;} B;

A x;
B y;
x = y;  → Error
```

اسم این دو یکسان است . ولی اینها را متفاوت می‌داند !
(incompatible)

درحالی که ساختاری است . یعنی زبان نتوانسته structured type equality را تشخیص دهد.

تمرین های فصل ششم : 6.1, 2, 5, 7, 8, 9, 11
(جلسه آینده)

Scope, functions, and storage management:

Block-structured languages:

از زبان‌هایی که مورد استفاده امروزی اغلب از این نوع هستند. پیاده سازی اینها در جاهای دیگر مطرح می‌شود.

Block یک محدوده از program text است که برای مشخص کردن بلوک، از begin و end یا نمادها استفاده می‌شوند. مثلاً در C از "{ }" استفاده می‌کنند. به جای "begin" و "end" ، یا در ML ،

"let" ... "in" ... "end"

- A block is a region of program text.

Subject:
Year. Month. Date. ( )

بلاک‌ها می‌توانند nested باشند.

```
{int x = 2;
    {int y = 3;
        x = y + 2;
    }
}
```

outer block — inner block

متغیرهایی که در یک بلاک تعریف شده‌اند، local هستند.
(declare)

متغیرهایی که در یک بلاک تعریف شده‌اند و در بلاک درونی استفاده شوند global هستند (برای بلاک درونی)

معمولاً در static scope مشخص می‌کند متغیر global از نزدیک‌ترین مقدار بلاک‌ها assign می‌شود.

و dynamic scope یعنی به جایی که در روند اجرا آخرین مقدار متغیر را می‌گیرد.

ویژگی‌های Block_structured languages:

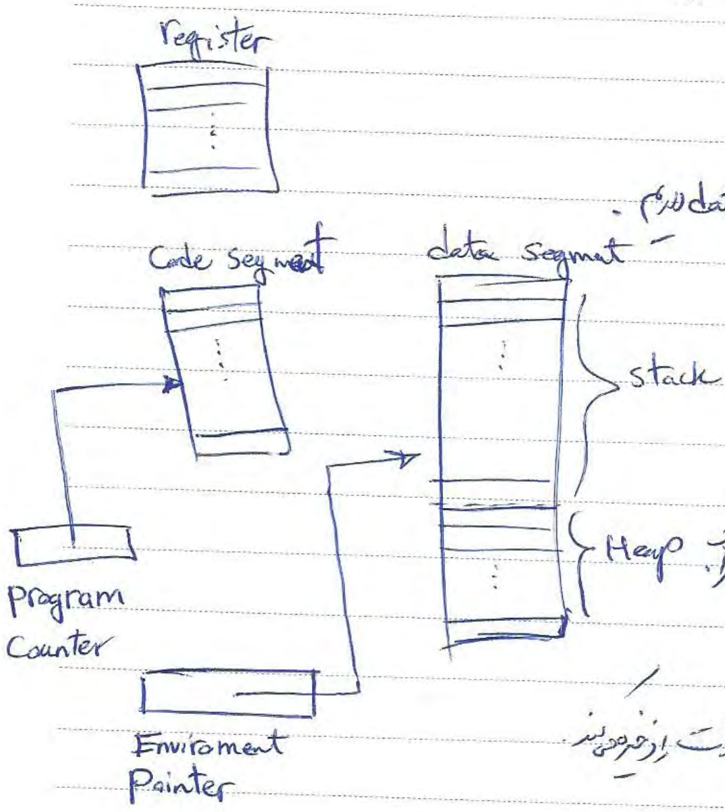
* متغیرهای جدید می‌توانند در هر جایی declare شوند
* هر declaration در یک محدوده‌ی خاص دیده می‌شود که به آن محدوده بلاک گویند
* دو بلاک نمی‌توانند با هم overlap داشته باشند مگر اینکه nested باشند.
* حافظه برای متغیرهای declare شده، تخصیص می‌یابد. (در زمان اجرا)

به محض اینکه برنامه از بلاک خارج شود، deallocate می‌شود.

سه نوع متغیر داریم برای تخصیص حافظه:

1) local variables 2) parameters 3) global variables

مربوط به توابع هستند.



بخشی از حافظه به register ها هستند.

دو بخش code segment و data segment داریم.

پس از تبدیل برنامه به زبان ماشین (کامپایل) در code segment قرار می‌گیرد.

Program Counter به دستوری است که باید اکنون اجرا شود اشاره می‌کند.

Enviroment Pointer تمام بخش از داده و متغیر (محلی) مورد نیاز را در خود دارد.

Reference Implementation:

طرح آن را در زمان Runtime به طور کامل بحث کنیم.

مهم این است در بحث پیاده‌سازی راه‌های مختلفی داشته باشیم.

هر بار وارد شدن به بلاک در ابتدای stack قرار می‌دهیم (در پایین outer block stack (انتها))

stack decipline → قرار می‌گیرد.

Subject:
Year. Month. Date. ( )

(بلوک inner در stack روی outer قرار می‌گیرد)

دو نوع بلاک داریم ← In-line blocks ← declaration ها در اواسط کار انجام می‌شوند.

Function call یا procedure call ها

داشته باشیم که اینطوری بلوک ها ایجاد می‌شوند.

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

به بخشی از حافظه که به ازای هر متغیر ساخته می‌شود activation record گفته می‌شود.

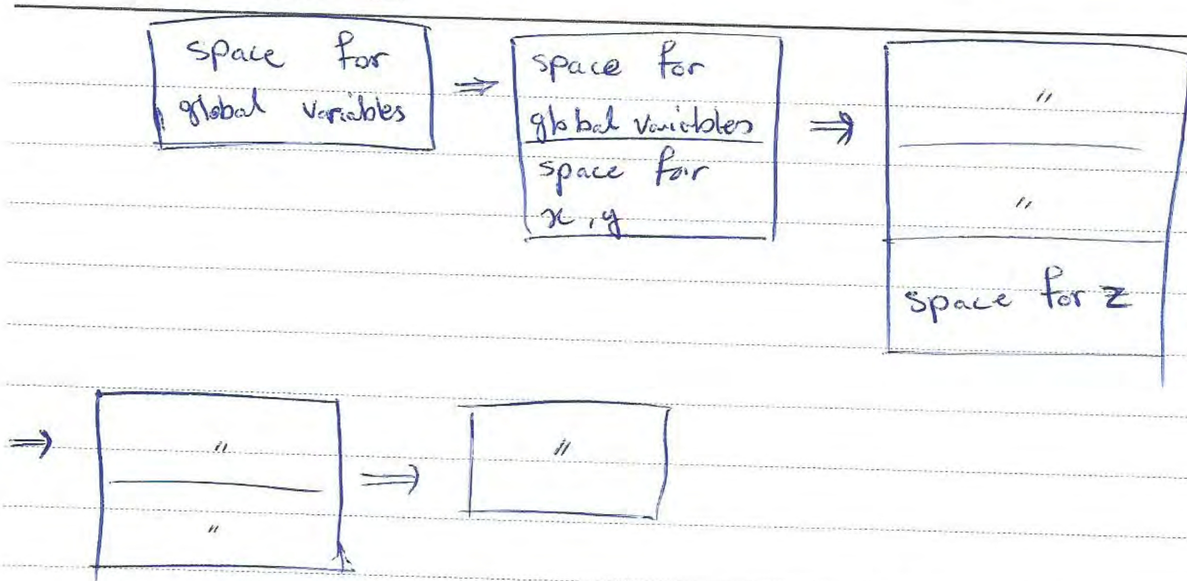
همانطور که از این کد بالا، که جا بگیرن (قبل اجرا) درست می‌کنند ← به این جزئیات خیلی توجه نداریم.

با رسیدن به begin اجرای "{" activation record را ایجاد می‌کند. با رسیدن به end "}" آن را deallocate کردن

activation record را انجام می‌دهد.

به activation record، stack frame هم گفته می‌شود.

مثالی که در stack در اجرا آمده است:



به همین ترتیب تا انتها دستورات گفته شده توسط کامپایلر، مراحل بالا انجام می‌شود.

اگر x و y و z را سراسری در نظر بگیریم بجز عملیات جمع x+y هم تخصیص حافظه دارد.
(مثلاً محاسبات جمع ذخیره شوند)

Scope vs. lifetime:

scope محدوده‌ای است که متغیر declare شده، visible است و هرجا آن متغیر دیده شود، آن متغیر تعریف شده برای متغیر کار می‌شود.

lifetime زمانی است که متغیر در حافظه حضور دارد و allocate شده است.

لزوماً lifetime و scope یکسان نباشند. مثال:

```
{int x = ... ;
  {int y = ... ;
    {int x = ... ;
     ==
    } ;
  } ;
} ;
```

lifetime x — lifetime y — lifetime x

} → scope x بیرونی (در این محدوده) نیست. (سوراخی در scope x بیرونی وجود دارد.)

Subject:
Year. Month. Date. ( )

در ML:

```
- fun f(x) = x+1;

fun g(y) = f(y) + 2;

g(3);
```

هر بلاک در نظر گرفته شده به صورت nested هستند.

می‌توان اینگونه نگه داشت که بلاک می‌سازد:

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

declarations → بین let و in

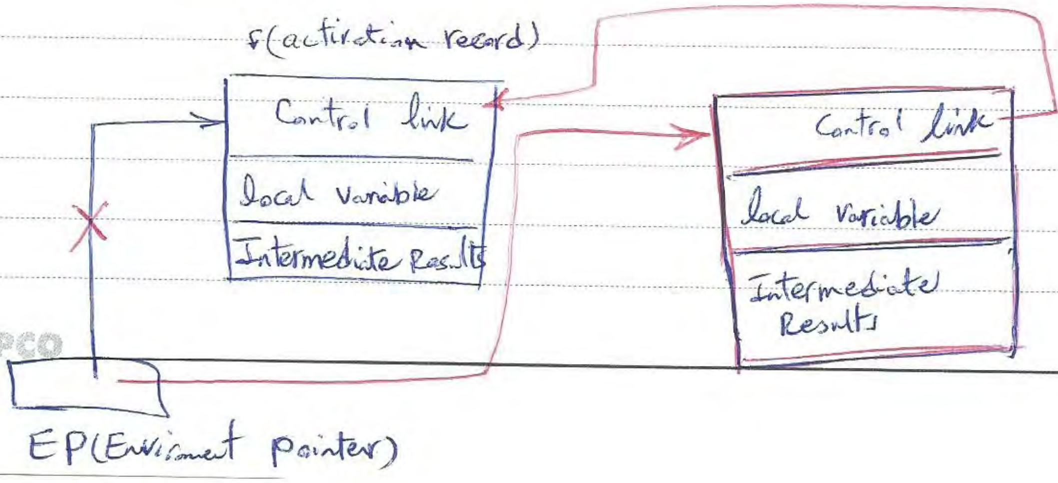
expression(s) → بین in و end

global variables:

برای دسترسی به global variable در activation record ها، کنش Control link اضافه می‌شود.

↓

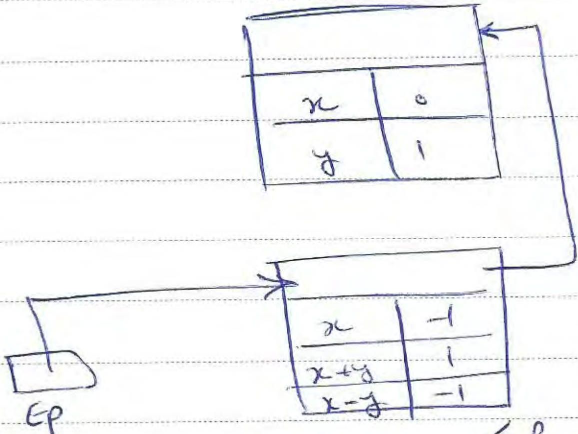
activation record به بزرگترین لینک می‌دهد.



در حالت اول : EP ، Control link اولین آن می‌شود (آخرین رکوردها) . در هر جدیدی با ایجاد activation record جدید ،

EP ، Control link دوم است و می‌کند ، به Control link دومی به Control link متن است رو می‌کند . (قرار رکوردها)

در مثال :



ممکن است مانند مثال قبل سازی انجام دهد و بعد از آن ... باشد در activation record جدید بسازد .

چون ساخت activation record مدیریت می‌کند، push و pop کردن stack ... سمت راست ... .

در Fortran ، به ازای هر function call یک activation record می‌سازد بس .

recursive call نداشت . یک activation record داشت و با فراخوانی بازگشتی ، overwrite می‌کرد .

Procedures & Functions:

procedure ها صرفاً side effect دارند . (return ندارد)

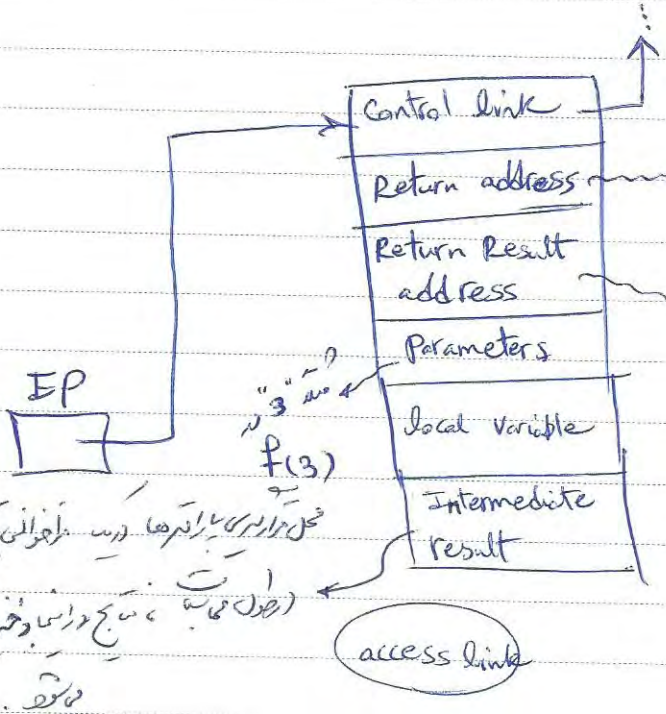
function ها می‌توانند side effect داشته باشند یا نداشته باشند . (return دارد.)

(در کل هیچ فرقی با هم ندارند (بعد از قسمت return)

Subject: PL
Year 1392 Month 09 Date 19

۶. (function call)

عمل فراخوانی function ها در زمان اجرا با activation record



آدرس دستور بعدی از Code Segment که پس از اجرای function باید اجرا شود.

آدرس شروع ناحیه (stack) data segment که activation record مربوط به function در آن ساخته می‌شود.

آدرس Calling program که پس از اتمام function باید کنترل به آن برگردد (مثلاً f(3)+4 یا f(3) که مقدار برگشتی ذخیره می‌شود).

محل ذخیره‌سازی پارامترهای ورودی فراخوانی شونده (مقادیر پارامترها در این ناحیه کپی می‌شوند، نتایج واسط ذخیره می‌شوند).

access link

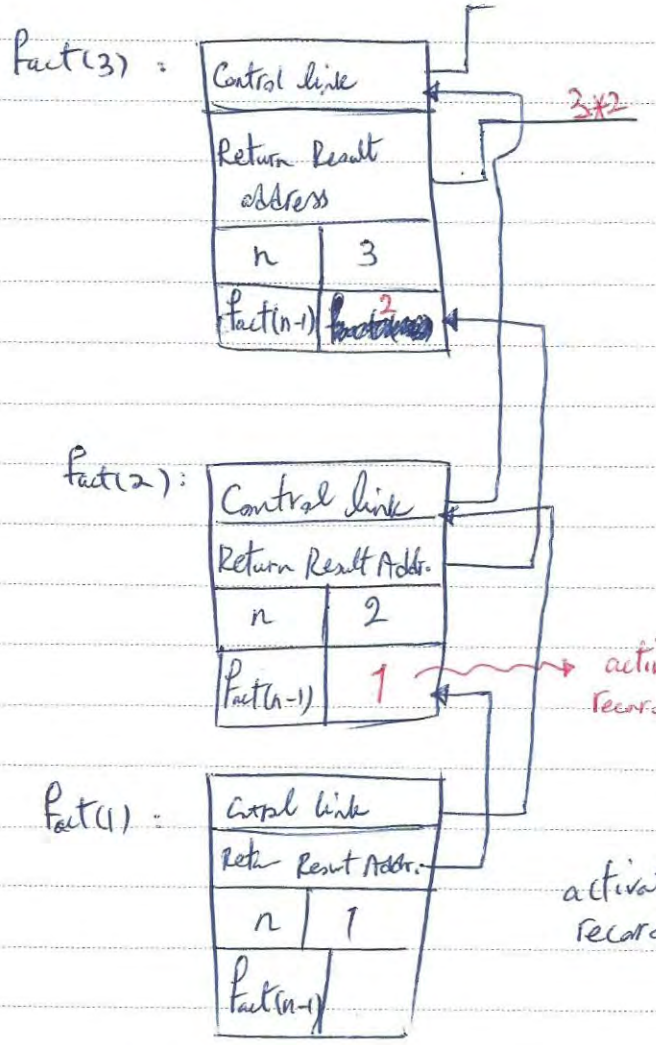
در هر call یک function یک activation record جدید ساخته می‌شود. access link برای پیاده‌سازی scope استاتیک (مثلاً global) است. (مشخص می‌کند که در زمان اجرای function از کجا باید متغیرها خوانده شوند)

مثال) fun fact(n) = if n<=1 then 1 else n x fact(n-1)

فرض کنید fact(3)+1 را محاسبه می‌کنیم:

- Calling program
- Calling block
- Calling function

} یک activation record ایجاد می‌شود.



activation record ، fact(3) برای

deallocate ، activation record

activation record

Parameter passing:

Call-by-value Call-by-name

Subject:
Year. Month. Date. ( )

formal parameters:

actual parameters:

```
proc p (int x, int y)
    if (x>y) then ... else ...
    :
    x := y*2 + 5;
}

p(z, 4*z + 1);
```

I- The time that the actual parameter is calculated.

II- The location used to store the parameter value.

اگر ابتدا پارامترها را محاسبه نکنیم پس برویم سراغ تابع. ولی ممکن است هزینه‌بر باشد.

در بین روش‌های پاس دادن پارامترها (ارسال از بیرون به درون)، از زبانی به زبان دیگر، دو دسته عمده داریم:

1) pass-by-reference (متناظر با): pass by L-value
در ML نیز هست!

2) Pass-by-value : pass by R-value

از سه جهت این دو با هم متفاوت است:

I - side effect → pass-by-value side effect ندارد.

II - Aliasing → (نام‌گذاری برای یک چیز) ⇒ pass-by-reference منجر به Aliasing می‌شود.
↓
یعنی دو یا چند name، identifier که به یک خانه از حافظه اشاره می‌کنند. (چون در call-by-value این مشکل را ندارد.)

III - Efficiency → pass-by-value ممکن است مشکل‌ساز باشد
(برای تعداد زیاد، که زمان زیادی می‌برد)
(structure)

برای pass-by-reference structure ها خیلی بهتر است.

در این بخش pass-by-reference کاراتر است.

برای توضیح این مکانیزم‌ها می‌توان از زبان ML استفاده کرد.

Semantics of pass-by-value:
(Translational Semantics)

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

... f(y); ...

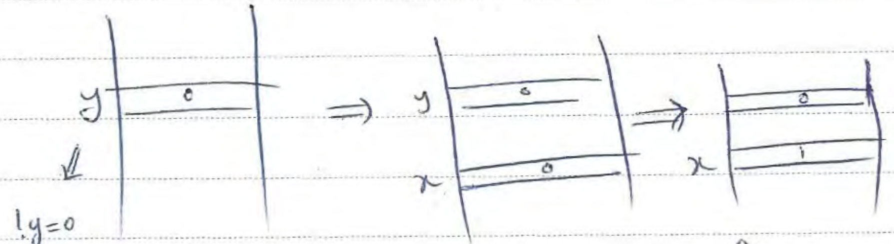
اگر بخواهیم pass-by-value باشد، به معادل زبان ML آن می‌رسیم:

$$\text{fun } f(z:\text{int}) = \text{let } \overset{\text{val}}{x} = \text{ref } z \text{ in } x := !x+1; !x \text{ end}$$

... f(!y) ...

به جای اشاره‌گر شده به آن عدد است.

نقض نمی‌شود value و قرار است pass by value باشد.



مقدار y را عوض نمی‌کند. side effect و Aliasing در pass-by-value نیست.

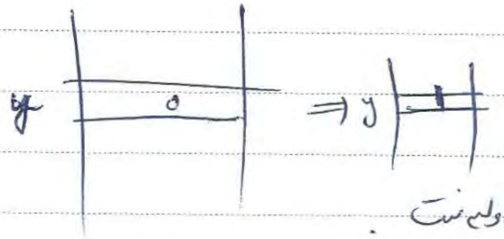
Semantics of Pass-by-reference:

$$\text{fun } f(x:\text{int ref}) = (x := !x+1; !x)$$

... f(y) ...

→ زیرا Aliasing است.

~~[illegible]~~



$\Rightarrow$ side effect دارد.

پس از تابع ـ بعد، مقدار y تغییر می‌کند و مقدار اولیه نیست.

```
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);
```
(مثال

معادل ML داریم:

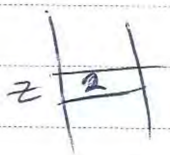
```
fun f (x : int ref, y : int)
  let val yy = ref y in
      x := 2;
      yy := 1;
      if (!x=1) then 1 else 2;
  end;
val z = ref 0;
f(z, !z);
```

Subject: Programming Languages
Year 1392 Month 09 Date 25

هر وقت می‌خواهیم by-value پاس دهیم، به reference پاس می‌دهیم (رست می‌کنیم) به عنوان چیزی که قرار است پاس شود

" by-reference مانند حالت عادی می‌نویسیم چون می‌خواهیم متوالی "!" می‌گذاریم.

در زبان کار:



از هر دو را با reference پیاده می‌کنیم، Aliasing رخ می‌دهد.

توجه. این همین مفهوم کلی

در کل برنامه‌ها Aliasing معضل بزرگی است! همین از وضعیت - Control flow

Global Variables:

C, C++, Java → static scoping → متن خود برنامه Text of Program

Lisp قدیمی → dynamic scarping → آنچه در حال اجرا اتفاق می‌افتد.

(older Lisp)

دو شیوه دارد

static scope → closest enclosing block in program text containing
نزدیک‌ترین بلوکی که احاطه کرده از سورس برنامه، در آن تعریف می‌گردد

dynamic scope → the declaration of the global variable.

آخرین activation record ایجاد شده دنبال متغیر می‌گردد. به سورس برنامه کاری ندارد
declaration

The most recent activation record containing the global variable.

متغیر مشخص شده در دو شیوه فوق می‌تواند با یکدیگر متفاوت باشد.

در زبان‌های امروزی از static استفاده می‌کنند چون برنامه‌نویس راحت‌تر درک می‌کند، در حالی که در کوه اجرای برنامه

Subject: 139
Year. Month. Date. ( )

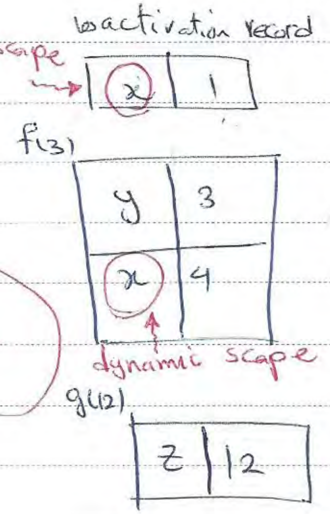
older Lisp, TEX, LATEX, Exceptions in languages, macros → dynamic scope

Java, C, ML, Algol, ... → static scope
(زبان‌های مدرن)

Example –

```
int x = 1;
function g(z) = x + z;        x + 12
function f(y) = {
    int x = y + 1;             ~> x = 4
    return g(y * x)            ~> g(12)
};
f(3);
```

static scope



در static scope میشود: (نزدیک‌ترین بلوک) 1 + 12 = 13

در dynamic scope میشود: (آخرین activation record) 4 + 12 = 16

باید بتوانیم بفهمیم static را پیدا کنیم، این می‌تواند کار راحتی باشد. (در static scope

برای این کار یک عنصر به عناصر موجود در Activation Record در فراخوانی‌ها، یک access link اضافه می‌کنیم

↓

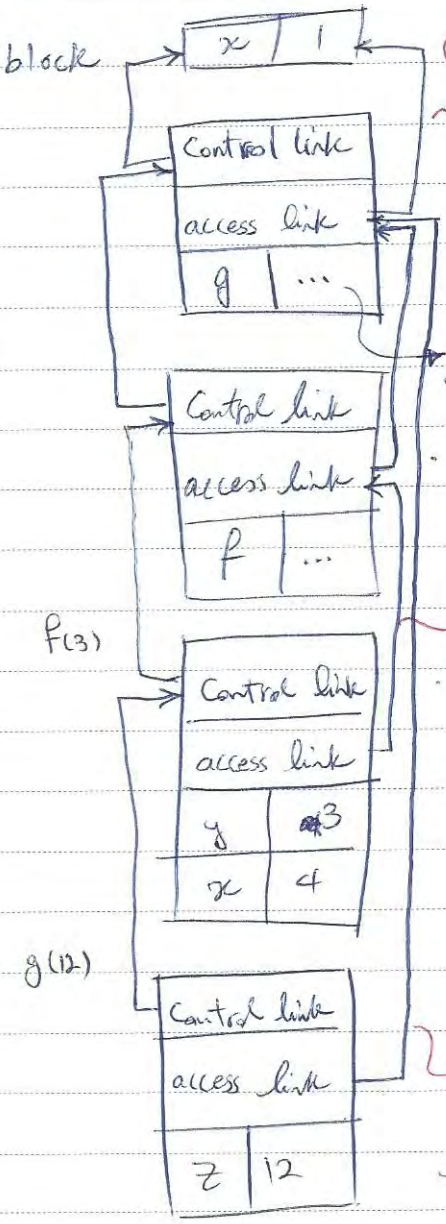
static scope اشاره‌گر به رکورد ... تا بتواند متغیر مناسب را پیدا کند.

۲- access link، static link هم گفته می‌شود.

این اشاره‌گر را با ... تنظیم می‌کنند.

Subject:

Year 1392 Month 09 Date 25



Function declaration

activation records

closest Enclosing block

activation record

Subject: 141
Year. Month. Date. ( )

Tail Recursion:

(همچنین می‌تواند از activation record ها مجدداً استفاده کند) ← Tail Recursion

tail call : در declaration یک تابع، تابعی دیگر فراخوانی شود.

این فراخوانی tail call است اگر:
اگر مقداری که از فراخوانی تابع درون تابع اصلی برگردانده می‌شود همان چیزی باشد که منظورمان از تعریف تابع است

مثال)

fun f(x) = if x=0 then g(x) else g(x)*2

↓ Tail Call

tail recursive: تابعی است که همه self invokation هایش tail call باشد
(در همه فراخوانی‌هایش، Tail Call باشد)

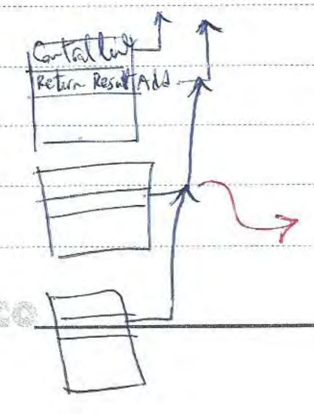
مثال) تابع فاکتوریل به شکل Tail Recursive:

```
fun tlfact (n, a) = if n<=1 then a else tlfact(n-1, n*a);
```

tlfact (3,1) → tlfact (2, 3*1) → tlfact(1, 2*3*1) → 2*3*1

صفحه 31

برای پیاده‌سازی نیاز به activation record های زیادی نیست.



چون هیچ محاسبه دیگری نیاز نیست.

Subject:
Year 1392 Month 09 Date 25 ( )

پس می‌توان حافظه زیر را داشت:

Control link	
Return Result Address	
n	~~3~~ ~~2~~ 1
a	~~1~~ ~~3~~ 6

Tail Recursion یعنی توابعی که Recursive ای دارند. مثل loop عمل می‌کنند یعنی

Iterative مطرح می‌شود که یکی می‌شود Recursive.

خیلی از کامپایلرها ... که از روش Iterative استفاده می‌کنند یعنی تبدیل را انجام می‌دهند:

```
fun tlFact(n,a) = (while !n>0 do (a:=!n*!a; n:=!n-1); !a);
```

n	3 2 1
a	1 3 6

تنها یک activation record جدید است.

Higher-order functions:

تابعی که آرگومانش یک تابع است و خروجی‌اش می‌تواند تابع باشد ← Second-order function

⋮

First class functions:

چون توابع second-order هستند.

چیزی را که در بالاتر پاس دادیم یک جفت شامل دو اشاره‌گر است. یک اشاره‌گر به activation record اشاره می‌کند (اشاره به خود تابع) و یک اشاره‌گر دیگر به کد مربوط به تابع. این دو اشاره‌گر را که

زوج مرتب پاس می‌دهیم. به این زوج مرتب، closure گفته می‌شود.

Subject:
Year 3012 Month 09 Date 25

```
fun map (f, nil) = nil
  | map (f, x::xs) = f(x) :: map (f, xs);
```

مثال)

(↑ flod & unfold)

این تابع یک تابع first class است. یعنی به عنوان پارامتر، تابع (را) به closure پاس کرد.

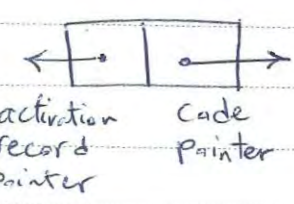
مسائل فصل هفتم:
(تا حل از امکان)

7.1, 2, 4, 6, 8, 12, 13, 15

downward funarg problem:

آرگومان‌ها که تابع هستند، به سمت تابع‌های پایینی پاس می‌شوند. که کمک در پیاده‌سازی این‌ها است می‌باشد

closure

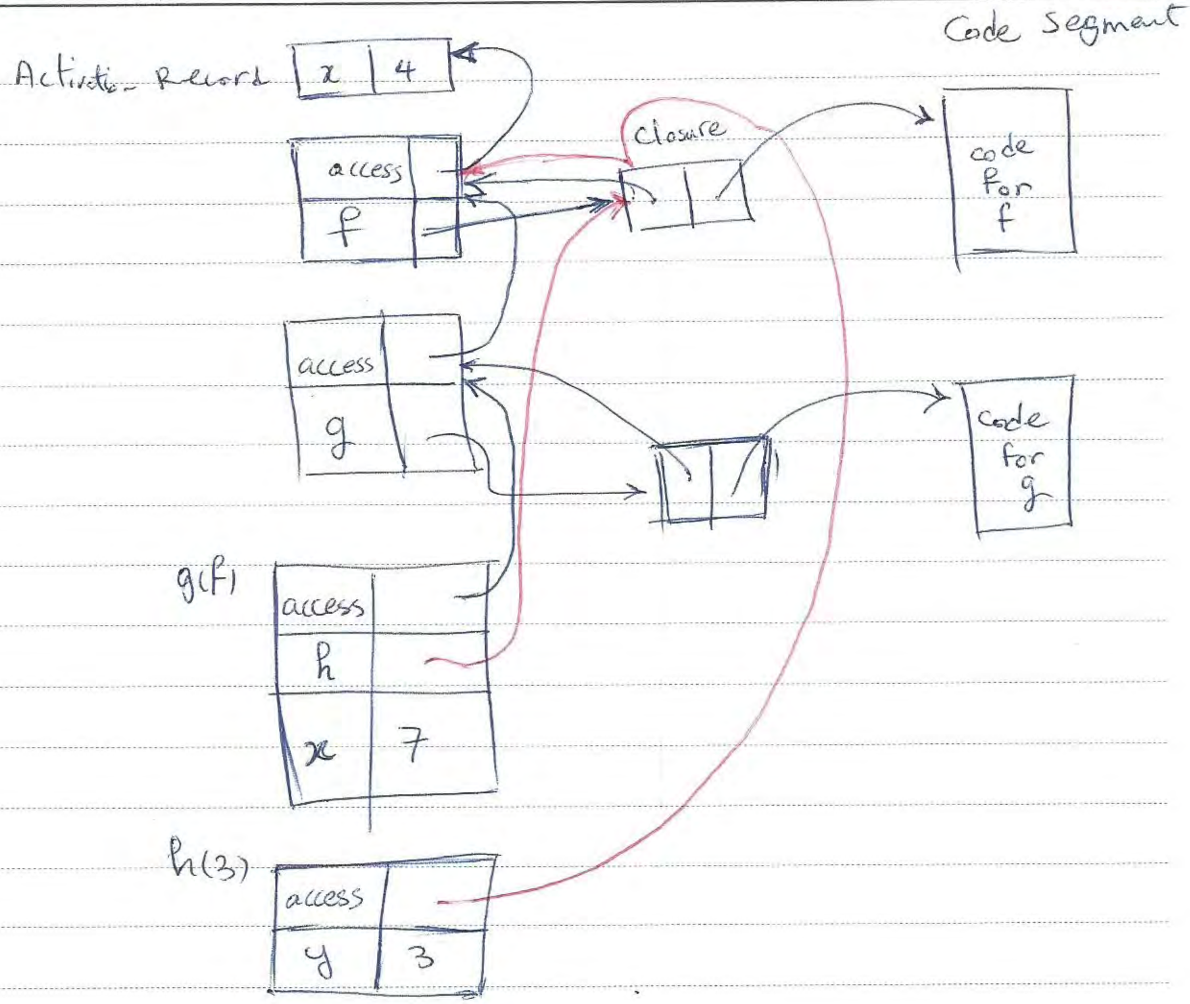


(enviroment pointer)

زبانی که توابع را به آن پاس می‌دهند ممکن است نیاز به closure داشته باشد.

مثال)

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



h is formal parameter of g.

کامپایلر نمی‌تواند دقیق دستوراتی را ایجاد کند که به access link اشاره شود.

زمانی closure ایجاد شده access link را می‌سازد.

(می‌دانیم access link را که اشاره کنیم، از closure می‌گیریم)

stack decipline وقتی می‌خواهد return value ها function باشد، به هم می‌خورد و بنابراین (& nested scope)

نمی‌شود (pop نشود) و closure هم باید در heap استفاده شود.

موضوع PL

تاریخ 1392/09/26

مثال) می‌خواهیم یک دستور داشته باشیم که یک function را به عنوان counter تعریف کنیم. این function است که یک private integer value دارد. وقتی counter را با مقدار inc فراخوانی کنیم، آن عدد را به اندازه inc افزایش می‌دهد و مقدار جدید را برمی‌گرداند. (یعنی این تابع state-full است).

```
fun make_counter (init : int) =
    let val count = ref init        ~> initialize counter شده
        fun counter (inc : int) = (count := !count + inc; !count)
    in
        counter
    end;
val C = make_counter (1);
C(2) + C(2) ;
```

→ val makeCounter = fn : int → (int → int)
(int → int) : تابع counter

→ val C = fn : int → int
→ 8 : int

state-full است یعنی C(2) اول با C(2) دوم مقدار متفاوتی برمی‌گرداند.

اول: C(2) = 3 = 1+2 (1 = initial)

دوم: C(2) = 5 = 3+2

این فراخوانی تابع statement است، و state-full است که state خودش را تغییر می‌دهد.

موضوع .....

تاریخ / /

(مثال مثلی به مفهوم object سیار ترسیم است)