Computer Programs Et09-12

dr. H. Corporaal
dr.ir. A.J.C. van Gemund
Computer Systems Group

http://cs.et.tudelft.nl/~heco
email: heco@cs.et.tudelft.nl

Session 1: The Basics

Slide 1.1.1

Who, what, where

- Docenten: dr. H. Corporaal,
  dr. ir. A.J.C. van Gemund
- Book: Structure and Interpretation of Computer Programs (SICP)
- Lecture notes

Slide 1.1.2

Communication

- Questions: email h.corporaal@its.tudelft.nl
- Practical work: practical organization
- Material: book plus lecture notes
- Help on material: lectures
- Exams: multiple choice

Slide 1.1.3

Preparation, per session

- Before Session
  - read text
  - check learning goals
- During Session
  - clarify difficulties
- After Session
  - perform practical work

Slide 1.1.4

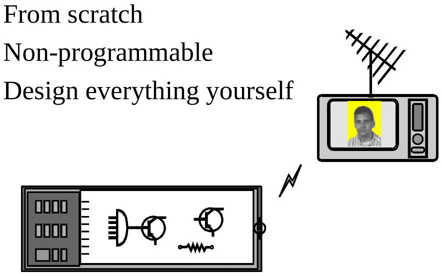
Preparation exam

- Study text
- Study lecture notes
- Check understanding of learning goals
- Make example exams
- Cross your thumbs

Slide 1.1.5

Brute force design

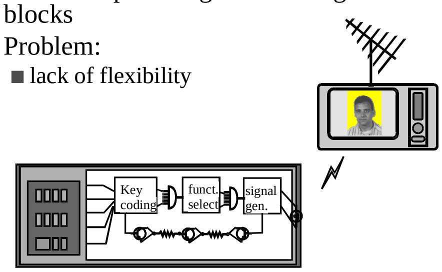
- From scratch
- Non-programmable
- Design everything yourself



Slide 1.1.6

Black box design

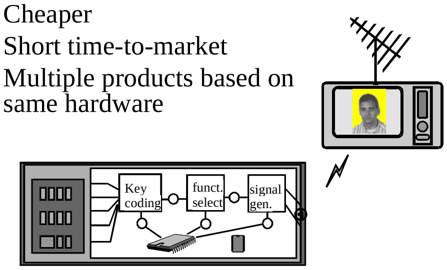
- Use fixed pre-designed building blocks
- Problem:
  - lack of flexibility



Slide 1.1.7

Programmable design

- Flexible design
- Cheaper
- Short time-to-market
- Multiple products based on same hardware



Slide 1.1.8

Goals of this course

- How to use a programmable system
- Learn the possibilities
- Translate problems into programs
- Understand how it works
- Know the cost

Slide 1.1.9

Discussion on Processing Systems

- Basic processor system
  - Core processor
  - Instruction and Data memories
  - Input and Output (I/O)
- CPU: Central Processing Unit
- How to speed-up CPUs?
  - Realization level
  - Implementation level
  - Architectural level
  - Advanced code generation and scheduling

Slide 1.1.10

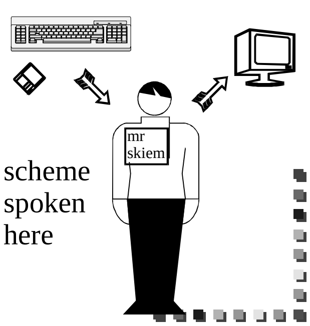
Discussion on Languages

- Languages
  - Binary
  - Hexadecimal
  - Assembler
  - HLL: High Level Languages
    - Scheme
    - C
- Compilation and Interpretation trajectories

Slide 1.1.11

Introducing Mr. Skiem

- Input keyboard/file
- Reply on screen
- Side effects
  - sound
  - network
  - printer
  - etc



scheme
spoken
here

Slide 1.1.12

Mr. Skiem, session 1.1.a,b

- Constants:
  - numbers
  - booleans
  - strings
- Expressions

Slide 1.1.13

Expressions

Options:

- Prefix
  - arbitrary number of arguments
  - easy interpretation: operator decides what to do
- Infix
  - easy readability
- Postfix
  - fits stack model of computation

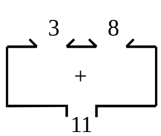
Slide 1.1.14

Substitution model: Combination

(+ 3 8)

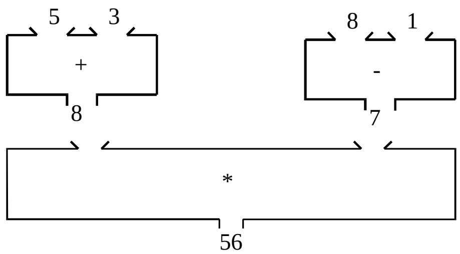
↓

11



Slide 1.1.15

Combination, evaluation



(* (+ 5 3) (- 8 1)

Slide 1.1.16

Combination, evaluation

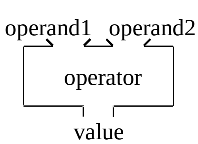
(* (+ 5 3) (- 8 1))

(* 8 7)

56

Slide 1.1.17

Combination, evaluation



(operator
operand1
operand2)

value

Slide 1.1.18

Pretty printing of large expressions

```
(+ (* 3 (+ (* 2 4) (+ 3 5)))
  (+ (- 10 7) 6))
```

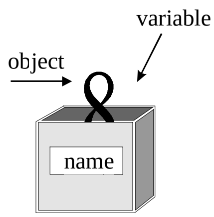
After pretty printing:

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

Slide 1.1.19

Variable/name/object

(define name 8)



Slide 1.1.20

DEFINE is a special form

- When encountering a combination containing a define, evaluation is special. The second subexpression is not evaluated !

(define jan 1)

no value

value: 1

Slide 1.1.21

Variable substitution

```
(define jan 1)
(define piet 2)

(+ jan piet)
    ↓    ↓
(+  1    2)
↓
3
```

Slide 1.1.22

Substitution model: combinations

- To evaluate a combination:
  - evaluate subexpressions
  - apply the procedure that is the value of the leftmost subexpression (the operator) to the values of the other subexpressions (the operands)

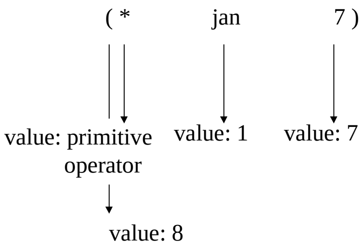
Slide 1.1.23

Evaluating subexpressions

- Values of numericals are the numbers they represent
- Values of primitive operators are the primitive machine instructions that carry out the corresponding actions
- Values of names are the objects associated with those names in the current environment

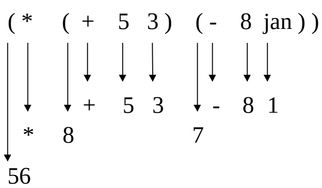
Slide 1.1.24

Example evaluation 1



Slide 1.1.25

Example evaluation 2



Slide 1.1.26

Compound procedures

- Until now: primitive operators (* / + -)
- Wanted: new operator
  example: (square x)
  substitution: (* x x)
- Method: DEFINE
  (define (square x) (* x x))

Slide 1.1.27

Compound procedure definition

(define (name formal_parameters) body)

- Define a new procedure name
- If name is used as the leftmost value in a combination, evaluate body
- When evaluating body, replace formal parameters by other values in combination

Slide 1.1.28

Substitution model: compound procedures

- If the operator is a compound procedure, evaluate the body of that procedure with each occurence of a formal parameter replaced by the value of the corresponding argument.

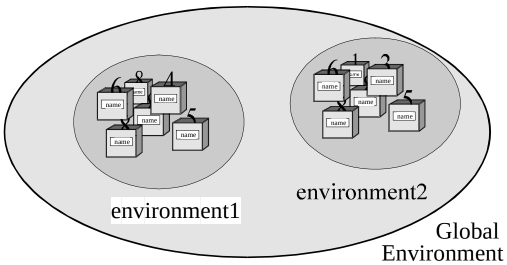
Slide 1.1.29

Environment

- Name/object pairs are stored in environments
- These pairs represent variables/procedures
- There is one top or global environment
- Environments can be nested
- A name can exist in multiple environments
  - A4 when looking at a printer
  - A4 when looking at a chessboard

Slide 1.1.30

Nested Environments



Mr. Skiem, session 1.1.c,d

- Define: name-value binding
- Re-define
- Compound procedure usage

True or False

- In Scheme, everything is true, except #F
- Sometimes, () is also false

#F → false
1 → true
(< 1 2) → true, '<' is a primitive op.
(< 2 1) → false

COND: conditional expressions

```
(define ( abs x)
        (cond ( ( > x 0) x)
              ( ( = x 0) 0)
              ( ( < x 0) (- x) )  )
```

Slide 1.1.34

COND: definition

```
(cond ( <p1> <e1> )
      ( <p2 > <e2> )
      ...
      ( <pn> <en> ) )
```

(<p> <e>) → clause

<p> → predicate

<e> → consequent

Slide 1.1.35

IF: conditional expression

```
( define ( abs x )
      (if    ( < x 0 )
             ( - x)
             x ) )
```

Slide 1.1.36

IF: definition

(if <p>
<c>
<a>)

<p> ⟶ predicate

<c> ⟶ consequence

<a> ⟶ alternative

Slide 1.1.37

Conditional expressions:

- < = > AND OR NOT are all operators that will return a true or false
- COND and IF are special forms (!!!!)
- Always use #F, never ()

Slide 1.1.38

Mr. Skiem, session 1.1.e

- Conditionals

Slide 1.1.39

Substitution model

- Normal order versus Applicative order evaluation

```
(define (sq x) (* x x))
(define (sum-of-sq x y)
  (+ (sq x)(sq y)) )
(define (f a)
  (sum-of-sq (+ a 1)(* a 2)))
```

How will (f 5) be evaluated?

Slide 1.1.40

Normal order evaluation

```
(f 5)
(sum-of-sq (+ 5 1)(* 5 2))
(+ (sq (+ 5 1) (* 5 2))
(+ (* (+ 5 1)(+ 5 1)) (* (* 5 2)(* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Delay evaluation as far as possible!

Slide 1.1.41

Applicative order evaluation

```
(f 5)
(sum-of-sq (+ 5 1)(* 5 2))
(sum-of-sq 6 10)
(+ (sq 6) (sq 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Evaluate operands completely before applying the operator !!

Slide 1.1.42