Principles of Programming Languages

Alan Wood

2014

WARNING

- These notes are 'extended' versions of the lecture slides.
- They do *not* constitute a self-contained lecture course . . . you will *not* be able to pass the exam solely by reading these notes.
- There are likely to be errors of varying degrees of importance here . . . what is taught in the lectures and practicals is definitive. Corrections may be made to the notes during lectures or practicals.

[1] As with the topic of syntax, semantics is properly dealt with in other modules (such as PCOC and CLAD), so we won't cover it in detail here.

Languages

Languages

This module is about using

$$\textit{Languages} \text{ for } \textit{Modelling}$$

$$\Rightarrow \begin{array}{ccc} \textit{Abstraction} & \rightarrow & \text{Pattern discovery} \\ + & & \\ \textit{Composition} & \rightarrow & \text{Glue} \end{array}$$

$$\textit{Programming} \text{ languages} \Rightarrow \left\{ \begin{array}{c} \textit{Computational} \text{ patterns} \\ + \\ \textit{Calculational} \text{ glue} \end{array} \right.$$

1 Programming Languages

1.1 Purposes

⚠ The following will be dealt with in depth in other modules (e.g. SYAC), However we need to cover some material here.

Programming languages can be used for a variety of purposes:

- Means for 'making' computers *compute* . . . obviously!

  But they are (obviously) *languages*, so also have the usual properties of *any* language:

- Means of *communication*

- Means of *organising ideas* ⇒ *thinking tools*

The last two points beg the question, "About *what*?"

- Natural languages are used to communicate and organise ideas 'about' many things:

  food, beauty, beliefs, literature, politics . . .

For the purposes of POPL, we will consider programming languages as being 'about' *processes*
⇒ a *useful* programming language must be able to be used for:

$$\left.\begin{array}{r}\text{describing}\\ \text{analysing}\\ \text{designing}\\ \text{building}\\ \ldots\end{array}\right\}\ \boxed{\text{Computational Processes}}$$

So we need to consider what things a programming language needs to fulfill these purposes.
However, we also need to be able to *talk* about programming languages *as languages*
⇒ we need to specify their:

- Form (syntax), and

- Meaning (semantics)

These are the fundamental Elements of Language in general

2 Elements of Language

2.1 General

Elements of Language

Abstractly a language is an infinite set of *strings* — *sentences*.

$\Rightarrow$ not *every* possible combination of characters is a sentence,

$\Rightarrow$ we need a way of specifying *which* strings are in the language,

... and these strings can be infinite.

This sounds like a difficult task, but there is a *standard* way of specifying *infinite sets* of *infinitely long* strings

... in a *finite way*:

2.2 Language and MetaLanguage

It's *vital* to distinguish between what you're talking *about*, and what you're talking *with* ...

Object Language

is the language *being described* etc.

Meta Language

is the language *of description*: the language being *used* to describe the object language.

Since all languages involve (strings of) symbols, it's *vital* that you know which are in the object-language, and which are in the meta-language.

Often this is easy, but ...

⚠ Some symbols are used in *both* the object- and the meta-languages! For example:

- the semi-colon at the end of a CUP definition
- '=' in some formal mathematics

2.3 Backus-Naur Form

Backus-Naur Form (BNF) is a way of specifying a language — set of sentences — by giving a the rules that any string in the *alphabet* of the language must obey in order to be called a 'sentence'.

The rules are called the language's

Syntax

or *grammar*

BNF could, in principle, be used to specify *any* language, including (most) "natural", or human, languages. However, we shall only use it in the context of specifying *programming* languages.

In *programming language* terms,

a *sentence* is a *program*

⇒ the BNF specification of a programming language gives the *syntax rules* that any *grammatically-correct* program in that language must follow.

There are many (slightly) different varieties — "dialects" — of BNF so ...

⚠ Be prepared to deal with different notations when reading BNF specifications.

For POPL we shall be using a very simple version ...

A BNF specification consists of a set of

Productions of the form:

```
s ::= a b c ...  | d e f ...| ...
```

where:

`s` is some symbol/name/identifier called a *non-terminal symbol*,

`a b c ...` are symbols which may be *non-terminals* or *terminals*,

`|` is read as "or",

`::=` is read as "is defined by", or "can be" etc.

BNF Facts

- Every non-terminal must appear on the LHS of *at least* one production,
- Terminals are not defined by any production,
- `::=` and `|` are symbols in the language of BNF, *not* of the language being defined:
  ⇒ they are *meta-language* symbols.
  ... since BNF is a *language* which *describes languages*.

Example

A C/C++ or Java-like declaration can be written:

type **::=** INT | FLOAT | BOOLEAN | CHAR

declaration **::=** *type* IDENTIFIER

Notes

- This specification has:
  - Two *non-terminals* . . . the LHSs of the productions
  - Five terminals . . . symbols that *don't* appear as LHSs.
    ⇒ they are *undefined*
- *declaration* is defined in terms of *type*.
  ⇒ this lets us *design* the specifications in a more *structured* way.
  ⇒ recursive (mutually self-referencing) productions are allowed (see later)
- It's *conventional* (but not required) to CAPITALIZE terminals.
  ⇒ *type* ::= INT | FLOAT | BOOLEAN | CHAR would be preferable
  . . . this makes it clearer that:
  a) these *are* terminals
  b) they *stand for* something that's defined elsewhere
     ⇒ they do not (necessarily) represent the actual string of characters in the symbol
  ⇒ we could (should?) write:
  type ::= INTEGER | FLT | BOOL | BURBLE
  and it would still represent the same syntactic entity (abstractly).

2.3.1 BNF FAQs

? How are 'symbolic' terminals such as:

`; ( , + - /*`

represented?

⇒ They stand for themselves

? What happens if I want to use a meta-symbol in the object language?

⇒ Either:

1. Quote it:
   There are many ways — in BNF notations — to quote (a string of) symbols,
   e.g. ' | ', " ::= ", . . .
   . . . but then the ' or " symbols become meta-language symbols!
   ⇒ how to quote quotes?
   or
2. Give it a *terminal symbol* name, e.g. `BAR` or `DEFINES`,

⇒ 2 is best!

? How can a *finite* set of (finite) productions define an *infinitely long* sentence?

2.3.2 Recursive Productions

Example

binary_number ::= '0' | '1'
  | '0' *binary_number*
  | '1' *binary_number*

binary_expression ::= *binary_number*
  | *binary_expression* `AND` *binary_expression*
  | *binary_expression* `OR` *binary_expression*
  | `NOT` *binary_expression*
  ⋮
  | '(' *binary_expression* ')'

It would, of course, be preferable to specify the *binary_number* production as:

binary_number ::= `ZERO` | `ONE`
  | `ZERO` *binary_number*
  | `ONE` *binary_number*

and the last part of *binary_ expression* as:

binary_ expression	::=	. . .
	⋮	⋮
	|	LPAREN *binary_ expression* RPAREN

? Where do the *terminals* come from?

⇒ It's normal to have some *external* definition of what strings are represented by terminal symbols.[2]

Usually these definitions are in a language different form BNF . . . often the language of *regular expressions* (see most editors, advanced search/replace dialogues etc.)

The process of creating a 'stream' of terminal symbols, or *tokens* (or 'lexical items', or 'lexemes') from the string of characters which is the program, is called

Lexical Analysis

The process of *checking* the stream of lexical tokens (non-terminals) against the BNF specification is called:

Syntax Analysis

Lexical analysis is done by an algorithm (program) called a *lexer* (or 'scanner').

Syntax analysis is done by an algorithm called a *parser*.

Other modules[3] look in detail into these processes.

POPL requires the ability to *read* and *understand* BNF specifications in order to discuss language structures.

[2] *Quoted* strings *are* terminals, remember.
[3] and the POPL practicals

2.4 BNF Example

```
program    ::=  stmt_list
stmt_list  ::=  stmt
           |    stmt_list SEMI stmt
stmt       ::=  ID ASSIGN expr
           |    WHILE ID DO stmt_list
           |    BEGIN stmt_list END
           |    IF expr THEN stmt
           |    IF expr THEN stmt ELSE stmt
expr       ::=  NUMBER | BOOLEAN | ID
           |    expt BINOP expr
           |    LPAREN expr RPAREN
```

Notes

program initial non-terminal.

- root of the syntax tree
- the "thing" being defined

stmt_list *recursive* definition

⇒ *infinite* sequence of statements is allowed.

⇒ Must have *at least one* base-case.

SEMI not using quoted symbols, such as ';'

stmt alternative forms for a 'statement'

⇒ a *case analysis*

WHILE. . . *stmt_list* Note the indirect recursion: a case of a *stmt* is being defined in terms of a *stmt_list* which uses the definition of *stmt* . . .

BEGIN. . . END defining a *compound statement*

. . . syntactically a *single stmt*

BEGIN. . . END could be '{','}', 'begin','end', . . . or special indentation etc. As long as the *lexer* produces the BEGIN and END tokens correctly, it doesn't matter syntactically what the language designer's choice was.

IF-THEN, IF-THEN-ELSE different forms of conditional. Use recursion again.

BINOP intended to represent *any* binary operator: +, −, / || etc.

⇒ *syntax* doesn't need to distinguish between them

IF-THEN, IF-THEN-ELSE What's the problem here?!!

⇒ This definition makes the grammar *ambiguous*.

. . . How can this be corrected?

⇒ See SYAC

2.4.1 Parsing Example

Is the following program accepted by the grammar in section 2.4:

```
x := true;
while y do
begin
   y := x & y;
   x := false;
end;
```

Work through this by hand as an exercise!

Notes This, of course, depends on the way the *lexical structure* is specified — for instance, if the string of characters "`while`" were specified to be translated into the token `LPAREN` by the lexer, then it's unlikely that the above program would conform to the grammar.

So, make some 'reasonable' assumptions about what stream of token the lexer would produce for the program and go from there!

2.5 Meaning and Correctness

Is the program fragment in section 2.4.1 *correct*?

To answer this, you need to know the result of the question posed above . . . and even then you may not be able to say whether it's correct or not! So why's that?

Obviously, if a program — a string of tokens — is *grammatically* incorrect, then it is incorrect! But what about strings of characters that are *syntactically* correct, but 'do the wrong thing'? We would regard those as incorrect too.

Notes

- There is an infinitely large number of syntactically / grammatically incorrect programs

  . . . there is even an infinitely large number of syntactically correct programs for a particular language, that are *incorrect* for *all other languages*.

  ⇒ a *string of text* is only syntactically correct or incorrect *relative to a language's grammar*
- A syntactically incorrect program is, literally, *nonsense*.

  ⇒ it cannot be given any *meaning*

So, what does it mean for a syntactically *correct* program to have 'errors'?

$$\Rightarrow \text{ it does not mean what } \left\{ \begin{array}{r} \text{you} \\ \text{the designer} \\ \text{the customer} \\ \vdots \end{array} \right\} \left\{ \begin{array}{l} \text{thinks it} \\ \text{designed it} \\ \text{specified it} \\ \vdots \end{array} \right\} \text{ to mean!}$$

A program is 'correct' *relative to a specification*

⇒ The specification must say what the program must *mean*

The *meaning* of a program is given by its

Semantics

. . .

2.6 Semantics[4]

The subject of semantics is quite complex, and requires a formal mathematical approach to be precise.

In POPL I shall *imprecisely* use English to convey the meanings of programs. However, I shall base the 'natural language' discussions on the fundamental principles of one common formal way of describing the semantics of programs — *Denotational Semantics.* An excellent full treatment of this is in the book by David Schmidt [2].[5]

Abstractly, the meaning of a program written in a language specified (formally) by a BNF grammar is a *function* which takes syntactical entities as input, and maps these to some mathematical 'object' or 'model':

$$\text{meaning} : \text{syntax} \rightarrow \text{model}$$

Informally we shall say that the meaning of the elements of a programming language is the *Computational Process* that is generated when it is executed or evaluated. Of course, this begs any number of questions such as:

- what *is* evaluation

  ... this will be answered to some extent later

- what is meant by a 'computational process'

  ... this will be side-stepped in POPL, although Abelson and Sussman's wonderful book [1] makes this clear

For our purposes we can say that the meaning of a program can be described in terms of:

- the meanings of its *primitive expressions*,
- the meanings of its *compositional mechanisms*, which form new expression from old,
- the meanings of its *abstraction mechanisms*, which encapsulate the meanings of their component expressions.

The next sections will deal with these in detail.

[4] As with the topic of syntax, semantics is properly dealt with in other modules (such as PCOC and CLAD), so we won't cover it in detail here.

[5] PDF available at: `http://www.bcl.hamilton.ie/~barak/teach/F2008/NUIM/CS424/texts/ds.pdf`

3 Elements of Programming

Programming languages are *characterised* by what they provide in *three* areas:

1. Primitive Expressions

2. Composition Mechanisms, and

3. Abstraction Mechanisms

⇒ languages differ *in essence* when these differ

⇒ languages are *essentially the same* if they provide (essentially) the same sets of characteristics . . . despite how they “look”.

Remember:

⚠ *All* languages are *computationally* equivalent

. . . *Turing Completeness*

While this is a very important concept, it merely tells us that any language can be used to create programs that compute anything that is computable. *However* it does ***not*** say that it’s as easy (or difficult) to describe a particular computation in one language as it is in another.

All that is needed to make a language Turing Complete is a way of specifying what to do next on the basis of the *current state* of the ‘universe’. In other words, all that’s needed is a ‘conditional branch’!

3.1 Primitive Expressions

1. Primitive Expressions

are the *atoms* of the languages and *represent* the simplest 'things' that the language can *express* (hence the name).

⚠ These are *not* 'values' (see later) but are *representations* of values.

- It's convenient — at the moment — to think of programming as dealing with two kinds of 'thing:
  - *data*: information that we must manipulate, and
  - *procedures*: the manipulators.

    ⇒ a language must provide primitive expressions for both *(primitive) data* and *(primitive) procedures.*
- Since everything in a programming language consists of sequences of characters, all the primitive expressions will be character sequences.

  However, they should be seen as *atomic*, *unstructured* entities.

Examples

Numbers ('numerals' etc.)	`124 124.0 0124 0x124 124L 124e10`
Truth values	`true #f 0 124`
Characters	`'x' '\n' '\033' $A '\u0231' #\x`
Strings	`"hello" 'hello' 'x'`
Identifiers (Variables, Names)	`x hello O124 %map table $sum $A 123+123`
Symbols ('atoms' etc.)	`'a 'thing :y #f`
Primitive Procedures (Operators)	`+ - && ! , ; . * if := : ==`
Miscellaneous	`null nil [] define lambda \ this super`

3.2 Composition

2. Composition Mechanisms

are the methods by that a language provides for forming *compound expressions*.

⇒ methods for forming *expressions* from other *expressions* ... whether *primitive* or *compound*.

- Correspond to *phrases* in 'natural' languages.
- *Compound* ⇒ has *identifiable components*

  ⇒ we must *know what these components are.*

Examples

- Create a single statement / expression from several.

  This includes:

  - Constructing 'blocks' of statements.

    These can then regarded, both semantically and syntactically, as a *single* statement. Notice that this a good example of a *recursive* definition ... the meaning, or the structure, of a (compound) statement is defined in terms of the meaning or structure of its component statements, which may themselves be compounds.

pascal:	`begin stmt stmt ...end`
C:	`{ stmt stmt ...}`

  - 'Nesting' expressions.

    The same comments apply to the components of an expression possibly being (compound) expressions.

- Apply a *procedure* to *arguments*

  This is, in essence, merely another example of composing expressions: the function is an expression — of a special type — that is composed with a collection of argument expressions. However, not all languages regard the function part as an expression (we shall see this later), and so *application* often needs to be treated as a distinct composition mechanism.

Java etc.:	`function ( expr )`
Haskell:	`f x (y*z)`
scheme:	`(g 2 3)`

- Sequentially or concurrently join expressions / statements

  Some languages have explicit ways of composing expression so that they get evaluated *in parallel* rather than sequentially. *Most*[6] languages have a way of composing

[6] You might want to find a language for which this isn't true!

expressions sequentially so that the order of their evaluation can be made explicit.

Java etc:	`stmt ;`	`stmt ;`	`stmt`
C:	`expr ,`	`expr ,`	`expr`
occam:	`par`	`seq`	
	`a`	`p`	
	`b`	`q`	

- Conditionally join expressions / statements
  Conditional selection of statements/expressions for evaluation is a fundamental requirement of *all* programming languages

ADA:	`if _ then _ else _ end if`
Java, C++:	`expr ?  expr :  expr`

- Functional / procedural composition In mathematics, it is possible to take two functions and create a third, using the *composition operator* ($\circ$), such that:

$$f \circ g(x) = f(g(x))$$

  Several modern programming languages have a corresponding way of composing functions. Of course, this is not possible in languages in which functions are not first-class values, and so this facility is mostly confined to the so-called 'functional' languages such as Haskell, ML and Erlang.

Haskell:	`f.g x`

- Form data structures
  Every 'high-level' programming language has mechanisms — often many — for composing data elements into structures of data elements. This will be covered more fully later in the module.

C:	`struct{ int x; int y; }`
	`union{ int x; int y; }`
C++:	`class{ int x; int y; }`
Haskell:	`[ expr, expr, expr ]`
scheme:	`(cons l xs)`

3.3 Abstraction

3. Abstraction Mechanisms

are ways that a languages provides to enable us to *hide (irrelevant) details.*

Examples

- Naming
- Procedures
- Objects (in the O-O sense)
- Packages, modules
- Interfaces
- Scope
- Data types
- ...

We shall be dealing with all these *in detail* later ...

** Abstraction is the principal way that we *control complexity* in programming.

⇒ A powerful set of abstraction mechanisms is characteristic of a useful language.

Values, Names and Expressions

Values, Names and Expressions

- Computation involves *transforming information.*
- Information in programming languages is represented by *values.*
- *Values* are *created* by expressions.

  ⇒ *Expressions* transform information.
- *Names* refer to values.

4 Values and Names

4.1 Values

are the *carriers of information* in computations.

... this is quite subtle!

- Values are *abstract*

  ⇒ we can't see (touch / smell ..) them.

  ... they have *syntactic representations*,

  but which are *not* necessarily *unique*

- Values are created by, are the result of, or are represented by *expressions*.

  ... *primitive* or *compound*

- Primitive values are created by primitive expressions.

  ... also known as 'literals'

4.2 Value Classes

Values have one or more of the following properties:

- *Denotable*

  ⇒ values that can be *named*

- *Expressible*

  ⇒ values that can be given by *expressions* (other than a name).

- *Storable*

  ⇒ values that can be *stored and retrieved* from "memory".

First-class values are those with all three properties.

Programming languages have values which are *not* first-class, and these differ between languages

A *fundamental* requirement is to know which class a language's values fall into

Examples

	Language	Denotable?	Expressible?	Storable?
Basic values	any	✓	✓	✓
procedures / methods / functions	scheme	✓	✓	✓
	Java / C	✓	×	×
constants	C (K&R)	×	✓	✓
statement labels	C (ANSI)	✓	×	×
	C (K&R)	✓	×	✓
Types	most	✓	×	×
wildcard generic types	Java	×	✓	×
arrays	most	✓	×	✓
	APL	✓	✓	✓

4.3 Names

are the way values are referenced

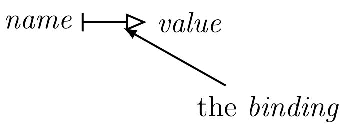
⇒ names *refer to* values.

⇒ We need *linguistic* means for associating a names with a value.

- This is called *binding* a *name* to a value.

A *binding*

is a *definition* of a name.



Binding is a *simple* but *subtle* concept.

⇒ needs a clear head!

This is *not* the same as assignment!

4.4 Bindings

Examples

scheme:
```
(define x 10)
(let (x 20) x)
```

Java / C:
```
float x;
```

C:
```
const int x = 20;
```

Take care to understand what the *value part* of a binding *actually is*! . . .

4.5 Binding to a Constant

The value *is* the constant.

Examples

C: `const float pi = 3.1415926;`

Java: `final static float pi = 3.1415926;`

$$\texttt{pi} \mapsto \texttt{3.1415926}$$

Evaluation rule:

Evaluating the name → *bound value*

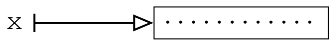
4.6 Binding to a Variable

A *variable* is, in programming terms, a *storage location* large enough to hold the representation of the value.

The qualification 'representation of' is very important to remember, but is often skipped over ('elide d') in informal use. Remember: values are *abstract* things which may be represented in different ways 'in a computer' — see section 4.1

Example

C: `float x;`



In this example, the *value* of `x` is the *storage location*

Take care with the term 'storage location' . . . this is *not* the same thing as a byte/word in a computer's memory. A storage location has an 'address' but, again, this is not necessarily the same as an address in RAM (or whatever).

- locations are *identified by addresses* (integers)

⇒ *name is bound to an integer . . .*

an address, *not the contents* of the address

Evaluation rule:

Evaluating the name → value *contained in* the bound *variable.*

NB Some texts (and computer scientists) are *imprecise*, and tend to say things like:

> "`v`'s value is `10`"

which is *wrong*, since the value bound to `v` is *not* `10`, but a 'variable'!

What they *mean* is:

> "the value *contained in* the *variable* whose *storage location* is the *value bound* to `v`' is `10`".

but life is too short to always say things like that!

I expect that I shall fall into this imprecision as well . . . beware!!

4.7 Assignment v. Binding

It is important not to confuse the ideas of *assignment* and *binding*:

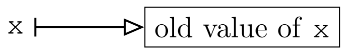
An *assignment*

such as:

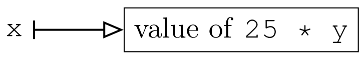
C / Java: `x = 25 * y`

changes the *contents of the variable bound to* x, *not* the binding of x to the variable.

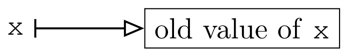
Pictorially, the *assignment* can be thought of thus:[7]



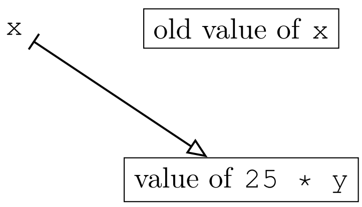
After `x = 25 * y` ...



However, if the example expression caused a change in the *binding* of x, this would be seen as:



After `x = 25 * y` ...



The difference shows up if there are *other* names bound to the same variable as x ... in the first case, evaluation of the other names gives the new value, whereas in the 're-binding' case, the other names' values are not changed.

Most languages *do* have ways of changing bindings, usually *as well as* having assignment, so take care!

[7] Assuming that x is bound to a variable!

4.8 Binding Time

A binding declaration says *what* should be bound to a name, but doesn't tell you *when* it should happen!

Is this context, 'when' is limited to two possibilities: *statically*, or *dynamically*:

Static Binding

is when the value is bound to a name *before* the process generated by the program starts running, *and* doesn't change during execution.

Dynamic Binding

is when the binding occurs *during* program execution.

NB Sebesta [3] chapter 5 has a good treatment of binding.

Examples

Java:
```
class Pair {Object left, right;};
Pair gloves   = new Pair();
Pair politics = new Pair();
```

The bindings for `left` and `right` happen when the objects are *constructed*(created). This happens during the execution — at *runtime*. The bindings are *different* in the two objects, that is they refer to different instances of `Pair`.
⇒ This is *dynamic* binding.

C:
```
int silly( int y )
    { int x; x = 2*y; return x; }
```

In this case, we still have *dynamic* binding, but the value bound to `x` *changes* every time `silly` is called — since it is a *local* name (see section 6 *et seq.*.)

4.9 Names as Values

There are two ways that *names* can be thought of as values *in their own right*:

1. *Atoms* or *symbols*

2. *Pointers*

4.10 Atoms

An *atom*

is a *name* that is *'bound to itself'*

Consequently, a binding to an $\left\{\begin{array}{c}\text{atom}\\ \text{symbol}\end{array}\right\}$ *refers to itself.*
This apparently bizarre idea gives rise to the only significant property that atoms have:

An atom is only equal to itself.

Pictorially, a binding of an atom is: 

Evaluation rule:

evaluating the atom → the atom

There are not many languages that have 'proper' atoms in the sense described here . . . some 'simulate' them — often with restricted forms of strings — without necessarily guaranteeing the self-equality property.

This may be because language designers are not generally aware of how useful atoms can be, especially whenever some form of 'symbolic' computation is being built such as Mathematics (algebra etc.) and Artificial Intelligence applications.

Example

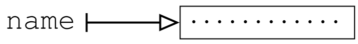
scheme: `'x` , `'thing` , `'atom` , `'1-2` In scheme, the most fundamental equality test is the `(eq? _ _)` predicate, which returns `#t`, or `#f` depending on whether its two arguments evaluate to *identical* values.

So:

```
(eq? 'x 'x)            →#t
(eq? 'thing 'x)        →#f
(eq? "thing" 'thing)→#f
```

4.11 Pointers

In a *variable binding*:

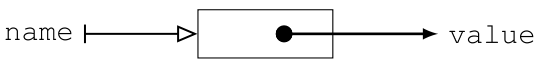


the binding is *uniquely associated* with an *address*.
⇒ we can say that the

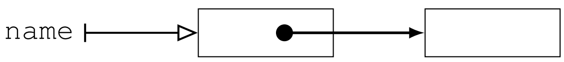
(value of the) binding *itself* '*is* the address.

Some languages have primitive expression which evaluate to the address that a name is bound to.

These *addresses* are usually called pointers . . . values that *refer to* values.
A *pointer binding* would, *in principle*, be though of:



However, *in practice* it's always visualised



⇒ the pointer value normally refers to a *variable*.

Evaluation rule:

Evaluating the name → *address* of the value referred to.

Languages with *explicitly* usable pointers[8] must have two fundamental operations (primitive expressions):

Pointer (value) creation

Need a primitive expression to provide addresses of values.

C: `float v;`
`&v` is `v`'s address

Pointer dereferencing

Once a pointer to a value is available, we need to be able 'provide' the value referred to, for instance within some expression.

⇒ the *contents* of the variable pointed to must be provided.

For example, in C

C: `*p` is the variable referred to by the pointer value contained in (the variable bound to) `p`.
⇒ 'the value that `p` points to'

The following C code snippet shows how pointer dereferencing might be used:

C:	`float *p;`	declares a pointer to a float
	`p = &v;`	'points `p` at `v`'
	`v = *p + 10.0;`	dereferences `p`
		. . . same as `v = v + 10.0;`

See Sebesta [3] section 6.9 for more on pointers.

[8] We'll see languages later that, although they implicitly 'have' pointers, those pointers are not usable — modifiable etc. — by the programmer.

Languages with Pointers

- *Explicit* pointers are in:

  C, C++, Pascal, ADA, scheme, and (sort of) in ML (a functional language)

- Java has pointers

  → all *object variables* 'are' pointers.

  ... but this isn't obvious since:

  a) they can't be changed *except* to point at another object
     → the pointer value is not *expressible*

  b) dereferencing 'looks' the same as non-pointer variables.
     → there is no (explicit) dereferencing mechanism

  However, pointers and procedures (methods) mix in subtle ways.

  ⇒ see section 5.10

> Now we have the fundamental principles of Values and Names, we can consider how to put them together — *compose* them — to create/compute new values ...

5 Expressions and Procedures

5.1 Expressions

Expressions are the linguistic means for 'creating' new values from 'existing' ones.

Since we have to start from *something*, there are two types of expression:

- *Primitive* expressions, and
- *Compound* expressions, which consist of
  - an *operator* / function / procedure, and its
  - *arguments*
  - *all* of which can be *expressions*

Terminology

- The number of arguments that an operator needs is called its arity

  This very ugly word comes from the use of the suffix '-ary' in the formal names for the varieties of this property, e.g. un**ary**, bin**ary** etc.
- *Where*, in relation to the arguments, the operator (symbol) is placed is called its fixity

  This is *also* an appalling 'word'! It come form the use of the suffix '-fix' in words such as pre**fix**, in**fix**, etc.[9]

[9]Note that the word 'suffix' *isn't* used, instead 'postfix' *is*!

Examples

arity	arguments	examples
unary	1	`*p`
binary	2	`a + 2` `(+ a 2)`
ternary	3	`a==4 ? b=10 : exit()`
4-ary	4	`(f a b c d)`

fixity	position	examples
prefix	*before* its arguments	`(* x y)` `sqrt x`
infix	*between* its arguments	`a == 4`
postfix	*after* its arguments	`b++` `23!`
"outfix"	*around* its arguments	`| x-2 |` `[ 1 2 3 ]`

The use of the word "outfix" is *not* standard (there is no agreed standard term), but is a logical extension of the others.

5.2 Expression Evaluation

The *syntax* of expressions is interesting, but doesn't vary *much* between languages.

However, we must be sure what any expression *means*.

⇒ we have to understand the *evaluation rules* for expressions in any language we are concerned with.

Expression Evaluation

is the process of obtaining the value represented by the expression.

The answer to the question "what *value* does an expression represent?" requires answers to *two* sub-questions:

a) *When* are expressions evaluated,

⇒ given all the syntactic components of an expression, *in what order* are they evaluated?

b) *How* are expressions evaluated

⇒ what are the*meanings* of the components, and how are these *meanings* (values) composed?

✷ The answers to these questions are *heavily* language-dependent.

... but there are some common themes across most languages.

Evaluation

a) *When:*

Evaluation *normally* occurs when "control" reaches the expression. To understand this we must have some concept of this term 'control', and languages differ in their models of control.

However for our purposes, and in general, we can say that expression evaluation is *demanded* by "*mentioning*" the expression.

For example, for these (sub-)expressions:

```
...42 ...(2+3)*n ..."hello"
```

their values are produced *when required by* further expressions, without having to *explicitly* cause the valuation to take place. If that were the case, then we would have to be supplied with operations for evaluating a primitive expression, and for applying an operator to argument values. So instead of writing `42` we would have to write *evaluate(42)*, or in place of `(2+3)*n` it would be necessary to do:

apply(, apply(+, evaluate(2), evaluate(3)), evaluate(n))*

Although this kind of thing would be annoying and obscure in most cases, some languages *do* provide such operations. This can be a very powerful technique to use in the right circumstances.

For example:

scheme:
```
(define n 10)
(apply (eval *) (list (apply (eval +) '(2 3)) (eval n)))
```

The second line has the same effect as `(* (+ 2 3) n)`

NB The use of the apostrophe (`'`) in this expression, which *quotes* the pair of arguments to the + operator, is crucial. We shall see why later (§5.3)

b) *How:*

The *Evaluation Rules* for expressions in a language are determined by the language's underlying Computational Model

Language's computational models differ in two ways:

Radically: which requires the programmer to acquire a new outlook when going form one language to another. This involves a certain amount of intellectual work but, since the models' differences will be obvious, there is less danger of confusion.

Subtly: which can give rise to 'dangerous' situations where are programmer *assumes* a particular aspect of the model, from past experience with other languages, say, and doesn't realise that the subtle difference is causing an error.

⚠ You must understand a language's computational model!

Examples of some (simple) computational models will be seen later.

5.3 Expression Abstraction

Abstraction

consists of *factoring out* the fixed and variable parts of something.

At the most basic level, that is, when there are *no* variable parts *abstracting* something is merely *naming* it.

⇒ for *fixed* expressions, abstracting an expression ≡ *naming* an expression

We've seen that *naming* something creates a *binding*. Consequently, for expressions we're looking at this situation:

name ⟼ expression

⚠ an *expression value*, *not* the value of the expression!

Therefore, in order to be able to abstract fixed expressions, a language must provide means for:

a) making *expression values*

b) binding them to names

c) evaluating expression values that are bound to names.

✷ No language[10] has a way of creating "pure" expression values

⇒

Expression values are not expressible!

⇒ There are no ways to create expression values *directly*.
However, the *nearest* to having language to having this facility is scheme:

Example

```
(list '+ 'x '2)          →'(+ x 2)
'(+ 1 2)                 →'(+ 1 2)
(eval '(+ 1 2))          →3
(eval '(+ x 2))          →?
(define x 10)            →
(eval '(+ x 2))          →12
                         →
(eval '(eval (+ 1 2)))→?
```

This works because in scheme (LISP) all *compound expressions* are of the form:

`( op arg1 arg2 arg3 ...)`

where `op, arg1, arg2, arg3,` are expressions,

and `op` evaluates to a *procedure value*

⇒ a compound expression *is* a list of values

✷ In scheme, *lists* are primitive values

⇒ compound expressions and lists are the "same thing". These are called *S-expressions*.

So in LISP-like languages, the three requirements for having abstraction of fixed expressions are met as follows:

a) make expression value	`'(+ 2 3)`
b) bind to name	`(define exp '(* 2 x))`
c) evaluate named expression	`(eval exp)`

[10] That I know of!

NB Some languages have ‘evaluators’ that take *strings* and evaluate them as if they were fragments of program.

Example

Python is such a language:

```
eval( '1+2' )           →3
eval( eval( '1+2') )    →
eval( 'eval( "1+2")' )→3
```

However, this is not the same as abstracting *expressions* as an expression-value is *not* a string-value.

5.4 Procedures

Procedures[11] are *values* that represent expressions in which some sub-expressions are *fixed*, while others are *variable*.

Following the normal principles of abstraction, the *variable* parts are given *names*, which represent values which will be determined when that name is evaluated.[12]

In a *procedural abstraction* the *variable* parts are called the (formal) parameters.

The procedure's *expression* — the fixed and variable parts together — is called the (procedural) body.

⇒ There must exist *syntactic* (linguistic) means for specifying the parameters and the body.

When it is needed to evaluate the *body* of the procedure — and there may be several ways in which this could be done depending on the computational model — requires that the *parameters* be *bound to values*.

The *values* bound to the *parameters* when a procedure's body is evaluated are called the *arguments*.

[11] Procedures are also known as: sub-programs, subroutines, functions, function subprograms, methods ...

[12] Recall that the evaluation rule for a *name* is that it is evaluated to the value to which it is bound.

Examples

Java / C: `int f( int x ) { return 2*x; };` — Definition
`f( 23 );` — Evaluation

- binds `f` to a *procedure value*
- the procedure takes *one argument*
- the argument will be dynamically bound to the parameter `x` when `f` is invoked

scheme: (method 1) `(define (f x) (* 2 x))` — Definition
`(f 23);` — Evaluation

- Same description as Java

Haskell: (method 1): `let f x = 2*x` — Definition
`f 23` — Evaluation

- Same description as Java

`let h x y = x * y` — Definition
`h 3 4` — Evaluation
`h 2 4` — (re-)Evaluation

- binds `h` to a *procedure value*
- the procedure takes *two* arguments
- the argument will be dynamically bound to the parameters `x` and `y` when `h` is invoked
- the parameters are *re-bound* to the arguments in the second evaluation

5.5 Procedures as Values

In section 5.4 is was said that " ... procedures are values ... ", and that the procedure *definitions* in the above examples "bind the [procedure name] to the procedure value.

That is, the definitions above have this effect:

`name` $\longmapsto$ procedural value

This should immediately make you ask, "What *class* of value (§ 4.2) is a *procedure*?"

Earlier (§ 4.2) we saw that procedures / functions /methods varied according to the language:

	Denotable?	Expressible?	Storable?
Java / C	✓	×	×
scheme / Haskell	✓	✓	✓

So ...

In some languages (scheme, Haskell ...) procedure values are *first-class*

⇒ they are *expressible* values

⇒ can be *results of expressions* Therefore, in languages in which procedures / functions are *first-class*, we must have *syntactic* means for creating procedural values.

That is, we need a *primitive expression* which evaluates to a procedural value. That primitive is called a *lambda expression*.

Lambda Expressions

A *lambda expression*

is the fundamental mathematical way of creating a procedure (function).

It is rather unfortunate that it has this strange name, as this tends to frighten people off a simple concept. However, it comes from the branch of mathematics called the *lambda calculus* — another off-putting term[13] — which studies the nature of functions and the abstraction of expressions, and so we use it!

In fact it's quite useful to know that, in the Lambda Calculus, the standard notation for a (lambda) function is of the form:

$$\lambda \quad \text{parameter-names . function-body}$$

In programming languages the syntax, of course, varies with the language:

[13] It has nothing to do with differentiation and integration!

5.6 Lambda Expressions

Lambda Expression syntax

scheme: `(lambda (` *parameters* `)` *body*`)`

Haskell: `\` *parameters* `->` *body*

python: `lambda` *parameters* `:` *body*

javascript: `function(` *parameters* `) {` *body* `}`

✷ Each of these a primitive expression that evaluates to a *procedure.*

Therefore, we can bind names to procedures *directly* in these languages, using the same binding mechanism(s) that bind names to *any* values.

Java 8: `(` *parameters* `) ->` *body*

C++11: `[](`*parameters*`) {` *body* `}`

Not clear if these are *first-class* function values.

Lambda binding Examples

scheme (method 2)

```
(define g (lambda (x y) (* x y)))
(g 2 20)
```

Haskell (method 2):

```
let h = \ x y -> x*y
h 2 20
```

⇒ The examples given in the table on page 50 can be seen as *syntactic alternatives* to the explicit binding of a name to a procedure given by a lambda expression.

For instance:

```
(define (g x y) (* x y))
```

has exactly the same effect as the scheme example above, and the Haskell (method 2) above is exactly equivalent to the Haskell (method 1) on page 50

The crucial point about *first-class* procedural values is that they can be used in the same ways as other values, for example they can be:

- stored in parts of *data structures*,
- passed as *arguments* to procedures,
- *returned* as the *results* of procedures.

5.7 Procedure Evaluation

If a language provides procedures (expression abstractions), it is necessary to know how they can be *evaluated*.

The terms used for demanding the evaluation of a procedure vary according to the language in question, but the most common are:

a procedure { call / application / invocation }

We must also distinguish between the evaluation of *primitive*, or 'built-in' procedures, and *compound* or user-defined procedures:

Primitive Procedures are those which are supplied by the language. The only way in which a programmer knows how these are to be evaluated is to *read the documentation of the language*. These evaluation rules can be different from other procedures, compound or primitive, in the language, or between languages. For instance, Boolean 'OR' primitive procedure[14] may evaluate *both* arguments, or the first argument and then the second *only if* the first evaluates to 'false', or the other way round!

Compound Procedures are those defined by the programmer[15] using any of the methods allowed by the language (including 'anonymous' procedures which are the result of lambda expressions).

To understand how *these* are evaluated requires that the user understands the *Computational Model* of the language.

There are several distinct computational models, and you will see these in other modules as you deal with different languages. However, one of the most straight-forward is:

The Substitution Model

To evaluate a procedure call:

- replace each parameter occurrence in the body with its corresponding argument
- evaluate the body with these substitutions.

See SICP [1] §1.1.5 for more details

⚠ This is only OK where we don't have any 'side-effecting' operations in the language, such as *assignment*.

[14] Note that, although this is generally called an 'operator' and is usually *infix*, it is still a (primitive) procedure.

[15] . . . or the writer of a library that the programmer is using.

⇒ It is a *model*

⇒ Some ‘pure’ languages, such as Haskell, conform to this model

It is a *model*, and *not* an implementation specification.

A more general model, the Environment Model [1], is needed to allow for assignment.

A short example illustrates this simple model . . .

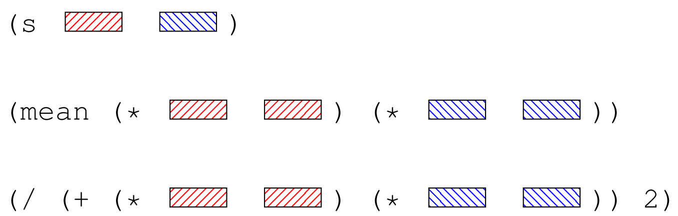
This is not an adequate model for *all* languages

Example

scheme:

```
(define (s x y) (mean (* x x) (* y y)))
(define (mean a b) (/ (+ a b) 2))
```

then:



→ no more compounds ⇒ evaluate the primitives

Now that we have an idea how the *body* abstraction is turned into an evaluatable expression by a procedure call, we need to ask:

? What exactly *is* substituted for the parameters in the body

⇒ We must now focus on the parameters, the arguments, and how one 'becomes the other'.

5.8 Nullary Procedures

Before considering parameters in more detail, we need to consider the special case of

procedures with **no** parameters

.

Nullary Procedures are those whose *arity* $= 0$

⇒ have *no parameters*

⇒ take *no arguments*

- A general way of *naming* (abstracting) *expressions*
  - ⇒ *delay the evaluation* of an expression,
  - ⇒ *evaluate it later* by invoking it (with no arguments)
- also known as *Thunks*

Example (scheme)

`(define 2xThing (* 2 thing))`

evaluates `(* 2 thing)` *now*

`(define (2xThing) (* 2 thing))`

evaluates `(* 2 thing)` when the 'thunk' `2xThing` is invoked by `(2xThing)`

5.9 Parameters

The *variable part* of the abstraction that is represented by a procedure is factored out as its parameters.

⇒ To *evaluate* a procedure the parameters must be *bound* to the *arguments* supplied in the procedure invocation.

This process of binding arguments to parameters is known by several terms:

⚠ Unfortunately, the most common term — parameter passing — is the least precise, since it is *arguments* that are passed, not parameters! However, we shall use the common term due to its history!

$$\left.\begin{array}{r}\text{parameter}\\ \text{argument}\end{array}\right\} \;\text{—}\; \left\{\begin{array}{l}\text{passing}\\ \text{transmission}\end{array}\right.$$

⚠ Just to confuse matters further, one often finds — in older texts — alternative terms for *parameters* and *arguments*:

Alternative Terminology

- 'parameters' ≡ 'formal parameters'
- 'arguments' ≡ 'actual parameters'

5.10 Argument Transmission

As mentioned above, this is also known (imprecisely) as:

- Parameter Passing, or
- Parameter Transmission

Parameters are *bound* to the corresponding arguments when a procedure call is evaluated, then the *body* is evaluated. But this, correct, statement leaves two vital questions unanswered:

a) What 'property' of the argument is bound?

b) *When* is the bound property evaluated?

Remembering that *arguments* are *expressions*, and *parameters* are *names* . . .

5.11 Parameter Binding

There are essentially *three* choices for the bound *property*:

a) Bind the *value* if the argument.

This gives the argument transmission method called call-by-*value*[16]

b) Bind the *address* of the argument

This is called call-by-*reference*

c) Bind the *actual expression* that is given by that argument.

This is called lazy evaluation, of which there are two varieties:

⇒ call-by-*name*

⇒ call-by-*need*

Note It is the *language designer* who decides which parameter-passing mechanisms are available, *not* the programmer.

⇒ If the language doesn't have the mechanism *syntactically*, then it would have to be explicitly programmed by the user.

[16]Sebesta [3] calls this *pass*-by-value, which is a *much* better term. However, 'call-by-value' is common usage, so we will stick to this!

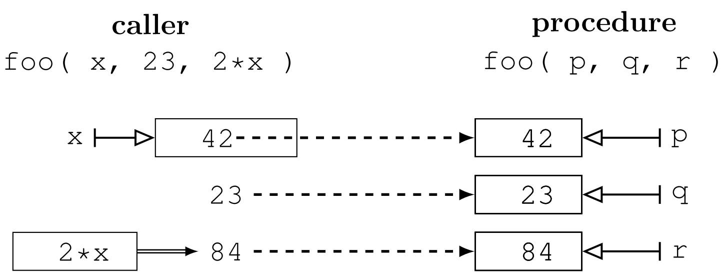
a) Call-by-Value

The parameter is bound to a (fresh) variable.[17] Then the argument expression is evaluated, and the resulting value is copied into the newly bound variable.

At least, this is what happens *conceptually* . . . the actual implementation may be different!

⇒ Access to the parameter within the procedure body can have no effect on the argument. For instance, assigning to the parameter within the body will do nothing to the argument, and so will be 'invisible' to anything 'outside' the procedure body. See §6 later for a full treatment of these ideas.

The situation can be illustrated thus:



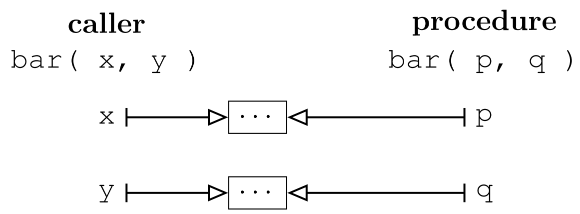
[17]That is a location, large enough to hold the argument's value, that is not bound to any other name in the programme at that point.

b) Call-by-Reference

When arguments are passed *by reference*, it is an argument value's *location* that is bound to a parameter name.

This implies that the argument expression is inspected first to see if it's value already has a location — for instance, if it is name bound to a variable — or whether it needs further evaluation before the value 'has' and address, as would be the case for an argument which is a compound expression. In the latter case the expression would be evaluated and placed in a variable, and the variable's address would then be bound to the parameter name.

A simple pictorial example is:



✷ The crucial difference between call-by-value, and call-by-reference is that in the latter case, any operation within the procedure body that modifies the parameter's value *also* modifies that of the argument. Thus, any references to the argument's value 'outside' the procedure body will 'see' the effect of the internal operation.

5.12 Lazy Evaluation

is a general term for *two* types of argument passing mechanism:

- Call-by-name, and
- Call-by-need

The principle behind both of these is that an expression is evaluated *only when* it can be proved that *its value will be required.*

Some languages — principally the modern 'functional' languages — use lazy evaluation implicitly, and so the programmer doesn't have to do anything special to make this happen.[18] However, providing a language has *first-class procedure values*, then the effects of lazy evaluation can be achieved by using *thunks* explicitly.

Lazy evaluation can be implemented in many ways depending on the language, but it is helpful to 'imagine' how a language could be lazy', as follows:

Conceptually:

- argument *expression* is 'wrapped' in a *thunk*
- parameter is bound to the thunk
  ⇒ parameter evaluation in procedure body is a *thunk invocation.*

For example, the situation we saw in the Scheme practical with the definition of the `(either a b)` procedure, is a perfect example of using *explicit* thunking to simulate lazy evaluation.

With this model in your mind you can see that the evaluation of the argument expression will be *delayed* until the thunk is invoked within the body of the procedure.

Multiple parameter occurrences

If the argument evaluation is *delayed* until it's required in the procedure body, what happens if its value is required in *several* places?

This question boils down to asking whether a 'lazily evaluated' argument is evaluated more than once?

Answer: *it depends* . . .

- In languages *without* side-effecting operations — primarily the modern functional programming languages such as Haskell — it would be evaluated *once* when the value is first needed, and then the value is '*shared*' with other instances of the parameter to which the argument is bound

[18] Although, of course, the programmer *must* have this computational model in mind all the time when writing their programmes!

- For languages *with* side-effects the argument must, in general, be re-evaluated every time the bound parameter's value is required. In essence, this is because any computations between places where the parameter's value is needed might have had side-effects which could change the value . . . for instance, a variable in the 'thunked' expression might have been assigned to.

Lazy Evaluation is (normally) automated, so that there is no need to explicitly 'thunk' argument expressions, nor to explicitly invoke the thunk in a procedure body.

In essence, this means that the placing of argument expressions in a procedure call's argument list 'looks like' it would if there were *applicative-order* (non-lazy) evaluation.

call-by-name	=	lazy evaluation *without* sharing
call-by-need	=	lazy evaluation *with* sharing

call-by-name can also be *though of* as a "textual" replacement of the parameters in the procedure body by the *actual* argument expression.

However, in this case, the *names* in the argument expression must be carefully checked to see if any of them 'clashes' with names in the procedure body expression. If so, then they must be consistently changed to overcome this.[19]

To understand this more fully, see SICP [1] on the Substitution Model.

[19] In the mathematical theory of the Lambda Calculus, this process of alteration is called α-conversion.

5.13 Argument Passing Comparisons

call-by-	Popularity	Clarity	Safety	Cost
value	*high*	*easy*	*good*	*expensive*
reference	*medium* in modern languages	*easy*	*bad*	*cheap*
need	*common* in 'pure' languages, *uncommon* in other languages	*easy*	*excellent*	*cheap* in pure languages
name	*very unusual*	*awkward*	*bad*	*cheap*

5.14 Argument Transmission Examples

scheme:

call-by value (mostly)

- exceptions are the 'special forms': `if, define, set! ...`

call-by-name (lazy evaluation)

- using the `delay` and `force` special forms

C: *call-by-value*

- call-by-reference is *simulated* by passing pointers (by value)

C++: ***call-by-value***

call-by-reference

Haskell: *lazy evaluation* (call-by-need)

Java: *call-by-value*

6 Scope and Environments

6.1 Scope

The *scope* of a binding

is the set of expressions where it holds.

- expressions where *a name is defined* by that binding,
- expressions where that binding is *visible*.

An *environment*

is a *collection of bindings*

Created by

program blocks:

- `begin ... end, {...}, (let (...)  ...)` etc.,
- explicit bindings within the blocks:

  `int x;  (define x 10);  (let ( (x 20) ) ...)` etc.

procedure definitions

- explicit bindings within the body (block), *and*
- procedure parameters

6.2 Environments

✷ **Program blocks, and procedures, can be *nested*:**

⇒ Bindings can *change* between blocks

⇒ Environments need to reflect this feature

Conceptually: use a *stack* of environments.

⇒ the environment containing the ‘current’ binding for a name is the *topmost* in which the binding appears.

Scoping Mechanisms

There are two ways to determine the *scope* of a binding:

a) *Static Scoping*

Static ⇒ can be done *before* runtime

b) *Dynamic Scoping*

Dynamic ⇒ *must* be done *at runtime*

- Distinguished by how the environment stack is formed ...

6.3 Static Scoping

The binding is given by *textually closest* definition to the use of the name.

⇒ environment is pushed when a block / procedure definition occurs in the *program text*

⇒ can be done at *translation time* (compile-time)

Examples (blocks)

Java: `{ int x;` ← scope of (binding for) `x` → `}`

Java: `{ int x;` ← scope of `x ...`
`{ float y;` ← scope of `x` and `y` → `}`
...scope of `x` continued → `}`

Nesting

C: `{ int x;` ← scope of `x ...`
`{ float x;` ← scope of *new* `x` → `}`
...scope of *original* `x` continued → `}`

re-binding

⇒ An *inner* scope *hides* enclosing bindings for *new* definitions.

Examples (procedures)

Java: `void foo( float z ) { int x;` ← scope of `x` *and* `z` → `}`

scheme:

```
(define (f p))
  ← scope of p ...
    (let ( (q 10) )
      ← scope of p and q ...
      (let ( (x p) (q 99) )
        ← scope of x, new q, and p →
      )
      ...scope of p and original q continued →
    )
  ...scope of p continued
)
```

scheme: `(lambda (x) (let ((y 10)) )` ← scope of `x` *and* `y` → `)`

6.4 Dynamic Scoping

The binding is given by definition *most recent in time* to the use of the name.

⇒ environment is pushed when a block / procedure is *entered at runtime*

⇒ must be done at *runtime*

6.5 Static v. Dynamic Scope

Example

```
(define (outer)
  (let ( (X 10) )
     (define (inner1)
       (let ( (X 20) )
          ; body of inner1
          (inner2)
       )
     )
     (define (inner2)
       ; body of inner2
       X
     )

     ; body of outer
     (inner1)
  )
)
```

Call Sequence

- outer calls inner1
  - inner1 calls inner2
    - inner2 returns X

Static Scoping

⇒ outer's call to inner1 returns 10

⇒ inner2's X is outer's X

Dynamic Scoping

⇒ outer's call to inner1 returns 20

⇒ inner2's X is inner1's X

Example

perl:
```
$x = "f's value of x";

sub f { return "f is returning " . $x; }

sub g_static  {  my   $x = "g's value of x";
                  return f(); }

sub g_dynamic {  local $x = "g's value of x";
                  return f(); }
```

`g_static` returns: `"f is returning f's value of x"`

`g_dynamic` returns: `"f is returning g's value of x"`

6.6 Closures

A *Closure*

is a procedure definition *plus* the environment active when it was created.

Example

```
(define (mul x) (lambda (y) (* x y)))
```

```
(define double (mul 2))
```

→ a procedure which multiplies its argument by 2

⇒ `x` in `mul` is bound to `2`

```
(define triple (mul 3))
```

→ a procedure which multiplies its argument by 3

⇒ `x` in `mul` is bound to `3`

Control Flow

Control Flow

Turing-complete language
⇒ *conditional* control of evaluation sequence

Useful Turing-complete language
⇒ *repetition* constructs.

Both of these are ways of *modifying* the *flow of control*

7 Choice and Repetition

7.1 Choice

***Choice* mechanisms**

allow *redirecting* of the flow of control:

⇒ *Do* this, or do that, [. . . or do that, or that, or . . .]

⇒ *Evaluate* this, or that, [. . . or that, or . . .]

- Decision is based on the *current state*.
  - *state* ⇒ set of values in the running process
    . . . which may be denotable or expressible

⇒ Need linguistic means for checking the state

⇒ *predicates* (expressions returning truth values) on the state

***Predicates* can be:**

- *primitive*, or
- *compound*

Primitive Predicate examples

Equality	`=` `eq?` `==` ...
Inequality	`!=` `<` ...
Type membership	`instanceof` `symbol?` ...
Data structure membership	$\in$ `in` `subset` ...

Composition operators

Unary	`!` $\neg$
Binary	`&` `|` `&&` `||` `.and.` `.or.` ...

Multi-Way Choice

can always be simulated by *nested* two-way choices

. . . but special syntax aids readability and writability:

Statement-oriented:

Java / C:
```
switch (expression) {
    case value: statement;
    case value: statement;
        ⋮
    default: statement;
}
```

Expression-oriented:

scheme:
```
(cond ( predicate expression )
      ( predicate expression )
          ⋮
      ( else expression )
)
```

Other choice mechanisms

are given in some languages, but all reduce to a two-way conditional:

Examples

Exceptions:

Java: `try{...A...} catch (Exception) {...B...}`

≈ `if Exception-occurs-in { ...A... } then {...B...}`

Pattern matching:

Haskell:

```
data Expr = Val Int | Sum Expr Expr | ...

eval (Val n) = n
eval (Sum e1 e2) = (eval e1) + (eval e2)
```

⇒ similar to a `switch / cond`

Evaluation of conditionals

1. evaluate *predicate*, then
2. evaluate *selected branch*

⇒ scheme: `(if pred expr1 expr2)` cannot be evaluated like other expressions ...

Normal rule: evaluate all the components and then apply the first to the rest.

Consider:

```
(define (doit n)    (if (= n 1) 0 extremely-long-computation) )
(doit 1)
```

or worse:

```
(define (forever) (forever) ) (if #t 1 (forever) )
```

7.2 Repetition

Repetition mechanisms

allow sequences of expressions to be evaluated repeatedly, until some *condition* holds.

⇒ must involve some *variation* in each repetition otherwise . . .

- pointless,
- will either never repeat, or will never stop.

Condition is a predicate on computational state

⇒ use the same expressions as choice mechanisms

Terminology

An *Iterative Process*

exists when the memory needed to control the repetition *does not increase* with each cycle.

An *Iterative Definition*

generates an iterative process when evaluated.

A *Recursive Process*

exists when the control memory required *increases* with each cycle.

A *Recursive Definition*

may generate a recursive *or an iterative* process when evaluated.

Depends on:

- form of the definition, and
- language implementation

Iteration Constructs

Bounded: `for`

Unbounded: `while` `do...while` `repeat...until` etc.

Combined: `for` (C, Java)

Issues

control variable scope:

- is the loop counter *local* to the loop?
- can it be accessed *after* the loop has exited?

exceptional exit: `break` (C, Java)

exceptional entry: is jumping *into* the body of a loop allowed?

7.3 Recursion

A *Recursive* (inductive) *Definition*

is when *self-reference* occurs within the definition.

This is *well-defined* iff:

- one or more *base-cases* exist,
- ... which will always be reached.

A *base-case*

is a non self-referential expression which provides the returned value.

Examples

C:
```
int fact(int n) {
        return n==0 ?  1 :  n * fact(n-1);
    }
```

scheme:
```
(define (!  n)
  (if (= n 0) 1
              (* n (! (- n 1)))
  )
)
```

7.4 Tail Recursion

occurs if the result of the *recursive call* is the *result of the procedure*

... with nothing else 'in between'

Executing a tail call ⇒ all control memory can be *re-used*

1. overwrite local variables and parameters, and
2. *jump* to the start of the procedure's code.

a tail call ≡ *a* `goto` (with parameters|)

Example:

C: `int factR( int n ) { return n==0 ? 1 : n * factR( n-1 ); }` *is not* tail recursive, but:

C: `int factI( int n, int r ) { return n==0 ? r : factI( n-1, r*n ); }` *is* tail recursive.

. . . *But does it give an iterative process?*

⇒ yes — with a good compiler . . .

`gcc factI.cc`

```
factI:
       pushl   %ebp
       movl    %esp, %ebp
       movl    8(%ebp), %edx
       movl    12(%ebp), %ecx
       movl    %ecx, %eax
       testl   %edx, %edx
       je      .L1
       subl    $8, %esp
       movl    %edx, %eax
       imull   %ecx, %eax
       pushl   %eax
       decl    %edx
       pushl   %edx
       call    factI
.L1:   movl    %ebp, %esp
       popl    %ebp
       ret
```

`gcc -O2 factI.cc`

```
factI:
       pushl   %ebp
       movl    %esp, %ebp
       movl    8(%ebp), %edx
       movl    12(%ebp), %eax

.L2:   testl   %edx, %edx
       je      .L1


       imull   %edx, %eax

       decl    %edx

       jmp     .L2
.L1:
       popl    %ebp
       ret
```

is not only a space-saver . . .

Example:

scheme:
```
(define (fib n)            ; doubly recursive version
  (cond ( (= n 0) 0 )
        ( (= n 1) 1 )
        (else (+ (fib (- n 1)) (fib (- n 2))))


(define (fibI a b n)       ; iterative version
  (if (= n 0)
      b
      (fibI (+ a b) a (- n 1))))
```

Performance:

- *recursive* Fibonacci — time $\propto n^2$
- *iterative* Fibonacci — time $\propto n$

Data Types

Data Types

A programming language's *Data Types* are *characterised* by the:

1. *Primitive* types
2. Type *composition* mechanisms
3. Type *abstraction* mechanisms

⇒ The same principle as *expressions*

8 Data Types

8.1 Theory 1

A *Type* is

- a *set* of *values*, together with
  - a collection of *operations* on those values

Types:

- *name* sets of values
  - *hide representations* of values
  - *name* operations on elements of the type
  - *hide representations* of the operations

Types are *abstractions* of values

Primitive Types

Boolean (truth values)

- `bool, Boolean, boolean`

Numerical

- `int, Integer, float, real, double, long, complex ...`

Characters

- `char`

Strings

- `string, String,` ... often not primitive.

Enumerations

- `enum {second, initial, tuesday, X}`

Subranges

- `Real 1.0 .. 2.0`

... of *ordinal* types

8.2 Theory 2

Composition Mechanisms

Types are *sets* $\Rightarrow$ composition mechanisms are *set operations*:

Three fundamental $\left\{ \begin{array}{l} \text{composition operators} \\ \text{type constructors} \end{array} \right\}$

		Type terminology	*Set terminology*	
1.	Product:	A×B	Cartesian Product:	$A \times B$

- A×B is the *type* containing *pairs of values* of type A *and* B
- operation is *projection* $\Rightarrow$ *extract* the A value or B value

2.	Sum:	A + B	Disjoint Union:	$A \uplus B$

- A + B is the *type* containing values of type A *or* B
- operation is *injection* $\Rightarrow$ *test* the type of the value

3.	Function:	A → B	Exponentiation:	B^A

- A → B is the *type* containing functions from a value of type A to one of type B
- operation is *application* $\Rightarrow$ *return* the B value identified by the A value.

8.3 Type Composition

Product Types

Examples:

C: `struct { int x; int y; }`

ADA:
```
record
  x:  Integer; y:  Integer;
end record
```

Java: `class { public int x; public int y; }`

projections: `.`*name* `...` `thing.`*x*`,` `thing.`*y*

scheme: `(cons x y)` . . . a *pair*

projections: `(car` *pair* `)`, `(cdr` *pair* `)`

Haskell: `( x, y )` . . . a *tuple*

projections: `fst` *tuple*, `snd` *tuple* . . . or ‘pattern matching’

⚠ Products of more than 2 types are still products since:

$$A\times B\times C \quad\simeq\quad A\times(\ B\times C\ ) \quad\simeq\quad (A\times B)\times C$$

Sum Types

are (almost) *unions* / *variants* etc.

⚠ Must carry the *injection* operations ⇒ *type predicates* (testers).

Examples:

Haskell: `data t = A a | B x y | C c`

injections: pattern matching on tags / discriminants `A, B, C ...`

ADA: *variant records* . . . see POPL Part 2!

Java: simulated with *sub-classing*:

```
class T{};
   class A extends T{};  class B extends T{}; ...
```

injections: injection operation is `instanceof`

C: `union { int x; double y; }`

injections: *None!* ⇒ type system can't check!

scheme: no (static) types ⇒ no sum types (unions)!

⚠ Sums of more than 2 types are still sums since:

$$\boxed{A + B + C \quad \simeq \quad A + (B + C) \quad \simeq \quad (A + B) + C}$$

Function Types

are *procedures* / *functions* / *methods* / *subprograms* etc.

Not *expressible* in many languages such as:

- Java, ADA, C, scheme (no types!) ...

e.g. C: `int f (int x, int y) { ... }`

... creates an *instance* of int × int → int,

but it's not possible to say:

$$\texttt{(int} \times \texttt{int} \rightarrow \texttt{int) f;}$$

Examples:

Haskell: `(Int, Int) -> Int`

ML: `(Int * Int) -> Int`

8.4 Theory 3

Currying

A function of *two* arguments has type:

$$(A \times B) \rightarrow C$$

Isomorphic to — essentially the same as – a function of *one* argument that *returns a function of one argument*:

$$A \rightarrow (B \rightarrow C)$$

This is called a *curried* form of the 2-argument version

Examples

scheme:

```
(define (f a b) (+ a b))
     ; number × number → number
(define (f a) (lambda (b) (+ a b)))
     ; number → number → number
```

Haskell:

```
f (x,y) = x+y    :: (Int ,   Int) -> Int
f  x y  = x+y    ::  Int -> (Int  -> Int)
```

8.5 Type Abstraction

The fundamental abstraction mechanism is *naming*.

⚠ Not possible in all languages

Examples

Java:
```
class C { ... };
```

C:
```
typedef int Length; typedef char* String;
typedef struct { int left; int right; } Sides;
```

Haskell:
```
type Sides = (Int,Int)
type BinaryOp = Int -> Int -> Int
```

ADA:
```
type Complex is record
                  real_part :  Real;
                  imaginary_part :  Real;
                end record;
```

8.6 Abstraction Mechanisms

Factor out the variable parts from the fixed parts . . .

Parametrisation (Generics)

Gives rise to *type functions*

⇒ take *types* and return a *type*

Examples:

Java:

```
class Stack<T> { T stack[...]; ...
                 T pop(){...};
                 void push(T x) {...};
               }
```

Haskell:

```
type BinaryOp a = a -> a -> a
data Tree a = EmptyTree
            |(Tree a) (Node a) (Tree a)
```

8.7 Type Checking

is ensuring that *argument types* are *compatible* with *parameter types*

⇒ applies to all operations: primitives, user-defined procedures etc.

Terminology

Static typing:

all types are known, and can be checked, *before* run-time

Dynamic typing:

(some) types aren't known *until* run-time

⇒ can get *runtime type errors*

Strongly typed:

all type errors are detectable

... whether statically or dynamically

... But what does *compatible* mean?

Type Compatibility

Two types are *compatible* iff:

- they are the *same type*, or
- a value of one type can be translated into a unique value of the other
  $\Rightarrow$ *coercion*
  e.g. `645` (integer) can-be-coerced-to `645.0` (float)

Coercion can be:

- automatic
  $\Rightarrow$ language's coercion rules are applied at compile-time
  e.g. sub-types are coerced to super-types when necessary

- user specified
  - normally called type *casting*

Type Casting

breaks type system safety

Example

Java:
```
long l = 656666L;      int i = (int) l;
```
⇒ type OK, statically and dynamically

```
long l = 6566660000L; int i = (int) l;
```
⇒ type OK, statically and dynamically
... but *semantically wrong!*

```
Integer caster(Object o) { return (Integer) o;}
```

```
caster( new Integer(42) );
```
⇒ type OK, statically and dynamically

```
caster( new String() );
```
⇒ type OK statically, *but runtime error*

8.8 Type Equivalence

rules in a language say when two types are *the same*

Two approaches:

1. Name Equivalence

- Two entities have the *same type* if their types have the *same name*

⇒ Cheap to check

⇒ *Too strict* in some circumstances

Example

C:
```
typedef struct {int x; int y;} Pair;
Pair p1, p2;
struct {int x; int y;} p3;
p1 = p2;        //OK
p3 = p2;        //type error
```

2. Structure Equivalence

- Two types are the same if they have the *same structure*

⇒ More difficult to check (especially dynamically)

⇒ *Too lax* in some circumstances

Example

notJava:
```
class Place {float x; float y;};
      x,y coordinates (lat, long)
class Rectangle {float x; float y;};
      width and height

Place earth; Rectangle field;
bool isSquare(Rectangle r) {
                return r.x == r.y;
}
isSquare(field);         //(structurally) type correct
isSquare(earth);         //(structurally) type correct
```

8.9 Type Inference

occurs in modern languages where types can be *deduced*

Types of primitive operators are known

⇒ type of an expression can (often) be inferred

⇒ not necessary to *declare* the type of every entity

Example

Haskell:
```
map f [] = []
map f (x:xs) = (f x) :  (map f xs)
```

1. `map` is a function of *two* arguments. In Haskell this is curried:

$$\texttt{map} :: \alpha \to \delta \to \epsilon$$

2. δ and ϵ are 'lists of something' — either `[]`, or a 'cons' (`:`)

$$\texttt{map} :: \alpha \to \text{List of } \beta \to \text{List of } \gamma$$

3. `f` is a function of one argument, returning an element of `map`'s result:

$$\alpha = \beta \to \gamma$$

4. Hence `map` has the type:

$$(\beta \to \gamma) \to \text{List of } \beta \to \text{list of } \gamma$$

$\Rightarrow$ `map ::  (a -> b) -> [a] -> [b]` . . . the *most general type*

Deduced by the compiler

Encapsulation

Encapsulation

Abstraction is *the* complexity control mechanism

- collect related ‘things’
- name the collection
- extract (abstract) their differences
- ignore their commonalities

An *Encapsulation is* an abstraction
. . . but normally used to refer to a *collection of abstractions*
i.e an *abstraction of abstractions*

8.10 Abstract Data Types

are *compound* Data Types . . . i.e. not primitive

- a set of *values*

  ⇒ a *type*

- a collection of *operations* on those values

  ⇒ function / procedure definitions

  ⇒ implementation *hidden*, andsignature (interface) *exposed*

Examples

Java:	*classes* can be used as ADTs . . . but are more
ADA:	*packages* can be used as ADTs . . . but are more
scheme:	*procedures and environments* can simulate ADTs

8.11 Modules

A Module or package

- is a *collection* of ADTs, plus
- a mechanism for controlling the *visibility of names*

⇒ an *Encapsulation*

Examples

Java:	`package stuff` ... *class definitions* ...	
or:		
	`interface I {` *specification* `}` `class C implements I {` *implementation* `}`	*public* names are visible
ADA:	`package P is` *specification* `end P;` `package body P is` *implementation* `end P;`	
Haskell:	`Module M (` *exported names* `) where` ... *type and function definitions* ...	*exported names* are visible

8.12 Abstraction

Why Abstract?

1. Abstraction controls complexity

   ... but abstractions become complex

   ⇒ *Abstract the abstractions*

   ⇒ raise the abstraction level

2. Separate compilation

   - Dependencies between program components (implied or explicit) determine what needs to be recompiled when some part of a system changes.
   - Separating interface and implementation:
     - changing implementation *doesn't* require re-compiling all uses of the abstraction.
       ... will require re-linking though
     - changing specification *does* require re-compilation.

Why not?

...

9 Health Warning

9.1 Health Warning

The Pathology of Abstraction

Abstraction ⇒ ignoring (some details)

⇒ simplification

⇒ generalisation

. . . but can go too far!

“Simplify as far as possible . . . but no further!”

Einstein

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] David A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. William C. Brown Publishers, 1986.

[3] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Publishing Company, 9th edition, 2009.