IMPERATIVE PROGRAMMING LANGUAGES

(or: why the actual choice of programming language doesn't really matter)

by

David Toman

University of Waterloo

based on David A. Schmidt:

The Structure of Typed Programming Languages

MIT Press, 1994

List of Slides

Outline

- A core language
  - $\Rightarrow$ basic syntax
  - $\Rightarrow$ how to understand what does it do
- Extensions
  - $\Rightarrow$ giving things names
  - $\Rightarrow$ giving things parameters
  - $\Rightarrow$ allowing local definitions
  - $\Rightarrow$ “real” variables
- Syntax Summary

The Core Language

$$
\begin{array}{lcl}
cmd & ::= & \textbf{skip} \\
 & | & loc(\tau) := exp(\tau) \\
 & | & cmd \; ; \; cmd \\
 & | & \textbf{if } exp(bool) \textbf{ then } cmd \textbf{ else } cmd \textbf{ fi} \\
 & | & \textbf{while } exp(bool) \textbf{ do } cmd \textbf{ od} \\
 & & \\
exp(int) & ::= & const(int) \\
 & | & loc(int) \\
 & | & exp(int) + exp(int) \\
 & & \\
exp(bool) & ::= & exp(int) = exp(int) \\
 & | & loc(bool) \\
 & | & \neg exp(bool)
\end{array}
$$

Abstract Syntax Tree

$$loc_1 := 0; \textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od}$$

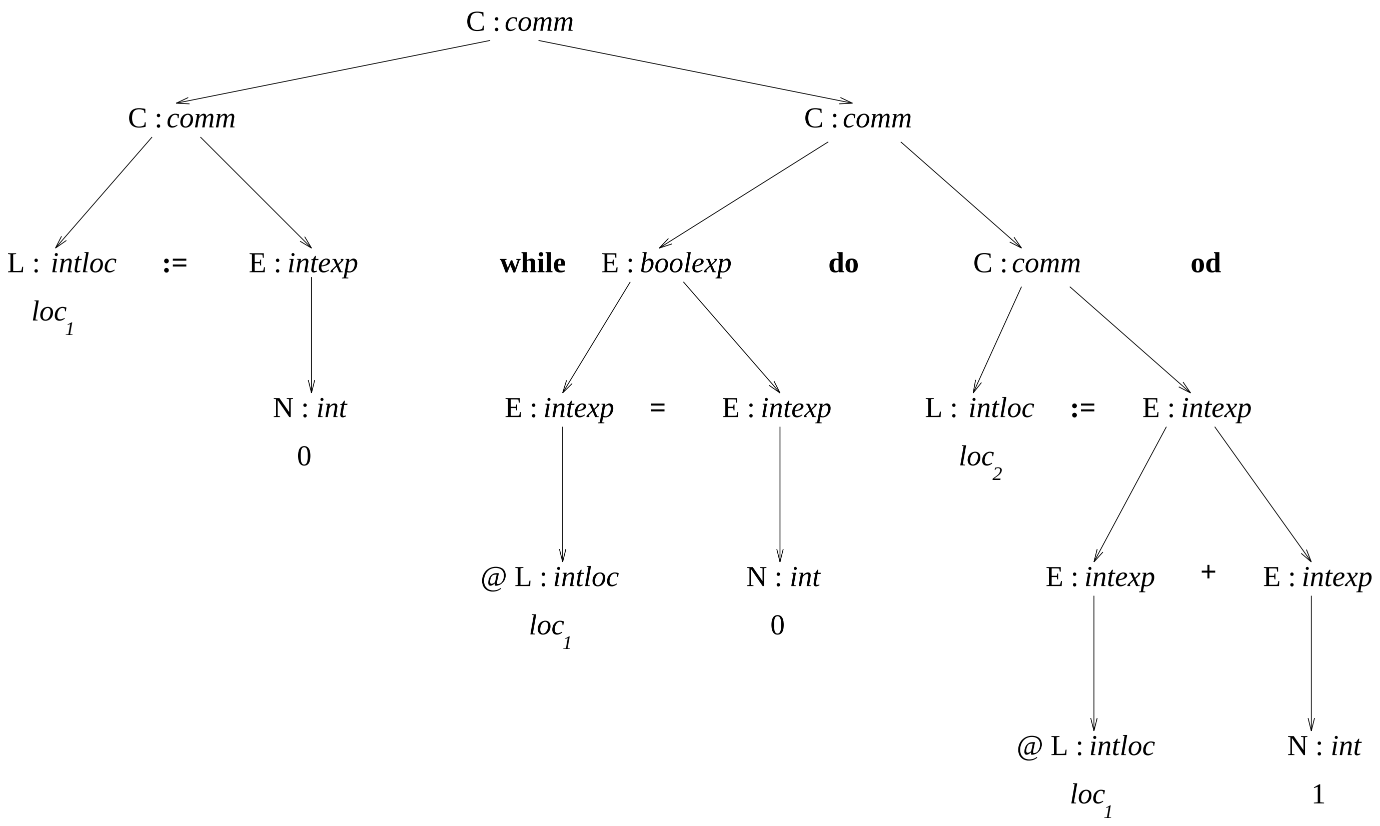

Figure 1.3 (corrected)

What does it do?

IDEA: programs (commands) use *computer memory* (store) to keep intermediate results and state of the computation

$\Rightarrow$ store (memory) is an **array** of **locations**

loc_0	loc_1	loc_2	$\cdots$	loc_{256M-1}

$\Rightarrow$ executing a command **changes** values in this array

— given a location we can retrieve a value

— given a location and a value we can store it

What does it do? (cont.)

Now we can *precisely* say what does it mean to execute a command (defined by cmd) as a *function* that transforms the store S

- commands are functions from store to store
- expressions are functions from store to a value

What does it do? (commands)

skip (a no operation)

$$F(\textbf{skip}, S) = S$$

$loc(\tau) := exp(\tau)$ (an assignment)

$$F(loc := exp, S) = \text{assign}(loc, F(exp, S), S)$$

$cmd \; ; \; cmd$ (sequential composition)

$$F(cmd_1 \; ; \; cmd_2, S) = F(cmd_2, F(cmd_1, S))$$

if $exp(bool)$ **then** cmd **else** cmd **fi** (if-conditional)

$$F(\textbf{if } exp \textbf{ then } cmd_1 \textbf{ else } cmd_2 \textbf{ fi}, S) =$$
$$F(cmd_1, S) \text{ if } F(exp, S) = \text{true}$$
$$F(cmd_2, S) \text{ if } F(exp, S) = \text{false}$$

while $exp(bool)$ **do** cmd **od** (while-loop)

$$F(\textbf{while } exp \textbf{ do } cmd \textbf{ od}, S) =$$
$$S \text{ if } F(exp, S) = \text{false}$$
$$F(\textbf{while } exp \textbf{ do } cmd \textbf{ od}, (F(cmd, S)) \text{ otherwise}$$

What does it do? (expressions)

$loc(int)$ (value lookup)

$$F(loc, S) = \text{lookup}(loc, S)$$

$constant$ (a integer constant c)

$$F(constant, S) = c$$

$exp + exp$ (integer addition)

$$F(exp_1 + exp_2, S) = \text{plus}(F(exp_1, S), F(exp_2, S))$$

...

Example

$F(loc_1 := 0; \textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od}, [7, 5, \ldots]) =$

$F(\textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od},$

$\qquad F(loc_1 := 0, [7, 5, \ldots])) =$

$F(\textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od}, [7, 0, \ldots]) =$

$F(\textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od},$

$\qquad F(loc_1 := loc_1 + 1, [7, 0, \ldots]) =$

$F(\textbf{while } loc_1 = 0 \textbf{ do } loc_1 := loc_1 + 1 \textbf{ od}, [7, 1, \ldots] =$

$[7, 1, \ldots]$

How to make it into a full-blown PL?

How do we get *named constants*, *functions*, *procedures*, *modules* (all of these *parametrized*), and *local values/hiding mechanisms*?

$\Rightarrow$ by applying *three simple* principles:

1. every syntactic construction can be given a name
2. every syntactic construction can be given a parameter
3. every syntactic construction can be given local definitions

Naming Things

Generic form for a *declaration*:

$$D ::= \textbf{define } id = V \mid D; D$$

Depending on what V is we get:

- a procedures for V a command
- a function and constant for V an expression
- a module for V a declaration
- a class/record for V a type structure (coming)

Use: "$D \ \mathbf{in} \ V$" where V can "use" identifiers defined in D

Parameters

Generic form:

$$\lambda p.V$$

... takes a value for p, substitutes it into V and then does whatever V used to do.

Depending on what V is we get parametrized versions of the above constructs, in particular

$\Rightarrow$ parametrized procedures
$\Rightarrow$ parametrized functions

In most languages parameters can be given only to *named* things; the usual form then is

$$\textbf{define } id(p_1, \ldots, p_k) = V$$

where V contains identifiers $p_1, \ldots, p_k$.

Local Definitions (hiding)

Generic form:

$$\textbf{begin } D \textbf{ in } V \textbf{ end}$$

... declarations D are "visible" only in V

- command blocks (most common; V is a command)
- local (private) variables in classes/modules/ADTs

Storage Management

ONE MAJOR missing ingredient:

$\Rightarrow$ variable declarations!!!

Problem: a variable declaration

```
var i: integer
```

does two things:

1. allocates space (an unused location) for an `integer`
2. names the obtained location `i`

$\Rightarrow$ type structures (essentially records)
$\Rightarrow$ **space allocation** routines (essentially commands)

Pointer Types

IDEA:

(1) locations are indexed (essentially) by *integers*
(2) integers can be stored in locations

$\Rightarrow$ pointer types

- type of a pointer variable tells us what is the data type of the location of whose index is stored in this location
- anonymous locations (without explicit names)

  $\Rightarrow$ dynamic storage allocation (**`new`**)
  $\Rightarrow$ **`new`**'s parameter is a *type-structure*
  $\Rightarrow$ **`new`** returns a location of the appropriate type

The Result

Essentially PASCAL/ALGOL 60/JAVA w/o inheritance-like language; the only thing remaining is to change "generic" constructions into specific keywords:

feature	name in PASCAL	name in JAVA
named command	procedure	(void) method (class)
named expression	function	method (class)
module	n/a	static class
type structure	record	class
. . .		

Syntax Summary

Types:

```
<type> ::=  integer
         |  char
         |  array [m..n] of <type>
         |  record id1:<type>,
                  ..., idk:<type> end
         |  ^<type>
```

Expressions:

```
<exp>  ::= <const>
        |  <exp>+<exp>
        |  ...
        |  ID(<exp>, ..., <exp>)
```

Syntax Summary (cont.)

Commands:

```
<cmd> ::= <loc> := <exp>
        |  begin <cmd>; ... ; <cmd> end
        |  if <exp> then <cmd>
                     else <cmd>
        |  while <exp> do <cmd>
        |  for <id>:=<exp> to <exp>
                               do <cmd>
        |  new(p)
        |  ID(<exp>, ..., <exp>)
```

`repeat-until`-loop is special case of **`while`**-loop; **`case`**-statement is special case of **`if`** statements (nested).

Syntax Summary (cont.)

Declarations:

```
type <id> = <type>;

var  <id>:  <type>;

function <id>(<id1>:<type1>,...,
              <idk>:<typek>): <type>;
begin
   <cmd>;
   <id> := <exp>
end;

procedure <id>(<id1>:<type1>,...,
               <idk>:<typek>);
begin
   <cmd>
end;
```