Updating Statements

- Update statements are assignment statements used to increment some value
  $l := l + c$
- Many languages provide a special notation for update statements

INC(l, c)	Modula-3
l += c	C
ADD c TO l	COBOL
l +:= c	ALGOL 68

1

Updating Statements

- The *l-value* of l in l +:= c can be evaluated only once
- The assignment $l := l + c$ is equal to l +:= c only in the absence of *side effects*
- Compare
  A[f(x)] := A[f(x)] + 2
  with
  A[f(x)] +:= 2

2

Multiple Target Assignments

- Some languages permit more than one location to be given a value at the same time

  $l_1, l_2, \ldots, l_n := e$ ALGOL 60, PL/I

3

Multiple Assignment Statements

- Using this form of the assignment, the operation of swapping the values of two variables can be written $x, y := y, x$
- The semantics is *ambiguous* when the expressions on the left-hand side denote the *same l-value*
- This form of assignment statement does not appear in programming languages that have actually been implemented

4

L-Valued Expressions

- Languages often have expressions other than identifiers for *l-values*, e.g.,
  a[i + 1], and r.f
  denote both an *l-value* and an *r-value*
- Some languages permit *conditional l-valued expressions*
  (if i = 3 then a[j] else x) := 2

5

L-Valued Expressions

- In C++ and ML, *user-defined functions* can be in *l-valued* expressions

```
int  a[10];
int&  f( int i ) { return (a[i]); }
f(5) = 17;                    // a[5] := 17

val x = ref 1;
val y = ref 2;
fun f (n) = if (n mod 2) = 0 then x else y;
f(3) := 4;                    // y := 4
```

6

Aliases

- Two *l-valued* expressions may denote the *same* location, in which case they are called *aliases* for that location

EQUIVALENCE	FORTRAN
union	C

- Aliasing often arises *unexpectedly* when using *pointers* and in *argument passing*

7

Control

- A great part of imperative programming is *arranging the order* in which assignment statements are executed to cause the desired values to be left at the appropriate *addresses*
- A single construct, the *test-and-branch*, is capable of controlling the flow of execution to all conceivable ends

8

Goto Statements

- The *goto* construct is a special case, an *unconditional branch*
- Programming with goto statements is *unstructured*, because they reveal little about the programmer's intent

9

For Constructs

- The for construct in Ada

```
for i in lb .. ub loop
        <sequence of statements>
end loop;
```

- The for construct in Modula-3

```
FOR i := FIRST(A) TO LAST(A) DO
        <sequence of statements>
END;
```

10

If Constructs

- The if construct in Ada

```
if <condition> then
        <sequence of statements>
elsif <condition> then
        <sequence of statements>
else
        <sequence of statements>
end if;
```

11

While and Loop Constructs

- The while construct in Ada

```
while <condition> loop
        <sequence of statements>
end loop;
```

- The loop construct in Ada

```
loop
        <sequence of statements>
        exit when <condition>
        <sequence of statements>
end loop;
```

12

Guarded Commands

- A *guarded command* is a Boolean expression followed by a statement
- The statement is executed only if the *guard* is *true*
- The *order* of the guarded commands is irrelevant
- If more than one guard is true, then one is *nondeterministically* selected

13

Guarded Commands

- The guarded commands like the if construct

  if $x \geq y \rightarrow m := x$
  $\square\ y \geq x \rightarrow m := y$
  fi.

- The guarded commands like the while construct

  do $x > y \rightarrow x := x - y$
  $\square\ y > x \rightarrow y := y - x$
  od.

14

Binding

- The *association* of a *name* to an *attribute* is called a *binding*
- Names are bound to *types*, *locations*, *values*, and other attributes at various points
  - *compile time*
  - *link time*
  - *load time*
  - *run time*

15

Scope

- The *scope* of a name is the *section of the program text* in which the name has the *attributes* established by the declaration
- The scope of names are usually determined by the *structure* of the program

16

Extent

- The *extent* of a location in memory is *that period of program execution* in which the location *can be accessed* by some name or expression in the program
- The extent of a location is determined by its *storage class*

17

Storage Classes

- There are three major categories:
- *Static* (C), *own* (ALGOL 68)
- *Local*, *automatic* (C, PL/I), *loc* (ALGOL 68)
- *Dynamic* (C), *based* (PL/I), *heap* (ALGOL 68)

18

Static Storage

- FORTRAN originally used *static storage* for all variables
- In static storage all identifiers are bound to locations whose extent is the *entire life of the program*
- If a static variable's scope is local to a procedure, it can be used to *retain the state* from call to call

19

Automatic Storage

- The *extent* of the locations for the identifiers declared *locally* to a procedure is only for the *lifetime of the procedure activation*
- Automatic storage is typically *allocated whenever the procedure is invoked*
- Automatic storage is typically *reclaimed after the procedure exits*

20

Dynamic Storage

- *Dynamic storage* is allocated and deallocated *as needed* during the execution of the program
- Dynamic storage may be *explicitly* allocated and deallocated by the *programmer* like C
- Dynamic storage may be *implicitly* allocated and deallocated by the *runtime system* like LISP

21

Dynamic Storage

- Dynamic storage is usually in a separate and formless area called the *heap*
- When the extent of a location allocated on the heap is up, and the programmer didn't deallocate the storage, *inaccessible storage* or *garbage* is generated
- In some languages, e.g., Modula-3, the runtime system performs *garbage collocation*

22

Mutable and Immutable Objects

- An *immutable* object is one that no operation can change
- *Integers* are examples of immutable objects
- A *mutable* object has a state which may be modified by certain operations without changing the identify of the object
- *Arrays* and *records* are examples of mutable objects

23

Binding to Immutable Objects

- Before the binding, *A* is bound to the value 2 and *B* is bound to 3
- The let binding let A = B causes a new binding
- This new binding eclipses the previous binding to *A*, and the current bindings of *A* and *B* are both to the value 3
- This sort of binding is appropriate for *immutable* objects

24

Value Semantics

- The let binding can be modeled by a simple semantics, called *value semantics*
- An *environment* is a mapping from *names* to *values*:
  $\rho : Ident \rightarrow Nat$
- Before the binding $\rho_0(A) = 2$ and $\rho_0(B) = 3$
- After the binding $\rho_1 = \rho_0[A \rightarrowtail \rho_0(B)]$; hence, $\rho_1(A) = 3$ and $\rho_1(B) = 3$

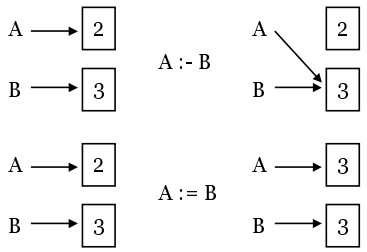
25

Binding to Mutable Objects

- If the program has bindings to mutable objects and changing one object effects the others, then these objects exhibit *pointer semantics - copying pointers* (SIMULA, CLU)
- The usual semantics in which the object denoted by the target of the assignment is treated as a box is called *storage semantics - copying values* (C, Pascal)

26

Binding to Mutable Objects



27

Pointer and Storage Semantics

- An *environment* is a mapping from *names* to *locations*:
  $\rho : Ident \rightarrow Loc$
- A *state* is a mapping from *locations* to *values*:
  $\sigma : Loc \rightarrow Nat$
- Two identifiers *A* and *B* are *aliases* if $\rho(A) = \rho(B)$

28

Pointer and Storage Semantics

- A :- B changes the *environment*, *not* the *state*
- Before the binding we have ρ_0 and σ_0
- After the binding we have $\rho_1 = \rho_0[A \rightarrowtail \rho_0(B)]$, and $\sigma_1 = \sigma_0$
- A := B changes the *state*, *not* the *environment*
- Before the binding we have ρ_0 and σ_0
- After the binding we have $\rho_1 = \rho_0$ and $\sigma_1 = \sigma_0$ $[\rho_0(A) \rightarrowtail \sigma_0(\rho_0(B))]$

29

Pointer and Storage Semantics

- For *immutable* objects, pointer semantics is the *same* as storage semantics
- Both pointer semantics and storage semantics may be used in the same language

```
void f( ) {
        int  i = 1,  j = 2;
        int &r = i;
        r = j;
}
```

30

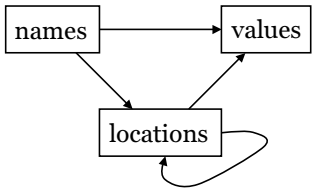
Pointer and Storage Semantics

	int i, j	i = 1; j = 2	int &r = i	r = j
ρ(i)	α_1	α_1		
ρ(j)	α_2	α_2		
ρ(r)			α_1	α_1
$\sigma(\alpha_1)$		1	1	2
$\sigma(\alpha_2)$		2	2	2

r = i changes ρ(r) and r = j changes $\sigma(\rho(r))$

31

Storage Model



32

Pointers

- When a storage location is not just denotable (by l-expressions), but can be a value (r-value) itself, we have a *pointer*
- Pointers are declared statically, but the storage locations may be created and disposed *dynamically* under the control of the program
- If a pointer doesn't point to any valid location, it has a special value, called *nil*

33

Recursive Data Types

- Pointers are most often used to create *recursive data structures* like lists, trees, and graphs
- Another approach to creating recursive data structures is to provide *recursive data types*

34

Recursive Data Structures

- Use *pointers* in C:

```
struct tree {
   int  n;
   struct tree *left;
   struct tree *right;
};

t = (struct tree *) malloc( sizeof(struct tree))
t->n = 3;
t->left = t->right = (struct tree *) 0;
```

35

Recursive Data Structures

- Use *recursive data types* in ML:

```
datatype tree = empty
              | node of int * tree * tree

empty
node(3, empty, empty)
```

36

Dangling Reference

- A *dangling reference* is a pointer to a location that has been disposed or has been used for another purpose

```
var p, q: ^integer;
begin
   new(p);
   q := p;
   dispose(p);
end;
```

```
var p: ^integer;
procedure q;
   var i: integer;
begin
   p := ADDR(i)
end;
```

37

Collections

- A *collection* is an *aggregate of pointers*
- Each collection is initially *empty*
- The *new* statement *adds* a pointer to a collection
- The *free* statement *deletes* a pointer from a collection
- Each pointer is *restricted* to locate items in *a* particular collection
- This partitions *heap* into *collections*

38

An Example

```
var list: collection of
   record
      next: pointer to list
      name: string(30)
   end record
var first, last: pointer to list := nil(list)
```

39

An Example

```
procedure append(p: pointer to list)
   if first = nil(list) then
      first := p
   else
      list(last).next := p
   end if
   last := p
   list(p).next := nil(list)
end append
```

40

An Example

```
var item: pointer to list
new list, item
list(item).name := "A. M. Turing"
append(item)
free list, item
list(item).name := "J. Von Neuman"
            % Use of a dangling pointer
```

41

Identifiers

- The *simplist* expressions are *identifiers*
- Identifiers are the *names* of values
- Identifiers usually consist of *alphanumeric* characters beginning with a *letter*
- Languages with modules often have names with a *prefix* that chooses the *module* follows by an *identifier* that selects the desired *module component*

42

Identifiers

- In some language, identifiers are syntactically divided into separate categories forming different *name spaces*
- In ML, type variables are identifiers beginning with the *apostrophe*
- In Haskell, all types must begin with a *capital letter*

43

Literals

- A *literal* is a *constant expression* whose value can be *immediately determined*

365	integer
5.11E-8	real
TRUE	boolean value
'A'	character
"string"	string
[ ]	empty list

44

Constants

- A *constant* is an identifier whose *r-value* does not change at run time
- If the value of a constant can be determined at compile time, it is said to be a *static constant*

```
LIMIT : constant INTEGER := 10_000;
LOW_LIMIT : constant INTEGER := LIMIT/10;
TOLERANCE :
    constant FLOAT := DISPERSION(1.15)
```

45

Operators

- *Functions* may operate on constants, locations, and the values of other functions to compose more complex expressions
- Many such functions are built in to the *language definition*, or its *environment*
- Almost every language permits the programmer to *define additional functions*

46

Operators in Ada

and	1	conjunction
or	1	disjunction
xor	1	exclusive or
and then	1	conditional and
or else	1	conditional or
=	2	equality
/=	2	inequality
<	2	less than
<=	2	less or equal
>	2	greater than
>=	2	greater or equal
in	2	membership
not in	2	not member
+	3	addition
-	3	subtraction
&	3	concatenation
*	4	multiplication
/	4	division
mod	4	modulus
rem	4	remainder
**	5	exponentiation
abs	5	absolute value
not	5	negation

47

Overloaded Operators

- In Ada only the listed operators may be used in their *infix* form
- In Ada it is not possible to change the "*fixity*" or *precedence* of operators
- In Ada operators and functions may be *overloaded* by the user

48

Short-Circuit Operators

- For *short-circuit operators*, the operand on the *left* hand side is evaluated *first* and the operand on the *right* hand side is evaluated *only if necessary*
- Sometimes the user wants everything evaluated (for side effects); sometimes it is more efficient to evaluate only as much as needed
- In Ada *both* operators are supported: *and then*, *or else*, and *and*, *or*

49

Expression Languages

- In some languages like C, APL, LISP, and ALGOL 68 *assignments are expressions* (with side effects)

```
i = j = k = 1;      /* C */
```

- An *expression language* is one in which every statement or group of statements can deliver a value

50

Expression Languages

- The *comma operator* in C ensures that side effects can be fit in as many parts of the language as possible

```
/*  reverses the string s in place */
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

51

Referential Transparency

- A language is *referential transparent* if the meaning of a sentence remains unchanged when a part of the sentence is replaced by an expression having the same meaning

Tully was a roman
Cicero was a roman
William Rufus was so-called because of the color of his hair
King William II was so-called because of the color of his hair

52

Referential Transparency

- A language in which the context does not affect the meaning of expressions is said to be *referential transparency*
- In *mathematics* expressions denote the same values regardless of context
- In *imperative* programming languages, this property is violated

53