Expressions

- operands & operators
- precedence rules
  - Java: nested first, then precedence, then L→R

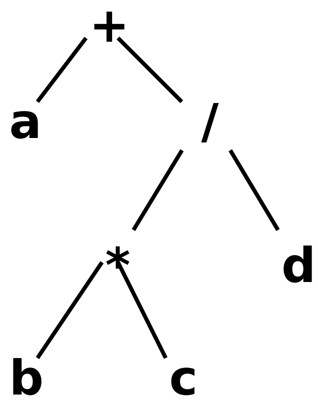
    Java operator precedence (highest at top)

```
!
*, /, %
+, -
>, <, >=, <=
==, !=
&&
||
=
```

  - APL: all operators have the same precedence, R→L
  - Pascal: fewer levels of precedence and Boolean operators have higher precedence than the relational operators:

```
if (a=b) | c then … ;
```

- expression tree, e.g. `a+b*c/d`



- side effects
  - function calls
- operator vs function
- operator forms
  - binary (dyadic), unary (monadic), trinary (tridaic)
  - overloading

Boolean Expressions

- logical values
- relational operators
- Boolean operators
  - and, or, not
- precedence
  - Ada vs Pascal

```
a > b and c + 4 < d
```

- short-circuit

```
if (i < 0) or (a[i] > a[i+1]) then …
```

  - operators

Java	Ada
`&&`	`andthen`
`||`	`orelse`
`&`	`and`
`|`	`or`

Mixed-mode Expressions

- operands of different types

```
var a : integer;
var b : float;
        :
… a + b …
```

  - strongly-typed → not legal, e.g. Ada
  - firmly-typed → widening conversion occurs, e.g. Java
  - weakly-typed → most conversions occur, e.g. C
- casting

```
… Float(a) + b …
```

Statements

- change state and perform actions
- assignment statement
- conditional statement
- looping statement
- procedure call statement
- Algol 68 - no difference between statement and expression

Assignment Statement

- l-value vs r-value
- restrictions on l-value
  - variable vs expression
- conversions
  - strongly-typed vs firmly-typed vs weakly-typed
  - widening vs narrowing
  - casting
- assignment as operator
  - Compound assignment in C, C++ and Java (`+=`, `-=`, `*=`, `/=`)

    `a += expression;` vs `a = a + expression;`
  - increment and decrement operator (`++`,`--`)

    `a = 1; b = ++a;` vs `a = 1; b = a++;`

- multiple assignment
  - assignment as expression (Algol 68, C, C++, Java)

```
a = b = c = expression;
```

  - lists or pairs as a left-hand side (Perl)

```
($a, $b) = ($b, $a);
```

Compound Statement

- where control structures allow only a statement
  - e.g. Pascal, Java

```
if (a < b)
  c = d;
```

and

```
if (a < b) {
   c = d
   e = f;
}
```

- explicit statement terminators
  - e.g. Ada

```
if (a < b) then c := d; e := f; end if;
```

  - advantages
- vs blocks
  - declarations

Selection Statements - If

- FORTRAN
  - arithmetic if `IF (C1) L1, L2, L3`
  - logical if `IF (logical expression) statement`
  - implied goto
- ALGOL 60
  - single statement
  - compound statement
  - single entry, single exit - structures statements
  - dangling else (Pascal, C, Java)

```
if C1 then if C2 then S1 else S2
```

- Ada if
  - statement terminator
  - `elsif`

Conditional Operator

- There's a ternary operator offered by some languages that's commonly called a 'conditional operator' (or an 'inline-if')
  - I don't feel like making a slide about it. So remind me to do it on the board, k?

Selection Statements - Case

- early versions - computed GOTO (FORTRAN) `GOTO (10,20,30) I`
- first implemented in ALGOL W
- Pascal `case month of`
  - discrete type `1, 3, 5, 7, 8, 10, 12 : days := 31;`

```
 4, 6, 9, 11 : days := 30;
 2 : if years mod 4 = 0
then days := 29
else days := 28
end
```

- Ada `case ch is`
  - ranges of values `when '0' .. '9' => put_line("digit");`
  - default condition `when 'A' .. 'Z' => put_line("letter");`

```
 when others     => put_line("special");
end case;
```

- C, C++, Java
  - switch statement
  - closer to computed `goto`
  - use of `break`

```
switch (month) {
  case 4:
  case 6:
  case 9:
  case 11: days = 30;
           break;
  case 2:  if (years % 4 == 0) days = 29; else days =
  28;
           break;
  default: days = 31;
           break;
}
```

Conditional Loop

- while statement
  - pre-test loop
  - compound statement vs terminator
  - `break` or `exit`
- repeat statement
  - post-test loop
  - Pascal vs C, Java
- loop statement
  - Ada - infinite loop
  - `exit` or `break`
  - in-test loop
  - multi-exit loop

```
sum := 0;
      loop
  get(number);
   exit when number < 0;
  sum := sum + number;
end loop;
```

Fixed Iteration Loop

- `DO` statement (FORTRAN)
- `for` statement
  - fixed increment - Pascal, Ada

    `for cv in low .. high loop … end loop;`
  - generalized increment - ALGOL 60 (index can be real)

    `for i := 0.0 step 0.1 until 6.0 do S`
- final index value
  - first that fails vs last that passes vs undefined
  - scope of index

    `for (int i = 0; i < n; i++) S`

- predetermination of number of repetitions
  - test and final value expressions
  - changes to index within loop
  - premature exit from loop
  - e.g.

```
j := 2;
for i := 1 to 10*j step 7-j do
    j := j + 1;
end;
```

- Pascal (final value and increment are computed once)

i	j	final value	increment	new j
1	2	20	5	3
6	3	20	5	4
11	4	20	5	5
16	5	20	5	6

- Algol60, C, C++, Java (final value and increment are computed in every iteration, negative increment is decrement)

i	j	final value	increment	new j
1	2	20	5	3
6	3	30	4	4
10	4	40	3	5
13	5	50	2	6
15	6	60	1	7
16	7	70	0	8
16	8	80	-1	9
15	9	90	-2	10
:	:	:	:	:

- C, C++ & Java `for` loop
  - really a `while` loop

    ```
    for (e1; e2 ; e3) S  ⇔     e1;
                               while (e2) {S; e3}
    ```

  - may lead to unreadable programs

    ```
    for(i = f = 1; i < n; f *= ++i) {}
    ```

    - what is computed by this program?

Iterator

- iterate over all elements of a data structure
- e.g. Java 1.5

  `for (  VarDcl : Expression ) Statement`

  - where `Expression` is an array or an object implementing `Iterable`
  - the statement is executed with the variable referencing the elements of the array or data structure in no specified order
  - e.g.

```
int  a[];     List  l;
a = new int[100];    l = new List();
        :            :
for ( int e : a ) { for ( Object o : l ) {
    sum = sum + e;       out.println(o);
};     };
```

`Goto` Statement

- heavily used in early languages
  - machine language branch instruction
- Dijkstra's "Goto Statement Considered Harmful"
  - reasoning about programs
  - structured programming
- restricted `goto`'s
  - `break`, `exit`
    - with label
  - `continue`
  - exception handling

Example Time!

Exception Handling

- exception
  - exceptional but not necessarily unexpected event
  - unexpected event
- handling exceptions
  - up to programmer, else crash
  - language support
- exception handling
  - PL/I, Ada, Java, C++
  - event occurs, exception raised (thrown), current unit terminated, exception handler invoked
- user defined exceptions

- in Ada & Java
  - handler attached to block (frame, try statement)
  - replaces execution of block
  - if not caught, is propagated to caller

Ada	Java
`begin`	`try {`
`   …`	`   …`
`   if a(i) < 0 then …`	`   if (a[i] < 0) …`
`   …`	`   …`
`exception`	`}`
`   when constraint_error => …`	`catch (ArrayIndexOutofBoundsException e) {`
`end;`	`… }`

- an example (Ada)

```
loop
   begin
      get(x);
      exit;
   exception
      when data_error =>
         skip_line;
         put_line("Error in number, try again");
   end;
end loop;
```

- Java
  - checked vs unchecked exceptions
  - exceptions are objects
- C++
  - more complex and less general
- Eiffel
  - two choices
    - report error and terminate
    - fix up and retry failed operation (retry command)

- exceptions shouldn’t be overused
  - only for exceptional conditions

```
type Days is
   (Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday);
```

```
if today = Saturday then
  tomorrow := Sunday;
else
   tomorrow := Days’succ(today);
```

```
begin
  tomorrow := Days’succ(today);
exception
   when constraint_error =>
     tomorrow := Sunday;
end;
```

Assertions?

- If I look bored, ask me about assertions

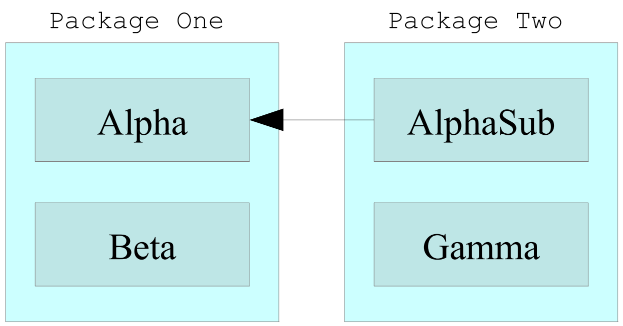
Program Structure

- effects of design process on language design
- procedural
  - stepwise refinement or top-down design
  - control structures
  - substeps are procedures
- abstraction
  - hide unwanted detail to control complexity
  - procedural abstraction
    - hide implementation
- encapsulation
  - prevent or control access to what is hidden
  - interface defines what is visible
    - signature for procedure

(Encapsulation)

- When it comes to Information Hiding, some languages (e.g. Java) permit a decent level of control
- Consider the following example from Oracle's Java tutorial page (http://download.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)

	Visibility of Alpha's Members			
Modifier	Alpha	Beta	AlphaSub	Gamma
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
(none)	Yes	Yes	No	No
private	Yes	No	No	No



- procedural abstraction doesn't scale up
  - teams of programmers
- modules
  - collections of related procedures and methods
  - encapsulation with well-defined interface
  - raises the level of abstraction
  - in object-oriented languages class is a module
- classes
  - represent real-world entities
  - same entities in analysis, design and implementation
  - attributes, services and behavior
  - hidden state, public interface
  - multiple instances
  - object identity
  - interaction
    - client object and server object

Abstract Data Types

- built-in types
  - set of values and set of operations
  - don’t need to know representation (encapsulation)
- set of types defines applicability of language
  - add more built-in types
  - define new types - extension
    - declare type and operations
    - hide representation and implementation
    - abstract data type
- module vs class
  - contains a type vs defines a type

Modifiability and Reuse

- modifiability
  - large systems
  - limit scope of change
  - loosely-coupled modules
- reuse
  - unit of reuse
    - procedure
    - module
    - class

Procedural vs Object-Oriented

- procedure as separate entity or part of class or both
- function vs procedure
  - change state vs product a value
  - pure function
  - don’t differentiate - Algol 68 & C++
- methods
  - aka procedures, member functions

Java

```
class BankAccount {
  // private attribute declarations
  …
  public BankAccount() {
  …
  } // constructor
  public void deposit(int amount) {
  …
        } //deposit
  public int getBalance() {
  …
  } // getBalance
} // BankAccount
```

Java

- declaration and creation

```
BankAccount bk1 = new BankAccount();
BankAccount bk2 = new BankAccount();
```

- method calls

```
bk1.deposit(6);
int am1 = bk1.getBalance();
```

Pascal

```
program BankAccountEx(input, output);
  type BankAccount = …
  var bk1, bk2 : BankAccount;
         am1 : Integer;
  …
  procedure makeBankAccount(var b : BankAccount);
  begin … end {makeBankAccount};

  procedure deposit(var b : BankAccount; amount : Integer);
  begin … end {deposit};

  function getBalance(b : BankAccount) : Integer;
  begin … end {getBalance};

begin
  makeBankAccount(bk1);
  makeBankAccount(bk2);
  … deposit(bk1,6);
  … am1 := getBalance(bk1); …
end.
```

Pascal

- monolithic
- type declarations
- initialization procedure
- “self” parameter
- no connection between type and procedures for type
- no hiding of representation of type

Ada

```
package BankAccounts is
  type BankAccount is private;
  procedure makeBankAccount(b : out BankAccount);
  procedure deposit(b : in out BankAccount;
                    amount : in Integer);
  function getBalance(b : BankAccount) return Integer;
private
  type BankAccount is …
end BankAccounts;


package body BankAccounts is
  -- definitions of makeBankAccount, deposit and getBalance
end BankAccounts;
```

Ada

- exports `BankAccount` type and procedures
- specification defines type and procedure signatures
- body gives implementation
- representation given in private part of specification