CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Final Review

Final Exam Review

- Comprehensive, but focused on the 2nd part of the course
- Final exam structure
  - Theory questions
    - True/false
    - Multiple choice
    - “Regular” questions (justify the answer)
  - Problems
    - Given some type definitions, specify if the types are equivalent under name equivalence / structural equivalence
    - Given a program, what does it print with parameter passing by value / reference / value-result / name?
    - Given a program, what is the content of the display or run-time stack (with its static chain) at some given moment?
    - Given a C++ program with static / dynamic method binding, what does it print?
    - Write a predicate in Prolog

Final Exam Review

- Final exam content (from the 2nd part of the course):
  - Chapters 7, 8 – Data types
  - Chapter 9 – Subroutines and control abstraction
  - The Prolog programming language
  - The Java programming language
  - Chapter 10 – Data Abstraction and Object Orientation

What Have We Accomplished?

- Programming languages – what is “under the hood”?
  - Language specification:
    - translation – regular expressions, grammars, scanning, parsing
  - Language implementation:
    - scopes and binding – scoping rules, symbol tables
    - control flow – evaluation, selection, loops, iteration, recursion
    - data types – type checking, allocation, garbage collection
    - subroutines – parameter passing, stack organization
    - data abstraction – modules, classes, inheritance, dynamic binding

- Useful programming constructs:
  - in-line functions
  - closures
  - unions
  - modules
  - iterators
  - coroutines and threads
  - exceptions
  - interfaces

What Have We Accomplished?

- Languages studied – why these?
  - Scheme – functional programming, clean syntax and semantics, recursion
  - Prolog – logic programming, unification, backtracking
  - Java – object-oriented, GUI – event-driven programming
- How do we think about a problem?
  - make best use of the language style
    - imperative, functional, logic, object-oriented
  - implementation
    - similar concepts across languages
    - different tools (recursion, backtracking)

Data Types

- Chapters 7, 8 – Data types
- Type checking
  - Type equivalence
  - Type conversion and casts
  - Type compatibility and coercion
  - Type inference
- Data types
  - Records
  - Variant records
  - Arrays
  - Pointers
  - …

Data Types

- Relation between an object type and the context where it is used:
  - Type equivalence
  - Type compatibility
  - Type inference

- Type equivalence:
  - Structural equivalence - same components, put together in the same way
  - Name equivalence - each definition introduces a new type
    - strict name equivalence – aliases are distinct
    - loose name equivalence – aliases are equivalent

Data Types

- Compatibility issues:
  - conversion (casting) - explicit
  - coercion - implicit
  - non-converting cast - does not change the bits, just interpret them as another type
- Type inference
  - Infer the type of an expression, given the types of its operands

Type Equivalence

- Which of the following types are equivalent?

```
type student = record
    name, address : string
    age : integer
type school = record
    name, address : string
    age : integer
type college = school
```

  - Under structural equivalence – student, school and college are all equivalent
  - Under strict name equivalence – student, school and college are all distinct
  - Under loose name equivalence – only school and college are equivalent

Data Types

- Records
  - Heterogeneous data
  - May have holes, due to alignment requirements
  - Bit-by-bit assignments and equality tests (problems)
  - with statements
- Variant records
  - Alternative fields (variants) – share memory space
  - Tag (discriminant)
- Arrays
  - Homogeneous data
  - Contiguous allocation (row-major, column-major)
  - Row pointers
  - Computing addresses

Data Types

- Pointers
  - to heap objects only, or also to static and stack objects
  - pointer-array duality in C

  - Dangling references – need to detect them
    - Tombstones
    - Locks and keys

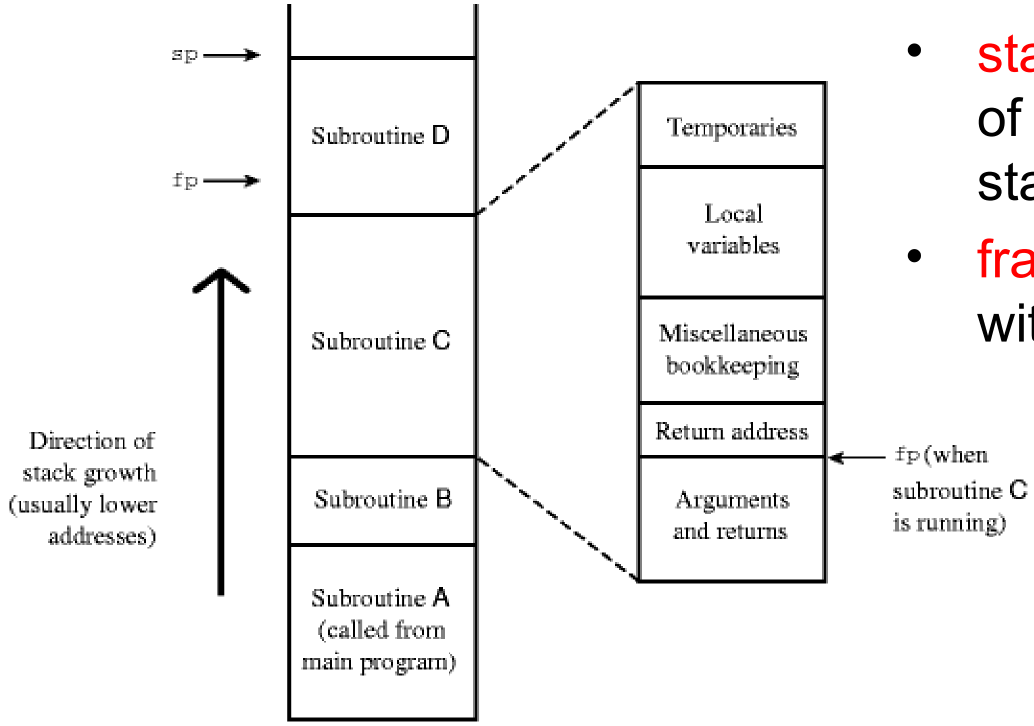
  - Garbage (memory leaks)
    - Reference counts – deallocate "on the fly"
    - Mark-and-sweep – pause execution to deallocate all garbage
      - Stop-and-copy – also does compaction

Subroutines and Control Abstraction

- Chapter 9 – Subroutines and control abstraction
- Stack layout
- Calling sequences
- Parameter passing
- Exception handling
- Iterators

Stack Layout

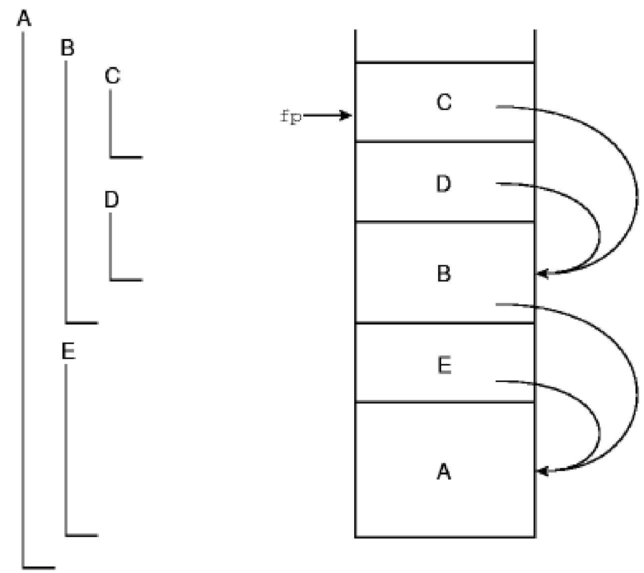
- When calling a subroutine, push a new entry on the stack - stack frame (activation record)
- When retuning from a subroutine, pop its frame from the stack



- stack pointer register (sp) - address of the first unused location at top of stack
- frame pointer register (fp) - address within current stack frame

Stack Layout

- In a language with nested subroutines - how can we access non-local objects?
  - static chain
  - display



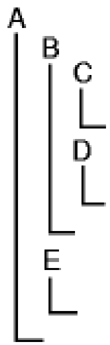
- Static chain - composed of static links:

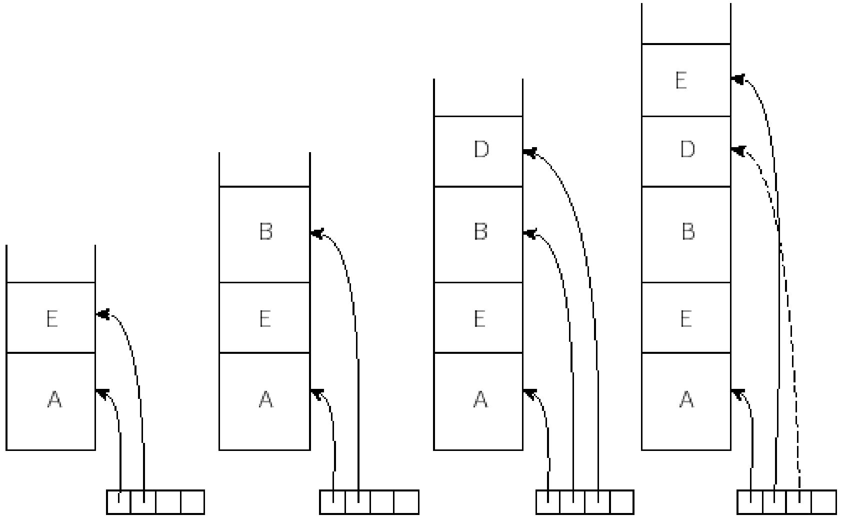
- Static link - from a subroutine to the lexically surrounding subroutine

- Disadvantage - to access an object k levels deeper → need to dereference k pointers

Stack Layout

- Display:





- Element j in display - reference to most recently called subroutine at lexical nesting level j
- From a subroutine at lexical level i, to access an object k levels outwards:
  - follow only one pointer, stored in element i-k in display
  - constant access time

Parameter Modes

- Main parameter-passing modes:
  - call by value
    - the value of actual parameter is copied into formal parameter
    - the two are independent
  - call by reference
    - the address of actual parameter is passed
    - the formal parameter is an alias for the actual parameter
- Speed vs. safety
- Semantic issue:
  - argument passed by reference - is it because it's large, or because changes should be allowed?
  - what if we want to pass a large argument, but not to allow changes?

Parameter Modes

- Ada:
  - three parameter modes:
    - in - read only
    - out - write only
    - in out - read and write
  - for scalar types - always pass values
  - call by value/result
    - if it's an out or in out parameter - copy formal into actual parameter upon return
    - change to actual parameter becomes visible only at return
- Algol 60, Simula: call by name
  - parameters are re-evaluated in the caller's referencing environment every time they are used
  - similar to a macro (textual expansion)

Exception Handling

- Example (C++):

```
void f ()
{
    ...
    try
    {
        g();
    }
    catch (exc)
    {
        // handle exception of type exc
    }
    ...
}
```

```
void g ()
{
    ...
    h();
    ...
}

void h()
{
    ...
    if (...)
        throw exc;
    ...
}
```

Exception Handling

- C++, Ada, Java, ML – structured approach:
  - handlers (catch in C++) are lexically bound to blocks of protected code (the code inside a try block in C++)

- Exception propagation:
  - if an exception is raised (throw in C++):
    - if the exception is not handled in the current subroutine, return abruptly from subroutine
    - return abruptly from each subroutine in the dynamic chain of calls, until a handler is found
    - if found, execute the handler, then continue with code after handler
    - if no handler is found until outermost level (main program), terminate program

Iterators

- Iterator - control abstraction that allows enumerating the items of an abstract data type
- Clu – a for loop implemented as an iterator:

```
for i in from_to_by (first, last, step) do
    ...
end
```

```
from_to_by = iter (from, to, by : int) yields (int)
    i : int := from
    if by > 0 then
        while i <= to do
            yield i
            i +:= by
        end
    else
        while i >= to do
            yield i
            i +:= by
        end
    end
end from_to_by
```

  - yield – returns control with current value of i
  - Next iteration – continues from where it has left

The Prolog Programming Language

- The Prolog programming language
- Clauses – facts, rules, queries
- Terms – constants, variables, structures
- Predicates
- Unification
- Backtracking
- Lists, recursion

The Prolog Programming Language

- General approach in logic programming:
  - Express the problem as a collection of relationships (constraints) between objects
  - The implementation will find the values that satisfy all constraints
- Problem: how many elements are in a list?
  - Imperative programming:
    - traverse the list from first element until last; at each element increment the number of elements N
  - Functional programming:
    - the number of elements N is 0 for an empty list; otherwise, it is 1 plus the number of elements in the list tail (without the first element)
  - Logic programming:
    - the proposition nr_elem (L, N) is true if
      - list L is empty and N is 0, or
      - list L has a head h and a tail t, and nr_elem (t, N-1) is true

Unification

- Prolog rules for unification:

- A constant unifies only with itself

- An uninstantiated variable unifies with any object
  - if the object has a value (is a constant or an instantiated variable), the first variable becomes instantiated to that object
  - if the object is an uninstantiated variable, the two variables will remain uninstantiated, but will co-refer

- A structure will unify with another structure if they have the same functor and arity (number of arguments), and the corresponding arguments also unify recursively; same rule applies for predicates

Recursion

- Substitute every occurrence of X with Y in a list:

```
subst(X, Y, [], []).
subst(X, Y, [X|T], [Y|Z]) :- subst(X, Y, T, Z).
subst(X, Y, [H|T], [H|Z]) :- subst(X, Y, T, Z).
```

- Take the first N elements from a list:

```
take(_, 0, []).
take([], _, []).
take([H|T], N, [H|R]) :- N1 is N-1, take(T, N1, R).
```