Semester Review

CSC 301

Programming Language Classes

There are many different programming language classes, but four classes or paradigms stand out:

- Imperative Languages
  - assignment and iteration
- Functional Languages
  - recursion and single valued variables
- Logic/Rule Based Languages
  - programs consist of **rules** that specify the problem solution - axiomatization
- Object-Oriented Languages
  - bundle data with the allowed operations ☞ Objects

Formal Language Specification

Language Specifications consist of two parts:

- The syntax of a programming language is the part of the language definition that says what programs look like; their form and structure.
- The semantics of a programming language is the part of the language definition that says what programs do; their behavior and meaning.

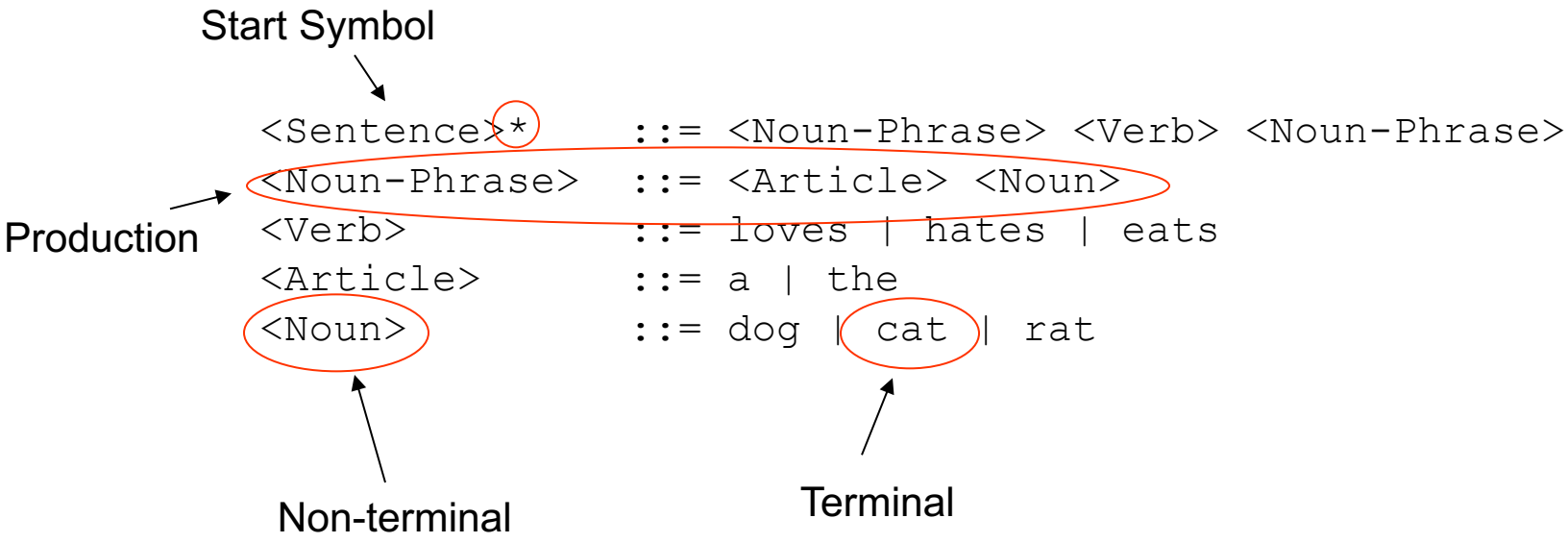
Formal Language Specification

In order to insure conciseness of language specifications we need tools:

- Grammars are used to define the syntax.
- Mathematical constructs (such as predicates and sets) are used to define the semantics.

Grammars

Example: a grammar for simple English sentences.



☞ Grammars capture the structure of a language.

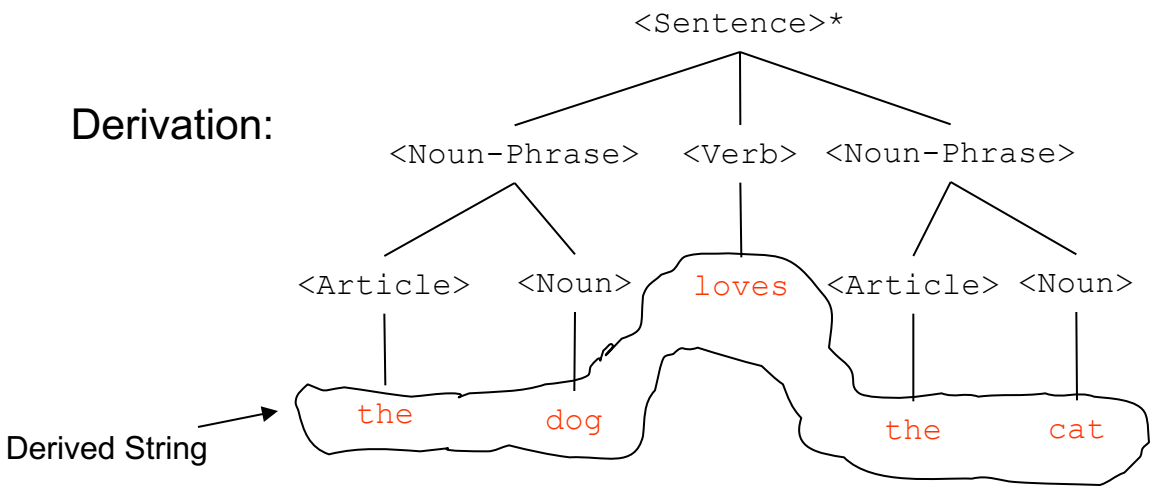
How do Grammars work?

We can view grammars as rules for building parse trees or derivation trees for sentences in the language defined by the grammar. In these parse or derivation trees the start symbol will always be at the root of the tree.

```
<Sentence>*        ::= <Noun-Phrase> <Verb> <Noun-Phrase>
<Noun-Phrase>      ::= <Article> <Noun>
<Verb>             ::= loves | hates | eats
<Article>          ::= a | the
<Noun>             ::= dog | cat | rat
```

$L(G) = \{ s \mid s \text{ can be derived from } G \}$

Derivation:

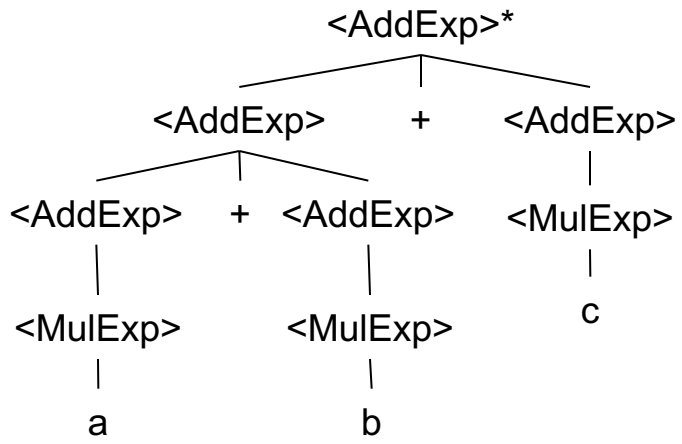


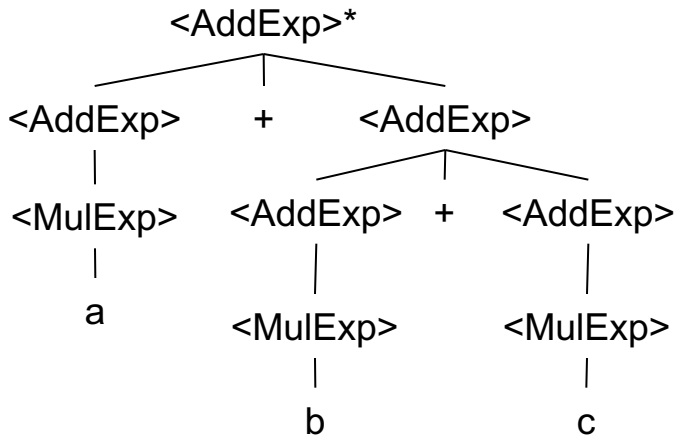
Grammars and Semantics

Given grammar G, consider the sentence a+b+c; here we have two possible parse trees:

```
G: <AddExp> ::= <AddExp> + <AddExp>
             |   <MulExp>
   <MulExp> ::= <MulExpr> * <MulExp>
             | a | b | c
```

A grammar is ambiguous if there exists more than one parse tree for a string of terminals.

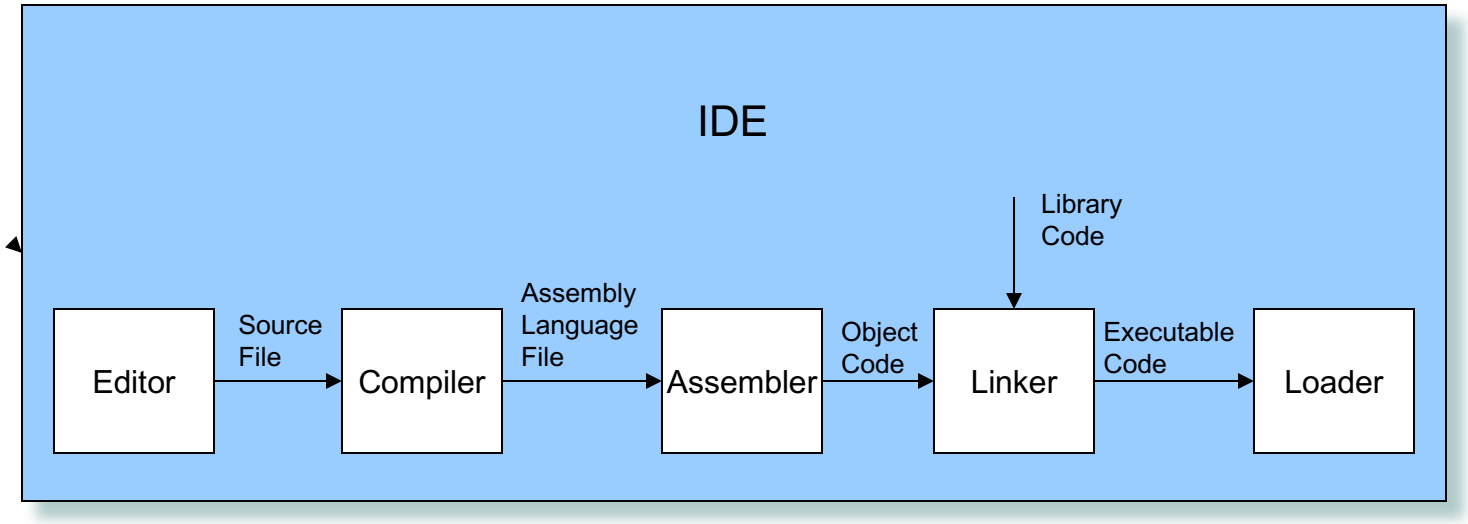




Language Systems

What actually happens in your IDE? IDE ≡ Integrated Development Environment

Classical Sequence: C++, C, Fortran



NOTE: The IDE is not a compiler, it contains a compiler.

NOTE: Many different IDE structures possible, depending on the language.

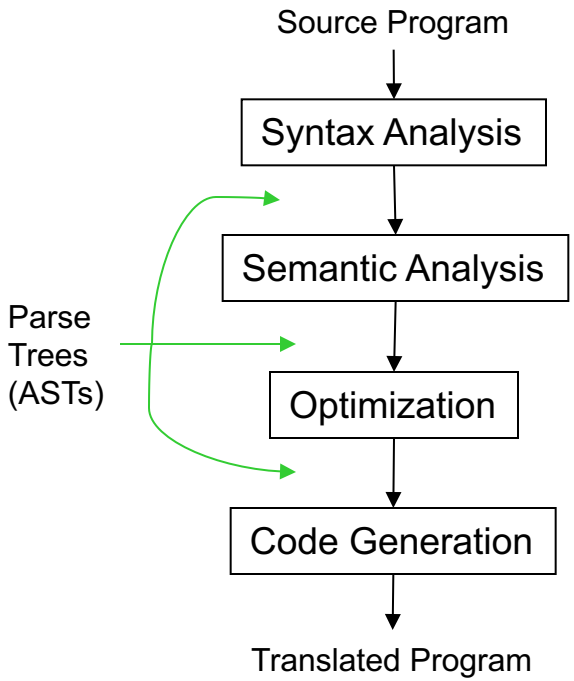
Compilers vs. Interpreters

- Compilers translate high-level languages (Java, C, C++) into low-level languages (Java Byte Code, assembly language).
- Interpreters execute high-level languages directly (Lisp).

Observation: Virtual machines can be considered interpreters for low-level languages; they directly execute a low-level language without first translating it.

Observation: Compilers can generate very efficient code and, consequently, compiled programs run faster than interpreted programs.

The Anatomy of a Compiler



Recognize the structure of a source program, generate parse tree

Recognize/validate the meaning of a source program

Recognition Phases

Reorganize the parse tree/AST to make computations more efficient

Translate parse tree/AST into low-level language

Code Generation Phases

Observations:

-Language definitions have two parts: syntax and semantics

-Compilers have two phases which deal with each of these language definition components: syntax analysis, semantic analysis.

ML

ML is a functional programming language, typical statements in this language are:

```
- fun reverse ([ ]) = [ ]
      | reverse (x :: xs) = reverse(xs) @ [x];

- map (fn x => x + 2) [1,2,3];
```

Polymorphism

polymorphism ≡ comes from Greek meaning ‘many forms’

In programming:

Def: A function or operator is polymorphic if it has at least two possible types.

Polymorphism

i) Overloading

Def: An overloaded function name or operator is one that has at least two definitions, all of different types.

ii) Parameter Coercion

Def: An implicit type conversion is called a coercion.

iii) Parametric Polymorphism

Def: A function exhibits parametric polymorphism if it has a type that contains one or more type variables.

iv) Subtype Polymorphism

Def: A function or operator exhibits subtype polymorphism if one or more of its constructed types have subtypes.

Note: one way to think about this is that this is type coercion on constructed types.

Scope & Namespaces

Def: A definition is anything that establishes a possible binding to a name.

Def: Scope is a programming language tool to limit the visibility of definitions.

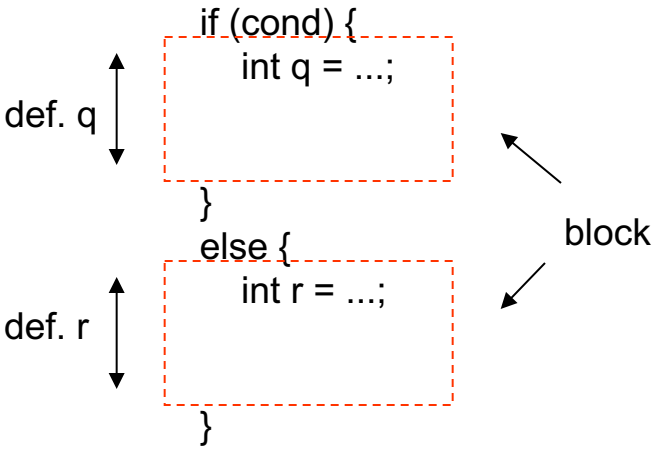
Def: A namespace is a zone in a programming language which is populated by names. In a namespace, each name must be unique.

The most common namespace in programming languages is the block.

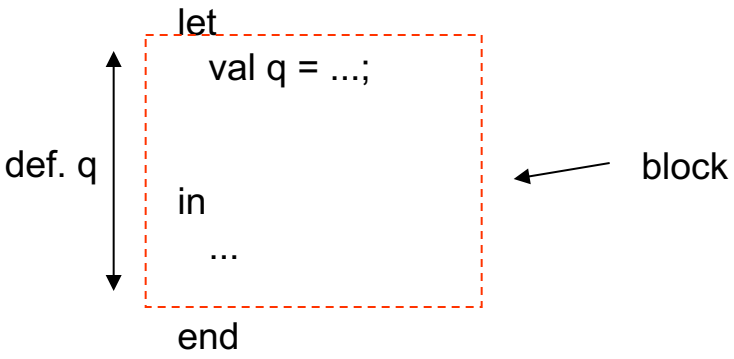
Scoping with Blocks

Def: A block is any language construct that contains definitions and delineates the region of the program where those definitions apply.

Example: Java



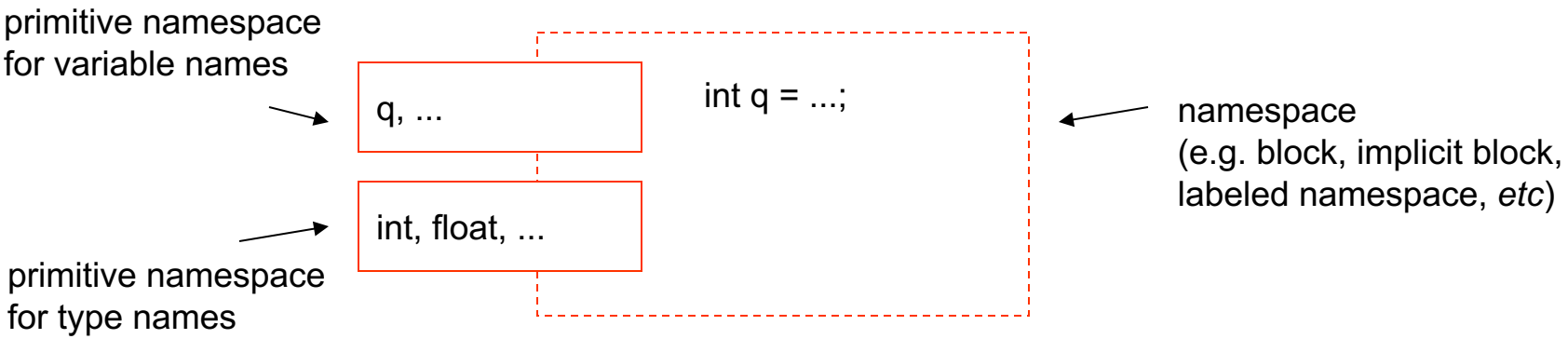
Example: ML



Primitive Namespaces

Def: A primitive namespace is a language construct that contains definitions and delineates a region of the program where those definitions apply; but the region was defined at language design time (similar to primitive data types, you can use them but not define them).

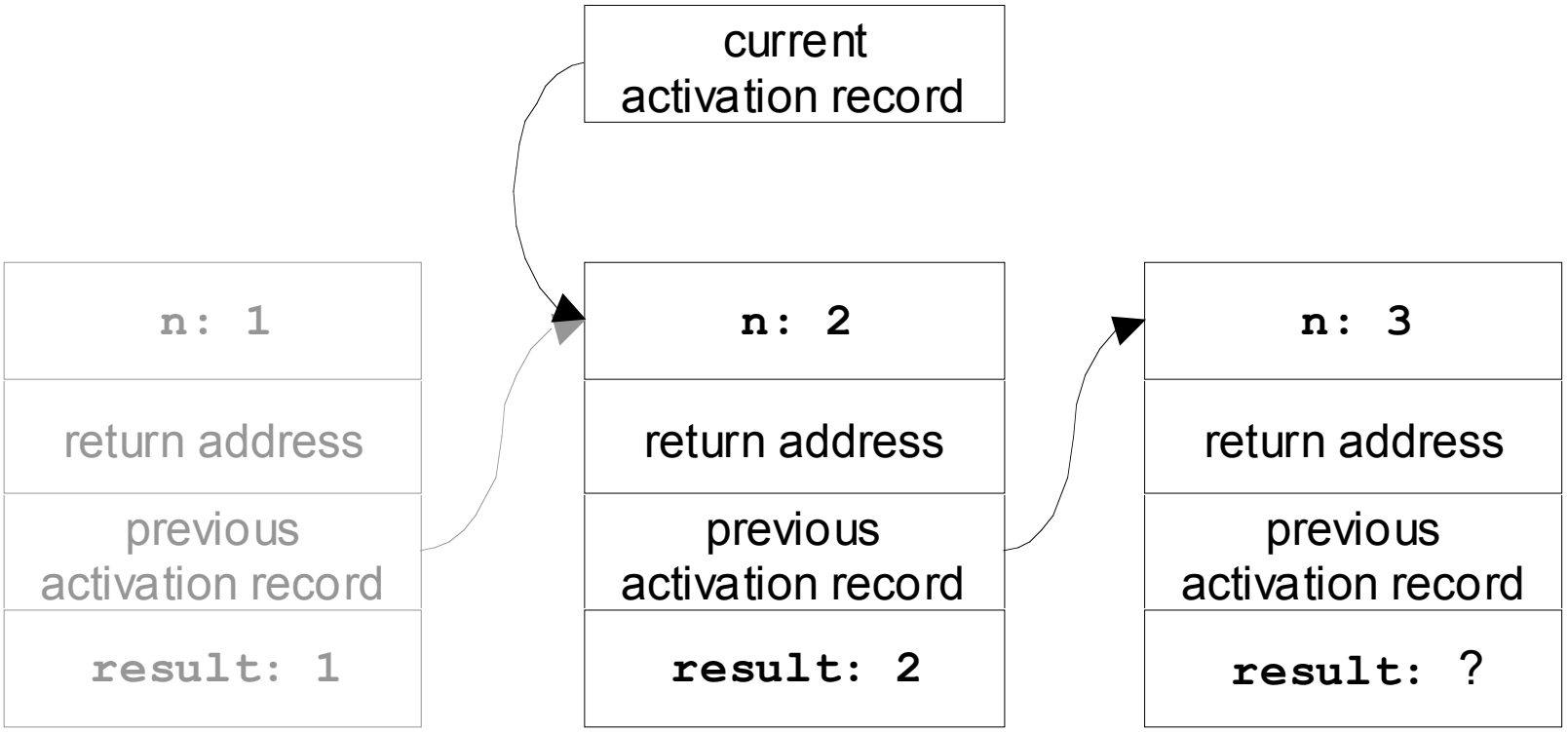
Most modern programming languages define two primitive namespaces – one for user defined variable names and one for type names (both primitive and constructed).



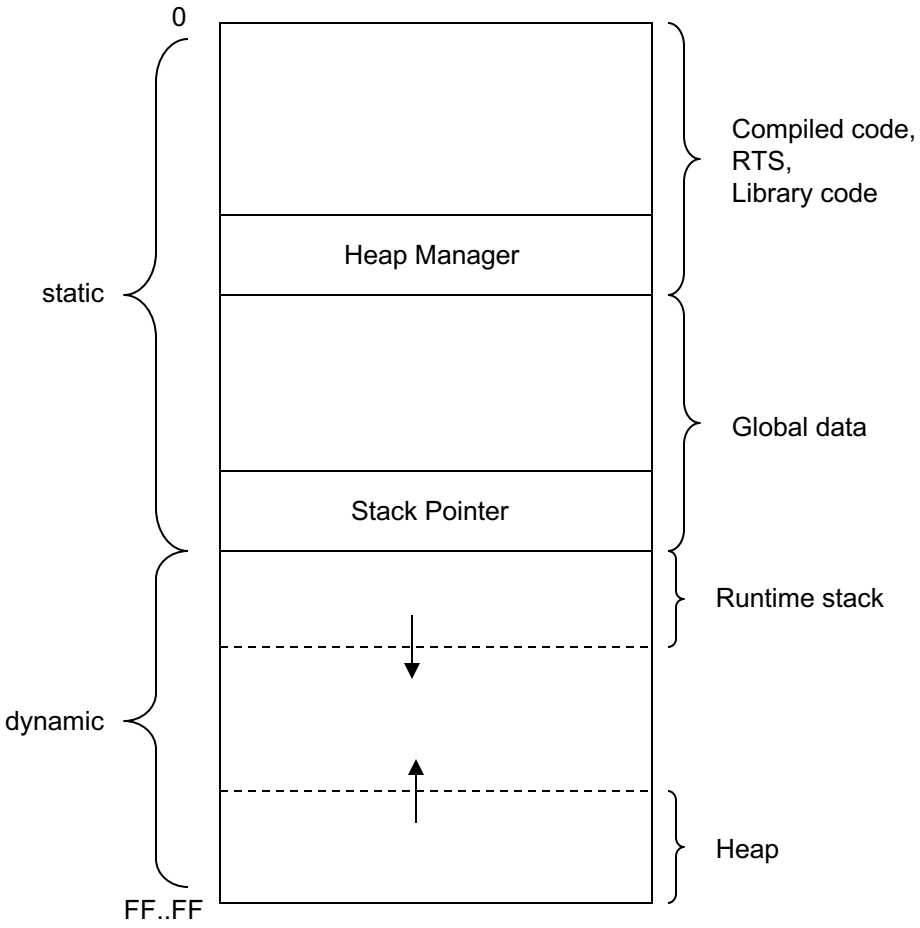
Activation Records & Runtime Stack

The second activation is about to return.

```
int fact(int n) {
  int result;
  if (n<2) result = 1;
  else result = n * fact(n-1);
  return result;
}
```



Memory Management



A typical memory layout for languages such as C and Java

NOTES:

(1) if the runtime stack and the heap meet ⇒ out of memory

(2) Also: memory leaks, dangling pointers and garbage collection…

Parameter Passing

- How is the correspondence between acutal and formal parameters established?
  - Most often positional correspondence
- How is the value of an actual parameter transmitted to a formal parameter?
  - Most popular techniques: by-value, by-reference

Prolog

- Programming language based on first-order logic
  - Predicates
  - Quantification
  - Modus-ponens
- Can be made executable using Horn-clause logic
  - *Deduction is computation!*

Prolog

Typical Programs:

```
last([A],A).
last([A|L],E) :- last(L,E).

append([ ], List, List).
append([H|T], List, [H|Result]) :- append(T, List, Result).

length([ ], 0).
length(L, N) :- L = [H|T], length(T,NT), N is NT + 1.
```

Formal Semantics

Grammars define the structure of a language, but what defines semantics or meaning?

⇒ Behavior!

The most straight forward way to define semantics is to provide a simple interpreter for the programming language that highlights the behavior of the language,

⇒ Operational Semantics

We used Prolog to define abstract interpreters for our languages, I.e., operational semantics for these languages.