UCLA COMPUTER SCIENCE FIELD NOTES

# CS 131

## Programming Languages

Syntax, semantics, types, scope, functional, object-oriented, concurrent, and logic programming

**How these notes are written.** I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

**Daniel Mastick**

Curriculum map, working notes, practice, and solutions

## Course map

| Week | Core thread | What should be solid by the end |
|---|---|---|
| 1 | **Syntax, grammars, and parsing** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 2 | **Operational and denotational semantics** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 3 | **Scope, binding, and closures** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 4 | **Type systems and inference** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 5 | **Functional programming** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 6 | **Object-oriented and prototype models** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 7 | **Concurrency and control effects** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 8 | **Logic programming and language evaluation** | Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision. |
| 9 | **Integration workshop** | Connect at least three chapters in one end-to-end problem and explain where their contracts meet. |
| 10 | **Synthesis and project review** | Audit assumptions and explain a complete solution from interface to failure handling. |

Scope anchor: https://catalog.registrar.ucla.edu/course/2025/COMSCI131. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

## How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

# Contents

# CHAPTER 1 Syntax, grammars, and parsing

**The idea**

Concrete syntax is written form; abstract syntax captures structure. Grammars define legal programs, while lexical analysis, parsing, precedence, associativity, and ambiguity determine how text becomes an AST.

## Rebuild the chapter from first principles

The claim I need to own is this: Concrete syntax is written form; abstract syntax captures structure. Grammars define legal programs, while lexical analysis, parsing, precedence, associativity, and ambiguity determine how text becomes an AST. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of syntax, grammars, and parsing. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

## Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach syntax, grammars, and parsing on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

### How this chapter connects

I read this chapter as the bridge from the course foundations to operational and denotational semantics. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

### What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Concrete syntax is written form; abstract syntax captures structure.
- **Mechanism.** Grammars define legal programs, while lexical analysis, parsing, precedence, associativity, and ambiguity determine how text becomes an AST.
- **Boundary.** The useful test is whether I can apply syntax, grammars, and parsing to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when syntax, grammars, and parsing is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

I work from the AST because punctuation is not the program's real structure.

**The easy version**

Concrete syntax is written form; abstract syntax captures structure.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Resolve 1+2*3 with precedence. The conclusion is The AST has addition at the root with left child 1 and right child multiplication of 2 and 3, producing 7. A flat ambiguous grammar needs precedence rules.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Resolve 1+2*3 with precedence.
**Solution.** The AST has addition at the root with left child 1 and right child multiplication of 2 and 3, producing 7. A flat ambiguous grammar needs precedence rules.

**Practice 2.** Give a short oral explanation of syntax, grammars, and parsing that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct syntax, grammars, and parsing from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a syntax, grammars, and parsing problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

## CHAPTER 2 Operational and denotational semantics

**The idea**

Semantics assigns meaning to syntax. Small-step rules model individual transitions; big-step rules relate expressions to final results; environments map names and stores map locations to values.

### Rebuild the chapter from first principles

The claim I need to own is this: Semantics assigns meaning to syntax. Small-step rules model individual transitions; big-step rules relate expressions to final results; environments map names and stores map locations to values. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of operational and denotational semantics. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

## Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach operational and denotational semantics on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

## How this chapter connects

I read this chapter as the bridge from syntax, grammars, and parsing to scope, binding, and closures. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

## What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Semantics assigns meaning to syntax.
- **Mechanism.** Small-step rules model individual transitions; big-step rules relate expressions to final results; environments map names and stores map locations to values.
- **Boundary.** The useful test is whether I can apply operational and denotational semantics to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when operational and denotational semantics is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

I keep environment and store separate so aliasing and assignment remain visible.

**The easy version**

Semantics assigns meaning to syntax.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Evaluate a let binding with shadowing. The conclusion is Extend the environment for the inner scope without mutating the outer binding; lookup chooses the nearest binding, and leaving the scope restores the outer environment.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Evaluate a let binding with shadowing.
**Solution.** Extend the environment for the inner scope without mutating the outer binding; lookup chooses the nearest binding, and leaving the scope restores the outer environment.

**Practice 2.** Give a short oral explanation of operational and denotational semantics that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct operational and denotational semantics from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects

and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a operational and denotational semantics problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

## CHAPTER 3 Scope, binding, and closures

**The idea**

Lexical scope resolves names by program structure; dynamic scope uses the call chain. A closure packages code with the environment needed for its free variables. Parameter passing and mutation determine alias behavior.

### Rebuild the chapter from first principles

The claim I need to own is this: Lexical scope resolves names by program structure; dynamic scope uses the call chain. A closure packages code with the environment needed for its free variables. Parameter passing and mutation determine alias behavior. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of scope, binding, and closures. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

### Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach scope, binding, and closures on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

### How this chapter connects

I read this chapter as the bridge from operational and denotational semantics to type systems and inference. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

### What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Lexical scope resolves names by program structure; dynamic scope uses the call chain.
- **Mechanism.** A closure packages code with the environment needed for its free variables.
- **Boundary.** Parameter passing and mutation determine alias behavior.
- **Decision test.** I should be able to say when scope, binding, and closures is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

My shorthand for this chapter is: my shorthand for this chapter is: a closure is not just a function pointer; it carries the remembered world its body expects.

**The easy version**

Lexical scope resolves names by program structure; dynamic scope uses the call chain.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: What does a counter closure capture? The conclusion is It captures a private mutable location for count. Each call reads and updates that same location; independently created counters capture different locations.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** What does a counter closure capture?
**Solution.** It captures a private mutable location for count. Each call reads and updates that same location; independently created counters capture different locations.

**Practice 2.** Give a short oral explanation of scope, binding, and closures that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct scope, binding, and closures from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a scope, binding, and closures problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

# CHAPTER 4 Type systems and inference

**The idea**

A type system classifies expressions to rule out certain behaviors before execution. Static versus dynamic checking, nominal versus structural typing, subtyping, parametric polymorphism, variance, and inference trade guarantees and flexibility.

## Rebuild the chapter from first principles

The claim I need to own is this: A type system classifies expressions to rule out certain behaviors before execution. Static versus dynamic checking, nominal versus structural typing, subtyping, parametric polymorphism, variance, and inference trade guarantees and flexibility. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of type systems and inference. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

## Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach type systems and inference on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

## How this chapter connects

I read this chapter as the bridge from scope, binding, and closures to functional programming. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

## What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

## Deep notes

- **Model.** A type system classifies expressions to rule out certain behaviors before execution.
- **Mechanism.** Static versus dynamic checking, nominal versus structural typing, subtyping, parametric polymorphism, variance, and inference trade guarantees and flexibility.
- **Boundary.** The useful test is whether I can apply type systems and inference to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when type systems and inference is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

My shorthand for this chapter is: my shorthand for this chapter is: types prove selected impossibilities, not total correctness.

**The easy version**

A type system classifies expressions to rule out certain behaviors before execution.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why is a mutable list of Dog not safely a list of Animal? The conclusion is A caller allowed to treat it as List<Animal> could insert a Cat, violating the original List<Dog>. Mutable generic containers are therefore invariant in common

systems.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Why is a mutable list of Dog not safely a list of Animal?
**Solution.** A caller allowed to treat it as List<Animal> could insert a Cat, violating the original List<Dog>. Mutable generic containers are therefore invariant in common systems.

**Practice 2.** Give a short oral explanation of type systems and inference that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct type systems and inference from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a type systems and inference problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

## CHAPTER 5 Functional programming

**The idea**

Functional languages emphasize immutable values, first-class functions, recursion, algebraic data types, pattern matching, higher-order operations, and controlled effects. Referential transparency supports local reasoning.

### Rebuild the chapter from first principles

The claim I need to own is this: Functional languages emphasize immutable values, first-class functions, recursion, algebraic data types, pattern matching, higher-order operations, and controlled effects. Referential transparency supports local reasoning. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of functional programming. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

### Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach functional programming on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

### How this chapter connects

I read this chapter as the bridge from type systems and inference to object-oriented and prototype models. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

### What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Functional languages emphasize immutable values, first-class functions, recursion, algebraic data types, pattern matching, higher-order operations, and controlled effects.
- **Mechanism.** Referential transparency supports local reasoning.
- **Boundary.** The useful test is whether I can apply functional programming to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when functional programming is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

I replace loop state with the value each recursive call promises to return.

**The easy version**

Functional languages emphasize immutable values, first-class functions, recursion, algebraic data types, pattern matching, higher-order operations, and controlled effects.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Define map recursively. The conclusion is Map over empty returns empty. Map f over head::tail returns f(head)::map f tail. Structural recursion decreases the list and preserves shape.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Define map recursively.
**Solution.** Map over empty returns empty. Map f over head::tail returns f(head)::map f tail. Structural recursion decreases the list and preserves shape.

**Practice 2.** Give a short oral explanation of functional programming that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that

licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct functional programming from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a functional programming problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

# CHAPTER 6 Object-oriented and prototype models

**The idea**

Objects combine identity, state, and behavior. Classes, prototypes, dispatch, encapsulation, inheritance, delegation, and mixins offer different reuse mechanisms. The fragile-base problem warns that inherited implementation creates hidden coupling.

## Rebuild the chapter from first principles

The claim I need to own is this: Objects combine identity, state, and behavior. Classes, prototypes, dispatch, encapsulation, inheritance, delegation, and mixins offer different reuse mechanisms. The fragile-base problem warns that inherited implementation creates hidden coupling. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of object-oriented and prototype models. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.

4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

### Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach object-oriented and prototype models on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

### How this chapter connects

I read this chapter as the bridge from functional programming to concurrency and control effects. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

### What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Objects combine identity, state, and behavior.
- **Mechanism.** Classes, prototypes, dispatch, encapsulation, inheritance, delegation, and mixins offer different reuse mechanisms.
- **Boundary.** The fragile-base problem warns that inherited implementation creates hidden coupling.
- **Decision test.** I should be able to say when object-oriented and prototype models is the right model, when its assumptions fail, and what evidence would change my choice.

### How I actually think about it

I distinguish subtype promises from code reuse; inheritance tries to do both and can do neither well.

### The easy version

Objects combine identity, state, and behavior.

### Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Explain dynamic dispatch. The conclusion is A call through a base interface selects the method associated with the runtime object's class or prototype chain, not merely the static reference type.

### Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

### Solved chapter practice

**Practice.** Explain dynamic dispatch.
**Solution.** A call through a base interface selects the method associated with the runtime object's class or prototype chain, not merely the static reference type.

**Practice 2.** Give a short oral explanation of object-oriented and prototype models that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct object-oriented and prototype models from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a object-oriented and prototype models problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

# CHAPTER 7 Concurrency and control effects

**The idea**

Threads, actors, async tasks, continuations, exceptions, generators, and coroutines change control flow. Shared memory requires synchronization; message passing shifts coordination into protocols. Structured concurrency ties task lifetime to lexical scope.

## Rebuild the chapter from first principles

The claim I need to own is this: Threads, actors, async tasks, continuations, exceptions, generators, and coroutines change control flow. Shared memory requires synchronization; message passing shifts coordination into protocols. Structured concurrency ties task lifetime to lexical scope. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of concurrency and control effects. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

## Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach concurrency and control effects on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

## How this chapter connects

I read this chapter as the bridge from object-oriented and prototype models to logic programming and language evaluation. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

## What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

## Deep notes

- **Model.** Threads, actors, async tasks, continuations, exceptions, generators, and coroutines change control flow.
- **Mechanism.** Shared memory requires synchronization; message passing shifts coordination into protocols.
- **Boundary.** Structured concurrency ties task lifetime to lexical scope.
- **Decision test.** I should be able to say when concurrency and control effects is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

I draw who can run, who waits, and who owns cancellation.

**The easy version**

Threads, actors, async tasks, continuations, exceptions, generators, and coroutines change control flow.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why is check-then-act racy? The conclusion is Another execution may change shared state between the check and action. Protect the compound operation with a lock or use an atomic primitive or ownership protocol.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Why is check-then-act racy?
**Solution.** Another execution may change shared state between the check and action. Protect the compound operation with a lock or use an atomic primitive or ownership protocol.

**Practice 2.** Give a short oral explanation of concurrency and control effects that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct concurrency and control effects from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a concurrency and control effects problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

## CHAPTER 8 Logic programming and language evaluation

**The idea**

Logic programs state facts and rules; unification solves symbolic constraints and search explores proofs. Backtracking order affects termination even when declarative meaning is unchanged. Language choice should be judged by domain fit, safety, tooling, and operational costs.

### Rebuild the chapter from first principles

The claim I need to own is this: Logic programs state facts and rules; unification solves symbolic constraints and search explores proofs. Backtracking order affects termination even when declarative meaning is unchanged. Language choice should be judged by domain fit, safety, tooling, and operational costs. I do not count that as learned just because the sentence looks familiar. I should be able to name

the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

**The move**

**My working sequence**

1. **Translate.** Rewrite the prompt in the language of logic programming and language evaluation. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

### Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

**Sanity check**

**Closed-note check.** Can I teach logic programming and language evaluation on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

### How this chapter connects

I read this chapter as the bridge from concurrency and control effects to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

### What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

### Deep notes

- **Model.** Logic programs state facts and rules; unification solves symbolic constraints and search explores proofs.
- **Mechanism.** Backtracking order affects termination even when declarative meaning is unchanged.
- **Boundary.** Language choice should be judged by domain fit, safety, tooling, and operational costs.
- **Decision test.** I should be able to say when logic programming and language evaluation is the right model, when its assumptions fail, and what evidence would change my choice.

**How I actually think about it**

My shorthand for this chapter is: my shorthand for this chapter is: declarative does not mean cost-free; the search strategy still decides whether an answer appears.

**The easy version**

Logic programs state facts and rules; unification solves symbolic constraints and search explores proofs.

**Worked example**

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Unify f(X,a) with f(b,Y). The conclusion is The most general substitution is X=b and Y=a. Function symbols and arity match, so corresponding arguments can be unified consistently.

**Common miss**

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

**Solved chapter practice**

**Practice.** Unify f(X,a) with f(b,Y).
**Solution.** The most general substitution is X=b and Y=a. Function symbols and arity match, so corresponding arguments can be unified consistently.

**Practice 2.** Give a short oral explanation of logic programming and language evaluation that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.

**Solution.** A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

**Mastery practice A.** Reconstruct logic programming and language evaluation from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
**Solution.** Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

**Mastery practice B.** A classmate gets a polished-looking answer to a logic programming and language evaluation problem but never states a model or checks the result. Write the shortest useful review of that solution.
**Solution.** I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

## Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.