Disclaimer: this presentation contains key concepts
All slides from the lectures are important (apart from those clearly marked non-examinable)
Topics

- Syntax to execution
- Types
- Reasoning about programs
- Concurrency in programming languages
- Semantics of concurrency
Syntax to execution

- What are the main PL families?
- What are some evaluation criteria of PL?
- What is syntax and how is it defined and handled?
- Names, variables, binding and scope
- Lexing and parsing: from program string to abstract syntax
- Compilers vs interpreters, VMs
- Evaluation strategies, dispatch
BNF and ambiguity

- BNF - formal way of presenting syntax
- Ambiguity - more than one way of deriving a sentence in some grammar
- Solving ambiguity:
  - operator precedence
  - if-then-else type ambiguity
- EBNF
Names and Variables

- Names (or identifiers) are a syntactic notion, with rules specific to each language
- Variables - place holders for elements of some type
- Binding - association between an entity and an attribute
  - e.g. variable and its type, variable and its scope
- **When** variables are bound is extremely important
  - static vs dynamic
- **How** variables are bound is also important
  - type inference vs type annotation
Lexing and parsing

- Lexer identifies the tokens
- Parser produces abstract syntax
- Lexers typically rely on regexp and finite automata

Parsing
- top-down, recursive descent, LL grammars
- bottom-up, LR grammars
Topics

- Syntax to execution
- Types
- Reasoning about programs
- Concurrency in programming languages
- Semantics of concurrency
Concurrency in programming languages

- Processes, threads, shared memory
- Races, critical regions and mutual exclusion
- Atomic instructions in hardware and software
- Locks and deadlocks
- Java monitors, volatile variables, atomic data structures
- Performance and scalability, Amdahl’s law
- CAS, Treiber stack
- Message passing and actor model
Threads and context switch

- The operation of switching control from one thread to another is called **context switch**
  - context switch happens at the granularity of machine level instructions
  - a context switch can happen in the middle of a “high-level” operation (e.g. assignment to a variable)
- On multi-many core, threads can run truly concurrently on different cores (and so have different caches! we will talk about **visibility** later this week)
- Shared memory concurrency is hard!
Race conditions

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_OF_TRANS 10

double accountBalance;

void withdraw(double outAmount) {
    if (accountBalance > outAmount) {
        accountBalance = accountBalance - outAmount;
        printf("Withdrew %.2f. Your balance is now %.2f.\n", outAmount, accountBalance);
    } else printf("You don't have enough cash!\n");
}

void credit(double inAmount) {
    accountBalance = accountBalance + inAmount;
    printf("Credited %.2f. Your balance is now %.2f.\n", inAmount, accountBalance);
}
```

What happens if we run `withdraw(10.0)` and `credit(20.0)` as two threads, supposing that `accountBalance` is initially 100.0?
Locks

- a classic programming abstraction for concurrency
- may use hardware atomicity guarantees or OS functionality for implementation
- it is a low-level abstraction that is difficult to get right and does not always interact well with other programming features
Problems with locks

❖ Locks often lead to deadlocks!
❖ e.g. classic dining philosopher problem

There is one fork between each philosopher
A philosopher can think or eat
To eat, a philosopher must pick up two forks
  (this is some kind of strange Dutch-American spaghetti)
Assume that every philosopher tries to pick up their left fork first

Starvation (or deadlock)
Critical regions

• one solution to problems with shared resources: memory, I/O, files etc

• **critical region**: part of program where a shared resource is accessed

• **mutual exclusion**: if one thread is in its critical region for a resource then no other threads are allowed to enter their critical regions for that resource.

• if no two threads are in a critical region at the same time then there will be no races
Java Monitors

• locking done by declaring certain methods synchronized
• one condition variable per object
• wait(), notify(), notifyAll() correspond to wait(&lock), signal() and broadcast()
Atomic data structures

- Atomic data structures are an example of **thread-safe** data structures
- e.g. `AtomicInteger` class
- some methods
  - `int get()`
  - `int set(int newValue)`
  - `int getAndAdd(int delta)`
- The operations are atomic and visible — thus atomic data structures are like a “better” volatile
Performance and scalability

- Threading can increase performance but adds overheads
- Scalability: can performance be increased by throwing more resources at the problem?
Amdahl’s Law

- Some tasks are naturally parallel: throwing extra cores at the problem improves performance
- Sequential tasks do not benefit at all from extra processors

**Amdahl’s law**: suppose that $F$ is the fraction of task that is sequential. Let $N$ be the number of processors available.

$$\text{speedup} \leq \frac{1}{F + \frac{1-F}{N}}$$

e.g. suppose that half of task is sequential. Then maximal speedup is 2, regardless of number of processors thrown at the problem!
Improving scalability

- **Reduce lock contention** (multiple threads competing for the same lock) is a problem
  - reduce the duration for which locks are held (reducing lock granularity)
  - reduce the frequency with which locks are requested
  - replace exclusive locks with other coordination mechanisms that permit greater concurrency
  - eg. fine-grained concurrency with CAS
Compare and Set (CAS)

- Supported by most processor architectures
  - \( \text{CAS}(V, A, B) \)
    - \( V \) - memory reference
    - \( A \) - old value
    - \( B \) - new value
- There are two possible executions
  - If \( V \) contains \( A \) then it is atomically changed to \( B \) and \( \text{CAS} \) returns \text{true}
  - If \( V \) does not contain \( A \) then nothing happens and \( \text{CAS} \) returns \text{false}
- When multiple threads attempt to modify a memory location using \( \text{CAS} \), one wins, the others lose
- From Java 5.0, \( \text{CAS} \) is available on \text{int}, \text{long} and object references: it is used in implementations of many thread safe data structures in \text{java.util.concurrent}
Message Passing

- forget about critical regions by not allowing shared memory — all communication is explicit: processes send each other messages
- easier: fewer risks of deadlock, races, visibility bus, etc.
- more general: distributed systems do not have shared memory
- increasing hardware support for message passing in many/multi core hardware
Synchrony vs Asynchrony

- Synchronous communication
  - sender blocks until receiver is ready to receive
  - popular e.g. in languages for hardware design
  - synchronous behaviour is difficult (sometimes impossible!) to guarantee in a distributed setting

- Asynchronous communication
  - sender sends and continues execution
  - callbacks
  - popular in distributed applications
Actor model

- each process is considered to be an **actor**
- each actor has an address and a message queue (typically FIFO)
- actors can send messages to each other’s addresses where they are enqueued on the message queue until processed
Topics

- Syntax to execution
- Types
- Reasoning about programs
- Concurrency in programming languages
- Semantics of concurrency
Semantics of concurrency

- Labelled transition systems
- Traces and trace equivalence
- Simulation and simulation equivalence (similarity)
  - simulation game
- Bisimulation and bisimulation equivalence (bisimilarity)
  - bisimulation game