Programming Paradigms Summary

Nora Jasharaj

Somewhere in 2025

Contents

1 Syntax

1.0.1 Motivation

Goal: Specify a programming language.

1.1 Concepts to Specify Syntax

- Concatenation
- Alternation/Choice (das Ding ist da, das andere au und dann aber au das ohne das andere au)
- Repetition (Kleene closure *)
- Recursion
- Regular expr → Recognized by scanners
- CFGs → Recognized by parsers

Concepts 1-3 are included in RegEx
⇒ **recognized by scanners** With 4 we are in total in context-free grammars (CFG).

1.1.1 Syntax vs Semantics

Syntax: Structure of code
Semantics: Meaning of code
PLs can have different syntax but same semantics, no shit blyat.
⇒ **recognized by parsers**

1.1.2 Tokens

Definition:
Basic building blocks of every PL. Keywords,identifiers, constants,operators.
REGEX are used to specify tokens.
A regex is one of a character, the empty string, the concatenation of two regexes, a REGEX followed by the Kleene star *.
Numeric constants accepted by calculators.

```
    number -> integer | real
    integer -> digit digit * (also ein digit mit 0 oder mehr digits)
    real -> (integer exponent) | decimal (exponent | epsilon)
    decimal -> digit* /. digit | digit.) digit*
    exponent -> (e | E) (+ | - | epsilon) integer
    digit -> 0 | 1 |... | 9
```

More compact syntax for REGEX

```
Language of tokens: {c, cac, cbc, cacac, cbcbc, cacbc, cbcac,...}
Token -> cMore*
More -> AorB c
AorB -> a | b
```

```
Shorter notation: Nest all into one.

    c( (a | b) c ) *
    -> thats ur token in the nested, more compaact verison
```

1.1.3 Identifiers in popular PLs

Different PLs allow different identifiers.
Case sensitive vs case- insenstive.
Letters and digits are almost always allowed
Underscore: allowed in most languages
In addition to syntax rules: Conventions (for instance Java: ClassName, variableName)

1.1.4 Whitespace in popular PLs

Free format vs. formatting as syntax:
Spaces and tabs sometimes matter, line breaks sometimes matter (Python, js)

1.2 Context free grammars

basically REGEX + recursion
Ex: Arithmetic expressions
Definition CFG:
$G = (N, T, R, s)$
N = *finite* set of non-terminals (left side in grammar)
T = *finite* set of terminals = alphabet of the language = (for Pls) tokens of the language
R = *finite* relation from N to $(N \cup T)*$ = production rules
s = start symbol

1.2.1 Derivations

Create concrete strigs from the grammar.
Begin with start symbol, repeat until no non-terminams remain.
Example:

```
    expr -> id | number | expr) | expr op expr
    op -> + | -| * | /

    Derivation of foo * x + bar

    expr --> (derivation) expr op expr
    --> expre op id
    --> expr +  id
    --> expr op expr + id
    --> expr op id + id
    --> expr * id + id
    --> expr id * id + id
```

1.2.2 Parse trees

Tree structure representation of a derivation
root = start symbol
leaf nodes = tokens that result from derivation
und da fehlt noch etwas aber digga Syntaxbaum mein Gott nt so schwierig.
Example:

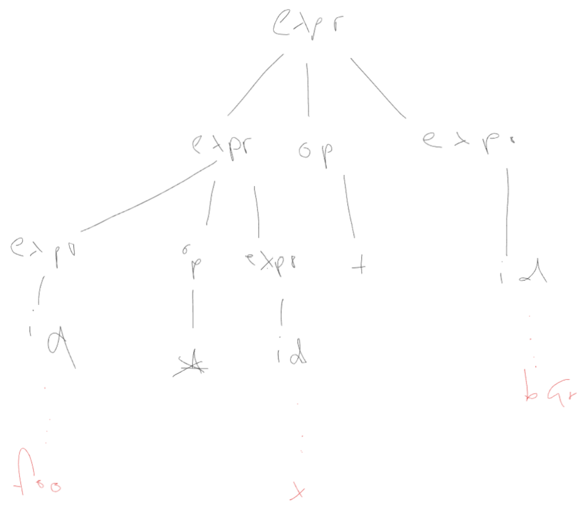


Figure 1: Example of a derivation of our string

Not all grammars are equal!
Each language has infinitely many grammars. Some grammars are even ambiguous (a single string may have multiple derivations; unambiguous grammars facilitate parsing)
A grammar should reflect the **internal structure** of the PL(E.g associativity and precedence of operators).

1.3 Scanning

1.3.1 Big Picture

Source code = sequence of characters
General idea:
read one character at a time, whenever a full token is recognized, return it.
When no token can be recognized, report an error.
Sometimes, need to look multiple characters ahead to determine the next token.

1.4 Option 1: Ad-hoc Scanners

- Manually implemented.
- Handle common tokens first.
- Used in production compilers: Compact code, efficient.

1.5 From Reg. Expr. to DFA

- Regex → NFA → DFA (avoid multiple states) → Minimal DFA (remove unreachable/non-distinguishable).

1.6 From DFA to Scanner

- Options: Switch statements (hand-written) or Table-based (auto-generated, driver indexes table).

1.7 Table-based Scanning

- Transition table by state/input.
- Driver: Moves state, returns token, or errors.

1.8 Recognizing Multiple Tokens

- Merge token automata with ϵ to starts.
- Apply NFA-DFA transformation.

1.9 Longest Possible Token Rule

- If prefix overlap (e.g., 3.1 vs 3.141): Accept longest.
- Look ahead 1 char to decide end.

1.10 Top-down vs. Bottom-up Parsing

- Top-down: From root, expand non-terminals, predict rule.
- Bottom-up: Combine tokens to subtrees, add parents.

1.11 Classes of Parsing Algorithms

	LL(k)	LR(k)
Tree construction	Top-down	Bottom-up
Scanning	Left-to-right	Left-to-right
Derivations	Left-most	Right-most
Algorithm	Predictive	Shift-reduce

1.12 Top-down Parsing

- LL(k): Left-to-right, Left-most, k lookahead.
- Approaches: Recursive descent (manual for simple langs) or Table-driven (auto table + driver).

1.13 Recursive Descent parser:

One function for each non-terminal N:
Chooses production based on next k non-terminal on the right-hand on right-hand side, call their function.
For terminals on the right-hand side, call *match function (consumes input token(is expected) or raises an error*

1.14 Generating Top-Down Parser

- LL(k): Predict rule using PREDICT sets from FIRST(N) (terminals first when expanding N) and FOLLOW(N) (terminals after N).
- Computing FIRST: For A $\rightarrow$ X1...Xk, add FIRST(X1) - , then if in FIRST(X1), add FIRST(X2), etc.; if all nullable, add .
- Computing FOLLOW: Start symbol gets EOF; for B $\rightarrow$ A, add FIRST() - to FOLLOW(A); if nullable or no , add FOLLOW(B) to FOLLOW(A).
- PREDICT(A $\rightarrow$): If not in FIRST(), = FIRST(); else (FIRST() -) FOLLOW(A).
- Parse Table: For each A $\rightarrow$, for t in PREDICT, M[A,t] = A $\rightarrow$; undefined = error.

1.15 Table-based Predictive Parsing

- Stack: Push EOF, start symbol.
- Repeat: Pop x; if terminal/EOF, match next token; if non-terminal, push RHS of M[x, next] in reverse.
- Error if mismatch or no entry.

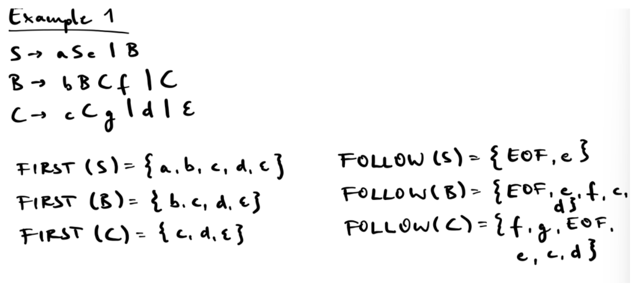
1.16 Bottom-up Parsing

- LR(k): Left-to-right, right-most derivation, k lookahead.
- Shift-reduce: Shift tokens, reduce groups to non-terminals.
- Table-based: Action table (shift/reduce/accept/error), Goto table (state after reduce).
- Stack: (symbol, state) pairs.
- Algorithm: Based on action[s, next]: shift push (token, s'), reduce pop m, push (N, goto[s', N]), accept/ error.
- Tables mostly auto generater

1.16.1 FOLLOW Sets

Set of all terminals that may follow A in some derivation. Including symbol EOF for *end of file*. **Never includes ϵ!**
Computing follow sets

- if A is start symbol but **EOF** in FOLLOW(A)
- Productions based on the form B $\rightarrow \alpha A \beta$ Add FIRST(β) without epsilon to FOLLOW(A)
- Productions of the form $B \rightarrow \alpha$ or B $\rightarrow \alpha A \beta$ where $\beta \rightarrow \epsilon$ Add FOLLOW(B) to FOLLOW(A)



Example 1

S → aSe | B
B → bBCf | C
C → cCg | d | ε

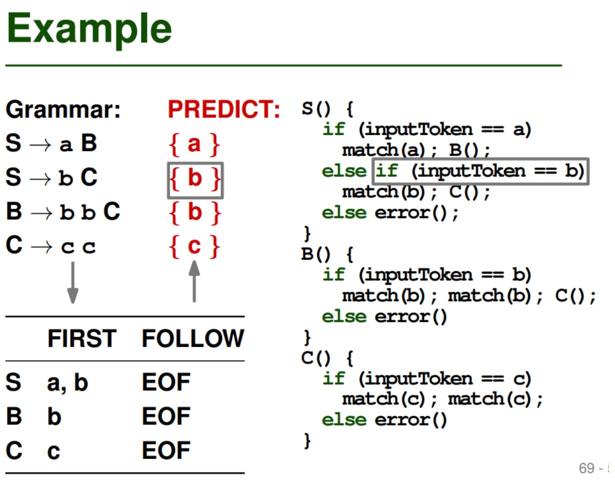
FIRST (S) = {a, b, c, d, ε}
FIRST (B) = {b, c, d, ε}
FIRST (C) = {c, d, ε}

FOLLOW (S) = {EOF, e}
FOLLOW(B) = {EOF, e, f, c, d}
FOLLOW(C) = {f, g, EOF, e, c, d}

Figure 2: Follow set example

1.16.2 PREDICT SETS

Rule: Which terminals to look for in LL(1) parser.
If next input token is in PREDICT of rule, apply the rule.
Computing the PREDICT set for rule $A \rightarrow \epsilon$:
If ϵ in FIRST(α) : PREDICT($A \rightarrow \alpha$) = FIRST (α) $- \epsilon$)$\cup$ FOLLOW(A)
Otherwise: PREDICT($A \rightarrow \alpha$ = FIRST(α).



Example

Grammar:	PREDICT:
S → a B	{ a }
S → b C	{ b }
B → b b C	{ b }
C → c c	{ c }

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

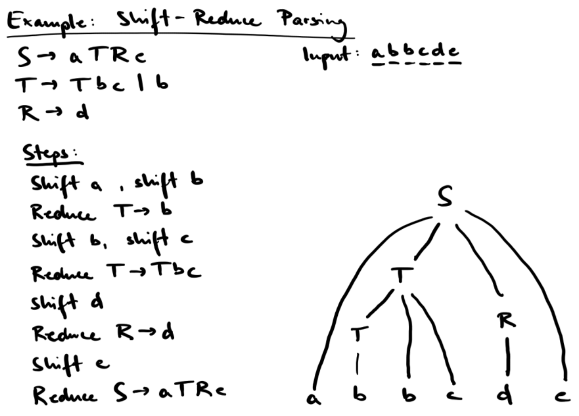
```
S() {
  if (inputToken == a)
    match(a); B();
  else if (inputToken == b)
    match(b); C();
  else error();
}
B() {
  if (inputToken == b)
    match(b); match(b); C();
  else error()
}
C() {
  if (inputToken == c)
    match(c); match(c);
  else error()
}
```

69 -

Figure 3: Predict example

1.17 Shift reduce Algorithm

Repeat until all tokens read and all symbols reduced to start symbol.
Shift(i.e read) input tokens.
Try to reduce a group of symbols into a single non-terminal.



Example: Shift - Reduce Parsing

S → aTRc
T → Tbc | b
R → d

Input: abbcde

Steps:
Shift a , shift b
Reduce T → b
Shift b, shift c
Reduce T → Tbc
Shift d
Reduce R → d
Shift e
Reduce S → aTRc

Figure 4: Examle shift reduce parsing

2 Names, Scopes, and Bindings

2.1 Binding

- between entities to their names. **For instance:** a variable bound to a memory object, a function bound to the code implementing the function.
- Static (compile-time) vs dynamic (run-time).

2.2 Scope

- Region where binding active; maximal no-change region.
- Nested scopes for subroutines.
- Static scoping: From text, up nested scopes.
- Dynamic scoping: From control flow, up call stack.
- Built-in objects: Invisible outer-most scope.

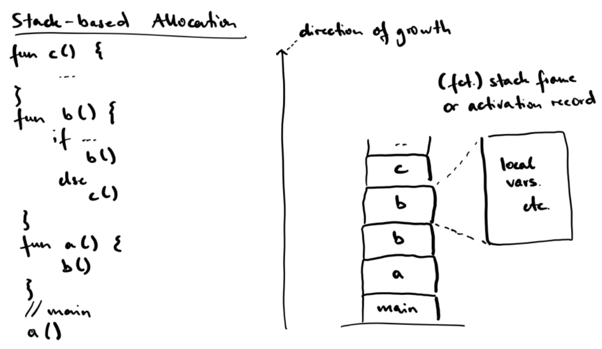
2.3 Object Lifetime and Storage

- Lifetime: Global (program), local (function), heap (arbitrary).
- Storage: Static (globals, constants), stack (locals, activation records), heap (dynamic, managed GC).

2.3.1 Statically allocated memory

Depending on the PL, used for: global variables, constant literals, symbol tables, programm code itself, compile times constants.

2.3.2 Stack based allocation



Stack-based Allocation

```
fun c() {
    ...
}
fun b() {
    if ...
        b()
    else
        c()
}
fun a() {
    b()
}
// main
a()
```

Figure 5: Stack based allocation

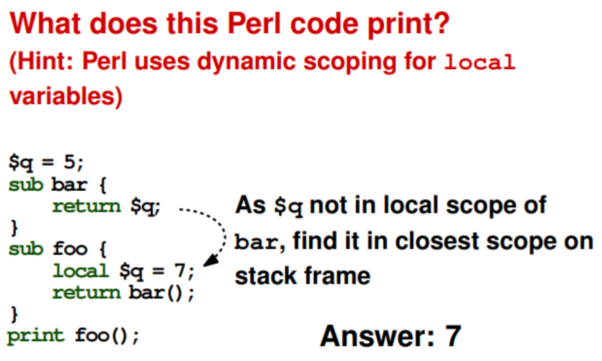
2.3.3 Heap based allocation

For **dynamically allocated date structures** and objects whose **size is statically unknown** (for exmaple objects in java)
Some PLs managed memory:
Unreachable objects: implicitly deallocated, unreachable = no active binding
Less control but fewer bugs e.g no use-after-free

2.3.4 Static vs dynamic Scoping

Static scoping: binding of a name can be derived from program text.
Move up nested scopes until finding a binding.

Dynamic scoping: binding of a name depends on control flow.
Move up functions on call stack until finding a binding.



What does this Perl code print?
(Hint: Perl uses dynamic scoping for `local` variables)

```
$q = 5;
sub bar {
    return $q;
}
sub foo {
    local $q = 7;
    return bar();
}
print foo();
```

As `$q` not in local scope of `bar`, find it in closest scope on stack frame

Answer: 7

Figure 6: Example on dynamic scoping, really just looking for ANY point where it has been declared in its scope, eve if after calling it

2.4 Aliasing and Overloading

- Aliasing: Multiple names to one object (pointers, references).
- Overloading: One name to multiple objects.

2.5 Referencing Environment

- Set of bindings.
- For function refs: Shallow binding (at call, dynamic scoping), deep binding (at creation, static scoping, closures).
- Closure: Function + captured environment.

2.5.1 Shallow binding

Referencing environment created when the function is called.
Common in languages with **dynamic scoping**

2.5.2 Deep binding

Referencing environment created when the reference to the function is created.
Common in languages with **static scoping**

2.5.3 Closure

Implementation of deep binding.
Is the representation of referencing environment + function itself?
When creating a reference to a function, closure is created.

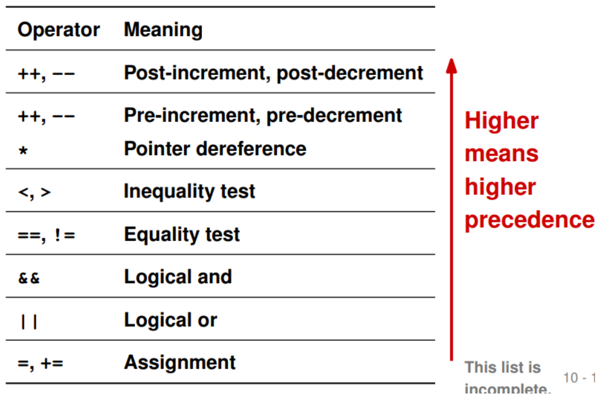
3 Control Flow

4 Control Flow

- Ordering of instructions: Fundamental to computation.
- Mechanisms: Sequencing, selection, iteration, recursion, concurrency, exceptions.
- Each PL has own rules, focus on concepts (or urself queen/king).

4.1 Expression Evaluation

- Operator (built-in function with simple syntax) vs operand (arguments of operator).
- Notations: Prefix (op a b), Infix (a op b), Postfix (a b op).
- Multiplicity (Number of arguments expected by an operator): Unary (a++) or (!cond), Binary (a + b), Ternary (cond ? a : b).
- Precedence: Order operators (e.g., * ¿ +).
- Associativity(decides on same level operators): Left-associative (a - b - c = (a - b) - c), Right- associative (a = b = c = a = (b = c)).
- Operand order: Left-to-right (Java), Undefined (C).
- Short-circuit: skips if first false, —— skips if first true.



Operator	Meaning
++, −−	Post-increment, post-decrement
++, −−	Pre-increment, pre-decrement
*	Pointer dereference
<, >	Inequality test
==, !=	Equality test
&&	Logical and
||	Logical or
=, +=	Assignment

Higher means higher precedence
This list is incomplete.
10 - 1

Figure 7: Precedence levels in C

4.2 Unstructured Control Flow

- **Gotos:** Jump to label, unstructured. Basically fuck gotos.
- **Continuations:** Generalize gotos, define new constructs (exceptions, iterators, coroutines).
- **Low-level:** Code address, referencing environment, another continuation.

4.3 Selection

- Branch on condition: if-else, case/switch.
- Case/switch: Compare expr to constants, many conditions that compare the same expression to different compile time constants.

4.4 Iteration

- **Enumeration-controlled:** Initial, bound, step (e.g., for i=1 to 10 by 2).
- **Logically controlled:** Pre (while), Post (do-while), Mid (loop with break).
- **Iterators:** True/generators (yield), Objects (hasNext/next), First-class functions (apply func to each).

4.5 Recursion

- Equivalent to iteration, your new bestie in functional languages)(because the recursive function typically doesn´t update any non-local variables)
- Efficiency: Stack frames; tail recursion optimizes (last call reuses frame).

5 Types

5.0.1 Why do we need types?

1. Provide context for operations (for example a+b could be concatenation of strings or addition.)
2. Limit valid operations. Helping to find bugs early.
3. Code readability and understandability. But sometimes type can make code harder to write.
4. Compile time optimazation. Compiler knows which behavior is possible and which isn´t

5.1 Composite Types

Composite types are formed by combining simpler types using type constructors. Common composite types include records, arrays, strings, sets, pointers, and lists.

5.1.1 Records

Records (also structs or structures) store related data of **heterogeneous types** together. Each data component is called a field.

Most programming languages offer some record-like type constructor:

- C: structs
- C++: special form of class
- Fortran 90: simple "types"
- C#, Swift: struct types
- OCaml: tuples (order irrelevant)
- Java: records (immutable fields since Java 14)

5.1.2 Memory Layout

Records are typically stored with fields in adjacent memory locations. Field access uses address + offset calculations. Alignment constraints may create "holes" in memory layout depending on architecture requirements.

5.1.3 Optimization Strategies

- **Packing**: Avoid holes but break alignment (requires additional instructions)
- **Reordering**: Minimize holes while respecting alignment constraints

Note: **C and C++ don't reorder fields by default** as system-level programmers may rely on specific memory layouts.

5.1.4 Nested Records

Records can be nested either lexically (anonymous inner records) or through fields of record type.

5.1.5 Semantics: Reference vs. Value Model

- **Reference model**: Variable refers to memory location
- **Value model**: Variable contains the actual value
- C uses reference model for LHS assignments, value model otherwise
- Java uses value model only for built-in types

5.1.6 Variant Records (Unions)

Unions reuse the same memory location for multiple variables, assuming they're never used simultaneously. Size equals the largest member.

5.2 Arrays

Arrays are the most common composite data type, conceptually representing a mapping from index type to element type.

5.2.1 Multi-Dimensional Arrays

Arrays can have multiple dimensions (2D matrices, 3D matrices, etc.). Different languages use different indexing conventions (C: row-major, Fortran: column-major).

5.2.2 Array Operations

- **Slicing**: Extracting rectangular portions of arrays
- **Comparison**: Element-wise comparison of equal-length arrays
- **Mathematical operations**: Element-wise addition, subtraction, etc.

5.2.3 Memory Layout Significance

Memory layout determines efficiency of nested loops through multi-dimensional arrays due to CPU cache behavior. Row-major vs. column-major layouts affect which access patterns are most efficient.

6 Type Systems

Type systems define types and their association with programming language constructs, including rules for type equivalence, compatibility, and inference.

6.1 Why Types Matter

- Provide context for operations (e.g., addition vs. concatenation)
- Limit invalid operations to catch bugs early
- Improve code readability and documentation
- Enable compile-time optimizations

6.2 Type Checking Spectrum

- **Strongly typed**: Operations only on proper types (most languages since 1970s)
- **Statically typed**: Mostly checked at compile-time
- **Dynamically typed**: Checking delayed until runtime
- **Gradual typing**: Optional type annotations with partial static checking

6.3 Polymorphism

- **Parametric polymorphism**: Code takes types as parameters (generics)
- **Subtype polymorphism**: Extending/refining supertypes (subclasses)
- **Polymorphic variables**: Variables that can reference different types

6.4 Type Equivalence

- **Structural equivalence**: Same structure = same type
- **Name (nominal) equivalence**: Same name = same type

Each approach has limitations: structural equivalence cannot distinguish differently named but structurally identical types, while name equivalence struggles with type aliases.

7 Data Abstraction

Data abstraction describes classes of memory objects and their associated behavior through abstract data types (sets of values and operations).

7.0.1 Subclasses vs. Subtypes

- Subclassing: About reusing code inside a class
- Subtyping: Enables code reuse in clients of a class

7.0.2 Liskov's Substitutability Principle

Each subtype should behave like the supertype when used through the supertype interface. Objects of type A should be replaceable by objects of type B without affecting clients.

7.0.3 Modifying Inherited Members

- Java: Any method can be overridden
- C++: Only virtual methods can be overridden
- Java/C#: Cannot change visibility of inherited members
- Eiffel: Can restrict or increase visibility
- C++: Public/protected/private inheritance affects member visibility for clients

7.1 Initialization and Finalization

7.1.1 Constructors

- Distinguished by number/type of arguments (C++, Java, C#) or name (Eiffel)
- Java: Constructors must always be called explicitly
- C++: Constructors sometimes called implicitly (value model)
- Superclass constructors execute before subclass constructors

7.1.2 Destructors and Finalization

- C++: Destructors called when object goes out of scope or deleted
- Execution order: Subclass destructor before superclass destructors (reverse of constructors)
- Java/C#: Finalizers (deprecated/removed in modern versions) called before garbage collection

7.2 Dynamic Method Binding

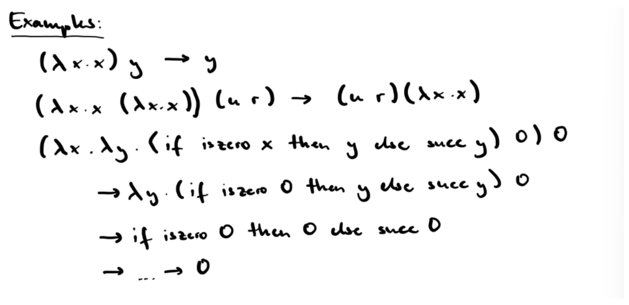
7.2.1 Static vs. Dynamic Binding

- Static binding: Based on variable type (compile-time resolution)
- Dynamic binding: Based on object type (runtime resolution)
- Dynamic binding allows subclasses to control their state but has performance overhead

7.2.2 Language Support

- Dynamic binding by default: Smalltalk, Python, Ruby
- Dynamic default but can mark as non-overridable: Java, Eiffel
- Static default but can specify dynamic: C++, C#
- Virtual method tables (vtables) commonly implement dynamic binding

8 Lambda Calculus


Examples:

$(\lambda x.x)\ y \to y$

$(\lambda x.x\ (\lambda x.x))\ (u\ r) \to (u\ r)(\lambda x.x)$

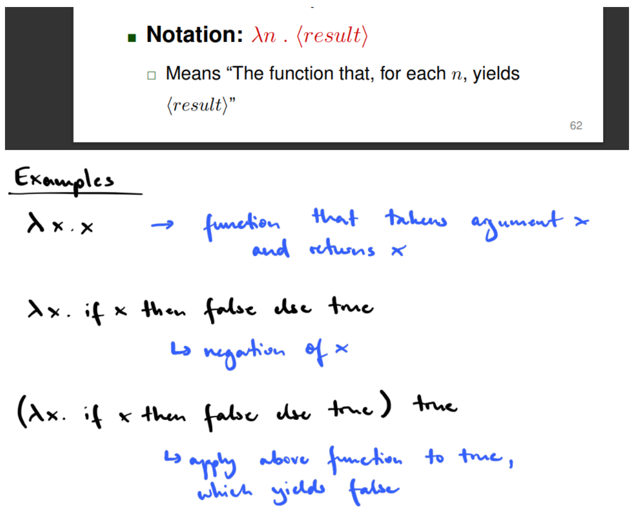
$(\lambda x.\lambda y.(\text{if iszero } x \text{ then } y \text{ else succ } y)\ 0)\ 0$

$\to \lambda y.(\text{if iszero } 0 \text{ then } y \text{ else succ } y)\ 0$

$\to \text{if iszero } 0 \text{ then } 0 \text{ else succ } 0$

$\to \ldots \to 0$

Figure 8: Example


- **Notation:** $\lambda n\ .\ \langle result \rangle$
  - Means "The function that, for each n, yields $\langle result \rangle$"

62

Examples

$\lambda x.x$ → function that takes argument x and returns x

$\lambda x.$ if x then false else true

↳ negation of x

($\lambda x.$ if x then false else true) true

↳ apply above function to true, which yields false

Figure 9: Exapmle2

9 Control Abstraction

Control abstraction focuses on abstracting well-defined operations (subroutines, exception handlers) rather than information representation.

9.1 Calling Sequences

Low-level code executed to maintain the call stack, including:

- Parameter and return value passing
- return address saving
- Register saving/restoring
- Stack and frame pointer updates

9.1.1 Stack Layout

Each procedure call creates a stack frame (activation record) with:

- Frame pointer: Base address for accessing current frame data
- Stack pointer: First unused/last used location in current frame

9.1.2 Stack Smashing

Buffer overflow vulnerability where malicious input overwrites return addresses, enabling execution of arbitrary code. lets goooooooo

9.2 Coroutines

Control abstraction that allows suspending and resuming execution, implementing non-preemptive multitasking.

9.2.1 Characteristics

- Explicit transfer of control (non-preemptive)
- Only one coroutine runs at a time
- Represented by closures (code address + referencing environment)

9.2.2 Comparison vs. Threads:

- Threads transfer control implicitly and preemptively, multiple threads may run concurrently
- vs. Continuations: Coroutines change program counter when running, continuations remain fixed

9.2.3 Language Support

- Native support: Ruby, Go
- Library support: Java, C#, JavaScript, Kotlin
- Specialized variants: Python generators

9.3 Promises, Async, and Await

9.3.1 Motivation

Parts of programs may take very long (I/O operations), requiring mechanisms to continue execution without blocking.

9.3.2 Approaches

- Event-driven programming: Register callbacks
- Promises/futures: Objects representing not-yet-computed values
- Async/await: Syntactic sugar for promise-based programming

9.3.3 Benefits over Event-Driven Code

- Cleaner control flow
- Explicit synchronization (e.g., Promise.all)
- Centralized error handling
- Async/await provides sequential-looking syntax without blocking

10 Composite Types Part 2 (yes this is exactly where it is supposed to be)

10.1 Pointers and Recursive Types

- Pointers: References to memory locations (addresses)
- Recursive types: Composite types with references to objects of the same type
- Reference model languages: No explicit pointers needed
- Value model languages: Require explicit pointers to avoid copying

10.2 Operations on Pointers

- Creation: Constructor calls, allocation functions, address-of operators
- Allocation: Implicit (OCaml, Java) or explicit (C)
- Dereferencing: Accessing referred memory objects
- Deallocation: Explicit (C, C++, Rust) or garbage collection (Java, C#, Python)

10.3 Pointers and Arrays in C

- Closely related constructs
- Array access equivalent to pointer arithmetic: E1[E2] *(E1 + E2)
- Pointer arithmetic includes subtraction and comparison, scaled by type size
- Key difference: Arrays implicitly allocated, pointers require explicit allocation

10.4 Dangling References and Garbage Collection

10.4.1 Dangling References

- Live pointers to invalid objects
- Caused by: Pointers to stack objects escaping scope, heap objects deallocated with living pointers
- Dereferencing behavior is undefined

10.4.2 Garbage Collection

- Automatic memory deallocation managed by language implementation
- Common in managed languages (Java, Python, JavaScript)
- Prevents dangling references and memory leaks

10.4.3 Implementation Approaches

- Reference counting: Count pointers to each object, deallocate when count reaches zero
- Problem: Circular dependencies prevent deallocation
- Mark and sweep: Identify useless blocks by marking reachable objects from external references
- Optimizations: Pointer reversal, stop-and-copy, generational garbage collection

11 Concurrency

11.1 Motivation

- Capturing logical structure of inherently concurrent problems
- Exploiting parallel hardware for performance (multi-core processors since 2005)
- Coping with physical distribution of systems

Key Terminology:

- **Concurrent**: Tasks whose execution may be at unpredictable points
- **Parallel**: Tasks actively executing simultaneously (requires multiple cores)
- **Distributed**: Tasks on physically separated processors

Levels of Parallelism:

- Circuit/gate level (hardware handled)
- Instruction-level (hardware handled)
- Vector parallelism (programmer specified)
- Thread-level (programmer specified)

11.2 Data Races and Correctness

Data Race Definition: Two accesses to the same shared memory location where at least one is a write and ordering is non-deterministic.

Data races lead to unpredictable program behavior and must be avoided through proper synchronization.

11.3 Communication Models

- **Shared Memory**: Variables accessible by multiple threads (Java, C#, C/C++)
- **Message Passing**: No shared state, threads communicate via messages (Erlang, Go)

11.4 Thread Creation and Management

Process: (OS LEVEL) OS construct that may execute threads.
Thread (PL LEVEL) Active entity that the programmer thinks of as running concurrently with other threads.
Task: (LOGICAL LEVEL) Unit of work that must be performed by some thread.

Co-begin: Compound statement where all statements are executed **concurrently.**
Parallel Loops: Loop iterations execute concurrently instead of sequentially

- Can specify data sharing: shared, private, or reduction variables
- Examples: OpenMP in C, Task Parallel Library in C#

Fork/Join: Explicit thread creation (fork) and termination waiting (join)
Thread Pools: Separate tasks from thread execution

- Reuse threads to reduce creation overhead
- Pool implementation handles scheduling

POSIX Threads (pthreads): Low-level API for thread creation and management
Message Passing: Independent threads (actors, goroutines) or exchange messages via channels to avoid pitfalls of shared memory(race conditions) and need for low-level concurrency mechanisms.

11.5 CILK

spawn: calls a fucntion to be executed as a logically concurrent task. Fang einfach an mit parallel
sync: joins all tasks spawned by the calling task. Warte bis alle gespawnten fertig sind.

11.6 Synchronization

Purpose: Control relative order of operations in different threads
Explicit in **shared memory model.**
Implicit in message-passing model.

Two Forms:

- **Spinning/Busy-waiting**: Thread repeatedly checks condition until true
- **Blocking**: Thread stops/waits until condition becomes true, scheduler reactivates

Two Goals:

- **Mutual Exclusion**: Make computations atomic (only one thread in critical section)
- **Condition Synchronization**: Delay operations until preconditions hold

Trade-off: Synchronization ensures correctness but reduces parallelism

11.7 Synchronization Mechanisms

Spin Locks:

- Provide mutual exclusion using special hardware instructions
- **Test-and-Set**: Atomically set boolean and return previous value. Problem: repeated writes when lokc is already acquired harms performance ("contention").
  9
- **Test-and-Test-and-Set**: Avoid contention by reading before writing

Barriers: Ensure all threads finish one phase before entering next

- Implementation uses atomic fetch-and-decrement
- Shared counter and boolean flag coordination

Monitors: Objects with operations, internal state, and condition variables

- Only one operation active at a time

- Operations can wait on or signal condition variables
- Java: Every object can serve as monitor via `synchronized`

Transactional Memory: Atomicity without explicit locks

- Speculatively execute code blocks
- Check for conflicts and commit or rollback

11.8 Language Support for Synchronization

Implicit Synchronization in Parallel Loops: Some languages (e.g., Fortran 95) provide automatic synchronization on assignments. **All reads on right hand side are before writes on left hand side**
Implicit Synchronization: Compiler determines dependencies and adds synchronization automatically (extremely difficult in practice)

11.8.1 Monitors

Object with operations, internal state and condition variables.
Only one operation is active at goven time

11.9 Memory Consistency

Sequential Consistency: Most programmer expectation

- Each thread's instructions execute in specified order
- Shared memory behaves like global array with immediate reads/writes

Relaxed Memory Models: Reads and writes may occur out of order

- Hardware and compilers reorder instructions for efficiency
- Different hardware has different reordering behavior

Programming Language Memory Models: PLs define their own consistency guarantees

- **Java Memory Model**: Writes not immediately visible without synchronization. Force explicit synchronization.
- **Volatile fields**: Ensure visibility across threads
- **Synchronized blocks**: Order reads and writes

12 Functional Languages

12.1 Introduction to Functional Programming

Functional programming represents an alternative to imperative programming languages where:

- Output is a mathematical function of input (often Lamndas)
- No internal state or side effects (in pure functional languages)
- In practice, boundaries are fuzzy - many "imperative" languages have functional features and vice versa

12.2 Key Features of Functional Languages

- **First-class functions**: Functions can be assigned to variables, passed as arguments, or used as return values
- **Extensive polymorphism**: Functions work on different types of values through type inference
- **List types and operators**: Ideal for recursion (handle first element, then recursively process remainder)
- **Structured function returns**: Functions can return any structured data including lists and functions
- **Constructors**: Build aggregate objects inline and all-at-once
- **Garbage collection**: Necessary due to frequent creation of temporary data

12.3 Types of Functional Languages

Purely Functional Languages

- Functions depend only on their parameters
- No global or local state dependencies
- Order of evaluation is irrelevant (eager and lazy evaluation yield same results)
- Example: Haskell

Non-Pure Functional Languages

- Mix functional features with assignments
- Examples: Scheme (Lisp dialect), OCaml (ML with OO features)

12.4 Scheme Language Constructs

- **Function Application**: Parentheses denote function calls, first expression is the function, remaining are arguments
- **Lambda Expressions**: Create anonymous functions with formal parameters and function body
- **Variable Bindings**: `let` binds names to values with specific scoping rules
- **Conditionals**: `if` for simple conditions, `cond` for multiway conditionals. second/third argument is vlaue returned if true/false
- **Dynamic Typing**: Types determined and checked at runtime, enabling implicit polymorphism
- **List Operations**: **car** (first element), **cdr** (rest), **cons** (join head to tail)
- **Side Effects**: **set!** for variable assignment, **set-car!** assigning head of list and **set-cdr!** for assigning tail of list

12.5 Evaluation Order

Two primary evaluation strategies:

- **Applicative-order**: Evaluate arguments before passing to function Meistens von rechts nach links schonmal alles in Klammern ausrechnen.
- **Normal-order**: Pass arguments unevaluated, evaluate when needed

Evaluation order affects both performance and correctness. Lazy evaluation evaluates subexpressions on-demand and memoizes results, transparent to programmers only in pure functional languages.

C Memory Management Cheat Sheet

Memory Layout (typical C program)

Heap: dynamic memory (controlled by `malloc`, `calloc`, `realloc`, `free`).

Stack: function calls, local variables, return addresses.

Key Functions

- `void* malloc(size_t size)`
  - Allocates a block of `size` bytes on the **heap**.
  - Contents are **uninitialized** (may contain garbage values).
  - Returns a `void*` pointer (NULL if allocation fails).
- `void* calloc(size_t n, size_t size)`
  - Allocates space for `n` elements of given `size`.
  - Memory is set to **zero**.
  - Returns a pointer to heap memory.
- `void* realloc(void* ptr, size_t new_size)`
  - Changes size of memory block pointed by `ptr` to `new_size`.
  - May move block to a new location in the **heap**.
  - Contents are preserved (up to the minimum of old and new size).
- `void free(void* ptr)`
  - Deallocates (releases) memory previously allocated on the heap.
  - Does not change `ptr` itself (dangling pointer if reused).

Notes

- Always check if allocation succeeded (`if (p == NULL) ...`).
- Memory leaks occur if allocated memory is not freed.
- Accessing freed memory = **undefined behavior**.