

Programmiersprache für KIs

Aktuell alle Programmiersprachen für Menschen entwickelt. Hättest du eine Programmiersprache für KIs entwickeln? Welche Prinzipien würdest du zur Grunde setzen? Wie stark würde deine Programmiersprache und aktuelle unterscheiden? Oder würdest du alles auf Assembler erledugen?

Heute sind Programmiersprachen zwar „für Menschen“ gestaltet (Lesbarkeit, Wartbarkeit, Tooling), aber sie werden **auch** stark für **Compiler/Optimierer/VMs** entworfen – also indirekt schon für Maschinen. Für „KIs“ im Sinne von *LLMs als Programmierer/Agenten* würde ich trotzdem eine eigene Sprache/Umgebung bauen – aber nicht, weil KI Assembler „braucht“, sondern weil KI andere Superkräfte und andere Schwächen hat als Menschen.

Würde ich eine Programmiersprache für KIs entwickeln?

Ja – eher als **KI-first Language + Runtime**: eine Sprache, deren Kernidee ist, dass **ein Modell sicher planen, ausführen, prüfen und korrigieren** kann, mit starken Leitplanken. Mehr „orchestrieren + verifizieren“ statt „alles von Hand ausprogrammieren“.

Prinzipien, die ich zugrunde legen würde

1) Spezifikation vor Implementierung (Design by Contract, aber zentral)

- Jede Funktion hat **Vorbedingungen, Nachbedingungen, Invarianten**
- „Was muss gelten?“ ist genauso wichtig wie „wie wird's gemacht?“
KIs sind gut im Erzeugen von Code, aber noch besser werden sie, wenn sie **einen formalen Zielrahmen** haben, den sie automatisch testen/prüfen können.

2) Determinismus und Reproduzierbarkeit als Default

- Reproduzierbare Builds, feste Seeds, deklarierte Abhängigkeiten
- Side-Effects nur in expliziten Blöcken („effect scopes“)
KIs halluzinieren/variierten; die Sprache sollte Variation **eindämmen**.

3) Eingebaute Verifikation: Typen + Property-Tests + Modellchecking (wo möglich)

- Starke Typen (auch für Datenformate/Units/Constraints)
- Property-based Testing als Standard
- Optional: formale Beweise / SMT-Solver-Hooks
Damit kann eine KI nicht nur „Code schreiben“, sondern **belegen**, dass er plausibel korrekt ist.

4) Capability-basierte Security (Sicherheit per Sprachdesign)

- Code bekommt **Capabilities** (Dateisystem, Netzwerk, Uhrzeit, Prozessstart) nur explizit

- Sandboxing als Default, sichere Standardbibliothek
Agenten/KIs müssen „dürfen“ klar vom „können“ trennen.

5) Explizite Datenflüsse & Provenienz

- Jede Datenquelle ist getaggt (z. B. `user_input`, `external_api`, `trusted_db`)
- Sensitive sinks (Logs, Netzwerk, Persistenz) prüfen diese Tags
Das reduziert Datenlecks und macht Audits leichter.

6) Fehlertoleranz: „Recovery-first“ statt nur Exceptions

- Statt nur `try/catch`: eingebaute Strategien wie `retry`, `fallback`, `compensate`
- Transaktionen/Idempotenz als Sprachkonstrukt
KI-Agenten arbeiten oft in unsicheren Umgebungen; Fehlerbehandlung ist nicht „Randfall“, sondern Alltag.

7) Modul- und Tooling-Design für KI: „Editability“

- AST-first, stabile Formatierung, kein „Style-Chaos“
- Jede Änderung ist als Patch nachvollziehbar
LLMs sind besser im **gezielten Umformen** als im „Freestyle im Text“.

8) Native „Plan/Execute/Observe“-Struktur

- Erst Plan als deklaratives Artefakt, dann Ausführungsschritte
- Logs/Beobachtungen sind Typen, nicht Strings
Das macht Agenten verlässlicher als „ein Skript, das alles macht“.

Wie stark würde sich so eine Sprache von aktuellen unterscheiden?

Vom Look & Feel vielleicht **weniger**, als man denkt (sie könnte „modern“ wirken wie Rust/Swift/TypeScript). Der große Unterschied wäre:

- **Mehr deklarativ + constraints** (Ziele, Regeln, Invarianten)
- **Weniger implizite Nebenwirkungen**
- **Eingebaute Verifikation, Security, Reproducibility**
- **Ausführung als kontrollierter Workflow**, nicht als „lauf einfach“

Also: nicht „eine KI schreibt Assembler“, sondern „eine KI schreibt Spezifikation + überprüfbaren Code, der sicher ausgeführt wird“.

Würde ich alles auf Assembler erledigen?

Nein. Assembler ist aus KI-Sicht meistens das Gegenteil von optimal:

- extrem lang, fehleranfällig, wenig semantische Struktur
- schwer zu verifizieren, schwer zu refactoren
- moderne Compiler optimieren besser, als ein LLM „per Hand“ stabil nachbauen würde

Was ich **stattdessen** machen würde:

- High-Level Code schreiben lassen
- dann **Compiler/IR** nutzen (LLVM-IR o.ä.) für Optimierung
- und **Verifier/Tests** drüber laufen lassen

Assembler wäre nur für:

- winzige Hotspots
- Embedded/Kernel/Hard-Realtime-Spezialfälle
- Reverse Engineering / sehr spezielle Optimierungen

Kurzbild der „KI-Sprache“

Wenn ich sie benennen müsste: **“SpecLang” oder “Guardrail”**:

- Programmiert „Was“ + „Beweise/Checks“ + „Erlaubnisse“
- Runtime führt nur aus, was innerhalb der Leitplanken liegt
- Tooling unterstützt automatische Korrektur-Schleifen

Wenn du willst, kann ich dir als Nächstes ein kleines Beispiel zeigen, wie so eine Syntax aussehen könnte (z. B. Funktion mit Vor-/Nachbedingungen, Capability-Scopes und Property-Tests) – einmal „menschlich“ und einmal „KI-first“.

gern!

Cool — hier sind zwei Mini-Beispiele: einmal „normal menschlich“, einmal „KI-first“ (SpecLang/Guardrail-Stil). Gleiche Aufgabe: **JSON mit Nutzern einlesen, E-Mails normalisieren, Duplikate entfernen, nur valide E-Mails ausgeben.**

1) „Menschlich“ (typisch TypeScript/Python-Stil)

Wenig formale Zusagen, Validierung und Fehlerbehandlung sind „per Hand“:

```

type User = { id: string; email?: string };

function normalizeEmail(e: string): string {
  return e.trim().toLowerCase();
}

function isValidEmail(e: string): boolean {
  return /^[^s@]+@[^s@]+\.[^s@]+$/ .test(e);
}

export function extractUniqueEmails(users: User[]): string[] {
  const seen = new Set<string>();
  const out: string[] = [];

  for (const u of users) {
    if (!u.email) continue;
    const e = normalizeEmail(u.email);
    if (!isValidEmail(e)) continue;
    if (seen.has(e)) continue;
    seen.add(e);
    out.push(e);
  }
  return out;
}

```

Das ist okay, aber:

- Nebenwirkungen/Regeln sind implizit.
- Kein maschinenlesbarer Vertrag („was garantiert die Funktion?“).
- Security/Provenienz nicht modelliert.

2) „KI-first“ (SpecLang/Guardrail - erfundene Syntax)

Hier ist die Idee: **Vertrag + Effekte + Datenherkunft + Tests** sind Teil der Sprache.

```

// Datentypen mit Constraints
type Email = String where matches("^[^\\s@]+@[^\\s@]+\\. [^\\s@]+$");
type UserId = String where len(1..64);

record User {
    id: UserId
    email?: String
}

// Capabilities: standardmäßig KEIN Zugriff auf IO/Netz/Clock.
capability none;

// Pure function: keine Effekte erlaubt.
fn normalize_email(raw: String) -> String
    effects none
    ensures result == lower(trim(raw));

fn parse_email(raw: String) -> Option<Email>
    effects none
    ensures result.is_some() => matches(result.unwrap(), "^[^\\s@]+@[^\\s@]+\\. [^\\s@]+$");

fn unique_emails(users: List<User>) -> List<Email>
    effects none
    ensures is_unique(result)           // keine Duplikate
    ensures forall e in result => is_valid(e)   // alles valide
    ensures forall e in result => exists u in users
        where parse_email(normalize_email(u.email? or "")) == Some(e)
{
    let seen: Set<Email> = {};
    let out: List<Email> = [];

    for u in users {
        match u.email {
            None => continue,
            Some(raw) => {
                let norm = normalize_email(raw);
                match parse_email(norm) {
                    None => continue,
                    Some(e) => if not seen.contains(e) {
                        seen.add(e);
                        out.push(e);
                    }
                }
            }
        }
    }
    return out;
}

// Eingebaute property-based tests (QuickCheck-Stil)
property "unique_emails outputs valid unique emails" {
    forall users in gen.list(gen.user()) {
        let out = unique_emails(users);
        assert is_unique(out);
        assert forall e in out => is_valid(e);
    }
}

```

Was daran „KI-first“ ist:

- **ensures** macht das Ziel formal: Eine KI kann beim Refactor/Optimieren nicht „aus Versehen“ Regeln brechen.
- **effects none** verhindert heimliche IO/Netz-Aufrufe.
- **Email** ist ein **Constraint-Typ** – Fehler wandern nach „vorne“ ins Typsystem.
- Property-Tests sind Standard, nicht Add-on.

Bonus: Agenten-Workflow als Sprachelement (Plan → Execute → Observe)

Wenn die Aufgabe „API abfragen und speichern“ wäre, würde ich das so modellieren:

```
capability net("api.example.com");
capability fs("/data/out");

workflow fetch_and_store {
    step plan {
        // deklarativ: was wollen wir erreichen?
        goal: "Fetch users, store normalized unique emails"
        constraints: [
            "no PII in logs",
            "retry up to 3 on transient network errors",
            "write output atomically"
        ]
    }

    step fetch effects net {
        let users_json = http.get("/users");
        output users_json;
    }

    step transform effects none {
        let users = decode_json<List<User>>(fetch.output);
        let emails = unique_emails(users);
        output encode_json(emails);
    }

    step store effects fs {
        fs.write_atomic("/data/out/emails.json", transform.output);
    }
}
```

Hier kann eine KI:

- Schritte isoliert testen
- nur mit den benötigten Capabilities arbeiten
- Logs/PII-Regeln automatisch erzwingen

Wenn du magst, mache ich als nächstes eins davon:

1. Ein kleines “SpecLang”-Mini-Design (10–15 Kernfeatures + Syntax-Regeln), oder
2. Ein größeres Beispiel (z. B. „Webcrawler mit Guardrails“, „ETL-Pipeline“, „Agent der Tickets triagiert“), oder
3. Mapping: Wie würde man so was heute mit Rust/TS/Python + Tools (Types, contracts, sandboxing) nachbauen?