L-OPS

Logical Operator Protocol System

Offizielle Sprachspezifikation

Version 1.0

Contents

1 Vorwort

Zweck dieses Dokuments

Dieses Dokument definiert die offene und frei zugängliche Sprachspezifikation **L-OPS** (Logical Operator Protocol System). L-OPS ist eine deklarative Metasprache zur strukturierten Beschreibung von Abläufen, Entscheidungsstrukturen und kontrollierter Ausführung in komplexen Systemen.

Ziel dieses Dokuments ist es:

- die Sprache vollständig und konsistent zu beschreiben,
- ihre Nutzung dauerhaft und unwiderruflich der Öffentlichkeit bereitzustellen,

- eine stabile Grundlage für Implementierungen zu schaffen, ohne selbst eine technische Implementierung zu enthalten.

Diese Spezifikation umfasst ausschließlich die Struktur, Syntax und Regeln der Sprache. Technische Verfahren, interne Mechanismen oder Auswertungslogiken sind ausdrücklich nicht Bestandteil dieses Dokuments.

1.1 Was L-OPS ist

L-OPS ist eine kontrollierende, deklarative Sprache, die festlegt:

- *was* ein System ausführen soll,
- *in welcher Reihenfolge* Ausführungsschritte erfolgen,
- *unter welchen Bedingungen* Schritte aktiviert werden,
- *welche Bewertungsebenen* (Layer) während der Auswertung aktiv sind,
- *welche Entscheidungen* ein Ablauf erzeugt.

Die Sprache ist deterministisch, transparent und modular aufgebaut und stellt eine formale Beschreibungsebene über beliebigen technischen oder probabilistischen Systemen dar.

1.2 Was L-OPS nicht ist

L-OPS ist nicht:

- eine Programmiersprache zur numerischen oder algorithmischen Berechnung,
- eine technische Evaluations- oder Entscheidungsmaschine,
- eine Risiko-, Konflikt- oder Optimierungslogik,
- eine architekturelle Lösung zur Modellsteuerung,
- eine Sammlung technischer Verfahren oder Algorithmen.

Diese Spezifikation beschreibt ausschließlich die deklarative Ausdrucksform der Sprache. Die technischen Verfahren zur Auswertung oder Umsetzung bleiben vollständig außerhalb dieses Dokuments und können separat patentiert werden.

2 Wesentliche Elemente von L-OPS

Dieses Kapitel beschreibt die grundlegenden Bausteine der Sprache L-OPS. Diese Elemente bilden die deklarative Struktur, aus der vollständige Abläufe, Entscheidungen und Programme zusammengesetzt werden. Alle beschriebenen Komponenten sind sprachspezifisch; ihre technische Umsetzung liegt außerhalb des Scopes dieser Spezifikation.

2.1 PROFILE

Ein `PROFILE` definiert einen Betriebsmodus der Sprache. Es legt verbindliche Rahmenbedingungen fest, die für alle ihm zugeordneten BLOCKs gelten.

Typische Inhalte eines PROFILE sind:

- Standard-LAYER, die automatisch aktiv sind,
- generelle Einschränkungen oder Modusdefinitionen,
- Richtlinien für Entscheidungen oder Übergänge,
- globale Einstellungen für die Ausführung eines PROGRAMS.

Ein PROFILE enthält keine technischen Verfahren, sondern ausschließlich deklarative Vorgaben.

2.2 BLOCK

Ein `BLOCK` ist die grundlegende Verarbeitungseinheit in L-OPS. Jeder BLOCK beschreibt:

- einen Operator,
- Eingaben und Ausgaben,
- optionale Bedingungen,
- zulässige Übergänge zu anderen BLOCKs.

BLOCKs definieren einen Schritt im Ablauf, jedoch niemals die interne technische Auswertung des Schritts.

2.3 LAYER

Ein `LAYER` bezeichnet eine evaluative Ebene, die während der Ausführung eines BLOCKs aktiviert werden kann. Die Sprache definiert neun sichtbare Ebenen (E1–E9), die verschiedene Perspektiven der Bewertung repräsentieren.

LAYER regeln:

- welche Aspekte eines BLOCKs betrachtet werden,
- wie die Struktur eines BLOCKs interpretiert wird,
- welche deklarativen Rahmenbedingungen aktiv sind.

Die Ebenen selbst sind abstrakt und beschreiben keinerlei technische Bewertungsmethoden.

2.4 PROGRAM

Ein `PROGRAM` definiert:

- den Einstiegspunkt des Ablaufs,
- das aktive PROFILE,
- den globalen Rahmen einer Ausführung.

Es verbindet BLOCKs zu einem vollständigen, deterministischen Ablauf.

2.5 STATE

Der `STATE` repräsentiert während eines PROGRAM-Laufs alle deklarativ erzeugten Werte. Der STATE darf ausschließlich über explizite `outputs` eines BLOCKs verändert werden.

Merkmale:

- deterministische Fortschreibung,
- keine verdeckten Änderungen,
- vollständige Sichtbarkeit aller erzeugten Werte.

2.6 CONTEXT

Der `CONTEXT` repräsentiert alle externen Eingaben eines PROGRAMS und ist unveränderlich. Er dient als Ausgangspunkt für BLOCK-Verarbeitung, ohne selbst modifiziert zu werden.

Typische Inhalte:

- Roh-Eingabe,
- erkannte Ziele oder Objekte,
- grundlegende externe Informationen.

CONTEXT ist strikt read-only und beeinflusst die Auswertung ausschließlich über deklarative Nutzung in BLOCKs.

3 Syntax – Form und Grammatik

Dieses Kapitel beschreibt die formale Struktur der Sprache L-OPS. Alle Syntaxelemente sind deklarativ definiert und geben ausschließlich die Form der Sprache vor, nicht deren technische Auswertung. Die Syntax soll für Menschen wie Maschinen klar lesbar sein und eine deterministische, eindeutige Interpretation ermöglichen.

3.1 Grundstruktur von Sprachelementen

Die allgemeine Form eines Sprachelements in L-OPS folgt einem einheitlichen Blockmuster:

```
KEYWORD NAME {
    parameter = value
    parameter2 = value
    ...
}
```

Das Schlüsselwort (`KEYWORD`) bestimmt den Typ des Elements. Die zulässigen Schlüsselwörter lauten:

- `PROFILE`
- `BLOCK`
- `LAYER`
- `PROGRAM`

3.2 Parameter und Zuweisungen

Parameter innerhalb eines Sprachelements besitzen die Form:

```
parameter = value
```

Alle Parameter müssen explizit angegeben werden. Es gibt keine impliziten Werte oder automatischen Ergänzungen durch die Sprache selbst.

Parameterbezeichner folgen einfachen Regeln:

- Buchstaben (A–Z, a–z),
- Ziffern (0–9),
- Unterstriche (_).

Ein Parameter darf nicht mit einer Ziffer beginnen.

3.3 Werte-Typen

L-OPS verwendet eine kleine Menge deklarativer Wertetypen:

- **STRING**: Zeichenketten ohne Formatvorgaben.
- **NUMBER**: numerische Werte (Ganzzahl oder Dezimalzahl).
- **BOOL**: `true` oder `false`.
- **ENUM**: definierte Auswahl aus endlichen Begriffen.

- **LIST**: geordnete Auflistung von Werten in eckigen Klammern.

Beispiele:

```
name = "example"
count = 4
active = true
mode = STRICT
layers = [E1, E2, E9]
```

3.4 Kommentarregeln

L-OPS erlaubt zwei Formen von Kommentaren:

- `#` für einzeilige Kommentare
- `/* ... */` für mehrzeilige Kommentare

Beispiele:

```
# dies ist ein Kommentar

/*
mehrzeiliger
Kommentar
*/
```

Kommentare beeinflussen die Auswertung nicht und können überall eingesetzt werden, außer innerhalb eines Schlüsselworts oder Namens.

3.5 Grammatik (BNF-ähnlich)

Die folgende formale Beschreibung ist eine vereinfachte, deklarative Grammatik, die die Struktur der Sprache definiert. Sie ist nicht an eine bestimmte technische Implementierung gebunden.

```
PROGRAM     ::= "PROGRAM" IDENT "{" program_body "}"

program_body ::= profile_assign entry_assign

profile_assign ::= "profile" "=" IDENT
entry_assign   ::= "entry"   "=" IDENT

PROFILE    ::= "PROFILE" IDENT "{" profile_body "}"
profile_body ::= (parameter_assign)*

BLOCK      ::= "BLOCK" IDENT "{" block_body "}"
block_body ::=
```

```
      operator_assign
    | inputs_assign
    | outputs_assign
    | layers_assign
    | requires_assign
    | ensures_assign
    | next_assign
    | on_error_assign
    | (parameter_assign)*

operator_assign ::= "operator" "=" IDENT
inputs_assign   ::= "inputs"   "=" LIST
outputs_assign  ::= "outputs"  "=" LIST
layers_assign   ::= "layers"   "=" LIST
requires_assign ::= "requires" "=" LIST
ensures_assign  ::= "ensures"  "=" LIST
next_assign     ::= "next"     "=" IDENT
on_error_assign ::= "on_error" "=" IDENT

parameter_assign ::= IDENT "=" value

value ::= STRING | NUMBER | BOOL | ENUM | LIST

IDENT ::= (letter)(letter | digit | "_")*
letter ::= A{Z | a{z
digit  ::= 0{9
```

Diese Grammatik definiert die Form der Sprache vollständig, ohne Funktionslogik oder technische Verfahren offenzulegen.

4 PROFILE

Ein `PROFILE` definiert einen deklarativen Betriebsmodus innerhalb der Sprache L-OPS. PROFILE dienen dazu, globale Rahmenbedingungen, Standardparameter und Ausführungsregeln zu spezifizieren, die für alle ihnen zugeordneten BLOCKs gelten. Sie enthalten ausschließlich deklarative Vorgaben und keinerlei technische Verfahren zur Interpretation dieser Vorgaben.

4.1 4.1 Zweck eines PROFILE

PROFILE strukturieren Abläufe in eindeutig definierte Modi. Sie werden verwendet, um:

- Standard-LAYER festzulegen,
- globale Ausführungsregeln zu definieren,

- Sicherheits- oder Betriebsmodi zu beschreiben,
- konsistente Parameter für BLOCKs bereitzustellen.

PROFILE bilden somit die konfigurierbare Obermenge eines Ablaufkontextes.

4.2 Aufbau eines PROFILE

Die allgemeine Form lautet:

```
PROFILE NAME {
    parameter = value
    parameter2 = value
    ...
}
```

Ein PROFILE enthält ausschließlich Parameterzuweisungen. Alle Parameter müssen explizit angegeben werden. Die Sprache definiert keine impliziten oder automatisch abgeleiteten Werte.

4.3 Standardfelder eines PROFILE

Obwohl PROFILE flexibel gestaltet werden können, sind folgende Felder gebräuchlich:

- `default_layers`
  Eine LIST der LAYER (E1–E9), die automatisch für BLOCKs aktiv sind, sofern der BLOCK keine eigene Layerliste angibt.
- `allow_world_access`
  Ein BOOL, das beschreibt, ob externe Aktionen prinzipiell erlaubt sind.
- `mode`
  Ein ENUM zur Beschreibung des Profilmodus (z. B. `STRICT`, `BALANCED` oder `PERMISSIVE`).
- `decision_policy`
  Eine strukturierte Angabe deklarativer Richtwerte für Entscheidungen. (Diese Richtwerte sind rein deklarativ und definieren keine technischen Auswertungsverfahren.)

PROFILE können beliebige weitere deklarative Parameter enthalten, solange sie der allgemeinen Syntax entsprechen.

4.4 Regeln für PROFILE

- PROFILE dürfen nur deklarative Parameter enthalten.
- PROFILE dürfen keine technischen Ablauflogiken oder Bewertungsschritte definieren.

- BLOCKs, die ein PROFILE verwenden, erben dessen Standardparameter (z. B. `default_layers`).
- Ein PROGRAM muss genau ein aktives PROFILE angeben.

PROFILE definieren Rahmenbedingungen, keine operativen Schritte.

4.5 Beispielprofil

Nachfolgend ein exemplarisches PROFILE zur Illustration der Syntax:

```
PROFILE SAFE_MODE {
    allow_world_access = false
    default_layers     = [E1, E2, E5, E6, E7, E9]
    mode               = STRICT

    decision_policy = {
        allow_max_risk = 0.25
        sandbox_above  = 0.25
        block_above    = 0.75
    }
}
```

Dieses Beispiel ist rein deklarativ. Es enthält keinerlei Angaben über interne Bewertungsmechanismen oder technische Ausführungsverfahren.

5 LAYER – Die neun sichtbaren Ebenen

Ein `LAYER` in L-OPS bezeichnet eine deklarative Bewertungsebene, die festlegt, welche perspektivischen Rahmenbedingungen während der Auswertung eines BLOCKs aktiv sind. LAYER definieren keine technischen Verfahren, sondern ausschließlich *welche Art von Betrachtung* ein BLOCK im deklarativen Sinne einbeziehen soll.

Die Sprache kennt neun sichtbare Ebenen, bezeichnet als `E1` bis `E9`. Diese Ebenen sind abstrakt, logisch getrennt und vollständig unabhängig von technischen Implementierungen.

5.1 Allgemeines Strukturprinzip

LAYER werden einem BLOCK wie folgt zugeordnet:

```
layers = [E1, E3, E5]
```

Wird kein `layers`-Feld angegeben, so erbt der BLOCK die `default_layers` des aktiven PROFILE.

Ein LAYER:

- ist rein deklarativ,

- beschreibt keine technische Logik,
- beeinflusst ausschließlich den strukturellen Rahmen eines BLOCKs,
- wird durch seine Bezeichnung eindeutig identifiziert.

Im Folgenden werden die neun sichtbaren Ebenen beschrieben.

5.2 E1 – Strukturebene

Die Ebene `E1` stellt die Betrachtung der formalen Struktur sicher. Sie betrifft die syntaktische und regelbasierte Gültigkeit eines BLOCKs. Diese Ebene prüft deklarativ, ob ein Block den vorgegebenen formalen Rahmen einhält.

5.3 E2 – Beobachtungsebene

`E2` beschreibt die deklarative Perspektive der Beobachtung. Sie legt fest, dass der BLOCK Eingaben, Zielobjekte oder Kontextinformationen betrachtet, ohne sie technisch auszuwerten. Die Ebene steht für eine strukturierte Erfassung der vorliegenden Informationen.

5.4 E3 – Musterebene

Die Ebene `E3` steht für deklarative Mustererkennung auf struktureller Ebene. Sie gibt an, dass ein BLOCK Muster, Beziehungen oder Zusammenhänge in den dargestellten Daten berücksichtigt, ohne technische Methoden vorzuschreiben.

5.5 E4 – Vergleichsebene

`E4` bezeichnet eine Ebene, die strukturelle Analogien oder Vergleiche zwischen deklarativ dargestellten Fällen einbezieht. Sie ist rein formal und beschreibt keine technische Ähnlichkeitsberechnung.

5.6 E5 – Prinzipienebene

Die Ebene `E5` stellt deklarative Meta-Prinzipien bereit, die einem BLOCK-Kontext zugeordnet werden können. Sie beschreibt die Einbeziehung übergeordneter Regeln oder Prioritäten, jedoch ohne technische Umsetzung.

5.7 E6 – Konsequenzenebene

`E6` beschreibt die Perspektive der strukturellen Folgenabschätzung. Ein BLOCK, der diese Ebene nutzt, bezieht deklarative Betrachtungen möglicher Auswirkungen seiner Ausgaben ein — jedoch ohne technische Berechnungsmethoden.

5.8 E7 – Menschliche Perspektive

Die Ebene `E7` steht für die Berücksichtigung deklarativer, menschenbezogener Sichtweisen wie Rollen, Rechte oder Auswirkungen auf Personen. Sie definiert keinen technischen Bewertungsmechanismus.

5.9 E8 – Regelhierarchie

`E8` beschreibt die deklarative Einbindung von Regelprioritäten oder strukturellen Hierarchien. Diese Ebene ordnet Regeln oder Profile zueinander, ohne technische Auswertungslogik zu spezifizieren.

5.10 E9 – Abschlussebene (Closure)

Die Ebene `E9` dient der deklarativen Sicherstellung vollständiger und konsistenter Abläufe. Sie beschreibt Abschlussbedingungen, Vollständigkeit von Ausgaben oder strukturelle Sicherheiten.

5.11 Zuweisung von LAYER in BLOCKs

Beispiel:

```
BLOCK ANALYZE_INPUT {
    layers   = [E1, E2, E3, E6, E9]
    operator = CHECK
    inputs   = [input_raw]
    outputs  = [parsed]
}
```

Dieses Beispiel ist deklarativ und beschreibt keine technischen Verfahren im Hintergrund.

6 BLOCK-System

Ein `BLOCK` ist die zentrale Verarbeitungseinheit der Sprache L-OPS. BLOCKs definieren deklarativ, welche Schritte ein Ablauf umfasst, welche Eingaben und Ausgaben sichtbar sind und wie der Ablauf zu anderen BLOCKs übergeht. Sie enthalten keine technischen Auswertungsverfahren und keine impliziten Aktionen.

6.1 Aufbau eines BLOCKs

Ein BLOCK besitzt die folgende Grundform:

```
BLOCK NAME {
    parameter = value
    ...
}
```

Alle Eigenschaften eines BLOCKs müssen explizit angegeben werden. BLOCKs dürfen keine verborgenen Zustandsänderungen, verdeckten Übergänge oder impliziten Operationen enthalten.

6.2 Pflichtfelder eines BLOCKs

Ein BLOCK *muss* mindestens folgende deklarative Felder enthalten:

- `operator`
  Ein ENUM-Wert, der die deklarative Kategorie der auszuführenden Operation benennt.
- `inputs` oder `target`
  Gibt an, auf welche deklarativen Werte der BLOCK zugreift.

Diese Felder beschreiben ausschließlich die Struktur der Anweisung, nicht ihre technische Umsetzung.

6.3 Optionale Felder eines BLOCKs

Zusätzliche deklarative Felder erweitern die Struktur:

- `outputs`
  Deklarative Werte, die der BLOCK erzeugt.
- `layers`
  Liste der aktiven LAYER (E1–E9), falls sie nicht vom PROFILE geerbt werden.
- `requires`
  Liste deklarativer Bedingungen, die erfüllt sein müssen, damit der BLOCK ausgeführt wird.
- `ensures`
  Liste deklarativer Bedingungen, die nach Ausführung des BLOCKs erfüllt sein sollen.
- `next`
  Name des nächsten BLOCKs bei erfolgreicher Auswertung.
- `on_error`
  BLOCK, zu dem der Ablauf im Falle nicht erfüllter Bedingungen übergeht.

Diese Felder bestimmen die deklarative Struktur von Abläufen, ohne technische Auswertung zu definieren.

6.4 Operatorzuweisung

Ein BLOCK verwendet einen Operator, etwa:

```
operator = CHECK
```

Zulässige Operatorfamilien (rein deklarativ):

- `CHECK`
- `VALIDATE`
- `TRANSFORM`
- `ROUTE`
- `EXECUTE`
- `SANDBOX`
- `BLOCK`

Die Operatoren benennen deklarative Kategorien und enthalten keine Funktionsbeschreibung.

6.5 Übergangsfelder

BLOCKs definieren explizite Übergänge mittels:

- `next`
- `on_error`

Beispiel:

```
next     = VALIDATE_TARGETS
on_error = REPORT_ISSUE
```

Es gibt keine impliziten oder automatischen Übergänge.

6.6 Ablauf eines BLOCKs (deklaratives Modell)

Der Ablauf eines BLOCKs ist wie folgt definiert:

1. Annahme der deklarierten `inputs`,
2. Anwendung des deklarierten `operator`-Typs,
3. Erzeugung der deklarierten `outputs`,
4. Prüfung der deklarierten `ensures`,
5. Übergang zu `next` oder `on_error`.

Diese Beschreibung ist rein deklarativ und legt keine technischen Methoden fest.

6.7 Beispiel eines vollständigen BLOCKs

```
BLOCK ANALYZE_RISK {
    layers   = [E3, E5, E6, E7]
    operator = VALIDATE

    inputs   = [intent, target_entities]
    outputs  = [risk_score]

    ensures  = [risk_score >= 0]
    next     = DECIDE_EXECUTION
}
```

Dieses Beispiel illustriert die deklarative Struktur eines BLOCKs, ohne interne Auswertungslogiken darzustellen.

7 Operatorfamilien

Die Sprache L-OPS verwendet deklarative Operatoren, um die Art eines Blockschrittes zu beschreiben. Ein Operator ist ein *sprachliches Etikett*, das eine Kategorie von Handlungstypen benennt, ohne deren technische Durchführung festzulegen. Operatoren definieren somit nur die semantische Rolle eines BLOCKs, nicht dessen interne Logik.

7.1 Allgemeines Prinzip

Ein Operator wird in einem BLOCK wie folgt angegeben:

```
operator = OPERATOR_NAME
```

Alle Operatoren sind ENUM-Werte. Sie besitzen keine Parameter und keine technische Semantik — sie bestimmen ausschließlich den *Typus* der deklarativen Handlung.

Die folgenden Operatorfamilien bilden die in L-OPS vorgesehenen Kategorien.

7.2 CHECK

Der Operator `CHECK` beschreibt deklarativ einen Schritt, der strukturelle, formale oder fundamentale Eigenschaften von Eingaben oder Kontextwerten berücksichtigt. Diese Betrachtung ist rein deklarativ und enthält keinerlei technische Prüflogik.

Beispiel:

```
operator = CHECK
```

7.3 VALIDATE

`VALIDATE` kennzeichnet deklarativ die Einbeziehung von Regeln, Bedingungen oder strukturellen Bewertungskriterien zur Beurteilung eines BLOCK-Kontextes. Es handelt sich ausschließlich um eine formale Beschreibung der Art des Schritts.

Beispiel:

```
operator = VALIDATE
```

7.4 TRANSFORM

Der Operator `TRANSFORM` beschreibt die deklarative Umformung oder Neuordnung von Eingabewerten. Die Operatorfamilie spezifiziert keine technischen Umwandlungsfunktionen, sondern benennt nur die Art der strukturellen Veränderung.

Beispiel:

```
operator = TRANSFORM
```

7.5 ROUTE

Der Operator `ROUTE` definiert deklarativ eine Auswahl oder Verzweigung innerhalb des Ablaufes, abhängig von im BLOCK deklarierten Werten. Es wird nicht festgelegt, wie die Auswahl technisch erfolgt.

Beispiel:

```
operator = ROUTE
```

7.6 EXECUTE

Die Operatorfamilie `EXECUTE` bezeichnet deklarativ die Ausführung eines strukturellen Plans oder einer Handlung auf der Ebene der Sprache. Sie ist nicht mit einer technischen Ausführung gleichzusetzen.

Beispiel:

```
operator = EXECUTE
```

7.7 SANDBOX

Der Operator `SANDBOX` beschreibt deklarativ, dass ein Schritt innerhalb eines isolierten Rahmens stattfinden soll. Die Sandbox-Funktion ist nicht technisch definiert; sie ist rein sprachlich als Konzept vorhanden.

Beispiel:

```
operator = SANDBOX
```

7.8 BLOCK

Der Operator `BLOCK` steht für die deklarative Beendigung oder Ablehnung eines Ablaufs. Er benennt einen Endpunkt im Prozess, ohne technische Blockiermechanismen zu beschreiben.

Beispiel:

```
operator = BLOCK
```

7.9 Zusammenfassung

Die Operatorfamilien dienen in L-OPS ausschließlich der strukturellen Gliederung von BLOCKs. Sie bieten folgende Eigenschaften:

- klare deklarative Kategorisierung eines Schrittes,
- vollständige Trennung von technischer Umsetzung,
- eindeutige, maschinenlesbare Struktur,
- konsistente Zuordnung innerhalb des gesamten Sprachsystems.

Die Operatoren bilden damit ein semantisches Gerüst für die Sprache, ohne dass sie Verfahren, Methoden oder technische Abläufe preisgeben.

8 STATE & CONTEXT

Dieses Kapitel definiert die deklarativen Datenstrukturen `STATE` und `CONTEXT`, wie sie innerhalb der Sprache L-OPS verwendet werden. Beide Strukturen bilden die Grundlage für nachvollziehbare, deterministische Abläufe, ohne technische Implementierungsdetails oder interne Auswertungsverfahren zu enthalten.

8.1 Grundprinzip

L-OPS unterscheidet streng zwischen:

- **CONTEXT** (unveränderliche externe Eingaben),
- **STATE** (deterministisch fortschreitender Ablaufzustand).

Diese Trennung stellt sicher, dass kein BLOCK verdeckte Nebenwirkungen oder nicht deklarierte Zustandsveränderungen erzeugen kann.

8.2 CONTEXT

Der `CONTEXT` repräsentiert alle externen Eingaben, die ein `PROGRAM` zur Ausführung benötigt. Er ist unveränderlich (*immutable*) und wird ausschließlich zu Beginn eines Ablaufs bereitgestellt.

Typische deklarative Felder können sein:

- `input_raw`
  Rohform der Eingabe als STRING.
- `operation_type`
  ENUM, das die Art der beabsichtigten Handlung beschreibt.
- `target_entities`
  LIST identifizierter Objekte oder Ziele.
- `context_flags`
  MAP zusätzlicher deklarativer Hinweise.

Der CONTEXT wird niemals verändert:

```
CONTEXT is read-only.
```

8.3 STATE

Der `STATE` speichert alle Werte, die im Verlauf eines `PROGRAM`-Ablaufs erzeugt oder überschrieben werden. Der STATE ist deterministisch: Er ändert sich ausschließlich an Stellen, an denen ein BLOCK explizite `outputs` deklariert.

Ein STATE-Eintrag entsteht nur über:

```
outputs = [value1, value2, ...]
```

Merkmale:

- keine impliziten Änderungen,
- keine automatischen Ergänzungen,
- vollständige Sichtbarkeit aller Werte,
- deterministische Fortschreibung pro BLOCK.

8.4 Regeln für STATE-Veränderungen

Ein BLOCK darf den STATE nur wie folgt verändern:

1. durch explizite Deklaration eines oder mehrerer Werte in `outputs`,
2. durch Überschreiben bereits existierender STATE-Werte,
3. oder durch Hinzufügen neuer STATE-Werte.

Beispiel einer deklarativen STATE-Erweiterung:

```
outputs = [parsed_intent, target_list]
```

Es gibt keine verdeckten STATE-Modifikationen.

8.5 Audit-Log (optional)

Ein `STATE` kann ein deklaratives Audit-Log enthalten:

```
audit_log = [
    { block: "CHECK_INPUT", status: "ok" },
    { block: "VALIDATE_TARGETS", result: "clean" }
]
```

Das Audit-Log ist rein deklarativ und beschreibt keine technischen Verfahren zur Protokollierung.

8.6 Zusammenspiel von STATE & CONTEXT

Der Ablauf verläuft wie folgt:

1. `CONTEXT` wird zu Beginn bereitgestellt und bleibt unverändert.
2. `STATE` beginnt leer oder mit Minimalwerten.
3. BLOCKs lesen deklarativ aus `inputs`.
4. BLOCKs schreiben deklarativ in `outputs`.
5. Der STATE wächst deterministisch mit jedem BLOCK.

8.7 8.7 Beispiel einer Nutzung in einem BLOCK

```
BLOCK ANALYZE_INPUT {
    layers   = [E1, E2]
    operator = CHECK

    inputs   = [input_raw]
    outputs  = [intent, target_entities]

    next     = ANALYZE_RISK
}
```

Dieses Beispiel zeigt:

- ein Lesen aus dem CONTEXT (`input_raw`),
- das Erzeugen neuer STATE-Werte (`intent`, `target_entities`),
- die vollständige Deklaration des Schritts,
- aber keinerlei technische Auswertung.

9 Decision-Signale

Dieses Kapitel beschreibt die deklarativen Entscheidungssignale der Sprache L-OPS. Decision-Signale stellen standardisierte Ausgänge dar, die von bestimmten BLOCKs erzeugt werden können, um den weiteren Ablauf eines PROGRAMS eindeutig zu bestimmen. Sie definieren keine technischen Bewertungsverfahren und enthalten keinerlei interne Logik.

9.1 Zweck der Decision-Signale

Decision-Signale dienen der strukturierten Steuerung eines Ablaufs. Sie ermöglichen:

- die eindeutige Deklaration eines Entscheidungsergebnisses,
- die deterministische Weiterleitung an nachgelagerte BLOCKs,
- eine klare Trennung zwischen deklarativer Entscheidung und technischer Auswertung.

9.2 Decision-Typ

Ein BLOCK kann eine deklarative Entscheidung erzeugen, indem er ein Ausgabefeld des Typs `DECISION` deklariert:

```
decision_type = DECISION
```

Das Decision-Signal selbst wird in `outputs` erzeugt:

```
outputs = [decision]
```

Der Wert von `decision` ist ein ENUM aus vier zulässigen Signalen.

9.3 Die vier Decision-Signale

L-OPS definiert genau vier deklarative Entscheidungssignale:

- **ALLOW**
  Deklariert, dass der nächste Ausführungsschritt regulär erfolgen darf.
- **WARN**
  Deklariert, dass der Ablauf fortgesetzt werden kann, jedoch mit erhöhter Aufmerksamkeit oder Kennzeichnung.
- **SANDBOX**
  Deklariert, dass der nächste BLOCK in einer isolierten Ausführung vorgesehen ist.
- **BLOCK**
  Deklariert, dass der Ablauf gestoppt wird und keine reguläre Ausführung folgen soll.

Diese ENUM-Werte besitzen keinerlei technische Funktionalität und beschreiben nur die deklarativen Steueroptionen.

9.4 Verwendung der Decision-Signale in BLOCKs

Ein BLOCK, der eine Entscheidung erzeugt, nutzt folgende Struktur:

```
BLOCK DECIDE_EXECUTION {
    operator      = VALIDATE
    inputs        = [risk_score, context_flags]
    outputs       = [decision]
    decision_type = DECISION
    next          = ROUTE_DECISION
}
```

Dies beschreibt ausschließlich:

- die Existenz des Entscheidungssignals,
- dessen Speicherung im STATE,
- den Übergang zum nächsten BLOCK.

Keine technische Entscheidungslogik ist Bestandteil dieser Spezifikation.

9.5 Routing basierend auf Decision-Signalen

Ein BLOCK kann verschiedene deklarative Übergänge definieren, abhängig vom Decision-Signal:

```
BLOCK ROUTE_DECISION {
    operator   = ROUTE
    inputs     = [decision]

    on_allow   = EXECUTE_PLAN
    on_warn    = EXECUTE_WITH_LOG
    on_sandbox = RUN_IN_SANDBOX
    on_block   = REPORT_DENIAL
}
```

Diese Struktur zeigt:

- deklarative Abbildung eines ENUM-Werts auf BLOCK-Namen,
- vollständige Transparenz der möglichen Folgepfade,
- keinerlei technische Auswertung oder Berechnung.

9.6 Beispiel für einen Decision-Flow

```
BLOCK ANALYZE_RISK {
    operator = VALIDATE
    inputs   = [intent, targets]
    outputs  = [risk_score]
    next     = DECIDE_EXECUTION
}

BLOCK DECIDE_EXECUTION {
    operator      = VALIDATE
    inputs        = [risk_score]
    outputs       = [decision]
    decision_type = DECISION
    next          = ROUTE_DECISION
}

BLOCK ROUTE_DECISION {
    operator   = ROUTE
    inputs     = [decision]
    on_allow   = EXECUTE
    on_warn    = EXECUTE_WITH_LOG
    on_sandbox = RUN_SANDBOX
    on_block   = DENY
}
```

Diese Beispiele bilden den deklarativen Mechanismus der Decision-Signale vollständig ab, ohne technische Entscheidungsverfahren offenzulegen.

10 Programmlogik

Ein `PROGRAM` in L-OPS definiert einen vollständigen, deterministischen Ablauf, der aus einer Sequenz deklarativer BLOCKs besteht. Das PROGRAM legt fest, welches PROFILE aktiv ist, wo der Ablauf beginnt und wie er sich strukturell fortsetzt. Die Programmlogik beschreibt ausschließlich die deklarative Struktur eines Ablaufs und keine technische Ausführungslogik.

10.1 PROGRAM-Struktur

Die Grundform eines PROGRAM lautet:

```
PROGRAM NAME {
    profile = PROFILE_NAME
    entry   = BLOCK_NAME
}
```

Ein PROGRAM besteht zwingend aus:

- einer Profilzuweisung,
- einer Einstiegspunktdefinition.

Das PROGRAM enthält selbst keine BLOCKs, sondern verweist auf sie.

10.2 Einstiegspunkt (entry)

Das `entry`-Feld definiert den ersten BLOCK eines Ablaufs:

```
entry = INITIAL_BLOCK
```

Der Einstiegspunkt:

- bestimmt den Start des deterministischen Ablaufs,
- muss ein existierender BLOCK-Name sein,
- darf nur einmal pro PROGRAM definiert werden.

10.3 Profilbindung

Das `profile`-Feld bestimmt, welche deklarativen Rahmenbedingungen für das gesamte PROGRAM gelten:

```
profile = SAFE_MODE
```

Die Profilbindung gilt:

- für alle BLOCKs des PROGRAMS,
- für deren Standardparameter,
- für deren geerbte LAYER,
- für alle deklarativ abgeleiteten Strukturregeln.

PROFILE ändern keine BLOCK-Struktur, sondern ergänzen sie deklarativ.

10.4 Ablaufregeln

Ein PROGRAM definiert einen Ablauf, der sich aus den Übergängen der BLOCKs ergibt. Die Regeln lauten:

1. Beginne mit dem BLOCK im Feld `entry`.
2. Führe dessen deklarative Schritte aus.
3. Bestimme über `next` oder ein deklaratives Entscheidsignal den Folgeblock.
4. Setze diesen Prozess fort, bis ein BLOCK ohne Folgeblock erreicht wird.

Diese Regeln beschreiben nur den deklarativen Rahmen, nicht die technische Auswertung.

10.5 Termination

Ein PROGRAM endet, wenn einer der folgenden Zustände erreicht ist:

- Ein BLOCK besitzt kein `next`-Feld.
- Ein BLOCK ist vom Operator `BLOCK` gesteuert.
- Ein expliziter Endpunkt über PROFILE-Regeln ist erreicht.

Termination ist ein strukturelles Konzept und enthält keine technischen Ausführungsmerkmale.

10.6 Fehlerpfade

BLOCKs können alternative deklarative Übergänge definieren:

```
on_error = FALLBACK_BLOCK
```

Fehlerpfade:

- sind rein deklarativ,
- definieren alternative strukturelle Sequenzen,
- beschreiben keine technischen Fehlerbehandlungen.

10.7 Best Practices zur Strukturierung

Deklarative Empfehlungen:

- Jeder BLOCK sollte genau einen klaren Zweck erfüllen.
- NAMEs sollen funktional beschreibend, aber technisch neutral sein.
- BLOCKs sollen keine übermäßig langen `requires`- oder `ensures`-Listen haben.
- Decision-Flows sollen eigene BLOCKs verwenden, um deklarative Klarheit sicherzustellen.
- PROFILE sollten konsistent innerhalb eines PROGRAMS genutzt werden.

10.8 10.8 Beispiel eines vollständigen PROGRAM-Skeletts

```
PROGRAM MAIN_FLOW {
    profile = SAFE_MODE
    entry   = ANALYZE_INPUT
}
```

Dieses PROGRAM verweist auf BLOCKs, die den vollständigen Ablauf beschreiben, jedoch selbst keine technische Umsetzung enthalten.

11 End-to-End-Referenzprogramm

Dieses Kapitel zeigt ein vollständiges, durchgängiges Beispiel für ein L-OPS PROGRAM. Das Beispiel illustriert ausschließlich die deklarative Struktur der Sprache:

- PROFILE-Nutzung,
- BLOCK-Komposition,
- LAYER-Zuweisung,
- STATE- und CONTEXT-Verweise,
- Decision-Signale,
- deterministische Ablaufsteuerung.

Es enthält keinerlei technische Verfahren, Berechnungen oder Auswertungsmechanismen.

11.1 Beispiel-CONFIG: PROFILE

```
PROFILE SAFE_MODE {
    allow_world_access = false
    default_layers     = [E1, E2, E5, E7, E9]
    mode               = STRICT

    decision_policy = {
        allow_max_risk = 0.20
        sandbox_above  = 0.20
        block_above    = 0.70
    }
}
```

Dieses PROFILE dient als deklarative Grundlage für alle folgenden BLOCKs.

11.2 Beispiel: BLOCK-Definitionen

BLOCK ANALYZE_INPUT

```
BLOCK ANALYZE_INPUT {
    layers   = [E1, E2]
    operator = CHECK

    inputs   = [input_raw]
    outputs  = [intent, target_entities]

    ensures  = [intent != ""]
    next     = ANALYZE_RISK
}
```

11.2.1 BLOCK ANALYZE_RISK

```
BLOCK ANALYZE_RISK {
    layers   = [E3, E5, E6]
    operator = VALIDATE

    inputs   = [intent, target_entities]
    outputs  = [risk_score]

    ensures  = [risk_score >= 0]
    next     = DECIDE_EXECUTION
}
```

11.2.2 BLOCK DECIDE_EXECUTION

```
BLOCK DECIDE_EXECUTION {
    layers       = [E5, E6, E7, E9]
    operator     = VALIDATE

    inputs        = [risk_score]
    outputs       = [decision]
    decision_type = DECISION

    next          = ROUTE_DECISION
}
```

11.2.3 BLOCK ROUTE_DECISION

```
BLOCK ROUTE_DECISION {
    operator = ROUTE
    inputs   = [decision]

    on_allow   = EXECUTE_PLAN
    on_warn    = EXECUTE_WITH_LOG
    on_sandbox = RUN_SANDBOX
    on_block   = DENY
}
```

11.2.4 BLOCK EXECUTE_PLAN

```
BLOCK EXECUTE_PLAN {
    operator = EXECUTE

    inputs  = [intent, target_entities]
    outputs = [result]

    ensures = [result != null]
```

```
}
```

11.2.5 BLOCK EXECUTE_WITH_LOG

```
BLOCK EXECUTE_WITH_LOG {
    operator = EXECUTE

    inputs  = [intent]
    outputs = [result, warning_log]
}
```

11.2.6 BLOCK RUN_SANDBOX

```
BLOCK RUN_SANDBOX {
    operator = SANDBOX
    inputs   = [intent, target_entities]
    outputs  = [sandbox_result]
}
```

11.2.7 BLOCK DENY

```
BLOCK DENY {
    operator = BLOCK
    inputs   = [intent]
}
```

11.3 Beispiel: PROGRAM

```
PROGRAM MAIN_FLOW {
    profile = SAFE_MODE
    entry   = ANALYZE_INPUT
}
```

11.4 Ablaufbeschreibung (deklarativ)

Der Ablauf dieses PROGRAMS verläuft deterministisch:

1. `ANALYZE_INPUT` erzeugt `intent` und `target_entities`.
2. `ANALYZE_RISK` erzeugt `risk_score`.
3. `DECIDE_EXECUTION` erzeugt `decision`.
4. `ROUTE_DECISION` verzweigt den Ablauf.
5. Einer der BLOCKs `EXECUTE_PLAN`, `EXECUTE_WITH_LOG`, `RUN_SANDBOX` oder `DENY` bildet den Endpunkt.

Dies ist ausschließlich eine strukturelle Beschreibung. Die technische Ausführung bleibt vollständig außerhalb der Spezifikation.

12 Vollständige formale Grammatik

Dieses Kapitel enthält die formale, BNF-ähnliche Grammatik der Sprache L-OPS. Die Grammatik beschreibt ausschließlich die Struktur der Sprache, ohne technische Auswertungsverfahren oder interne Mechanismen einzubeziehen. Alle Regeln sind deklarativ und definieren die Form syntaktisch gültiger L-OPS-Programme.

12.1 Notation

Die folgende Notation wird verwendet:

- Nichtterminale sind in Großbuchstaben gesetzt.
- Terminalsymbole erscheinen in Anführungszeichen.
- `|` bedeutet Alternative.
- `*` bedeutet Wiederholung (0 oder mehr).
- `+` bedeutet mindestens eine Wiederholung.
- Eckige Klammern werden in der Grammatik als Literale verwendet.

12.2 Oberste Strukturebene

```
SPECIFICATION ::= (PROFILE | BLOCK | PROGRAM)*
```

Eine L-OPS-Spezifikation besteht aus beliebig vielen PROFILE-, BLOCK- oder PROGRAM-Definitionen.

12.3 PROGRAM-Grammatik

```
PROGRAM ::= "PROGRAM" IDENT "{" PROGRAM_BODY "}"

PROGRAM_BODY ::= PROFILE_ASSIGN ENTRY_ASSIGN

PROFILE_ASSIGN ::= "profile" "=" IDENT
ENTRY_ASSIGN   ::= "entry"   "=" IDENT
```

12.4 PROFILE-Grammatik

```
PROFILE ::= "PROFILE" IDENT "{" PROFILE_BODY "}"

PROFILE_BODY ::= PARAM_ASSIGN*
```

Alle PROFILE-Felder bestehen aus einfachen Parameterzuweisungen.

12.5 BLOCK-Grammatik

```
BLOCK ::= "BLOCK" IDENT "{" BLOCK_BODY "}"

BLOCK_BODY ::=
      OPERATOR_ASSIGN
    | INPUTS_ASSIGN
    | OUTPUTS_ASSIGN
    | LAYERS_ASSIGN
    | REQUIRES_ASSIGN
    | ENSURES_ASSIGN
    | NEXT_ASSIGN
    | ERROR_ASSIGN
    | PARAM_ASSIGN
    | BLOCK_BODY BLOCK_BODY
```

12.6 Parameterzuweisungen

```
PARAM_ASSIGN ::= IDENT "=" VALUE
```

Dies ist die generische Form einer Zuweisung.

12.7 BLOCK-spezifische Felder

```
OPERATOR_ASSIGN ::= "operator" "=" IDENT
INPUTS_ASSIGN   ::= "inputs"   "=" LIST
OUTPUTS_ASSIGN  ::= "outputs"  "=" LIST
LAYERS_ASSIGN   ::= "layers"   "=" LIST
REQUIRES_ASSIGN ::= "requires" "=" LIST
ENSURES_ASSIGN  ::= "ensures"  "=" LIST

NEXT_ASSIGN     ::= "next"     "=" IDENT
ERROR_ASSIGN    ::= "on_error" "=" IDENT
```

12.8 Decision-Felder (optional)

```
DECISION_ASSIGN ::= "decision_type" "=" "DECISION"
```

12.9 Wertetypen

```
VALUE ::= STRING | NUMBER | BOOL | ENUM | LIST
```

12.10 Listen

```
LIST ::= "[" LIST_VALUES "]"

LIST_VALUES ::= VALUE
              | VALUE "," LIST_VALUES
```

12.11 Identifikatoren

```
IDENT ::= LETTER (LETTER | DIGIT | "_")*

LETTER ::= "A".."Z" | "a".."z"
DIGIT  ::= "0".."9"
```

12.12 Literale

```
STRING ::= "\"" (ANY_CHAR)* "\""

NUMBER ::= DIGIT+ ("." DIGIT+)?

BOOL   ::= "true" | "false"

ENUM   ::= IDENT
```

12.13 Kommentare

Kommentare sind außerhalb der Grammatik definiert und werden von der Auswertung ignoriert.

```
COMMENT_SINGLE ::= "#" ANY_CHAR*
COMMENT_MULTI  ::= "/*" ANY_CHAR* "*/"
```

12.14 Zusammenfassung

Die obenstehende Grammatik definiert:

- die vollständige Struktur eines PROGRAMS,
- die vollständige Struktur eines PROFILE,
- die vollständige Struktur eines BLOCKS,
- alle Parameter- und Wertetypen,
- die syntaktische Form aller L-OPS-Elemente.

Damit ist die deklarative Form der Sprache vollständig formalisiert, ohne technische Ausführungsdetails offenzulegen.

13 Anhang

Der Anhang enthält ergänzende deklarative Materialien zur Sprache L-OPS. Alle Inhalte dienen ausschließlich der strukturellen Klarheit und erweitern nicht den normativen Kern der Spezifikation.

13.1 Glossar

PROGRAM Oberste Einheit eines L-OPS-Ablaufs. Definiert ein aktives PROFILE und einen Einstiegspunkt.

PROFILE Deklarativer Betriebsmodus, der Standardparameter und Rahmenbedingungen vorgibt.

BLOCK Grundlegende Verarbeitungseinheit. Beschreibt einen Schritt innerhalb eines Ablaufs.

LAYER (E1–E9) Neun abstrakte, deklarative Bewertungsebenen, die die Perspektive eines BLOCKs bestimmen.

STATE Deterministisch fortschreibender Ablaufzustand. Enthält nur deklarativ erzeugte Werte.

CONTEXT Unveränderliche Eingaben eines PROGRAMs.

operator ENUM-Wert, der die deklarative Kategorie eines BLOCK-Schritts benennt.

requires Deklarative Vorbedingungen eines BLOCKs.

ensures Deklarative Nachbedingungen eines BLOCKs.

decision ENUM-Wert eines Decision-Signals (`ALLOW`, `WARN`, `SANDBOX`, `BLOCK`).

next Deklarativer Übergang zu einem nachfolgenden BLOCK.

on_error Alternativer, deklarativer Übergang bei nicht erfüllten Bedingungen.

on_allow / on_warn / on_sandbox / on_block Strukturierter, deklarativer Übergang abhängig vom Entscheidungssignal.

13.2 Tabellenübersichten

13.2.1 Operatorfamilien

Operator	Beschreibung (deklarativ)
CHECK	strukturelle Prüfung
VALIDATE	regelbasierte Betrachtung
TRANSFORM	Umformung deklarativer Werte
ROUTE	deklarative Verzweigung
EXECUTE	strukturelle Handlungsausführung
SANDBOX	isolierte Deklaration
BLOCK	strukturelle Beendigung des Ablaufs

13.2.2 LAYER (E1–E9)

Layer	Beschreibung
E1	Strukturprüfung
E2	Beobachtungsebene
E3	Musterebene
E4	Vergleichsebene
E5	Prinzipienebene
E6	Konsequenzenebene
E7	Menschliche Perspektive
E8	Regelhierarchie
E9	Abschluss / Closure

13.2.3 Entscheidungssignale

Signal	Bedeutung (deklarativ)
ALLOW	reguläre Fortsetzung
WARN	Fortsetzung mit Hinweis
SANDBOX	isolierte Folgeausführung
BLOCK	strukturierter Abbruch

13.3 Beispielmuster für BLOCK-Namen

Die folgenden Namensmuster dienen lediglich als Stilhilfe:

- `ANALYZE_INPUT`
- `CHECK_ENTITY`
- `VALIDATE_TARGETS`
- `TRANSFORM_STRUCTURE`
- `ROUTE_DECISION`
- `EXECUTE_PLAN`
- `REPORT_ISSUE`

BLOCK-Namen sollen:

- klar,
- sprechend,
- technisch neutral,
- strukturell motiviert

sein.