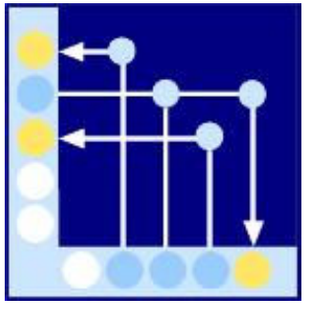
Hochschule Aalen
Fakultät Elektronik und Informatik
Studienbereich Informatik



Programmieren in MOSTflexiPL

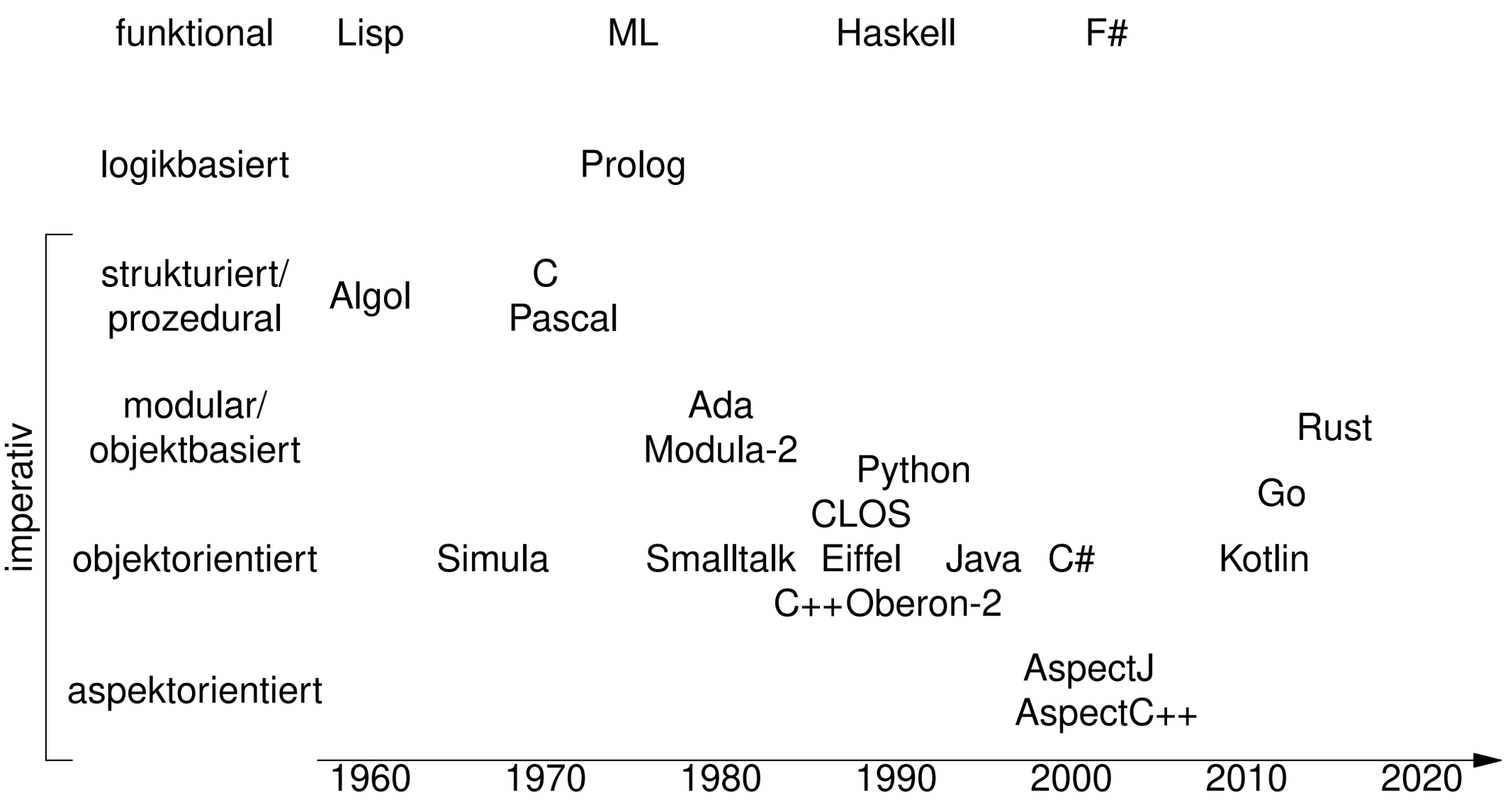
Vorlesung im Wintersemester 2025/2026

Prof. Dr. habil. Christian Heinlein

christian.heinleins.net

1 Einleitung

1.1 Bekannte Programmiersprachen und -paradigmen



1.2 Imperative und funktionale Programmierung

1.2.1 Unterschied

- *Imperative* Programme sind *zustandsbasiert*, d. h. sie können den Zustand ihrer Objekte durch Zuweisungen an (globale, lokale oder Objekt-) Variablen *verändern*, die wiederum in Bedingungen von Fallunterscheidungen und Schleifen verwendet werden und damit das Programmverhalten beeinflussen.

- Wenn ein Objekt an verschiedenen Stellen eines Programms verwendet wird, kann eine Veränderung an einer Stelle leicht unerwartete und unerwünschte Nebeneffekte an anderen Stellen haben.

- Demgegenüber sind *funktionale* Programme *zustandslos*, d. h. es gibt nur Konstanten, aber keine Variablen und Zuweisungen und somit keinen veränderbaren Zustand. Dadurch sind unerwartete Nebeneffekte grundsätzlich ausgeschlossen.

- Fallunterscheidungen (deren Bedingungen auf konstanten Werten beruhen) sind genauso möglich, aber anstelle von Schleifen muss grundsätzlich mit Rekursion gearbeitet werden.

- Eine häufig in funktionalen Programmen verwendete Datenstruktur sind Listen, die aus einem ersten Element (head) und einer (ggf. leeren) Restliste (tail) bestehen.

- Funktionen können solche Listen bequem verarbeiten, indem sie das erste Element direkt und die Restliste durch einen rekursiven Aufruf der Funktion verarbeiten.

- Die Korrektheit solcher Funktionen kann typischerweise leicht durch eine vollständige Induktion über die Länge der Liste (meist mit Induktionsanfang 0) bewiesen werden.

1.2.2 Beispiele

- Erstellen einer Liste mit den natürlichen Zahlen von `m` bis `n`

- Ersetzen eines Werts `u` durch einen Wert `v` in einer Liste ganzer Zahlen

Implementierung in der imperativen Programmiersprache Java

```
List<Integer> nat (int m, int n) {
  List<Integer> xs = new LinkedList<Integer>();
  for (int i = m; i <= n; i++) xs.add(i);
  return xs;
}

List<Integer> subst (List<Integer> xs, int u, int v) {
  List<Integer> ys = new LinkedList<Integer>();
  for (int x : xs) {
    if (x == u) x = v;
    ys.add(x);
  }
  return ys;
}
```

- Die Listen `xs` in der Methode `nat` und `ys` in der Methode `subst` werden innerhalb der Schleife immer wieder durch Ausführung der Methode `add` verändert.

Implementierung in der funktionalen Programmiersprache Haskell

```
nat m n = if m > n then [] else m : (nat (m+1) n)

subst [] u v = []
subst (x:xs) u v = (if x == u then v else x) : (subst xs u v)
```

- Hier liefert jeder Aufruf der Funktionen `nat` und `subst`, egal ob direkt von außen oder durch Rekursion, eine unveränderliche Liste ganzer Zahlen.
- Eine Definition der Gestalt `f x ... = impl` definiert eine Funktion mit dem Namen `f`, Parametern `x ...` und Implementierung `impl`, die anschließend mit Ausdrücken der Gestalt `f a ...` mit den Argumenten `a ...` aufgerufen werden kann.
- Ein Ausdruck der Gestalt `if x then y else z` liefert, abhängig vom Wert der Bedingung `x`, entweder den Wert von `y` oder den Wert von `z`.
- `[]` bezeichnet die leere Liste, ein Ausdruck der Gestalt `x:xs` konstruiert und liefert eine neue Liste mit erstem Element `x` und Restliste `xs`.
  (Technisch erstellt `x:xs` nur einen neuen Listenknoten, der das Element `x` und den Zeiger `xs` enthält und liefert einen Zeiger auf den Knoten zurück, der logisch die gesamte Liste repräsentiert.)

- Die beiden Definitionen der Funktion `subst` realisieren eine implizite Fallunterscheidung durch sog. *pattern matching*:
  Wenn die übergebene Liste von der Gestalt `[]` (also leer) ist, wird die erste Definition aufgerufen, wenn sie von der Gestalt `x:xs` ist (also mindestens ein Element `x` enthält), wird die zweite Definition aufgerufen, deren Parameter `x` bzw. `xs` dann das erste Element bzw. die (ggf. leere) Restliste der übergebenen Liste enthält.

- Wie in § 1.2.1 erwähnt, kann die Korrektheit der Funktionen `nat` und `subst` jeweils leicht durch vollständige Induktion bewiesen werden (bei `subst` nach der Länge der übergebenen Liste, bei `nat` nach der Länge $k = \max(n - m + 1, 0)$ der zurückgelieferten Liste).

- Obwohl Haskell statisch typisiert ist, kann man die Parameter- und Resultattypen von Funktionen auch weglassen, wenn sie vom Übersetzer deduziert werden können. (Damit kann die Funktion `nat` tatsächlich für beliebige numerische Typen und die Funktion `subst` für Listen mit beliebigen Typen aufgerufen werden.)

1.2.3 Anmerkungen

- Imperative und funktionale Programmierung sind zunächst nur *Programmierparadigmen*, d. h. sie definieren die grundsätzliche Art und Weise, wie man programmiert (bzw. beim Programmieren denkt).

- Imperative bzw. funktionale Programmiersprachen stellen die Mechanismen bereit, die für die Programmierung im jeweiligen Paradigma benötigt werden. Beispielsweise muss eine imperative Programmiersprache Variablen und Zuweisungen anbieten, während eine funktionale Programmiersprache Rekursion anbieten muss und diese auch möglichst effizient unterstützen sollte (z. B. indem der Übersetzer automatisch sog. Endrekursionen eliminiert).

- Nichtsdestotrotz kann man mit einer imperativen Programmiersprache normalerweise (wenn sie Rekursion erlaubt) auch funktional programmieren, indem man vollständig auf Zuweisungen verzichtet.

- Umgekehrt kann man in einer rein funktionalen Sprache aber nicht imperativ programmieren, weil es keine Variablen gibt.

1.3 Bedeutung des Namens MOSTflexiPL

Das Akronym MOSTflexiPL (Aussprache wie *most flexible*, aber mit *p* statt *b*) steht für:

- **MO**dular
  - Ein umfangreiches Programm kann in überschaubare Module zerlegt werden.
  - Ein Modul kann in verschiedenen Programmen als Bibliothek verwendet werden.
- **S**tatically **T**yped
  - Ein Programm wird zur Übersetzungszeit vollständig auf Typkorrektheit überprüft.
  - Zur Laufzeit können keine Typfehler auftreten.
- **fl**exibly **ex**tens**i**ble
  - Die Syntax der Sprache kann von jedem Programmierer mit geringem Aufwand nahezu beliebig erweitert und angepasst werden.
- **P**rogramming **L**anguage
  - Es handelt sich um eine universell einsetzbare Programmiersprache.

1.4 Anschauliche Vergleiche

1.4.1 Umgestaltung eines Raums

- In einem Raum kann man
  - Möbel beliebig verschieben
  - Lampen an der Decke montieren und Regalbretter an der Wand befestigen
  - Wände beliebig streichen oder tapezieren
  - usw.
- Aber man kann nicht einfach
  - Türen und Fenster versetzen
  - Wände verschieben oder entfernen
  - Decken höher oder tiefer hängen
  - usw.

Vergleich mit Programmiersprachen

- Mit einer konventionellen Programmiersprache kann man
  - neue Datentypen und Funktionen definieren
  - eventuell die vordefinierten Operatoren überladen (z. B. in C++)
  - eventuell neue Präfix-, Infix- und Postfixoperatoren definieren (z. B. in Haskell)
- Aber man kann nicht einfach
  - die Syntax von Typen und Deklarationen erweitern
  - neue Operatoren mit beliebiger Syntax definieren
  - neue Anweisungen zur Ablaufsteuerung definieren
- Mit MOSTflexiPL kann man all das und noch viel mehr, so wie wenn man einen Raum vollkommen uneingeschränkt umgestalten könnte.

1.4.2 Natürliche Sprache und mathematische Notation

- In einer natürlichen Sprache darf man
  - beliebige neue Begriffe definieren und anschließend verwenden
  - beliebige Abkürzungen definieren und anschließend verwenden
- Aber man darf nicht einfach
  - neue Satzzeichen definieren und anschließend verwenden
  - die Grammatik der Sprache verändern
  - die Regeln für Groß- und Kleinschreibung verändern
  - in die „verkehrte“ Richtung schreiben
- In einer mathematischen Abhandlung darf man
  - beliebige neue Symbole mit beliebiger Notation definieren und anschließend verwenden, zum Beispiel $\sum_{k=1}^{n} a_k$ oder $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$
- Damit sind konventionelle Programmiersprachen ähnlich eingeschränkt wie natürliche Sprachen, während MOSTflexiPL ähnliche Möglichkeiten wie mathematische Notation bietet.

1.4.3 Modelleisenbahnen

- Ein Modelleisenbahnsystem bietet
  - gerade Gleise unterschiedlicher Länge
  - gebogene Gleise mit unterschiedlichen Radien und Winkeln
  - verschiedene Arten von Weichen und Kreuzungen
- Aber man erhält damit nicht
  - gerade Gleise beliebiger Länge
  - gebogene Gleise mit beliebigem Radius oder Winkel
  - davon abweichende Gleisbilder (z. B. Ellipsen)
  - Weichen und Kreuzungen beliebiger Art
    (z. B. Weichen mit mehr als drei Wegen oder Kreuzungen mit beliebigen Winkeln)

Vergleich mit Programmiersprachen

- Konventionelle Programmiersprachen sind ähnlich eingeschränkt wie normale Modelleisenbahnsysteme.

- MOSTflexiPL entspricht einem fiktiven Modelleisenbahnsystem, mit dem man
  - gerade Gleise auf eine beliebige Länge dehnen oder stauchen kann
  - gebogene Gleise beliebig formen kann
    (das gibt es bei manchen Herstellern tatsächlich)
  - neue Weichen- und Kreuzungsformen einfach durch Übereinanderlegen mehrerer gebogener oder gerader Gleise erstellen kann

- Diese Flexibilität kommt auch im Logo der Sprache zum Ausdruck:

1.5 Grundprinzipien

- Alles ist ein *Ausdruck*, d. h. die Anwendung eines *Operators* auf Teilausdrücke (*Operanden*).
- Ein Operator kann beliebig viele Namen und Operanden (dargestellt durch •) in beliebiger Reihenfolge besitzen, zum Beispiel (vordefinierte und benutzerdefinierte Operatoren):
  - Infixoperatoren: Addition `•+•`, Zuweisung `•=!•`
  - Präfixoperatoren: Vorzeichenwechsel `-•`, logische Negation `not •`
  - Postfixoperatoren: Fakultät `•!`, Quadrat $\bullet^2$
  - Zirkumfixoperatoren: Klammern `(•)`, Absolutbetrag $|\bullet|$
  - Anweisungen zur Ablaufsteuerung:
    Verzweigung `if•then•else•end`, Schleife `while•do•end`
  - Deklarationen: Konstantendeklaration `•:•=•`, Operatordeklaration `•->(•=•)`
  - Typausdrücke: Variablentyp `•?`, Reihentyp `•[]`
- Jeder Ausdruck liefert zur Laufzeit einen Wert.
- Basierend auf einer kleinen Menge vordefinierter Operatoren (für Arithmetik, Logik, Ablaufsteuerung), können beliebige benutzerdefinierte Operatoren definiert werden.

1.6 Übersetzer

1.6.1 Verfügbarkeit

- Der MOSTflexiPL-Übersetzer `flxc` ist auf einem virtuellen bwCloud-Server mit Ubuntu 24.04 LTS im Verzeichnis `/usr/local/bin` installiert. Die IP-Adresse des Servers wird in der Vorlesung bekanntgegeben.

- Nach entsprechender Freischaltung und Hinterlegen des öffentlichen SSH-Schlüssels (der ggf. mit `ssh-keygen` erzeugt werden kann), kann man sich mit SSH mit Benutzername `studNNN` (wobei `NNN` die eigene Matrikelnummer ist) ohne Kennwort auf diesem Server anmelden.

- Der Compiler wird im Laufe des Semesters regelmäßig aktualisiert.

1.6.2 Aufruf

- `flxc filename.flx` überprüft das MOSTflexiPL-Programm in der angegebenen Quelldatei auf syntaktische und semantische Korrektheit und erstellt den zugehörigen Syntaxbaum.

- Wenn das Programm korrekt und eindeutig ist, wird der Syntaxbaum anschließend direkt ausgeführt und außerdem in binärer Form in der Datei `filename.flx.bin` gespeichert.

- Wenn die Quelldatei (und der Übersetzer) seit dem letzten erfolgreichen Aufruf des Übersetzers nicht verändert wurde (d. h. wenn die Binärdatei neuer als die Quelldatei ist und von derselben Version des Übersetzers stammt), entfällt die zeitaufwendige Korrektheitsüberprüfung des Programms; dann wird der Syntaxbaum direkt aus der Binärdatei gelesen und sofort ausgeführt, was bei umfangreichen Programmen wesentlich schneller geht.

1.6.3 Fehlermeldungen

- Wenn das Programm nicht korrekt ist, erhält man eine oder mehrere Diagnoseausgaben über mögliche Fehlerursachen.

- Hierbei werden drei wesentliche Arten von Fehlern unterschieden:

  - *Einsetzungsfehler*:
    Ein Teilausdruck kann nicht als Operand in einen noch unvollständigen anderen Ausdruck eingesetzt werden, entweder weil sein Typ nicht passt oder weil eine Ausschlussangabe (vgl. § 2.9) verletzt werden würde.

    Zum Beispiel: `1 + true`
    Der Teilausdruck `true` mit Wert `bool` kann nicht als Operand in die noch unvollständige Addition `1 +` eingesetzt werden.

  - *Fortsetzungsfehler*:
    Ein noch unvollständiger Ausdruck kann nicht fortgesetzt werden, weil in der Eingabe kein passendes Wort folgt.

    Zum Beispiel: `if x > 2 else y end`
    Der noch unvollständige Ausdruck `if x > 2` kann nicht fortgesetzt werden, weil das dafür nötige Wort `then` nicht in der Eingabe folgt.
    (Alternativ könnte zunächst auch der Teilausdruck `x > 2` irgendwie fortgesetzt werden, z. B. zu `x > 2 and x < 5`, oder der Teilausdruck `2`, z. B. zu `2 * 3`.)

- *Abschlussfehler*:
  Erst bei der abschließenden Überprüfung eines bereits vollständigen Ausdrucks wird ein Fehler festgestellt.
  Die Ursachen hierfür sind sehr speziell, z. B. dass der Typ eines Teilausdrucks nicht eindeutig ermittelt werden kann oder dass ein impliziter Parameter nicht belegt werden kann.

- Die an einem möglichen Fehler beteiligten (vollständigen oder unvollständigen) Ausdrücke werden immer in der Form `12:34(...)56:78` ausgegeben, wobei `12:34` und `56:78` die Anfangs- bzw. Endposition des Ausdrucks ist, die jeweils aus einer Zeilennummer und einer Zeichenposition innerhalb dieser Zeile besteht, die beide ab 1 gezählt werden. (Der Ausdruck beginnt an der Anfangsposition und endet unmittelbar *vor* der Endposition.)

- Zwischen den Klammern steht der Operator des Ausdrucks, z. B. `add/sub:` •`(+|-)`• für den vordefinierten Operator für Addition und Subtraktion oder `true` für die vordefinierte Konstante `true`.

- Wenn es sich um einen benutzerdefinierten Operator handelt, steht zwischen den Klammern wiederum eine Angabe der Form `21:43[9]65:87`, wobei `21:43` und `65:87` die Anfangs- bzw. Endposition der Definition des Operators und `9` die Nummer der Quelldatei ist, in der sich diese Definition befindet.

- Die zu diesen Nummern gehörenden Namen der Quelldateien werden ganz am Ende der Fehlerausgabe angezeigt.

- Die Lesbarkeit der Fehlermeldungen wird durch die Verwendung unterschiedlicher Farben verbessert.

- Wenn `flxc` mit der Option `-v` aufgerufen wird, werden noch ausführlichere Fehlermeldungen ausgegeben:
  Zum einen wird von benutzerdefinierten Operatoren zusätzlich ihre Signatur (vgl. § 2.8) ausgegeben, zum anderen werden für jede mögliche Fehlerursache zusätzlich die zugehörige(n) Zeile(n) der Quelldatei ausgegeben.

1.6.4 Ausgabe von Mehrdeutigkeiten

- Wenn das Programm mehrdeutig ist, erhält man eine oder mehrere Diagnoseausgaben der Art:

  ```
  ambiguity from position 12:34 to 56:78
  ```

  Dabei bezeichnen `12:34` und `56:78` wieder die Anfangs- bzw. Endposition des mehrdeutigen Quelltextbereichs.

- In den nachfolgenden Zeilen wird dieser Quelltextbereich (ohne den darin enthaltenen Zwischenraum) für jede mögliche Interpretation zweimal mit unterschiedlichen Farben ausgegeben:

- Im ersten Teil jeder Zeile (vor dem senkrechten Trennstrich) repräsentiert jede Farbe einen bestimmten (vordefinierten oder benutzerdefinierten) Operator.
  Die zu diesen Farben gehörenden Operatoren werden anschließend ebenfalls mit der jeweiligen Farbe ausgeben.

- Im zweiten Teil jeder Zeile (nach dem senkrechten Trennstrich) repräsentiert jede Farbe eine bestimmte Vorrangebene.
  Die Farben dieser Vorrangebenen sind von oben (niedriger Vorrang) nach unten (hoher Vorrang) immer Rot, Gelb, Grün, Blau usw. und werden am Ende der Zeile `ambiguity from ...` angezeigt.

Beispiel

```
ambiguity from position 3:6 to 3:12: operators | precedence levels (■■■)
2+3² | 2+3²
2+3² | 2+3²
+ operator defined from position 1:1 to 1:20 in module ambig5.flx
+ predefined operator add/sub: •(+|-)•
+ predefined operator int literal
```

- Weil für den in der Quelldatei `ambig5.flx` definierten Operator $\bullet^2$ (links vom Trennstrich rot dargestellt) kein Vorrang festgelegt wurde, kann der Ausdruck `2 + 3`^{2} auf zwei Arten interpretiert werden, die rechts vom Trennstrich angezeigt werden:
  - Entweder wird der Quadratoperator (rot) auf die Addition (gelb) von `2` und `3` (jeweils grün) angewandt.
  - Oder die Addition (rot) wird auf `2` (gelb) und die Anwendung des Quadratoperators (ebenfalls gelb) auf `3` (grün) angewandt.

1.6.5 Integration in Visual Studio Code

- Im Verzeichnis `/usr/local/share/flxc` des virtuellen Servers steht eine Erweiterung für Visual Studio Code zur Verfügung, die den Übersetzer in diesen Editor integriert.
- Wenn diese Erweiterung in VS Code für Linux installiert ist, wird eine MOSTflexiPL-Quelldatei immer dann vom Übersetzer überprüft, wenn sie geöffnet oder gespeichert wird (d. h. der Code wird nicht inkrementell überprüft). Eventuelle Fehler oder Mehrdeutigkeiten werden dann direkt in VS Code angezeigt.
- Außerdem stehen folgende Funktionen zur Verfügung:
  - Einfache Syntaxfärbung, z. B. für Kommentare und Klammern.
  - Wenn eine Quelldatei keine Fehler, sondern höchstens Mehrdeutigkeiten enthält, kann mit „Go to Definition“ zur Definition eines Namens und mit „Go to References“ zu den Verwendungsstellen eines Namens gesprungen werden.
- Immer, wenn der Compiler aktualisiert wird, wird auch diese Erweiterung für VS Code entsprechend aktualisiert.

1.6.6 Integration in Vi IMproved

- Eine entsprechende Integration in Vi IMproved ist in Arbeit.

1.7 Praxistipps

- Vergessen Sie fast alles, was Sie von anderen Programmiersprachen gewohnt sind!
- Lassen Sie Ihrer Phantasie freien Lauf!
  Gestalten Sie Ihren „Programmier-Raum“ nach Herzenslust!
- Semikolon ist ein Infix-Operator und darf deshalb nur *zwischen* Teilausdrücken, aber nicht am Ende einer „Ausdruckskette“ stehen.
- Zwischenraum zwischen den Namen und Operanden eines Ausdrucks ist *immer* optional. Deshalb ist z. B. `printonlyx` anstelle von `print only x` prinzipiell korrekt, sofern es eine geeignete Konstante `x` gibt. (`pr int on ly x` wäre aber nicht korrekt: Innerhalb eines einzelnen Namens darf kein Zwischenraum stehen.)
- Wenn ein Fehler oder eine Mehrdeutigkeit nicht offensichtlich ist, reduzieren Sie das Programm schrittweise so lange, bis das Problem verschwindet.
- Entwickeln Sie Programme umgekehrt in vielen kleinen Schritten und überprüfen Sie jeden einzelnen Schritt durch Aufruf des Übersetzers.
- Wenn sich ein Fehler auch nach längerer Suche nicht erklären lässt, fragen Sie nach. (Auch der Übersetzer ist leider noch nicht fehlerfrei.)

1.8 Literaturhinweise

❒ C. Heinlein: “MOSTflexiPL – An Extremely Flexible Programming Language.” In: T. Noll, I. Fesefeldt (eds.): *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Aachener Informatik-Berichte AIB-2023-02, Department of Computer Science, RWTH Aachen, 2023, 55–72.

Die aktuellste Veröffentlichung.

❒ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

Die bisher umfangreichste Veröffentlichung.

❒ C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: S. Jähnichen, B. Rumpe, H. Schlingloff (eds.): *Software Engineering 2012 Workshopband* (Fachtagung des GI-Fachbereichs Softwaretechnik; Februar/März 2012; Berlin). Lecture Notes in Informatics P-199, Gesellschaft für Informatik e. V., Bonn, 2012, 45–60.

Eine kürzere Beschreibung der Sprache.

- C. Heinlein: “Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL.” In: K. Schmid et al. (ed.): *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014* (Kiel, February 2014). CEUR Workshop Proceedings, 193–212, http://ceur-ws.org/Vol-1129/paper4A.pdf.

  Eine Beschreibung sogenannter virtueller Operatoren.

Alle Publikationen außer der ersten verwenden eine inzwischen veraltete Syntax der Sprache.

Sie sind alle auf `flexipl.info/publications` verfügbar.