Die Programmiersprache SimPL

Christian Silberbauer

Motivation

A language that doesn't affect the way you think about programming, is not worth knowing.

- Quelle: Perlis, Alan, SIGPLAN Notices Vol. 17, No. 9, September 1982

Merkmale

- SimPL ist mit dem Ziel entworfen, die Programmierung zu vereinfachen.

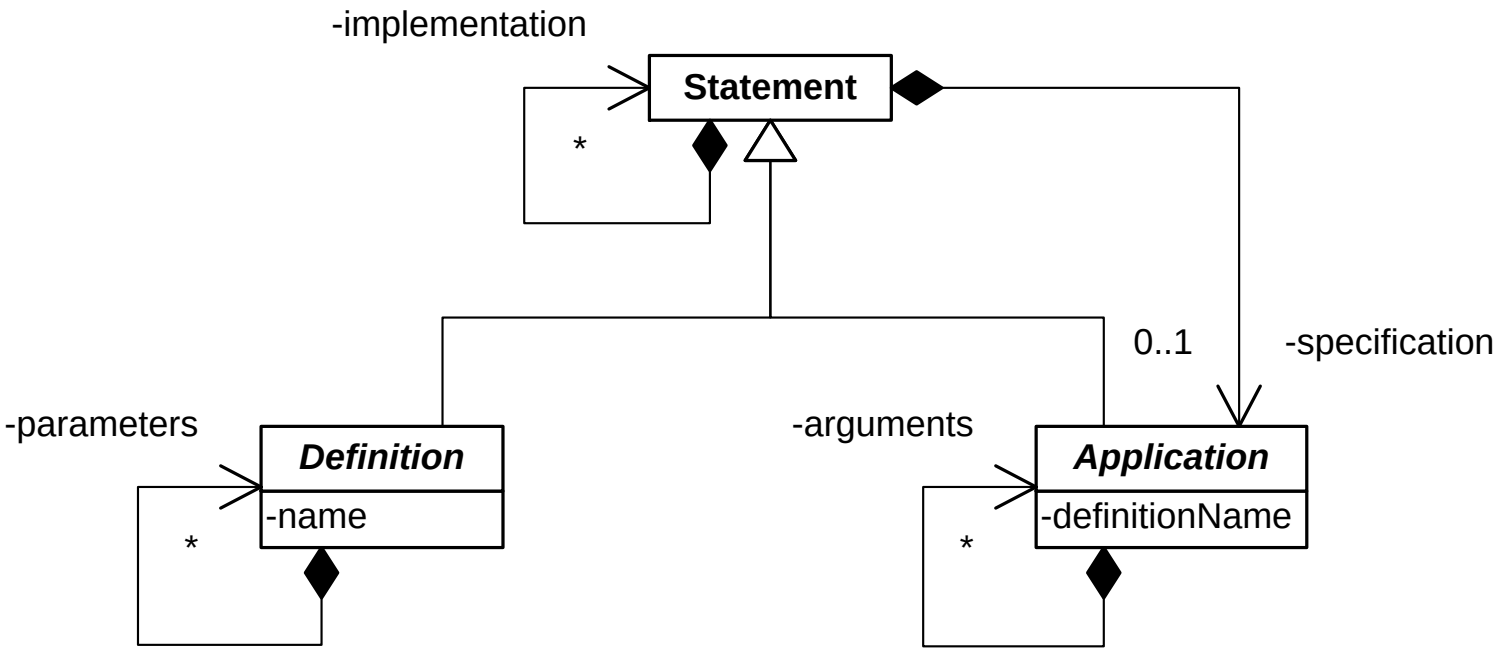
- Merkmale:
  - Minimale Syntax
  - Unterstützt die vier grundlegenden Differenzierungsaspekte
  - Erlaubt Metaprogrammierung
  - Ist zustandsbehaftet
  - Strenger Fokus auf Anwendungsentwicklung
  - Eingebettet in SimPL-Kontext

Strenger Fokus auf Anwendungsentwicklung

- SimPL unterstützt nichts, was Rücksicht auf Hardware nimmt.
- Anders formuliert: SimPL nimmt an, dass die Hardware auf der es läuft, unendlich schnell und immer verfügbar ist.
- Deshalb:
  - Keine Referenzen/Zeiger, nur Werte
  - Keine Nebenläufigkeiten
  - Keine Persistenz
  - Keine Interoperabilität

- Jeglichen Hardwarebezug übernimmt der SimPL-Kontext.
- Die Reduzierung syntaktischer Möglichkeiten bietet ein Mehr an Performanceoptimierung durch den SimPL-Kontext.

Syntax


-implementation
Statement
*
0..1
-specification
-parameters
Definition
-name
*
-arguments
Application
-definitionName
*

Syntax

- Ein SimPL-Programm besteht aus Anweisungen.
- Sie können Definitionen oder Applikationen sein.
- Definitionen besitzen zu ihrer Identifizierung einen Namen.
- Applikation beschreiben Referenzen auf Definitionen.
- Applikationen könnten Argumente, Definition können Parameter haben.
- Definitionen ohne Parameter und ohne Implementierung stellen Variablen dar.
- Es werden drei Klassen von Namen unterschieden: Zahlen, Zeichenketten und sonstige Namen.

Syntax

- Die Syntax von Zahlen ist wie folgt:

```
Sign ::= '+' | ‘-’
Separator ::= '.'
PosDigit ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit ::= '0' | PosDigit
NumberLiteral ::= Sign? (PosDigit Digit* | '0') ( Separator Digit+ )?
```

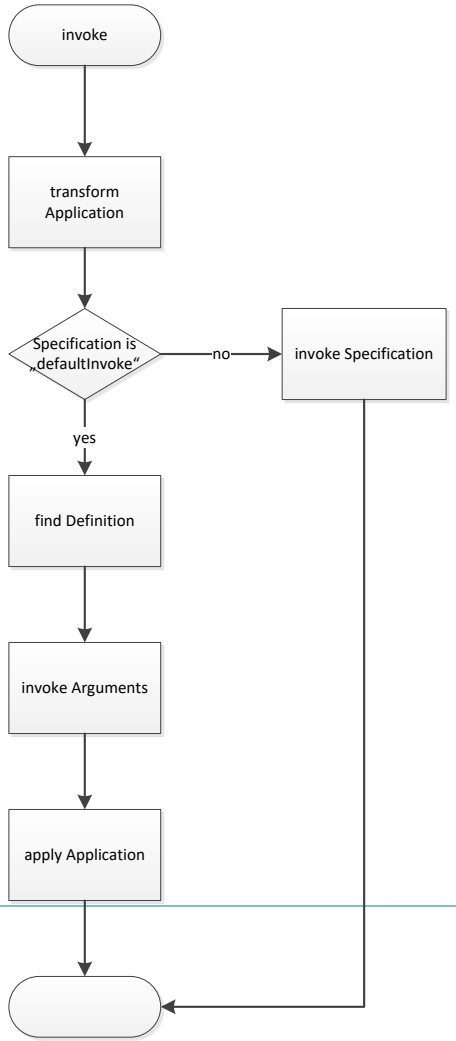
- Zeichenketten werden durch Anführungszeichen abgegrenzt.
- Sonstige Namen beginnen mit einem Klein- oder Großbuchstaben, einem Unterstrich oder Dollarzeichen, gefolgt von einer beliebigen Anzahl Zeichen derselben Grundmenge angereichert mit Ziffern.

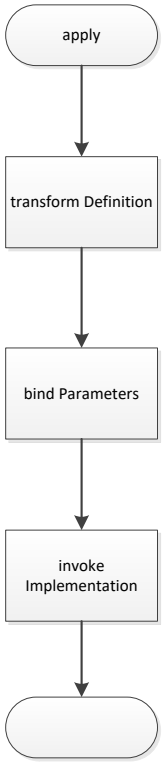
SimPL-Interpreter

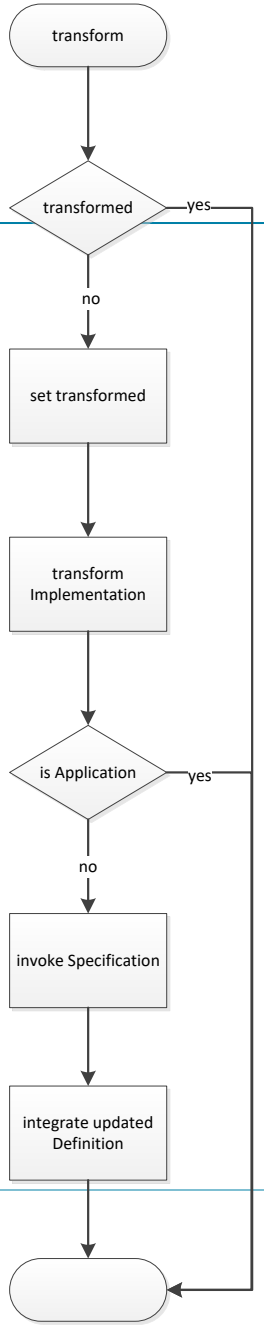
- Die Interpretation einer Applikation glieder sich im Wesentlichen in drei Schritte:
  - Aufruf
  - Anwendung
  - Transformation

- Startpunkt eines SimPL-Programms ist ein Aufruf.

SimPL-Interpreter







SimPL-Interpreter

- Applikationen werden durch ihre Spezifikation ausgeführt, außer diese ist „defaultInvoke“. Dann erfolgt standardmäßige Ausführung.
- Für Definition bedeutet die Berücksichtigung der Spezifikation, dass sie transformiert wird durch Anwendung der Spezifikation.
- Die Spezifikation hat Zugriff auf ihre Anweisung per `instance`-Variable.

Variablen und Werte

- Definitionen ohne Implementierung und ohne Parameter sind Variablen.
- Variablen sind Wertetypen. Es gibt also keine Nebeneffekte.
- Variablenwerte bleiben nach der Anwendung einer Definition erhalten. Sie verhalten sich daher im Sinne von Java eher wie Attribute einer Klasse als lokale Variablen.

- Definitionen haben keine expliziten Rückgabewert. Stattdessen erlauben sie nach Anwendung den Zugriff auf die (ggf. veränderten) Variablenwerte.
- In diesem Sinne erlauben Definitionen sowohl beliebig viele Eingabe- als auch Rückgabewerte.

Elementare Definitionen

- Elementare Definition sind wie folgt gegliedert:
  - Typen
  - Operationen
  - Ablaufsteuerung
  - Metaprogrammierung

Typen

- `null` ist der Initialwert aller Variablen.
- `Boolean` bietet die Grundlage für logische Ausdrücke. Entsprechende Instanzen können die Werte `true` und `false` annehmen. Der Typ hat keine Operationen.
- `Number` ist der (einzige) elementare numerische Typ. Einzige Operation des Typs ist `negate`. Dadurch lässt sich ein Zahlenwert negieren.
- String ermöglich die Verarbeitung von Zeichenketten. Operationen sind:

`substr(Number begin, Number end) : String value`	Bestimmt einen Ausschnitt einer Zeichenkette
`contains(String s) : Boolean value, Number pos`	Prüft, ob eine Zeichenkette in dieser enthalten ist
`length : Number value`	Länge einer Zeichenkette

Typen

- `List` ist der elementare Typ für Collections.
- List-Elemente sind geordnet.
- Operationen:

`Number size`	Anzahl der Elemente der Liste
`append(elem)`	Fügt ein Element am Ende der Liste hinzu
`insertAt(Number pos, elem)`	Fügt ein Element an einer bestimmten Position ein
`set(Number pos, elem)`	Ersetzt ein Element an einer bestimmten Position
`get(Number pos) : value`	Liest ein Element an einer bestimmten Position aus
`remove(Number pos)`	Entfernt ein Element an einer bestimmten Position
`clear`	Entfernt alle Elemente aus der Liste

Operationen

- Unter den elementaren Operationen werden logische, arithmetische, Zeichenketten- und IO-Operationen unterschieden.
- Logische Operationen:

`not(Boolean val) : Boolean value`	Not verneint einen logischen Ausdruck.
`and(Boolean val1, Boolean val2) : Boolean value`	And („Und-Verknüpfung“) und or („Oder-Verknüpfung“) ermöglichen logische Verknüpfungen.
`or(Boolean val1, Boolean val2) : Boolean value`	
`eq(val1, val2) : Boolean value`	eq („equal“) und ne („not equal“) vergleichen inhaltlich beliebige Werte. Gleichheit bedeutet gleiche Struktur und gleiche elementare Werte.
`ne(val1, val2) : Boolean value`	
`gt(val1, val2) : Boolean value`	gt („greater than“), lt („lower than“), ge („greater equal“) und le („lower equal“) erlauben Vergleiche. Sind die Argumente zwei Zahlen, erfolgt ein numerischer Vergleich, sind es zwei Zeichenketten ein lexikografischer Vergleich.
`lt(val1, val2) : Boolean value`	
`ge(val1, val2) : Boolean value`	
`le(val1, val2) : Boolean value`	

Operationen

- Zeichenketten-Operationen:

`concat(String val1, String val2) : String value`	Verknüpft zwei Zeichenketten
`trim(String val) : String value`	Entfernt Whitespaces an Beginn und Ende einer Zeichenkette
`toUpperCase(String val) : String value`	Wandelt Kleinbuchstaben einer Zeichenkette in Großbuchstaben um
`toLowerCase(String val) : String value`	Wandelt Großbuchstaben einer Zeichenkette in Kleinbuchstaben um

- I/O-Operationen:

`print(val)`	Gibt val auf Konsole aus
`printInstance(Statement val)`	Gibt die Instanz von val aus, wobei als val eine Definition zur Beschreibung einer Anweisung erwartet wird

Ablaufsteuerung

- Applikationen werden standmäßig der Reihe nach ausgeführt in der sie deklariert sind.
- Neben diesem Grundverhalten bietet SimPL eine if-Anweisung für Alternationen, eine repeat-Anweisung für Iterationen und eine try-Anweisung für Ausnahmebehandlungen.
- If-Anweisungen ermöglichen Verzweigungen im Programmablauf anhand eines logischen Ausdrucks. Ist dieser wahr, wir der then-Block ausgeführt, andernfalls der else-Block.
- Die Implementierung einer repeat-Anweisungen wird wiederholt ausgeführt. Eine solche Schleife kann durch eine break-Anweisung abgebrochen werden. Darüber hinaus ermöglicht in Schleifen eine continue-Anweisung den Abbruch der aktuellen Iteration. Damit beginnt der Durchlauf von Beginn.

Ablaufsteuerung

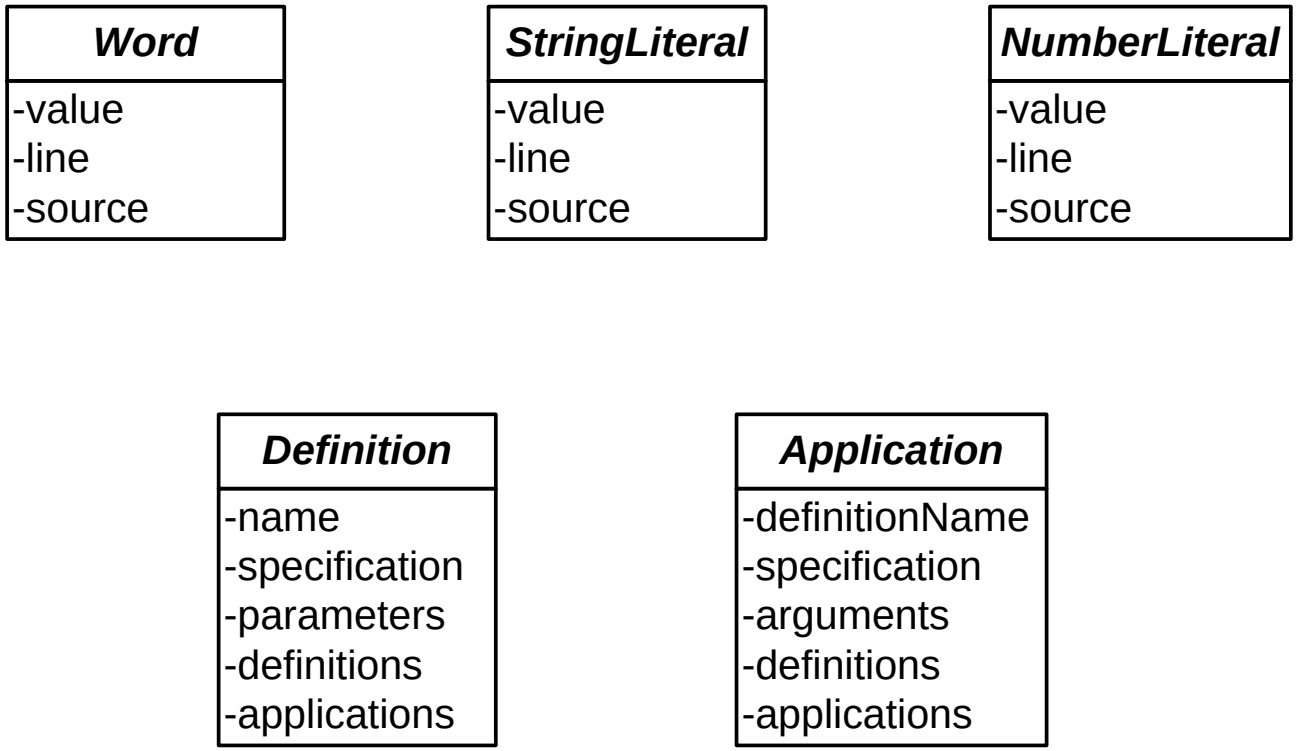
- Try-Anweisungen erlauben in Kombination mit throw-Anweisungen Exception-Handling.
- Eine try-Anweisung besitzt einen that-Block und einen or-Block.
- Der that-Block wird sequenziell ausgeführt. Wird in der Folge eine throw-Anweisung ausgeführt, wird der Programmfluss im or-Block fortgesetzt. Ausgeführt wird darin lediglich ein passender catch-Block.
- Passend bedeutet hierbei passend zum Argument der throw-Anweisung, zur Exception.
- Wenn kein passender catch-Block definiert ist, wird die Exception weitergeworfen.
- Gibt es überhaupt keinen passenden catch-Block in der Aufrufkette, führt dies zum Programmende.

Ablaufsteuerung

- Zudem erlaubt eine return-Anweisung den Abbruch der Ausführung einer Definition.

Metaprogrammierung

- Folgende Metatypen sind definiert:



Metaprogrammierung

- Zur Metaprogrammierung stehen folgende Variablen bereit:

`instance`	Variable, die implizit Spezifikationen für den Zugriff auf die zugehörige Anweisungsbeschreibung zur Verfügung gestellt wird. Eine Manipulation bei der impliziten, regulären Ausführung der Spezifikation einer Definition führt zur entsprechenden Manipulation dieser Definition.
`controlState`	Ermöglicht Zugriff auf den Steuerstatus, um den Programmablauf zu beeinflussen. Next: es folgt die nächste Anweisung Return: Ende der Definition Break: Abbruch der Applikationsimplementierung Continue: Neubeginn der Applikationsimplementierung Throw: Fortsetzung der Anwendung beim passenden catch.

- Sie stehen bereit in Spezifikationen.

Metaprogrammierung

- Zudem gibt es folgende Operationen.
- Dabei ist specof() insofern speziell, als dass das Argument val nicht im Vorfeld ausgeführt wird.

Metaprogrammierung

`defaultInvoke`	Führt als Spezifikation zu einem finalen Standardaufruf
`context(Statement s) : Statement value`	Ermittelt die Beschreibung des Kontextes auf Basis der Beschreibung einer Anweisung.
`term(Application a, Definition d) : instance`	term() führt zu einer finalen Anwendung auf Basis einer Applikationbeschreibung a und der zugehörigen Definitionbeschreibung d.
`invoke(Application a) : instance`	invoke() initiiert einen Aufruf auf Basis einer Applikationsbeschreibung. Beide Funktionen erlauben den Zugriff auf die angewendete Definitionsinstanz.
`report(type, message, context)`	Realisiert die Kommunikation mit der Entwicklungsumgebung während einer Transformation. type ist der Berichtstyp. Unterschieden werden Error, Warning und Info. Ersterer führt zum Abbruch der Anwendung, nachdem die aktuelle Transformation abgeschlossen ist. message erwartet eine Textnachricht. context hält die Anweisungsbeschreibung, auf die sich die Nachricht bezieht.
`specof(val) : Specification value`	Ermöglicht den Zugriff auf die Spezifikation einer referenzierten Definition.

Notation

- SimPL ist als Zwischensprache definiert.
- Sie hat natürlich auch eine Notation, um mit ihr direkt zu programmieren.

```
Name ::= NumberLiteral | StringLiteral | Word

Main ::= Block
Block ::= Statement NEWLINE Statement*
Statement ::= Application | Definition

Application ::= QSimpleApplication Implementation?
QSimpleApplication ::= (SimpleApplication '.')* SimpleApplication
SimpleApplication ::= Name Arguments?
Arguments ::= '(' Argument (',' Argument)* ')'
Argument ::= SimpleApplication

Definition ::= SimpleDefinition Implementation?
SimpleDefinition ::= 'def' Specification ':' Word Parameters?
Parameters ::= '(' Parameter (',' Parameter)* ')'
Parameter ::= Specification ':' Name

Specification ::= SimpleApplication
Implementation ::= (NEWLINE INDENT Block DEDENT)?
```

Beispiel 1: „Hello World“

```
print("Hello World!")
```

Beispiel 2: Fibonaccifolge

```
def null: fib(null: x)
   def null: value
   if (or(eq(x, 0).value, eq(x, 1).value).value)
      then
         value(x)
         return
   value(add(fib(sub(x, 1).value).value, fib(sub(x, 2).value).value).value)

def null: i
i(0)

repeat
   if (eq(i, 10).value)
      then
         break
   print(fib(i).value)
   i(add(i, 1).value)
```

Beispiel 3: Vererbung

```
def Class(null): A
   def null: x
   print("create A")

   def null: sayHello
      print("Hello A")

def Class(specof(A).value): B
   def null: y
   print("create B")

   def null: sayHello
      print("Hello B")

def null: p
p(B)
print(p.y(3))
print(p.x(5))
p.sayHello
```

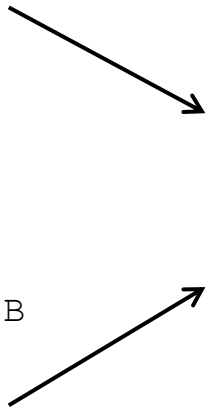
Ausgabe:

```
"create A"
"create B"
3
5
"Hello B"
```

Beispiel 3: Vererbung

```
def Class(null): A

   print("create A")

   def null: x
   def null: sayHello
      print("Hello A")
```

```
def Class(specof(A).value): B

   print("create B")

   def null: y
   def null: sayHello
      print("Hello B")
```



```
def Class(specof(A).value): B

   print("create A")
   print("create B")

   def null: x
   def null: y
   def null: sayHello
      print("Hello B")
```

Beispiel 3: Vererbung

```
def null: Class(null: super)

   if (eq(super, null).value)
      then
         return

   def null: superAppIdx
   superAppIdx(0)

   repeat
      if (eq(superAppIdx, super.instance.applications.size).value)
         then
            break

      instance.applications.insertAt(superAppIdx, \
       super.instance.applications.get(superAppIdx).value)

      superAppIdx(add(superAppIdx, 1).value)
   …
```

Beispiel 3: Vererbung

```
…
def null: superDefIdx
superDefIdx(0)

def null: insDefIdx
insDefIdx(0)

repeat
   if (eq(superDefIdx, super.instance.definitions.size).value)
      then
         break

   def null: defIdx
   defIdx(insDefIdx)

   def null: defFound
   defFound(false)
   …
```

Beispiel 3: Vererbung

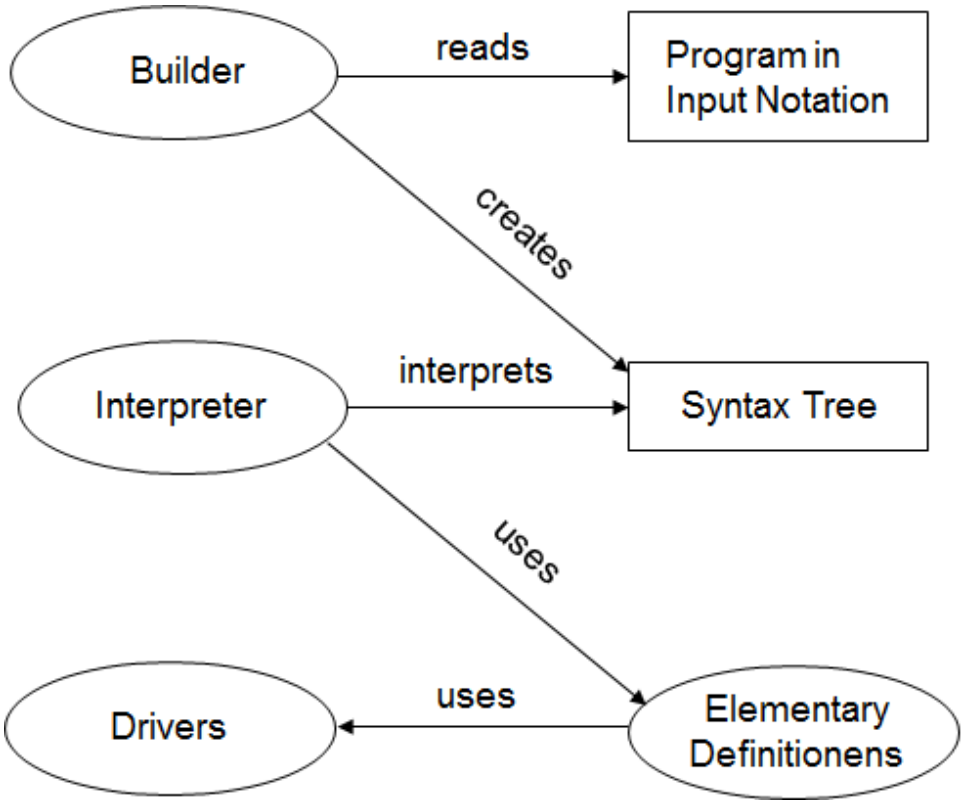
```
…
repeat
   if (eq(defIdx, instance.definitions.size).value)
      then
         break

   if (eq(super.instance.definitions.get(superDefIdx).value.name.value, \
            instance.definitions.get(defIdx).value.name.value).value)
      then
         defFound(true)
         break

   defIdx(add(defIdx, 1).value)

if (not(defFound).value)
   then
      instance.definitions.insertAt(insDefIdx, \
       super.instance.definitions.get(superDefIdx).value)

      insDefIdx(add(insDefIdx, 1).value)

superDefIdx(add(superDefIdx, 1).value)
```

SimPL-Kontext

- SimPL-Anwendungen sind in einen SimPL-Kontext eingebettet.
- Dieser folgt einer 3-Schichtenarchitektur:

- Die Anwendungssprache wird in SimPL übersetzt.
- Treiber erlauben Anbindung an die Umgebung.

SimPL-Kontext



SimPL-Kontext

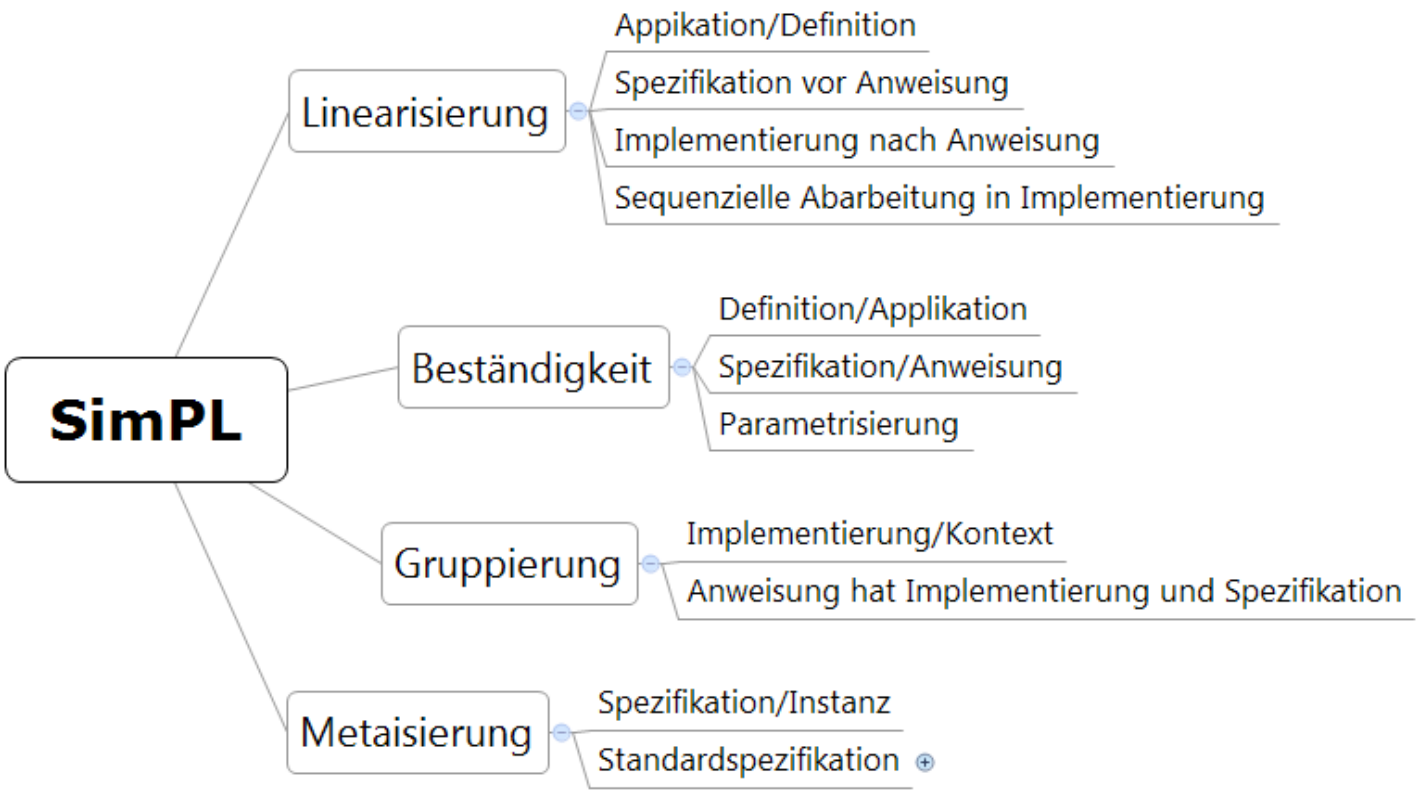
- In der Programmierung ist dieser Ansatz nicht unüblich.
- Z.B. basiert die Programmiersprache Java und andere auf dem Java-Bytecode, welcher die Hardware abstrahiert.
- Bei Microsoft .NET wird die mittlere Schicht als MSIL (Microsoft Intermediate Langage) bezeichnet.
- Allerdings sind diese Mittelschichten relativ komplex, heterogen und unterstützen Metaisierung kaum.

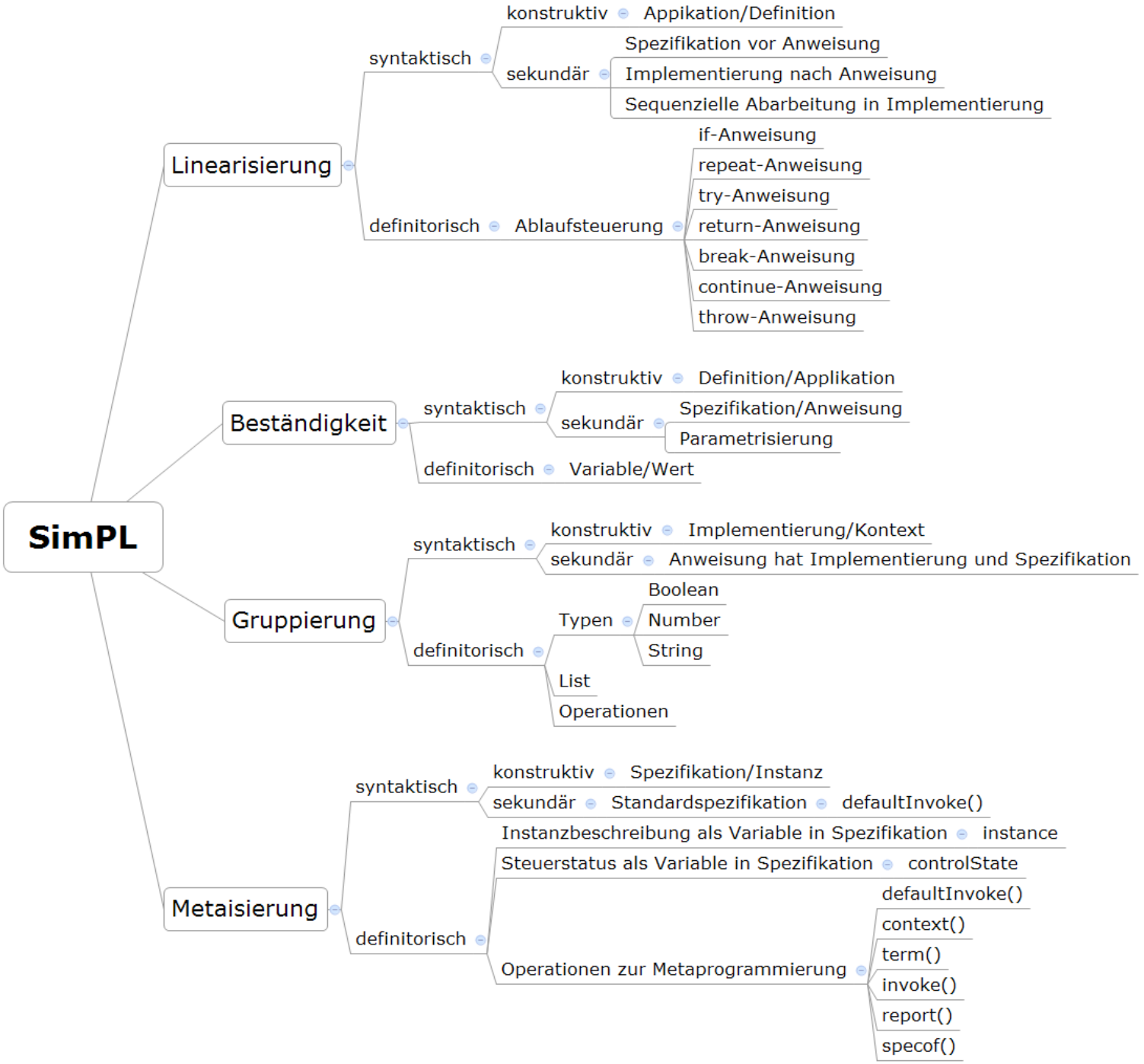
SimPL-Kontext

- Der SimPL-Kontext kann verteilt sein.
- Treiber und Anwendungsteile sind dynamisch adaptierbar.

- Somit kann eine immerwährende SimPL-Anwendung eine Art Internet darstellen, wobei Kommunikation durch Metaprogrammierung kontrollierbar wird.

Aspekte zur Komplexität




SimPL
Linearisierung
syntaktisch
konstruktiv
Appikation/Definition
sekundär
Spezifikation vor Anweisung
Implementierung nach Anweisung
Sequenzielle Abarbeitung in Implementierung
definitorisch
Ablaufsteuerung
if-Anweisung
repeat-Anweisung
try-Anweisung
return-Anweisung
break-Anweisung
continue-Anweisung
throw-Anweisung
Beständigkeit
syntaktisch
konstruktiv
Definition/Applikation
sekundär
Spezifikation/Anweisung
Parametrisierung
definitorisch
Variable/Wert
Gruppierung
syntaktisch
konstruktiv
Implementierung/Kontext
sekundär
Anweisung hat Implementierung und Spezifikation
definitorisch
Typen
Boolean
Number
String
List
Operationen
Metaisierung
syntaktisch
konstruktiv
Spezifikation/Instanz
sekundär
Standardspezifikation
defaultInvoke()
definitorisch
Instanzbeschreibung als Variable in Spezifikation
instance
Steuerstatus als Variable in Spezifikation
controlState
Operationen zur Metaprogrammierung
defaultInvoke()
context()
term()
invoke()
report()
specof()

Möglichkeiten der Metaprogrammierung

- Direkte Unterstützung von:
  - Codegenerierung
  - Modelltransformationen
  - Modellvalidierung, (statische) Typsicherheit
  - Objektorientierter Programmierung
  - Aspektorientierter Programmierung
  - Subjektorientierter Programmierung
  - Eventgetriebener Programmierung
  - Domänenspezifischer Sprachen