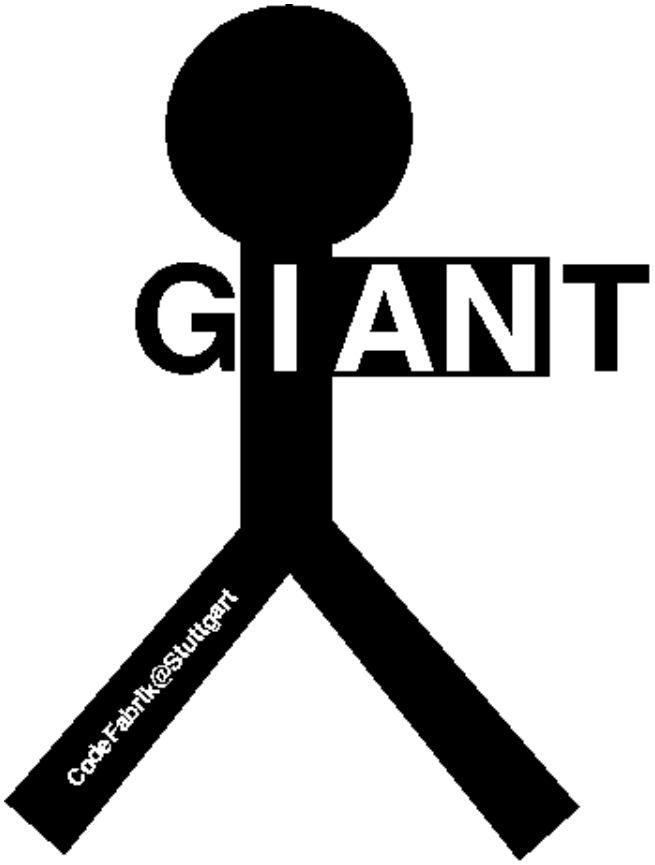
Universität Stuttgart
Studienprojekt A – IML Browser
CodeFabrik@Stuttgart

GIANT Scripting Language Spezifikation

Version 1.0



Inhaltsverzeichnis

1. GIANT Scripting Language

In diesem Dokument wird die GIANT Scripting Language (GSL) sowie Interpreter zur Ausführung von GSL Scripts spezifiziert. GSL ist eine Skriptsprache für das Software-System GIANT. Mit Hilfe von GSL können Anfragen an die IML-Bibliothek gestellt werden und auf den Resultaten Aktionen ausgeführt werden.

Hier wird zunächst die Struktur von IML-Graphen aus der Sicht von GIANT beschrieben und danach Syntax und Sematik von GSL definiert. Schließlich folgt die vollständige Beschreibung der angebotenen Sprachumgebung.

1.1. IML-Graphen

Der Kunde stellt die Reflektion zur Verfügung. GIANT verwendet diese Bibliothek, um auf IML-Graph Dateien zuzugreifen.

1.1.1. IML-Datenbasis

In jeder IML-Graph Datei sind IML-Daten enthalten. Diese bestehen aus

1. IML-Knoten
2. Attributen von IML-Knoten
3. nichts sonst.

Es gibt in jeder IML-Graph Datei genau einen besonderen IML-Knoten, der Wurzelknoten genannt wird.

1.1.1.1. Attribute von IML-Knoten

Jeder IML-Knoten hat einen Typ. Für jeden Typ von IML-Knoten existiert ein String, der den Typ eindeutig identifiziert.

Jeder IML-Knoten verfügt über eine endliche Folge von Attributen. Diese Folge definiert eine endliche Anzahl einzelner Attribute, die der IML-Knoten besitzt, sowie eine Reihenfolge dieser Attribute. Weder die Folge von Attributen noch die in der Folge enthaltenen Attribute eines IML-Knoten können sich während der Laufzeit von GIANT verändern.

Jedes Attribut besitzt einen Attribut-Namen. Kein IML-Knoten besitzt zwei verschiedene Attribute, die einen gleichen Namen besitzen.

Einige Attribute sind Verweise. Sie verweisen auf höchstens einen IML-Knoten. Es werden unterschieden:

genutzte Verweise Verweise, die auf genau einen IML-Knoten verweisen.

ungenutzte Verweise Verweise, die auf einen besonderen Wert verweisen, der kein IML-Knoten ist.

Jedes Attribut eines IML-Knoten gehört genau einer der folgenden Klassen an:

Einfache Attribute : Haben einen bestimmten Wert. Mögliche Typen dieses Werts sind:

1. Source Location
   a) Zeilennummer (Natural)
   b) Spaltennummer (Natural)
   c) Filename (String)
   d) Pfadname (String)
2. Boolean
3. Natural
4. String
5. Folge von Strings

Genutztes Verweis-Attribut Ein genutzter Verweis

Ungenutztes Verweis-Attribut Ein ungenutzter Verweis

Verweisfolgen-Attribut Eine endliche Folge von genutzten Verweisen

Verweismengen-Attribut Eine endliche Menge von genutzten Verweisen

1.1.2. IML-Graphen

Die IML-Daten jeder IML-Graph Datei definieren einen zugehörigen IML-Graph. Sprechweise: Die IML-Daten liegen dem IML-Graph zugrunde. Ein IML-Graph ist ein gerichteter knoten- und kantenannotierter Graph mit Schlingen und Mehrfachkanten. Er ist definiert durch die folgende Vorschrift:

1. Ein IML-Graph besitzt als Knotenmenge die Menge aller IML-Knoten der zugrunde liegenden IML-Daten. Der IML-Graph besitzt keine weiteren Knoten.
2. Annotationen eines IML-Knoten sind alle Attribute des IML-Knoten.
3. Eine Kante im IML-Graph heißt IML-Kante. Eine IML-Kante e von einem IML-Knoten v zu einem IML-Knoten w des selben IML-Graph existiert genau dann, wenn eine der folgenden Bedingungen erfüllt ist:
   a) v besitzt ein Genutztes Verweis-Attribut, das auf w verweist.

i. Dann besitzt e als Annotation den Attribut-Namen s des Genutzten Verweis-Attributs.

ii. Das Paar (v, s) heißt Typ der IML-Kante e

b) v besitzt ein Verweisfolgen-Attribut f. Ein genutzter Verweis aus f verweist auf w.

i. dann besitzt e als Annotation den Attribut-Namen s von f sowie die Nummer innerhalb der Folge f des genutzten Verweises auf w.

ii. Das Paar (v, s) heißt Typ der IML-Kante e.

c) v besitzt ein Verweismengen-Attribut m. Ein genutzter Verweis aus m verweist auf w.

i. dann besitzt e als Annotation den Attribut-Namen s von m.

ii. Das Paar (v, s) heißt Typ der IML-Kante e.

4. Falls eine IML-Kante e existiert, so hat e keine Annotationen, außer den in Punkt 3 genannten.

1.2. GSL Interpreter

Ein GSL Interpreter ist ein System, das GSL Ausdrücke auswertet, gemäß der Sprachdefinition in Kapitel 1.4. Jeder GSL Interpreter muss die vordefinierte Sprachumgebung nach Kapitel 1.5 unterstützen.

1.2.1. Start einer Auswertung

Beim Start des GSL Interpreters erzeugt dieser zunächst die erste Aktivierungsumgebung $Standard$. Dann wertet der GSL Interpreter den Ausdruck aus, der in der Datei „standard.gsl“ gespeichert ist. Dieser Ausdruck initialisiert die Sprachumgebung, die in Kapitel 1.5.3 beschrieben wird.

Danach ist der GSL Interpreter bereit und ihm kann ein beliebiger GSL Ausdruck zur Auswertung übergeben werden. In GIANT kann dies geschehen, indem der Ausdruck als Kommandozeilenparameter angegeben wird oder indem ein Ausdruck in das dafür vorgesehene Anzeigefenster eingegeben wird.

Dieser Ausdruck kann gespeicherte GSL Scripts aufrufen und mit GIANT kommunizieren.

1.2.2. Kontext des Interpreters

Der GSL Interpreter kann im Kontext eines GIANT-Anzeigefensters ausgeführt werden. Ist dies der Fall, so können die ausgewerteten GSL Ausdrücke Aktionen in diesem Anzeigefenster durchführen.

Wird der GSL Interpreter nicht im Kontext eines Anzeigefensters ausgeführt, so können die GSL Ausdrücke keine Aktion in einem Anzeigefenster auslösen.

Ein GSL Script kann den GSL Interpreter anweisen in den Kontext eines bestimmten Anzeigefensters zu wechseln.

1.2.3. Verhalten im Fehlerfall

1.2.3.1. Syntaxfehler

Der GSL Interpreter muss einen Ausdruck zurückweisen, falls eine Forderung der Sprachdefinition durch diesen Ausdruck verletzt wird. In diesem Fall muss der Interpreter dem Benutzer eine Fehlermeldung anzeigen. Andernfalls startet der GSL Interpreter die Auswertung des GSL Ausdrucks.

1.2.3.2. Laufzeitfehler

Der GSL Interpreter muss die Ausführung sofort stoppen, sobald ein Laufzeitfehler auftritt. Dies kann nur dann geschehen und muss jedes Mal geschehen, wenn die Semantikdefinition es vorschreibt. Der GSL Interpreter darf die Ausführung bereits nach einem früheren Ausführungsschritt abbrechen, wenn sichergestellt ist, dass nach einem späteren Ausführungsschritt ein Laufzeitfehler auftreten wird. Der GSL Interpreter zeigt dem Benutzer eine Fehlermeldung an.

Der GSL Interpreter muss nach einem Fehler entweder alle seit Beginn der Ausführung bereits ausgeführten Aktionen rückgängig machen, oder er muss dem Benutzer anzeigen, welche Aktionen bereits ausgeführt wurden, bevor der Fehler passierte. Anhand dieser Information soll es dem Benutzer möglich sein, die Aktionen manuell rückgängig zu machen.

1.3. Konzepte

Der Entwurf von GSL legt das größte Gewicht auf eine einfache Erweiterbarkeit der Ausdrucksmächtigkeit der Sprache. Durch Hinzufügen von zusätzlichen vordefinierten Scripts können der Sprache neue Fähigkeiten hinzugefügt werden ohne dabei Änderungen an der Syntax vornehmen zu müssen. Diesem wichtigsten Ziel wurden Aspekte wie eine intuitiv erfassbare Syntax untergeordnet.

1.3.1. Werte und Typen

1.3.1.1. Vordefinierte Typen

GSL verarbeitet Werte verschiedener Typen. Werte können das Ergebnis der Auswertung von Ausdrücken sein. Werte können in Objekten gespeichert werden.

Nachfolgend wird eine Liste aller Typen sowie der Werte angegeben, die ein Ausdruck des entsprechenden Typs annehmen kann:

Node_Id Verweis auf einen einzelnen Knoten.

Edge_Id Verweis auf eine einzelne Kante.

Node_Set Endliche Menge von Werten des Typs Node_Id.

Edge_Set Endliche Menge von Werten des Typs Edge_Id.

String Endliche Folge von Zeichen.

Boolean Der Wahrheitswerte **false** und **true**

Natural Natürliche Zahl, innerhalb eines bestimmten Bereichs, der von der Reflektion vorgegeben wird.

List Endliche Folge von Werten. Jeder dieser Werte ist ein Wert eines beliebigen Typs aus dieser Aufzählung oder **null**.

Var_Reference Zugriffspfad auf ein Objekt.

Script_Reference Aktivierungsinformation für ein bestimmtes Script.

Zusätzlich gibt es noch einen besonderen Typ, der keinen Namen besitzt. Dieser anonyme Typ umfasst genau einen Wert, nämlich den Wert **null**.

1.3.1.2. Zusammengesetzte Typen

Werte zusammengesetzter Typen sind zusammengesetzt aus mehreren Werten der vordefinierten Typen. In dieser Spezifikation wird im Folgenden nicht mehr zwischen vordefinierten und zusammengesetzten Typen unterschieden.

Object_Set Wert des vordefinierten Typs List mit genau zwei Einträgen N und E. Dabei muss N ein Wert des Typs Node_Set und E ein Wert des Typs Edge_Set sein.

1.3.2. Ausdrücke

Die Ausführung von GSL geschieht ausschließlich durch das Auswerten von Ausdrücken. Die Auswertung eines Ausdrucks hat als Ergebnis stets genau einen bestimmten Wert und kann Nebeneffekte haben. Nebeneffekte sind Veränderungen an den Werten von Objekten oder die Durchführung von Aktionen.

Der Typ eines Ergebnis steht im Allgemeinen erst fest, nachdem ein Ausdruck ausgewertet wurde. Die Auswertung eines Ausdrucks kann fehlschlagen, falls ein Laufzeitfehler auftritt. In diesem Fall stoppt die Auswertung sofort.

1.3.3. Variablen und Objekte

Um während der Auswertung von Ausdrücken bestimmte Werte zur späteren Wiederverwendung zu speichern werden *Objekte* verwendet. Der Zugriff auf ein Objekt erfolgt durch eine Variable.

In *Aktivierungsumgebungen* werden Variablen an Objekte gebunden.

Variablen können *inspiziert* werden. Das Ergebnis einer Variableninspektion ist der Wert des Objekts, an das die Variable gebunden ist.

Eine *Referenz* einer Variable kann genommen werden. Diese Referenz ist ein Zugriffspfad auf das Objekt an das die Variable gebunden ist. Der Zugriffspfad wird benötigt, um den Wert des Objekts zu verändern.

Eine Variable kann entweder sichtbar oder unsichtbar sein. Siehe hierzu Kapitel 1.3.4.3.

Jedes Objekt kann einen Wert w_1 jedes beliebigen Typs t_1 aus 1.3.1 annehmen. Durch eine weitere Zuweisung an das Objekt kann das Objekt einen anderen Wert w_2 des Typs t_2 annehmen. Dabei dürfen die beiden Werte aus verschiedenen Typen sein (dh. $t_1 \neq t_2$ ist zulässig).

1.3.4. Aktivierungsumgebungen

1.3.4.1. Standard-Aktivierungsumgebung

Eine Aktivierungsumgebung bindet Variablen an Objekte. Beim Start des GSL Interpreters existiert eine erste Aktivierungsumgebung $Standard$.

In $Standard$ existiert für jeden zulässigen Namen `<Name>` eines IML-Teilgraphen in GIANT eine Variable „**subgraph . `<Name>`**“. (Es gibt also aufzählbar unendlich viele solche Variablen.) Jede dieser Variablen ist an den IML-Teilgraphen mit dem entsprechenden Namen `<Name>` gebunden, sofern dieser IML-Teilgraph existiert. Existiert der IML-Teilgraph nicht, so ist die Variable an ein besonderes Objekt gebunden, das den Wert **null** hat.

Falls der GSL Interpreter im Kontext eines Anzeigefensters W ausgeführt wird, dann existiert in $Standard$ für jeden zulässigen Namen `<Name>` einer Selektion im Fenster W eine Variable „**selection . `<Name>`**“. (Es gibt also aufzählbar unendlich viele solche Variablen.) Jede dieser Variablen ist an die Selektion mit dem entsprechenden Namen `<Name>` gebunden, sofern diese Selektion existiert. Existiert die Selektion nicht, so ist die Variable an ein besonderes Objekt gebunden, das den Wert **null** hat. Wird der GSL Interpreter nicht im Kontext einen Anzeigefensters ausgeführt, so existiert keine dieser Variablen.

Die erste Aktivierungsumgebung $Standard$ ist anfangs *aktuell*. Der Ausdruck aus der Datei „standard.gsl“ wird ausgewertet. Dieser Ausdruck kann z.B. neue Objekte erzeugen und in $Standard$ Variablen an diese Objekte binden. Auf diese Weise wird die Sprachumgebung erstellt, die in Kapitel 1.5.3 beschrieben wird. Durch Änderungen an der Datei „standard.gsl“ kann die Sprachumgebung an neue Anforderungen angepasst werden. Die Implementierung des GSL Interpreters muss dazu nicht bekannt sein.

1.3.4.2. Vererbung von Aktivierungsumgebungen

Wann immer der GSL Interpreter die Auswertung eines benutzerdefinierten GSL Ausdrucks startet (siehe 1.2.1), wird eine neue Aktivierungsumgebung A_{start} erzeugt. A_{start} *beerbt* dabei $Standard$. A_{start} wird aktuell bevor die Auswertung des benutzerdefinierten Ausdrucks beginnt. Nachdem die Auswertung abgeschlossen ist, wird A_{start} zerstört und $Standard$ wird wieder aktuell.

Falls während der Auswertung eines Ausdrucks die Aktivierung eines Scripts erfolgt, so wird eine neue Aktivierungsumgebung für die Auswertung dieses Scripts erzeugt. Diese neue Aktivierungsumgebung A_{neu} beerbt eine bestimmte Aktivierungsumgebung A_{alt}, die durch die Aktivierungsinformation bestimmt wird (siehe 1.4.7.6). A_{neu} wird aktuell. Nachdem die Auswertung des Script beendet ist wird A_{neu} zerstört und A_{alt} wird wieder aktuell.

Während ein Script ausgewertet wird können weitere Scripts aktiviert werden. Deshalb können mehrere Aktivierungsumgebungen gleichzeitig existieren. Auf diesen Aktivierungsumgebungen existiert eine Vererbungsbeziehung die durch eine partielle Ordnung beschrieben $\leq$ werden kann:

1. Für zwei Aktivierungsumgebungen A_1, A_2 gilt: (falls A_2 beerbt A_1) $\Rightarrow A_1 \leq A_2$
2. Für jede Aktivierungsumgebung A gilt: $A \leq A$
3. Für zwei Aktivierungsumgebung A_1, A_2 gilt: $A_1 \leq A_2 \wedge A_2 \leq A_1 \Rightarrow A_1 = A_2$
4. Für drei Aktivierungsumgebungen A_1, A_2, A_3 gilt: $A_1 \leq A_2 \wedge A_2 \leq A_3 \Rightarrow A_1 \leq A_3$

Anmerkung Weil nach Start des GSL Interpreters eine erste Aktivierungsumgebung $Standard$ erzeugt wird, aufgrund der beschriebenen Vererbung und aufgrund der Transitivität von $\leq$ gilt für jede Aktivierungsumgebung A: $Standard \leq A$.

1.3.4.3. Sichtbarkeit

Annahme: Die Aktivierungsumgebung A_{neu} beerbt die Aktivierungsumgebung A_{alt}. Dann bindet A_{neu} mindestens alle Variablen, die auch A_{alt} bindet. In A_{neu} können zusätzlich

1. weitere Variablen gebunden werden, die in A_{alt} nicht gebunden sind.
2. Variablen, die in A_{alt} gebunden sind, an andere Objekte gebunden werden (überladen).

Eine Variable v heißt *lokal* in A_{neu}, falls sie eine der folgenden Bedingungen erfüllt:

1. Es existiert keine Aktivierungsumgebung A_{alt}, die von A_{neu} beerbt wird oder
2. A_{neu} beerbt A_{alt}. v ist in A_{alt} an ein Objekt o_1 gebunden ist und v ist in A_{neu} an ein Objekt o_2 mit $o_1 \neq o_2$ gebunden oder
3. A_{neu} beerbt A_{alt}. v ist in A_{alt} nicht gebunden, und v ist in A_{neu} gebunden an irgend ein Objekt.

Jede Variable, die in A_{neu} an irgend ein Objekt gebunden ist, heißt *sichtbar* in A_{neu}.

Jede Variable, die in A_{neu} sichtbar ist, aber nicht lokal in A_{neu} ist, heißt *global* in A_{neu}.

Anmerkung In einer Aktivierungsumgebung A_1 sind alle Variablen sichtbar, die in irgend einer Aktivierungsumgebung A_2 mit $A_2 \leq A_1$ sichtbar sind.

Beispiel

```
[ // Beginn einer Sequence

  // die Variable subgraph.xyz ist hier global

  // die Variable union wird in standard.gsl
  // erzeugt und ist deshalb hier global.

  +a; // neue lokale Variable
  +b; // neue lokale Variable

  { (), // neues Script
    [
      // a ist hier global
```

```
      // b ist hier global

      +c; // neue lokale Variable

      +b; // neue lokale Variable, die alte
          // Variable b ist nicht mehr sichtbar.
    ]
  };

  // c existiert hier nicht.

] // Ende der Sequence
```

1.3.5. Aktivierung von Scripts

Ein Script ist ein Paar aus

Parameterliste einer (möglicherweise leeren) Liste von Ausdrücken, deren Ergebnis Zugriffspfade auf Objekte sein müssen.

Rumpf einem Ausdruck.

Beispiel einer Script-Deklaration

```
// Beispiel zur Script-Deklaration
// Zwischen den geschweiften Klammern
// Steht die Parameterliste und
// der Rumpf, getrennt durch ,

{ (+param1, +param2), // Parameterliste
  [                   // Rumpf, hier: Sequence
    action1;
    action2;
  ]                   // Ende der Sequence
}
```

1.3.5.1. Ablauf

Ein Script S kann *aktiviert* werden. Wenn S aktiviert wird, dann wird eine neue Aktivierungsumgebung A_{neu} erzeugt. Es werden die Argumente des Aufrufs an die Parameter von S übergeben. Der Rumpf von S wird ausgewertet. Das Ergebnis der Auswertung von S ist das Ergebnis der Auswertung des Rumpfs von S. Für die genaue Beschreibung siehe 1.4.7.6.

1.3.5.2. Vorraussetzung

Ein Script S wird aktiviert durch Auswertung eines Ausdrucks, der als Ergebnisaus einen Wert des Typs Script_Reference hat. Dieser Wert wird als Aktivierungsinformation bezeichet.

Die Aktivierungsinformation bestimmt

1. Welches Script ausgeführt wird (Quelltext-Position von S oder eine gleichwertige Darstellung).
2. Welche Aktivierungsumgebung von der neuen Aktivierungsumgebung dieser Aktivierung beerbt wird.

Vor der Aktivierung des Scripts S wird eine neue Aktivierungsumgebung A_{neu} erzeugt und danach aktuell. Das zu aktivierende Skript S kann eine in A_{neu} globale Variable v inspizieren, oder die Referenz von v nehmen. Es soll sichergestellt werden, dass die neue Aktivierungsumgebung A_{neu} nur solche Aktivierungsumgebungen A_{alt} beerben kann, so dass die Variable v in A_{alt} genau die Bedeutung hat, die der Autor von S erwartet. Dies stellt Punkt 2 der vorangehenden Aufzählung sicher. A_{alt} ist *nicht* zwangsläuftig die vor der Aktivierung von S aktuelle Aktivierungsumgebung. A_{alt} ist diejenige Aktivierungsumgebung, die zum Zeitpunkt der Auswertung der Script-Deklaration von S 1.4.7.3 aktuell war.

Beispiel zur Scriptaktivierung

```
// Beispiel zur Script-Aktivierung

[ // Beginn einer Sequence

  assign           // action ist eine Variable, die an ein
   (+result,       // Objekt gebunden ist, dessen Wert eine
    action ());    // Aktivierungsinformation ist.
                   // Die Aktivierung wird durch () ausgelöst.
                   // Das Ergebnis wird dem neuen Objekt
                   // zugewiesen, das an die neue Variable
                   // result gebunden ist.

  assign           // Die neue lokale Variable var_name
   (+var_name,     // wird an ein neues Objekt gebunden.
    {(), null});   // Das neue Objekt erhält als Wert eine
                   // Aktivierungsinformation für das
                   // Script {(), null}.
  var_name ();     // Inspektion von var_name und Aktivierung
                   // des referenzierten Scripts

  { (),            // Deklaration eines Scripts...
    null
  } ();            // ...und sofortige Aktivierung.

] // Ende der Sequence
```

1.3.5.3. Parameterübergabe

In dem Ausdruck dessen Auswertung die Aktivierung eines Scripts veranlasst wird eine Liste von Argumenten $L_e = (e_1, ..., e_n)$ angegeben mit $n \in 0, 1, 2, ...$ und $\forall 1 \leq i \leq n : (e_i$ ist ein Ausdruck$)$. Bevor die Aktivierung beginnt wird L_e ausgewertet und hat als Ergebnis den Wert $L_a = (a_1, ..., a_n)$.

Nachdem die Aktivierung begonnen hat, wird die Parameterliste ausgewertet (siehe Abschnitt 1.4.7.6). Das Ergebnis dieser Auswertung muss ein Wert L_p vom Typ Liste sein. Es muss gelten $L_p = (p_1, ..., p_n)$ mit $\forall 1 \leq i \leq n : (p_i$ ist ein Zugriffspfad auf ein Objekt$)$. Andernfalls führt die Auswertung zu einem Laufzeitfehler.

Dann wird für alle i jedem Objekt das durch den Zugriffspfad p_i erreicht wird der Wert a_i zugewiesen. In GSL formuliert wird der Ausdruck `(assign (`p_1`, `a_1`), ..., assign (`p_n`, `a_n`))` ausgewertet und sein Ergebnis verworfen (Hinweis: Dieser Ausdruck ist kein korrekter GSL-Ausdruck, sondern dient nur der intuitiven Erläuterung).

Danach ist die Parameterübergabe abgeschlossen und der Rumpf des Scripts wird ausgewertet.

Richtlinie Für die Parameter p_i sollte jeweils eine neue lokale Variable erzeugt werden, z.B. durch „`+name`“.

Beispiel zur Parameterübergabe

```
[ // Beginn einer Sequence

  assign       // neues Script
   (+my_script,
    { (+a, +b, +c),
      assign (a, add (b, c))
    });        // An den Parameter a muss ein Zugriffspfad
               // auf ein Objekt übergeben werden!
               // An das durch a referenzierte Objekt wird
               // das Ergebnis der Aktivierung von add (b, c)
               // zugewiesen. add berechnet auf Natural-Werten
               // die Summe.

  assign
   (+x, 17);

  my_script    // Aktivierung
   (+result,   // 1. Argument: Referenz auf das neue
               //     Objekt, das an result gebunden ist
    x,         // 2. Argument: Inspektion von x (17)
    13);       // 3. Argument: Integer-Literal (13)

  result;      // jetzt liefert result als Ergebnis 30

] // Ende der Sequence, Ergebnis ist 30
```

1.4. Syntax und Semantik

1.4.1. Notation

Die Syntax von GSL wird als kontextfreie Grammatik beschrieben.

1. Nichtterminale werden in einer serifenlosen Schrift dargestellt (z.B. identifier).
2. Terminalsymbole werden fett gedruckt (z.B. **null**).
3. Eine Regel wird geschrieben als

   linke Seite ::= rechte Seite

   wobei auf der linken Seite genau ein Nichtterminal steht, das bei Anwendung der Regel durch die rechte Seite ersetzt werden darf.
4. Alternativen werden angegeben als *erste Möglichkeit* | *zweite Möglichkeit*
5. Die Klammern ⟨ *Inhalt* ⟩ werden verwendet um ihren Inhalt zusammenzufassen.
6. Ein Teil kann gar nicht, einmal oder beliebig oft expandiert werden, wenn er von einem * gefolgt wird.
7. Ein Teil ist optional, falls er von einem ? gefolgt wird.

Die Beschreibung der Bedeutung einer Regel erfolgt in einer Aufzählung, die unter der Regel steht. In der Beschreibung werden die Nichtterminale aus der rechten Seite der Regel als Platzhalter für ihre Ableitung (also den Terminalstring zu dem sie im Programmtext erweitert sind) verwendet.

1.4.2. Schlüsselwörter und lexikalische Elemente

(	**)**	**{**	**}**
[	**]**	**,**	**.**
;	**’**	**+**	**_**
selection	**subgraph**	**//**	**”**
false	**true**	**null**	****

Es wird zwischen Groß- und Kleinschreibung unterschieden.

1.4.3. Kommentare

In GSL-Dateien können Kommentare eingegeben werden, die auf die Semantik des Programms keinen Einfluss haben. Ein Kommentar beginnt mit // und endet am Ende der Zeile. Alle Zeichen dazwischen gehören zu dem Kommentar.

1.4.4. White Space

Leerzeichen, Tabulatoren, Zeilenumbrüche und Kommentare werden als Trennzeichen verwendet. Zwischen Schlüsselwörtern, den in 1.4.2 aufgeführten Zeichen und identifier dürfen beliebig viele

Trennzeichen geschrieben werden.

1.4.5. Identifier

identifier ::= char ⟨ char | digit | _ ⟩*

1. Der Text eines identifier darf nicht gleich sein wie ein GSL-Schlüsselwort (siehe 1.4.2).
2. Die Groß- und Kleinschreibung eines identifier wird unterschieden (Beispiel: „Name“ ≠ „name“).

1.4.6. Literale

literal ::= boolean_literal | int_literal | string_literal | null_literal

1. Ein literal hat einen bestimmten Typ, und als Ergebnis einen Wert dieses Typs.
2. Typ und Wert eines literal stehen bereits zum Übersetzungszeitpunkt fest. (siehe entsprechendes Nichtterminal der rechten Seite).

1.4.6.1. Boolean-Literal

boolean_literal ::= **false** | **true**

1. Ein boolean_literal ist vom Typ Boolean.
2. Der Wert ist **false**, falls die rechte Seite **false** gewählt wurde.
3. Der Wert ist **true**, falls die rechte Seite **true** gewählt wurde.

1.4.6.2. Integer-Literal

int_literal ::= digit digit*

1. Ein int_literal ist vom Typ Natural.
2. Der Wert ist die Interpretation des Texts im Zehnersystem.

1.4.6.3. String-Literal

string_literal ::= **”** ⟨ **”** | \\ | \n | \t | direct_char ⟩* **”**

Dabei steht direct_char für jedes Zeichen des (implementationsabhängigen) Zeichensatzes außer den Zeichen \ und **”**.

1. Ein string_literal ist vom Typ String.
2. Der Wert ist die ersetzte Folge von Zeichen zwischen den **”**.
3. Die Zeichenfolge **”** wird dabei ersetzt durch ein einfaches **”**.
4. \\ wird ersetzt durch das Zeichen \.

5. \n wird ersetzt durch das Zeichen Zeilenumbruch.
6. \t wird ersetzt durch ein Tabulatorzeichen.

1.4.6.4. Null-Literal

null_literal ::= **null**

1. Der Typ eines null_literal ist der anonyme Typ. Dieser Typ hat keinen Namen.
2. Der Wert ist **null**.

1.4.7. Ausdrücke

expression ::= literal | inspection | reference | script_decl | list | sequence | script_activation

1. expression können zur Laufzeit *ausgewertet* werden.
2. Während der Auswertung kann ein Laufzeitfehler auftreten. In diesem Fall stoppt die Auswertung (siehe 1.2.3.2).
3. Um eine expression auszuwerten wird die entsprechende rechte Seite ausgewertet. Näheres dazu ist der Beschreibung der rechten Seite zu entnehmen.
4. Die Auswertung einer expression kann *Seiteneffekte* haben. Die Seiteneffekte hängen von der gewählten rechten Seite ab.
5. Die Auswertung einer expression hat ein Ergebnis, falls kein Laufzeitfehler aufgetreten ist. Das Ergebnis ist stets ein Wert eines bestimmten Typs. Das Ergebnis ist identisch mit dem Ergebnis der Auswertung der rechten Seite der gewählten Regel. Siehe Beschreibung der rechten Seiten.

1.4.7.1. Inspektion

inspection ::= visible_var | global_var

1. Eine inspection inspiziert den Wert des Objekts, das an die zu inspizierende Variable gebunden ist (siehe 1.3.3).
2. Die verschiedenen rechten Seiten werden verwendet um zwischen Variablen zu unterscheiden, die an GSL-Objekte gebunden sind, und Variablen, die an IML-Teilgraphen oder Selektionen in GIANT gebunden sind.
3. Das Ergebnis der inspection ist identisch mit dem Ergebnis der gewählten rechten Seite.

visible_var ::= identifier

1. Inspiziert das Objekt, das in der aktuellen Aktivierungsumgebung an die sichtbare Variable mit Name identifier gebunden ist (siehe 1.3.4.3).
2. Falls keine sichtbare Variable dieses Namens existiert, dann ist die Auswertung ein Laufzeitfehler. Andernfalls ist die Inspektion erfolgreich.

3. Falls die Inspektion erfolgreich ist, so ist das Ergebnis der Wert des Objekts an das die Variable gebunden ist.

global_var ::= ⟨ **subgraph** | **selection** ⟩ **.** identifier

1. Falls **subgraph** angegeben ist:
   a) Diese Inspektion ist eine Schnittstelle zu GIANT.
   b) Falls in GIANT ein IML-Teilgraph T mit Namen identifier existiert, dann ist das Ergebnis dieser Inspektion ein Wert M vom zusammengesetzten Typ Object_Set. M enthält alle Graph-Knoten aus T und alle Graph-Kanten aus T und nichts sonst.
   c) Falls in GIANT T nicht existiert, so ist das Ergebnis dieser Inspektion der Wert **null**.
   d) Formal erfolgt eine Inspektion der Variable mit Namen **subgraph**.identifier. Diese Variable ist stets in der Aktivierungsumgebung $Standard$ gebunden und deshalb stets sichtbar.
2. Falls **selection** angegeben ist:
   a) Diese Inspektion ist eine Schnittstelle zu GIANT.
   b) Falls der GSL Interpreter im Kontext eines Anzeigefensters ausgeführt wird und in diesem Anzeigefenster eine Selektion S mit Namen identifier existiert, dann ist das Ergebnis dieser Inspektion ein Wert M vom Typ Object_Set. M enthält alle Fenster-Knoten aus S und alle Fenster-Kanten aus S und nichts sonst.
   c) Falls der GSL Interpreter im Kontext eines Anzeigefensters ausgeführt wird und in diesem Anzeigefenster keine Selektion mit Namen identifier existiert, so ist das Ergebnis dieser Inspektion der Wert **null**.
   d) Falls der GSL Interpreter nicht im Kontext eines Anzeigefensters ausgeführt wird, so ist die Auswertung dieser Inspektion ein Laufzeitfehler.
   e) Formal erfolgt eine Inspektion der Variable mit Namen **selection**.identifier. Diese Variable ist, falls der GSL Interpreter im Kontext eines Anzeigefensters ausgeführt wird, in der Aktivierungsumgebung $Standard$ gebunden und deshalb sichtbar. Falls der GSL Interpreter nicht im Kontext eines Anzeigefensters ausgeführt wird, so existiert die Variable nicht.

1.4.7.2. Referenz

reference ::= visible_ref | var_creation | global_ref

1. Eine reference nimmt die Referenz einer Variable (siehe 1.3.3). Die verschiedenen rechten Seiten werden verwendet um Variablen zu unterscheiden, die in verschiedenen Aktivierungsumgebungen gebunden sind.
2. Das Ergebnis der reference ist identisch mit dem Ergebnis der gewählten rechten Seite.

visible_ref ::= ' identifier

1. Nimmt die Referenz der Variable mit Name identifier, die in der aktuellen Aktivierungsumgebung sichtbar ist (siehe 1.3.4.3).

2. Falls in der aktuellen Aktivierungsumgebung keine sichtbare Variable dieses Namens existiert, so führt die Auswertung zu einem Laufzeitfehler. Andernfalls ist die Auswertung erfolgreich.
3. Falls die Auswertung erfolgreich ist, so ist das Ergebnis ein Zugriffspfad auf das Objekt an das die Variable gebunden ist.

var_creation ::= + identifier

1. Bindet in der aktuellen Aktivierungsumgebung (siehe 1.3.4.2) die Variable mit Namen identifier an ein neues Objekt.
2. Falls in der aktuellen Aktivierungsumgebung bereits eine Variable mit diesem Namen gebunden war, so führt die Auswertung zu einem Laufzeitfehler. Andernfalls ist die Auswertung erfolgreich.
3. Das neue Objekt hat den Wert **null**.

global_ref ::= **'** ⟨ **subgraph** | **selection** ⟩ **.** identifier

1. Falls **subgraph** angegeben ist:
   a) Diese Zugriffspfadbestimmung ist eine Schnittstelle zu GIANT.
   b) Falls in GIANT kein IML-Teilgraph mit Name identifier existiert, dann erzeugt GIANT diesen IML-Teilgraph. Er enthält keine Graph-Knoten und keine Graph-Kanten.
   c) Das Ergebnis ist ein Zugriffspfad auf den IML-Teilgraph mit Name identifier.
2. Falls **selection** angegeben ist:
   a) Diese Zugriffspfadbestimmung ist eine Schnittstelle zu GIANT.
   b) Falls der GSL Interpreter nicht im Kontext eines Anzeigefensters ausgeführt wird, so führt die Auswertung dieser Zugriffspfadbestimmung zu einem Laufzeitfehler. Andernfalls ist die Auswertung erfolgreich.
   c) Ist die Auswertung erfolgreich und existiert im Anzeigefenster keine Selektion mit Name identifier, so erzeugt GIANT im Anzeigefenster diese Selektion. Sie enthält keine Fenster-Knoten und keine Fenster-Kanten.
   d) Ist die Auswertung erfolgreich, so ist das Ergebnis ein Zugriffspfad auf die Selektion mit Name identifier in dem Anzeigefenster, in dessen Kontext der GSL Interpreter ausgeführt wird.

1.4.7.3. Scriptdeklaration

script_decl ::= **{** list **,** expression **}**

1. Eine script_decl deklariert ein neues Script S.
2. Ergebnis ist eine Aktivierungsinformation I vom Typ Script_Reference für S.
3. Die zulässige Aktivierungsumgebung in I ist die (zum Zeitpunkt der Auswertung der script_decl) aktuelle Aktivierungsumgebung

4. Die list heißt Parameterliste von S (vgl. 1.4.7.6).
5. Die expression heißt Rumpf von S (vgl. 1.4.7.6).

1.4.7.4. Liste

list ::= **(** ⟨ expression ⟨ **,** expression ⟩* ⟩? **)**

1. Eine list kann ausgewertet werden und hat als Ergebnis einen Wert vom Typ List.
2. Zur Auswertung einer list werden der Reihe nach von links nach rechts alle expression ausgewertet, falls überhaupt welche angegeben sind.
3. Die Auswertung führt zu einem Laufzeitfehler, falls die Auswertung einer der expression zu einem Laufzeitfehler führt. Andernfalls ist die Auswertung erfolgreich.
4. Falls die Auswertung erfolgreich ist, so ist das Ergebnis ein Wert vom Typ List, der die Ergebnisse aller expression in ihrer korrekten Reihenfolge enthält und sonst nichts.

1.4.7.5. Sequenz

sequence ::= **[** ⟨ expression **;** ⟩* **]**

1. Eine sequence kann ausgewertet werden und hat ein Ergebnis.
2. Zur Auswertung einer sequence werden der Reihe nach von links nach rechts alle expression ausgewertet, falls überhaupt welche angegeben sind.
3. Die Auswertung führt zu einem Laufzeitfehler, falls die Auswertung einer der expression zu einem Laufzeitfehler führt. Andernfalls ist die Auswertung erfolgreich.
4. Falls die Auswertung erfolgreich ist, so ist das Ergebnis der Wert, den die Auswertung der letzten (der am weitesten rechts stehenden) expression zum Ergebnis hatte. Falls die sequence keine expression enthält, so ist das Ergebnis **null**.

1.4.7.6. Scriptaktivierung

script_activation ::= expression list

1. Die expression wird ausgewertet und hat als Ergebnis den Wert I.
2. I muss vom Typ Script_Reference sein. Andernfalls führt die Auswertung zu einem Laufzeitfehler. I enthält einen Zugriffspfad auf das Skript S.
3. Die list wird ausgewertet.
4. Es wird eine neue Aktivierungsumgebung A erzeugt. A beerbt die in der Aktivierungsinformation angegebene zulässige Aktivierungsumgebung.
5. A wird aktuelle Aktivierungsumgebung.
6. Die Parameterliste P von S wird ausgewertet. Falls einer der Werte in P nicht vom Typ Var_Reference ist, so ist dies ein Laufzeitfehler.

7. P muss genauso viele Werte enthalten wie list, andernfalls ist dies ein Laufzeitfehler.

8. Den Var_Reference in P werden die Werte der Einträge des Werts von list zugewiesen (genaue Beschreibung siehe 1.3.5.3).

9. Die expression wird ausgewertet und hat den Wert R als Ergebnis.

10. A wird zerstört. Alle Objekte, die von lokalen Variablen in A gebunden wurden dürfen zerstört werden.

11. Die Aktivierungsumgebung, die zuvor aktuell war wird wieder aktuell.

12. Falls kein Laufzeitfehler aufgetreten ist, so ist das Ergebnis der Auswertung der Wert R (siehe Punkt 9).

1.5. Vordefinierte Sprachumgebung

In diesem Kapitel werden Scripts beschrieben, die in GSL standardmäßig zur Verfügung stehen. Dabei wird folgende Form verwendet:

Variablenname

Param_Name	Typ	Beschreibung des Parameters. In der Spalte „Typ“ werden alle möglichen Typen aufgelistet, die zulässig sind. Falls ein Wert eines anderen Typs übergeben wird, so soll während der Auswertung des Scripts ein Laufzeitfehler auftreten.
Second_Name	Typ	Die Parameter werden in der Reihenfolge aufgelistet, in der die entsprechenden Argumente angegeben werden müssen.
Ergebnis	Typ	Beschreibung des Ergebnis, in der Spalte „Typ“ wird eine Zusicherung angegeben. Das Ergebnis muss ein Wert dieses Typs sein.
Beschreibung der Funktion und des genauen Ablaufs des Scripts. Der Variablenname ist der Name einer Variable, deren Inspektion eine Script_Reference auf das beschriebene Script zum Ergebnis hat.		

1.5.1. Elementare Scripts

Die hier beschriebenen Scripts realisieren elementare Funktionalität, die der GSL Interpreter direkt zur Verfügung stellt. Der GSL Interpreter erzeugt die Objekte, in denen die Aktivierungsinformation gespeichert ist. Vor der Auswertung des Ausdrucks in der Datei „standard.gsl“ erzeugt der GSL Interpreter in der Aktivierungsumgebung $Standard$ die hier genannten Variablen.

1.5.1.1. Zuweisung

assign

lhs	Var_Reference	Zugriffspfad auf eine sichtbare Variable
rhs	jeder Typ	Zuzuweisender Wert eines beliebigen Typs. Es gibt jedoch gewisse Einschränkungen (siehe Beschreibung)
Ergebnis		**null**

Das Objekt, das durch lhs referenziert wird, ändert seinen Wert zu rhs. Falls rhs einen Wert eines der Typen

1. Script_Reference
2. Var_Reference

hat, so muss für die Aktivierungsumgebungen gelten: $A_{rhs} \leq A_{lhs}$, andernfalls führt die Auswertung zu einem Laufzeitfehler.
A_{lhs} ist dabei die kleinste Aktivierungsumgebung, in der das durch lhs referenzierte Objekt an eine Variable gebunden ist. A_{rhs} ist die kleinste Aktivierungsumgebung, in der das durch A_{rhs} referenzierte Objekt an eine Variable gebunden ist.
Durch diese Einschränkung wird sichergestellt, dass zu keinem Zeitpunkt ein Wert eines der Typen Script_Reference oder Var_Reference existiert, der ein Objekt referenziert, das an keine Variable gebunden ist. Dieser Fall könnte sonst eintreten nachdem eine Aktivierungsumgebung zerstört wurde.

1.5.1.2. Kontrollfluss

if

condition	Boolean oder Script_Reference	Ein Wert vom Typ Boolean oder die Aktivierungsinformation für ein parameterloses Script, dessen Ergebnis ein Wert vom Typ Boolean ist.
true_branch	jeder Typ	Falls der Typ Script_Reference ist, so muss true_branch die Aktivierunginformation für ein parameterloses Script sein, dessen Ergebnis beliebigen Typs sein darf.
false_branch	jeder Typ	Falls der Typ Script_Reference ist, so muss false_branch die Aktivierunginformation für ein parameterloses Script sein, dessen Ergebnis beliebigen Typs sein darf.
Ergebnis	von Argumenten abhängig	Siehe Beschreibung
Falls condition eine Aktivierungsinformation ist, so wird das entsprechende Script ohne Argumente aktiviert. Ergebnis ist der Wert b vom Typ Boolean. Falls condition vom Typ Boolean ist, so ist $b =$ condition. Falls b erfüllt ist, so ist $c =$ true_branch, andernfalls $c =$ false_branch. Falls c eine Aktivierungsinformation ist, so wird das referenzierte Script ohne Argumente aktiviert. Ergebnis des Scripts if ist das Ergebnis von c. Falls c keine Aktivierungsinformation ist, so ist das Ergebnis des Scripts if der Wert von c. **Vorsicht:** falls der Ausdruck in true_branch oder false_branch Seiteneffekte hat, so ist ist die Übergabe einer Aktivierungsinformation geraten. Andernfalls würde der Ausdruck ausgewertet unabhängig davon, ob b erfüllt ist, oder nicht.		

loop

body	Script_Reference	Aktivierungsinformation für parameterloses Script mit Ergebnistyp Boolean
Ergebnis		**null**
Aktiviert body. Falls body das Ergebnis **true** hat, aktiviert body so oft ein weiteres Mal, bis body das Ergebnis **false** hat.		

error

message	String	Fehlermeldung
Ergebnis		
Löst einen Laufzeitfehler geziehlt aus und veranlasst den GSL Interpreter dem Benutzer die Fehlermeldung message anzuzeigen.		

run

file_name	String	Name einer Bibliothek. An diesen Namen wird noch der Text „.gsl“ angehängt
Ergebnis		Ergebnis des ausgeführten Ausdrucks
An den file_name wird „.gsl“ angehängt. Dann sucht der GSL Interpreter die Standard Directories nach einem File mit dem Namen. Dann führt der GSL Interpreter den in diesem File enthaltenen GSL Ausdruck in der aktuellen Aktivierungsumgebung aus. Mit Hilfe dieses Scripts können Bibliotheken von benutzerdefinierten Scripts geladen werden. Eine Bibliothek sollte eine sequence sein und in den expression der sequence sollten neue Variablen erzeugt werden, die die Aktivierungsinformation von den Scripts der Bibliothek enthalten.		

1.5.1.3. Rechenoperationen

add

a	Natural oder Var_Reference	Entweder eine Zahl oder ein Zugriffspfad auf ein Objekt, das einen Wert aus Node_Set oder Edge_Set hat
b	Natural bzw. Node_Id bzw. Edge_Id	Abhängig von dem Typ von a ebenfalls eine Zahl oder der entsprechende Elementtyp
Ergebnis	Natural bzw. der namenlose Typ	Abhängig von den Argumenten entweder die Summe der beiden Zahlen oder der Wert **null**

Es werden verschiedene Verwendungen unterschieden:

1. a ist vom Typ Natural und b ist vom Typ Natural. Dann hat dieses Script die Summe der beiden Zahlen als Ergebnis. Es ist ein Laufzeitfehler, falls der (implementationsabhängige) maximale Wertebereich überschritten wird.
2. a ist vom Typ Var_Reference und hat als Wert einen Zugriffspfad auf ein Objekt, dessen Wert N vom Typ Node_Set ist. Dann muss b vom Typ Node_Id sein. Der Wert N wird um das Element b vergrößert, falls b zuvor noch nicht in N enthalten war, andernfalls wird nichts getan. Das Ergebnis ist **null**.
3. a ist vom Typ Var_Reference und hat als Wert einen Zugriffspfad auf ein Objekt, dessen Wert E vom Typ Edge_Set ist. Dann muss b vom Typ Edge_Id sein. Der Wert E wird um das Element b vergrößert, falls b zuvor noch nicht in E enthalten war, andernfalls wird nichts getan. Das Ergebnis ist **null**.

sub

a	Natural oder Var_Reference	Entweder eine Zahl oder ein Zugriffspfad auf ein Objekt, das einen Wert aus Node_Set oder Edge_Set hat
b	Natural bzw. Node_Id bzw. Edge_Id	Abhängig von dem Typ von a ebenfalls eine Zahl oder der entsprechende Elementtyp
Ergebnis	Natural bzw. der namenlose Typ	Abhängig von den Argumenten entweder die Differenz der beiden Zahlen oder der Wert **null**

Es werden verschiedene Verwendungen unterschieden:

1. a ist vom Typ Natural und b ist vom Typ Natural. Dann hat dieses Script die Differenz a − b der beiden Zahlen als Ergebnis. Es ist ein Laufzeitfehler, falls der (implementationsabhängige) maximale Wertebereich überschritten wird.
2. a ist vom Typ Var_Reference und hat als Wert einen Zugriffspfad auf ein Objekt, dessen Wert N vom Typ Node_Set ist. Dann muss b vom Typ Node_Id sein. Der Wert N wird um das Element b verkleinert, falls b zuvor in N enthalten war, andernfalls wird nichts getan. Das Ergebnis ist **null**.
3. a ist vom Typ Var_Reference und hat als Wert einen Zugriffspfad auf ein Objekt, dessen Wert E vom Typ Edge_Set ist. Dann muss b vom Typ Edge_Id sein. Der Wert E wird um das Element b verkleinert, falls b zuvor in E enthalten war, andernfalls wird nichts getan. Das Ergebnis ist **null**.

cat

left	String	Der linke String
right	String	Der rechte String
Ergebnis	String	Konkatenation von left und right

Berechnet die Konkatenation zweier Strings.

1.5.1.4. Vergleiche

less

a	Natural, Edge_Id, Node_Id, String	
b	der selbe Typ wie a	
Ergebnis	Boolean	Ergebnis der Ungleichung a < b

Berechet Enthaltensein des Paars (a, b) in einer irreflexiven antisymmetrischen und transitiven binären Relation. Die Relation ist auf den verschiedenen Typen folgendermaßen definiert:

Natural Wie in der Mathematik für < üblich.

Edge_Id Implementationsabhängig, hat keine besondere Semantik, kann verwendet werden um eine konsistente Sortierung verschiedener Listen zu erreichen.

Node_Id Implementierungsabhängig, hat keine besondere Semantik, kann verwendet werden um eine konsistente Sortierung verschiedener Listen zu erreichen.

String Imlementierungsabhängige lexikographische Ordnung.

equal

a	Natural, Edge_Id, Node_Id, String	
b	der selbe Typ wie a	
Ergebnis	Boolean	**true**, falls a und b gleiche Werte sind, **false** sonst
Testet ob a = b.		

in_regexp

s	String	Teststring
regexp	String	Ein regulärer Ausdruck nach der Spezifikation des Ada95 Pakets GNAT.Reg_Pat
Ergebnis	Boolean	**true** falls s in der durch regexp beschriebenen Sprache enthalten ist, **false** sonst
Ermittelt, ob ein String in der Sprache eines regulären Ausdrucks enthalten ist. Es ist ein Laufzeitfehler, falls regexp syntaktisch nicht korrekt ist.		

type_in

type	String	Typname eines IML-Knoten oder einer IML-Kante
class_set	String	Name einer Klassenmenge in GIANT
Ergebnis	Boolean	**true** falls type in class_set enthalten ist, **false** sonst
Testet ob der Typname type in der Klassenmenge class_set enthalten ist. Der Typname einer IML-Kante hat das Format „Knotentyp.Attributname“. Es ist ein Laufzeitfehler, falls class_set kein Name einer Klassenmenge ist.		

1.5.1.5. Mengen und Listen

In dieses Kapitel gehören auch die Variablen **add** und **sub**. Da diese jedoch nicht ausschließlich auf Mengen angewendet werden, sind sie in dem allgemeineren Kapitel 1.5.1.3 aufgeführt.

empty_node_set

Ergebnis	Node_Set	Ein Wert M mit equal (size_of (M), 0)
Hat als Ergebnis eine leere Knotenmenge.		

empty_edge_set

Ergebnis	Edge_Set	Ein Wert M mit equal (size_of (M), 0)
Hat als Ergebnis eine leere Kantenmenge.		

is_in

set	Node_Set oder Edge_Set	Ein Wert eines der Mengentypen
element	Node_Id bzw. Edge_Id	Ein Element, der Typ muss zum Argument set passend sein. Falls set ein Wert aus Node_Set ist, so muss element ein Wert vom Typ Node_Id sein. Falls set ein Wert aus Edge_Set ist, so muss element ein Wert vom Typ Edge_Id sein.
Ergebnis	Boolean	**true**, falls element in set enthalten ist, **false** sonst
Testet, ob element in set enthalten ist.		

for_each

set	Node_Set oder Edge_Set	Grundmenge
action	Script_Reference	Aktivierungsinformation für ein Script, das genau einen Parameter vom Elementtyp der Grundmenge hat (Also Node_Id für Node_Set bzw. Edge_Id für Edge_Set)
Ergebnis		**null**
Iteriert über alle Elemente der Grundmenge und aktiviert action je einmal für jedes Element. Übergibt das Element an action als einzigen Parameter.		

size_of

obj	List oder Node_Set oder Edge_Set	Irgendeine Liste, oder eine Menge
Ergebnis	Natural	Anzahl der Werte in obj
Ermittelt die Anzahl der Einträge der Liste bzw. die Kardinalität der Menge.		

get_entry

list	List	Irgendeine nicht-leere Liste
index	Natural	Index des gewünschten Eintrags. Der am weitesten links stehende Eintrag hat die Nummer 1, der am weitesten rechts stehende hat die Nummer size_of (list)
Ergebnis		Wert des index-ten Eintrags von list
Ermittelt den Wert eines bestimmten Eintrags von list		

1.5.1.6. Typfeststellungen

is_nodeid

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Node_Id ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Node_Id ist.		

is_edgeid

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Edge_Id ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Edge_Id ist.		

is_node_set

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Node_Set ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Node_Set ist.		

is_edge_set

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Edge_Set ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Edge_Set ist.		

is_string

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ String ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ String ist.		

is_boolean

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Boolean ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Boolean ist.		

is_natural

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Natural ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Natural ist.		

is_list

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ List ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ List ist.		

is_reference

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Var_Reference ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Var_Reference ist.		

is_script

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert vom Typ Script_Reference ist, **false** andernfalls.
Überprüft, ob der Wert obj vom Typ Script_Reference ist.		

is_null

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert des namenlosen Typs ist, **false** andernfalls.
Überprüft, ob der Wert von obj der Wert **null** des namenlosen Typs ist.		

1.5.2. Interaktion mit GIANT

In diesem Kapitel werden alle Aktionen und alle Anfragen definiert, die mit GSL ausgelöst werden können.

1.5.2.1. Interaktion mit dem GSL Interpreter

get_current_window

Ergebnis	String oder der namenlose Typ	Name des Anzeigefensters oder **null**
Falls der GSL Interpreter im Kontext eines Anzeigefenster ausgeführt wird, hat dieses Script als Ergebnis den Namen dieses Anzeigefensters. Andernfalls ist das Ergebnis der Wert **null**.		

set_current_window

window_name	String	Name eines Anzeigefensters
Ergebnis	Boolean	**true**, falls der Kontext des GSL Interpreters geändert wurde, **false** sonst
Ändert den Kontext des GSL Interpreters, so dass der GSL Interpreter danach im Kontext des Anzeigefensters mit Namen window_name ausgeführt wird. Diese Aktion kann aus einer Vielzahl von Gründen fehlschlagen. Ein GSL Interpreter darf den Wechsel des Kontexts verbieten, falls kein Anzeigefenster mit Namen window_name existiert oder falls Objekte existieren, die Referenzen auf Selektionen in einem anderen Anzeigefenster enthalten. Auf jeden Fall sollte deshalb das Ergebnis dieses Scripts beachtet werden.		

1.5.2.2. Anfragen an den IML-Graph

root_node

Ergebnis	Node_Id	Der Wurzelknoten des IML-Graphen
Hat als Ergebnis den Wurzelknoten des IML-Graphen.		

all_nodes

Ergebnis	Node_Set	Menge aller IML-Knoten des IML-Graph
Hat als Ergebnis die Menge aller IML-Knoten des IML-Graph.		

has_attribute

n	Node_Id	Ein IML-Knoten
attrib	String	Name eines Attributs
Ergebnis	Boolean	**true**, falls der IML-Knoten n ein Attribut mit Name attrib besitzt, **false** sonst
Testet, ob ein bestimmter IML-Knoten ein bestimmtes Attribut besitzt.		

get_attribute

n	Node_Id	Ein IML-Knoten
attrib	String	Name eines Attributs
Ergebnis		Wert des Attributs oder **null**
Stellt fest, ob der IML-Knoten n ein Attribut mit Namen attrib besitzt. Falls nein ist das Ergebnis **null**. Andernfalls ist das Ergebnis der Wert des Attributs. Solche Werte, die in der Reflektion durch einen Enumerator dargestellt werden, werden hier als List dargestellt.		

get_type

obj	Node_Id oder Edge_Id	Ein IML-Knoten oder eine IML-Kante
Ergebnis	String	Typ von obj
Ergebnis ist die Typbezeichnung von obj. Für einen IML-Knoten ist dieser Text identisch mit dem Typname. Für eine IML-Kante setzt sich dieser Text zusammen aus Name des Typs des Startknoten . Name des Attributs dieser IML-Kante.		

get_source

e	Edge_Id	Eine IML-Kante
Ergebnis	Node_Id	Startknoten von e
Ermittelt den Startknoten einer IML-Kante.		

get_target

e	Edge_Id	Eine IML-Kante
Ergebnis	Node_Id	Zielknoten von e
Ermittelt den Zielknoten einer IML-Kante.		

1.5.2.3. Anfragen an die GUI

exists_window

window_name	String	Name eines Anzeigefensters
Ergebnis	Boolean	**true**, falls das Anzeigefenster existiert, **false** sonst
Ermittelt, ob in GIANT ein Anzeigefenster mit Name window_name existiert.		

get_window_content

window_name	String	Name eines Anzeigefensters
Ergebnis	Object_Set	Alle Fenster-Knoten und Fenster-Kanten des Anzeigefensters
Ermittelt den Teilgraphen des IML-Graphen der in dem Anzeigefenster window_name enthalten ist. Es ist ein Laufzeitfehler, falls kein Anzeigefenster mit Namen window_name existiert.		

1.5.2.4. Aktionen auf der GUI

create_window

window_name	String	Der Name des zu erzeugenden Anzeigefensters.
Ergebnis	Boolean	**true** falls das Anzeigefenster erzeugt wurde, **false** sonst.
Erzeugt ein neues Anzeigefenster mit Name window_name in GIANT, falls ein solches noch nicht existiert. Öffnet das neue Anzeigefenster.		

insert_into_window

window_name	String	Name eines Anzeigefensters
layout_algo	String	Aufrufstring eines Layoutalgorithmus
add_content	Object_Set	Einzufügender IML-Teilgraph
Ergebnis	Natural	Anzahl der eingefügten IML-Knoten und IML-Kanten
Bildet die größte Teilmenge G von add_content, so dass jede IML-Kante in G zu genau zwei IML-Knoten in G inzident ist und G alle IML-Knoten aus add_content enthält. Wendet den Layoutalgorithmus layout_algo auf content an. Fügt das Resultat in das Anzeigefenster mit Name window_name ein. Es ist ein Laufzeitfehler, falls layout_algo kein korrekter Aufrufstring ist. Es ist ein Laufzeitfehler, falls kein Anzeigefenster mit Namen window_name existiert.		

remove_from_window

window_name	String	Name eines Anzeigefensters
remove_content	Object_Set	Zu entfernende IML-Knoten und IML-Kanten
Ergebnis	Natural	Anzahl der entfernten IML-Knoten und IML-Kanten
Entfernt alle diejenigen IML-Knoten und IML-Kanten, die sowohl in remove_content als auch in dem Anzeigefenster mit mit Name window_name enthalten sind aus dem Anzeigefenster. Zusätzlich werden alle Fenster-Kanten aus dem Anzeigefenster entfernt, deren Start- oder Zielknoten im Fenster nicht mehr enthalten ist. Es ist ein Laufzeitfehler, falls kein Anzeigefenster mit Namen window_name existiert.		

1.5.3. Häufig verwendete Scripts

Die hier definierten Scripts bilden eine Umgebung, die das Schreiben von benutzerdefinierten Scripts vereinfachen soll. Jedes hier vorgestellte Script ist redundant.

Diese Scripts werden durch den Ausdruck in der Datei „standard.gsl“ definiert.

not

a	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
Ergebnis	Boolean	Negation von a
Logische Negation		

and_then

a	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, dass den boolschen Wert zum Ergebnis hat
b	nur Script_Reference	Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
Ergebnis	Boolean	Logische Konjunktion von a und b
Berechnet die logische Konjunktion. Falls a eine Aktivierungsinformation ist, so wird das zugehörige Script aktiviert und liefert als Ergebnis v. Andernfalls ist $v =$ a. Falls v nicht erfüllt ist, so wird das durch b referenzierte Script nicht aktiviert. Andernfalls wird es ausgewertet und die Konjunktion berechnet.		

and

a	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
b	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
Ergebnis	Boolean	Logische Konjunktion von a und b
Berechnet die logische Konjunktion. Falls Aktivierungsinformationen übergeben wurden, so werden die zugehörigen Scripts auf jeden Fall aktiviert. Falls a und b beides Aktivierungsinformationen sind, so wird zuerst a, dann b aktiviert.		

or_else

a	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
b	nur Script_Reference	Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat
Ergebnis	Boolean	Logische Disjunktion von a und b
Berechnet die logische Diskunktion. Falls a eine Aktivierungsinformation ist, so wird das zugehörige Script aktiviert und liefert als Ergebnis v. Andernfalls ist $v =$ a. Falls v erfüllt ist, so wird das durch b referenzierte Script nicht aktiviert. Andernfalls wird es ausgewertet und die Disjunktion berechnet.		

or

a	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat.
b	Boolean oder Script_Reference	boolscher Wert oder Aktivierungsinformation für ein parameterloses Script, das den boolschen Wert zum Ergebnis hat.
Ergebnis	Boolean	Logische Disjunktion von a und b
Berechnet die logische Disjunktion. Falls Aktivierungsinformationen übergeben wurden, so werden die zugehörigen Scripts auf jeden Fall aktiviert. Falls a und b beides Aktivierungsinformationen sind, so wird zuerst a, dann b aktiviert.		

while

b	Script_Reference	Aktivierungsinformation für ein parameterloses Script das als Ergebnis einen Wert vom Typ Boolean hat.
c	Script_Reference	Aktivierungsinformation für ein parameterloses Script.
Ergebnis		**null**
Wertet wiederholt abwechselnd zuerst b und dann c aus. Stoppt, sobald b das Ergebnis **false** hat.		

repeat

c	Script_Reference	Aktivierungsinformation für ein parameterloses Script.
until	Script_Reference	Aktivierungsinformation für ein parameterloses Script das als Ergebnis einen Wert vom Typ Boolean hat.
Ergebnis		**null**
Wertet c einmal aus. Wertet dann wiederholt abwechselnd zuerst until und dann c aus. Stoppt, sobald until das Ergebnis **false** hat.		

union

A	Node_Set Edge_Set	oder
B	der selbe Typ wie A	
Ergebnis	der selbe Typ wie A	
Berechnet die Mengenvereinigung von A und B.		

intersection

A	Node_Set Edge_Set	oder
B	der selbe Typ wie A	
Ergebnis	der selbe Typ wie A	
Berechnet die Schnittmenge von A und B.		

difference

A	Node_Set Edge_Set	oder
B	der selbe Typ wie A	
Ergebnis	der selbe Typ wie A	
Berechnet die Mengendifferenz A - B.		

select_nodes

from	Node_Set	Gundmenge
node_predicate	Script_Reference	Aktivierungsinformation für ein Script, das als einzigen Parameter einen Wert vom Typ Node_Id nimmt und ein Ergebnis vom Typ Boolean hat
Ergebnis	Node_Set	
Ermittelt die größte Teilmenge von from, so dass alle IML-Knoten in dieser Teilmenge das node_predicate erfüllen.		

build_nodes

edge_set	Edge_Set	Kantenmenge
node_predicate	Script_Reference	Aktivierungsinformation für ein Script, das als einzigen Parameter einen Wert vom Typ Node_Id nimmt und ein Ergebnis vom Typ Boolean hat
Ergebnis	Node_Set	
Ermittelt die Menge aller IML-Knoten, die zu einer Kante aus edge_set inzident sind und das node_predicate erfüllen.		

is_object_set

obj	jeder Typ	Der Wert irgendeines Ausdrucks
Ergebnis	Boolean	**true**, falls obj ein Wert des zusammengesetzten Typs Object_Set ist, **false** andernfalls.
Überprüft, ob der Wert obj vom zusammengesetzten Typ Object_Set ist.		