

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin

Diplomarbeit

Ein automatisch generierter SDL-Compiler



vorgelegt von: Michael Piefel
Betreuer: Prof. Dr. Joachim Fischer
Dr. Andreas Prinz
Berlin, 16. August 2000

Diese Diplomarbeit wurde mit dem Textsatzsystem $\text{T}_{\text{E}}\text{X}$ erstellt.
Dabei kamen $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ und das KOMA-Script-Paket zum Einsatz.
Gesetzt in ITC Slimbach in 10 Punkt.

Zusammenfassung

Die *Specification and Description Language* (SDL) der ITU-T zeichnet sich vor allem dadurch aus, dass ihre Semantik mathematisch rigoros definiert ist. Dies geschieht anhand einer abstrakten Form der Syntax, welche die Struktur eines Programmes abbildet und ohne semantisch irrelevante Details wie Begrenzer und lexikalische Regeln auskommt.

Die vorliegende Arbeit zeigt eine Möglichkeit, diese Form zu gewinnen. Sie ist Teil des Projekts SDLC, das sich mit der automatischen Generierung eines SDL-Compilers aus Definitionen von Syntax und Semantik beschäftigt. Dazu werden zwei Werkzeugstrecken implementiert, die Teile des SDL-Compilers erzeugen. Die erste generiert aus der Syntaxbeschreibung Scanner und Parser, die zweite aus Umformregeln die Programmteile, die den Syntaxbaum in eine abstraktere und dabei mathematisch greifbarere Form bringen. Diese kann in fortführenden Schritten analysiert oder ausgeführt werden.

Inhaltsverzeichnis

I	Einleitung	1
1	Einordnung	2
1.1	Überblick	2
1.2	Vom Programm zum abstrakten Syntaxbaum	3
1.3	Das Project SDLC	5
2	Über diese Arbeit	8
2.1	Konventionen	8
2.2	Begriffe	8
2.3	Verwendete Werkzeuge	13
2.4	Zusammenfassung der Schritte	15
2.5	Abgrenzung der Arbeit im Projekt SDLC	16
II	Eine AS0-Repräsentation	18
3	Die AS0	19
3.1	Vorläufige AS0-Definition des Z.100 Annex F	19
3.2	Sinn der AS0	19
3.3	Concrete Grammar (AS0)	21
4	Die Werkzeugkette »Syntax«	23
4.1	Rahmen	23
4.2	Text- und Kimwitu-Repräsentation	25
4.3	Bison-Parser	27
4.4	Weitere Hilfsprodukte	27

III Übergang von AS0 zu AS1	30
5 Spezifikation der Umformregeln	31
5.1 Bezeichner	31
5.2 Funktionen	32
5.3 Domänendefinitionen	33
5.4 Terme	33
5.5 Transitionen	34
6 Die Werkzeugkette »Satanic«	36
6.1 Handgeschriebene Funktionen	36
7 Der funktionale Teil	38
7.1 Generierte Funktionen	38
7.2 Das <i>Mapping</i>	39
8 Die Transitionen	40
8.1 Einfache Transitionen	40
8.2 Die Elternverknüpfung	40
8.3 Die let-Anweisung	41
8.4 Abhängige Transitionen	42
A Abkürzungen	44
B BNF nach Z.100 und Meta IV	45
B.1 BNF	45
B.2 Meta IV	45
Literaturverzeichnis	47

Teil I

Einleitung

1 Einordnung

Dieses Kapitel gibt einen Überblick über die Aufgaben, die in dieser Diplomarbeit bearbeitet wurden, und über ihre Stellung im Rahmen des Projektes SDLC (für »SDL-Compiler«). Da es sich um eine Arbeit im Bereich des Compilerbaus handelt, werden Grundkenntnisse dieser Richtung vorausgesetzt. In Kapitel 2 werden einige Erläuterungen zu verwendeten Konzepten und Begriffen gegeben.

1.1 Überblick

SDL ist eine Beschreibungssprache, die im Standard Z.100 der ITU-T (siehe [ITU99]) definiert ist. Für fast alle heute gebräuchlichen Sprachen gibt es eine formale Beschreibung der Syntax, also des Aufbaus (der Satzbildung, wobei im Computersprachen ein solcher Satz als ein Programm oder eine Spezifikation in der Sprache bezeichnet wird). Die Semantik, also die Bedeutung von Sätzen, wird lediglich informell beschrieben. Dies führt gelegentlich zu Mehrdeutigkeiten und lässt sich zudem nicht mit mathematischen Mitteln handhaben. Die Semantik von SDL dagegen ist mathematisch rigoros definiert.¹ Dies geschieht im Annex F des Z.100 ([ITU00], kurz Z.100.F²)

Die Definition der Semantik von SDL mittels formaler Mittel erlaubt bzw. erleichtert unter anderem Folgendes:

- die Analyse der Semantik eines konkreten SDL-Programms mit formalen Methoden,
- die Verifikation der Korrektheit eines SDL-Compilers und im Besonderen das Erzeugen eines Referenzcompilers,

sowie sogar

- das automatische Generieren eines solchen Compilers aus der formalen Spezifikation.

Letzterem Punkt gilt das Augenmerk des Projektes SDLC. Die Aufgabe besteht darin, verschiedene Werkzeuge bzw. Werkzeugketten zu erzeugen, die ihrerseits dann den SDL-Compiler generieren.

¹ Zumindest galt das für frühere Sprachversionen. Zum Zeitpunkt dieser Arbeit ist der Annex F, der die Semantik definiert, noch nicht vollständig an die neue Sprachversion (SDL-2000) angepasst worden.

² Wenn im folgenden von Z.100 die Rede ist, so ist damit der Hauptteil des Standards, also ohne Annex F gemeint. Z.100.F bezeichnet dagegen Annex F für SDL-2000, auch wenn dieser erst als Entwurf vorliegt.

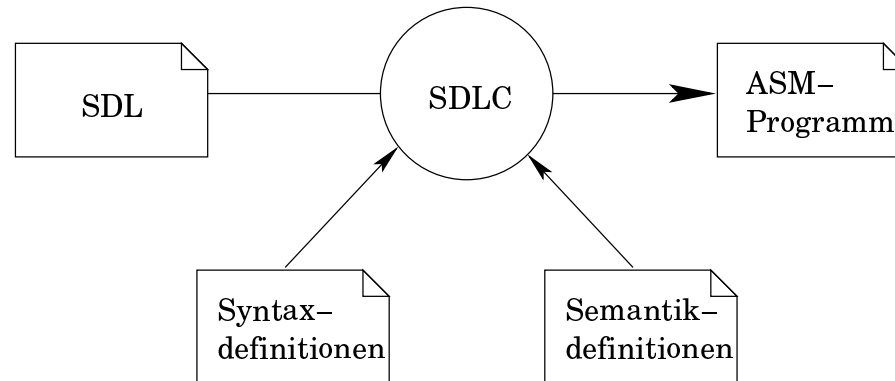


Abbildung 1.1: SDLC – mittels Syntax- und Semantikdefinitionen wird aus einem SDL-Programm das dazugehörige ASM-Programm

Die Definition der dynamischen Semantik geschieht mittels *Abstract State Machines* (ASM). Ziel von SDLC ist es dementsprechend, aus einem konkreten SDL-Programm seine ASM-Repräsentation herzustellen, und zwar mittels der Syntax- und Semantikdefinitionen, wie in Abbildung 1.1 dargestellt.

Dies geschieht über Zwischenschritte, die im Standard aufgeführt sind. Neben der bekannten konkreten Syntax, die das Aussehen von SDL-Programmen beschreibt, gibt es eine abstrakte Syntax, die nur noch einen Syntaxbaum beschreibt und dadurch die Struktur des SDL-Programmes repräsentiert. Diese heißt im Kontext dieser Arbeit AS1. Das Augenmerk der vorliegenden Arbeit liegt auf dem Prozess bis zur Gewinnung der AS1.

1.2 Vom Programm zum abstrakten Syntaxbaum

SDL ist eine sehr umfangreiche Sprache. Für eine Sprache mit einer so reichen Syntax ist es vorteilhaft, Semantikdefinitionen nicht für die ganze Sprache anzugeben, sondern nur für die Kernkonzepte. Für sie existieren Beschreibungen ihres Verhaltens in ASM; der Rest der Sprache wird dann auf diese Kernkonzepte zurückgeführt. Die Repräsentation des Kerns heißt die abstrakte Grammatik von SDL. Sie wird in Z.100 *abstract syntax* genannt, in Z.100.F und in dieser Arbeit zur Unterscheidung von anderen Formen dagegen AS1.

Um eine AS1-Darstellung eines gegebenen SDL-Programms zu erhalten, muss ein Übergang von der konkreten Syntax zur abstrakten definiert werden. Die konkrete Syntax definiert dabei die Menge aller gültigen Programme in SDL,³ die abstrakte Syntax die Menge aller Syntaxbäume von SDL-Programmen. Der Ausgangspunkt dieser Umformung ist allerdings nicht die

³ SDL hat eine textuelle (SDL/PR – *textual phrase representation*) und eine graphische (SDL/GR – *graphical representation*) Darstellung. Diese werden beide von der (textuellen) konkreten Syntax (CS) abgedeckt; die GR wird mittels Pseudo-Konstruktoren in die PR integriert.

konkrete Syntax selbst (oft abgekürzt mit CS), sondern die AS0. Diese AS0 ist eine einfache Zwischenstufe, die aus der konkreten Syntax abgeleitet wird. Sie ist eng an diese angelehnt; im Wesentlichen sind die Unterschiede

- das Weglassen semantisch irrelevanter Details, dazu zählen Begrenzer, lexikalische Regeln und ebenfalls die meisten Schlüsselwörter, sofern diese nicht Träger der eigentlichen Information sind, und
- das konsequente Unterscheiden in Regeln mit Sequenzen (also mit Tupelbildung) und Regeln mit Alternativen (also Mengenvereinigung).⁴

Wie gezeigt werden wird, können diese einfachen Umformungen bereits in einem generierten Parser durchgeführt werden, der im Laufe dieser Arbeit entstand.

Eine Definition der AS0 findet sich in Z.100.F, wurde hier jedoch nicht verwendet. Vielmehr wird die AS0 direkt aus der CS generiert, was den Vorteil hat, dass sie wirklich äquivalent zu dieser ist. Diese generierte Version wird schließlich Eingang in den Standard finden.

Die Umformungen von AS0 zu AS1 selbst fallen in zwei Kategorien, diese sind

- Transitionen, welche innerhalb der AS0 stattfinden, und
- Abbildungen, die von einem Term in AS0 in einen äquivalenten Term in AS1 überführen.

Dazu gibt es noch

- Bedingungen (logische Prädikate), die an bestimmten Stellen des Umformungsprozesses gelten müssen.

Die Transitionen überführen den Syntaxbaum in eine immer abstraktere und immer weniger auf Nicht-Kernkonzepten basierende Form, bis schließlich die Abbildung auf die AS1 des Standards ein einfacher Schritt ist.

Um daher den für die ASM notwendigen AS1-Baum aus einem vorliegenden SDL-Programm zu erhalten, sind folgende Schritte nötig (siehe auch Abbildung 1.2): Ein SDL-Programm wird mittels eines Scanners und eines Parsers in seinen AS0-

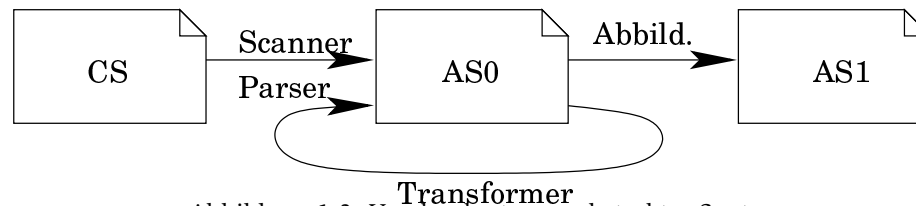


Abbildung 1.2: Von konkreter zu abstrakter Syntax

Syntaxbaum eingelesen, die Transitionen innerhalb der AS0 durchgeführt und schließlich auf die AS1 abgebildet.

⁴ Diese Unterscheidung ist sinnvoll für die Syntaxbäume, die ja das Ziel sind: Die Tupelbildung definiert Knoten des Syntaxbaums.

Um diese Umformungen automatisch vornehmen zu können, sollte ein SDL-Compiler, der SDLC, geschaffen werden.⁵ Der derzeitige Stand ist ein Werkzeug namens RSDLC, was für *Restricted SDL Compiler* steht.⁶ Dieser nimmt ein SDL-Programm und erzeugt daraus entsprechend der Umformregeln im Standard eine ASM-Repräsentation.

Die Umformregeln liegen als Text vor. Ziel des Projekts SDLC ist es nun nicht, ein Programm zu schreiben, das unter Beachtung dieser Regeln eine Umformung von einem SDL-Programm in seine ASM-Darstellung vornimmt. Vielmehr sollen Generatoren entwickelt werden, die aus den Regeln im Standard ein solches Programm erzeugen. Die Gründe liegen auf der Hand: Auch nach einer Modifikation der vorliegenden Regeln bleiben die Generatoren auf Grund der größeren Abstraktion einsatzfähig und es kann sofort ein neuer SDL-Compiler generiert werden.

Eine Beschreibung des Aufbaus von (R)SDLC folgt in Abschnitt 1.3. Die vorliegende Arbeit beschäftigt sich mit wichtigen Teilgebieten der Implementation von SDLC.

Diese Arbeit unterteilt sich in zwei Abschnitte, auch Schritte genannt:

1. Das Entwickeln der AS0-Definition und Umformen eines konkreten SDL-Programms in seine AS0-Darstellung.
2. Das Schreiben des Programmgenerators SATANIC, der die Programmteile von SDLC generiert, die ein Programm, das in AS0-Darstellung vorliegt, in seine AS1-Darstellung überführen.

Im Abschnitt 2.4 wird nochmals ein kurzer Überblick über diese Schritte gegeben, bevor sich je ein Teil dieser Arbeit mit einem Schritt beschäftigt.

1.3 Das Project SDLC

Diese Diplomarbeit entstand im Rahmen eines Projektes zum Compilieren von SDL, genannt SDLC. Es ist Gegenstand von [Pri00]. Ziel ist dabei, ein SDL-Programm in eine ASM-Repräsentation zu überführen. Für diese ASM existieren Werkzeugumgebungen, die es ermöglichen, ASM-Programme auszuführen und zu analysieren. Damit sind verschiedene Möglichkeiten gegeben, SDL-Programme auf ihre Semantik hin zu untersuchen.

Die Umformregeln sind in Z.100.F ebenfalls als ASM aufgeführt. Sie könnten daher mit vorhandenen Werkzeugen direkt ausgeführt werden und zum Übertragen eines SDL-Programms in seine entsprechende ASM-Abstraktion dienen. Das ist jedoch nicht praktikabel, da die Ausführung sehr langsam vonstatten geht. Es ist deswegen wünschenswert, die ASM-Regeln in eine gängige Programmiersprache zu überführen.

Der SDL-Compiler von SDLC ist ein solches Programm. Es wird aus der im Standard gefundenen Beschreibung direkt generiert. Die Generierung ist völlig automatisch: Der Standard liegt im Format von Microsoft Word vor, alle Texte sind konsequent mit Formatvorlagen formatiert; mittels eines Makros kann daraus eine reine Textvariante (ASCII) erzeugt werden, die leicht geparkt werden kann. Aus dieser Vorlage entsteht mittels mehrerer Generatoren schließlich ein SDL-Compiler.

⁵ Zur besseren Unterscheidung von Projekt und Programm gleichen Namens wird der Compiler hier meist mit SDL-Compiler bezeichnet, tatsächlich ist der Programmname `sdlcomp`.

⁶ Siehe Anhang A und 1.3.

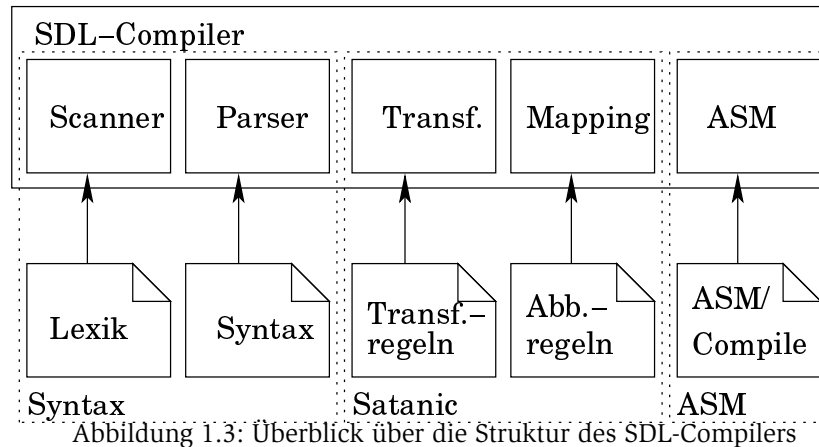


Abbildung 1.3: Überblick über die Struktur des SDL-Compilers

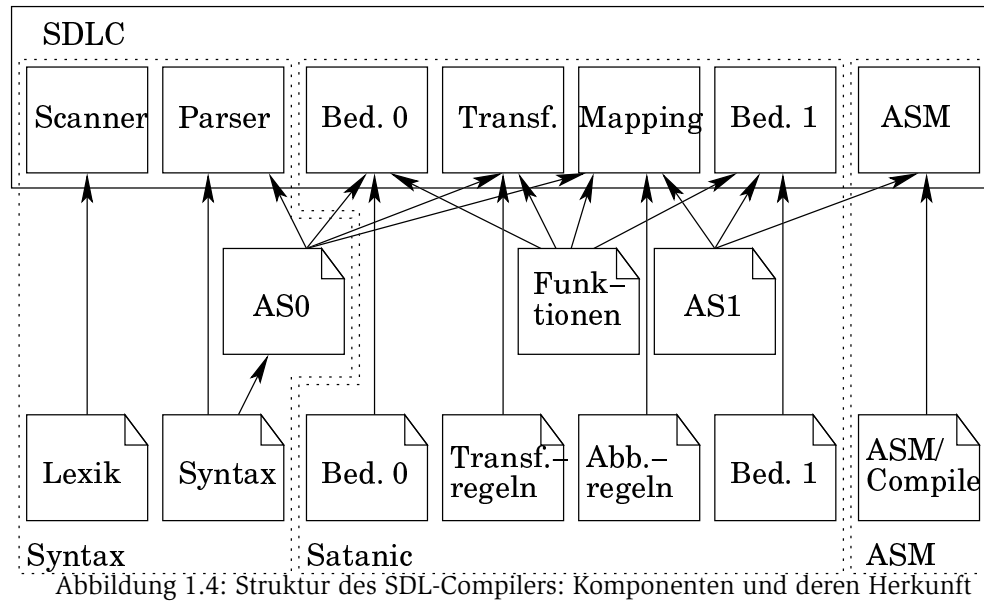
Die Grobstruktur von SDLC ist in Abbildung 1.3 dargestellt. Die Eingabedateien (in der Abbildung mit Knick in der Ecke) werden aus dem Standard extrahiert und dann in Programmcode umgewandelt. Die Generatoren bestehen dabei aus drei Werkzeugstrecken:

- SYNTAX – Diese Strecke arbeitet auf den Dateien, die die konkrete Syntax von SDL repräsentieren (»Lexik« und »Syntax« im Diagramm). Erzeugt werden hier der SDL-Scanner und -Parser. Ferner entstehen hier die AS0-Definition und interne Repräsentationen von AS0 und AS1 sowie einige Hilfsdateien. Siehe dazu Teil II.
- SATANIC – Hier werden aus Bedingungen auf AS0 und AS1, Transitionsregeln und Abbildungsregeln die entsprechenden SDLC-Teile erzeugt, außerdem aus den Funktionsdefinitionen die Funktionsimplementationen. Damit beschäftigt sich Teil III.
- ASM – Diese Strecke erzeugt die Programmteile, die aus einem SDL-Programm in seiner AS1-Syntaxbaumform die zugehörigen ASM erzeugt. Dieses Gebiet liegt außerhalb des Fokus dieser Arbeit.

Abbildung 1.4 zeigt das Diagramm in noch detaillierterer Form (auf Kosten der Übersichtlichkeit). Die AS0 wird aus der Syntax generiert und ist in ihrer Kimwitu++Form Bestandteil vieler Programmteile von SDLC; die AS1 in ihrer Kimwitu++-Form wird für die Abbildungen (die ja erst einen konkreten AS1-Baum erzeugen), die Bedingungen auf der AS1 sowie bei der Erzeugung der ASM-Programme genutzt; die Bedingungen auf AS0 und AS1 werden in Prüffunktionen umgewandelt; Funktionen, die in vielen Programmteilen genutzt werden, werden aus ihrer Spezifikation generiert.

Reicht RSDL?

Der vorläufige Stand des Projektes SDLC arbeitet noch nicht auf der vollen, sehr umfangreichen Sprache SDL. Stattdessen wird nur RSDL benutzt, also eine eingeschränkte Form von SDL. Es stellt sich die Frage, ob sich die Ergebnisse dieses Projektes



RSLDC auf das geplante Projekt SDLC übertragen lassen.

Die Werkzeugstrecken von RSDL sind größtenteils sprachunabhängig geschaffen worden. Es ist nicht einmal zwingend erforderlich, dass die Werkzeugstrecken auf SDL (oder RSDL) arbeiten. Tatsächlich ist zu vermuten, dass SATANIC und ASM ohne Änderungen ebenfalls in SDLC funktionieren werden.

SYNTAX dagegen hat noch einige Einschränkungen. Das Weglassen der grafischen Elementen von SDL in RSDL hat die abstrakte Grammatik, mit der die Grammatik von SDL beschrieben wird, vereinfacht. An dieser Stelle müssen demnach noch Ergänzungen vorgenommen werden, wobei auch hier zu erwarten ist, dass diese Änderungen sich in Grenzen halten werden.

Mit Gelingen von RSDL ist also ein großer Teil des Weges zurückgelegt; SDLC ist von hier aus nur noch ein verhältnismäßig kleiner Schritt.

2 Über diese Arbeit

2.1 Konventionen

Typografische Konventionen

Diese Arbeit benutzt neben regulärem Text in ITC Slimbach:

- *kursive Schrift für Betonungen sowie für nicht übersetzte Fachbegriffe*, die in der Informatik oder im speziellen Sprachgebrauch von Autor und Betreuer üblich sind, keine deutsche Entsprechung haben, aber auch nicht so selbstverständlich sind, dass sie als gewöhnliche Fremdwörter anzusehen sind,
- **Boton für Programmtext**, einschließlich weiterer Auszeichnungen wie Fettdruck für Schlüsselwörter oder schräggestellte Schrift für Kommentare.

Orthographie

In dieser Arbeit kommen die Regeln der deutschen Rechtschreibung in der Fassung der amtlichen Regelung von 1996, die am 1. August 1998 in Kraft traten, zur Anwendung. Benutzt wurde hierzu [Dud96]. Es wurde eine konservative Anwendung der neuen Regeln verfolgt.

2.2 Begriffe

2.2.1 Grammatiken

Konkrete Grammatiken

Diese Art von Grammatik ist die am häufigsten auftretende; meist meint man eine konkrete Grammatik, wenn man einfach nur Grammatik sagt.

Eine Grammatik beschreibt den Aufbau einer Sprache und bestimmt, welche Sätze in der jeweiligen Sprache gültig sind; dies gilt für natürliche Sprachen ebenso wie für Computersprachen. Für eine genaue Definition vergleiche man Grundlagenliteratur, z. B. [FL91]. Grob gesagt versteht man in der Informatik unter einer Grammatik üblicherweise ein Viertupel $G = (T, N, R, s)$ mit

einer Menge T von Terminalsymbolen, einer Menge N von Nichtterminalen, einer Menge R von Regeln sowie einem Startsymbol $s \in N$.

Im der folgenden Beispielgrammatik ist natürlich s das Startsymbol s ; die Terminale STR und INT bilden T ; und $N = \{s, a, b, c, d\}$.

```
s := a | b
a := STR
b := c [d]
c := STR | INT
d := INT
```

Ableiten ist der Vorgang, bei dem ein Nichtterminalsymbol ersetzt wird durch eine Folge von Terminal- und Nichtterminalsymbolen, die der rechten Seite der das Nichtterminalsymbol definierenden Regel entspricht. In diesem Fall wäre eine gültige Ableitung aus s z. B. b und eine aus b z. B. $c d$. Die Menge aller in einer Grammatik G aus s ableitbaren Sätze von Terminalsymbolen nennt man die Sprache $L(G)$ von G . Diese lauten in diesem Fall:

```
STR          INT          STR INT          INT INT
```

Abstrakte Grammatiken

Wir nennen eine Grammatik abstrakt, wenn sie keine Sprache mehr definiert (bzw. die definierte Sprache irrelevant ist), sondern uns der Ableitungsbaum selbst interessiert. Ein Beispiel mit einer konkreten und der zugehörigen abstrakten Grammatik:

```
io := input | output          io = input | output
input := "input" INT ", " STR  input :: INT STR
output := "output" INT ", " STR output :: INT STR
```

Ein gültiger Satz des Ausschnitts der abstrakten Grammatik wäre $INT STR$, aber die Information ist verloren, ob es sich um Eingabe oder Ausgabe handelt. Ein Ausschnitt aus dem Syntaxbaum dagegen lautet:

```
input(
  INT,
  STR
)
```

Eindeutigkeit

In einer Grammatik können mehrere verschiedene Ableitungen den gleichen Satz erzeugen. In der einfachen Grammatik oben lässt sich der Satz STR auf zwei verschiedenen Wegen ableiten:

```
s -> a          s -> b
a -> STR        b -> c
                c -> STR
```

Dies ist kein Problem, wenn es tatsächlich nur darum geht, die Sprache zu definieren. In Z.100 kommt dies häufig vor, denn die dort gegebene Grammatik ist eine Repräsentationsgrammatik, dazu gedacht, von Menschen gelesen zu werden. Heißt es dort etwa: »Hier kann ein Variablenbezeichner stehen oder ein beliebiger Ausdruck,« so ist das an dieser Stelle eventuell sinnvoll, um bestimmte Dinge näher zu erläutern. Natürlich ist ein »Bezeichner« auch ein »Ausdruck«, und die Grammatik damit nicht eindeutig.

Der Aufbau eines Syntaxbaums in einem Parser ist dann aber ebenfalls nicht eindeutig. Damit die Sprache geparkt werden kann, muss die Grammatik erst eindeutig gemacht werden. Eine solche eindeutig gemachte Version der Grammatik von SDL stand als Ausgangsbasis zur Verfügung, um daraus den Parser zu generieren.

Im Kontext dieser Arbeit ist das genau genommen unerheblich, da die Werkzeugketten unabhängig von ihrer Eingabe funktionieren. Durch eine nicht eindeutige Eingabegrammatik würde allerdings ein inkorrekt Parser erzeugt.

2.2.2 Allgemeine Compilertechnologie

Der erste Schritt bei der Übersetzung eines Programms ist das Parsen der Eingabe, das heißt die Zerlegung in ihre grammatischen Bestandteile. Das Parsen steht in enger Beziehung zum Ableiten: Zu einem Satz (oder Programm) der Sprache sucht der Parser die Folge der Ableitungsschritte, die zu diesem Satz geführt haben.

Dem Parser vorgeschaltet ist meist ein Scanner. Die Abgrenzung zwischen Scanner und Parser ist nicht immer klar; es ist oft möglich, Funktionalität in den einen oder den anderen zu legen, und die genaue Aufteilung ist dem Autor überlassen. Üblicherweise zerlegt der Scanner den Strom von Eingabezeichen in sogenannte *Token*, in den meisten Sprachen sind dies genau die Terminalsymbole.

Scanner und Parser können von Hand geschrieben werden. Es gibt aber auch Metawerkzeuge, die dies vereinfachen. Für den Scanner bietet sich etwa *lex* an. Dieses Programm nimmt eine Beschreibung der *Token* mittels regulärer Ausdrücke und erzeugt daraus C-Code. Eine Variante von *lex* ist *flex*, das schnellere Scanner erzeugt. Als Parsergenerator gibt es z. B. *Yacc* bzw. dessen GNU-Variante *Bison*. Diese erzeugen ebenfalls C aus einer abstrakten Spezifikation.

2.2.3 Begriffe im Projekt SDLC

Einige Begriffe werden oft in unterschiedlichen Bedeutungen gebraucht, meist fehlt die Abgrenzung zwischen abstrakter und konkreter Ausprägung. Ein populäres Beispiel ist das Wort *Objekt*, das häufig für Klasse oder Instanz gebraucht wird. Nicht immer geht aus dem Kontext für den Lesenden restlos klar hervor, welche Bedeutung an einer bestimmten Stelle gemeint ist. Auch im Umfeld dieser Arbeit gab es ähnliche Fälle. Es wurde versucht, an jeder Stelle immer deutlich zu machen, was gemeint ist.

Für die verschiedenen Notationen in den folgenden Ausführungen sei auf Anhang B verwiesen.

CS

Die CS ist die konkrete Syntax von SDL. Zum einen ist damit der Text einer Definition gemeint, wie er auch in Z.100 zu finden ist. In diesem Fall handelt es sich um Regeln in BNF, die Ableitungsregeln der Grammatik von SDL sind. Einige lauten zum

Beispiel:

```

<output> ::=
    output <output body>
<output body> ::=
    <signal<identifier> [<actual parameters>]
    {, <signal<identifier> [<actual parameters>] }*
    <communication constraints>
<output symbol> ::=
    <plain output symbol>
    | <internal output symbol>

```

Die Regeln dieser Grammatik beschreiben alle gültigen SDL-Programme. Sie können als Grundlage für einen Akzeptor oder einen Parser dienen, der zu einem SDL-Programm, das in Textform vorliegt, die Folge der Ableitungen findet, die zu ihm geführt haben.

Ein solches Programm lautet etwa auschnittsweise:

```

state s;
input a;
output b, c;
nextstate s;
endstate;

```

ASO

Die ASO ist die abstrakte Syntax 0. Zum einen ist diese ein Text, wie er auch in Z.100.F zu finden sein könnte,⁷ einige Regeln lauten:⁸

```

<output> :: <output body>
<output body> :: <output body gen identifier>+ <communication constraints>
<output body gen identifier> :: <identifier> [<actual parameters>]
<output symbol> = <plain output symbol> | <internal output symbol>

```

Diese AS0-Definition ist also, ähnlich wie die CS, eine Beschreibung des Aufbaus eines SDL-Programms. Diese abstrakte Grammatik beschreibt aber keine Sprache mehr (siehe Abschnitt 2.2.1), sondern vielmehr den Aufbau des Syntaxbaumes.

Zum anderen meint man mit ASO die Darstellung der ASO im Format des Termprozessors Kimwitu++ (siehe Abschnitt 2.3). Die zu obigen Zeilen gehörenden Definitionen lauten:

```

| ASO_output( ASO_rule )
| ASO_output_body( ASO_rule ASO_rule )
| ASO_output_body_gen_identifier( ASO_rule ASO_rule )

```

⁷ ...und in Zukunft sein wird. Die vorliegende Arbeit bildet dafür die Grundlage.

⁸ Die Form, die hier gezeigt wird, ist eine Repräsentation in einfachem, unformatiertem Text. Diese wird aus dem Standard, der als Microsoft-Word-Dokument vorliegt und Auszeichnungen wie Unterstreichung sowie Sonderzeichen enthält, automatisch gewonnen, um eine leicht weiterverarbeitbare Form zu erhalten.

Schließlich bedeutet AS0 (auch »AS0-Baum« oder »konkrete AS0«) auch eine SDL-Spezifikation als Syntaxbaum; der Baum wird durch die AS0-Definition beschrieben und ist im Rahmen dieser Arbeit üblicherweise eine konkreten Instanz der Darstellung von Kimwitu++.

AS1

Die AS1 ist die abstrakte Syntax 1. In Z.100 wird sie stets einfach als *abstract syntax* bzw. *abstract grammar* bezeichnet. Zum einen ist diese ein Text, wie er auch in Z.100 und Z.100.F zu finden ist und sieht zum Beispiel so aus:

```
Signal-identifier = Identifier
Output-node      :: Signal-identifier
                  [Expression]*
                  [Signal-destination]
```

Diese AS1-Definition ist, ähnlich wie die AS0, eine Beschreibung des Aufbaus eines SDL-Programms. Sie beschreibt keine Sprache, sondern ist ebenfalls abstrakt.

Zum anderen meint man mit AS1 die Darstellung der AS1 im Format des Termprozessors Kimwitu++ (siehe Abschnitt 2.3). Die zu obigen Zeilen gehörenden Definition lautet:

```
| AS1_Output_node( AS1_rule AS1_rule AS1_rule )
```

Schließlich bedeutet AS1 (auch »AS1-Baum« oder »konkrete AS1«) ein SDL-Programm als Syntaxbaum; der Baum wird durch die AS1-Definition beschrieben und ist im Rahmen dieser Arbeit üblicherweise eine konkreten Instanz der Darstellung von Kimwitu++, er entsteht aus der AS0 durch Anwenden aller Transitionen und Abbildungen. Der abstrakte Syntaxbaum zu obigem Ausschnitt aus einem SDL-Programm lautet etwa (vereinfacht):

```
State-node(
  Name(s),
  Input-node(
    Identifier(a),
    Transition(
      Output-node(Identifier(b)),
      Output-node(Identifier(c)),
      Nextstate-node(Name(s))
    )
  )
)
```

2.3 Verwendete Werkzeuge

2.3.1 Wahl der Werkzeuge

Das Projekt SDLC benötigt mehrere Scanner, zum einen handgeschriebene zum Einlesen der aus dem Standard extrahierten Dateien, zum anderen generierte für SDL. Für alle Scanner wird flex [Pax95] eingesetzt.

Das Projekt SDLC benötigt auch mehrere Parser, wiederum zum einen handgeschriebene, zum anderen generierte. Für alle Parser wird Bison [CS95] eingesetzt. Bison kann lediglich LALR(1)-Grammatiken parsen. Das ist ausreichend für die Spezifikationen der Umformregeln und Funktionen; die Syntax für diese Teile lag letztlich bei den Autoren. Leider hat jedoch SDL eine anspruchsvollere Grammatik – sie ist nicht LALR(1) oder auch nur LR(1). Es wird über die Verwendung eines alternativen, mächtigeren Parsergenerators für den Schritt von CS zu AS0 nachgedacht: Kandidaten sind z. B. ANTLR [Par00], das LL(k)-Analyse beherrscht, oder BTYacc [Mas00], ein Yacc mit *Backtracking*.

Die Verwaltung der Syntaxbäume, ihre Speicherung, ihr Umformen und ihre formatierte Ausgabe erfolgt mit dem Werkzeug Kimwitu++ [KC++] (siehe auch [EB96] und [Pie99b]). Die Entscheidung fiel aus mehreren Gründen für diese Sprache. Zum einen wird Kimwitu++ am Lehrstuhl für Systemanalyse traditionell für viele Probleme angewendet, auch der Autor hat schon hinreichend Erfahrung damit gesammelt (siehe [Pie99a]).

Zum anderen ist Kimwitu++ für die hier anfallenden Arbeiten bestens geeignet. Es bietet eine einfache Verwendung von Syntaxbäumen an, denn das ist schließlich die Hauptaufgabe von Kimwitu++. Es erlaubt die mustergesteuerte Anwendung von *Rewrite*- und *Unparse*-Regeln (siehe Abschnitt 2.3.2), also Regeln zur Umformung von Termen (bzw. Unterbäumen) in andere Terme und Regeln für das Abschreiten des Syntaxbaums, z. B. für die formatierte Ausgabe.

Die Programmgeneratoren flex, Bison und Kimwitu++ erzeugen aus ihrer Eingabe stets C und C++. Das legt natürlich für die handgeschriebenen Programmteile auch nahe, C oder C++ zu verwenden. Dabei wurde C++ praktisch nur als *“simply a better C”* [Str97, Kap. 24.2.1] benutzt.

2.3.2 Die Technik der Werkzeugketten

Kenntnis der Compilertechnologie wird vorausgesetzt, das heißt Vertrautheit mit flex [Pax95] und Bison [CS95]. Auch Wissen über Kimwitu++ [KC++] wird benötigt; auf Grund der geringen Verbreitung folgt hier eine kurze Übersicht.

Der Termprozessor

Kimwitu++ erlaubt das Aufbauen, Umformen und Abschreiten von typisierten Bäumen. Ein typisierter Baum hat Knoten, die jeweils einen bestimmten Typ haben; von diesem Typ hängt ab, wieviele Söhne der Knoten haben kann und von welchem Typ wiederum diese sind. Der bekannteste Anwendungsfall für solche Bäume sind natürlich Syntaxbäume.

In Kimwitu++ werden diese Bäume in einer Yacc-ähnlichen Syntax definiert. Ein Knotentyp⁹ besteht aus einer Anzahl

⁹ ...der in Kimwitu++ »Phylum« heißt. Dieser Begriff wird auch für konkrete Instanzen der Knotentyps gebraucht.

möglicher Ausprägungen, Operatoren genannt:

```
depTransform:
  DependentTransformation(letStatements expr expr)
  | DependentForall(casestring expr letStatements expr)
  ;
```

Die Operatordefinitionen stellen gleichzeitig Funktionen zur Verfügung, mit denen Knoten des entsprechenden Typs konstruiert werden können.

Kimwitu++ kann als eine Erweiterung von C++ aufgefasst werden. Die Übersetzung von Kimwitu++-Programmen erzeugt C++-Code. Innerhalb von Kimwitu++-Programmen kann, ähnlich wie in den semantischen Aktionen von Yacc, an bestimmten Stellen C++-Code geschrieben werden. Neben den unten beschriebenen *Rewrite*- und *Unparse*-Regeln gibt es zusätzliche Schlüsselwörter, die in C++-Funktionen benutzt werden können, um Musterüberdeckung durchzuführen.

Muster sehen etwa so aus:

```
DependentForall(*, Variable(v1), *, *)
```

Hier stimmen Muster und konkreter Knoten nur überein, wenn der Knoten die Ausprägung *DependentForall* und sein zweiter Unterknoten die Ausprägung *Variable* hat; die anderen Knoten können eine beliebige Ausprägung haben. Außerdem wird *v1* der Unterknoten dieses Unterknotens zugewiesen. Es wird immer das speziellste Muster ausgewählt.

Rewrite-Regeln haben ein Muster auf der linken Seite und eine Knotenkonstruktion auf der rechten. Wird das Muster überdeckt, dann wird der Knoten während eines *Rewrite*-Vorgangs durch den neu gebildeten ersetzt; dies geschieht so lange, bis kein Muster mehr überdeckt werden kann. Zum Beispiel kann man einen Term, der die Ausprägung *Plus* hat, ersetzen durch einen der Ausprägung *Number*, wenn dieser neue Term gerade die Summe der Teilterme des ersten enthält:

```
Plus(a, b) -> <comp: Number( add(a, b) )>;
```

Unparse-Regeln dienen dazu, den Baum auszugeben (damit quasi das Parsen rückgängig zu machen). Für diesen Zweck muss dem *Unparse*-Vorgang eine Ausgabefunktion mit übergeben werden. Diese kann auch gar nichts machen, so dass man nicht gezwungen ist, wirklich etwas auszugeben. Somit kann der *Unparse*-Vorgang für ein beliebiges Abschreiten des Baumes benutzt werden. Die Voreinstellung für einen Knoten, auf den kein Muster passt, ist, alle Kinder von links nach rechts abzarbeiten. Im Beispiel soll ein Term der Ausprägung *Plus* ausgegeben werden; zu diesem Zweck wird ein Zeichenkettenliteral "+" zusätzlich zu den beiden Untertermen ausgegeben:

```
Plus(a, b) -> [pre: "+" a b ];
Plus(a, b) -> [post: a b "+" ];
```

Sowohl *Rewrite* als auch *Unparse* können in so genannte *Views* gruppiert werden. So ist etwa im obigen Beispiel der *Unparse-View pre* für das Ausgeben in Präfixform, *post* dagegen für die Postfixform verantwortlich.

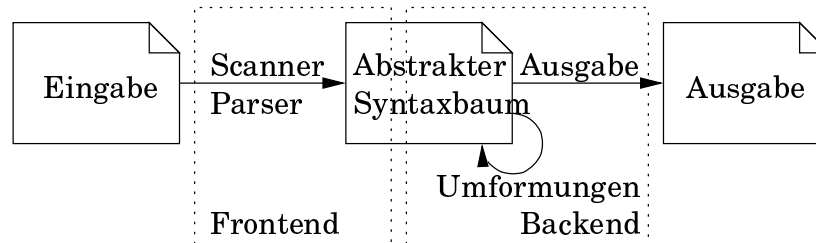


Abbildung 2.1: Überblick des Werkzeugaufbaus

Aufbau der Werkzeuge

Alle Werkzeugketten in SDLC und auch der SDL-Compiler selbst haben den gleichen grundsätzlichen Aufbau. Es gibt eine Trennung in *Frontend* und *Backend*. Diese sind meist in getrennten Programmen untergebracht. Eine Übersicht gibt Abbildung 2.1.

Das Frontend besteht aus einem flex-Scanner und einem Bison-Parser. Letzterer baut (unter Aufrufen des Ersteren) mittels der Operatoren von Kimwitu++ einen Syntaxbaum auf. Der Baum ist eine Instanz von verschiedenen abstrakten Grammatiken, diese sind abhängig von der jeweiligen Werkzeugkette. Er wird dann in einem Kimwitu++-spezifischen Format abgespeichert (welches an das Speicherformat des Synthesizer Generator [Gra97] angelehnt ist). Diese Dateien haben die Endung *.ast* für »*abstract syntax tree*«.

Im Backend werden verschiedene Umformungen an dem wieder eingelesenen Baum durchgeführt; dies geschieht mit *Rewrite*-Schritten in Kimwitu++. Schließlich wird mittels eines *Unparse*-Schritts das Zielformat des entsprechenden Werkzeugs ausgegeben.

2.4 Zusammenfassung der Schritte

2.4.1 Das Entwickeln der AS0

Der erste Schritt bei der Konvertierung in den AS1-Baum ist das Scannen und Parsen eines SDL-Programms, um den AS0-Baum zu erhalten. Hierzu werden drei Dinge benötigt: ein Scanner, ein Parser, und natürlich das Zielformat für den Parser, also eine Datenstrukturdefinition.

Dafür steht aus dem Standard lediglich die Definition der konkreten Syntax von SDL in Textform zur Verfügung.

Das Zielformat ist die Kimwitu++-Darstellung der AS0. Auch diese war jedoch nicht vorhanden, sondern wird erst aus der CS gewonnen. Hier gibt es also eine Werkzeugstrecke, die aus der vorliegenden abstrakten CS hauptsächlich folgendes generiert:

- die AS0-Definition als Text,

- die Kimwitu++-Repräsentation der AS0-Definition,
- einen Scanner und
- einen Parser, der aus einer konkreten CS (also einem SDL-Programm) eine konkrete AS0 (also einen Syntaxbaum des Programms) erzeugt.

Mit den gleichen Werkzeugen kann dann auch sofort die Kimwitu++-Repräsentation der AS1 erstellt werden. Die AS1 selbst liegt fertig vor, sie ist direkt dem Standard entnommen.

Die Regeln, mit denen man aus der konkreten Syntax aus Z.100 die AS0-Definition erhält, konnten nicht aus dem Entwurf des Standards entnommen werden. Sie greifen zu kurz; die erste Aufgabe war es daher herauszuarbeiten, wie die AS0 tatsächlich auszusehen hat. Das Ergebnis wird dann in den Standard einfließen.

2.4.2 Das Schreiben von SATANIC

Der zweite Schritt auf dem Weg von CS nach AS1 ist das Anwenden der Transitionsregeln und der Abbildungsregeln. Die Transitionen werden auf die konkrete AS0 angewandt; sie führen die erweiterten SDL-Konzepte auf die Kernkonzepte zurück, die aber noch in AS0-Schreibweise vorliegen. Die Abbildungen formen dann eine solche »Kernkonzept-AS0« in eine konkrete AS1 um.

Diese Umformungen liegen als Text vor und es muss Programmcode generiert werden, der sie durchführt. Sie benötigen als Hilfsmittel noch die *Funktionen* und die *Bedingungen*. Diese Funktionen sind definiert über AS0 und AS1; sie werden aus in Textform vorliegenden Definitionen generiert. Zum Absichern der semantischen Korrektheit des Programms müssen die Bedingungen erfüllt sein, die als logische Prädikate gegeben sind; Routinen, die sie testen, werden generiert.

Für diese Zwecke gibt es SATANIC. Alle Regeln, Funktionen und Bedingungen liegen in Z.100.F in der gleichen Schreibweise vor und haben auch eine gemeinsame Basis (die ASM), so dass große Teile der Behandlung z. B. von Termen gemeinsam für sie verwendet werden können. Tatsächlich enthält SATANIC keinen eigenen Scanner oder Parser, sondern erhält stattdessen alle Eingaben im internen Kimwitu++-Austauschformat, das in der Werkzeugkette ASM erzeugt wurde (siehe Abschnitt 1.3).

2.5 Abgrenzung der Arbeit im Projekt SDLC

Diese Diplomarbeit entstand im Rahmen des Projektes SDLC. Es ist daher wichtig, abzugrenzen, welche Teile tatsächlich vom Autor stammen und welche aus anderen Quellen.

In Bild 2.2 ist das Projekt SDLC noch einmal in seiner Struktur aufgezeigt. Die gestrichelte Linie umschließt dabei die Teile, die diese Diplomarbeit abdeckt. Zur Erläuterung:

- SATANIC stammt vollständig vom Autor. Es liest seine Eingaben jedoch nicht selbst ein, sondern bekommt einen fertigen Syntaxbaum seiner Eingabe im Kimwitu++-Austauschformat geliefert. Diese Eingabe gehört zur Werkzeugkette ASM und stammt von Andreas Prinz.

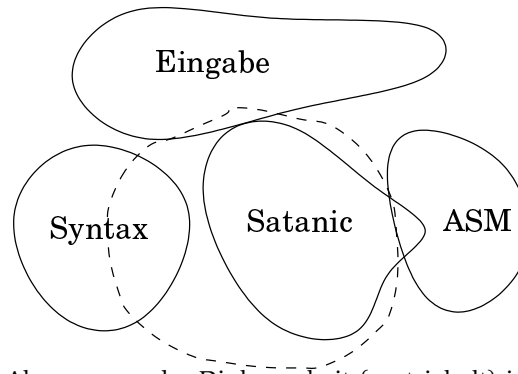


Abbildung 2.2: Abgrenzung der Diplomarbeit (gestrichelt) innerhalb von SDLC

- Die Kette SYNTAX besteht aus etwa 20 Teilprogrammen. Die Scanner und Parser, eine Semantikprüfung für die Eingabe, die Generierung des Scanners für den SDL-Compiler und verschiedene ASM-Teile stammen von Andreas Prinz.
Die Erzeugung von Hilfsdateien für SATANIC, die Generierung der Textform der AS0, der Kimwitu++-Form von AS0 und AS1 sowie des Parsers für die konkrete Syntax stammen vom Autor.

Teil II

Eine AS0-Repräsentation

3 Die AS0

3.1 Vorläufige AS0-Definition des Z.100 Annex F

In Z.100.F wird die AS0 informell definiert.¹⁰ Es heißt, die lexikalischen Einheiten `<name>`, `<quoted operator>`, `<character string>`, `<hex string>` und `<bit string>` seien »wertvoll«, weil sie verschiedene semantische Werte haben können. Es wird eine typografische Konvention geändert: Unterstrichenes der Form `<procedure name>` wird nun als `<procedure<name>` geschrieben, denn alle anderen Definitionen werden auch mit ihrem Namen referenziert, und dieser ist hier einfach nur `<name>`.

Weiter wird überall `<end>` und alles, was mit Makros zu tun hat, entfernt, und schließlich von je zwei lexikalischen Einheiten hintereinander, von denen eine wertvoll ist, die andere aber nicht, die nicht wertvolle gelöscht.

Diese Regeln reichen jedoch für unsere Zwecke noch bei weitem nicht aus. Im Laufe der vorliegenden Arbeit wurde eine neue AS0-Definition geschaffen und für sie Regeln aufgestellt. Der Hauptunterschied zur oben angeführten Definition ist dabei, dass wie in der AS1 unterschieden wird zwischen Sequenzen und Alternativen; eine AS0-Regel kann nur eines von beiden enthalten.

3.2 Sinn der AS0

In Z.100 existieren zwei Grammatiken für SDL.¹¹ Zum einen gibt es eine konkrete Grammatik CS, vollständig mit Begrenzern und allen Schlüsselwörtern. Die dazu gehörige Sprache L(CS) ist die Menge aller rein syntaktisch korrekten SDL-Programme.

Die andere Grammatik ist abstrakt (*Abstract Syntax* im Standard, hier AS1). Sie definiert keine Sprache. Stattdessen repräsentiert die Ableitung aus dem Startsymbol ein dazu äquivalentes SDL-Programm in einem Syntaxbaum. Die AS1 unterscheidet dabei in zwei Arten von Produktionen: Sequenzen und Alternativen, die Tupelbildung respektive Mengenvereinigung definieren. Ein Tupel stellt einen Knoten des Baumes dar. Eine Mengenvereinigung zeigt Typäquivalenzen an, da ein Typ nichts anderes als eine Menge von gerade zulässigen Elementen ist.

Die AS0 soll eine Zwischenstufe zwischen beiden Grammatiken sein. Sie ist abstrakt in dem Sinne, dass sie keine Sprache mehr definiert, sondern die repräsentierende Struktur ausschlaggebend ist. Auch in ihr wird bereits zwischen Tupelbildung und Mengenvereinigung unterschieden. Sie ist eine eindeutige Grammatik, es gibt also immer nur eine Möglichkeit, eine gegebene

¹⁰ Zur Zeit liegt Z.100.F nur als Entwurf vor. In *Draft 11/99* findet sich die Definition in Abschnitt 1.20.2, sie ist nicht endgültig.

¹¹ Eigentlich gibt es noch mehr Grammatiken. Wir betrachten hier als CS eine Textform, die die SDL/PR und SDL/GR mit einschließt.

Spezifikation zu parsen. Wie in der AS1 fehlen viele der semantisch irrelevanten Details. Dazu gehören Begrenzungssymbole und rein lexikalische Regeln.

Diese Unterscheidungen und Streichungen vereinfachen die folgenden Schritte, die in Form von Transformationsregeln und Abbildungsregeln vorliegen, ganz erheblich. Wenn es also relativ einfach ist, ein SDL-Programm in seine AS0 zu überführen, so wird damit der gesamte Prozess der Erzeugung von AS1 erleichtert. Wie gezeigt werden wird, ist dies der Fall. Es lassen sich einige wenige Regeln angeben, mit deren Hilfe man von der CS zur AS0 übergehen kann (Abschnitt 3.3). Aus diesen können Werkzeuge generiert werden, die die verschiedenen AS0-Ausprägungen automatisch erzeugen, also die AS0-Definition aus der konkreten Syntax und einen AS0-Baum aus einer SDL-Spezifikation. Die wesentlich zahlreicheren Transitions- und Abbildungsregeln lassen sich dann deutlich kürzer formulieren.

Ist die AS0 unbedingt erforderlich? Nein, sie vereinfacht nur die Transitionen und Abbildungen. Sie steht allerdings im Standard, deshalb muss sie auch in SDLC benutzt werden.

Dazu ein Beispiel für die möglichen Vereinfachungen. In der Regel

```
<qualifier> ::=
  <qualifier begin sign> <path item> { / <path item> }* <qualifier end sign>
```

erscheinen die Begrenzungssymbole <qualifier begin... und ...end sign>. Sie sind nötig, um ein SDL-Programm eindeutig parsen zu können. (Manche Begrenzer dienen sogar lediglich der Lesbarkeit für den Nutzer.) Da jedoch die AS0 eines konkreten SDL-Programms intern als Syntaxbaum mit typisierten Knoten vorliegt, sind sie überflüssig geworden. Das Trennzeichen / war nur zum Parsen erforderlich und kann jetzt ebenfalls entfallen. Die Regel ändert sich zu

```
<qualifier> ::=
  <path item> <path item>*
```

Jetzt kann auch diese Regel noch einmal vereinfacht werden, indem man zusammenfasst:

```
<qualifier> ::=
  <path item>+
```

Ein Knoten vom Typ <qualifier> hat demnach nur einen Sohn, nämlich eine Liste mit Knoten vom Typ <path item>. Unabhängig von der Bedeutung der folgenden Regel ist die Vereinfachung erkennbar; eine Abbildungsregel aus Z.100.F lautet:

```
<qualifier>(q) => Mapping(q)
```

Müsste die gleiche Regel direkt auf einen der CS entsprechenden Baum angewandt werden, so könnte sie etwa lauten:

```
<qualifier>(*, qh, qt, *) => concat(Mapping(qh), Mapping(qt))
<qualifier gen solidus>(*, q) => Mapping(q)
```

3.3 Concrete Grammar (AS0)

Die Überschrift ist Z.100.F entnommen. Dieser Abschnitt soll die Grundlage für eine Neuformulierung des entsprechenden Abschnitts des Standards sein.

Für die Zwecke der Definition der formalen Semantik benutzen wir nicht die konkrete Grammatik, wie sie in Z.100 definiert ist. Stattdessen wird eine abstrakte Syntax benutzt, die AS0 genannt wird. Die Sprachteile, die vor der AS0 liegen, werden in dieser Formalisierung der Verarbeitung der formalen Semantik nicht betrachtet. Diese sind die lexikalischen Regeln, die Makroregeln und die konkrete Syntax. Es gibt jedoch einen Mechanismus, um die konkrete Syntax auf die AS0 abzubilden. Die unten folgenden 7 Schritte müssen dazu durchgeführt werden.

Folgende lexikalische Einheiten werden a priori als *wertvolle* Nichtterminale angesehen: `<name>`, `<text>`, `<quoted operation name>`, `<character string>`, `<hex string>` und `<bit string>`. Sie sind wertvoll, da sie von verschiedenen Buchstabenfolgen gebildet werden können und dann unterschiedliche semantische Bedeutung haben.

Es ist zu beachten, dass die Darstellung der Syntax sich von der in Z.100 leicht unterscheidet. Ein Nichtterminal mit einem erläuternden, unterstrichenen Teil wie `<procedure name>` in Z.100 wird hier als `<procedure<name>` dargestellt, denn alle normalen Nichtterminale sind Referenzen auf ihre Definition, und die Referenz ist in diesem Fall `<name>`.

1. Entferne überall `<end>` und `<comment>`.
2. Entferne alles, was mit Makros zu tun hat.
3. Entscheide für alle Symbole, ob sie wertvoll sind oder nicht:
 - Für Terminale gilt: Schlüsselwörter sind vorerst wertvoll, einzelne Zeichen nicht. Es existieren nur diese beiden Klassen von Terminalsymbolen.
 - Alle Nichtterminale, die sich auf lexikalische Regeln beziehen, sind nicht wertvoll. Alle anderen Nichtterminale sind wertvoll.
4. Entscheide für alle Regeln, ob sie »Gleichregeln« oder »Doppelpunktregeln« sind. Die »Gleichregeln« haben als Symbol in der AS0 `=`; sie stellen Mengenäquivalenz und -vereinigung (also Typgleichheit) dar. Die »Doppelpunktregeln« dagegen haben als Symbol in der AS0 `::` und stellen Tupelbildung, also Knotendefinitionen für den Syntaxbaum, dar. Alle Regeln, die mehr als eine Alternative haben oder in der einzigen Alternative nur *ein* Nichtterminal enthalten, werden dabei als Gleichregeln betrachtet; alle anderen seien Doppelpunktregeln.
5. Wenn Alternativen und Tupel gemischt sind, oder Tupel innerhalb von Unterausdrücken (also in geschweiften oder eckigen Klammern) auftreten, so erzeuge für alle Tupel eine neue Regel (das heißt ein neues Nichtterminal), und setze dieses anstelle der Tupel ein.
6. Entferne alle irrelevanten Symbole, und zwar:
 - alle Terminale, die nicht wertvoll sind,

- alle nichtwertvollen Nichtterminale, die einem wertvollen Terminalsymbol benachbart sind und
 - alle Terminale und alle nichtwertvollen Nichtterminalsymbole, die in einer Sequenz hinter oder vor einem anderen Symbol (also nicht allein) stehen.
7. Eine Sequenz von Symbolen, gefolgt von einem Unterausdruck, der beliebig oft wiederholt werden kann (mit *) und die gleiche Sequenz enthält, wird ersetzt durch diese Sequenz mit nichtleerer Wiederholung (mit +).

4 Die Werkzeugkette »Syntax«

Die Werkzeugkette SYNTAX ist Teil des Projekts SDLC. Sie geht aus von direkt aus dem Standard mittels Makros extrahierten Dateien; diese enthalten die lexikalischen Regeln, die konkrete Syntax (CS) und die abstrakte Syntax (AS1) von SDL. Daraus erstellt SYNTAX für SLDC die folgenden Teile: einen Scanner; einen Parser; die abstrakte Syntax (AS0) in Textform, so dass sie in den Standard eingehen kann; die abstrakte Syntax (AS0 und AS1) in der Kimwitu++-Form; schließlich einige Hilfsdateien für die Werkzeugketten SATANIC (siehe Kapitel 6) und ASM.

4.1 Rahmen

Alle Programmteile fußen auf einer gemeinsamen Kimwitu++-Repräsentation, die die konkreten und abstrakten Syntaxen darstellen kann.

Die einzelnen Programme von SYNTAX sind als Filter angelegt. Sie lesen von der Standardeingabe und schreiben auf die Standardausgabe. Als Zwischenformat für alle Teile dient das Kimwitu++-Speicherformat. Es werden keine *Pipes* gebildet, sondern alle Zwischenstufen abgespeichert; einige Zwischenformate sind Ausgangsstufe mehrerer Werkzeuge. Obwohl es sich immer um das gleiche Format handelt, tragen die temporären Dateien unterschiedliche Dateiendungen, um ihre Stellung zu verdeutlichen: `.ast` für die noch unbearbeiteten, direkten Repräsentationen der Eingabedateien, `.gst` als gemeinsame Zwischenstufe für die Generierung von Parser und AS0 als Text, schließlich `.kst` als direkte Repräsentation der AS0.

Die Werkzeugkette besteht insgesamt aus etwa 20 Werkzeugen. Eine grobe Übersicht ist in den Abbildungen 4.1 und 4.2 zu finden. Wie dort angedeutet, sind diese 20 Werkzeuge integriert, und es gibt hauptsächlich zwei Programme: Frontend und Backend.

4.1.1 Frontend

Frontend ist für das Einlesen der Eingabedateien (Textform von lexikalischen Regeln, CS und AS1) verantwortlich. Dafür gibt es drei verschiedene Scanner, die allerdings an ein und denselben Parser angeschlossen sind. Ebenso stimmen sie in allen anderen Programmteilen (Hauptprogramm, abstrakte Syntax) überein. Die drei Frontend-Programme `lex2ast`, `cs2ast` und `as12ast` lesen ihre Eingabe und produzieren eine `.ast`-Datei. Diese Programmteile stammen alle von Andreas Prinz.

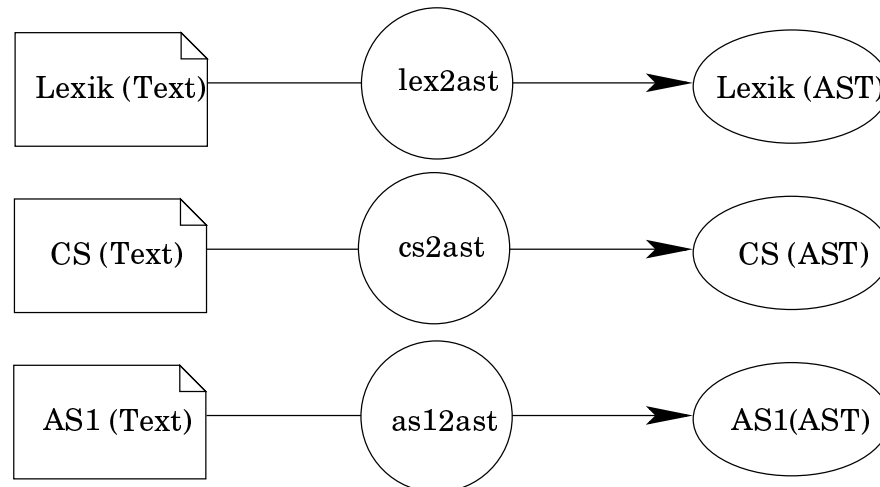


Abbildung 4.1: Übersicht der Werkzeugkette SYNTAX: Frontend

4.1.2 Backend

Alle Backend-Programme sind in einem einzigem Programm zusammengefasst. Backend prüft, mit welchem Namen es aufgerufen wurde (über `argv[0]`), und wählt anhand dessen sein Verhalten.

Die Umformungen an den Syntaxbäumen geschehen mit Kimwitu++. Dort existieren *Rewrite*- und *Unparse*-Regeln, deren *View* dem Namen des Verarbeitungsschritts entspricht. Backend liest als erstes von seiner Standardeingabe mittels der Dateifunktionen von Kimwitu++ den Eingabesyntaxbaum ein. Dann sucht es nach *Rewrite-Views*, die den Namen tragen, mit dem Backend aufgerufen wurde, und zwar mit verschiedenen Präfixen: Als erstes `r_`, existiert dies, dann noch `rzahl_` für beliebig aufsteigende Zahlen. Für jeden gefundenen *View* wird der Syntaxbaum mit jenem *View* umgeschrieben.

Existiert ein *Unparse-View* mit entsprechendem Namen, dann wird der Baum mit ihm ausgeschrieben und die Ausgabe von *unparse* ist auch die Ausgabe des Programms. Anderenfalls ist der Kimwitu++-Baum aus der Dateiausgabefunktion die Ausgabe.

Die Programmteile zur Generierung der Text- und Kimwitu++-Repräsentation, des Parsers sowie der unterstützenden Teile für SATANIC stammen vom Autor, der Rest wiederum von Andreas Prinz.

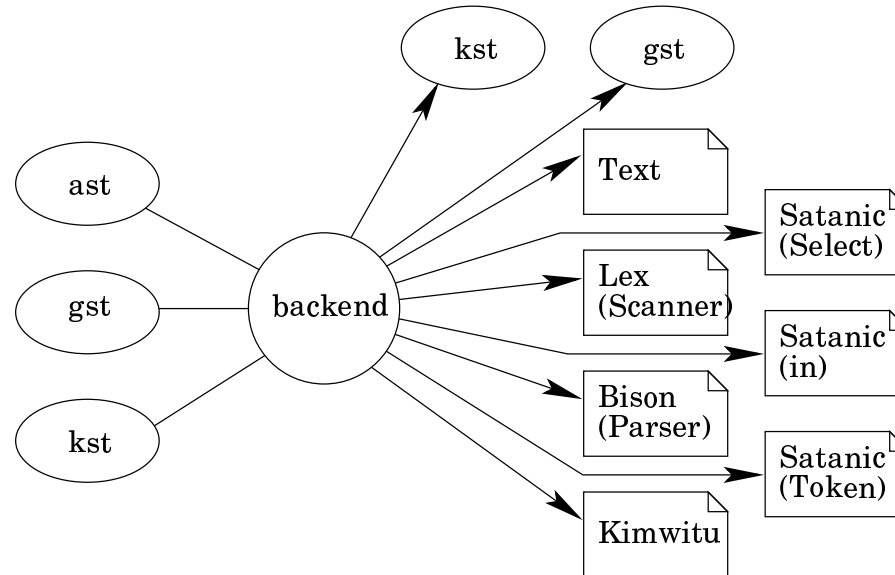


Abbildung 4.2: Übersicht der Werkzeugkette SYNTAX: Backend

4.2 Text- und Kimwitu-Repräsentation

In diesem Programmteil wird der Text der AS0 generiert. Dazu entsprechend entsteht eine Kimwitu++-Definition, die ein Programm in AS0 darstellen kann. Obwohl der Bison-Parser (Abschnitt 4.3) eben diese Kimwitu++-Definition benutzt und konkrete Instanzen von ihr erzeugt, wird er auf einem etwas anderen Wege hergestellt (siehe Abbildung 4.3).

Diese verschiedenen Wege begründen sich in Regel 6 in Abschnitt 3.3, die besagt, dass irrelevante Symbole entfernt werden sollen. Sie sind tatsächlich für die AS0 irrelevant; für den Parser sind sie es nicht. Dieser *parst* nämlich die CS, erzeugt aber die AS0. Deswegen wird das Streichen so weit wie möglich nach hinten verschoben. Das Ergebnis aller ohne Streichen möglichen Umformungen ist die .gst-Zwischenform. Auf ihr beruhen beide Zweige. In einem nächsten Schritt werden dann die Streichungen und finale Umformungen vorgenommen. Aus der damit entstandenen .kst können die Text- und Kimwitu++-Repräsentation ohne weitere *Rewrite*-Schritte direkt per *Unparse* gewonnen werden.

Dabei wird für die Kimwitu++-Form ein einziger Knotentyp generiert, dieses enthält als Operatoren sämtliche Doppelpunktregeln. Die Gleichregeln haben keine Entsprechung: Da sie nur Äquivalenzen von verschiedenen Domänen (Mengen) angeben, erscheinen sie in einem Syntaxbaum nicht mehr.

Die Beschränkung auf einen einzigen Knotentyp hat den Nachteil, dass man etwas Typsicherheit verliert. Ein erster, intuitiver Ansatz wäre vielleicht, die Gleichregeln als Knotentypen zu verwenden, denn in ihnen können ja Alternativen auftreten wie

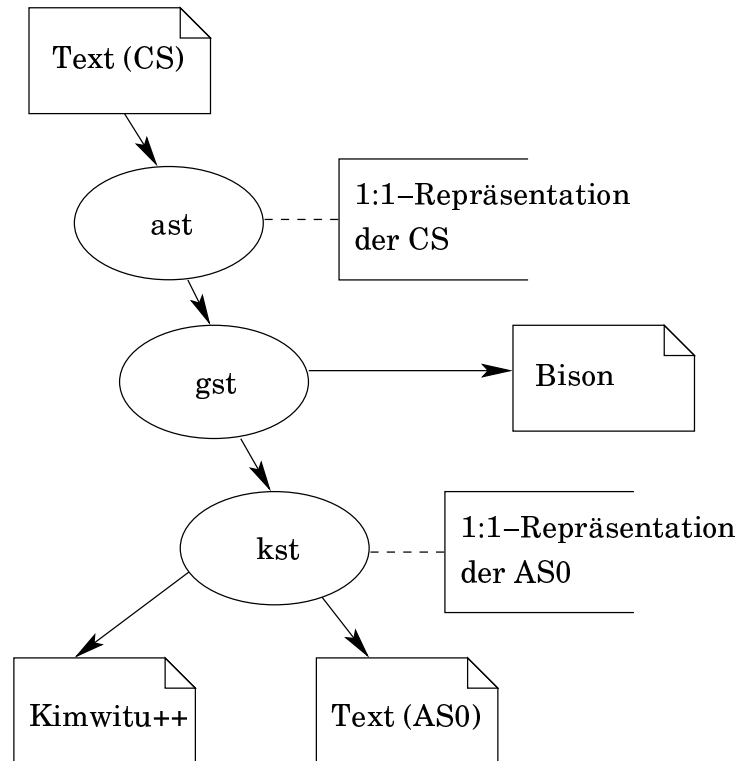


Abbildung 4.3: AS0-Generierung mit Zwischenstufe .kst

auch in den Phylumdefinitionen von Kimwitu++. Zum einen sind jedoch oft die Bestandteile von Alternativen selbst wieder Gleichregeln. Zum anderen kann natürlich ein Nichtterminalsymbol, das eine Doppelpunktregel und damit in Kimwitu++ einen Operator repräsentiert, mehrmals auf der rechten Seite von Regeln auftreten – In Kimwitu++ ist jedoch ein Operator immer Bestandteil genau eines Phylums. Die Typsicherheit ist also mit den verwendeten Werkzeugen nur sehr schwer erreichbar, sie muss später explizit überprüft werden.

4.3 Bison-Parser

Ein SDL-Programm in PR (*Phrase Representation*) liegt in einer Textform vor, die eine Zeichenkette der Sprache L(CS) der kontextfreien Grammatik CS von SDL ist. Es ist leicht, einen Akzeptor für SDL zu generieren; er kann direkt aus der .gst-Form erzeugt werden.

Die Aufgabe für einen Parser ist es, eine interne Darstellung des geparsten Programms zu erzeugen. Diese soll aber nicht eine Repräsentation der CS, sondern vielmehr der AS0 sein, und zwar in Form eines Kimwitu++-Baums. Die Schwierigkeit besteht nun darin, den Akzeptor mit den richtigen semantischen Aktionen zu versehen, um aus der CS direkt AS0 zu erzeugen.

Die Parser-Erzeugung geschieht aus der .gst-Form heraus (siehe Abbildung 4.3). In dieser wurden noch keine semantisch irrelevanten Details gestrichen. Das größte Problem ergibt sich daraus, dass einige der Regeln in Abschnitt 3.3 unterschiedliches Verhalten haben abhängig davon, ob es zu streichende Teile gibt. Zwei Beispiele:

1. Für $\langle nt1 \rangle \langle nt2 \rangle$ muss eine neue Regel erzeugt werden, die $\langle nt1 \rangle \langle nt2 \rangle$ enthält, für $[\langle nt \rangle]$ aber nicht, denn das Komma kann entfernt werden, und $[\langle nt \rangle]$ ist in der AS0 zulässig.
2. Für $\langle nt1 \rangle \{ \langle nt2 \rangle \langle nt3 \rangle \}^*$ muss für $\langle nt2 \rangle \langle nt3 \rangle$ eine neue Regel erzeugt werden, in $\langle nt \rangle \{ \langle nt \rangle \}^*$ ist das nicht nötig, da nach Streichen des Kommas zusammengefasst werden kann zu $\langle nt \rangle^+$.

Diese Fälle müssen unterschieden werden, ohne die Streichungen wirklich vorzunehmen. An einigen Stellen erforderte dies, die tatsächlich auftretenden Fälle mit Spezialfällen in Kimwitu++ abzudecken, weil eine allgemeine Formulierung erheblich höheren Aufwand erfordert hätte. Daher ist an dieser Stelle der Anspruch von SDLC verletzt worden, völlig allgemeingültig und unabhängig von Änderungen an der Eingabesprache zu sein.

Wenn die Spezialfälle nicht ausreichen, gibt es keine versteckten (semantischen) Fehler. Vielmehr treten bei der Generierung des Parsers Fehler auf: Durch das fehlende Streichen versucht der Parser, Knoten mit der falschen Sohnanzahl zu erzeugen. Dies ruft beim Compilieren des Parsers Fehler hervor; der SDL-Compiler wird sich nicht übersetzen lassen. In diesem Fall ist es leicht, einen weiteren Spezialfall hinzuzufügen. Sollte es sich als notwendig erweisen, so kann an dieser Stelle nachgebessert und eine wirklich allgemeine Lösung gefunden werden, jedoch liegt das außerhalb der Aufgaben dieser Diplomarbeit.

4.4 Weitere Hilfsprodukte

SATANIC (Kapitel 6) hat die Aufgabe, die von der AS0 zur AS1 führenden Programmteile von SDLC zu generieren. Dabei hat SATANIC aber lediglich die Übergangsregeln als Eingabe – es weiß nichts über die AS0. Es gibt jedoch einige Stellen, an denen etwas Wissen unbedingt erforderlich ist. Dafür werden hier einige Hilfsdateien produziert.

4.4.1 Select-Funktion

SDLC benötigt eine Funktion zum Auswählen eines bestimmten Unterknotens. Die Benutzung in Z.100.F erfolgt beispielsweise so:

```
a.s-<qualifier>
```

Dies wählt den ersten Unterknoten von `a` aus, dessen Typ `<qualifier>` ist. Diese Wahl kann ein Programm nur treffen, wenn die Struktur der AS0 und der AS1 bekannt ist. Auch wenn es in den Aufgabenbereich von SATANIC fällt, die Funktion zu generieren, so geschieht doch das Wichtigste hier: Es wird eine Struktur aufgebaut, die für jeden Typ alle Subknoten aufzählt.

Dies sieht zum Beispiel so aus:

```
char *s_ASO_identifier[]={"<qualifier>", "<name>", 0};
s__[sel_ASO_identifier]=s_ASO_identifier;
```

Hier enthält das Feld `s__` am Index `sel_ASO_identifier` wiederum ein Feld, das der Reihe nach Strings für die Subknoten enthält. Die aufgebaute Struktur wird in der Select-Funktion benutzt, siehe Abschnitt 6.1.2.

4.4.2 Token

In SDL gibt es lexikalische Regeln, also Nichtterminalsymbole, die als Ableitung lediglich ein Symbol haben, das üblicherweise im Scanner erkannt wird. Diese tauchen in der AS0 nicht als Regeln auf, und es gibt keine Entsprechung in der Kimwitu++-Repräsentation. In den Transition (siehe Abschnitt 8) werden sie dennoch als Konstruktorrufe benutzt. Damit SATANIC erkennen kann, dass es sich um lexikalische Regeln handelt und sich entsprechend verhält, wird in der Werkzeugkette SYNTAX eine Liste aller lexikalischen Regeln und ihrer Ableitung erzeugt.

Diese Liste ist ein assoziatives Feld, der Schlüssel ist ein String. Ein Eintrag in dieser Liste lautet etwa:

```
tokenMap["<asterisk>"]="\*\\";
```

4.4.3 In-Operator

In den Funktionen, Transitionen und Abbildungen, die der SDL-Compiler durchführen soll, kommen Mengen vor, und mit ihnen auch der In- oder Element-Operator. Nun sind einige der vorkommenden Mengen zwar Mengen im mathematischen Sinne, aber keine konkret repräsentierten Mengen während der Berechnung. Dabei handelt es sich um die Domänen, die letztendlich Typen darstellen. Eine Frage `s∈<qualifier>` bedeutet also: Ist `s` vom Typ `<qualifier>`?

Wird der gefragte Typ durch eine Doppelpunktregel definiert, so ist die Frage leicht zu beantworten. Da nämlich jede Doppelpunktregel in Kimwitu++ durch einen Operator repräsentiert wird, kann einfach der Typ des Objekts geprüft werden.

Gleichregeln haben jedoch in Kimwitu++ keine Repräsentation. Deshalb wird in SYNTAX für jede Regel (gleich, ob Doppelpunkt- oder Gleichregel) eine Liste erzeugt, die sämtliche zulässige Operatoren aufzählt. Für Doppelpunktregeln ist die Länge der Liste 1.

Diese Listen sind Kimwitu++-Listen. Sie sind in einem assoziativen Feld gespeichert, dessen Schlüssel ein String ist. Einträge lauten etwa:

```
inMap["<connect list>"]=Consalternative(mkcasestring("<asterisk connect list>"),
    Consalternative(mkcasestring("<name>"), Nilalternative()));
inMap["<terminator statement>"]=Consalternative(
    mkcasestring("<terminator statement>"), Nilalternative());
```

Hier ist ein Knoten vom Typ `<connect list>`, wenn der tatsächliche Typ `<name>` oder `<asterisk connect list>` ist. Der zweite Eintrag ist für eine Doppelpunktregel.

Teil III

Übergang von AS0 zu AS1

5 Spezifikation der Umformregeln

In diesem zweiten Schritt auf dem Weg von einem vorliegenden SDL-Programm zu einer ausführbaren ASM geschieht die Umwandlung eines Programms von seiner konkreten AS0 nach AS1. Wie im Standard [ITU00, Kapitel 1] angeführt, wird zuerst innerhalb der AS0 transformiert und dann auf AS1 abgebildet.

Der erste dieser beiden Schritte ist nötig, um alle erweiterten Konzepte auf die Kernkonzepte von SDL zurückzuführen. Das Ergebnis ist immer noch in Form von AS0, aber die erweiterten Konzepte tauchen nicht mehr auf.

Danach wird eine Abbildung auf AS1 vorgenommen, die fast eins zu eins erfolgen kann.

An jeder Stelle des Umformprozesses muss darauf geachtet werden, dass bestimmte Vor- und Nachbedingungen eingehalten werden. Damit stellt man einerseits sicher, dass die Umformungen auch wirklich korrekt verlaufen; andererseits überprüft man die Korrektheit der statischen Semantik des zu verarbeitenden Programms.¹²

Im Laufe des Prozesses werden viele Funktionen benötigt. Einige werden a priori als vorhanden vorausgesetzt (sie sind im Sinne der ASM vordefiniert), andere werden ebenfalls gegeben. Für den ersten Typ werden Funktionen in Kimwitu++ von Hand geschrieben, der zweite Typ wird aus der Spezifikation generiert.

Alle diese Definitionen liegen in einer einheitlichen Sprache vor. Diese wird im Standard Z.100.F an all jenen Stellen eingesetzt, an denen mathematische Beschreibungen vonnöten sind. Auch die ASM werden dort mit Hilfe dieser Sprache beschrieben.

Die Werkzeuge, die den SDL-Compiler erzeugen, parsen eine Textform der Definitionen, die aus dem Standard automatisch extrahiert wurde. Im folgenden soll diese extrahierte Textform, soweit sie für die Werkzeugkette SATANIC relevant ist, beschrieben werden.

5.1 Bezeichner

Alle Bezeichner tragen zur einfacheren Unterscheidung Präfixe (dies vereinfacht vor allem Scanner und Parser, verringert aber die Lesbarkeit durch Menschen etwas). Es gibt dabei Präfixe für Funktionen (f), die Select-Funktion (s), Domänen (d) und Variablen und Konstanten (a, für »ASM-Namen«). Die Konstruktoren (siehe Abschnitt 5.2.1) bilden eine Ausnahme: Sie gibt

¹² Die Bedingungen und ihre Bearbeitung sind nicht Teil dieser Arbeit, sondern folgen erst in späteren Ergänzungen von SATANIC.

es mit Präfix (mk-d, für die AS1-Konstrukturen) und ohne (für die AS0-Konstrukturen). Einige Beispiele:

f-fullIdentifier	d-EntityKind
s-<qualifier>	mk-d-Number-of-instances
a-x	<number of instances>
a-channelKind	

5.2 Funktionen

5.2.1 Aufruf

Funktionen können einerseits »klassisch« aufgerufen werden, also in der Form Funktionsname, Argumente in Klammern:

```
f-parentAS1ofKind(a-n, d-State-transition-graph)
```

Zum anderen gibt es die Form des Funktionsaufrufs als Postfix-Operator, abgetrennt mit einem Punkt. Diese Notation ähnelt dem Methodenaufruf verschiedener Programmiersprachen und eignet sich besonders für verschachtelte Funktionsaufrufe, weil die Reihenfolge der Ausführung dabei offensichtlicher ist und weniger Klammerebenen verwendet werden (siehe aber auch [Knu73, Kapitel 2.6]). Eine solcherart aufgerufene Funktion hat als erstes Argument den Knoten, an dem es aufgerufen wird:

```
a-i.f-parentAS0.f-findScopeUnit
```

Select

Eine besondere Funktion ist die zur Auswahl eines Unterknotens von einem bestimmten Typ. So wählt

```
s-<qualifier>
```

den ersten Unterknoten aus, der den Typ <qualifier> hat, während

```
s2-<gate constraint>
```

den zweiten Unterknoten vom Typ <gate constraint> wählt.

Es ist möglich, dies auf zweierlei Art zu interpretieren: als eine Menge voneinander unabhängiger Funktionen, oder auch als eine Funktion, die – mit ungewöhnlicher Schreibweise – drei Argumente nimmt, nämlich den Typ des auszuwählenden Knotens, die Nummer unter den möglichen Unterknoten und den Knoten, dessen Unterknoten angefordert werden. Aus dieser zweiten Interpretation ergibt sich eine einfache Umsetzung in eine Kimwitu++-Funktion.

Konstrukturen

Neue Knoten eines bestimmten Typs werden erzeugt, indem der Name des Typs als Funktionsname verwendet wird; die Argumente dieses Funktionsaufrufs sind die Unterknoten des neu zu erzeugenden Knotens. Dieser Aufbau geschieht also

immer *Bottom-Up*, da die Unterknoten bereits existieren müssen. Im folgenden Beispiel wird ein neuer Knoten vom Typ `<save part>` generiert, sein Sohn ist in der Variablen `r` gegeben:

```
<save part>(a-r)
```

5.2.2 Definition

Es gibt fünf verschiedene Arten von Funktionen innerhalb des ASM-Kalküls, in SATANIC wird dabei nur unterschieden, ob die Funktion vom Typ *controlled* ist oder nicht. Eine Funktion, die *controlled* ist, liefert anfangs »undefiniert« zurück, bzw. »falsch« für boolesche Funktionen. Ihr Funktionswert kann aber auch gesetzt werden. Hat sie nur ein Argument (und nur solche treten hier auf), dann wird sie damit zu einem Attribut des Arguments, mit dem sie aufgerufen wird. Ihre Definition lautet z. B.:

```
\dstart controlled f-statesInserted: <variable definition> \rightarrow d-Boolean
```

Dies definiert die »gesteuerte« Funktion `statesInserted`, die für alle Knoten vom Typ `<variable definition>` einen Wert vom Typ `Boolean` liefert.

Die anderen Funktionen sind »abgeleitet«, das heißt ihre Definition benutzt andere Funktionen. Sie haben einen Kopf wie z. B.:

```
\dstart f-myImplicitVariableName(a-v: <variable definition>): <name> =def
```

Dies definiert die Funktion `myImplicitVariableName`, die ein Argument `a-v` vom Typ `<variable definition>` nimmt und einen Wert vom Typ `<name>` zurückgibt. Hinter dem `=def` steht dann ein Term, in dem unter Verwendung von `a-v` ein Wert vom Typ `<name>` gebildet wird.

5.3 Domänendefinitionen

Domänendefinitionen definieren neue Typen. Es handelt sich um Aufzählungstypen, und in der Definition werden alle möglichen Werte aufgezählt. Zum Zeitpunkt der Arbeit, also auf dem Stand von RSDL, gibt es nur eine solche Definition, sie lautet:

```
\dstart d-EntityKind =def
  { a-agentKind, a-agentTypeKind, a-channelKind, a-signalKind,
    a-variableKind, a-remoteVariableKind }
```

5.4 Terme

Die Terme der Eingabesprache sind weitestgehend Terme, wie sie auch in der Sprache der Mathematik gebräuchlich sind. Hinzu kommen einige spezielle Funktionssymbole, die im Folgenden erläutert werden.

5.4.1 If-Term

Es gibt einen bedingten Term mit `if`, `elseif`, `else` und `endif`. Dabei ist `endif` nicht obligatorisch. Dagegen muss immer ein `else` vorhanden sein, da der Wert des Terms sonst undefiniert wäre. Ein Beispiel findet sich im nächsten Abschnitt.

5.4.2 Case-Term

Mit `case`, `of` und `endcase` bildet man einen Term, dessen Wert von einer mit Musterüberdeckung bearbeiteten Eingabe abhängt. Es funktioniert ähnlich wie die `case`-Anweisung in C und noch mehr wie das `with` aus Kimwitu.

```

case a-x of
| a-g=d-Agent-type-definition(*, *, *, *, *, *, *, *, a-graph) =>
    if a-graph \neq a-undefined
      then f-startLabel(a-graph)
      else a-undefined
    endif
| a-g=d-State-transition-graph(a-start, *, *) => f-startLabel(a-start)
:
endcase

```

5.4.3 Mengenbeschreibung

Eine Mengenbeschreibung (*set comprehension*) – und äquivalent eine Listenbeschreibung (*sequence comprehension*) – ist in der Mathematik $\{x|H(x)\}$, wobei H eine Aussageform über x ist. Hier wird die Notation erweitert auf $\{f(x)|x \in X : H(x)\}$, wobei f eine Funktion über x und X eine beliebige Menge¹³ ist. Wenn $f(x) = x$, so kann der erste Teil weggelassen werden. Für Listen werden statt der geschweiften spitze Klammern verwendet.

5.5 Transitionen

Dieser Teil besteht aus einer Menge Transitionen. Diese sind *Rewrite*-Regeln, also Regeln zum Umschreiben von Termen. Das Prinzip ist das folgende: Ähnlich wie bei der Ableitung einer Sprache aus einer Grammatik existiert eine Menge von Regeln mit einer linken Seite, die durch die rechte Seite ersetzt wird. Hat man einen Knoten in der Hand, wird eine Musterüberdeckung des aktuellen Knotens mit den linken Seiten aller Regeln durchgeführt. Die, die am besten passt, kommt daraufhin zur Anwendung. Passt gar keine, gilt der Knoten vorerst als abgearbeitet. Wird einer der Nachkommen umgeschrieben, so muss erneut eine Musterüberdeckung stattfinden. Sind alle Knoten endgültig abgearbeitet, so ist die Termumschreibung beendet.¹⁴

¹³ Zumindest theoretisch. Wir beschränken uns hier auf einige Spezialfälle: Domänen und Mengen, die schon im Programm vorliegen.

¹⁴ Kimwitu++ wählt eine spezielle Reihenfolge für das Abarbeiten; sie stellt sicher, dass immer alle Söhne schon umgeschrieben wurden, bevor ein Knoten an die Reihe kommt.

Die Transitionen beschreiben den Übergang von einem AS0-Baum mit erweiterten Konzepten zu einem AS0-Baum, der sich auf Kernkonzepte beschränkt. Sie werden durch Anwenden der Transitionsregeln auf die Wurzel des Syntaxbaums durchgeführt. Nach vollständigem *Rewrite*-Vorgang ist somit sichergestellt, dass es keine Knoten mehr gibt, die auf die linke Seite einer Transitionsregel passen.

Eine einfache Transitionsregel sieht folgendermaßen aus:

```
<operand5>(<hyphen>, a-x) => <operation application>("-", <a-x>)
```

Dabei trägt der Pfeil, der den Übergang von linker zu rechter Seite kennzeichnet, eine Schrittnummer. Alle *Rewrite*-Regeln finden in nicht spezifizierter Reihenfolge statt. Will man dennoch auf die Reihenfolge Einfluss nehmen, ist es nötig, sie zu Schritten zu gruppieren, die dann einer nach dem anderen durchgeführt werden.

Neben dieser einfachsten Form der Regel gibt es noch mehrere Erweiterungen. Zum einen ist wie auch in Kimwitu++ ein Muster durch ein Prädikat, das heißt einen booleschen Term, erweiterbar. Dieses folgt auf der linken Seite der Regel hinter dem Schlüsselwort *provided*. Diese Regel wird als zum Muster zugehörig angesehen; sie wird bereits dann ausgewertet, wenn festgestellt wird, welche Muster auf den aktuellen Knoten passen.

Zum zweiten gibt es die abhängigen Transformationen. Wann immer die zugehörige Regel gefeuert wird, wird die abhängige Transformation ebenfalls durchgeführt. Dabei handelt es sich nicht um echte Transitionen, auch Muster sind nicht zulässig, sondern nur Zuweisungen.

Zum dritten gibt es die lokale Bindung von Ausdrücken an Variablen mittels *let*.

Ein komplexes Beispiel aus dem Standard folgt. Die spitzen Klammern zeigen an, dass es sich um Ausschnitte aus Listen handelt. Die Regel passt dann, wenn einem Knoten vom Typ *<channel to channel connection>* noch kein *implicit gate identifier* zugewiesen wurde. In diesem Fall wird eine neu generierte *<textual gate definition>* mit in die Liste eingefügt und der *implicit gate identifier* gesetzt.

```
let a-nn=f-newName(a-undefined) in
< a-c=<channel to channel connection>(*, *) >
    provided a-c.f-myImplicitGateIdentifier = a-undefined
=> < a-c, <textual gate definition>(a-nn,
    <gate constraint>(kw-out, f-allSignalsOut(a-c)),
    <gate constraint>(kw-in, f-allSignalsIn(a-c)) ) >
and
a-c.f-myImplicitGateIdentifier:= <identifier>(f-fullPath(a-c), a-nn)
```

6 Die Werkzeugkette »Satanic«

Dieser Generator trägt den Namen *SATANIC*, das steht für »*SDL AS0 To AS1 Nearly Immaculate Converter*«, also »Beinahe makelloser SDL-AS0-zu-AS1-Konverter«. Genaugenommen konvertiert allerdings *SATANIC* nicht von AS0 nach AS1, sondern generiert ein Programm, welches das tut (bzw. die Teile des (R)SDL-Compilers, die das tun).

SATANIC hat keinen eigenen Scanner oder Parser, um seine Eingabe einzulesen. Stattdessen erfolgt die Auswertung aller Eingabedateien in der Werkzeugkette *ASM*, denn diese liegen in der Sprache vor, mit deren Hilfe die *ASM* definiert werden. Diese erzeugt daraus ein *Kimwitu++*-Zwischenformat, welches die Eingabe von *SATANIC* ist.

Dieses Zwischenformat ist wieder eine *.ast*-Datei (siehe auch Abschnitt 9). Eine solche Datei nimmt *SATANIC* also immer als Eingabe, als Ausgabe wird *Kimwitu++* generiert. Folgende Programmteile sind vorhanden:

Mapping für die Abbildung von AS0 nach AS1,

Trans für die Transformationen innerhalb der AS0,

Fun für die im Standard angeführten Funktionen, die zu übersetzen sind,

Cond0 sowie

Cond1 für die Bedingungen (noch nicht implementiert).

Schließlich gibt es noch einige handgeschriebene Funktionen. Diese sind teilweise im Kontext des *ASM*-Kalküls vordefiniert und teilweise Hilfsfunktionen.

Im Gegensatz zu *SYNTAX* wählt *SATANIC* seinen Arbeitsmodus anhand des ersten Parameters aus. Es liest von der Standardeingabe eine *Kimwitu++*-Datei ein und führt dann den angeforderten *Unparse-View* durch. Einige Modi benötigen eigentlich keine Eingabe; aus Konsistenzgründen erhalten sie dennoch eine, die dann verworfen wird.

6.1 Handgeschriebene Funktionen

Die handgeschriebenen Funktionen werden sowohl in den Eingaben von Transitionen, Abbildungen und Funktionsdefinitionen verwendet als auch als Hilfsfunktionen genutzt. Zu ihnen gehören die trivialen Funktionen *head*, *tail*, *take*, *toSet*, *unite*, *concat*, *filter*, *map*, *parentAS0* und *parentAS1*; ferner auch die kaum interessanteren *parentAS0ofKind*, *parentAS1ofKind* sowie *collectRules*, das Mengen von Knoten eines bestimmten Typs aus dem Syntaxbaum zusammensammelt. Größere Beachtung verdienen lediglich *newName* und *Select*.

6.1.1 newName

Diese Funktion soll neue, einzigartige Namen erzeugen. Diese werden benutzt, um an Stellen, an denen im Originalprogramm anonyme Einheiten, wie unbenannte Tore (*Gate*), stehen, Namen zu erzeugen, denn in der AS1 ist diese Anonymität nicht mehr gestattet. Die Funktion `newName` nimmt einen Parameter. Für wiederholte Aufrufe der Funktion mit dem gleichen Parameter soll sie auch das gleiche Ergebnis liefern (sich also wie eine echte Funktion verhalten). Aufrufe mit einem explizit undefinierten Parameter führen dagegen immer zu neuen Namen (da zwei undefinierte Stellen natürlich nicht gleich sein müssen).

Die Funktion merkt sich jeden Aufruf mit einem von »undefiniert« verschiedenen Parameter in einer statischen Variablen, und zwar einer `map<>` aus der Standardbibliothek von C++. Für noch nicht vorgekommene Parameter und für »undefiniert« wird ein neuer Name mit dem Präfix `GEN_` und vier Pseudo-Zufallsbuchstaben erzeugt. Der Parameter und der neu erzeugte Name werden in der `map<>` gespeichert (sofern der Parameter nicht »undefiniert« war).

6.1.2 Select

Das Auswählen von Subknoten anhand des Typs (Abschnitt 5.2.1) kann auf zwei Arten interpretiert werden: Als Funktionsfamilie mit einem Parameter oder als eine Funktion mit drei Parametern.

Eine Funktionsfamilie hat den Nachteil, potenziell sehr viele Funktionen zu erzeugen. Der Vorteil könnte sein, dass diese Funktionen sehr klein sein können. Alle Funktionen haben gemeinschaftlich letztendlich das Wissen über die gesamte Struktur aller Knoten. Sie sind alle zusammen nur dann wesentlich kleiner als eine monolithische Funktion, wenn nur ein kleiner Teil aller möglichen Funktion gebraucht und damit auch generiert wird.

Einerseits weiß man das nicht im Vorherein, andererseits ist eine monolithische Funktion mit Hilfe einer Tabelle sehr einfach zu generieren. Aus diesen Gründen wurde der Ansatz verfolgt, nur eine Funktion zu erzeugen. Aus technischen Gründen (Typsicherheit) gibt es tatsächlich drei Funktionen: Zwei Schnittstellenfunktionen, die die Typkonvertierungen vornehmen und die dritte, die eigentliche, Funktion aufrufen. Die Funktion `Select` erwartet drei Parameter: Die verlangte Nummer und den Typ des Sohnknotens (als Zeichenkette) sowie den Knoten, an dem zu wählen ist.

Aus der Kette `SYNTAX` (in 4.4.1) stammt der eigentliche Kern der Funktion. Das geschieht mittels der Selektoren, die `Kimwitu++` bereitstellt: An jedem konkreten Phylum kann eine Methode namens `prod_sel()` gerufen werden. Sie liefert den Typ des Phylums zurück. Dieser kann als Index in eine Tabelle benutzt werden.

In `SYNTAX` wird ein Feld generiert, in dem es für alle `Kimwitu++`-Operatoren eine Liste mit den Typen der Sohnknoten gibt. In dieser kann die Funktion `Select` nach dem geforderten Typ suchen.

7 Der funktionale Teil

Idealerweise brauchte SATANIC keine separate Behandlung von Funktionen, Abbildungen, Transitionen und Bedingungen; alles baut auf einer gemeinsamen abstrakten Syntax auf. Die benutzte Sprache ist jedoch potenziell sehr mächtig, und der Aufwand für eine wirklich allgemeine Übersetzung in Kimwitu++ ist damit groß. Stattdessen wurde versucht, mit dem Wissen darüber, was in einem bestimmten Teil der Eingangsspezifikation stehen kann, das Programm erheblich zu vereinfachen.

Daraus entsteht gelegentlich eine gewisse Duplizierung von Lösungen. Es erscheint jedoch hier oft einfacher, zwei vereinfachte spezielle statt einer komplexen allgemeingültigen Lösung zu haben.

7.1 Generierte Funktionen

Controlled-Funktionen

Funktionen dieser Klasse stellen im Endeffekt Attribute der Knoten dar, die sie als Argument nehmen. Sie werden dementsprechend auch als Attribute des Phylums erzeugt, das die Knoten repräsentiert. Aus

```
\dstart controlled f-statesInserted: <variable definition> \rightarrow d-Boolean
```

wird dann etwa

```
ASO_rule: { bool m_statesInserted=false; /* <variable definition> */};
```

```
bool& statesInserted(ASO_rule r)  
{ return r->m_statesInserted; }
```

Das ist eine Attributdefinition am Phylum ASO_rule und eine Abfragefunktion. Diese ist keine Methode, damit sie wie eine gewöhnliche Funktion benutzt werden kann. Sie gibt eine Referenz zurück, die ein L-Wert und zuweisbar ist.

Diese Umsetzung bedeutet, dass die Funktion als Definitionsbereich nicht mehr nur <variable definition>, sondern ASO_rule hat. Dies ist keine Einschränkung, sondern eine Erweiterung; manche Fehler können eventuell unbemerkt entgehen.

Andere Funktionen

Jede Funktion besteht lediglich aus einer **return**-Anweisung. Diese enthält die Übersetzung des Terms auf der linken Seite der Funktionsdefinition. Damit wird aus

```
\dstart f-findScopeUnit(a-entity: d-DefinitionASO): d-DefinitionASO =def
  if a-entity = a-undefined then a-undefined
  elseif a-entity \in (<agent type definition> \union <agent definition>)
    then a-entity
  else f-findScopeUnit(a-entity.f-parentASO)
endif
```

der generierte Code

```
ASO_rule findScopeUnit(ASO_rule v_entity)
{
  return (v_entity->eq(ASO_UNDEF()) ? ASO_UNDEF() :
    (in_agent_type_definition(v_entity) || in_agent_definition(v_entity) ? v_entity :
    findScopeUnit(parentASO(v_entity))));
} .
```

Man sieht hier, dass if etc. durch verschachtelte Fragezeichenoperatoren in C++ realisiert werden. Der In-Operator wird durch eine eigens generierte Funktion umgesetzt.

7.2 Das Mapping

Die Mapping-Funktion ist eine normale Funktion. Sie enthält eine große **case**-Anweisung, die direkt in eine sogenannte **with**-Funktion in Kimwitu++ übersetzt wird.¹⁵ Das Wissen darum, dass es sich eben um Mapping() handelt, wird für die folgenden Ergänzungen benutzt:

- Es werden zusätzliche Spezialfälle eingeführt, um die nicht explizit erwähnten Infrastrukturoperatoren UNDEF, CONS und NIL zu verarbeiten.
- Jeder abgebildete, das heißt neu erzeugte AS1-Knoten erhält als zusätzliches Attribut den AS0-Knoten, von dem aus es abgebildet wurde. Das erlaubt später eine Definition der Umkehrfunktion Mapping^{-1} bzw. `inv_Mapping()`.

¹⁵ Diese enthält das Wort **with** gar nicht mehr, sondern es ist implizit, indem einer der Funktionsparameter mit einem \$ gekennzeichnet wird; dieser wird dann zum Argument des impliziten **with**.

8 Die Transitionen

Die Transitionen, die die AS0 mit *erweiterten* Konzepten in die AS0 bestehend aus *Kern*konzepten abbilden, lassen sich (in der Grobstruktur) ganz einfach in *Rewrite*-Regeln für Kimwitu++ umformen. Das ist kein Zufall: Einerseits wurde die Definition der Transitionen durch Kimwitu++ inspiriert; andererseits wurde im Zuge einer Neuveröffentlichung von Kimwitu++ [KC++] durch den Autor jenes erweitert um ein in den Transitionen dringend benötigtes Konzept (**provided**).

Am Ende des Kapitels ist ein komplexes Beispiel aufgeführt.

8.1 Einfache Transitionen

Wie in Abschnitt 5.5 erläutert, sieht eine Transitionsregel abstrakt so aus:

$$\text{LHS [provided COND] } \Rightarrow \text{RHS}$$

Hier wird im Transitionsschritt NO die linke Seite LHS durch die rechte RHS ersetzt, wenn die optionale Bedingung COND wahr ist. Eine *Rewrite*-Regel in Kimwitu++ sieht fast gleich aus:

$$\text{LHS [provided COND] } \rightarrow \langle \text{VIEW: RHS} \rangle ;$$

Dabei wird nur dann ersetzt, wenn gerade der *Rewrite-View* VIEW aktiv ist. Es ist naheliegend, für jeden der oben erwähnten Schritte einen entsprechenden *Rewrite-View* einzuführen.

8.2 Die Elternverknüpfung

Oft wird innerhalb der Transitionen (aber auch im *Mapping* und in den Funktionen) ein Zugriff auf den Vater des gerade aktuellen Knotens benötigt. Dafür existiert in jedem Knoten ein Verweis auf sein Elternteil.

Beim Konstruieren eines Knotens ist der Elternknoten allerdings noch nicht bekannt. Ein Knoten kann ja erst gebildet werden, wenn alle seine Kinder als Argumente für seinen Konstruktoraufwurf bereitstehen. Daher ist es die Aufgabe eines Knotens, bei der Konstruktion allen Kindern mitzuteilen, wessen Unterknoten sie jetzt sind. Dies stellt bei allen Knoten immer eine korrekte Verknüpfung mit den Eltern sicher.

Ein Phylum in Kimwitu++ sollte meist nicht mehr geändert werden. Das betrifft nicht die Attribute, sondern vielmehr die Kinder, also die Baumstruktur. Soll dies doch geschehen, so muss man »von Hand« Sorge tragen, dass die Elternverknüpfung erhalten bleibt.

8.3 Die let-Anweisung

In Transitionsregeln kann eine Anweisung der Form

```
let name=RHS in
```

vorkommen. In der Regel, die der Anweisung folgt, wird dann jedes Auftretens des Bezeichners `name` durch den Term `RHS` ersetzt.

Hierfür gibt es zwei Umsetzungsmöglichkeiten. Zum einen kann man an jeder Stelle, an der der fragliche Bezeichner auftaucht, stattdessen seine RHS einsetzen – dies entspricht genau dem textuellen Einsetzen, das auch mit `let` bezweckt wurde. Die andere Möglichkeit ist, das Ergebnis der RHS an eine temporäre Variable zuzuweisen. Dieses hat den Vorteil, dass RHS nur einmal ausgewertet werden muss – ein großer Vorteil, da einige Berechnungen sehr umfangreich sein können. Aus diesem Grund wurde diese letztere Variante gewählt.

Die konkrete Umsetzung erweist sich als problematisch. Kimwitu++ erlaubt auf der rechten Seite einer *Rewrite*-Regel nur eine eingeschränkte Menge von C++-Ausdrücken, und zwar im Wesentlichen Funktionsaufrufe (auch Methodenaufrufe). Auch wenn es möglich ist, beliebigen Text buchstabengetreu zu übernehmen, wenn er in einfache Anführungszeichen eingeschlossen ist, so wird doch immer noch die Entsprechung der rechten Seite in der anschließenden Übersetzung nach C++ an eine Variable zugewiesen. Die rechte Seite muss also in ihrer Übersetzung ein C++-Ausdruck werden.

Um die Zuweisungen durchzuführen, wird der Kommaoperator von C++ benutzt:

```
a=b, c
```

Das Ergebnis dieses Ausdrucks ist `c`, aber Seiteneffekte (in diesem Fall die Zuweisung von `b` an `a`) entfalten ihre Wirkung. Um die Anführungszeichen zu vermeiden, die nicht zur Übersichtlichkeit beitragen, und den generierten Code leichter lesbar zu gestalten, werden Makros verwendet, die eine Funktionsaufrufsyntax bieten:

```
#define RETURN  
#define LET(x, y) ((x)=(y))
```

Damit kann man schreiben

```
RETURN(LET(a, b), c)
```

Kimwitu++ hält das für normale Funktionsaufrufe, der C++-Präprozessor generiert den obigen Ausdruck daraus. Die »Funktion« `RETURN` kann beliebig viele »Argumente« nehmen.

C++ erlaubt innerhalb von Ausdrücken nicht die Definition von Variablen. Diese müssen also alle vorher schon definiert worden sein. Zu diesem Zweck werden statt der Namen, die in den `let`-Anweisungen standen, globale Namen der Form `let_temp_zahl` benutzt.

8.4 Abhängige Transitionen

Eine Transition kann abhängige Transitionen enthalten. Das sind Zuweisungen, die nur dann erfolgen, wenn ihre zugehörige Transition gefeuert wurde. Sie werden an diese mit dem Schlüsselwort `and` angehängt. Wenn die Transitionsregel »ausgeführt« wird, dann werden auch die Zuweisungen, die abhängig sind, vollzogen.

Dieser Mechanismus ist wesentlich schwieriger umzusetzen als die gewöhnlichen Transitionen, da er von Kimwitu++ nicht unterstützt wird; er kann nur mit einem Trick erreicht werden. Ganz ähnlich wie bei den `let`-Anweisungen wird hier eine Zuweisung in einem Makro versteckt, das wie ein Funktionsaufruf aussieht. Das Makro sorgt außerdem dafür, das die Elternverknüpfungen (siehe Abschnitt 8.2 richtig gesetzt werden:

```
#define DEP(x, y) (fraternize((x),(y)), (x)=(y))
```

Die Funktion `fraternize()` setzt dabei den Elternzeiger des zweiten Elements auf den des ersten, bevor der zweite Knoten umgehängt wird und an Stelle des ersten tritt.

Zum Schluss noch ein umfangreiches Beispiel. Es ist die Übersetzung des Beispiels aus Abschnitt 5.5.

```
ASO_CONS(v_c=ASO_channel_to_channel_connection(*, *), TAIL)
provided (myImplicitGateIdentifier(v_c)->eq(ASO_UNDEF()))
-> <trans_3: RETURN(LET(let_temp_2, newName(ASO_UNDEF())),
  DEP(myImplicitGateIdentifier(v_c), ASO_identifier(fullPath(v_c), let_temp_2)),
  ASO_CONS(v_c,
    ASO_CONS(ASO_textual_gate_definition(let_temp_2,
      ASO_gate_constraint(ASO_TOKEN(mkcasestring("out")), allSignalsOut(v_c)),
      ASO_gate_constraint(ASO_TOKEN(mkcasestring("in")), allSignalsIn(v_c))),
    TAIL)) >;
```

Der Operator `ASO_CONS` wird zur Bildung von Listen verwendet. Sein Verhalten entspricht weitestgehend dem des `Cons`-Operators gewöhnlicher Kimwitu++-Listen. Die lokale Bindung an die Variable `nn` geschieht unter anderem Namen, nämlich `let_temp_2`. Die Mustervariable `TAIL` taucht in der ursprünglichen Regel nicht auf. Da jedoch die spitzen Klammer *Ausschnitte* aus einer Liste darstellen sollen, muss eine neue Hilfsvariable eingeführt werden. Der Operator `ASO_TOKEN` erwartet als Parameter eine Kimwitu++-Zeichenkette, die mit `mkcasestring` erzeugt wird. Abgesehen von solchen syntaktischen Anpassungen gleichen sich Transitionen und *Rewrite*-Regeln sehr stark.

Anhang

A Abkürzungen

- AS0** Abstrakte Syntax 0. Erste Abstraktion der konkreten Syntax von SDL; es fallen die meisten Schlüsselwörter und Sonderzeichen weg.
- AS1** Abstrakte Syntax 1. Die abstrakte Syntax von SDL, in Z.100 einfach »*abstract syntax*« genannt.
- ASM** *Abstract State Machines*. Spezielle Zustandsautomaten. Sie werden genutzt, um die Semantik von SDL zu definieren. Theoretisch benutzt man sie auch dazu, die Umformung von eines SDL-Programms von AS0-Baum zu AS1-Baum vorzunehmen.
- BNF** Backus-Naur-Form. Siehe dazu Anhang B.
- CS** *Concrete Syntax*. Konkrete Syntax.
- ITU-T** *International Telecommunication Union - Telecommunication Standardization Sector*. Aus der CCITT hervorgegangene Standardisierungsorganisation.
- RSDL** *Restricted SDL*. Eine eingeschränkte Teilmenge von SDL. Die graphischen Elemente von SDL fehlen völlig, einige weitere Vereinfachungen wurden vorgenommen. Ziel war ein SDL, in dem schon einfache, aber sinnvolle Beispiele spezifiziert werden können, dessen Umfang aber noch in vertretbaren Rahmen bleibt. Zur Definition von RSDL siehe [Pri00].
- RSDLC** RSDL-Compiler. Ein SDLC für RSDL.
- SDL** *Specification and Description Language*. SDL ist definiert im Standard Z.100 und dessen Anhängen. Im Speziellen ist die Semantik von SDL definiert in Z.100.F.
- SDLC** SDL-Compiler. Der Name des Projektes, in welches diese Arbeit eingebettet ist. Ziel ist ein automatisch generierter Compiler für SDL.

B BNF nach Z.100 und Meta IV

In Z.100 wird eine Sprache zur Syntaxbeschreibung (der konkreten Syntax) genutzt, die als BNF bezeichnet wird. Sie ist allerdings recht verschieden von vielen anderen BNF-Varianten, deswegen wird sie hier kurz erläutert.

Für die abstrakte Syntax verwendet Z.100 dagegen Meta IV. Auch wenn dies so im Standard steht, ist es irreführend. Genaugenommen wird von Meta IV nur ein winziger Bruchteil benutzt, nämlich die Syntax der Domänendefinition.

Ferner wird in dieser Arbeit für die Darstellung der AS0 (als Zwischenstufe zwischen CS und AS1) eine Mischung beider Sprachen genutzt.

B.1 BNF

Das Produktionssymbol in Z.100 ist ::=.

Nichtterminalsymbole werden in spitzen Klammern angegeben, wie zum Beispiel `<block reference>`.

Terminalsymbole haben keine Kennzeichnung, sie werden einfach hingeschrieben, wenn es sich um Schlüsselwörter oder einzelne Zeichen handelt. Es gibt jedoch auch Terminale, die in spitzen Klammern geschrieben werden,¹⁶ so Symbole, die auch in der BNF eine Bedeutung haben wie `<asterisk>`, oder die zusammengesetzt sind wie `<implies sign>`, oder ein Terminal mit Inhalt wie `<name>`. Letzteres darf auch noch zusätzlich eine semantische Zusatzinformation erhalten, wie in `<procedure name>`; solche Terminale werden in dieser Arbeit stattdessen als Nichtterminale betrachtet.

Ein senkrechter Strich trennt Alternativen voneinander ab. Er hat die niedrigste Operatorpräzedenz.

Geschweifte Klammern haben allgemein gruppierende Funktion.

Optionale Teile werden in eckigen Klammern eingeschlossen.

Wiederholungen werden gekennzeichnet mit * für beliebiges oder mit + für mindestens einmaliges Auftreten, und zwar direkt hinter der syntaktischen Einheit, z. B. `<answer part>+` oder `{ <operation def> | <external operation def> }*`.

B.2 Meta IV

Im Kontext dieser Arbeit werden die Meta-IV-Definitionen als Domänen Deklarationen betrachtet. Eine Domäne ist einfach eine Menge. Sie werden als Typdefinitionen benutzt.

¹⁶ Dies erschwert die automatische Verarbeitung wesentlich, da so Nichtterminalsymbole und Terminalsymbole nicht zu unterscheiden sind.

Es gibt zwei Arten von Regeln. Die eine wird mit dem Produktionssymbol = gebildet und drückt Mengenäquivalenz aus. Der andere Regeltyp hat ein :: und bedeutet Tupelbildung.

Alle Mengennamen beginnen mit einem Großbuchstaben, als Trennzeichen zwischen Wörtern kommt der Bindestrich vor, z. B. Channel-path.

In den Regeln kann der senkrechte Strich vorkommen, der das gleiche bedeutet wie in der BNF, aber hier als Mengenvereinigung interpretiert werden kann.

Die Symbole * und + bilden auch hier Listen, als Gruppierungssymbol werden allerdings die runden Klammern gebraucht. Der Operator -set, der direkt an einen Mengennamen angeschlossen wird, bedeutet Bildung einer Menge aus Elementen der Ausgangsmengen und kann hier als Potenzmenge aufgefasst werden.

Schließlich kennzeichnen auch in Meta IV die eckigen Klammern optionale Teile.

Literaturverzeichnis

- CS95 CORBETT, ROBERT; STALLMAN, RICHARD M. *Bison parser generator manual (bison.info)*. Free Software Foundation, Inc., Cambridge, Massachusetts, 1995. Bison 1.25 13, 13
- Dud96 WISSENSCHAFTLICHER RAT DER DUDENREDAKTION (Herausgeber). *Duden – Die deutsche Rechtschreibung*. Mannheim: Bibliographisches Institut & F. A. Brockhaus AG, 21. Auflage, 1996 8
- EB96 VAN EIJK, PETER; BELINFANTE, AXEL. *The Term Processor Kimwitu. Manual and Cookbook*. Universiteit Twente, Enschede, Nederlands, 9. Auflage, Juli 1996 13
- FL91 FISCHER, CHARLES N.; LEBLANC, JR., RICHARD J. *Crafting a Compiler with C*. Redwood City: Benjamin/Cummings, 1991. ISBN 0-8053-2166-7 8
- Gra97 GRAMMATECH, INC. »Generation of Language-Based Editing Environments«, 1997
URL <http://www.grammatech.com/products/sg> 15
- ITU99 »ITU-T Recommendation Z.100: Languages for telecommunications applications - Specification and Description Language«. ITU-T-Empfehlung, 1999 2
- ITU00 »ITU-T Recommendation Z.100 Annex F: Languages for telecommunications applications - CCITT Specification and Description Language«. ITU-T-Empfehlung, 2000¹⁷ 2, 31
- KC++ PIEFEL, MICHAEL; VON LÖWIS, MARTIN. »Kimwitu++«, 2000
URL <http://site.informatik.hu-berlin.de/kimwitu++> 13, 13, 40
- Knu73 KNUTH, DONALD E. *The Art of Computer Programming: Fundamental Algorithms*, Band 1. Reading, Massachusetts: Addison-Wesley, 1973. ISBN 0-201-03809-9 32
- Mas00 MASLOV, VADIM. »BtYacc: BackTracking Yacc«, März 2000
URL <http://www.siber.com/btyacc> 13
- Par00 PARR, TERENCE. »ANTLR Website«, Februar 2000
URL <http://www.antlr.org> 13

¹⁷ Dieses Dokument ist zum Zeitpunkt der Fertigstellung dieser Arbeit noch nicht erschienen und lag in einer Arbeitsversion vor.

- Pax95 PAXSON, VERN. *flex — fast lexical analyzer generator (flex.info)*, 1995. Flex 2.5 13, 13
- Pie99a PIEFEL, MICHAEL. »Exemplarische Konvertierung eines SITE-Tools von Kimwitu nach Kimwitu++«. Studienarbeit, Humboldt-Universität zu Berlin, August 1999 13
- Pie99b PIEFEL, MICHAEL. »From Kimwitu to Kimwitu++«, Juni 1999. Preliminary Kimwitu++ documentation 13
- Pri00 PRINZ, ANDREAS. »Formal Semantics for RSDL: Definition and Implementation«. Habilitationsschrift, Humboldt-Universität zu Berlin, Juni 2000 5, 44
- Str97 STROUSTRUP, BJARNE. *The C++Programming Language*. Reading, Massachusetts: Addison-Wesley, dritte Auflage, 1997. ISBN 0-201-88954-4 13

Thesen zur Diplomarbeit

Das Projekt SDLC beschäftigt sich mit der Übersetzung eines SDL-Programms. Das Zielformat ist ein ASM-Programm (*Abstract State Machines*). Wichtige Zwischenformate auf dem Wege dorthin sind eine Abstraktion der konkreten Syntax namens AS0 und die im Standard Z.100 beschriebene *Abstrakte Syntax*, genannt AS1.

Hauptthese

- Ein SDL-Compiler (bzw. die Stufe des SDL-Compilers bis hin zur AS1) kann *automatisch* aus dem Text im Standard *generiert* werden kann.

Untergeordnete Thesen

- Das Werkzeug Kimwitu++ (es erzeugt Syntaxbäume, formt sie um und gibt sie aus) ist geeignet, Programme zu schreiben, die den SDL-Compiler zu generieren; es eignet sich auch als Implementationssprache für den SDL-Compiler selbst.
- Die Behandlung folgender Programmteile des SDL-Compilers können in relativ unabhängigen Werkzeugketten implementiert werden: Gewinnung der AS0 und Übergang von AS0 zu AS1
- Die erste Kette erzeugt einen Scanner und einen Parser. Ein Bison-Parser, der die konkrete Syntax einliest und einen Baum in AS0 aufbaut, kann inklusive der semantischen Aktionen automatisch generiert werden.
- Die zweite Kette erzeugt Programmteile für Transitionen und Abbildungen sowie den benötigten zusätzlichen, generierten Funktionen:
 - Transitionen entsprechen weitestgehend *Rewrite*-Regeln in Kimwitu++ und können als solche generiert werden.
 - Die Abbildung entspricht einer Kimwitu++-Funktion mit einem Musterargument.
 - Funktionen werden zu einfachen Kimwitu++-Funktionen, die nur aus einer *return*-Anweisung bestehen.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig, ohne unzulässige Hilfe und nur unter Nutzung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Ich erkläre mich damit einverstanden, dass meine Diplomarbeit öffentlich in der Universitätsbibliothek ausgestellt wird.

Berlin, 16. August 2000

Michael Piefel