

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

Studienarbeit

Exemplarische Konvertierung eines SITE-Tools von Kimwitu nach Kimwitu ++

Michael Piefel

30. August 1999



Der Termprozessor Kimwitu bietet Erweiterungen zu C an. Diese erleichtern die Erstellung von Programmen, die Datenstrukturen verwenden, die Bäume aus typisierten Knoten mit variabler Sohnanzahl sind. Das prominenteste Beispiel ist ein dynamischer Syntaxbaum. Kimwitu wird am Lehrstuhl für Systemanalyse eingesetzt für die Entwicklung der SITE-Tools.

Kimwitu++ beruht auf C++. Das erweitert nicht nur die Möglichkeiten, sondern es erfordert auch eine leicht abgewandelte Eingabesyntax. Die Aufgabe dieser Studienarbeit besteht nun darin, eines der Tools aus SITE von Kimwitu nach Kimwitu++ zu portieren. Die dabei gewonnenen Erfahrungen sind nützlich bei der Entwicklung eines Werkzeugs zur (halb-)automatischen Konvertierung bestehender Programme.

Inhaltsverzeichnis

1 Die Ausgangslage	4
2 Vorüberlegungen	6
2.1 Funktionsdefinitionen	6
2.2 Views	6
2.3 Phylum int und float	6
2.4 Methoden statt Funktionen	7
3 Konvertierung und aufgetretene Probleme	8
3.1 Vorüberlegtes	8
3.2 Weitere Änderungen	9
3.3 Die error-Symbole	11
A Tabellen	16
A.1 Quellengröße	16
A.2 Ausführbare Dateien	17
B From Kimwitu to Kimwitu++	19
Literaturverzeichnis	20

1 Die Ausgangslage

Am Lehrstuhl für Systemanalyse der Humboldt-Universität wird seit längerem Kimwitu [vEB96] als Werkzeug eingesetzt; dieses ist ein Werkzeug, das mittels einer Spracherweiterung von C erlaubt, Programme zu schreiben, die Datenstrukturen verwenden, die Bäume aus typisierten Knoten mit variabler Sohnanzahl sind. Nachdem der Wunsch aufgekommen war, statt mit C mit C++ zu interagieren, wurde nun ein abgewandeltes Kimwitu, nämlich Kimwitu++ (vgl. Anhang B) entwickelt, das objektorientiert arbeitet und auf C++ basiert.

Es werden aber kaum neue Komponenten entworfen, sondern es gibt eine große Codebasis, die auch weiterhin verwendet werden muss, für die bereits bestehenden Werkzeuge. Es ist daher erforderlich, Programme zu konvertieren, so dass sie Kimwitu++ statt Kimwitu benutzen. Ein möglichst nahtloser Übergang steht dabei im Vordergrund.

Im Folgenden geht es um die exemplarische Konvertierung eines ausgewählten Programms. Die Wahl fiel dabei auf gconv (siehe auch [Dum96] und [Dum99]), ein am Lehrstuhl viel benutztes Programm, das, teils aus historischen Gründen, auch als yacc2html bekannt ist. Gründe für diese Wahl waren:

Größe Das Programm hat eine Größe von etwa 8 000 Zeilen, davon 2 600 im Kimwitu-Quelltext, und ist damit relativ klein, was eine manuelle Konvertierung stark vereinfacht.

Andererseits ist es auch nicht zu klein, also kein reines Spielzeug. (Eine genaue Aufstellung über den Umfang des Ausgangsquelltextes befindet sich im Anhang A.)

Abwechslung In gconv kommen trotz seiner geringen Größe viele der Elemente von Kimwitu vor, während einige der SITE-Werkzeuge von den Sprachmerkmalen nur sehr eingeschränkten Gebrauch machen. Die damit zu sammelnden Erfahrungen wären weit geringer, als bei gconv.

Gconv ist teilweise ein Projekt zum Sammeln von Programmiererfahrungen gewesen und daher in seiner Benutzung von noch wenig verbreiteten Werkzeugen und Merkmalen dieser und auch althergebrachter Werkzeuge vielseitiger und anspruchsvoller. Zudem ist es, so weit möglich, bereits in C++ geschrieben und kann daher von der Verwendung von Kimwitu++ noch mehr profitieren.

Verifizierbarkeit Die Korrektheit von gconv lässt sich relativ leicht überprüfen, da es ein einfacher Filter ist. Bezogen auf die gleiche (in Textform vorliegende) Eingabe muss das konvertierte Tool genau die gleiche Ausgabe

liefern wie das Original.¹

Die nunmehr genauer formulierte Aufgabenstellung lautet also, das u. a. in Kimwitu geschriebene Programm `gconv` so abzuändern, dass es stattdessen Kimwitu++ benutzt. Dabei geht es nicht um das Ausreizen von zusätzlichen Möglichkeiten, sondern um die minimale Menge nötiger Änderungen, damit im Makefile `kc4` (die letzte Version des Kimwitu-Compilers) durch `kc++` ersetzt werden kann.

Die Aufgabenstellung ergibt sich aus dem Blick nach vorn: Soll Kimwitu++ in Zukunft benutzt werden, so müssen auch größere Projekte mit vertretbarem Aufwand umgestellt werden können. Erst dann kann daran gedacht werden, die gewonnenen Vorteile zu nutzen.

Diese Studienarbeit entsteht im Hinblick auf eine mögliche (halb-)automatische Konvertierung von Projekten. Dies ist ein weiterer Grund dafür, die Änderungen auf das Nötigste zu beschränken und sie konsistent zu halten.

¹Damit ist natürlich keine Aussage über die Korrektheit des Programms, sondern nur der Konvertierung gegeben.

2 Vorüberlegungen

Die Dokumentation zu Kimwitu++ besteht (mit Ausnahme der Originaldokumentation zu Kimwitu) nur aus [Pie99]. Dort werden einige Unterschiede der beiden Programme aufgeführt. An dieser Stelle sollen bereits die zu erwartenden Änderungen aufgeführt werden. Welche Arbeiten dann schließlich tatsächlich anfallen, wird später behandelt.

2.1 Funktionsdefinitionen

Kimwitu akzeptierte als Eingabe noch sowohl die ältere Art der Funktionsdefinition nach Kernighan und Ritchie als auch die neuere, ursprünglich von C++ stammende und schließlich in ANSI- und später ISO-C [C90] festgeschriebene. Da Kimwitu++ in einem C++-Kontext benutzt werden soll, werden nur noch ISO-C++-konforme [C++98] Funktionsdefinitionen (also im neuen Stil) akzeptiert. Sollten noch alte Definitionen vorhanden sein, müssen sie ersetzt werden.

2.2 Views

Während für Unparse-Regeln schon immer Views benutzt wurden, wurde diese Möglichkeit für Rewrite-Regeln erst vor relativ kurzer Zeit eingeführt. Damit sollte sich das Schlüsselwort und der Typname `view` zu `uview` ändern, ein neues Schlüsselwort und Typ `rview` wurde neu eingeführt. In Kimwitu war der alte Name aus Kompatibilitätsgründen noch zugelassen, in Kimwitu++ ist er es nicht, etwaiges Auftreten von `view` muss also ersetzt werden.

2.3 Phylum `int` und `float`

Kimwitu verfügt über fünf vordefinierte atomare Phyla. Auf alle nichtatomaren Phyla sowie auf `casestring` und `nocasestring` wird über Zeiger zugegriffen. Die Phyla `int`, `float` und `voidptr` werden jedoch direkt auf ihre C-Äquivalente abgebildet. Während also der Inhalt einer als `casestring cs`; definierten Variable `cs->name` ist, ist der Wert von `int i`; nach wie vor einfach `i`. Das betrifft auch Zuweisungen.

In Kimwitu++ ist dagegen jedes Phylum ein Objekt. Um den Unterschied deutlich zu machen, heißen die beiden arithmetischen atomaren Phyla jetzt `integer` und `real`. Auf ihren Wert kann jeweils mit `var->value` zugegriffen

werden. Zur Konstruktion einer solchen Variable muss `mkinteger()` bzw. `mkreal()` benutzt werden.

Dieses ist vermutlich das schwierigste Problem bei der Konvertierung. Zuweisungen der Art `i=5;` und `i=0;` müssen ersetzt werden durch `i=mkinteger(5);` und `i=mkinteger(0);`. Während jedoch in dem Fall, dass eine Stelle übersehen wurde, der Compiler die erste Zuweisung anmahnen würde, wäre es im zweiten Fall die legale Zuweisung eines Nullzeigers. Es ist darüber nachzudenken, Kimwitu++ dahingehend abzuändern, dass `integer` ein *Smartpointer*¹ ist, der solche Zuweisungen abweist.

2.4 Methoden statt Funktionen

In Kimwitu++ ist jeder Phylumtyp eine Klasse in C++ und damit jedes Phylum ein Objekt. Operationen auf einem Phylum werden deshalb auch nicht mehr als Aufruf einer globalen Funktion, die das zu bearbeitende Phylum als Parameter übergeben bekommt, umgesetzt, sondern als Methodenruf am Objekt selbst. Damit werden Konstruktionen wie

```
unparse_phylumdeclarations( Thephylumdeclarations, v_null_printer, view_init_stacks );  
kc_answer = copy_int( kc_line.yt_int, False );
```

zu

```
Thephylumdeclarations->unparse( v_null_printer, view_init_stacks );  
kc_answer = kc_line.yt_int->copy( false );
```

Dies ist eine sehr einfache, aber arbeitsaufwendige Änderung. Leider besteht Softwareproduktion, wie auch Brooks in [Bro87] schreibt, zu großen Teilen aus solchen Arbeiten.

¹Also ein Typ, der nach außen größtenteils wie ein normaler Zeiger arbeitet. Dereferenzierung mittels `*` und `->` muss funktionieren, aber Zuweisung ist z. B. nur von anderen integer-Variablen möglich.

3 Konvertierung und aufgetretene Probleme

3.1 Vorüberlegtes

Zuerst wurden die in Kapitel 2 vorausgesehenen Änderungen durchgeführt.

3.1.1 Funktionsdefinitionen

Das Programm `gconv` ist ein relativ junges Programm, und verwendet daher fast ausschließlich Funktionsdefinitionen im neueren C++-Stil. Lediglich die Datei „`ebnf2yacc.k`“ enthält einige K&R-Definitionen, die von Hand geändert wurden.

Diese Anpassung sollte in einer Konvertierung immer am Anfang stehen. Während für viele andere Fehler aussagekräftige Fehlermeldungen bereitstehen, ist Kimwitu++ hier nicht dazu in der Lage und findet keinen Ansatzpunkt für eine weitere Analyse des Quelltextes mehr. Das Ergebnis sind zahlreiche inkorrekte Fehlermeldungen, nur die erste ist ein Hinweis auf den wahren Fehler. Hier sollte die Fehlerstabilisierung von Kimwitu++ nachgebessert werden.

3.1.2 Views

Diese Anpassung (vgl. Abschnitt 2.2) war eine einfache Suchen-und-Ersetzen-Aktion.

Es stellte sich jedoch heraus, dass in Kimwitu++ das (nicht dokumentierte, aber in „`debug.k`“ benutzte) Feld `kc_view_names`, welches eine Liste aller Namen der Unparse-Views ist, nicht wie erwartet in `kc_uview_names` umbenannt wurde. Hier ist Kimwitu++ nachzubessern.

3.1.3 Phylum `int` und `float`

In `gconv` wird kein `float` verwendet. `int` tritt einmal auf, und zwar im Operator `TokenPos` des Phylums `token_pos`. Es wurde hier in `integer` überführt.

Das zog Änderungen an vier weiteren Stellen nach sich: zwei eingefügte `mkinteger` und zwei Ergänzungen mit `→value`¹. Insgesamt waren die Änderungen sehr übersichtlich.

3.1.4 Methoden statt Funktionen

Im Zuge der (an sich trivialen) Umschreibung fiel auf, dass die Funktion `concat` keine Methode innerhalb der Klassen ist. Deshalb ist die Schreibung `→concat(liste2)` statt des erwarteten

```
liste1 →concat(liste2);
```

vielmehr

```
concat(liste1, liste2);
```

Selbst wenn `concat` keinen Zugriff auf private Daten der Klasse benötigte, wäre `concat` als Methode zu bevorzugen, um Einheitlichkeit zu wahren. Eventuell könnten beide Varianten angeboten werden. Hier ist `Kimwitu++` nachzubessern.

3.2 Weitere Änderungen

Nachdem diese Änderungen vollständig durchgeführt worden waren, konnte `make` aufgerufen werden. Wie erwartet liefert `Kimwitu++` selbst zu diesem Zeitpunkt keine Fehlermeldungen mehr. Die erhoffte (aber nicht für wahrscheinlich gehaltene) völlige Fehlerfreiheit gab es jedoch nicht, weshalb weitere Modifikationen erforderlich wurden.

3.2.1 Das Makefile

`Kimwitu++` erzeugt im Gegensatz zu `Kimwitu` keine C-, sondern C++-Quellen. Das muss sich auch im Übersetzungsprozess widerspiegeln. Das Makefile wurde also folgendermaßen angepasst:

- Als Übersetzer für die `Kimwitu`-Quellen wurde `kc4` durch `kc++` ersetzt.
- Die erzeugten Quellen statt der Endung `„.c“` die Endung `„.cc“` in den Variablen `C_KFILES` und `KW_C_FILES`.
- Der Vollständigkeit halber wurden diese beiden Variablen noch umbenannt...
- ...und sie wurden bei den Variablen `CFILES` und `CCFILES` an die entsprechenden Stellen verschoben.

¹Hier hätte auch stattdessen eine Dereferenzierung der Variablen (`pos`) genügt, also `*pos`. Diese Schreibweise trägt aber nicht immer zur Lesbarkeit bei.

3.2.2 Boole'sche Variablen

Kimwitu stellt einen (undokumentierten) Typ namens `boolean` zur Verfügung, der die Werte `True` und `False` annehmen kann.² Dieser Typ existiert in Kimwitu++ nicht mehr. Stattdessen kann man den C++-Standardtyp `bool` mit den Literalen `true` und `false` benutzen.

Diese Änderung ist ein einfaches Suchen und Ersetzen.

3.2.3 Namenskonflikte

In Kimwitu++ gibt es für jedes Phylum $\langle phylum \rangle$ einen Typ $\langle phylum \rangle$ zum Zugriff auf konkrete Phyla sowie einen Typ `c_` $\langle phylum \rangle$, der aber nur zum Lesen (also `const`) ist.

In `gconv` gibt es nun ein Phylum `comment` sowie, in der lexikalischen Analyse, eine Startbedingung (siehe dazu [Coë93] und [Pax95]) namens `c_comment`. Dies führt zum Konflikt. Diese Startbedingung, wie auch – konsistent – die anderen mit `c_` beginnenden Bedingungen entsprechend, wurde in `C_comment` umbenannt.

Ein ähnlich gelagertes, aber hier nicht aufgetretenes und auch eher theoretisches Problem ist eine Definition:

```
fred:
  Fred( ... )
;
```

Ein Phylum `fred_Fred` kann jetzt nicht mehr definiert werden.

3.2.4 Zugriff auf die Union

In Kimwitu hat jedes Phylum eine Union `u`, welche in ihren Alternativen die Subphyla für die verschiedenen Operatoren hat. Das ist nicht dokumentiert, wird aber manchmal dennoch verwendet. Diese Union existiert zwar momentan in Kimwitu++ noch, doch es ist damit zu rechnen, dass sie, da sie überflüssig ist, wegfällt. In jedem Fall kann nicht auf sie zugegriffen werden, wenn der Typ des Phylums nicht genau bekannt ist.

In Kimwitu++ gibt es für jeden Phylumtyp eine Klasse, von der wiederum Klassen für alle Operatoren des Phylums abgeleitet sind. Üblicherweise besitzt man jedoch immer nur einen Zeiger auf die Phylumklasse. Da die Subphyla (in der Union) erst in den Operatorklassen definiert sind, muss man zum Zugriff auf die Union eine korrekte Typumwandlung durchführen. Aus

```
a->u.CAction.casestring_1 = mkcasestring(new_text);
```

²Natürlich handelt es sich intern nur um einen anderen Namen für den Typ `int`.

wird dadurch

```
dynamic_cast<kc_tag_rule_tail_part_CAction*>(a)->u.CAction.casestring_1  
= mkcasestring(new_text);
```

Das ist viel umständlicher. Der direkte Zugriff auf die Subphyla ist aber eine etwas unsaubere Technik (auch wenn es sich nicht immer vermeiden lässt), so dass es auch unsauber aussehen darf. In vielen Fällen gibt es eine bessere Möglichkeit: die Verwendung von Mustern. In `foreach`- und `with`-Anweisungen, Funktionen mit `$`-Parameter sowie `Unparse`- und `Rewrite`-Regeln kann man ein genaues Muster angeben und somit eine benannte Variable handhaben. Ein weiterer Vorteil: Ändert man oben die Definition von `CAction` so ab, dass es ein weiteres `casestring`-Subphylum noch vor dem bislang ersten gibt, so verschieben sich die anderen nach hinten. In der obigen Zeile wird vom Compiler kein Fehler gefunden.

Im Beispielfall lässt sich der direkte Zugriff aber nicht vermeiden, da in `Kimwitu++` die Variablen nach Musterüberdeckung einen Nur-Lese-Zugriff haben. Da der Zugriff auf Umwegen möglich ist, ist dieser Nur-Lese-Zugriff wirkungslos. Er erzwingt unsaubere und möglicherweise unrichtige Konstrukte. Hier ist eventuell `Kimwitu++` nachzubessern.

3.3 Die error-Symbole

Ein ausführlicher Test an dieser Stelle brachte das Ergebnis, dass das konvertierte Programm tatsächlich genau die gleiche Ausgabe liefert wie auch das Ausgangsprogramm. Dabei gab es jedoch eine Ausnahme.

Es gibt `Yacc`- bzw. `Bison`-Regeln, die zur Fehlerstabilisierung dienen (siehe auch [CS95]). Diese enthalten das spezielle Symbol `error`. Da sie nicht eigentlicher Bestandteil der Grammatik sind, bietet `gconv` die Möglichkeit, Alternativen, die `error` enthalten, auszufiltern und in der Ausgabe wegzulassen.

Das in `Kimwitu++` geschriebene `gconv` beließ jedoch diese Alternativen mit im Text, die Kommandozeilenoption `--error` (siehe [Dum99]) blieb ohne Wirkung. Die Ursache dafür war, dass eine Zuweisung `*rule_t = *tl;` scheinbar nicht ausgeführt wurde. Das offenbart ein grundlegendes Problem, welches aus dem Ausnutzen internen Strukturen von `Kimwitu` erwächst, die in `Kimwitu++` nicht mehr so vorhanden sind: die Modifikation bestehender Phyla.

3.3.1 Veränderliche Phyla

Phyla, die das Attribut `uniq` (siehe [vEB96]) haben, müssen auch (zumindest konzeptuell) konstant sein. Ein Beispiel mag das verdeutlichen:

```
struct Angestellter
{
    :
    integer alter;
    integer raumnr;
    :
}
:
a.alter=mkinteger(42);
a.raumnr=mkinteger(42);
:
if (today == a.geburtstag)
    a.alter->value++;
```

Der Angestellte ist jetzt nicht nur ein Jahr älter geworden, sondern auch umgezogen! Die Zuweisungen hätte man äquivalent schreiben können als:

```
integer Zweiundvierzig=mkinteger(42);
a.alter=Zweiundvierzig;
a.raumnr=Zweiundvierzig;
if (today == a.geburtstag)
    Zweiundvierzig->value++;
```

Hier wird deutlich, dass Phyla, die `uniq` definiert sind, nicht unabhängig von ihrem Inhalt existieren, sondern dass sie in gewisser Weise selbst definierte Literale darstellen, die nicht verändert werden dürfen.

Phyla, die nicht `uniq` sind, unterliegen dieser Beschränkung nicht. Es ist durchaus legitim, ihren Inhalt zu ändern. Dabei ist zu beachten, dass, wie immer beim Überschreiben von Zeigern, Speicherlöcher entstehen können, wenn man den letzten Zeiger auf ein Speicherobjekt überschreibt. Gelegentlich möchte man nicht ein einzelnes Attribut oder Subphylum, sondern gleich ein ganzes Phylum gegen ein anderes austauschen. Es wäre vielleicht die sauberste Lösung, stattdessen alle auf das Phylum verweisenden Zeiger umzustellen. Das ist jedoch nicht immer einfach möglich, wenn man etwa viele solcher Verweise hat oder über die Verweise nicht auf Buch geführt wird.

Man kann in diesem Fall in Kimwitu einfach schreiben `*ziel=*quelle;`, vorausgesetzt, dass `ziel` und `quelle` den gleichen Typ haben. Dabei werden nicht nur alle Zeiger auf Subphyla überschrieben, sondern eventuell sogar die Ausprägung des Phylums, der Operator, geändert.

Genau das ist in Kimwitu++ nicht möglich. Wie schon in Abschnitt 3.2.4 ausgeführt, sind im erzeugten C++-Quelltext Operatoren von dem dazugehörigen Phylum abgeleitete Klassen. Es ist natürlich möglich, die Zuweisung einfach durch Überschreiben des Speichers zu implementieren, aber das ist einerseits implementationsabhängig, andererseits auch nicht immer möglich. Abhängig von der Anzahl der Subphyla sind nämlich die Operatoren unterschiedlich groß.

Im erwähnten Problemfall führte die Zuweisung `*rule_t = *tl;` nur zur Zuweisung der in der Basisklasse bereits vorhandenen Attribute, da der Zuweisungsoperator nicht virtuell angewendet wird.³ Die Subphyla wurden damit nicht geändert, da sie erst in den Ableitungen definiert sind. Beide Phyla sind an dieser Stelle von gleicher Ausprägung, und deshalb resultierte die Anweisung im Endeffekt in keiner Veränderung.

3.3.2 Benutzung von `filter()`

In diesem speziellen Fall ging es darum, aus einer Liste die Elemente herauszufiltern, die das Symbol `error` enthalten. Dies ist natürlich eine Aufgabe für die Methode `filter()`, die jede Kimwitu++-Liste besitzt. Eine Auswahlfunktion für diesen Zweck war schon vorhanden, ihr Wahrheitswert musste nur negiert werden. Da es sich um eine Änderung der Struktur des Baumes ging, wurde eine Rewrite-Regel benutzt.

Das führte zu zwei Problemen: Zum einen erlaubt Kimwitu++ kein `->` in Rewrite-Regeln, da der Parser offenbar zu einfach ist. Hier muss Kimwitu++ nachgebessert werden. Zum anderen bringt `filter()` die Liste im Sinne von Rewrite nicht zu einer reduzierten Form, so dass die Regel immer wieder angewendet wird und nicht terminiert – ein Speicherüberlauf ist die Folge.

Also musste doch wieder eine Unparse-Regel benutzt werden, und zwar jeweils am Kopf der zu bearbeitenden Liste. Dies war leicht, da die Liste nur an zwei Stellen überhaupt verwendet wird. Nun wurden auch die Regeln, die `error` enthalten, ausgefiltert. Allerdings fielen in einer Yacc- bzw. Bison-Regel alle Kommentare weg, die *vor* einer der entfernten Regeln in der Liste waren. Eine Untersuchung führte zu dem Ergebnis, dass Kimwitu++ (wie auch schon Kimwitu) bei Benutzung von `filter()` eine neue Liste erzeugt; das Rückgrat wird neu kreiert, jedoch die Attribute des Rückgratphylums nicht mit kopiert. Die Richtigkeit dieses Verhaltens ist von Fall zu Fall unterschiedlich. Hier muss Kimwitu++ nachgebessert werden und optional ein Kopieren der Attribute erlauben.

³Wie sollte das auch funktionieren?

3.3.3 Lösung

Um das Problem in den Griff zu bekommen, gibt es drei Möglichkeiten:

1. Kimwitu++ ist so zu verändern/erweitern, dass ein Ersetzen eines Phylums durch ein Phylum gleichen Typs (aber eventuell anderen Operators) wie in Kimwitu möglich ist. (Bei Beibehaltung der Syntax oder auch durch eine Funktion.)
2. Kimwitu++ ist so zu erweitern, dass `filter()` optional auch die Attribute des Rückgrats einer Liste mit kopiert. Dies kann z. B. durch einen optionalen Parameter geschehen; wird er nicht angegeben, so hat die Funktion die gleiche Semantik wie in Kimwitu.
3. Durch eine andere Wahl der Muster in den Unparse-Regeln kann in der Liste ein Phylum durch ein anderes durch Umsetzen eines Zeigers ersetzt werden. Es existieren keine weiteren Zeiger auf bestimmte Listenelemente.

Lösung 1 ist die allgemeingültigste. Sie stellt sicher, dass *jedes* Kimwitu-Programm konvertierbar ist. Lösung 2 bringt für diesen speziellen Fall die sauberste und eleganteste Lösung. Beide Kimwitu++-Erweiterungen sollten auf alle Fälle vorgenommen werden.

Änderungen an Kimwitu++ übersteigen den Rahmen dieser Studienarbeit. Deswegen wurde Lösung 3 gewählt. Es gibt drei Stellen, an denen Listenköpfe auftreten, und diese erhalten bis auf Typkonvertierungen identische Regeln.⁴

⁴Mit Unparse-Regeln in Kimwitu++ verhält es sich wie mit Templates in C++: Mehrfaches Instanzieren bedeutet Codeduplizierung, auch wenn es nur einmal hingeschrieben wird. Es dreimal hinzuschreiben stellt also nur notationellen und Pflegeaufwand dar, ist aber für die Effizienz irrelevant.

Anhang

A Tabellen

A.1 Quellengröße

Die Tabelle [A.1](#) enthält eine Aufstellung der Größe aller im ursprünglichen gconv enthaltenen Dateien im Format von wc. Es fehlen die Dateien, welche keine Änderung erfahren haben.

Zeilen	Wörter	Zeichen	Datei
284	766	6091	com_gen.k
133	314	3080	debug.k
188	458	4286	ebnf2yacc.k
146	430	3138	ebnfComp.cc
316	837	8233	ebnf_conv.k
104	226	1876	gen_type_decl.k
841	2275	18266	g_pretty.k
505	1328	10209	main.cc
31	84	632	main.h
206	662	4904	Makefile
499	1372	10075	opt_val.cc
116	450	3198	opt_val.hh
1646	5158	41575	regexpr.c
108	262	2026	y_abstr.k
159	491	3470	yComp.cc
663	1789	16847	yScan.l
239	574	4848	y_sem.k
7961	23104	184927	insgesamt

Tabelle A.1: Ausgangslage

Tabelle [A.2](#) schließlich zeigt die Größen nach der Konvertierung. Die Unterschiede sind minimal. Interessant ist,

dass Programme in Kimwitu++ kürzer sind, oder es eigentlich sein können. Bei gconv wird das jedoch verhindert durch die Datei „y_sem.k“, die deutlich länger geworden ist. Das liegt an den in Abschnitt 3.3 aufgeführten Problemen. Lösung 3, mit dem dreifach aufgeschriebenen Muster, ist nicht sehr elegant und beansprucht Platz. Die Lösungen 1 und 2 hätten zu einem kleineren Programmtext geführt.

Zeilen	Wörter	Zeichen	Datei
284	766	6132	com_gen.k
133	314	3076	debug.k
183	452	4354	ebnf2yacc.k
146	430	3138	ebnfComp.cc
316	837	8220	ebnf_conv.k
104	226	1861	gen_type_decl.k
841	2276	18246	g_pretty.k
505	1313	9996	main.cc
31	84	623	main.h
206	662	4920	Makefile
499	1366	10015	opt_val.cc
116	450	3174	opt_val.hh
1647	5160	41592	regexpr.c
108	248	2012	y_abstr.k
159	491	3470	yComp.cc
663	1789	16811	yScan.l
267	613	5651	y_sem.k
7985	23105	185464	insgesamt

Tabelle A.2: Dateien nach Konvertierung

A.2 Ausführbare Dateien

Die Größe und Ausführungsgeschwindigkeit der ausführbaren Dateien ist natürlich vom gewählten Betriebssystem, der Rechnerarchitektur und vom Compiler abhängig. Ein exemplarischer Vergleich ist jedoch auf andere Umgebungen zumindest von der Größenordnung her übertragbar.

Tabelle A.3 zeigt einige Daten, wie sie auf einem Debian-GNU/Linux-System mit dem GNU-C-Compiler entstanden sind. Dabei bezieht sich der Name „gconv++“ auf die gconv-Version, die mit Kimwitu++ geschrieben wurde. Die Abarbeitungsgeschwindigkeit wurde ermittelt anhand der Datei „parse.y“, einer Yacc-Datei für C++ aus der Distribution des GCC [GCC], sowie mit „synana.y“ aus dem Parser von [Sch97]. Dazu wurden mehrere Aufrufe von gconv mittels time gemessen.

	gconv	gconv++
Dateigröße* (Bytes)	245544	310888
Zeit (s) „parse.y“	1,0	1,2
„synana.y“	2,3	2,8

* Keine Debugger-Symbole, dynamisch gelinkt, stripped.

Tabelle A.3: Daten der Ausführbaren

B From Kimwitu to Kimwitu++

Kimwitu++ funktioniert fast genauso wie sein Vorgänger Kimwitu. Da eine neue Dokumentation noch nicht geschrieben ist, kann nur eine Übersicht über die Unterschiede gegeben werden. Dies ist eine gekürzte Version von [Pie99], welches Teil der internen Kimwitu++-Quellen ist.

- Generally Kimwitu++ works together with C++ , while Kimwitu worked with C. This will possibly cause some of the usual C to C++ hassle.
- Kernighan & Ritchie style for C functions is not allowed in *.k files. Use proper ANSI C instead.
- There is no longer a predefined phylum called `int`. There is, however, a phylum providing the same functionality called `integer`. When converting, please note that `integer i; i=5;` will be caught by the compiler, while `i=0;` will not.

The right way to use them is `integer i; i=mkint(5);` and `if (i->value==5)` There is also a shortcut, a conversion to `int`, but since `integer` is like a pointer, you have to write `if (*i==5)`

- Similar things apply to `float` et. al., there is now a phylum `real`.
- Phyla are classes or objects, respectively. Instead of

```
unparse_completeSyntaxTree(synTree, printer, view );
```

you can use the more natural

```
synTree->unparse( printer, view );
```

- The keyword and typename `%view` and `view` are no longer available (they were deprecated anyway). Use `%uview` and `uview`.

Literaturverzeichnis

- Bro87 BROOKS, FRED: *No Silver Bullet*. IEEE Computer, 20(4), April 1987. 7
- C90 ISO/IEC 9899:1990 *Programming languages – C*. International Standard, 1990. 6
- Coë93 COËTMEUR, ALAIN: *flex++ — generate a scanner in C or C++ (flex++.1)*. R&D department (RDT), Informatique-CDC, France, März 1993. flex++ 2.3.8-7. 10
- C++98 ISO/IEC 14882:1998 *Programming languages – C++*. International Standard, 1998. 6
- CS95 CORBETT, ROBERT und RICHARD M. STALLMAN: *Bison parser generator manual (bison.info)*. Free Software Foundation, Inc., Cambridge, Massachusetts, 1995. Bison 1.25. 11
- Duden96 WISSENSCHAFTLICHER RAT DER DUDENREDAKTION (Herausgeber): *Duden – Die deutsche Rechtschreibung*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, 21. Auflage, 1996.
- Dum96 DUMKE, CARSTEN: *Lexikalische Analyse von SDL '92 mit ASN.1 und theoretische Grundlagen der Fehlerbehandlung*. Studienarbeit, Humboldt-Universität zu Berlin, Dezember 1996. 4
- Dum99 DUMKE, CARSTEN: *gconv, yacc2yacc, yacc2html, ebnf2yacc — grammar transformations (gconv.1)*. Berlin, März 1999. gconv 2.0. 4, 11
- GCC THE GCC TEAM: *GCC Project Home Page*, 1999. <http://www.gnu.org/software/gcc>. 18
- Pax95 PAXSON, VERN: *flex — fast lexical analyzer generator (flex.info)*, 1995. flex 2.5. 10
- Pie99 PIEFEL, MICHAEL: *From Kimwitu to Kimwitu++*. Preliminary Kimwitu++ documentation, Juni 1999. 6, 19
- Sch97 SCHRÖDER, RALF: *SDL Integrated Tool Environment (SITE)*, 1997. <http://www.informatik.hu-berlin.de/Institut/struktur/systemanalyse/SITE/SDL-tools.html>. 18
- vEB96 EIJK, PETER VAN und AXEL BELINFANTE: *The Term Processor Kimwitu. Manual and Cookbook*. Universiteit Twente, Enschede, Nederlands, 9. Auflage, Juli 1996. 4, 12