Humboldt Universität zu Berlin
Institut für Informatik
Lehrstuhl für Systemanalyse

Studienarbeit

Automatische Konvertierung von Kimwitu nach Kimwitu++

Markus Scheidgen
Betreuer: Dipl. Inf. Michael Piefel

Berlin, 24. April 2003

Kimwitu ist ein auf C basierender Termprozessorgenerator, der am Lehrstuhl zur Entwicklung von SDL-Compilerwerkzeugen, den SITE-Tools, eingesetzt wird.

Kimwitu++ ist ein auf C++ basierendes Kimwitu. Um C++ und damit objektorientierte Konzepte bei der Weiterentwicklung der SITE-Tools verwenden zu können, möchte man mit möglichst geringem Aufwand die bereits existierenden Kimwitu-Programme in Kimwitu++-Programme konvertieren.

Diese Studienarbeit beschäftig sich nun damit, ein Werkzeug zu entwickeln, welches eine solche Konvertierung so weit wie möglich automatisiert.

Inhaltsverzeichnis

1. Einleitung **7**
- 1.1. Die Ausgangslage . . . 7
- 1.2. Verwendete Begriffe . . . 7
- 1.3. Die Aufgabenstellung . . . 8
- 1.4. Vorhandene Vorarbeit . . . 8
- 1.5. Vorgehensweise . . . 9

2. Die Konvertierungsprobleme **10**
- 2.1. Vorüberlegungen . . . 10
- 2.2. Probleme in der Kimwitu-Syntax . . . 11
  - 2.2.1. Funktionsdefinitionen . . . 11
  - 2.2.2. Views . . . 12
  - 2.2.3. Die Phyla int und float . . . 12
- 2.3. Probleme durch Änderungen innerhalb der C/C++-Realisierung . . . 13
  - 2.3.1. Methoden- statt Funktionsrufe . . . 13
  - 2.3.2. Geänderte Bezeichner . . . 14
  - 2.3.3. Zuweisungen von int- und float-Phyla . . . 15
  - 2.3.4. Phylumzuweisungen . . . 15
  - 2.3.5. Const . . . 16
- 2.4. Weitere Konversionen . . . 17

3. Ein Werkzeug zur Konvertierung **20**
- 3.1. Anforderungen . . . 20
- 3.2. Design und Funktionsweise . . . 21
  - 3.2.1. Die Konversionen werden an der Kimwitu-Grammatik gesteuert 21
  - 3.2.2. Allgemeine Umsetzung, Einbettung und Management der Konversionen . . . 22
  - 3.2.3. Dokumentierung und Protokollierung . . . 24
  - 3.2.4. Benutzer Interface zur Kontrollierung der Konvertierung . . . 25
- 3.3. Verwendung bei der Konvertierung . . . 25
  - 3.3.1. Empfohlene Vorgehensweise . . . 25
  - 3.3.2. Scripte . . . 25
  - 3.3.3. Implementation von weiteren Konversionen . . . 27

A. Konversionen **30**

B. Statistiken der Konvertierung der SITE-Tools **33**

C. Die Manpage des Konverters kc2kc++ **34**

1. Einleitung

1.1. Die Ausgangslage

Kimwitu ist ein Werkzeug zur Generierung von Termprozessoren. Das heißt, es ist eigentlich ein Compiler für die gleichnamige Sprache Kimwitu, welche es mit der Hilfe von C-Erweiterungen erlaubt, Programme zu schreiben, die typisierte Bäume, also Terme, als Datenstrukturen verwenden.

Kimwitu wird seit Jahren am Lehrstuhl für Systemanalyse eingesetzt. Unter dem Einsatz dieses Werkzeuges sind die SDL[1]-Compiler-Werkzeuge SITE-Tools[2] entstanden.

Aus dem Wunsch heraus, objektorientierte Konzepte in Kimwitu-Programmen verwenden zu können, ist Kimwitu++ entstanden. Kimwitu++ ist ein Kimwitu, welches auf C++ basiert. Neben einer leicht abgewandelten Syntax besteht der größte Unterschied zu Kimwitu darin, dass in Kimwitu++ interne Strukturen objektorientiert in C++ realisiert werden. Und natürlich kann in einem Kimwitu++-Programm nun C++-Code anstelle von C-Code enthalten sein.

Im Laufe der Zeit sind eine Reihe von Kimwitu-Programmen entstanden, unter anderem auch die SITE-Tools. Verständlicherweise besteht nun der Wunsch, auch in diesen Programmen objektorientierte Konzepte verwenden zu können. Dazu sucht man eine geeignete Möglichkeit, bestehende Quellen von Kimwitu-Programmen mit geringem Aufwand in Kimwitu++-Code zu konvertieren.

Man stellt sich diese Konvertierung folgendermaßen vor: Aufgrund der Ähnlichkeit von Kimwitu und Kimwitu++, sind nur von einander Unabhängige, meist syntaktische Änderungen zu machen. Einfache Änderungen könne durch ein Werkzeug umgesetzt werden, schwierige müssen per Hand gemacht werden.

1.2. Verwendete Begriffe

Wenn von einem Kimwitu-Programm geredet wird, ist damit Programm als auszuführende, problemlösende Einheit gemeint. Wird dagegen über die Realisierung eines solchen Programms in Form von Kimwitu-Code geredet, wird dieser als Quellen eines Kimwitu-Programms bezeichnet. Wird über Kimwitu-Code gesprochen, ist damit allgemeiner Quelltext der in Kimwitu geschrieben wurde, gemeint, ohne dabei auf ein spezifisches Programm Bezug zu nehmen. Mit Kimwitu-Quellen wird der Quellcode des Werkzeugs Kimwitu bezeichnet. Diese Begriffe gelten gleichermaßen für Kimwitu++.

[1] Specification and Description Language [Z100]
[2] SITE, SDL Integrated Tool Environment

Mit Konvertierung wird allgemein die totale oder teilweise Umwandlung von Kimwitu-Code nach Kimwitu++-Code beschrieben. Bei der Konvertierung gemachte Änderungen können immer einem Konvertierungsproblem zugeordnet werden. Die Umwandlung von Funktionsrufen in Methodenrufe wäre zum Beispiel ein solches Konvertierungsproblem. Ein einzelner Funktionsruf würde dann als Instanz des Konvertierungsproblems bezeichnet werden. Ein Konvertierungsproblem wird durch eine Konversion beschrieben. Mit der Ausführung einer Konversion ist das Vornehmen der nötigen Änderungen an allen Instanzen des durch die Konversion beschriebenen Problems gemeint. Mit Änderung ist die Lösung eines Konvertierungsproblems, bzw. Ausführung einer Konversion an einer einzelnen Instanz des Problems gemeint. Ein Programm oder ein Teil eines Programms, welches eine Konversion ausführt, wird als Realisierung dieser Konversion bezeichnet.

Der Begriff Phylum, bzw. seine Pluralform Phyla, soll immer als Phylumtyp verstanden werden. Eine Phyluminstanz bezeichnet dann die Instanz eines Phylumtyps. Kimwitu erzeugt aus Kimwitu-Code C-Code. Dieser C-Code definiert programmiersprachliche Einheiten, kurz C-Konstrukte, welche die Kimwitu-Konzepte realisieren. Dies sind zum Beispiel die Definition von Strukturen zur Realisierung der Phyla oder die Definition von vordefinierten Funktionen, die in Kimwitu verwendet werden können. Diese C-Konstrukte werden als C-Realisierung der Kimwitu-Konzepte bezeichnet. Für Kimwitu++ gilt dasselbe, hier natürlich mit C++.

1.3. Die Aufgabenstellung

Die Aufgabe besteht nun darin, ein Werkzeug zu entwickeln, welches die Konvertierung vereinfacht, indem es einfache, aber immer wiederkehrende Teile der Konvertierungsarbeit automatisch ausführt.

1.4. Vorhandene Vorarbeit

Im Rahmen der Bestrebungen des Lehrstuhls, Kimwitu-Programme zu konvertieren, ist bereits eine Arbeit zu diesem Thema entstanden.

Die Studienarbeit *Exemplarische Konvertierung eines SITE-Tools von Kimwitu nach Kimwitu++* [Pie99] von Michael Piefel beschreibt die Konvertierung eines kleineren, aber beispielhaften Kimwitu-Programms nach Kimwitu++. In dieser Arbeit werden einzelne Konversionen identifiziert und es wird beschrieben, wie diese Konversionen auszuführen sind.

Diese Arbeit bildet die Grundlage für meine Arbeit. Es ist allerdings davon auszugehen, dass weitere Konversionen existieren, die nicht in dieser Arbeit beschrieben wurden, da hier nur ein einzelnes Programm beispielhaft übersetzt wurde.

1.5. Vorgehensweise

Zunächst ist es wichtig, die nötigen Konversionen zu identifizieren. Einige sind schon bekannt, andere sind noch zu finden. Danach muss für jede Konversion diskutiert werden, ob diese von einem Werkzeug realisiert werden kann oder ob sie per Hand auszuführen ist. Das zweite Kapitel dieser Arbeit, *Die Konvertierungsprobleme*, wird sich damit auseinandersetzen.

Dann müssen die gefunden Konversionen umgesetzt werden, das heißt es muss ein Werkzeug entstehen, welches diese Konversionen ausführen kann. Das dritte Kapitel, *Ein Werkzeug zur Konvertierung*, wird Entstehung, Design und Funktionsweise des entstandenen Werkzeugs beschreiben.

2. Die Konvertierungsprobleme

2.1. Vorüberlegungen

Kimwitu verwendet dieselben Konzepte wie Kimwitu++. Die Unterschiede zwischen Kimwitu und Kimwitu++ liegen nur in der Syntax beider Sprachen und in der jeweiligen C- bzw. C++-Realisierung. Aus diesem Grund sind die zu machenden Änderungen kleine von einander unabhängige Konversionen. Dies wird die Untersuchung der einzelnen Konversionen noch zeigen.

Viele dieser Konversionen sind bereits bekannt. Siehe [Pie99]. Die in [Pie99] beschriebenen Konvertierungsprobleme decken den dokumentierten Teil von Kimwitu ab. Die Konversionen, die durch die nicht vollständig dokumentierte C- bzw. C++-Realisierung von Kimwitu- und Kimwitu++-Konzepten entstehen, können entdeckt werden, in dem man mit den vorhandenen Konversionen ein großes Kimwitu-Programm konvertiert. Bei der Überprüfung des hierbei entstehenden Kimwitu++-Programms, werden dann versäumte, aber nötige Änderungen sichtbar. In Abschnitt 3.3 wird diese Vorgehensweise genauer beschrieben. Für die Analyse möglicher Konversionen wurde die Semantikanalyse der SITE-Tools verwendet. Auf Grund der Größe dieses Programms und seinem umfassenden Ausnutzen der erzeugten C-Realisierung ist damit zu rechnen, dass innerhalb der Quellen dieses Programms die meisten Konvertierungsprobleme auftauchen.

In diesem Kapitel geht es nun darum, alle gefundenen oder bereits vorhandenen Konversionen aufzuzählen und bezüglich ihrer Umsetzung mit einem Computerprogramm zu bewerten.

Um dies zu dokumentieren, wird folgendes Muster verwendet.

Problem In diesem Punkt wird erläutert, welcher Unterschied von Kimwitu und Kimwitu++ dieses Problem erzeugt.

Identifizierung Hier wird geklärt, wie sich eine Probleminstanz eines solches Konvertierungsproblem in Kimwitu-Code identifizieren lässt. Ob es sich durch ein Programm identifizieren lässt und wie dieses realisiert werden kann oder ob der Aufwand, ein solches Programm zu entwickeln, zu hoch ist.

Umsetzung Nun wird beschrieben, wie sich die identifizierten Codestellen in Kimwitu++-Code konvertieren lassen und ob dies durch ein Programm realisiert werden kann.

Lösung Hier wird geklärt, wie sich die Konversion in einem Werkzeug konkret realisieren lässt, welche Technologie dabei zum Einsatz kommt. Wenn die Konversion nicht durch ein Programm realisiert werden kann, lässt sich die manuelle Ausführung der Konversion wenigstens durch ein Programm unterstützen?

Die Auflistung der Konversionen nach diesem Muster kann dann direkt zur Konvertierung herangezogen werden. Sie bildet die Grundlage zur Entwicklung und Dokumentierung des Werkzeugs, sowie zur Ausführung der verbleibenden Handarbeit.

Die Konversionen werden durch verschiedene Konverter realisiert. Welche Konversionen in welchem Konverter realisiert wurden, kann in Anhang A nachgeschlagen werden. Die Gesamtheit dieser Konverter bildet dann das entstandene Konvertierungswerkzeug. Siehe Kapitel 3.

2.2. Probleme in der Kimwitu-Syntax

2.2.1. Funktionsdefinitionen

Problem In Kimwitu gibt es zwei syntaktische Möglichkeiten, eine Funktion zu definieren, zum einen nach Kernighan/Ritchie und zum anderen nach ISO C. Siehe [C90] und [C++98]. In Kimwitu++ ist nur noch die Definition nach ISO C erlaubt. Es müssen also alle Kernighan/Ritchie-Funktionsdefinitionen in ISO C-Definitionen umgewandelt werden.

Identifizierung Die Funktionsdefinitionen können leicht durch Parsen des Kimwitu-Codes mit Hilfe der Kimwitu-Grammatik gefunden werden, da diese Funktionen in der Grammatik als eigene Nichtterminale vorkommen.

Umsetzung Zum Ausschreiben der gefundenen Funktionsdefinition als ISO C Funktionsdefinitionen müssen die einzelnen Teile der Definition(Rückgabetyp, Name, Parametertypen und Parameternamen) identifiziert werden. Dies kann wiederum über die Kimwitu-Grammatik geschehen, da diese Entitäten auch hier als Symbole in der Grammatik enthalten sind. Danach müssen die einzelnen Teile nur noch neu angeordnet und als neue Funktionsdefinition ausgeschrieben werden.

```
<function-def>:= <qualifier> <type-expression> <name>
      '(' <argument-names> ')' <argument-type-list>
...
<argument-type>:= <qualifier> <type-expression>
     <argument-names>
```

Abbildung 2.1.: Kernighan/Ritchie Funktionsdefinitionen

```
<function-def>:= <qualifier> <type-expression> <name>
      '(' <argument-list > ')'
...
<argument>:= <qualifier> <type-expression>
     <argument-name>
```

Abbildung 2.2.: ISO C Funktionsdefinitionen

Lösung Eine Kernighan/Ritchie-Funktionsdefinition genügt der Syntax in Abbildung 2.1, eine ISO C-Definition der in Abbildung 2.2. Die beiden Grammatik konnten auf dieselben Symbole herunter gebrochen werden. Es müssen also nur die Werte der Symbole beim Parsen gespeichert werden, um sie dann als ISO C-Funktionsdefinition neu anzuordnen.

Als Technologie wird hier ein Parsergenerator verwendet. Hierfür wird vorhandener yacc-Code aus den Kimwitu-Quellen verwendet werden. Beim Parsen müssen die einzelnen Teile der Definition gespeichert werden und beim Schreiben ausgegeben werden. Für die Beschreibung des verwendeten Parsermechanismusses siehe Abschnitt 3.2.1.

2.2.2. Views

Problem In Kimwitu gibt es zwei Schlüsselwörter, um *Views* für unparse-Regeln zu definieren. Dies sind **uview** und **view**. In Kimwitu++ ist nur noch **uview** erlaubt.

Identifizierung Das Schlüsselwort **view** ist ein Terminalsymbol in der Kimwitu-Grammatik. Es Kann also beim Parsen mit dieser Grammatik leicht identifiziert werden.

Umsetzung Ein gefundenes **view** muss durch **uview** ersetzt werden.

Lösung Hierfür wird der in Abschnitt 3.2.1 beschriebene Parsermechanismus verwendet.

2.2.3. Die Phyla int und float

Problem In Kimwitu stehen die atomaren Phyla *int* und *float* zur Verfügung. Sie sind durch die gleichnamigen primitiven C-Typen realisiert. In Kimwitu++ heißen diese Phyla jetzt *integer* und *real* und sie sind durch Klassen realisiert.

Hier hat man es also mit zwei verschiedenen Problemen zu tun, einmal die Verwendung der Phylumbezeichner **int** und **float**, zum anderen die Problematik, die sich durch die unterschiedliche Realisierung ergibt. Der zweite Teil dieses Problems wird in Abschnitt 2.3.3 als eigene Konversion behandelt.

Identifizierung Mit Hilfe der Grammatik lässt sich die Verwendung von definierten Phyla innerhalb von Phylumdefinitionen an Hand ihrer Namen finden, da für diese Phylanamen ein Terminalsymbol verwendet wird.

Umsetzung Gefundene Verwendungen von *int*- und *float*-Phyla in einer Phylumdefinition müssen einfach durch die entsprechenden Namen der Kimwitu++-Phyla ersetzt werden.

Lösung Mit dem in Abschnitt 3.2.1 besprochenen Parsermechanismus lassen sich die innerhalb einer Phylumdefinition verwendeten Phyla, über ihren Namen identifizieren. Die gefundenen Phylanamen werden mit den Strings **int** und **real** verglichen und durch die entsprechenden geänderten Bezeichner ersetzt.

2.3. Probleme durch Änderungen innerhalb der C/C++-Realisierung

2.3.1. Methoden- statt Funktionsrufe

Problem Kimwitu stellt eine Reihe von vordefinierten Funktionen auf Phylainstanzen zur Verfügung. In Kimwitu++ werden die Phyla durch Klassen realisiert, und diese Funktionen sind nun Methoden der Phyla-Klassen.

Identifizierung Die Funktionsnamen in Kimwitu sind Zusammensetzungen aus dem eigentlichen Namen und dem Namen des Phylumtyps, auf deren Instanzen die Funktion angewendet werden kann. Um Verwechselungen mit unabhängig definierten Funktionen zu vermeiden, sollte hier bei der Konvertierung nach dem kompletten Namen gesucht werden.

Hierzu müssen die Namen aller definierten Phyla gesammelt werden. Dann kann man diese mit den Funktionsnamen zusammen setzen und aus den erhaltenen Zeichenketten reguläre Ausdrücke bilden. Anhand dieser Ausdrücke können dann innerhalb von C-Fragmenten die Funktionsrufe identifiziert werden.

Umsetzung Wurde ein Funktionsruf gefunden, so muss nun seine Argumentenliste untersucht werden und die gefundenen Argumentenausdrücke müssen gespeichert werden. Mit diesen Informationen kann der entsprechende Methodenruf gebildet werden und es kann der Funktionsruf damit ersetzt werden.

Lösung Um die Phylanamen zu erhalten, wird der in Abschnitt 3.2.1 beschriebene Parsermechanismus verwendet, da die Phylanamen als Terminalsymbol innerhalb von Phylumdefinitionen vorkommen. Es muss beachtet werden, dass Phyladefinitionen an beliebigen Stellen in Kimwitu-Programmen vorkommen können. Es muss also zuerst der gesamte Kimwitu-Code nach Phylumdefinitionen geparst werden, bevor er nach

Funktionsrufen durchsucht werden kann. Dies ist durch zwei getrennte Konverter realisiert, welche Daten, in diesem Fall die Phylanamen, mit einander austauschen können. Siehe hierzu Abschnitt 3.2.2: *SharedDataConverter*. Einer der Konverter durchsucht den Code nach Phylumdefinitionen und der zweite verwendet die Ergebnisse des ersten zum Suchen und Ersetzten von Funktionsrufen.

Die C-Fragmente lassen sich mit dem Parsermechanismus isolieren, siehe Abschnitt 3.2.1. Mit Hilfe einer Programmbibliothek für reguläre Ausdrücke wird dann innerhalb der C-Fragmente nach Funktionsrufen gesucht.

Zum Parsen der Argumente kommt ein handgeschriebener Parser zum Einsatz, da hier nur Klammern, Kommata, Kommentarbegrenzer sowie Begrenzer von Stringliteralen gezählt werden müssen. Einer der Argumentenausdrücke ist des Objekt, an dem die Funktion bzw. Methode gerufen wird. Welches der Argumente das Objekt beschreibt, ist von Funktion zu Funktion verschieden. Um das zu entscheiden, wird eine Abbildung verwendet, welche einem Funktionsnamen die entsprechende Argumentennummer zuweist. Siehe Anhang A.

Der Methodenruf wird dann aus dem Argument, welches die Phyluminstanz beschreibt, den aus dem Funktionsnamen gebildeten Methodennamen und den restlichen Argumenten konstruiert.

2.3.2. Geänderte Bezeichner

Problem Viele Bezeichner für Typen, Strukturen, Variablen, usw. innerhalb der von Kimwitu erzeugten C-Realisierung wurden in Kimwitu++ anders benannt.

Identifizierung Die entsprechenden Bezeichner können über reguläre Ausdrücke gefunden werden.

Umsetzung Die gefundenen Kimwitu-Bezeichner müssen durch ihre Kimwitu++-Entsprechungen ersetzt werden.

Lösung Die Kimwitu-Bezeichner und ihre Kimwitu++-Entsprechungen werden in einer statischen Tabelle geliefert. Anhand dieser Tabelle werden reguläre Ausdrücke gebildet. Mit diesen Ausdrücken wird ein Suchen und Ersetzen durchgeführt.

Bei der Bildung von regulären Ausdrücken muss der Kontext, in denen diese stehen können, beachtet werden. So ist es möglich Bezeichner als ganzes Wort zu suchen, Bezeichner als Prefix eines beliebigen Bezeichners zu suchen oder als Prefix eines Phylumnamens zu suchen. Für die letztere Variante wird das Verfahren aus Abschnitt 2.3.1 verwendet. Der zu einem Ausdruck gehörige Kontext steht als Teil der Bezeichnerabbildung zur Verfügung.

Die genaue Bezeichnerabbildung kann im Anhand nachgeschlagen werde. Siehe Anhang A. Siehe auch Abschnitt 3.3.3, um die Liste der regulären Ausdrücke zu erweitern.

2.3.3. Zuweisungen von int- und float-Phyla

Problem In Kimwitu sind die atomaren Phyla *int* und *float* durch die gleichnamigen primitiven C-Typen realisiert. In Kimwitu++ sind diese durch Zeiger auf Instanzen der generierten Klassen **integer** und **real** realisiert.

Die Zuweisungen von *int*- bzw. *float*-Werten an die entsprechenden Phyluminstanzen innerhalb von C-Fragmenten funktioniert in Kimwitu++ nicht mehr, da hier Werte der Typen *int* und *float* als Werte vom Typ Zeiger auf **integer** und **real** interpretiert werden.

Identifizierung Beim Compilieren eines Programms mit solchen falschen Zuweisungen als Kimwitu++-Programm, würde in dem meisten Fällen ein Compilerfehler generiert werden. Dieser kann zur Identifizierung verwendet werden. Aber wird einem Zeiger das *int* - oder *float* -Literal **0** zugewiesen, kann der Compiler dieses nicht vom Literal des Null-Zeigers unterscheiden.

Für eine sichere Identifizierung, müsste man also die einzelnen Zuweisungen innerhalb der C-Fragmente finden. Da diese Zuweisungen innerhalb beliebiger C-Konstruktionen vorkommen können und da die Phyla, denen zugewiesen wird, in Variablen stehen, deren Typ zuvor ermittelt werden müsste, wäre es hier nötig die Funktionsweise eines C-Parsers inklusive statischer Semantikanalyse zu großen Teilen nachzubilden.

Umsetzung In einer gefundenen Zuweisung müsste der zugewiesene Ausdruck durch einen Aufruf von **mkinteger** bzw. **mkreal**, mit dem zugewiesenen Ausdruck als Argument, ersetzt werden.

Lösung Die Nachbildung eines C-Compilers ist zu aufwendig. Diese Konversion muss manuell ausgeführt werden. [1]

2.3.4. Phylumzuweisungen

Problem Jede Phyluminstanz wird in Kimwitu sowie in Kimwitu++ durch eine Instanz eines entsprechenden generierten Struktur-Typs(**struct** bzw. **class**) realisiert. Diese Phyluminstanzen werden durch Zeiger referenziert. Die Instanzen können über Zeigedereferenzierung einander zugewiesen werden.

Jede Phyluminstanz wird durch einen Operator erzeugt. Ein Phylum kann verschiedene Operatoren definieren. Das heißt zwei Instanzen eines Phylums können durch verschiedene Operatoren erzeugt worden sein.

Die Zugehörigkeit einer Phyluminstanz zu einem der verschiedenen Operatoren ist in Kimwitu durch eine Union innerhalb der zum Phylum generierten C-Struktur realisiert. Bei einer Zuweisung wird diese Union somit kopiert. Wenn nun zwei Phyla unterschiedlicher Operatoren einander zugewiesen werden, wird die entsprechende Operatorzugehörigkeit mit zugewiesen.

In Kimwitu++ werden diese Operatoren durch Spezialisierungen der zum Phylum generierten Klasse realisiert. Die Phyluminstanzen werden durch einen Zeiger vom

[1] glücklicherweise spielt die Verwendung solcher Zuweisungen in den SITE-Tools keine große Rolle.

Typ *Zeiger auf Basistyp* referenziert. Bei der Zuweisung zweier Instanzen von verschiedenen Operatorspezialisierungen wird diese also auf Grundlage des Basistyps durchgeführt. Bei dieser Zuweisung wird die Spezialisierung und damit der Operator ignoriert.

Man hat also einen semantischen Unterschied in diesen Zuweisungen. Lässt man diese Stellen bei der Konvertierung unberührt so entsteht korrekter Kimwitu++-Code. Aber das entsprechende Kimwitu++-Programm tut etwas anderes.

Identifizierung Solche Zuweisungen können innerhalb beliebig komplizierter C-Konstruktionen vorkommen. Zur sicheren Identifizierung müsste die Funktionsweise großer Teile eines C-Compilers nachempfunden werden. Das ist zu aufwendig.

Bis jetzt sind nur Zuweisungen bekannt, welche ein relativ auffälliges Derefenrenzierungs- und Zuweisungs-Pattern aufweisen(z.B. ***phylum1=*phylum2**). Dieses könnte anhand eines regulären Ausdrucks gefunden werden. Dieses stellt allerdings keine sichere Identifizierung dar.

Eine weitere Möglichkeit wäre es, die Kimwitu++-Quellen, bzw. den von Kimwitu++ generierten Code, so zu modifizieren, dass in jeder Phylumklasse der Zugriff auf den Zuweisungsoperator beschränkt ist und so bei der Übersetzung des konvertierten Codes an jeder Zuweisung ein Compilerfehler generiert wird. Diese Modifizierung lässt sich an der Phylumbasisklasse vornehmen.

Umsetzung Wie der Kimwitu-Code zu ändern ist, ist von Fall zu Fall verschieden. Es ist unmöglich Instanzen zweier verschiedener Spezialisierungen einander zuzuweisen. Um die Semantik einer Zuweisung zu realisieren, müsste die Phyluminstanz, der zugewiesen werden soll, zerstört werden, um sie dann durch eine Kopie der anderen Phyluminstanz zu ersetzen.

Nun werden in Kimwitu diese Phyluminstanzen durch Zeiger referenziert. Um die Semantik aufrecht zu erhalten, müsste die Kopie also an der Adresse der alten Phyluminstanz erzeugt werden. Dies ist auf Grund unterschiedlicher Objektgrößen nicht immer möglich. Also muss die verwendete Adresse geändert werden. Nun ist es aber möglich, dass diese Adressen an den unterschiedlichsten Stellen gehalten wird.

Es ist also nicht möglich, die Umsetzung automatisch zu realisieren. Selbst der menschlicher Konvertierer stellt sich hier einer sehr fordernden Aufgabe.

Lösung Es wurde gezeigt, dass die Umsetzung nicht durch ein Computerprogramm realisiert werden kann. Diese Konversion bleibt also Handarbeit.

2.3.5. Const

Problem Die Sprachen C/C++ beherrschen das Konzept konstanter Variablen. Daher ist es selbstverständlich, dass dieses Konzept, welches hilft, die Semantik von Definitionen stärker und sicherer zu beschreiben, auch in Kimwitu++ Verwendung findet. Aber in Kimwitu ist dies nicht der Fall.

So sind in Kimwitu++durch Pattern erzeugte Variablen konstant und die Rückgabewerte und Parameter einiger vordefinierter Funktionen und Funktionstypdefinitionen sind konstant. In Kimwitu ist das nicht so.

Identifizierung Die Verwendung von konstanten Variablen hat nur auf die Veränderung von Werten dieser Variablen Einfluss. Also stellen diese Variablen nur dann ein Problem dar, wenn eine Wertänderung an ihnen vorgenommen wird. Aber genau diese Änderungen an Variablenwerten lösen in Kimwitu++ einen Compilerfehler aus. Diese Compilerfehler stellen also eine Identifizierung genau der Codestellen dar, die geändert werden müssen.

Eine Identifizierung direkt durch einen Konverter ist nicht ohne größeren Aufwand möglich, da hierfür große Teile der Funktionalität eines C/C++-Parsers und einer C/C++-Semantikanalyse nachempfunden werden müssten.

Umsetzung Eine Identifizierung durch ein Programm ist also nicht ohne weiteres Möglich. Dies ist aber unerheblich, da eine passende Umsetzung der identifizierten Codestellen ohnehin nicht möglich ist.

Eine saubere Umsetzung wäre es, die entsprechenden Codestellen so umzuschreiben, so dass diese ohne eine Wertänderung an einer konstanten Variable auskommen. Ein solches Vorgehen setzt aber ein genaues Verständnis der Implementierung voraus. Ein solches Wissen kann ein Programm weder haben noch erwerben.

Lösung Wie erläutert ist eine automatische Konvertierung nicht möglich. Durch die Compilerfehler ist aber eine sichere Identifizierung ermöglicht. Die identifizierten Codestellen müssen dann nur noch manuell angepasst werden.

2.4. Weitere Konversionen

Es wurde noch eine Reihe weiterer Konversionen gefunden. Auf Grund ihres geringen Vorkommens in typischem Kimwitu-Code, wie den SITE-Quellen, seien diese hier nur kurz besprochen.

Hashstatistiken In Kimwitu++ gibt es keine Möglichkeit mehr, Hashstatistiken zu erzeugen. Will man weiter hin Hashstatistiken erzeugen, muss entweder Kimwitu++ erweitert werden oder diese Funktionalität muss selbst implementiert werden. Anderen Falls müssen die Rufe an die Kimwitu Hashstatistikfunktionalität entfernt werden. Hierfür bietet das entstandene Werkzeug keine Funktionalität.

Speicher-Management Kimwitu erzeugt Funktionen für ein erweitertes Speichermanagement. Kimwitu++ tut dies nicht mehr. Werden diese Funktionen in zu konvertierenden Kimwitu-Code verwendet und soll diese Verwendung des Speichermanagements auch beibehalten werden, musst dieses selbst implementiert werden. Andernfalls müssen vorhandene Rufe an das Speichermanagement auf existierende Konstruk-

te, wie zum Beispiel den C++ Operatoren **new** und **delete**, umgeschrieben werden. Der geschriebene Konverter bietet hierfür keine Funktionalität.

Das Attribut prod_sel Das abstrakte Phylumattribut **prod_sel** ist in Kimwitu als Datum des entsprechenden C-Typs realisiert worden und ist in Kimwitu++ als Methode der entsprechenden Klasse implementiert. Diese Konversion wird durch einen regulären Ausdruck realisiert. Siehe Abschnitt 2.3.2.

Assertion Makros Kimwitu und Kimwitu++ erzeugen Makros, welche zum Bilden von Assertions verwendet werden können. Die Definitionen dieser Markos sind aber nicht dieselben. Auf Grund der Ähnlichkeit dieser Konversion zu der Konversion von Funktionsrufen, konnte diese Konversion mit dem selben Konverter realisiert werden, siehe Abschnitt 2.3.1.

Das Attribut is_list In Kimwitu hat jedes Phylum ein Attribut **is_list**, welches angibt, ob es sich um ein Listphylum handelt oder nicht. Dieses Attribut wird von Kimwitu++ nicht erzeugt. Der Kimwitu-Code muss hier per Hand so angepasst werden, so dass der Code ohne Verwendung dieses Attributes auskommt.

GLOBAL_ATTRIBUTE_PATCH In Kimwitu lässt sich über das Makro **GLOBAL_ATTRIBUTE_PATCH** die Erzeugung der Phylumattribute **__comment** und **__stuff** aktivieren. Kimwitu++ bietet diese Möglichkeit nicht. Werden diese Attribute benötigt, können diese leicht manuell dem abstrakten Phylumtyp und damit allen Phyla hinzugefügt werden. Das entstandene Werkzeug enthält hierfür keine Funktionalität.

Die Definitionen kc_voidptr, kc_voidfnptr Kimwitu erzeugt zwei Typdefinitionen **kc_voidptr** und **kc_voidfnptr**. Kimwitu++ tut dies nicht. Hier könnte man diese beiden Typdefinitionen einfach per Hand dem Code hinzufügen. Man sollte sich aber überlegen, ob die Verwendung des Typs **void** in Kimwitu++ noch angebracht ist. Da in Kimwitu++ alle Phyla Ableitungen von **abstract_phyla_impl** sind, sollte man hier Zeiger auf diesen abstrakten Typ verwenden. Das geschriebene Werkzeug enthält hierfür keine Funktionalität.

Viele kleine Konversionen Die von Kimwitu und Kimwitu++ generierten Realisierungen, enthalten viele kleinere Änderungen. Wie zum Beispiel Makros aus den Methoden werden oder Arrays auf Strings aus den Arrays auf Structs mit mehreren Strings werden, usw.. Es ist nun nicht möglich für jede erdenkliche Verwendung dieser Konstrukte eine Konversion zu formulieren und diese zu Realisieren. Es sind aber zwei Ausdrücke aufgefallen, die immer wieder benutzt werden und die diese kritischen, Konstrukte verwenden. Siehe Abbildung 2.3.

Kleine Konversionen, wie diese, sind sehr einfach aber nicht generisch. Ihre Realisierung verlangt also immer eine spezifische Vorgehensweise. Dies kann aber leicht durch kleine in einer Scriptsprache geschriebene Routinen realisiert werden. Diese

```
/* Kimwitu */
kc_view_names[<index>]
KC_SUBPHYLUM(<node>, op_info->suboffset[<index>])

/* entsprechender Kimwitu++-code */
kc_uviews[<index>].name
((abstract_phylum)<node>)->subphylum(<index>)
```

Abbildung 2.3.: Problematische Kimwitu-Ausdrücke

Konversionen werden daher durch eine Konverter, welcher Tcl-Scripts verwendet, implementiert. Siehe Kapitel 3 und Abschnitt 3.3.3.

3. Ein Werkzeug zur Konvertierung

3.1. Anforderungen

Die Konversionen und wie diese zu realisieren sind, ist jetzt bekannt. Siehe Kapitel 2. Bleibt nur noch die Aufgabe, die Realisierung dieser Konversionen in ein Werkzeug einzubetten. Diese Aufgabe lässt sich wie folgt formulieren:

Entwickeln Sie ein Werkzeug, welches vorhandenen Quelltext in einer Sprache an Hand von Konversionsbeschreibungen in Quelltext in einer anderen Sprache konvertiert. Der Quelltext muss sich weiterentwickeln lassen. Beachten Sie dabei, dass die beschriebenen Konversionen nicht vollständig sein müssen.

Aus dieser Aufgabenstellung heraus ergeben sich über die zu realisierenden Konversionen hinaus, eine Reihe allgemeiner Anforderungen für einen Codekonverter.

- Whitespaces und Kommentare im ursprünglichen Quelltext müssen erhalten bleiben. Würden die Whitespaces entfernt werden und damit die Struktur des Codes zerstört werden und würden die Kommentare entfernt werden und damit die Dokumentation des Codes zerstört werden, wäre der Code nicht mehr vom Menschen lesbar. Eine Weiterentwicklung wäre somit ausgeschlossen.

- Nachvollziehbarkeit. Die von dem Konverter gemachten Änderungen müssen vom Konverter dokumentiert werden. Dies muss aus zwei Gründen geschehen.

  Ersten, es darf die Möglichkeit, dass die vom Konverter gemachten Änderungen zum Teil falsch sind[1], nicht ignoriert werden. Für diesen Fall müssen die Änderungen, die auf Grund einzelner Konversionen entstanden sind, identifizierbar sein.

  Zweitens könnten die Änderungen, da sie automatisch ausgeführt werden, zum Teil nicht optimal oder sogar umständlich sein. Um hier beim späteren Lesen des Codes keine Verwirrung entstehen zu lassen und im Gegenteil das Verständnis zu fördern, sind die Änderungen zu Kommentieren[2].

- Erweiterbarkeit. Die gefundenen Konversionen müssen nicht vollständig sein. Von daher ist das Werkzeug so zu gestalten und zu dokumentieren, dass eine Erweiterung um weitere Konversionen leicht zu entwickeln und zu integrieren ist.

[1] Dies kann durch Fehler im Werkzeug, sowie durch falsche Anwendung geschehen
[2] Dies sind sowieso gute Gründe, Quelltext im allgemeinen umfassend zu kommentieren

- Gezieltes und mehrfaches Anwenden von Konversionen. Im Laufe der Konvertierung können noch weitere Konversionen auftreten, die auch automatisch ausgeführt werden sollten. Dies führt zur Erweiterung des Werkzeugs.

  Da die Konvertierung nicht nur aus der Anwendung des Werkzeugs besteht, sondern auch Handarbeit beinhaltet, muss die Konvertierung so flexibel wie möglich steuerbar sein, um bereits vorhandene Handarbeit auch bei weiteren automatischen Konvertierungen beibehalten zu können.

  Es müssen sich Konversionen einzeln ausführen lassen, es müssen sich Konversionen auf einzelnen Dateien ausführen lassen und es müssen sich vor allem Konversionen auf bereits zum Teil konvertierten Code ausführen lassen.

3.2. Design und Funktionsweise

Wie die einzelnen Konversionen realisiert sind, wurde bereits in Kapitel 2 ausführlich beschrieben. Hier geht es jetzt darum zu zeigen, wie die Architektur des entstandenen Werkzeugs aussieht und wie die Konversionen in ihr eingebettet wurden. Weiter ist beschrieben, wie die zuvor aufgezählten allgemeinen Anforderungen, umgesetzt wurden.

3.2.1. Die Konversionen werden an der Kimwitu-Grammatik gesteuert

Für viele der Konversionen muss Kimwitu-Code anhand einer Grammatik analysiert werden, da die Mächtigkeit von regulären Sprachen unzureichend ist. Als Grammatik wurde die bereits vorhandene Kimwitu-Grammatik genommen. Diese Grammatik liegt in Form von yacc-Code als Teil der Kimwitu-Quellen vor.

Die typische Vorgehensweise beim Compilerbau wäre es, den Quelltext zu parsen und einen abstrakten Syntaxbaum zu erzeugen, um dann diesen Baum zu analysieren und um daraus entsprechenden Zielcode zu generieren. Diese Vorgehensweise kann nicht verwendet werden. Zum einen wäre so etwas in der Realisierung sehr Komplex, zum anderen sollen alle Whitespaces und Kommentare erhalten bleiben. Dies würde bedeuten, dass die eigentliche Kimwitu-Grammatik, welche alle Whitespaces und Kommentare ignoriert, nicht verwendet werden kann.

Nun ist ein solcher Mechanismus, auf Grund der Art der Konversionen, gar nicht nötig. Die Vorgehensweise ist nun die folgende: Es werden sämtliche vom Parser gelesenen Zeichen in einer Zeichenkette gespeichert. Die Grammatik wird verwendet, um gesuchte Symbole zu identifizieren und als Referenz auf dieser Zeichenkette zur Verfügung zu stellen.

Sämtliche Konversionen werden auf dieser Zeichenkette ausgeführt. Das heißt, die Grammatik liefert lediglich die Position von Symbolen auf der Zeichenkette. Der Konverter verwendet diese Positionsangaben, um gesuchte Teile aus der Zeichenkette zu entnehmen, zu modifizieren und zu ersetzen. Durch diese Technik bleibt der Code und seine Struktur erhalten. Alle Änderungen sind lokal.

Dieses Vorgehen soll an einem Beispiel näher erläutert werden. Man stelle sich vor, man möchte C-Fragmente innerhalb von Kimwitu-Code identifizieren und mit einem regulären Ausdruck ein Suchen und Ersetzen auf diesen Fragmenten durchführen. C-Fragmente sind in der Grammatik durch ein Nichtterminal dargestellt. Verarbeitet der Parser dieses Symbol, liefert er einen Zeiger auf den Anfang, des durch dieses Symbol repräsentierten Codes, also des C-Fragments, und auf das Ende. Mit diesen Zeigern kann die String-Repräsentation des C-Fragments aus der Zeichenkette kopiert werden. Auf dieser Kopie kann das Suchen und Ersetzen ausgeführt werden. Das Resultat, schreibt man nun an die von den Zeigern angegebene Stelle in die Zeichenkette zurück.

Um die Skalierbarkeit bezüglich der Quelltextgröße zu waren, wird eine solche Zeichenkette nicht über ein ganzes Eingabedokument gehalten. Sondern sie wird an geeigneten Stellen in das Ausgabedokument geschrieben. Einmal ausgegeben, lässt sich der Code nicht mehr verändern. Geeignete Stellen sind also solche, an denen alle zu machenden Änderungen bereits abgeschlossen sind. Solche Stellen konnten in der Kimwitu-Grammatik gefunden werden. Diese Stellen sind nach dem vollständigen verarbeiten der Nichtterminale *phylumdeclaration, includedeclaration, functiondeclaration, rwdeclaration, unparsedeclaration, rviewdeclaration, uviewdeclaration, storageclassdeclaration*. Siehe [vEB00].

Die Kimwitu-Grammatik ist mit der Kimwitu++-Grammatik kompatibel, dass heißt, es lässt sich auch bereits konvertierter Code mit der Kimwitu-Grammatik parsen.

Der Parser ist in einer *lex* und yacc Kombination realisiert. Die beschriebene Zeichenkette ist durch die abstrakte Klasse **AbstractPrinter** realisiert. Von dieser Klasse wird für jede Konvertierung genau eine Instanz erzeugt. Der Scanner gibt diesem *Printer*-Objekt die gelesenen Zeichen. **AbstractPrinter** stellt Methoden zur Verfügung, die das Auslesen und das Ersetzten über Referenzen ermöglichen.

Die Abstraktion dieser Klasse bezieht sich lediglich auf das Schreiben des konvertierten Codes. Es gibt eine Realisierung **Printer**, welche das Schreiben in einen Ausgabestrom ermöglicht.

Die beschriebenen Referenzen auf die Symbole in der Zeichenkette sind über die Klasse **CodeReference** realisiert. Dieser Typ wird auch zur Bildung des *value types* verwendet. Also des Typen, der den Wertetyp der Symbole im Parser darstellt. Siehe [Pax95] und [DSt95].

3.2.2. Allgemeine Umsetzung, Einbettung und Management der Konversionen

Es wurden in Kapitel 2 Konversionen, also Typen von zu machenden Änderungen, identifiziert. Nun müssen diese Konversionen realisiert werden. Wie die einzelnen Konversionen zu realisieren sind, wurde bereits geklärt. Jetzt ist zu klären, wie diese Realisierungen in das Werkzeug eingebettet sind.

Hierbei ist zu beachten, dass gemeinsame Funktionalität in einer Basisklasse realisiert wird und eine Architektur entsteht, die das Hinzufügen von Realisierungen von Konversionen einfach ermöglicht.

Die entstandene Architektur sieht folgender maßen aus. Es gibt eine abstrakte Klasse **AbstractConverter**. Spezialisierungen dieser Klasse können eine oder mehrere

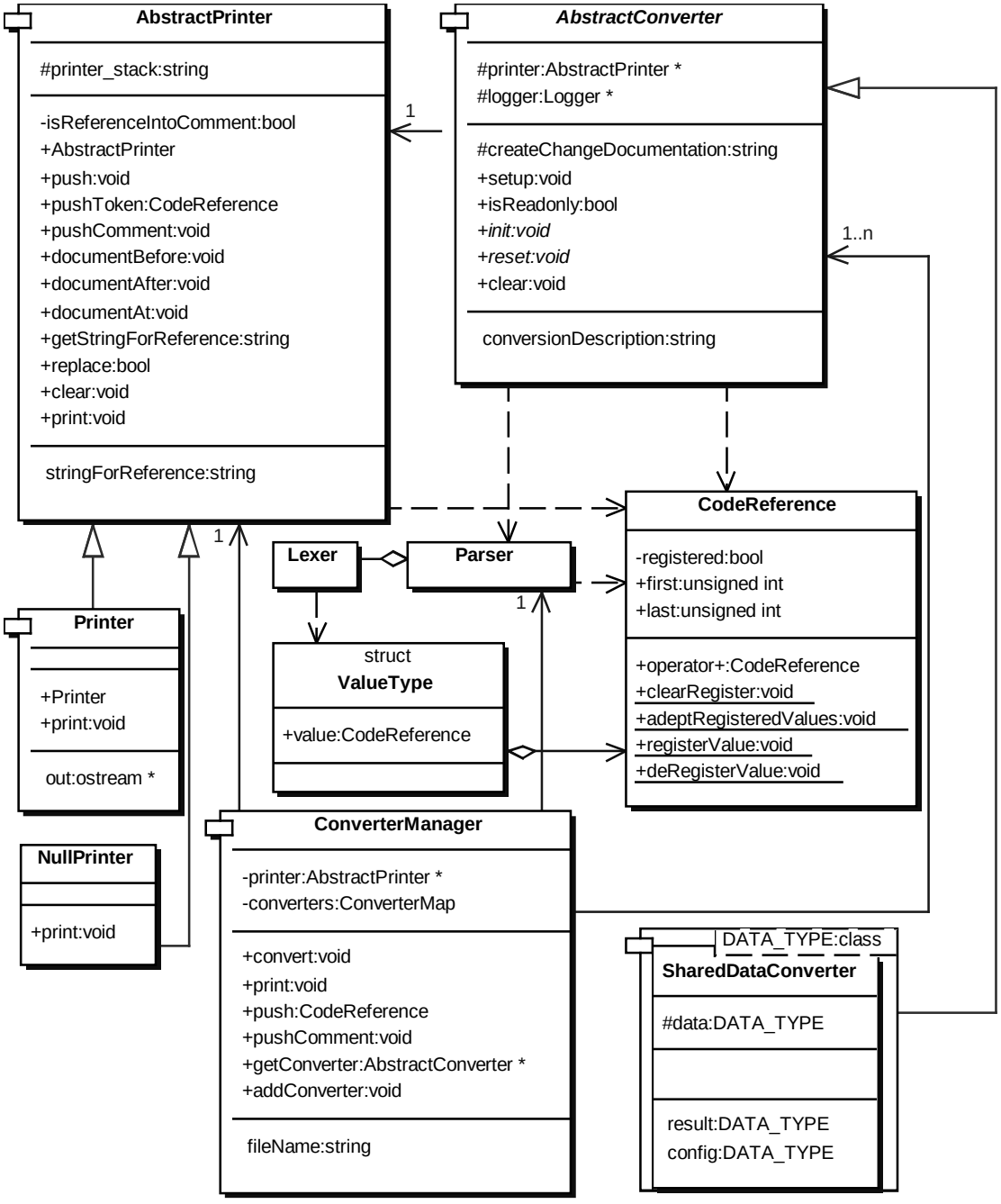

Abbildung 3.1.: Klassendiagramm, Grammatik gesteuertes Konvertieren

gleichartige Konversionen realisieren. Diese Klasse bietet ein Interface zum Initialisieren, Konfigurieren und Steuern solcher *Converter*-Objekte. Weiter bietet sie einem Zugang zum Parsermechanismus und zum *Printer*. Welche Konverterklasse welche Konversion realisiert, kann im Anhang A nachgeschlagen werden.

Weiter gibt es eine Klasse **ConverterManager**, von der eine Instanz gebildet wird. Dieses Objekt dient zum Verwalten einzelner *Converter*-Instanzen. Die bei einer Konvertierung zu verwendenden Konverter und damit Konversionen müssen an diesem Manager registriert werden. So lässt es sich realisieren, dass eine Konvertierung mit ausgewählten Konversionen möglich ist. Der Manager steuert und konfiguriert die registrierten Konverter über das durch **AbstractConverter** definierte Interface. Das heißt er initialisiert sie für jede zu konvertierende Datei neu und konfiguriert sie mit einer entsprechenden *Printer* Instanz. Er stellt die Verbindung zum Parser dar. Das heißt aus der Grammatik heraus werden die Methoden der Konverter nicht direkt gerufen, sondern die Rufe werden vom *Manager* delegiert. Je nachdem ob der entsprechende Konverter registriert ist oder nicht, werden diese Rufe an den Konverter weitergeleitet oder ignoriert.

Es gibt eine Spezialisierung von **AbstractConverter** die Klasse **SharedDataConverter**. Realisierungen dieser Klasse wird es ermöglicht, Daten mit einander auszutauschen, um so Daten gemeinsam zu nutzen. So ist es möglich das Verhalten eines Konverters und damit das einer Konversion, abhängig von einer anderen Konversion zu machen. So ist es möglich auch Konversionen zu realisieren, die nicht nur auf rein lokal vorhandenen Informationen angewiesen sind. Das ist zum Beispiel im Abschnitt 2.2.1 ausgenutzt worden.

SharedDataConverter ist eine Templateklasse mit einem Templateparameter vom Typ Klasse. Konverter, die diese Klasse mit dem gleichen Templateparameterwert realisieren, können gemeinsame Daten verwenden. Wobei der Typ dieser Daten durch den Templateparameter vorgegeben wird. Diese abstrakte Klasse liefert auch das Interface zum gegenseitigem Austausch dieser Daten.

3.2.3. Dokumentierung und Protokollierung

Die Klasse **AbstractPrinter** bietet weiterhin eine virtuelle Methode zum generieren Konverter spezifischer Änderungsinformationen, welche vom *Printer* zur Generierung von Kommentaren zur Dokumentierung der Änderungen im generierten Code verwendet wird.

Die Klasse **Logger**, von der eine globale Instanz erzeugt wird, ermöglicht die Protokollierung der Konvertierung. Das heißt sie ermöglicht es, die gemachten Änderungen zu zählen. Sie stellt Methoden zur Verfügung, die es den Konvertern ermöglicht, den *Logger* über eine gemachte Änderung zu informieren. Die *Logger*-Instanz wird von der *ConverterManager*-Instanz verwaltet.

3.2.4. Benutzer Interface zur Kontrollierung der Konvertierung

Es gibt eine Klasse **Config**, von der eine globale Instanz erzeugt wird. Diese Instanz verwaltet globale Einstellungen, die das Verhalten des Programms beeinflussen. Diese

Einstellungen sind die Aktivierung und Deaktivierung der Protokollierung, der Grad der generierten Dokumentierung und die Aktivierung und Deaktivierung der Ausgabe von Statusinformationen.

Um die Anforderung *Gezieltes und mehrfaches Anwenden* zu erfüllen, wurde die Entscheidung getroffen, dass eine Steuerung des Programmverhaltens durch Parameter ungeeignet ist. Statt dessen wurde die Scriptsprache Tcl[3] als Nutzerinterface verwendet.

Die Sprache Tcl wurde um Kommandos erweitert, die das Erzeugen von Konvertern, das Austauschen von Daten zwischen Konvertern, die einzelne Konvertierung von Dateien mit bestimmten Konvertern und das setzen von globalen Einstellungen ermöglicht. Siehe Abschnitt 3.3 und C.

Die Klasse **Interp** erzeugt einen Tcl-Interpreter, definiert die Tcl-Kommandos und führt das vom Nutzer übergebene Script aus. Die Tcl-Kommandos verwenden die *ConverterManager*-Instanz, um die Konvertierung gemäss des Scripts zu steuern. Die Klasse **Interp** verwendet die *Config*-Instanz, um globale Einstellung gemäss des Scripts zu setzen.

3.3. Verwendung bei der Konvertierung

3.3.1. Empfohlene Vorgehensweise

Die Vorgehensweise bei der Konvertierung liegt hier in Abbildung 3.2 als UML Aktivitätsdiagramm vor.

3.3.2. Scripte

Neben dem gewohnten Tcl Befehlsumfang können folgende Befehle zur Steuerung des Konverters verwendet werden.

```
converter create <type> [ro]
converter delete <converter-id>
converter config <target-id> <source-id>
```

Mit diesem Befehl können Instanzen der verschiedenen Konverter erzeugt und zerstört werden und es können Daten zwischen *SharedDataConverter* gleichen Typs ausgetauscht werden. Die einzelnen Konverter können mit der Eigenschaft *read only* erzeugt werden.

```
convert <converter-id list> <file-name list/map>
```

Mit diesem Befehl können die zuvor erzeugten Konverter ausgeführt werden. Für *read only* Konverter muss eine Liste von zu konvertierenden Dateien übergeben werden und für normale Konverter eine Abbildung von Quell- auf Zieldateien.

```
config <key> <value>
```

[3] Tool and Command Language. Siehe [Ost93].

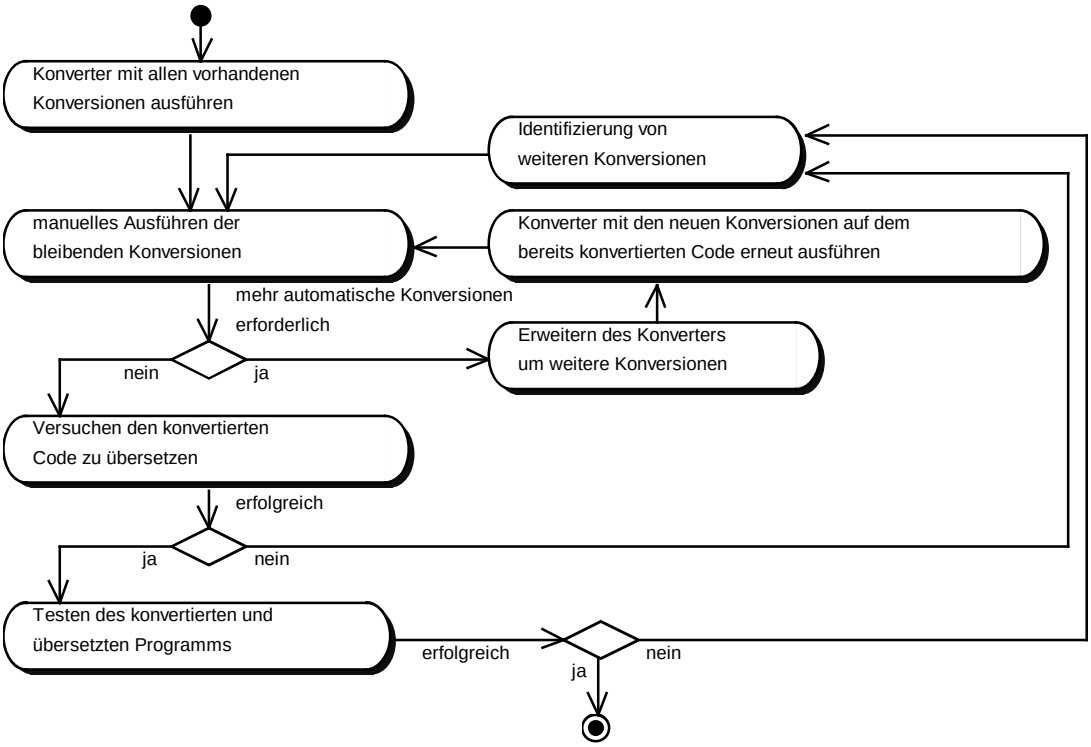

Abbildung 3.2.: Empfohlene Vorgehensweise als Aktivitätsdiagramm

Dieser Befehl kann verwendet werden, um die globale Konfiguration festzulegen. Mögliche Werte für **key** sind: **documentation, summary, verbose**.

Beispiel Listing 3.1 zeigt ein Beispielscript. Dieses Script nimmt alle Dateien mit der Endung *.k* in dem Verzeichnis *src*, konvertiert diese und schreibt die Ergebnisdateien mit demselben Namen in das Verzeichnis *target*. Dabei werden zuerst alle Dateien nach Phylumdefinitionen durchsucht, um dann mit den Namen der Phyla die Konvertierung von Funktionsrufen und das Konvertieren mit regulären Ausdrücken zu konfigurieren. siehe Abschnitt 2.2.1. Der Zielcode wird mit der stärkst möglichen Dokumentierung versehen. Es werden keine Statusinformationen ausgegeben. Es wird eine Übersicht über die Änderungen erzeugt und ausgegeben.

```
set srcDir    "./src"
set targetDir "./target"

set files [exec ls "$srcDir/*.k"]

set filesMap ""
set filesList ""
foreach file $files {
    lappend filesList $file
    lappend filesMap [list $file
            [string join "$targetDir/" [file tail $file]]
```

```
}

config documentation 3
config verbose 0
config logging 1

set converters(kr)  [converter create kr_functions]
set converters(uvv) [converter create uview_view]
set converters(re)  [converter create regular_expression]
set converters(cpn) [converter create collect_phyla ro]
set converters(srp) [converter create script]
set converters(ftm) [converter create function_to_method]

convert $converters(cpn) $fileList
converter config $converters(ftm) $converters(cpn)
converter config $converters(re) $converters(cpn)

convert [list $converters(kr) $converters(uvv)
        $converters(re) $converters(srp)
        $config(ftm)] $fileMap

converter delete $converters(kr)
converter delete $converters(uvv)
converter delete $converters(re)
converter delete $converters(cpn)
converter delete $converters(srp)
converter delete $converters(ftm)
```

Listing 3.1: Beispiel Script zur Konvertersteuerung

Für eine weitere Dokumentierung der vorhandenen Befehle sei auf die Manpage des Konverterwerkzeugs verwiesen, Anhang C.

3.3.3. Implementation von weiteren Konversionen

Es gibt je nach Art der hinzugekommenen Konversion, mehrere Möglichkeiten diese zu realisieren.

Hinzufügen eines regulären Ausdrucks

Wenn sich die Konversion nur durch ein reguläres Suchen und Ersetzen realisieren lässt, reicht es aus, den bereits existierenden Konverter für reguläre Ausdrücke, um einen weiteren Ausdruck zu erweitern. Die entsprechenden Definitionen befinden sich in der Datei *regular.h* im Quellcode-Verzeichnis der Konverterdistribution.

```
typedef enum {IDENTIFIER, CONTEXTED, PHYLUM_PREFIX} RETypes;
typedef enum {ALL = 0, CCODE = 1 } RETargets;

struct REConversion {
    const string match;
    const string substitude;
    const RETypes type;
    const RETargets target;
    string preText;
    string sufText;
    ...
};
```

Abbildung 3.3.: Der Typ REConversion

Dieser Konverter definiert eine statisches Array vom Typ **REConversion**. Siehe Abbildung 3.3. Das Feld **match** enthält den Ausdruck, der beschreibt, was ersetzt werden soll. Das Feld **substitude** den einzusetzenden String. Das Feld **type** beschreibt die Art bzw. den Kontext des Suchen und Ersetzens. Der Wert **IDENTIFIER** wird verwendet, um Bezeichner zu Suchen, dass heißt der angegebene reguläre Ausdruck wird um einen Kontext erweitert, der bewirk, dass **match** nur als ganzes Wort gefunden wird. Der Wert **CONTEXTED** ermöglicht es in **preText** und **sufText** einen eigenen Kontext anzugeben, wobei der reguläre Ausdruck folgendermaßen gebildet wird: *preText-match-sufText*. Das Feld **target** gibt an, wo die Suche durchgeführt wird. Bei **CODE** als Wert wird nur in C-Fragmenten gesucht, bei **ALL** im gesamten Code.

Hinzufügen eines Konvertierungsscripts

Es existiert bereits ein Konverter, welcher Tcl-Prozeduren verwendet. Siehe Abschnitt 2.4. Jede dieser Tcl-Prozeduren stellt die Realisierung einer Konversion dar. Hier kann die Datei *scripts.tcl* im Quellcodeverzeichnis der Konverterdistribution um eine weitere Prozedur erweitert werden.

Die Prozeduren sehen folgendermaßen aus: Der Name der Prozedur ist beliebig. Er muss in *script.h* dem Array **scripts** hinzugefügt werden. Die Prozedur darf keine Parameter haben. Die Prozedur wird immer auf einzelne Quellcode-Fragmente angewandt. Siehe Abschnitt 3.2.1. Das entsprechende Fragment liegt in der globalen Variable **string_to_convert**. Die von der Konversion zu machenden Änderungen können an dieser Variable vorgenommen werden. Als Rückgabewert dieser Prozedur, wird die Anzahl der gemachten Änderungen als Integer-Wert erwartet.

Implementierung eines zusätzlichen Konverters

Reichen die ersten beiden Möglichkeiten zur Realisierung der neuen Konversion nicht aus, bleibt nur noch die Möglichkeit, einen eigenen Konverter zu schreiben. Es soll nun

ein kurzer Überblick gegeben werden, wie bei einer solchen Implementation vorzugehen ist und was hierfür getan werden muss. Für eine nähere Dokumentation sei an den Code des existierenden Konverters **UVViewConverter** als Referenzimplementation verwiesen.

Welcher abstrakte Konverter ist als Basis zu verwenden? Um den entstehenden Konverter in dem bereits existierenden Framework verwenden zu können, muss dieser von einem der abstrakten Konverter erben.

Das kann die Klasse *AbstractConverter* sein. Diese ist dann zu verwenden, wenn der Konverter nicht auf den Austausch von Daten mit anderen Konvertern angewiesen ist. Die Klasse *SharedDataConverter* sollte als Basisklasse verwendet werden, wenn gerade dies erforderlich ist.

Implementation des Interfaces Die Abstrakten Klassen definieren die Methoden *init* und *reset*. Diese müssen implementiert werden. Die Methode *init* wird vor der Konvertierung einmal aufgerufen. Hier sollte der Konverter initialisiert werden. Die Methode *reset* wird nach dem Konvertieren jeder Quellcodedatei aufgerufen. Hier kann der Konverter auf ein erneutes Konvertieren vorbereitet werden.

Der Aufzählungstyp *ConverterTypes* enthält einen Wert für jeden Konverter. Dieser Typ steht in der Datei *converter.h*. Er muss um einen Wert für den neuen Konverter erweitert werden. Dieser Wert muss beim Aufruf des Basiskonstruktors übergeben werden.

Bei einem *SharedDataConverter* muss der Template Parameter mit dem Typ, der auszutauschenden Daten, als Wert gebunden werden.

Ansteuerung des Konverters Der Konverter wird über die Grammatik gesteuert, dass heißt in der Grammatik können die C-Fragemente um Methodenrufe an der von der *ConverterManager*-Instanz gelieferten Konverterinstanz erweitert werden. Hierfür sollte in *manager.h* ein Makro, entsprechend der bereits existierenden Makros, geschrieben werden.

Erweitern des Interpreters In der Datei *inter.cc* muss den Methoden **CreateConverter** sowie **DeleteConverter** und bei einem *SharedDataConverter* der Methode **ConfigureConverter**, entsprechend dem Code der existierenden Konverter, Code hinzugefügt werden.

A. Konversionen

Konversion	Referenz	Realisiert durch
Kernighan Ritchie Funtionsdefinitionen	2.2.1	KRFunctionConverter
Schlüsselwort *view*	2.2.2	ViewKeyWorkConverter
Phyladefinitionen mit *int* und *real*	2.2.3	IRPhylaConverter
Methodenrufe statt Funktionsrufe	2.3.1	FToMethodConverter
Geänderte C/C++-Bezeichner	2.3.1	REConverter
Zuweisung an *int*- und *real*-Phyla	2.3.3	-
Zuweisungen von Phyla	2.3.4	-
Nicht Berücksichtigung von *const*	2.3.5	-
Verwendung von Hashstatistiken	2.4	-
Verwendung des Speichermanagements	2.4	-
Geänderte Definition von *prod_sel*	2.4	REConverter
Änderungen an Assertion Makros	2.4	FToMethodConverter
Verwendung von *is_list*	2.4	-
GLOBAL_ATTRIBUTE_PATCH	2.4	-
Verwendung von *kc_voidptr, kc_voidnptr*	2.4	-
Zugriff auf Subphyla	2.4	ScriptConverter
Verwendung der Viewnamensliste	2.4	ScriptConverter

Tabelle A.1.: Bekannte Konversionen

Ausdruck	Ersetzung	Umgebung
view	uview	Bezeichner
boolean	bool	Bezeichner
True	true	Bezeichner
False	false	Bezeichner
prod_sel	prod_sel()	wie Beizeichner, es dar kein ‘(‘ folgen
base_view	base_uview	Bezeichner
kc_view_count	last_uview	Bezeichner
kc_PhylumInfo	phylum_info	Bezeichner
kc_OperatorInfo	operator_info	Bezeichner
kc_last_uview	last_uview	Bezeichner
kc_uviews	uviews	Bezeichner
kc_rviews	rviews	Bezeichner
kc_ht_reuse	ht_clear	Bezeichner
kc_ht_clear	ht_clear	Bezeichner
kc_ht_assign	ht_assign	Bezeichner
kc_ht_assigned	ht_assigned	Bezeichner
kc_tag_	impl_	Prefix eines Phylumnamens
kc_phylum_	phylum_	Prefix eines Phylumnames
kc_last_phylum	last_phylum	Bezeichner
kc_last_operator	last_operator	Bezeichner
kc_size_t	size_t	Bezeichner
kc_print_operator_statistics	print_operator_statistics	Bezeichner
kc_enum_operators	enum_operators	Bezeichner
kc_enum_phyla	enum_phyla	Bezeichner
kc_one_before_first_operator	one_before_first_operator	Bezeichner

Tabelle A.2.: Verwendete reguläre Ausdrücke

Kimwitu-Prefix	Kimwitu++-Name	Stelle	Art des Phylums	Art des Rufes
eq_	eq	1	Phylum	Methode
free_	free	1	Phylum	Methode
freelist_	freelist	1	Listphylum	Methode
concat_	concat	1	Listphylum	Funktion
reverse_	reverse	1	Listphylum	Methode
length_	length	1	Listphylum	Methode
last_	last	1	Listphylum	Methode
map_	map	1	Listphylum	Methode
filter_	filter	1	Listphylum	Methode
print_	print	1	Phylum	Methode
fprint_	fprint	2	Phylum	Methode
rewrite_	rewrite	1	Phylum	Methode
unparse_	unparse	1	Phylum	Methode
assert_	assertPhylum	3	Phylum	Assertion
copy_	copy	1	Phylum	Methode

Tabelle A.3.: Funktionsrufe und Assertions

B. Statistiken der Konvertierung der SITE-Tools

Hier die Zusammenfassung der von dem Werkzeug gemachten Änderungen an der SITE Semantik-Analyse und an dem SITE Code-Generator. Diese Übersichten wurden von dem Werkzeug erzeugt.

Die Semantik-Analyse

```
Kernighan Ritchie function changes: 751
int, real-phyla definition changes: 4
collected phyla: 302
view keyword changes: 11
funtion call changes: 765
regexp changes: 1763
script changes: 40

total change count : 3330
```

Der Code-Generator

```
Kernighan Ritchie function changes: 873
int, real-phyla definition changes: 4
collected phyla: 291
view keyword changes: 26
funtion call changes: 1771
regexp changes: 2486
script changes: 41

total change count : 5201
```

C. Die Manpage des Konverters kc2kc++

kc2kc++(1) User Manuals kc2kc++(1)

NAME
kc2kc++ - automatic code conversion from Kimwitu to Kimwitu++

SYNOPSIS
kc2kc++ -s script-file [script arguments]
kc2kc++ -b [-v -s -d=0..3] files out-directory

DESCRIPTION
kc2kc++ helps converting Kimwitu program code into Kimwitu++ program code. It's not built to convert Kinwitu code fully automatic, but supporting you by solving often occurring transformation problems. These problems are called conversions for shortness.
The converting process is controlled by tcl-scripts, that you have to provide or by the predefined built in script. The script controlling approach is made, to give you the full control on when, what conversion should be made and in which order and on what file they should be made.
The program works with the original kimwitu grammar. That means that the code to convert may not include any macros, that destroy the correct syntax of the kimwitu code. Anyway, the kimwitu grammar has good error recovery. Thus code containing macros may be converted partially.

OPTIONS
-b converts all files and writes output to out-dir. The files will be converted with all implemented conversions by a predefined script.

-v kc2kc++ prints verbose information, when converting files with the predefined script.

-s kc2kc++ prints a statistical summary after converting files with the predefined script.

-d=0..3
kc2kc++ generates documentation in the converted code, when converting files with the predefined script. The documentation is provided as C++ comments. There are four levels of documentation, 0 is the default:
0) no documentation at all, no comments are generated.
1) manipulated pieces of code are signed with a simple comment, that indicates that the code was modified by kc2kc++. All generated comments start with the string kc->kc++.
2) manipulated pieces of code are signed like 1, but with information about what converter converted that specific piece of code.
3) manipulated pieces of code are signed like 2, but with the original piece of code included in the comment. The strings /* and */ are escaped with |* and *|.

CONVERTERS
kc2kc++ consists of converters. A converter realizes one or more conversions. A conversion can be understood as the type of changes. There is a special class of converters, the shared data converters. These converters have the possibility to exchange data with each other. The normal converters run on there own.
A converter is run on one or more files including kimwitu-code. And outputs this code with all changes, described by the conversions realized by this converter.
Now follows a list of the kc2kc++ converters with an description of its conversions.

NAME	CLASS	SHARED DATA TYPE
kr_functions	KRFuntionConverter	
uview_view	ViewKeyWordConverter	
collect_phyla	CollectPhylaConverter	list of phyla names
function_to_method	FToMethodConverter	list of phyla names
regular_expression	REConverter	
script	ScriptConverter	
int_real_phyla	IRPhylaConverter	

kr_functions
Converts Kernighan/Ritchie function definitions to ANSI–C definitions.

uview_view
Changes all occurrences of the keyword view to uview.

collect_phyla
This converter makes no changes. Its is just a producer of shared data. It collects all phyla names from phyla definitions and provides them as shared data.

function_to_methods
This converter converts functioncalls on phylas to methodcalls on the according instances. It can only convert the functioncall on phyla that it knows. Thus the phyla names have to be provided as shared data. E.g. from the collect_phyla converter.

regular_expression
There are a lot of identifiers in the generated code of kimwitu und kimwitu++ that have changed. The regular_expression makes the needed changes. The expressions are listed in the src/regular.h file in your kc2kc++ source code distribution.

script This realizes smaller conversions on the use of generated kimwitu code. Its realized by tcl scripts. The scripts are listed in the src/scrips.tch file in your kc2kc++ source code distribution.

int_real_phyla
This converter changes the occurrences of int and real in phyladefinitions with the according kimwitu++ basic types.

SCRIPT LANGUAGE
The chosen script language is tcl. Thus all tcl commands are allowed. Tcl was extend by commands that allow to control the kc2kc++ converting process.

config <configuration–key> <value>

The config command sets global configuration settings. The configuration–keys are:

verbose
This key controls whether verbose information is print to stdout on conversation or not. Possible values are 0 and 1, on and off. Default is off.

summary
This key controls whether or nor a statistical summary is accumulated on the converting process or not. It is print to stdout, when the program terminates. Possible value are 0 and 1, on and off. Default is off.

documentation
This key controls the level of documentation. kc2kc++ generates documentation in the converted code, when converting files with the predefined script. The documentation is provided as C++ comments. There are four levels of documentation, that can be provided as value argument. 0 is the default:
0) no documentation at all, no comments are generated.
1) manipulated pieces of code are signed with a simple comment, that indicates that the code was modified by kc2kc++. All generated comments start with the string kc–>kc++.
2) manipulated pieces of code are signed like 1, but with information about what converter converted that specific piece of code.
3) manipulated pieces of code are signed like 2, but with the original piece of code included in the comment. The strings /* and */ are escaped with |* and *|.

The configuration can be change between different calls of convert, to enable different behavior for different converter runs or for different files.

converter <sub–command> <sub–command arguments>

This command allows you to create, delete converters and it allows you to interchange data between shared data converters of equal type. The sub–commands are:

converter create <converter–name> [ro]

This command creates a converter. The converter names are provided in the CONVERTER section. The optional parameter ro allows you to configure the converter read–only. This can be meaningful for shared data converter to produce data for share or for testing purposes. The command returns a converter–id. This is a handle for the converter instance created by the command.

converter share <target–converter–id> <source–converter–id>

This command triggers the data share between two converters. It transfers the data from the source-converter-id to target-convert-id. Only shared data converter can share data. Only converter of equal shared data type can share data.

```
converter delete <converter-id>
```

This command destroys the converter instance indicated by the converter-id.

```
convert <converter-id-list> <files>
convert <converter-id-list> <files-map>
```

The convert command converts files with the given converters. The converters have to be provided as a list of converter-ids. There are to ways to provide files. The first is a list of file names. This is for read-only converters only. A filename list must not be given to a converter with other then read-only converters. The second possibility is to provide a files-map. This is a key-value list of source-file-name/target-filename pairs.

Beside this commands there are a few supporting tcl procedures for creating file lists and file maps. These procedures are provided in the file src/procs.tcl of your kc2kc++ source code distribution.

Script-arguments that are provided when executing kc2kc++ with the -s option are provided in the global variable arguments as a list.

EXAMPLES

Assume you want convert Kernighan/Ritchie function definitions in file foobar.k

```
convert [converter create kr_functions] "./foobar.k ./foobar.k++"
```

Assume you want to read all phylum definitions from the automatically generated file ro_foobar.k and use it to convert the function calls in all kimwitu files provided as arguments and want to put the output to the ./out directory.

```
set collect [converter create collect_phyla ro]
convert $collect ./ro_foobar.k
set function_calls [converter create function_to_method]
converter share $function_calls $collect
convert $function_calls [create_filemap $arguments ./out]
```

A more complex example is the kc2kc++ build in script. Its located in the procedure convert_file in the file src/single.tcl of your kc2kc++ source code distribution.

FURTHER READING

There is a thesis about kc2kc++: Automatische konvertierung von kimwitu nach kimwitu++, http://www.informatik.hu-berlin.de/~scheidge. This document gives you information on all conversion that could be done by kc2kc++, a documentation on writing the controlling scripts including examples. And gives you a basic idea on how to extend kc2kc++ with new conversions.

BUGS

no bugs reported - before reporting bugs remind yourself that this tool is not supposed to be fully automatic converter that generates you hunderd percent syntax corrext and semantic equivalent Kimwitu++-Code out of your Kimwitu code.

AUTHOR

Markus Scheidgen <scheidge@informatik.hu-berlin.de>

SEE ALSO

tclsh, kimwitu, kimwitu++

Literaturverzeichnis

[ASU87] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers, Principles, Techniques and Tools*. Bell Telephone Laboratories, Inc. 1987.

[C90] *ISO/IEC 9899:1990 Programming languages - C*. International Standard, 1990.

[C++98] *ISO/IEC 14882:1998 Programming languages - C++*. International Standard, 1998.

[DSt95] Charles Donnelly, Richard Stallman: *Bison, The YACC-compatible Parser Generator*. Bison Version 1.25, Free Software Foundation, Boston, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Inc. 1995.

[NP01] Toby Neumann, Michael Piefel: *Kimwitu++, A Term Processor*. Users Guide 1.0, Humboldt-Universität zu Berlin, 2001.

[OMG01] Object Management Group: *Unified Modeling Language Specification*. OMG document formal/01-09-67, September 2001.

[Ost93] John K. Ousterhout: *Tcl and the Tk Toolkit*. University of California, Berkeley, 1993.

[Pie99] Piefel, Michael: *Exemplarische Konvertierung eines SITE-Tools von Kimwitu nach Kimwitu++*. Studienarbeit, Humboldt-Univarsität zu Berlin, 1999.

[Pax95] Vern Paxson: *Flex, A fast scanner generator*. Edition 2.5, University of California, Berkeley, 1995.

[vEB00] Peter van Eijk, Axel Belfinfante: *The Term Processor Kimwitu Manual and Cookbook*. University of Twente, Enschede, The Netherlands, 2000.

[Z100] CCITT: *SDL - Specification Description Language, International Standard Recommendation Z.100*. Genf, 1992.