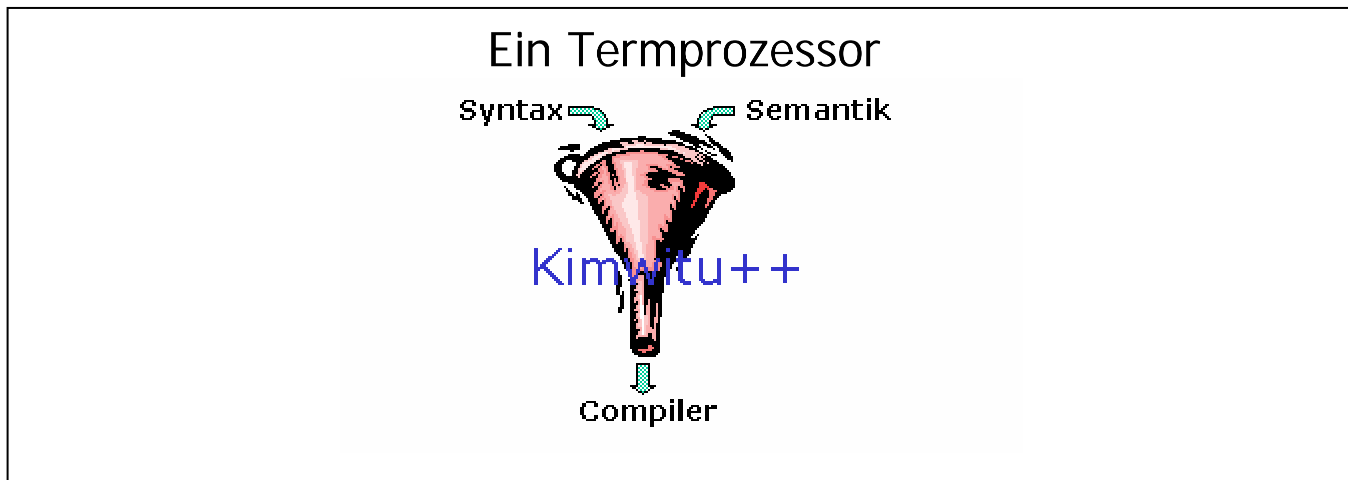
Systemanalyse

INSTITUT FÜR INFORMATIK
Humboldt-Universität zu Berlin

Kimwitu++

Konrad Voigt & Glenn Schütze

Ein Termprozessor



Gliederung

1. kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. komplexes Bsp.: <Minako>
6. kleines Bsp.: <ramcy>
7. Quellen
8. Fragen

1. **Kurze Geschichte**
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

1. kurze Geschichte

- Der Name Kimwitu
  - Swahli :
  - Witu– der Baum
  - M- Plural
  - Ki- -lig
  - Kimwutu ~ „bäumlig“

1. **Kurze Geschichte**
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

1. kurze Geschichte

- Vorgänger Kimwitu für C entstanden an der Universität Twente, Niederlande
- Anpassung an C++ am Lehrstuhl für Systemanalyse seit 1997
- Seit 2000 unter GPL lizensiert

1. **Kurze Geschichte**
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

1. kurze Geschichte

- Erweiterung zu C++
  - Erzeugt C++-Code
- Zweck: Baumbearbeitung
  - Baumdefintion -> Erzeugung
  - Abschreiten -> unparse
  - Umformen -> rewrite

2. AST

AST

1. Kurze Geschichte
2. **AST (Wdhl.)**
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

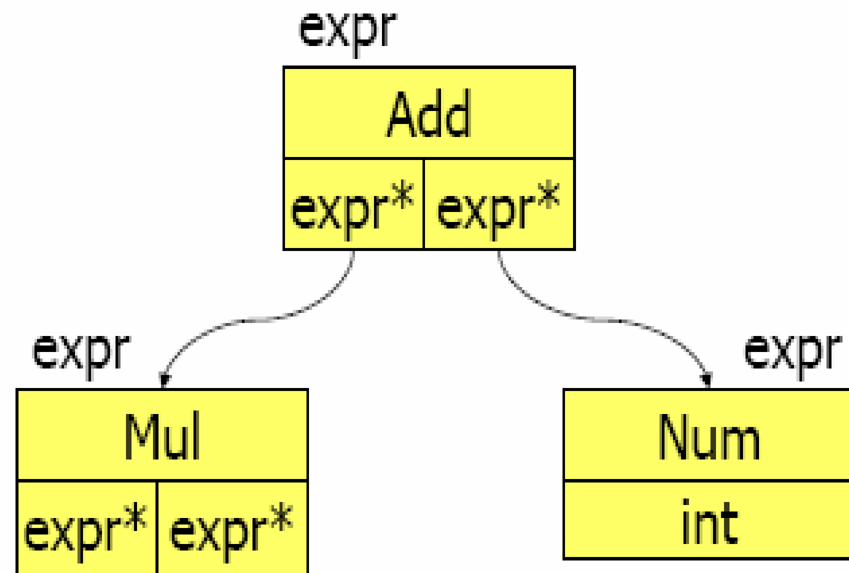
2. AST

- Def.: abstrakter Syntaxbaum
  - ein abstrakter Syntaxbaum (AST) ist ein komprimierter Parse-Baum, bei dem (fast alle) Knoten für Nicht-Terminalsymbole entfernt wurden

2. AST

1. Kurze Geschichte
2. **AST (Wdhl.)**
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen


```
expr: Add ( expr expr )
    | Mul ( expr expr )
    | Num ( int );
```



1. Kurze Geschichte
2. **AST (Wdhl.)**
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

2. AST

- In Kimwitu++
  - Knoten heißen Phylum (*Pl.:* Phyla)
  - Jedes Phylum besitzt Operatoren, die die Ausprägung der Kinderknoten bestimmen

3. Syntax

Syntax

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Eingabedatei(en):
  - Kimwitu++-Dateien enden auf .k
  - Möglich: eine Eingabe Datei
  - Besser: auf mehrere zweckbestimmt verteilt
  - Kimwitu++ generiert aus Abstrakter Grammatik den AST

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Schnittstellen:
  - Zu flex & bison mit Hilfe von YYSTYPE
  - Kimwitu++ Schalter --yystype generiert yystype.h
  - Enthält benötigte Typen:
    - Für jedes Phylum yt_phylumname
    - In flex yylval.yt_phylumname für Tokens mit Wert
  - Durch semantische Aktionen in bison:
    - Knotenerzeugung über Phylumoperatoraufruf

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Kimwitu++ Dateien enthalten:
  - C++ Quelltext (Funktionen etc.)
  - Phylumdefinitionen
  - Regeln (unparse und rewrite)
  - Kimwitu++ Deklarationen

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- C++ Quelltext
  - muss mit %{ und %} eingeschlossen werden
  - umleitbar mit Hilfe von Redirektoren:
    - KC_TYPES : k.cc
    - KC_UNPARSE : unpk.cc
    - KC_REWRITE : rk.cc
    - KC_FUNCTIONS_ file : file.cc
    - redirect_HEADER für Header der Datei, welche mit redirect gemeint ist

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Linke Seite: Phylum-Name;
- Rechte Seite: Liste von alternativen Operatoren

```
phylum := phylum_name ":" operator { "|" operator }* [ attributes ] ";„

operator := operator_name "(" [ phylum_name ] { phylum_name }+ ")"
```

Bsp.:

```
expr: Add ( expr expr )
     | Mul ( expr expr )
     | Num ( integer );
```

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Phyla können um Attribute erweitert werden

```
attributes :=
"{" { attr_type attr_name [ "=" init_value ] ";"} [ cpp_part ] "}"
cpp_part :=
"{" arbitrary_cpp_code "}"
```

- Bsp.:
  - Wertberechnung beim UPN Rechner

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Vordefinierte Phyla
  - mkinteger(int)
  - mkcasestring(char*)
  - mkreal(float)
  - mknocasestring(char*)

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Phyla als Listen
  - 1. Möglichkeit:
    - Rechtsrekursive Definition

      expr_list: Nilexpr_list ( )

      | Consexpr_list (expr expr_list);
  - 2. Möglichkeit (besser):
    - Von Kimwitu++ bereitgestellte Listenfunktion

      expr_list: list expr;

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Muster (pattern)
  - Literale für vordefinierte Phyla sowie „ * “
  - Phylum-Operatoren mit Muster als Parameter
  - Zuweisung von Muster an Variable
  - Aufzählung von Muster getrennt durch „ , “

1. Kurze Geschichte
2. AST (Wdhl.)
3. **Syntax**
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

3. Syntax

- Muster in Kimwitu++

Muster	Bedeutung
*	passt auf jeden Term
Add(*,*)	passt auf jedes Add
Add(a,b)	passt auf Add und weist die Kinder an Variablen a und b zu
Add(a,a)	passt auf Add, wenn die Kinder gleich sind und weist an a zu
Add(Num(*),*)	passt auf Add, wenn erstes Kind ein Num ist
Add, Mul	passt auf Add oder Mul

4. unparse

unparse & rewrite

(tree walking)

4. unparse

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

- Abschreiten des Baumes (Tree walking)
  - An Termen, bei gegebener Regeln, angegebene Aktion ausführen
  - Linke Seite: Muster, Rechte Seite: „unparse“ Anweisung

```
Muster ->[opt_view: unparse_Anweisung];
```

  - Unparse Anweisung:
    - Zeichenketten, Term-Variablen, Attribute, C-Code in geschweiften Klammern „${“ und „$} “

4. unparse

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

- Beispiel:

```
Add ( Num ( a ), b) -> [: "Add(" { printf("%d", a->value); } ", " b ];
```

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

4. unparse

- „views“
  - Zusätzliche Bedingung an Regel: wird ausgeführt, wenn Muster passt und aktueller „view“ in der „view“ Liste steht
    - Vordefiniert: „base_uview“, „base_rview“
    - Sind implizit in leerer „view“ Liste enthalten

```
%uview <unparse view liste>
%rview <rewrite view liste>
```

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

4. unparse

- Angabe des „view“ in unparse Regeln
  - Hinter der Variablen
    - benutzt angegebenen „view“ für das „unparsen“ dieses Terms (gesamter Teilbaum)
  - Bsp.:

```
%uview infix, postfix;
Add ( a, b ) -> [infix: a "+" b ],[postfix: a b "+" ];
```

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

4. rewrite

- Baumumformung entsprechend der Regeln
- Linke Seite: Muster
- Rechte Seite: einzusetzender neuer Term

```
Muster-> <opt_view: neuer_Term>;
```

- Neuer Term muss gleiches Phylum besitzen
- Rechts weiterhin möglich:
  - Operatoren
  - Term-Variablen
  - Funktionen

4. rewrite

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. **unparse & rewrite**
5. <minako>
6. <ramcy>
7. Quellen
8. Fragen

- Beispiel:

```
%rview shift, resolve;

Mul ( Num ( a ), Add ( b, c) ) ->
<resolve: Add ( Mul( a, b ), Mul( a, c ) ) >,
<shift: Mul ( Num ( c ), Add ( a, b ) ) >;
```

5. minako

minako

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. **<minako>**
6. <ramcy>
7. Quellen
8. Fragen

5. minako

- Eingabe:
  - flex und bison
  - bisons semantische Aktionen konstruieren Phyla
- Ausgabe
  - AST des Quellprogramms
  - Darstellung mit Hilfe der default print() Funktion von Kimwitu++

6. ramcy

ramcy

a RAM program compiler

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. **<ramcy>**
7. Quellen
8. Fragen

6. ramcy

- Compiler: RAM Programm -> C Quellcode
- RAM (random access machine)
- Eine 0-Adress Maschine mit unendlich vielen Registern
- Siehe VL Theoretische Informatik III (Dr. Till Nierhoff)

6. ramcy

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. **<ramcy>**
7. Quellen
8. Fragen

- Grammatik:

```
ramprogram = BEGIN { line }* END;

line = { LOAD | STORE | ADD | SUB | MUL | DIV } { direct_Index |
         indirect_Index | constant };

line = { GOTO line_number } | IF AKKU { = | < | > | <= | >= } int
         GOTO line_number;

direct_index = int;

indirect_index = „*“ int;

constant = „#“ int;

line_number = int;
```

7. Quellen

Quellen

1. Kurze Geschichte
2. AST (Wdhl.)
3. Syntax
4. unparse & rewrite
5. <minako>
6. <ramcy>
7. **Quellen**
8. Fragen

7. Quellen

- http://site.informatik.hu-berlin.de/kimwitu++/
- Michael Piefel
  - Sprechzeiten: Montag 11:00-11:15 Uhr, Freitag 13:00-13:15 Uhr
  - meist in Raum 3.3.08, Tel. 20 93-38 31
  - http://informatik.hu-berlin.de/~piefel
  - piefel@informatik.hu-berlin.de

8. Fragen