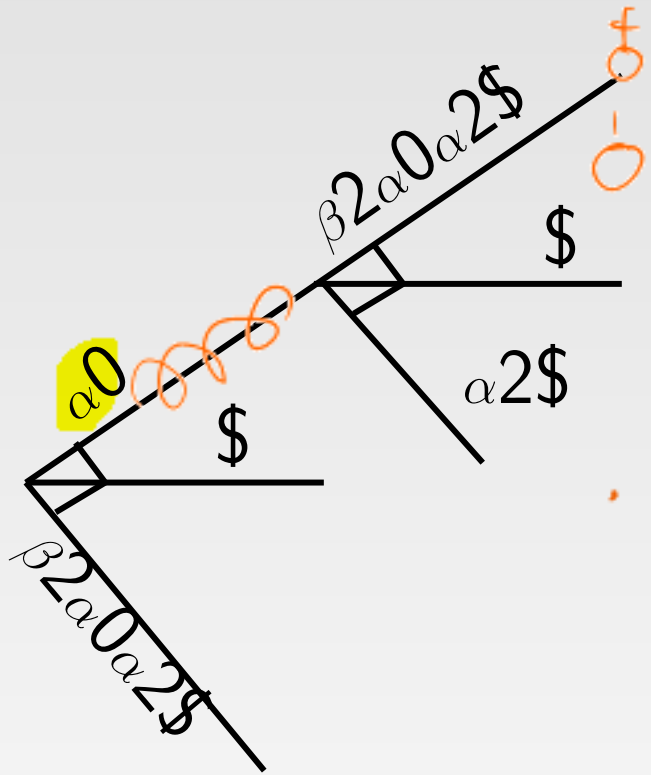
# P-Suffix-Baum

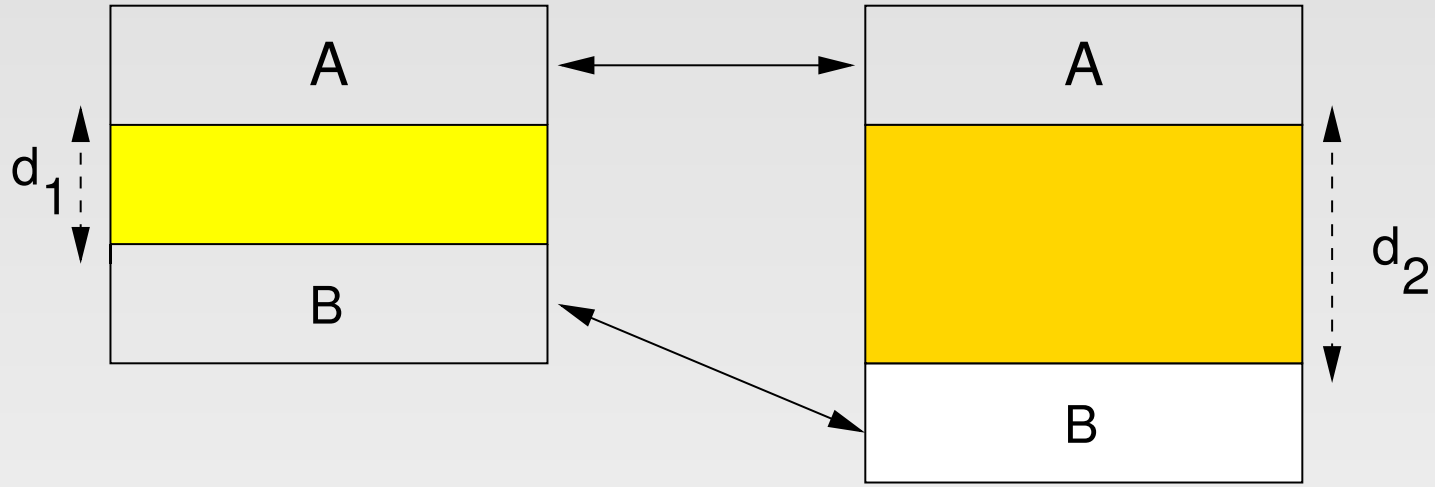
- Sei $S_i = s_i s_{i+1} \ldots s_n\$$ das $i$'te Suffix von $S$ ($ ist das Endezeichen). Der **P-Suffix-Baum** von $S$ enthält alle $prev(S_i)$ zu den Suffixen des P-Strings $S$.
  Beispiel: $S = \alpha y \beta y \alpha x \alpha x$

  $prev(S_1) = \alpha 0 \beta 2 \alpha 0 \alpha 2\$$
  $prev(S_2) = 0 \beta 2 \alpha 0 \alpha 2\$$
  $prev(S_3) = \beta 0 \alpha 0 \alpha 2\$$
  $prev(S_4) = 0 \alpha 0 \alpha 2\$$
  $prev(S_5) = \alpha 0 \alpha 2\$$
  $prev(S_6) = 0 \alpha 2\$$
  $prev(S_7) = \alpha 0\$$
  $prev(S_8) = 0\$$
  $prev(S_9) = \$$



# Typ-3-Erkennung

- Gefundene P-Matches werden, wenn möglich, zusammengefasst:




- Modi:
  - nur wenn $d_1 = d_2$
  - wenn $\max(d_1, d_2) \leq \Theta$

# Bewertung des Ansatzes von Baker

- Bakers Klonerkennung ist lexem-basiert und damit sehr schnell:
  - lineare Zeit- und Speicherkomplexität
  - Plattform: SGI IRIX 4.1, 40 Mhz R3000, 256 MB Hauptspeicher
  - System: 1,1 Mio LOC, mindestens zusammenhängende 30 LOC/Klon
  - nur 7 Minuten Analysezeit
- Der Ansatz ist invariant gegen Einfügung von Leerzeichen, Leerzeilen und Kommentaren.
- Der Ansatz ist weitgehend programmiersprachen-unabhängig (nur Regeln für Bezeichner/Tokens notwendig).

# Bewertung des Ansatzes von Baker

Allerdings entgehen dem Ansatz:

- äquivalente Ausdrücke mit kommutativen Operatoren:

| | |
|---|---|
| x = x + y | x = y + x |

- gleiche Anweisungsfolgen, die verschieden umgebrochen wurden:

| | |
|---|---|
| if (a > 1) {x = 1;} | if (a > 1)<br>{x = 1;} |

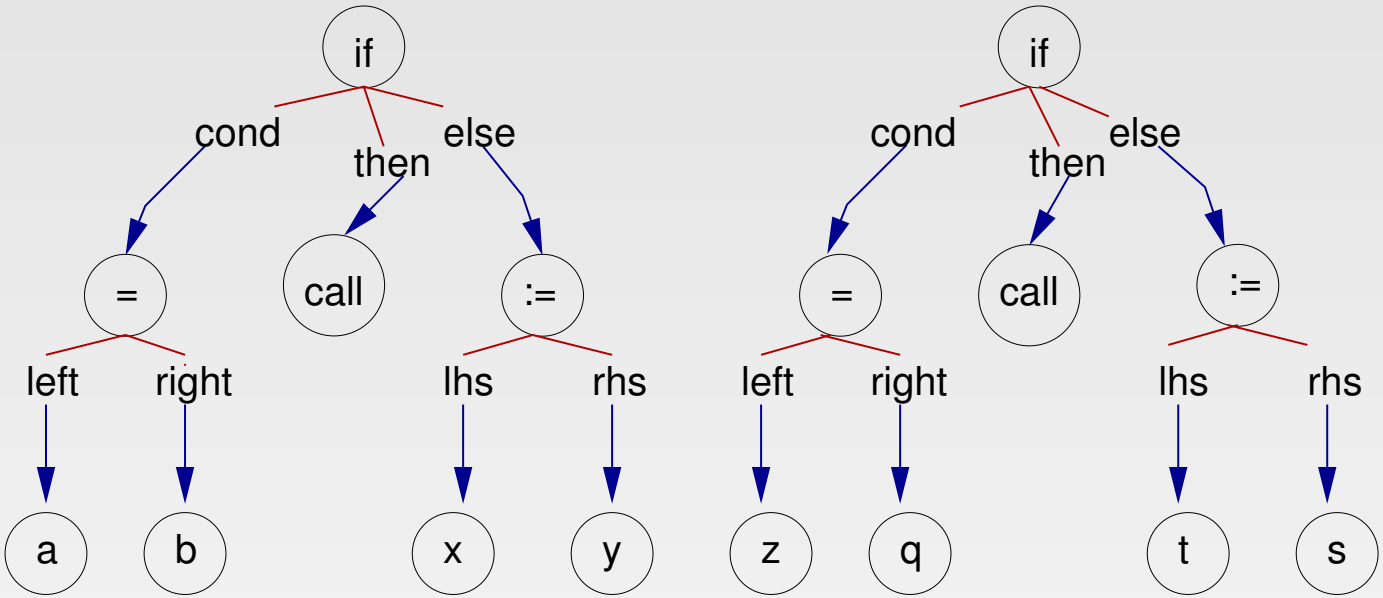
- gleiche Teilausdrücke:

| | |
|---|---|
| if (sp > 0) | if (sp > 0 && s[sp] != a) |

# AST-Matching

Ansatz basierend auf abstrakten Syntaxbäumen (ASTs):

- vergleiche jeden Teilbaum mit jedem anderen Teilbaum auf Gleichheit



# Verfahren nach Baxter u. a. (1998)

- Verfahren basiert auf ASTs
- Vermeidung des quadratischen Aufwands:
  - Partitionierung der zu vergleichenden Bäumen.
- Abstraktion von Bezeichnern:
  - Partitionierung und AST-Vergleich ignoriert Bezeichner.
- Typ-3-Klone:
  - Vergleich auf Ähnlichkeit statt Gleichheit.
  - Separater Schritt am Ende.

# Skalierungsproblem und Erkennung der Klontypen

- Bäume werden durch Hash-Funktion partitioniert.
- Nur Bäume innerhalb einer gemeinsamen Partition werden verglichen.
- Hash-Funktion gleich wie bei Erkennung von gemeinsamen Teilausdrücken durch optimierende Compiler:
  $h : \mathit{nodetype} \times \mathit{arg}_1 \times \mathit{arg}_2 \times \ldots \times \mathit{arg}_n \rightarrow \mathit{integer}$
  - Typ-1-Klonerkennung: h liefert genaue Partitionierung
  - Typ-2/3-Klonerkennung: h ignoriert Bezeichner

# Skalierungsproblem und Erkennung der Klontypen

- Bäume werden auf Ähnlichkeit verglichen (nicht auf Gleichheit):

$$Similarity(T_1, T_2) = \frac{2 \cdot Same(T_1, T_2)}{2 \cdot Same(T_1, T_2) + Difference(T_1, T_2)}$$

- Similarity muss über benutzerdefiniertem Schwellwert $\Theta_S$ liegen: $Similarity(T_1, T_2) \geq \Theta_S$
- Nur Bäume $T$ mit $mass(T) \geq \Theta_m$ werden betrachtet ($mass$ = Anzahl von Knoten, $\Theta_m$ = benutzerdefinierter Schwellwert).
- Bei kommutativen Operatoren werden wechselseitig beide Teilbäume verglichen.

# Probleme (und Lösungen) des AST-Matchings

- Skalierung
  - Vermeidung unnötiger Vergleiche
- Nicht nur Typ-1-Klone sollen erkannt werden.
  - Abstraktion von Bezeichnern.
  - Die Teilausdrücke müssen nicht gleich sein, sondern nur ähnlich genug.
- Klone sind Teil anderer Klone (geschachtelte Konstrukte).
  - Klone, die Teil eines anderen Klons sind, werden ignoriert, d.h. zusammengesetzte Klone subsumieren ihre Teile.

# Basis-AST-Matching (Schritt 1)

```
Clones := ∅;
for each subtree t loop
    if mass (t) ≥ Θ_m then
        hash t to bucket;
    end if;
end loop;
for each subtree s and t in the same bucket loop
    if Similarity (s, t) ≥ Θ_S then
        for each subtree s' of s and for each subtree t' of t
            and Is_Member (s', t', Clones)
        loop
            Remove_Clone_Pair (s', t', Clones);
        end loop;
        Add_Clone_Pair (s, t, Clones);
    end if;
end loop;
```

# AST-Matching von Sequenzen (Schritt 2)

- Sequenzen von Bäumen (z.B. Anweisungsfolgen):

```
void f(void)
{
  x = 0;
  a = a + 1;
  b = b + 2;
  c = c + 3;
  w = g();
}
```

```
void g(void)
{
  y = 2 * x;
  a = a + 1;
  b = b + 2;
  c = c + 3;
  i = h() * 3;
}
```

- Basis-Matching-Algorithmus identifiziert nur die gleichen Zuweisungen, ignoriert aber Gleichheit der beiden Anweisungsteilfolgen als Ganzes.
- Nächster Schritt identifiziert maximal lange gemeinsame Teilfolgen zweier Sequenzen, mit einer benutzerdefinierten Mindestlänge.

# Erkennung von Klonsequenzen (Schritt 2)

```
for k in Min_Length .. Max_Length loop
   place all subsequences of length k into buckets;
   for each subsequence i and j in same bucket loop
      if Compare_Sequence (i, j, k) ≥ Θ_S then
         Remove_Sequence_Subclones_Of (i, j, k, Clones);
         Add_Sequence_Clone_Pair (i, j, k, Clones);
      end if;
   end loop;
end loop;
```

# Erkennung zusammengesetzter Klone (Schritt 3)

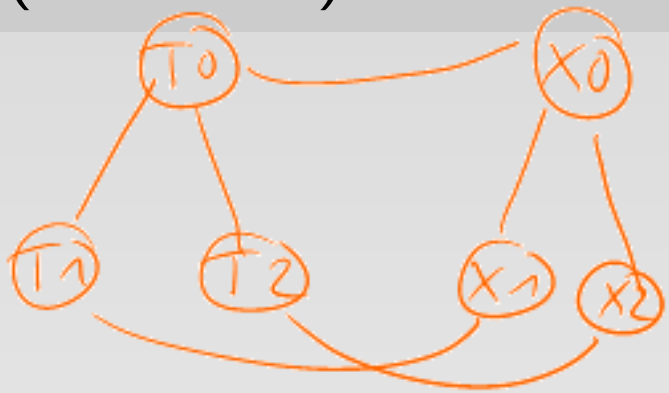
- Gemeinsame Anweisungsfolgen wurden möglicherweise erst im zweiten Schritt nach dem Basis-AST-Matching erkannt.
- Durch neu erkannte Klone könnten Konstrukte, die diese Klone umfassen, nunmehr doch ähnlich sein.

```
s = 0; x = y;
while (x > 0) {
    s = s + x;
    x = x - 1;
    ...
}
```

```
p = 0; a = b;
while (a > 0) {
    p = p * x;
    a = a - 1;
    ...
}
```

- D.h. Vaterknoten der als Klone erkannten Teilbäume müssen noch einmal evaluiert werden.

# Erkennung zusammengesetzter Klone (Schritt 3)




```
Clones_To_Generalize := Clones;
while Clones_To_Generalize ≠ ∅ loop
   Remove_Clone_Pair (s, t, Clones_To_Generalize);
   if Compare_Clones (Parent (s), Parent (t)) ≥ Θ_S then
      Remove_Clone_Pair (s, t, Clones);
      Add_Clone_Pair (Parent (s), Parent (t), Clones);
      Add_Clone_Pair (Parent (s), Parent (t),
                      Clones_To_Generalize);
   end if;
end loop;
```

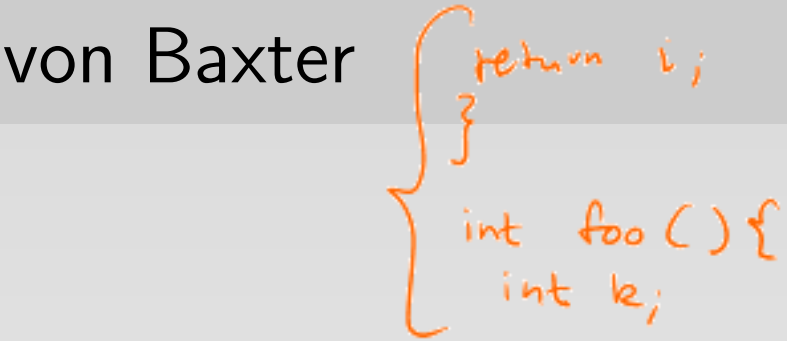
# Bewertung des Ansatzes von Baxter

- AST-Matching ist syntax-orientiert und deshalb aufwändiger als lexem-basierter Ansatz, da Parsing notwendig ist
  - beim lexem-basierten Ansatz müssen nur Schlüsselworte und Trennzeichen erkannt werden
  - was aber bei manchen Programmiersprachen eben doch Parsing voraussetzt, z.B. PL/1:

# Bewertung des Ansatzes von Baxter

- AST-Matching ist syntax-orientiert und deshalb aufwändiger als lexem-basierter Ansatz, da Parsing notwendig ist
  - beim lexem-basierten Ansatz müssen nur Schlüsselworte und Trennzeichen erkannt werden
  - was aber bei manchen Programmiersprachen eben doch Parsing voraussetzt, z.B. PL/1:
    IF IF = ELSE THEN ELSE := THEN ELSE IF := ELSE

# Bewertung des Ansatzes von Baxter




- AST-Matching ist syntax-orientiert und deshalb aufwändiger als lexem-basierter Ansatz, da Parsing notwendig ist
  - beim lexem-basierten Ansatz müssen nur Schlüsselworte und Trennzeichen erkannt werden
  - was aber bei manchen Programmiersprachen eben doch Parsing voraussetzt, z.B. PL/1:
    IF IF = ELSE THEN ELSE := THEN ELSE IF := ELSE
- dafür aber genauer:
  - kommutative Operatoren werden berücksichtigt
  - syntaktische Einheiten werden verglichen, statt einzelner Code-Zeilen
  - übereinstimmende Teilausdrücke werden erkannt

# Verfahren nach Mayrand u. a. (1996)

- Verfahren basiert auf Metriken des Codes.
- Vermeidung des quadratischen Aufwands:
  - im Prinzip immer noch quadratisch, Vergleich ist aber relativ billig.
- Abstraktion von Bezeichnern:
  - Bezeichner werden von Metriken ignoriert.
- Typ-3-Klone:
  - durch Toleranz der Metriken bzw. ihrer Zusammenfassung.

# Vergleich auf Basis von Kennzahlen

- Hoffnung:
  $Code_1 = Code_2 \quad \Leftrightarrow \quad Kennzahlen(Code_1) = Kennzahlen(Code_2)$
- Granularität üblicherweise Funktionsebene, da hierfür viele Metriken existieren
- Aspekte nach Mayrand u. a. (1996):
  - Namen
  - Layout
  - Anweisungen
  - Kontrollfluss

# Vergleichsmetriken (Mayrand u. a. 1996)

- Name
  - relative Anzahl gemeinsamer Zeichen
- Layout
  - Anzahl Zeichen von Kommentaren (Deklarationsteil, Implementierungsteil)
  - Anzahl mehrzeiliger Kommentare
  - Anzahl nicht-leerer Zeilen (inklusive Kommentare)
  - durchschnittliche Länge der Bezeichner

# Vergleichsmetriken (Mayrand u. a. 1996)

- Anweisungen
  - gesamte Anzahl von Funktionsaufrufen
  - Anzahl verschiedener Aufrufe
  - durchschnittliche Komplexität der Entscheidungen in der Funktion
  - Anzahl der Deklarationen
  - Anzahl ausführbarer Anweisungen
- Kontrollfluss
  - Anzahl der Kanten im Kontrollflussgraphen (KFG)
  - Anzahl der Knoten im KFG
  - Anzahl der Bedingungen im KFG
  - Anzahl der Pfade im KFG
  - Komplexitätsmetrik über dem KFG

# Zusammenfassung der Metriken

- Zwei Funktionen $f_1$ und $f_2$ sind in Bezug auf einen Aspekt:

| | |
|---|---|
| gleich: | gleiche Metrikwerte |
| ähnlich: | alle Metrikwerte liegen in einer gewissen Bandbreite (spezifisch für jede individuelle Kennzahl definiert), sind aber nicht gleich |
| verschieden: | mindestens ein Metrikwert liegt außerhalb einer Bandbreite |

# Klassifikation

| | |
|---|---|
| Exakte Kopie: | Funktionen sind in jedem Aspekt gleich (Typ-1-Klon) |
| Ähnliches Layout: | Ähnliches Layout und ähnliche Namen, gleiche Anweisungen und Kontrollfluss ($\approx$ Typ-2-Klon) |
| Ähnliche Ausdrücke: | Name und Layout sind verschieden, Anweisungen und Kontrollfluss sind gleich ($\approx$ Typ-2-Klon) |
| Verschieden: | Alle Aspekte sind verschieden. |

# Bewertung des Ansatzes von Mayrand u. a. (1996)

- Aspekte sind nicht unabhängig.
- Definition der Bandbreite ist notwendig.
- (Klassifikation ist unvollständig.)
- Präzision:
  - $Code_1 = Code_2 \quad \Rightarrow \quad Kennzahlen(Code_1) = Kennzahlen(Code_2)$ $\surd$
  - $Code_1 \approx Code_2 \quad \Rightarrow \quad Kennzahlen(Code_1) \approx Kennzahlen(Code_2)$ $\surd$
  - $Kennzahlen(Code_1) = Kennzahlen(Code_2) \quad \Rightarrow \quad Code_1 = Code_2$ ?
  - $Kennzahlen(Code_1) \approx Kennzahlen(Code_2) \quad \Rightarrow \quad Code_1 \approx Code_2$ ???