



**GTUG-Tagung  
am 19. und 20. Juni 2006 in Kiel**

**Ein Translator ist kein Compiler:  
Unterschiede zwischen Compilierung und  
Sprachkonvertierung am Beispiel von TAL**

Uwe Erdmenger

**pro et con** Innovative  
Informatikanwendungen GmbH



# Die Firma pro et con

- Gründung 1994, z.Z. 11 Mitarbeiter, Unternehmensstandort Chemnitz
- Reverse Engineering:
  - Flow Graph Manipulator FGM  
(Cobol, ScreenCobol, TAL, JAVA, Natural, SPL, PL/I,...)
- Software-Evolution, -Modernisierung, -Migration
  - MigMan für HP NonStop, BS2000,..



# Translatoren

- Amadeus Germany:
  - SPL to C++
- Cosmos Versicherung, msg München:
  - PL/I to C++
- Heidelberger Druckmaschinen,  
MAN Nutzfahrzeuge:
  - TAL to C/C++

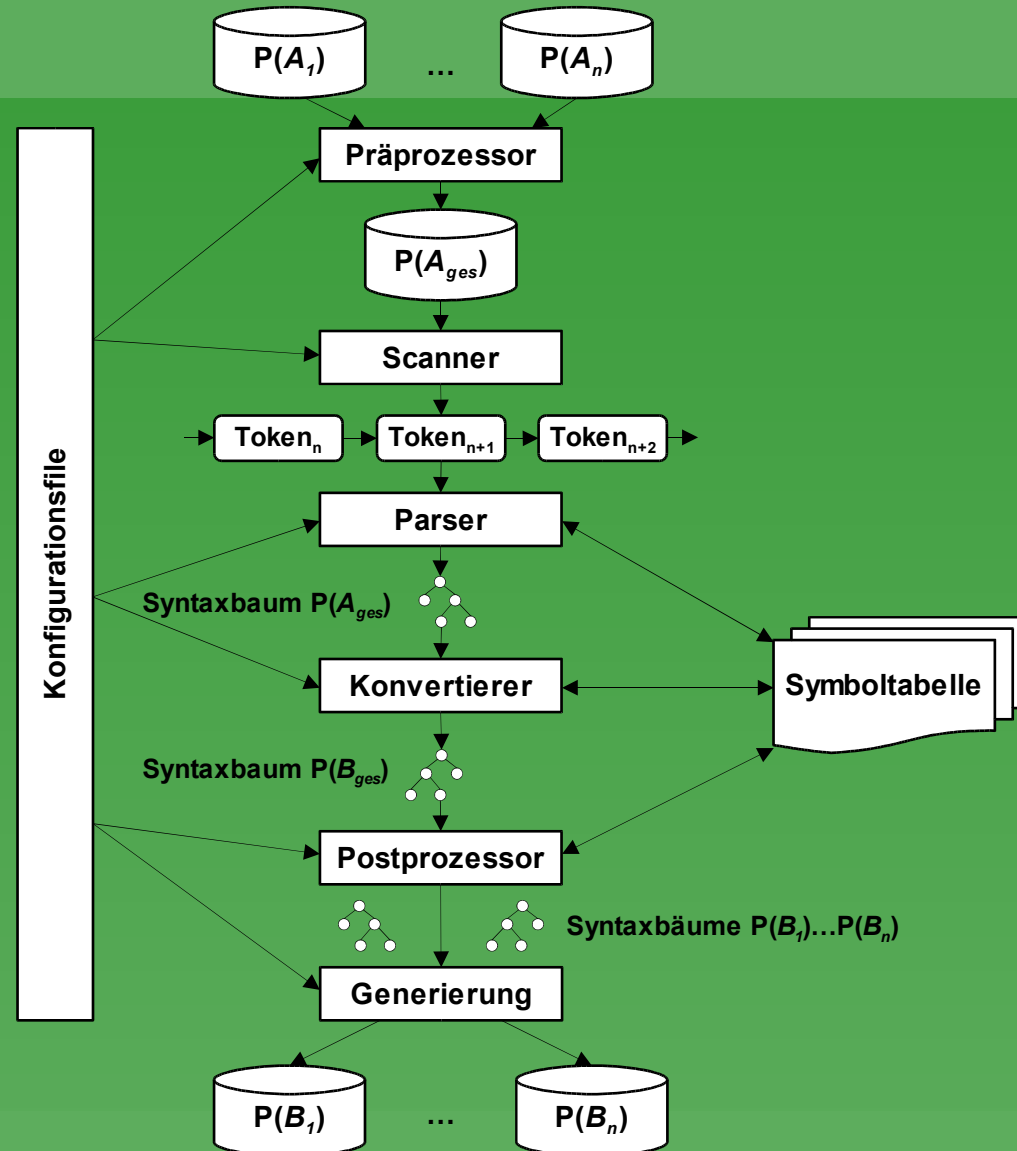


# Definition Translator

- Translator – "Übersetzer"  
Hier: Tool zur automatischen Konvertierung von Programmen aus einer Programmier-Hochsprache in eine andere
- Wesentliches Ziel: Wartung/Weiterentwicklung der Programme in der neuen Sprache
- Gemeinsamkeiten/Unterschiede zu Compilern
- Beispiel: TAL to C/C++-Translator (TTC)



# Das Translatormodell





# Unterschiede zu Compilern

- Frontend:
  - Erhaltung von Kommentaren
  - Erhaltung von Präprozessor-Informationen (z.B. `?SOURCE`)
  
- Generierung:
  - Wiedereinbau der Kommentare
  - Wiederaufteilung auf verschiedene Files
  - Wiedereinfügen von `DEFINES`
  - Formatierung einstellbar
  
- Interne Darstellung:
  - Grundsätzlich alle Informationen aufheben
  - Zwei vollständige Syntaxbäume



# Interne Darstellung: Tokenliste

- Alle Informationen zu Token müssen erhalten bleiben (im Gegensatz zu Compilern):
  - Tokentyp und -text
  - Positionsinformationen (File, Zeile, Spalte)
  - Flags ("Präprozessortoken", "Kommentartoken", ...)
  - Informationen zur DEFINE-Ersetzung
  - ...
- Speicherung in sogenannter Token-Liste in der Reihenfolge ihres Auftretens



# Interne Darstellung: Syntaxbaum

- Entscheidung:  
Klassenhierarchie oder einheitlicher Knotentyp?
- Lösung:  
Klassenhierarchie, aber Informationen zu  
Kindknoten sowie Knotenart alle in der  
Basisklasse
- Definition der Knotenarten in Metasprache  
→ Konvertierung mittels Perl-Programm in  
C++-Code



# Beispiel: Syntaxbaum

```
CLASS TAL_PLUS_NODE EXTENDS TAL_EXPR_NODE
  CHILD summand1 TYPE TAL_EXPR_SET
  CHILD summand2 TYPE TAL_EXPR_SET
  SPECIAL
END-CLASS TAL_PLUS_NODE
```

```
CLASS TAL_IDENT_NODE EXTENDS TAL_PRIMARY_NODE
  SPECIAL
  // speziell fuer diese Knotenart:
  class TAL_Symbol *sym;           // Symboltabelle
  class Tal_base_node *model;     // MODEL-Deklaration
  INIT
  sym = 0;
  model = 0;
END-CLASS TAL_IDENT_NODE
```



# Postprozessor

- Nach Konvertierung (s. Translatormodell) liegt ein C/C++-Syntaxbaum vor
- Notwendig sind:
  - Zerlegung in mehrere Bäume (einer für jedes File)
  - Substitution von Teilbäumen durch andere, die z.B. DEFINE-Aufrufe repräsentieren
  - Zuordnung der Kommentare zu einzelnen Teilbäumen
- Ergebnis:  
Mehrere, interne Syntaxbäume, die ohne weitere Manipulation in die einzelnen Files ausgegeben werden



# Aufteilung des generierten Codes

- Teilbäume, die aus einem Includefile stammen, aus dem "main-Baum" herausschneiden
- Ersetzen durch Bäume, die für eine Include-Anweisung stehen
- Informationen zur Herkunft der Statements werden bei der Konvertierung mit übertragen
- Beschränkung auf komplette Anweisungen bzw. Definitionen/Deklarationen
- Problem: Partielles Einziehen von Files (SECTION)



# Beispiel

## TAL-Main-File

```
?SOURCE BSP(SECT1)
...
?SOURCE BSP(SECT2)
```

## C-Main-File

```
#define SECT1
#include "bsp.h"
#undef SECT1
...
#define SECT2
#include "bsp.h"
#undef SECT2
```

## TAL-SOURCE-File

```
?SECTION SECT1
PROC P1;

?SECTION SECT2
PROC P2;
```

## C-Includefile

```
#ifdef SECT1
    void p1();
#endif

#ifdef SECT2
    void p2();
#endif
```



# Erhaltung von Kommentaren (1)

- Kommentare werden Bestandteil der Token-Liste
- Parser überliest diese Kommentartoken
- Heuristik:
  - Kommentar wird einem Teilbaum als "Vorkommentar" zugeordnet, wenn er direkt vor dem ersten, zum Teilbaum gehörenden Token steht (ohne syntaktisch relevante Token dazwischen)
  - "Nachkommentar" analog



## Erhaltung von Kommentaren (2)

- Entscheidung:  
Ist es Nachkommentar zum vorherigen Statement oder Vorkommentar zum folgenden  
→ Heuristik
- Nur für bestimmte Teilbäume (komplette Statements)
- Übertragung an C/C++-Bäume bei der Konvertierung
- Ausgabe bei der Generierung



# Wiedereinfügen von DEFINEs (1)

- Besonderheit bei TAL: DEFINEs haben Gültigkeitsbereiche (Blockstruktur)
- Entsprechendes `#undef` am Blockende nötig

```
PROC P1;  
BEGIN  
    INT i;  
    DEFINE val(a,b) = (a '<<' 12) LOR b;  
    ...  
    i := val(7,3);  
    ...  
END
```



## Wiedereinfügen von DEFINEs (2)

- Ohne Rücktransformation entstehen viele expandierte Varianten des Ausdrucks  
→ schwer wartbar
- Wiedereinfügung von Makros im Postprozessor:

```
void p1 ()
{
    int i;
#define VAL(A,B) ((A)<<12) | (B)
    ...
    i = VAL(7,3);
    ...
#undef VAL
}
```



## Wiedereinfügen von DEFINEs (3)

- Makro-Namen bei verschachtelten Blöcken müssen eindeutig sein – ggf. Umbenennung durch den Translator
- Problem: bei mehrfachem Vorkommen des Makroaufrufs im Block → sicherstellen, daß es immer zum gleichen C/C++-Code wird



# Konvertierung von Strings

- In TAL existiert der Datentyp `STRING` – meist als Array: `STRING s[0:5] = "ABCDEF"`
- Naheliegende Konvertierung als C-String funktioniert nicht (0-Zeichen)
- Vergrößern des Speicherbereichs um 1 Byte ist semantisch nicht äquivalent (Überlagerungen)
- Lösung in C++: Template-Klasse `STRING<n>`  
`n` als Template-Parameter benötigt keinen Speicherplatz
- Operatoren bzw. Template-Funktionen für die gängigen Stringoperationen entwickeln



# Überlagerungen in Strukturen

```
STRUCT STRUKT1 (*);  
BEGIN  
    STRING s[0:3];  
    INT(32) i = s;  
END;
```

- Naheliegende Konvertierung als `union` in C++ nicht möglich, da diese keine Objekte mit Konstruktor bzw. Zuweisungsoperator zulassen
- Lösung: Realisierung als Methoden, die Referenzen zurückgeben



# Konvertierte Überlagerung

```
struct {  
    STRING<4> s;  
    long &i () {  
        return *((long*)&s);  
    }  
} strukt1;
```

- Bezugnahme darauf:

```
strukt1.i() = 12345;
```



## Fazit

- Translatorarchitektur besitzt wesentliche Komponenten eines Compilers
- Erweiterungen in der Funktionalität
- Besonderen Aufwand verursacht die geforderte Wartbarkeit des Zielcodes
- Grundregel für die internen Strukturen: alles aufheben
- Manche Konvertierungen erfordern ungewöhnliche Lösungen (s. Überlagerungen)
- Zeitraum: 1,5 Jahre, Aufwand: 3 Mannjahre



# Kontakt

**pro et con** Innovative  
Informatikanwendungen GmbH

Annaberger Str. 240  
09125 Chemnitz

Tel. +49 371 5 347 353  
Fax. +49 371 5 347 345  
email: [proetcon@proetcon.de](mailto:proetcon@proetcon.de)  
<http://www.proetcon.de>