1 SPL-Sprachkonvertierung im Rahmen einer BS2000 Migration

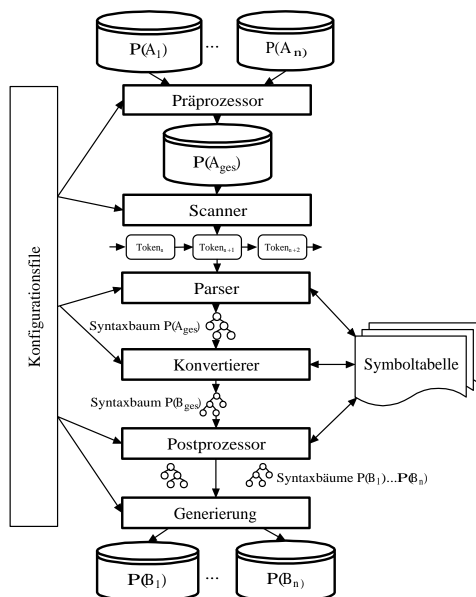
Uwe Erdmenger
pro et con Innovative Informatikanwendungen GmbH, Annaberger Straße 240, 09125 Chemnitz
Uwe.Erdmenger@proetcon.de

In bestimmten Migrationsprojekten existiert die Notwendigkeit für einen Wechsel der Programmiersprache. Dazu werden sogenannte Translatoren eingesetzt. Translatoren konvertieren in Analogie zu einem Compiler ein Programm einer Quellprogrammiersprache in ein Programm einer Zielprogrammiersprache.

Der folgende Beitrag diskutiert die erweiterten Anforderungen an Translatoren gegenüber klassischen Compilern am Beispiel eines „SPL to C++ Translators“ (STC). STC konvertiert die unter dem Betriebssystem BS2000 implementierte Programmiersprache SPL nach C++. STC wurde im Rahmen eines Migrationsprojektes der Firma Amadeus Germany von der Firma pro et con entwickelt.

1.1 Das Translatormodell

Das Modell von Translatoren, welche Quellprogramme einer Hochsprache A in Zielprogramme einer anderen Hochsprache B konvertieren, orientiert sich am klassischen Compilermodell. Es existieren jedoch Unterschiede. Diese Unterschiede sollen am Modell des STC expliziert werden. Die folgende Abbildung beschreibt das Translatormodell:



Gegenüber dem klassischen Compilermodell existieren die folgenden Unterschiede:

Kommentarerhaltung: Präprozessor und Scanner dürfen Kommentare nicht löschen, deren Informationen sind zu erhalten, damit sie optional in den späteren Zielcode eingefügt werden können.

Speicherung von Präprozessorinformationen: Genau wie beim klassischen Compiler müssen Präprozessorbefehle vor dem Scannen ausgeführt werden. Auch deren Informationen sind im Konvertierungsprozeß zu erhalten. Quellprogramm-Makros z.B. müssen optional als Makros im Zielprogramm auftauchen und die Include-Befehle dienen in der Phase Generierung der Aufteilung des Zielprogramms auf verschiedene Files. Aufgrund der erweiterten Funktionalität ist es nicht möglich, Präprozessoren, Scanner etc. von existierenden Compilern im Rahmen einer Translatorentwicklung als Bestandteile zu integrieren, sie sind prinzipiell neu zu entwickeln.

Schnittstelle zwischen Quell- und Zieldarstellung: Es existiert eine strikte Trennung. Der Parser liefert einen attributierten Syntaxbaum des Quellprogrammes, in diesem speziellen Fall SPL. Die Phase Konvertierung realisiert daraus einen attributierten Syntaxbaum des Zielprogrammes, in diesem speziellen Fall C++.

Postprozessor: Diese Komponente ist neu gegenüber dem klassischen Compilermodell. Eine wesentliche Aufgabe des Präprozessors ist die Zerteilung des vollständigen, attributierten Syntaxbaumes des Zielprogrammes in Teilsyntaxbäume. Details dazu diskutiert ein spezieller Abschnitt.

Generierung: Die Phase Generierung schreibt physisch Sourcecode durch Traversieren einzelner Syntax(teil)bäume. Es entstehen daraus beim STC mehrere .h-Files und ein main-Programm. Da die Forderung nach Wartbarkeit existiert, Wartbarkeit aber völlig unterschiedlich interpretiert wird, kann die formatierte Ausgabe entsprechend den Vorstellungen der Nutzer flexibel eingestellt werden.

Die folgenden Abschnitte diskutieren ausgewählte Aspekte der Arbeitsweise eines Translators.

1.2 Postprozessor

Die Aufgabe eines Präprozessors im klassischen Compiler ist es, Präprozessoranweisungen auszuführen und neuen Quellcode zu erzeugen. Das Ergebnis des Präprozessorlaufs ist ein Quellcodefile ohne Präprozessoranweisungen. Für den Translationsprozeß existieren in Präprozessoranweisungen wesentliche Informationen, wie z.B., aus welchen Includefiles sich das Quellprogramm zusammensetzt, einschließlich der Positionsangaben, welche Makros an welcher Position in den Includefiles auftreten und welche Kommentare an welcher Position vorkommen. Im Gegensatz zum klassischen Compiler müssen diese Informationen protokolliert werden. Diese Informationen werden im Translator zu Token im Tokenstrom. Sie werden beim Parsen an den Syntaxbaum des Quellprogrammes und bei der Konvertierung an den Syntaxbaum des Zielprogrammes notiert und bilden wesentliche Informationen für den Postprozessor. Dessen Aufgaben sind:

Zerteilung des Zielsyntaxbaumes. Dabei entsteht für jedes originale Includefile ein eigener Teil-Syntaxbaum mit semantisch äquivalentem Inhalt. Dazu sind aus dem gesamten Zielcodebaum Teilbäume für die Includefiles ab-

zuspalten und an deren Stelle im Zielcodebaum Syntaxteilbäume für Include-Anweisungen der Zielsprache einzufügen. Es existieren Fälle, bei denen im Zielcode ursprünglich im Quellcode zusammenstehende Sourcecodeteile getrennt generiert werden. Beispiel:

```
... (SPL-Main-File) ...    ... (C++-Main-File) ...
 DCL 1 PERSON,              struct {
       %INCLUDE BSP;        #define BSP_SECT1
                            #include "bsp.hpp"
                            #undef BSP_SECT1
                            } person = { TFixString<10>(
                            #define BSP_SECT2
                            #include "bsp.hpp"
                            #undef BSP_SECT2
                            ) };
... (SPL-Includefile) ...   ... (C++-Includefile) ...
  2 NAME CHAR(10)           #ifdef BSP_SECT1
    INIT('FRITZ');          TFixString<10>  name;
                            #endif
                            #ifdef BSP_SECT2
                            "FRITZ";
                            #endif
```

Da Definition von *name* und Initialisierung getrennt werden, zerfällt das Includefile in zwei unterschiedliche Sektionen.

Im Zusammenhang mit der Zerteilung ist ein weiteres Problem zu lösen: Da originale Includefiles in mehreren Quellprogrammen und dort an unterschiedlichen Positionen eingezogen werden, können beim Konvertieren mit dem STC mehrere, nicht identische Versionen im Zielcode entstehen. Die Ursache liegt im unterschiedlichen Kontext der umliegenden Quellprogramme, z.B., wenn ein originales Include ein initialisiertes Datenelement enthält und einmal als Strukturkomponente und an anderer Stelle als globale Variable eingezogen wird. Für die Includedopplung auf der Zielprogrammebene existieren unterschiedliche Lösungen. Tritt der Fall selten auf, so bietet sich eine Änderung des Quellprogrammes vor der eigentlichen Konvertierung an. Es existiert als Alternative die Möglichkeit, mehrere Includefiles auf Zielebene zu mischen. Dabei sind Sektionen zu vergleichen und ggf. umzubenennen (auch im Aufruf).

Platzierung von Makros. Im Zielsyntaxbaum sind alle C++ Teilbäume, welche aus SPL-Makros hervorgegangen sind, entsprechend anotiert. Der Postprozessor vergleicht diese Teilbäume, definiert ein C-Makro und ersetzt die Syntaxbäume durch Syntaxbäume der entsprechenden Makroaufrufe. Der Teilsyntaxbaum des Makros wird an die entsprechende Stelle des Syntaxbaumes eingefügt. Ist das nicht möglich, so wird eine Warnung ausgegeben.

Kommentarerhaltung. Die Kommentare werden als Vor- und Nachkommentar den einzelnen Teilbäumen der Zielsprache heuristisch zugeordnet. In der Phase Generierung werden sie dann vor oder nach dem entsprechenden Zielcodefragment platziert.

1.3 Interne Datenrepräsentation

Die in den einzelnen Konvertierungsphasen gewonnenen Informationen werden im STC hauptspeicherintern in C++-Objekten verwaltet. Die wesentlichen Datenstrukturen sind dabei die *Tokenliste* des Quellprogrammes und die *Syntaxbäume* von Quell- und Zielprogramm. Zu jedem Token werden die folgenden Informationen gespeichert:

- Start- und Endposition (File, Zeile, Spalte),
- Tokentyp und -text,
- spezielle Flags (z.B. „ist Kommentar“, „ist durch Substitution eingefügt“, „ist Präprozessortoken“, . . .).

Dabei werden in die Liste auch Token aufgenommen, die der Parser überliest, z.B. Kommentare und substituierte Quelltextteile. Alle Informationen bleiben bis zum Programmende erhalten und stehen nachfolgenden Phasen zur Verfügung.

Der Syntaxbaum besteht aus Knoten verschiedener Klassen. Diese werden in einer speziellen, deskriptiven Notation definiert, nachfolgend wird ein einfaches Beispiel angegeben:

```
/**************************************************
 * Bezeichnerknoten
 **************************************************/

CLASS SPL_IDENT_NODE EXTENDS SPL_PRIMARY_NODE
   SPECIAL
   // speziell fuer diese Knotenart:
   class SPL_Symbol *sym;       // Symboltabelle
   class Spl_base_node *model;  // MODEL-Deklaration
   INIT
   sym = 0;
   model = 0;
END-CLASS SPL_IDENT_NODE

/**************************************************
 * Addition
 **************************************************/

CLASS SPL_PLUS_NODE EXTENDS SPL_EXPR_NODE
   CHILD summand1 TYPE SPL_EXPR_SET
   CHILD summand2 TYPE SPL_EXPR_SET
   SPECIAL
END-CLASS SPL_PLUS_NODE

/**************************************************
 * Die Menge aller moeglichen Ausdruecke.
 **************************************************/
SET SPL_EXPR_SET = {SPL_PLUS_NODE, SPL_MINUS_NODE, ...}
```

Ein Perl-Programm generiert aus dieser formalen Notation den C++-Code zur Darstellung der Syntaxbäume. Dieses Verfahren automatisiert die Entwicklung des Baummodells für Quell- und Zielsprache und ist allgemein bei der Entwicklung von beliebigen Translatoren nutzbar.

1.4 Zusammenfassung

Die Ausführungen beschreiben kurz und exemplarisch Unterschiede in der Arbeitsweise von Translatoren und Compilern, die bei der Translatorentwicklung zu beachten sind. Sie beruhen auf Erfahrungen, welche die Firma pro et con bei der Entwicklung von nunmehr drei Translatoren für die Programmiersprachen PL/I, TAL und SPL gesammelt hat. In diesem Prozeß entstanden Werkzeuge (Meta-Tools), welche die Entwicklung weiterer Translatoren teilautomatiseren (Parsergenerator BTRACC, Generierungstool Cgen, Baummodell, Symboltabelle, Tokenliste). STC ordnet sich ein in einen Werkzeugkasten für komplette BS2000-Migrationen, welcher neben der SPL-Konvertierung auch die Datenmigration und die Konvertierung von JCL-Prozeduren nach Perl unterstützt.