Cross Language Transformations in TXL

TXL Working Paper 4
James R. Cordy
January 1995

Copyright 1995 James R. Cordy

This paper describes one paradigm for cross-language transformation using TXL, and points out other possibilities.

Basic Paradigm

The basic paradigm for cross-language transformation involves three steps: (1) create working grammars for both the target and the source language; (2) uniquely rename the nonterminals of the two grammars; (3) identify a minimal set of corresponding nonterminals to be used as transformation targets; (4) restructure and integrate the source and target grammars to form an integrated translation grammar; (5) build a set of independent translation rules for each corresponding nonterminal.

1. Create working grammars

Begin with a working TXL grammar for both the source and the target languages (independently). Often these can be obtained from the TXL grammar set or a third party. If it is necessary to create a new grammar, remember to use a reference (user-readable) grammar rather than an implementation (Yacc, etc.) grammar.

2. Uniquely rename grammar nonterminals

Because it will be necessary to work with both languages at once when translating, and because grammars for different languages frequently use the same nonterminal names, it is best to begin by uniformly renaming all nonterminals in each of the grammars. For example, if the source language is C and the target language is Pascal, the C nonterminals can be renamed using a "C_" prefix (e.g., [C_expression]), and the Pascal nonterminals can be renamed using a "P_" prefix (e.g., [P_expression]).

3. Identify corresponding nonterminals

Decide what the minimal set of media-of-exchange nonterminals to be transformed will be. For example, transformation rules to express the relation between C and Pascal needs only to happen at the levels of program, subprogram, declaration, statement and expression. All of the detailed rules for dealing with any individual special cases can be targeted at one of these.

It is important to minimize this set as much as possible - it is not necessary to have a large set of corresponding nonterminals. For example, to translate while statements between C and Pascal you need not have a rule aimed specifically at while statements; a rule targeted at [statement], but with a pattern that happens to be a while statement, will work identically.

The correspondence you are looking for is conceptual rather than strictly syntactic. It may be necessary to restructure the source and/or target grammars to some extent to better express their conceptual relation. Time spent on this step pays off big later, so do it with care!

4. Integrate source and target grammars

Build an integrated translation grammar for the media-of-exchange. The integrated grammar has one define for each medium-of-exchange, and it is of the exact form:

```
define expression
    [C_expression]
  | [P_expression]
end define
```

and so on. For each such definition, add redefines to attach the integrated form to the original grammars, like so:

```
redefine C_expression
    ...
  | [expression]
end redefine

redefine P_expression
    ...
  | [expression]
end redefine
```

This insures that both the untranslated and translated form of each medium-of-exchange is accepted in every context, allowing rules to work independently on partially translated components.

The target of the integrated grammar is of course [program], defined as:

```
define program
    [C_program]
  | [P_program]
end define
```

5. Build independent translation rules

Build a set of independent rules, each of which is targeted at one of the media-of-exchange and expresses the relation between exactly one possible pattern of the source language for that nonterminal and its reflection in the target language. Bundle the rules up into a single main function that invokes all of them. For example,

```
function main
    replace [program]
```

```
            P [program]
        by
            P [fix_while_statements]
              [fix_for_statements]
                .
                .
                .
    end function

    rule fix_while_statements
        replace [statement]
            'while E [expression] 'do {
                S [repeat statement]
            }
        by
            'while ( E ) 'do
                'begin
                    S
                'end ;
    end rule
```

6. Experience

This is the technique that has been used successfully at Queen's University for translations between C, Pascal and Turing, and at GMD Karlsruhe for translations between ISL, C++, Modula II and Ada. It definitely isn't always as easy as is outlined above, but it is a general strategy that can help guide your work.

At Queen's we do a lot of grammar programming in advance to make the rules easy - and we don't worry about how far off the real languages our grammar is except insofar as it expresses a superset of the real languages. Our experience teaches us that careful planning of the media-of-exchange and tinkering with the grammars to make similar forms parse similarly pays off big in the rule set. And we are *never* afraid to change the grammar if that is the easiest way to achieve a transform.

The tradeoff is the usual one - you can spend the time up front in the design stage, or be forced to spend it later making up for your lack of a careful design stage. Either way the time must be spent.

7. Other paradigms

Another paradigm that has proven very successful is the producer-consumer translation paradigm. In this paradigm the original program in the source language is graduate "consumed" (transformed to empty) as the translated program in the target language is "produced" (transformed from empty). This can be done either by using both forms as components of a single scope for the rules, or by passing the original source forms as parameters to translation functions that produce the corresponding translated target forms, or a combination of both strategies. This paradigm has also proven successful in several projects.