Excerpts from
The TXL Cookbook, Part II

Parsing Problems

James R. Cordy

School of Computing
Queen's University at Kingston,
Canada

Crafting Grammars

- The first stage of any TXL project is the creation or selection of a TXL grammar (*parser*) for the target language
  - Often one has already been written
- The form of the language *grammar* has a huge influence on the ease of writing transformation and analysis *rules*
- So in these first problems in the TXL Cookbook, we concentrate on *parsing problems* and solutions

tip: If you get the grammar right, TXL rules are simpler

Tiny Imperative Language

- Example problems in the TXL Cookbook are all based on the Tiny Imperative Language (*TIL*) designed for the purpose
  - Cordy & Visser 2005
  - Designed for demonstrating and comparing source transformation tools

```
// “factors.til” - Find all factors of an input number
var n;
write "Input n please";
read n;
write "The factors of n are";
var f;
f := 2;
while n != 1 do
    while (n / f) * f = n do
        write f;
        n := n / f;
    end;
    f := f + 1;
end;
```

Problem 1: Basic Parser

- In this first problem, our issue is simply the crafting of a new **TXL** grammar, for the *TIL* language
- It's not practical to show entire solutions in this presentation, so we will concentrate on key parts of each solution, and the general paradigms they introduce

- Paradigm: *The grammar is the parser.*
  - A **TXL** grammar is a directly interpreted parsing program
  - Must keep this in mind as we write the grammar
  - The *purpose* of a **TXL** grammar is to support analysis and transformation tasks, *not* to be a syntax checker, so forms can be approximate - simpler and looser

tip: Simpler forms are better, even if they are not precise

Problem 1: Basic Parser (cont’d)

- Begin with lexical forms

File "TIL.grm"

```
% TXL grammar for Tiny Imperative Language
% Jim Cordy, April 2005

% Keywords of TIL
keys
    var if then else while do for read write ’end
end keys

% Compound tokens
compounds
    := != <= >=
end compounds

% TIL comments - comments are ignored unless -comment is set
comments
    //
end comments
```

tip: Use default [id], [number], [stringlit] tokens as an approximation

Problem 1: Basic Parser (cont'd)

- Paradigm: *Sequences, not recursions.*
    - Because grammar directly interpreted, better to express sequences as [X*] rather than recursively

```
define program
    [statement*]
end define

define statement
    [declaration]
  | [assignment_statement]
  | [if_statement]
  | [while_statement]
  | [for_statement]
  | [read_statement]
  | [write_statement]
  | [comment_statement]
end define
```

```
program -> statements_opt

statements -> statement
            | statements statement
```

tip: TXL is designed and optimized for sequences, so use liberally

Problem 1: Basic Parser (cont’d)

- Paradigm: *Join similar forms.*
  - Because grammar directly interpreted, better to join multiple forms in TXL grammars

```
define if_statement
   if [expression] then [IN][NL]
      [statement*]      [EX]
   [opt else_statement]
   'end;               [NL]
end define

define else_statement
   else                [IN][NL]
      [statement*]      [EX]
end define
```

```
if_statement ->
        "if" expression "then"
            statements
        "end"  ";”
     |  "if" expression "then"
            statements
        "else"
            statements
        "end"  ";"
```

tip: Fewer forms also makes transformation patterns more general, avoiding accidentally missed cases

Problem 1: Basic Parser (cont'd)

- Paradigm: *Encode precedence and associativity directly in the grammar.*
    - Traditional way, without separate precedence tables

```
define expression
    [comparison]
  | [expression] [logop] [comparison]
end define

define logop
    and | or
end define

define comparison
    [term]
  | [comparison] [eqop] [term]
end define

define eqop
    = | != | > | < | >= | <=
end define

define term
    [factor]
  | [term] [addop] [factor]
end define
```

```
define addop
    + | -
end define

define factor
    [primary]
  | [factor] [mulop] [primary]
end define

define mulop
    * | /
end define

define primary
    [name]
  | [literal]
  | ( [expression] )
end define

define literal
    [number] | [stringlit]
end define
```

tip: Many tasks don't need precedence, can use even simpler grammar

Problem 1: Basic Parser (cont’d)

- *TIL* parser
  - A parser is just a transformation that does nothing but recognize the input

File "TILparser.txl"

```
% TXL parser for TIL
% Jim Cordy, April 2005

% Use the TIL grammar we crafted
include "TIL.grm”

% Only need to recognize the input
function main
    match [program]
        _ [program]
end function
```

Problem 1: Basic Parser (cont'd)

```
linux% txl multiples.til TILparser.txl -xml

TXL v10.5d (1.7.08) (c)1988-2008 Queen's University at Kingston
Compiling TILparser.txl ...
Parsing multiples.til ...
Transforming ...

<program>
 <repeat statement>
  <statement><for_statement> for
    <name><id>i</id></name> :=
    <expression><primary><literal><number>1</number></literal></primary></expression> to
    <expression><primary><literal><number>9</number></literal></primary></expression> do
    <repeat statement>
     <statement><for_statement> for
       <name><id>j</id></name> :=
       <expression><primary><literal><number>1</number></literal></primary></expression> to
       <expression><primary><literal><number>10</number></literal></primary></expression> do
       <repeat statement>
        <statement><write_statement> write
          <expression>
           <expression><primary><name><id>i</id></name></primary></expression>
           <op>*</op>
           <expression><primary><name><id>j</id></name></primary></expression>
          </expression> ;
         </write_statement>
        </statement>
       </repeat statement> end ;
      </for_statement>
     </statement>
    </repeat statement> end ;
   </for_statement>
  </statement>
 </repeat statement>
</program>
```

Problem 2: Pretty-printing

- The next problem is the crafting of a pretty-printer for *TIL*
  - Pretty-printing is a common problem, since transformations need to have usable output

- Paradigm: *Use formatting cues to control output format.*
  - Special nonterminals built in to TXL control output format
  - No effect on input

```
define if_statement
   if [expression] then [IN][NL]
      [statement*]      [EX]
   [opt else_statement]
   ’end;               [NL]
end define

define else_statement
   else                 [IN][NL]
      [statement*]      [EX]
end define
```

tip: Keep formatting cues separated on the right for readability

Problem 2: Pretty-printing (cont'd)

- Pretty-printers also need to preserve and format comments
  - By default **TXL** ignores comments in input

- Paradigm: *Preserving comments in output.*
  - A weakness of **TXL** is comments must be parsed to keep them
  - The *-comments* switch makes comments parseable tokens

```
define statement
        [declaration]
    |   [assignment_statement]
    |   [if_statement]
    |   [while_statement]
    |   [for_statement]
    |   [read_statement]
    |   [write_statement]
    |   [comment_statement]
end define
```

```
define comment_statement
        [NL] [comment] [NL]
end define
```

tip: Switches can be set in the TXL program itself using #pragma

Problem 2: Pretty-printing (cont’d)

File: “multiples.til”

```
// Output first 10 multiples of numbers 1 through 9
for i:=1 to 9 do for j:=1 to 10 do // Output each multiple
  write i*j; end; end;
```

```
linux% txl -comment multiples.til TILparser.txl
```

```
TXL v10.5d (1.7.08) (c)1988-2008 Queen’s University at Kingston
Compiling Txl/TILparser.txl ...
Parsing Examples/multiples.til ...
Transforming ...

// Output first 10 multiples of numbers 1 through 9
for i := 1 to 9 do
    for j := 1 to 10 do
        // Output each multiple
        write i * j;
    end;
end;
```

Problem 3: Language Extensions

- Syntactic extensions to *TIL*
  - Handling language *extensions, dialects* and embedded *DSLs* is a common problem when using source transformation systems

- Paradigm: *Extension of grammatical forms.*
  - Use *redefine* to add or modify existing forms
  - Normally stored in separate *grammar overrides* files

```
% Begin-end dialect of TIL
redefine statement
    ...                     % refers to all existing forms

  | [begin_statement]       % add alternative for our new form
end redefine

define begin_statement
    begin
        [statement*]
    ‘end
end define
```

```
include “TIL.grm”
include “TILbeginend.grm”
```

tip: Don’t change the base grammar - use redefines for variants

Problem 3: Language Extensions (cont’d)

- Paradigm: *Preferential ordering of grammatical forms.*
  - When extensions are independent, no problems, but some extensions may introduce conflicting forms
  - In TXL, alternatives are *ordered*, with earlier forms preferred

```
% Array dialect of TIL

redefine declaration
    var [name] [opt subscript] ;    [NL]
  | ...
end redefine

redefine primary
    [name] [opt subscript]
  | ...
end redefine

define subscript
    ‘[ [expression] ‘]
end define
```

tip: Pre-extension prefers new forms; post-extension old forms

Problem 3: Language Extensions (cont'd)

- Paradigm: *Replacement of grammatical forms.*
  - Redefinitions can also completely *replace* existing forms, forcing the new parse in all cases

```
% Array dialect of TIL (continued)

redefine assignment_statement
    [name] [opt subscript] := [expression] ;     [NL]
end redefine
```

- Paradigm: *Modification of grammatical forms.*
  - Extended forms need not be separate alternatives, they can simply *modify all* the original forms

```
% From the module dialect of the function dialect of TIL

redefine function_definition
    [opt 'public] ...
end redefine
```

tip: Extensions are more independent of base grammar forms

Problem 3: Language Extensions (cont'd)

- Paradigm: *Composition of dialects and extensions.*
  - Language extensions and dialects can be *composed* and *combined* to create more sophisticated dialects
  - Example: information-hiding *module* dialect of *function* dialect of *TIL*

```
include "TIL.grm"
include "TILarrays.grm"
include "TILfunctions.grm"
include "TILmodules.grm"
```

tip: Order matters, since redefines modify the previous definition

Problem 3: Language Extensions (cont’d)

File "TILmodules.grm"

```
% TXL grammar overrides for module extension to TIL
% Jim Cordy, March 2009

% Requires functions extension

redefine declaration
    ...                    % existing forms for [declaration]
  | [module_definition]    % new module form
end redefine

keys
    module public          % new keywords of this dialect
end keys

define module_definition
    module [name]          [IN][NL]
        [statement*]       [EX]
    'end ;                [NL][NL]
end define

redefine function_definition
    [opt 'public] ...
end redefine
```

Problem 3: Language Extensions (cont'd)

File "primes.mtil"

```
// determine primes up to maxprimes
// using the sieve method
var maxprimes;
var maxfactor;
maxprimes := 100;
maxfactor := 50;

// maxprimes div 2
var prime;
var notprime;
prime := 1;
notprime := 0;

module flags
    var flagvector [maxprimes];

    public function flagset (f, tf)
        flagvector [f] := tf;
    end;

    public function flagget (f) : tf
        tf := flagvector [f];
    end;
end;
```

```
// everything begins as prime
var i;
i := 1;
while i <= maxprimes do
    flagset (i, prime);
    i := i + 1;
end;
 . . .
```

- module dialect
- function dialect
- array dialect
- original TIL

Problem 4: Robust Parsing

- Robust statement parsing for *TIL*
  - *Robust parsing* is important in program analysis and transformation since languages often poorly documented, or compilers allow undocumented or local forms
  - Must allow *exceptions* for unknown forms

- Paradigm: *Fall-through forms.*
  - Exploit ordered parsing to allow for unexplained input as the *least preferred* alternative

```
redefine statement
    ...                     % known forms for [statement]
  | [unknown_statement]     % fall-through if not recognized
end redefine
```

tip: Can add robustness at multiple levels

Problem 4: Robust Parsing (cont'd)

- Paradigm: *Uninterpreted forms.*
  - Need a way to *accept* input we don't recognize
  - In TXL, this uses built-in types *[token]* and *[key]*

```
% TXL nonterminal type to accept one arbitrary item from input

define token_or_key
        [token]        % any input token that is not a keyword
     |  [key]          % any keyword
end define
```

- Paradigm: *Guarded forms.*
  - Need a way to *not accept* input we *can* recognize
  - In TXL, we use nonterminal *guards*

```
% TXL type to accept any item that is not a semicolon

define not_semicolon
        [not ';] [token_or_key]    % any input token except semicolon
end define
```

tip: Guards can be any nonterminal type, and accept no input

Problem 4: Robust Parsing (cont’d)

File "TILrobust.grm"

```
% TXL grammar overrides for robust parsing extension to TIL
% Jim Cordy, March 2009

redefine statement
    ...                         % all known forms for [statement]
  | [unknown_statement]         % fall-through if we don’t know it
end redefine

define unknown_statement
    [not_semicolon*] ;          [NL]
end define

define not_semicolon
    [not ’;] [token_or_key]    % any token except semicolon
end define

define token_or_key
    [token]                     % any token that is not a keyword
  | [key]                       % any keyword
end define
```

Problem 5: Island Grammars

- Agile parsing refers to the use of grammar tuning on an individual analysis or transformation task basis
  - *Island grammars* are a technique related to robust parsing in which the known things are the *exceptions* rather than the rule
  - In essence, the inverse of *robust parsing*, where the input is a sea of unknown things (the *water*) containing embedded instances of known things (the *islands*)
  - Examples: analyze only the *C* examples in a textbook, or only the *EXEC SQL* blocks in a large set of *Cobol* programs

- Paradigm: *Preferential island parsing.*
  - Exploit **TXL** ordered parsing to allow for the interesting input as the *most preferred* alternative, and fall through to the uninteresting input

Problem 5: Island Grammars (cont'd)

File "Islands.grm"

```
% Generic grammar for parsing documents with embedded islands
% Jim Cordy, June 2009

% Input is a sequence of interesting islands and uninteresting water
redefine program
    [island_or_water*]
end redefine

define island_or_water
    [island] | [water]
end define

% And the water is any input that is not an island
define water
    [not_island*]
end define

define not_island
    [not island] [token_or_key]    % any token not beginning an island
end define

define token_or_key
    [token] | [key]
end define
```

Problem 5: Island Grammars (cont'd)

File "TILislands.txl"

```
% TXL program for parsing documents with embedded TIL programs
% Jim Cordy, June 2009

% Begin with the TIL grammar
include "TIL.grm"

% And the generic island grammar
include "Islands.grm"

% In this case the islands are TIL programs
define island
    [til_program]
end define

define til_program
    [statement+]    % At least one TIL statement
end define

% Analysis or transformation can now target the embedded TIL parts
% In this case, delete the non-TIL parts to yield the TIL code only
rule main
    replace [not_island*]
        Water [not_island*]
    by
        % Nothing
end rule
```

Agile Parsing

- Agile parsing refers to the tuning of a grammar on an *individual task basis* to better support the particular analysis or transformation task
  - Use the *parser* to better *isolate* the parts of the program of interest, or to make them more *amenable* to the task
  - Can greatly *simplify the rules* necessary to perform the task
  - In essence, create a *special dialect grammar* to support the task

- Paradigm: *Transformation-specific forms.*
  - Use the same **TXL** *grammar overrides* technique to get a more appropriate or abstract parse for the task
  - Add special *intermediate or output forms* to support the transformation or analysis
  - Add *optional attributes and annotations* to store intermediate information used by the transformation or analysis

That’s It!

- Basically, that’s all about TXL parsing paradigms
- Next:
    - TXL lab this afternoon – more test problems to try
- Tomorrow:
    - Part III: Some Recipes for Analysis and Transformation Problems using TXL

Then:

- TXL challenge problems for those so inclined