ELEC 875
Design Recovery
and
Automated Evolution

Transformation
Paradigms
Week 6 Class 1

Today

- Combining Input and Output Grammars
  - Union Grammars
  - Consume/Edit Grammars
- Transformation Strategien
  - Union Transformations
  - Consume/Edit Transformations
- Annotation Strategy
- Ad Hoc Polymorphic transformation

About TXL

- Start at
  - ◊ http://www.txl.ca/txl-learn.html
    - Simple Intro + Examples
  - ◊ http://www.txl.ca/txl-docs.html
    - – Read the TXL Programming Language
  - ◊ back to http://www.txl.ca/txl-learn.html
    - – Do the TXL challenge

About TXL

- Some Quirks from Monday
  - ◊ TXL mixes both the TXL language and the language being transformed.
  - ◊ Sometimes the same keyword or symbol is used in both.
    - – e.g. square brackets are types and rules in TXL and arrays in C, some language uses the keyword end
    - – use a single quote to indicate that something is to be used as a data element.
      '[Expr [cexpression] ']
    - – since charlits have single quotes, they have to be quoted: ''foo'

Combining Input and Output Grammars

- Problem:
  - ◊ Our input and output grammars might not be the same!
    - – e.g. translate Java to Python
    - – both input and output must be parsed by the same grammar.

- Two General Solutions:
  - ◊ Union Grammars - combine the grammars into a single grammar at many levels
  - ◊ Consume/Emit Grammar - combine them only at the top level

Union Grammars

- Combine the grammars at several levels
  - ◊ Useful when the two grammars are similar or have similar concepts. For example, C and Pascal. Both are block/statement/expression based languages.
  - ◊ Combine at each level where they match.
  - ◊ In our example, we would combine the grammars at the global declaration level, the procedure level, the statement level and the expression level.
  - ◊ Assume that syntax of input is correct. Grammar will allow mixed programs as input.

C Grammar

```
define program
    [repeat decl]
end define

define decl
    [var_decl]
  |[proc_decl]
end define
define proc_decl
   [opt type] [id] [header]
      [block]
end define
define block
    '{
       [repeat var_decl]
       [repeat statement]
    '}
```

```
define statement
    ...
    | [if_statement]
    ...
    | [block]
end define
define if_statement
    'if [expression]
    [statement]
    'else [statement]
end define
```

Pascal Grammar

```
define program
  'program [id] [file_header]
   [repeat decl]
   [block] '.
end define
define decl
    [var_decl]
   |[proc_decl]
end define
define proc_decl
   [procedure_or_function]
     [id] [header]
      [repeat decl]
      [block]
end define
```

```
define block
    'begin
       [repeat statement]
    'end
end define
define statement
    ...
    | [if_statement]
    ...
    | [block]
end define
define if_statement
    'if [expression]
    'then [statement]
    'else [statement]
end define
```

Union Grammar - Technique 1

```
define program
        [pascal_program]
    |  [c_program]
end define
define pascal_program
  'program [id] [file_header]
    [repeat decl]
    [block] '.
end define
define c_program
  [repeat decl]
end define
```

Union Grammar - Technique 1

```
function main
  replace [program]
    'program _ [id] _ [file_header]
      Decls [repeat decl]
      MainBlock [block] '.
  construct MainProc
    int main (int argc, char * argv[])
      MainBlock [replaceBlock]
                [addReturn0]
  by
    Decls [replaceDecls]
    [. MainProc]
end function
```

Union Grammar - Technique 2

```
define block
  [begin_or_brace]
    [repeat decl]
    [repeat statement]
  [end_or_brace]
end define

define begin_or_brace
  'begin | '{
end define

define end_or_brace
  'end | '}
end define
```

```
define if_statement
    'if [expression]
    [opt 'then]
            [statement]
    'else [statement]
end define
```

Union Grammar - Technique 2

```
rule replace_block
    replace [block]
      'begin
        Stmts [repeat statement]
'end by
      '{
         Stmts [replaceStatements]
'} end rule
```

Union Grammars - Final Words

- Combine the grammars at several levels
  - ◊ Two general techniques - can be combined
  - ◊ Combine at each level where they match.

- Have to be careful of introducing ambiguities
  - ◊ Grammars may interact in unforeseen ways

Consume/Emit Grammars

- Combine the grammars only at the top level (or maybe not at all!!)
  - ◊ Useful when the two grammars very different.
  - ◊ Source grammar is separated from output grammar

- Several techniques:
  - ◊ Parallel decomposition/Construction
  - ◊ Global Variable Accumulation
  - ◊ Attribute/Extraction

Consume/Emit Parallel Execution

- Pass the input in as a parameter to the first rule
  - ◊ Rules construct output in the scope

- Example:
  - ◊ Convert Alphabet to Morse Code

Consume / Emit Parallel Execution

```
define program
    [repeat id]
  | [repeat stringlit]
end define

function main
    replace [program]
      Input [repeat id]
    construct EmptyResult [repeat stringlit]
      _
    by
      EmptyResult [asciiToMorse Input]
end function
```

Consume / Emit Parallel Execution

```
define AsciiMorse
    [id] [stringlit]
end define
function asciiToMorse Input [repeat id]
  construct Table [repeat AsciiMorse]
    A ".-" B "-..." C "-.-." ...
  deconstruct Input
    NextChar [id] RestInput [repeat id]
  deconstruct * [AsciiMorse] Table
    NextChar ResultMorse [stringlit]
  replace [repeat stringlit]
    FinalResult
  by
    FinalResult [. ResultMorse]
                [asaciiToMorse RestInput]
end function
```

Consume / Emit Parallel Execution

```
define program
    [repeat id] | [repeat stringlit]
end define
define AsciiMorse
    [id] [stringlit]
end define
function main
   replace [program]
     Input [repeat id]
   construct Table [repeat AsciiMorse]
     A ".-" B "-..." C "-.-." ...
   construct EmptyResult [repeat stringlit]
     _
   by
     EmptyResult [asciiToMorse Table each Input]
end function
```

Consume / Emit Parallel Execution

```
function asciiToMorse Table[repeat ASciiMorse] Input[id]

  deconstruct * [AsciiMorse] Table
    Input ResultMorse [stringlit]

  replace [repeat stringlit]
    FinalResult

  by
    FinalResult [. ResultMorse]

end function
```

Consume / Emit Global Variables

- Parallel execution constructs output in scope
  - ◊ Input only exists in parameters or variables
  - ◊ Sometimes complex patterns on input means it must remain the scope
- Build the result up in a global variable
  - ◊ Replace input with output at the last moment

Consume / Emit Global Vars

```
define program
    [repeat id]|[repeat stringlit]
end define
function main
   export Morse [repeat stringlit]

      _
   replace [program]
     Input [program]
   by
     Input [BuildResult]
           [replaceByResult]
end function
```

Consume / Emit Global Vars

```
define AsciiMorse
    [id] [stringlit]
end define
rule buildResult
  construct Table [repeat AsciiMorse]
    A ".-" B "-..." C "-.-." ...
  replace $ [repeat id]
    TheChar [id] Rest [repeat id]
  deconstruct * [AsciiMorse] Table
    TheChar TheMorse [stringlit]
  import Morse [repeat stringlit]
  export Morse
    Morse [. TheMorse]
  by
    TheChar Rest
end rule
```

Consume / Emit Global Vars

```
rule replaceByResult
  replace [program]
    _ [program]
  import Morse [repeat stringlit]
  by
    Morse
end rule
```

Consume/Emit Attribute Extraction

- Provide pockets in the input grammar to hold results
  - ◊ Put results in the pockets
  - ◊ Extract the results

- Pockets may introduce parsing ambiguities

Consume / Emit Attribute Extraction

```
define program
    [c_program] | [repeat facts]
end define
define fact
  '$ [id] '( [list fact_arg] ') '$
end define
redefine assignment
    [lvalue] '= [expression] [opt fact]
end redefine
function main
  replace [program]
    P [program]
  by
    P [annotate]
      [replaceByResult]
end function
```

Consume / Emit Attribute Extraction

```
rule annotateAssignment
    replace $ [assignment]
      Lval [lvalue] '= E [expression]
    by
      LVal '= E '$ assign(LVal,E) '$
end rule

function replaceByResult
  replace [program]
   P [program]
  construct Facts [repeat fact]
    _ [^ P]
  by
    Facts
end function
```

Annotation Strategy

- Task: highlight elements of interest
  - ◊ Like pockets, but do not do the extract and replace
  - ◊ "Pockets" are now markers that we turn on and off

- Use:
  - ◊ Mark and Transform Algorithms
  - ◊ Markup for Human Consumption

- HSML [Cordy et al.]
  - ◊ Automates much of this technique

Annotation Strategy

```
rule transformAssignment
    replace $ [assignment]
      Lval [lvalue] '= E [expression] '$
    by
      LVal [doSomething1]
       '=
       E   [doSomething2]
end rule
```

Ad Hoc Polymorphic Rules

- Everthing discussed until now has been strongly typed.
  - ◊ Impossible to build a bad tree
- [any] changes that
  - ◊ Allows more generic rules
  - ◊ As a pattern matches any tree
  - ◊ As a parameter type, accepts any tree
  - ◊ Cannot be constructed, only matched and bound
  - ◊ Tree retains its internal structure (can be searched)
  - ◊ Allows replacement to break grammar (dangerous)
  - ◊ Page 36-40 of the language reference manual

Ad Hoc Polymorphic Rules

```
rule markupStatementsMentioning Ids [repeat id]
                                Markup [id]
  skipping [markup]
  replace $ [statement]
    Stmt [statement]
  where
    Stmt [contains each Ids]
  by
    Stmt [markupWith Markup]
end rule
```

Ad Hoc Polymorphic Rules

```
define markup
    '< [id] '> [any] '</ [id] '>
end define

function markupWith Tag [id]
  replace [any]
    Any [any]
  construct Markup [markup]
    '< Tag '> Any '</ Tag '>
  deconstruct Markup
    MarkupAny [any]
  by
    MarkupAny
end function
```