Pretty-Printing As Compiling

Lior Zur-Lotan \ Ben-Gurion University

July 12, 2021

Outline

1. Pretty-Printers
2. Generating Pretty-Printers with UGLY
3. Discussion

Pretty-Printers

1. Origins and Uses
2. Existing Algorithms
3. Limitations of Modern Algorithms
4. Pretty-Printing as Compiling

Pretty-Printers — Origins and Uses

1. Introduced in 1960s for LISP
2. Commonly used today to:
   - Clarifying the structure of code
   - Linting
   - Enforcing/Applying coding standards

Pretty-Printers — Existing Algorithms

1. Oppens' Algorithm
2. Invertible Parsers - Syntax Descriptors
3. Universal Parsers - CodeBuff

Pretty-Printers — Limitations of Modern Algorithms

1. Cannot consider the syntactic context of the target language when formatting
2. Cannot insert or alter input tokens
3. Changing format requires editing the pretty-printer[1]
4. Require the user to provide a scanner for the input language

[1] Or the pretty-printer's training set

Pretty-Printing as Compiling

1. Pretty-Printing is an extension of parsing, where the AST is translated into text
2. Pretty-Printing can be seen as a special case of compiling from a formal language into a graphical representation of its code
    - The semantics of the language is irrelevant since the target language uses the same semantics

Generating Pretty-Printers with UGLY

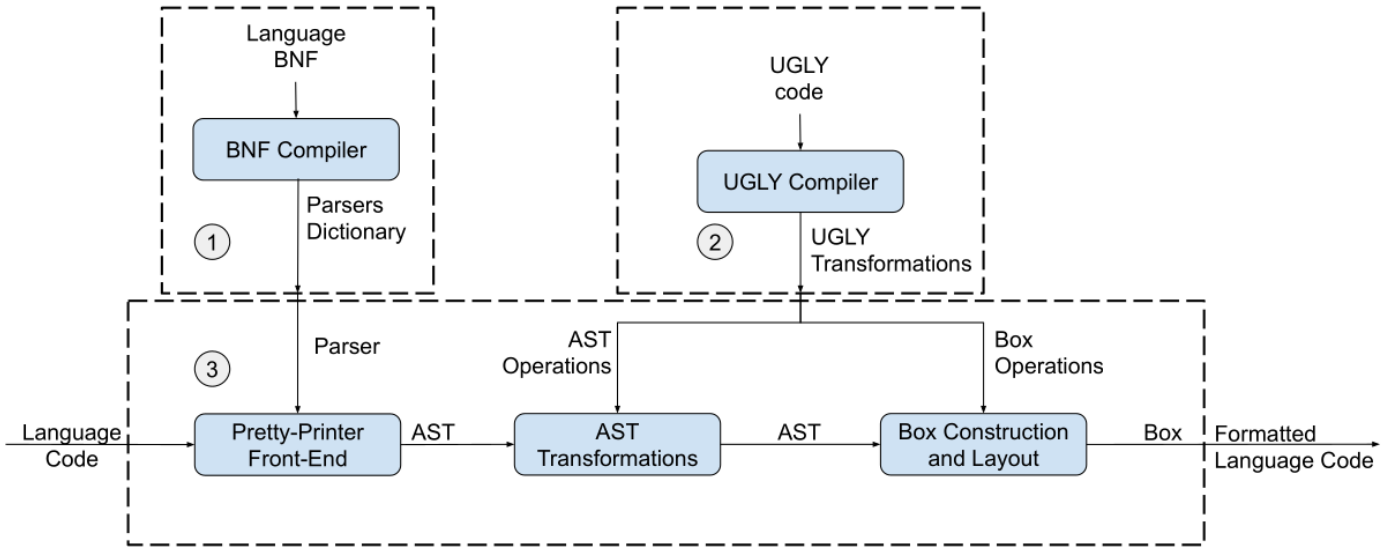
1. Universal Grammar LaYout — UGLY
2. UGLY Pipeline
3. UGLY Syntax
4. Examples
5. Complexity Analysis
6. Comparison With Existing Algorithms

UGLY — Universal Grammar LaYout

1. UGLY is a declarative DSL that's used to define transformations on the syntax of other languages
2. UGLY code defines transformations for the syntactic elements of the input language
3. UGLY uses a universal parser-generator as its frontend
   - Universal generator takes in a the syntax of the language to produce generic parsers
   - The concrete syntax is given as a BNF
   - Generic parsers output generic AST
4. The target language of UGLY is a language for representing 2D boxes
5. UGLY runtime applies transformations on the generic AST and box language to produce the formatted output

UGLY — Pipeline

1. Language BNF → BNF Compiler → Generic Parsers (frontend)
2. UGLY code → UGLY Compiler → Box Transformations (backend)



3. Input code → Generic parsers → Generic AST
4. Generic AST + Box Transformations → UGLY Runtime → Formatted output

UGLY — Syntax

1. Settings
   - Global controls over the output (e.g. line width)
2. Operators
   - Transformation on syntactic elements
3. Conditions
   - Controls when operations are applied to syntactic elements

UGLY examples — JSON

```
{"Name": "Code Generators", "Uses": "Generating code from DSL or data",
"Types": [{"Family": "Parser Generators",
    "Introduced": "1960s",
     "Common Use": "Generating a compiler's parsers frontend",
     "Members": [{"Name": "YACC","Language": "C","Published": 1975},
         {"Name" : "BISON","Language": "C","Published": 1985},
         {"Name" : "JavaCC","Language": "Java","Published": 1996}]},
    {"Family": "Lexer Generators",
     "Introduced": "1970s",
     "Common Use": "Generating lexers/scanners",
     "Members": [{"Name": "flex", "Language": "C", "Published": 1987},
         {"Name": "re2c", "Language": "C/C++", "Published": 1994}]},
    {"Family": "Pretty-Printer Generators",
     "Introduced" : "2010s","Common Use" : "Generating beautifiers",
     "Members": [{"Name": "UGLY", "Language": "UGLY DSL", "Published": 2019}]}]}
```

UGLY Examples — UGLY for JSON

```
verts([<object members>, <members>, <array elements>
       <elements>])
replace(<SPACES>,'')
append(<MEMBER_COLON>, ' ')
```

UGLY Examples — Formatted JSON

```
{"Name": "Code Generators",
 "Uses": "Generating code from DSL or data",
 "Types": [{"Family": "Parser-Generators",
            "Introduced": "1960s",
            "Common Use": "Generating a compiler's parsers front-end",
            "Members": [{"Name": "YACC",
                         "Language": "C",
                         "Published": 1975},
                        {"Name": "BISON",
                         "Language": "C",
                         "Published": 1985},
                        {"Name": "JavaCC",
                         "Language": "Java",
                         "Published": 1996}]},
           {"Family": "Lexer Generators",
            "Introduced": "1970s",
            "Common Use": "Generating lexers/scanners",
            "Members": [{"Name": "flex",
                         "Language": "C",
                         "Published": 1987},
                        {"Name": "re2c",
                         "Language": "C/C++",
                         "Published": 1994}]},
           {"Family": "Pretty-Printer Generators",
            "Introduced": "2010s",
            "Common Use": "Generating beautifiers",
            "Members": [{"Name": "UGLY",
                         "Language": "UGLY DSL",
                         "Published": 2019}]}]}
```

UGLY Examples — BNF

```
<digit> = '0'-'9'
<num> := ' '* <digit>+ ' '*
<CI word> -> `a`-`z`*
<word> ::= 'a'-'z'*
<S> ::= <num> | <word> | <CI word>
```

UGLY Examples — UGLY for BNF

```
replace(<expand-to>, '::=') ; Use " ::= " for "expand-to"
align_center(<grammar>)     ; Align productions to center
```

UGLY Examples — Formatted BNF

```
  <digit> ::= '0'-'9'
    <num> ::= ' '* <digit>+ ' '*
<CI word> ::= `a`-`z`*
   <word> ::= 'a'-'z'*
      <S> ::= <num> | <word> | <CI word>
```

UGLY Examples — Pascal

```
program permutations; var p: array[1 .. 10] of integer;
is_last: boolean; n: integer; demo: string; long_variable:
integer procedure next; var i, j, k, t: integer begin
is_last := true; i := n - 1; while i > 0 do begin if p[i] <
 p[i + 1] then begin is_last := false; break end; i := i - 1
 end; if is_last = false then begin j := i + 1; k := n;
while j < k do begin t := p[j]; p[j] := p[k]; p[k] := t; j
:= j + 1; k := k - 1 end; j := n; while p[j] > p[i] do  j :=
j - 1; j := j + 1; t := p[i]; p[i] := p[j]; p[j] := t end
end begin while (n < 10) do begin p[n] := n; n := n + 1 end;
write('Calculating all permutations for n = ', n, stdout);
while is_last = false do next end.
```

Discussion

1. Complexity Analysis
2. Usability Compared With Existing Algorithms
3. Capabilities Compared With Existing Algorithms
4. Future Work

Complexity Analysis

1. Runtime
   - Frontend — Same as the complexity of the generated parsers
   - Backend — Linear in the number of syntactic elements
2. Space
   - Frontend — Same as the complexity for the generated parsers
   - Backend — Linear in the number of syntactic elements

Usability Compared With Existing Algorithms

1. Benefits
   - Input — Only requires BNF syntax and UGLY code
2. Drawbacks
   - Language-support — Limited by the capabilities of the parser-generator

Capabilities Compared With Existing Algorithms

1. Benefits
   - User Controls Format — Changing format doesn't require rewriting any part of the Pretty-Printer
   - Transformations — UGLY transformations allow changing, removing and adding syntactic elements
   - Universal — Improving the supported languages for the parser-generator automatically adds support for those new languages
2. Drawbacks
   - Unsound — User can write UGLY code that can change the syntax and semantics of the input

Future Work

1. Improving Generic Parsers
2. Extending Box Language
3. Runtime Selectors
4. UGLY variables

Future Work — Improving Generic Parser

1. Allow the user to name a group of syntactic elements in the RHS of production rules. This will allow users to reference the group in UGLY
2. Replace the parser-generator to support more lanugages or improve it's runtime

Future Work — Extending Box Language

1. Add an operation to back-indent a syntactic element relative to its parent. This will support formats such as

```
(define foo (lambda (x)
  (+ x 1)))
```

   - The s-expression `(+ x 1)` is nested under the node that holds the `lambda` syntactic element in the AST. However, it's indented relative to the position of the `define` element.

Future Work — Runtime Selectors

- Allow the user to define multiple formats
- Have UGLY runtime select the best one using a heuristic function to that evaluates the utility of formats and selects the best one.

Future Work — UGLY Variables

- UGLY has the computational strength of a DFA.
- Adding variables to UGLY can increase the computational power to that of a stack-automata
- Allow many more transformations such as translating data between different storage formats (from XML to JSON, for instance).

Questions

Questions?