C2VPG:
Translating Practical Context-Free Grammars into Visibly Pushdown Grammars by Order-Based Tagging

Xiaodong Jia and Gang Tan

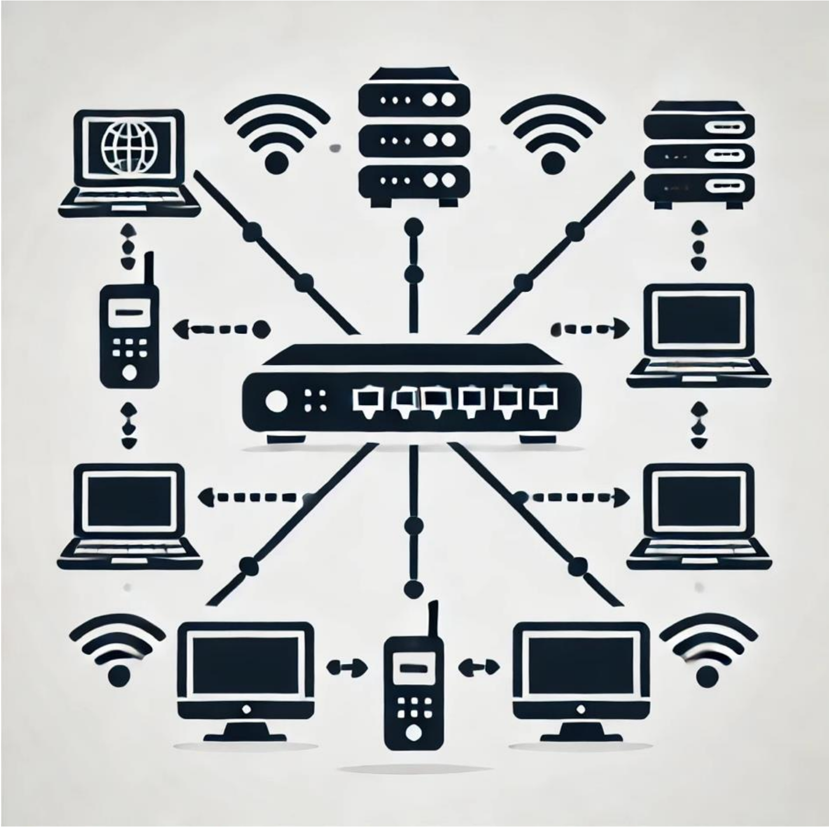
The Pennsylvania State University

LangSec 2025

The Importance of Parsing

Parsing is essential in computer systems:

- High-assurance parsers are vital for security in web applications.
- High-performance parsers are required in web browsers and network routers.
- Inefficiency is a problem:
  - Denial of Service via Algorithmic Complexity Attacks. USENIX Security ’03
  - REVEALER: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. S&P ’21



Motivation

- Parsers for Context-Free Grammars (CFGs) have efficiency limitations.
  - E.g., CYK Parser runs in $O(|s|^3)$.
- LL(k) and LR(k) parsers place restrictions on what CFGs can be accepted.
  - A burden to refactor the grammar.
- Visibly Pushdown Grammars (VPGs) for parsing:
  - Expressive: stronger than regular grammars but weaker than CFGs
  - Secure: formally verified parser generator
  - Efficient: Linear time parsing
  - Easy to use
    - Jia et al. (OOPSLA'21, TOPLAS'23): *A Derivative-Based Parser Generator for Visibly Pushdown Grammars*
    - Jia et al. (PLDI'24): *V-Star: Learning Visibly Pushdown Grammars from Program Inputs*

Background: Visibly Pushdown Grammars

A Visibly Pushdown Grammar (VPG) is a CFG $G = (V, \Sigma, P, L_0)$, where

- The set of terminals Σ is a *disjoint* union of call, plain and return symbols:

$$\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$$

- And each rule in P has only three forms:
    1. The epsilon rule $L \to \epsilon$
    2. The linear rule $L \to cL'$, where $c \in \Sigma_{\text{plain}}$
    3. The matching rule $L \to \langle a L_1 b \rangle L_2$

    where $L \in V$, c is a *plain* symbol, $\langle a$ is a *call* symbol and $b\rangle$ is a *return* symbol.

The first two forms describe the (right) regular grammar.

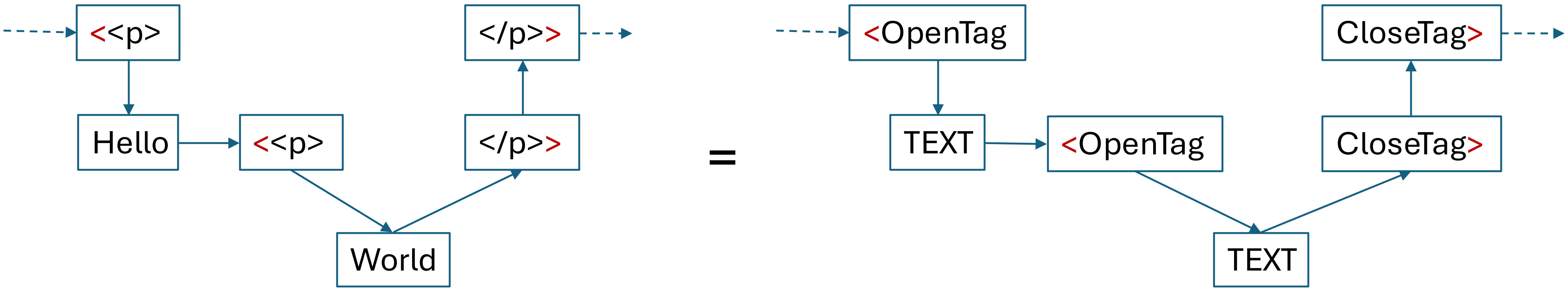
The third form introduces hierarchically nested matching of symbols.

Background: Model XMLs as VPGs

xml -> <OpenTag xml CloseTag> xml | TEXT xml | ϵ

Raw input: … <p>Hello<p>World</p></p> …

Tokenizer and Tagger



Tagged token stream

Tagged token stream

Background: Model CFGs as VPGs

Tagged CFGs: CFGs with tagging information.

- Tag terminals as call/plain/return.
- Mark call and return symbols by operators < and >.
- **Nest** each <u>non-tail</u> recursion within **paired** call and return symbol.

```
expr   : digits '+' expr
       | digits ;
digits : [0-9]+ ;
```

Example Tail Recursion

```
json  : value ;
obj   : <'{' pair (',' pair)* '}'>
      | <'{' '}'> ;
pair  : STRING ':' value ;
arr   : <'[' value (',' value)* ']'>
      | <'[' ']'> ;
value : STRING | NUMBER | obj | arr
      | 'true' | 'false' | 'null' ;
```

Tagged CFG for JSON

Background: Model CFGs as VPGs

Tagged CFGs: CFGs with tagging information.

- Tag terminals as call/plain/return.
- Mark call and return symbols by operators < and >.
- **Nest** each <u>non-tail</u> recursion within **paired** call and return symbol.

```
document  : prolog? misc* element misc* ;
prolog    : XMLDeclOpen attribute* SPECIAL_CLOSE ;
content   : chardata?
            ((element | reference | CDATA | PI | COMMENT) chardata?)* ;
element   : <OpenTag content CloseTag> | SingleTag ;
reference : EntityRef | CharRef ;
attribute : NAME '=' STRING ;
chardata  : TEXT | SEA_WS ;
misc      : COMMENT | PI | SEA_WS ;
```

Tagged CFG for XML

Tagging Function

- *Tagging function* f maps a context-free language to a new language:

$$f : \Sigma^* \to \{‹i, i, i› \mid i \in \Sigma^*\} = \{i, i, i \mid i \in \Sigma^*\}$$

- For example:

f({“a”:[1,2,[3]]}) = ‹{“a”: ‹[1,2, ‹[3] ›] ›} ›

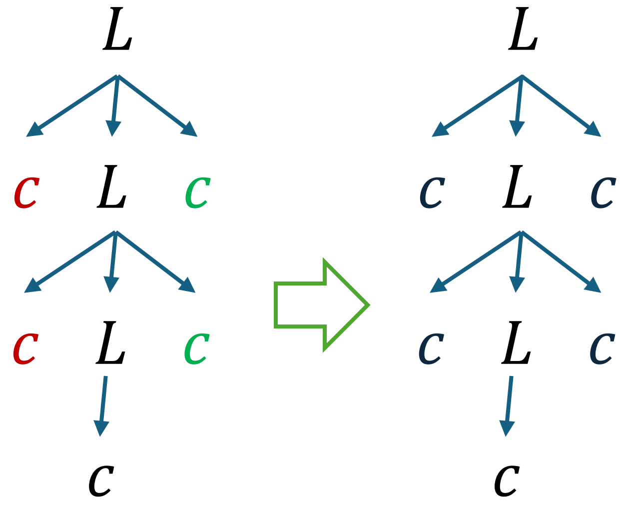
= {“a”:[1,2,[3]]}

- Observation: f can tag the same terminal differently, e.g.:

$$f(ccccc) = ‹c‹ccc›c› = ccccc$$

Tagging Function

- So, consider the CFG
$$G: \;\; L \to cLc \mid c$$
- We can convert it into a VPG by tagging f:
$$G': \;\; L \to cLc \mid c$$
- To parse string $s = ccccc$:
  - Using f to convert s to $s' = ccccc$
  - Parse s' using VPG G' and get parse tree:



3. Remove the tagging in the VPG parse tree to get the CFG parse tree

Tagging Function

Any CFG can be “converted” into a VPG by a tagging function.

(VPGs are weaker than CFGs, but the VPG above is on a new terminal set.)

The challenge is to find an efficient tagging function.

Tagging Function

So, consider the CFG

$$G\colon\ \ L \to cLc \mid c$$

We can convert it into a VPG by a tagging f:

$$G'\colon\ \ L \to \langle cLc\rangle \mid c$$

To parse string $s = ccccc$:

1. Using f to convert s to $s' = ccccc$

How to implement f?
One can see that f is quite flexible—it can be any function.
Caveat: The complexity of parsing CFG is shifted to f.

C2VPG: Order-Based Tagging

- The “dangling-else” challenge from practical grammars:

  if x then if y then z

  if x then if y then z else w

- How to tag the CFG?

```
if_expr -> IF if_expr THEN if_expr (ELSE if_expr)?
         | [a-z]
```

C2VPG: Order-Based Tagging

- How to tag the CFG? Two plausible ways:

  if_expr -> IF if_expr THEN if_expr (ELSE if_expr)?

  if_expr -> IF if_expr THEN if_expr (ELSE if_expr)?

- Neither will work. Instead, tag based on context:

  if_expr -> IF if_expr THEN if_expr

  | IF if_expr THEN if_expr ELSE if_expr

C2VPG: Order-Based Tagging

if_expr -> IF if_expr THEN if_expr
| IF if_expr THEN if_expr ELSE if_expr
| [a-z]

There is an **order** M on the terminals of return symbols:

M: THEN < ELSE

Construct efficient tagging function f with order M using a stack. For string:

if x then y else z

Push each terminal to the stack, one by one (omitting plain symbols):

[IF]

[IF, THEN] // If the string ends here, IF is paired with THEN

[IF, ELSE] // Removes THEN because THEN < ELSE

// So, IF is paired with ELSE

C2VPG: Order-Based Tagging

How to find such order M? C2VPG has two steps:

1. Build *recursion tuple* (x, y), where call must be in string x, and return must be in string y. E.g., for CFG:

```
if_expr -> IF if_expr THEN if_expr
         | IF if_expr THEN if_expr ELSE if_expr
         | [a-z]
```

The recursion tuples are:
1. (x_1, y_1) = (IF, THEN if_expr)
2. (x_2, y_2) = (IF if_expr THEN, ELSE if_expr)

Call symbol must be in x_1 =(IF) and x_2 = (IF if_expr THEN), and return symbol must be in y_1 = (THEN if_expr) and y_2 = (THEN, ELSE if_expr)

C2VPG: Order-Based Tagging

How to find such order M? C2VPG has two steps:

2. Backtrack all possible orderings in recursion tuples.

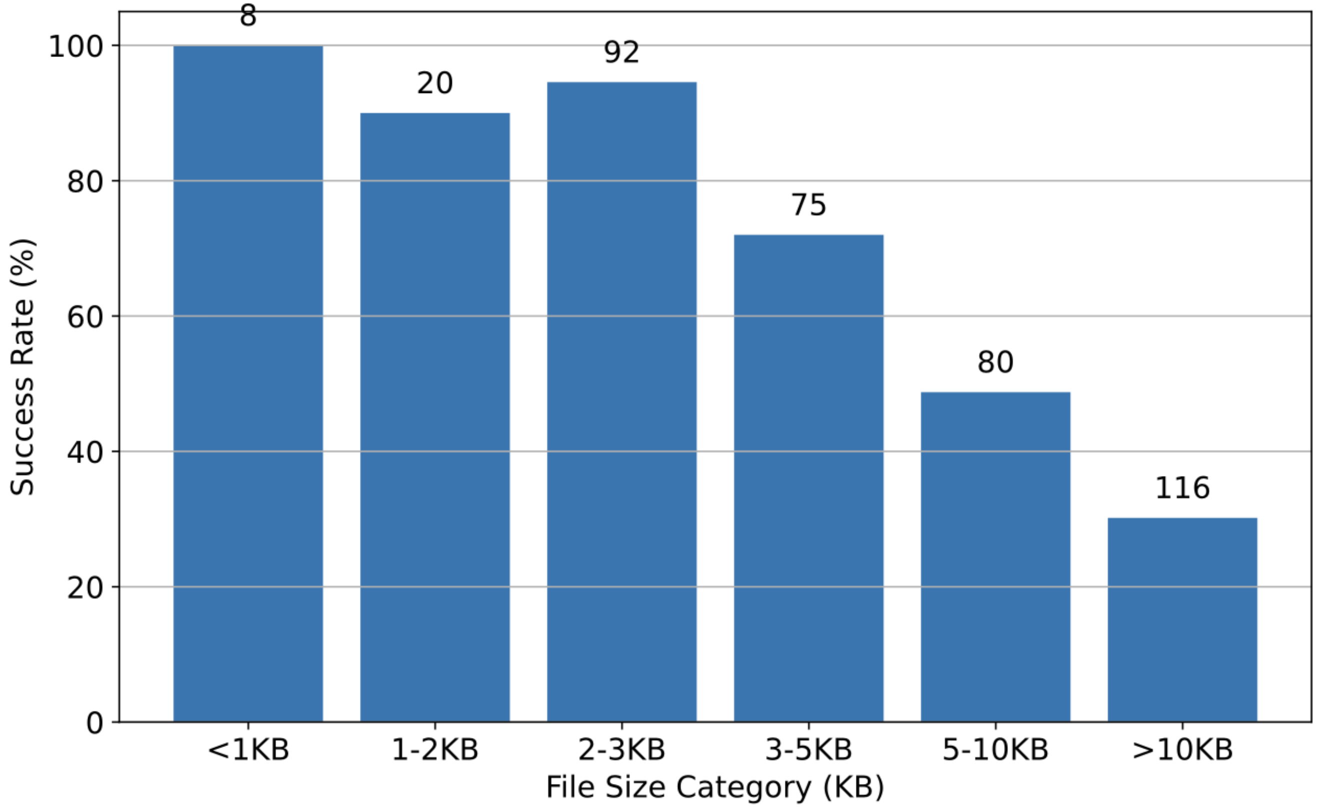
- From $(x_1, y_1) =$ (IF, THEN if_expr), select (IF,THEN)
- Check if all recursion tuples have call-return pair in x and y:
  - For $(x_2, y_2) =$ (IF if_expr THEN, ELSE if_expr)
  - Must select a return; select ELSE
  - But IF has been paired with THEN
  - So, add order THEN < ELSE

TABLE I
THE STATISTICS OF ALL 396 ANALYZED ANTLR GRAMMARS.

Type	Number	Ratio
Succeed	241	61%
Failed	150	38%
Out of Time (2 hours)	5	1%

TABLE II
THE RUNNING TIME OF THE TAGGING-INFERENCE FUNCTION ON ANALYZABLE ANTLR GRAMMARS.

Type	Number	Min (s)	Max (s)	Average (s)	Median (s)
Succeed	241	0.77	0.00	162.33	0.01
Failed	150	73.54	0.01	4165.08	0.10



Advantage:
For small to medium sized grammars, C2VPG can find the tagging efficiently with a high likelihood.

Discussion

Why C2VPG fails to identify a suitable tagger:

- Return symbols containing multiple tokens. In `Ada2005`, `IF` is paired with `END IF`.
- Call symbols embedded in nonterminals. In `Modelica`:

```
class_definition
: ('encapsulated')? class_prefixes
     ... 'end' ;
```

Discussion

Why C2VPG fails to identify a suitable tagger:

- Hard-to-detect regular sub-grammars. In `asmMASM`:

```
expression
: number 'DUP' expression ('+'
    number 'DUP' expression)*
| number ;
```

The CFG sub-grammar is equivalent to

```
(number 'DUP')* number ('+' (number
     'DUP')* number)*
```

Conclusion and Future Work

- When existing, efficient tagging functions can often be found quickly in practice.
- However, practical grammar features bring further challenges for CFG-to-VPG conversion.
- In general, how far can we push the boundary of “efficient” tagging functions?
- The tagging function brings a different aspect of parsing.

Thank you!