A BNF grammar extension designed for string generation

Devin Yeung

Agenda

Intro to BNF grammar

- Basic Term & How reduce works
- Identify the common pitfalls in BNF grammar

Extending the grammar

- Extend the grammar with regular language
- Extend the grammar with weights and invoke limits
- Extend the grammar with typing and identifier tracking

Beyond the grammar extension

- Explain the semantic analysis

Why do we want a grammar extension for generation

How do we check the implementation of parser is conform to the BNF grammar?

We need a generator to generate strings from the grammar!

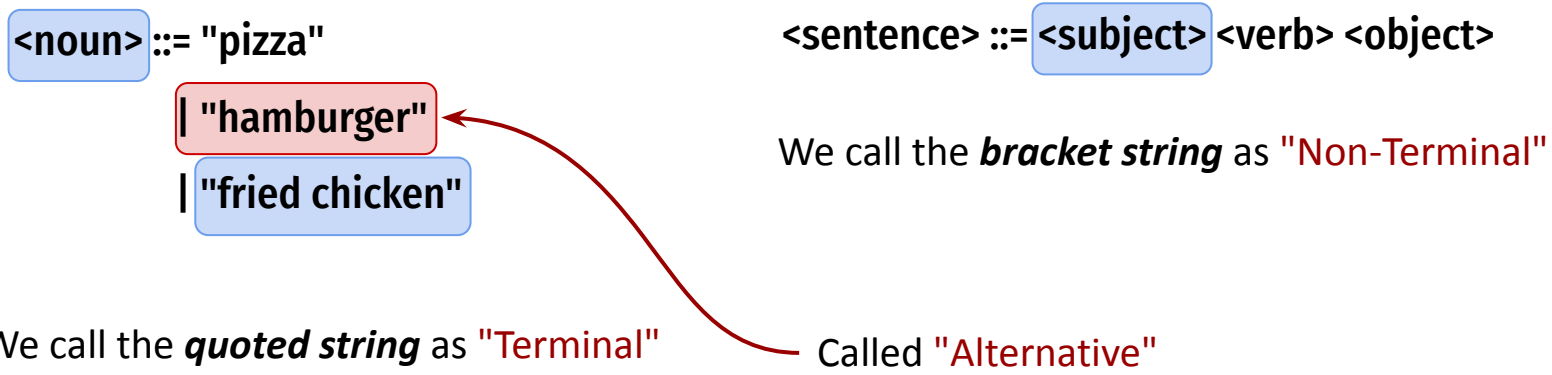
Furthermore, can we add more constraints to the generated string?

- Typing ?
- Generate more on the part we are interested in?

Terminology

They are basically only 4 key terms in BNF grammar

The name of **production**



BNF: Generation Perspective

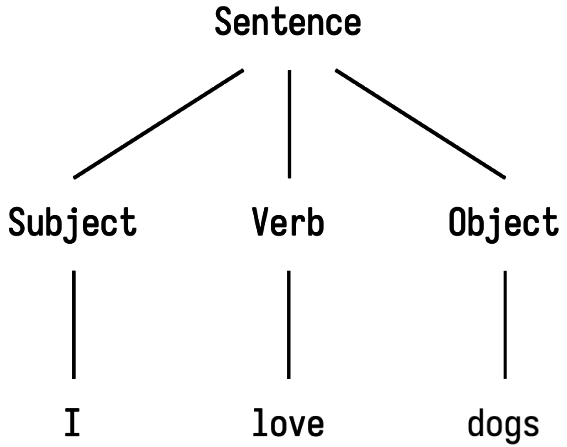
String generation with BNF grammar is constructing a tree!

<sentence> ::= <subject> <verb> <object>

<subject> ::= "I" | "you" | "he" | "she"

<verb> ::= "like" | "love" | "hate"

<object> ::= "cats" | "dogs" | "books"



We call this step "reduce"

Observation

BNF grammar is powerful in “describing” string

but not suitable for “generating” string.

Pitfalls in BNF grammar

The generation of recursive rules is hard to control

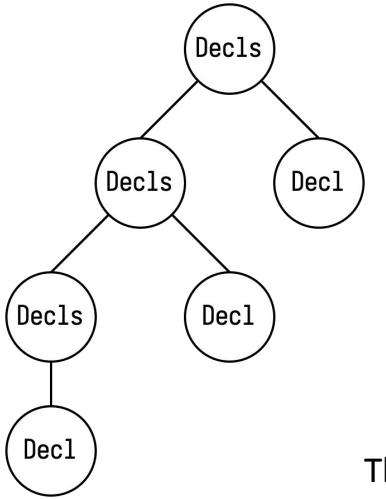
Consider the following rule:

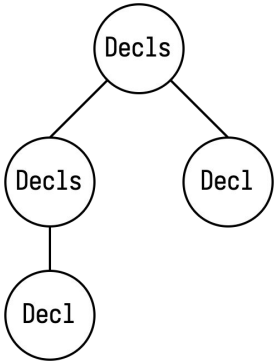
<decls> ::= <decl>

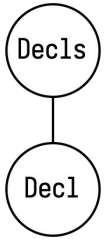
| <decls> <decl>

What if the generator always choose the 2nd alternative?

Infinite loops happens !







These trees are all valid according to the rule

Assuming that <decl> can be reduced to terminals through multiple steps

Pitfalls in BNF grammar

Some BNF rule may lead to infinite loop in generation

Consider the following rule:

```
<E> ::= <D> | <F>;
<C> ::= <D> ;
<D> ::= <C> ;
<F> ::= <G> ;
<G> ::= <F> | "Terminal" ;
```

If we choose <F> in generation, it seems we are in a loop, but we can escape from <G> produce "Terminal"

However, if we choose <D> in generation, it will be a disaster! Because it has no way out! It will loop forever

We call this situation "trap loop"

In convention, the first line of BNF grammar is the start symbol

Pitfalls in BNF grammar

BNF lacks maintainability (when expressing complex terminal)

Consider representing a identifier which *starts with alphabets or underscore,*

follow with alphabets, underscore or numbers

<alphabet> ::= "a" | "b" | … | "y" | "z" |
"A" | "B" | … | "Y" | "Z"

<id> ::= <char0> | <id> <char1>

<digit> ::= "0" | "1" | … | "8" | "9"

<char0> ::= <alphabet> | "_"

<char1> ::= <alphabet> | "_" | <digit>

We use 5 rules just to define an identifier!

It produce an complex non-terminal (in tree perspective) instead of a terminal

The … is for representation only, it's not a valid BNF grammar

Design Goal of BNF extension & generator

- The grammar should be friendly to generation (opposed to parser)
- The grammar should be ergonomic to used
- Any invalid patterns should be catch as early as possible with human readable error message
- Extend the semantic of BNF grammar (to generate more complex string)

*Note that we extend the **semantic** of the grammar, but not the **computation power** of the grammar.*

*The computation power of the extended grammar is still **context-free***

Define the BNF grammar

BNF grammar is self-described

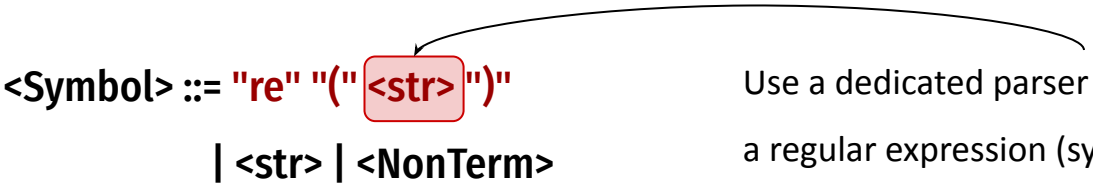
```
<Grammar> ::= <Rule> | <Grammar> <Rule>        ;  <Rule_1> <Rule_2> ... <Rule_n>
<Rule>    ::= "<" <id> ">"  "::=" <Alts> ";"   ;
<Alts>    ::= <Alt> | <Alts> <Alt>             ;  <Alt_1> <Alt_2> ... <Alt_n>
<Alt>     ::= <Symbols>                        ;
<Symbols> ::= <Symbol> | <Symbols <Symbol>     ;  <Symbol_1> <Symbol_2> ... <Symbol_n>
<Symbol>  ::= <str> | <NonTerm>                ;
<NonTerm> ::= "<" <id> ">"                     ;
```

<str> is quoted string. Since it's too complicated to represent in BNF, we didn't put it on the presentation.

Extending BNF grammar

Enable BNF grammar with the power of regular language

Add a new **regex** variant to the symbol rule



Use a dedicated parser to parse this string as a regular expression (syntax tree)

Regex almost acts as same as non-terminal, but with much powerful pattern matching power

The <id> can be represent equivalently using regex

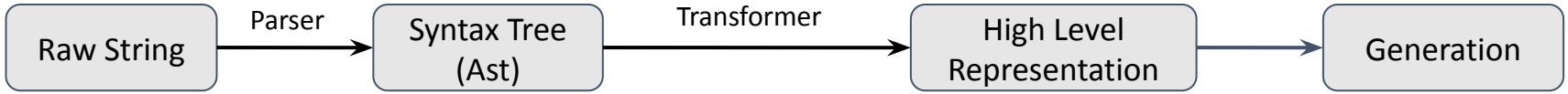
```
<id> ::= re("[a-zA-Z_][a-zA-Z0-9_]*")
```

Much neat and compact this time!

Extending BNF grammar

How to deal with regex in generation?

The raw regex string will be converted to HIR for further generation



Empty | Flags | Literal | Dot | Assertion | ClassUnicode | ClassPerl | ClassBracketed | Repetition | Group | Alternation | Concat

Empty | Literal | Repetition | Class | Concat | Alternation

Extending BNF grammar

Introduce Invoke Limit to solve the uncontrolled recursive generation

We use a <Limit> term to indicate the invoke limit of a certain **alternative**

```
<Alt>      ::= <Symbols>
             | <Symbols> <Limit>

<Limit>    ::= "{" <num> "}" | "{" <num> "," "}"
             | "{" <num> "," <num> "}"
```

{ 10 } means it must be invoked at exact 10 times

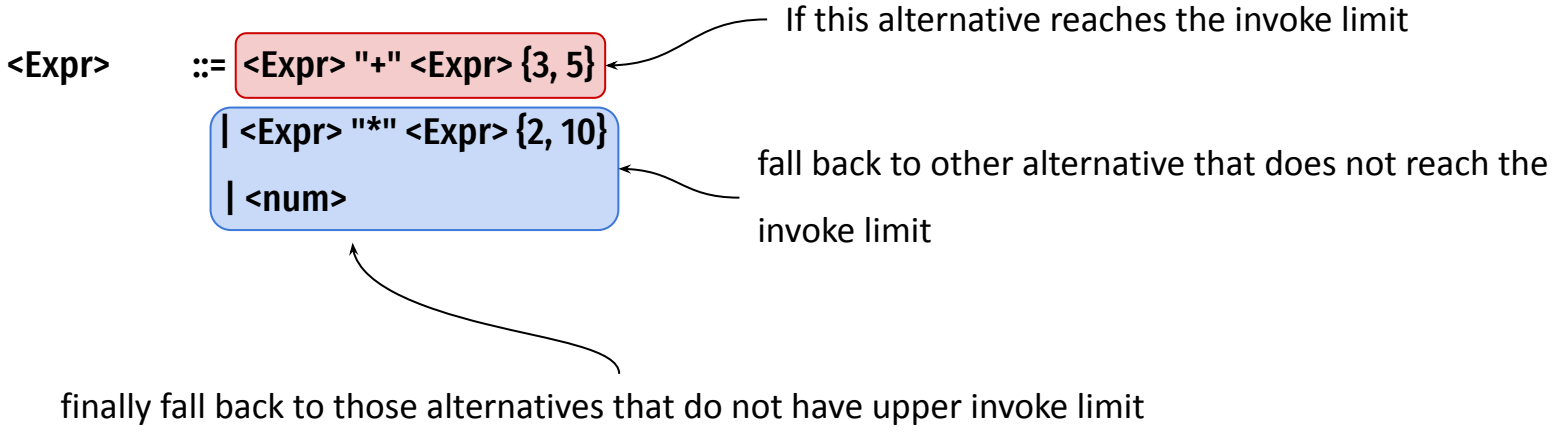
{ 2, 5 } means it must be invoked at least 2 times (inclusive) and at most 5 times (inclusive)

{ 5, } means it must be invoked at least 5 times (inclusive), but there's not upper limit.

Extending BNF grammar

Why it solves the problem?

We use a <Limit> term to indicate the invoke limit of a certain **alternative**



Extending BNF grammar

Enhancement: Weighted Alternatives

We use a <Weight> term to distribute different weights on different alternatives

```
<Alt>    ::= <Symbols>
           | <num><Alt> <Symbols> <Limit>
```

The weight of this Alternative

```
<Expr>   ::= 1 <Expr> "+" <Expr>
           | 1 <Expr> "*" <Expr>
           | 10 <num>
```

In this case, we have larger chance to choose the <num> alternative

But we still got a chance to choose the first two alts because of RNG

Note that the weighted alternative is fully compatible with invoke limit we introduced before.

We does not show the full-grammar for presentation purpose.

Extending BNF grammar

Increase the expressiveness: Introduce typing

Each **Production Name** and **Non-Terminal** can be typed

```
<Rule>      ::= "<" <id> ">"  "::=" <Alts> ";"
              | "<" <id> <Typed> ">" "::=" <Alts> ";"
<NonTerm>  ::= "<" <id> ">"  | "<" <id> <Typed> ">"
<Typed>    ::= ":" <str>
```

Constraints that introduced by weight and invoke limit also counts. Precedence: **Type > Limit > Weight**

The behavior of generator:

- A Non-Terminal **without type** can be reduced to symbol with **any type** or symbol **without type**
- A Non-Terminal **with type** will
  1. Try to reduced to the symbol with same type
  2. If no typed symbol matched, select a symbol without type as fallback
  3. If nothing left, panic!

Extending BNF grammar

Real-world example from COMP3043

```
<Expr0>    ::= "{" <Id> ":" <Predicate> "}"
             | "(" <Expr> ")" |  <Num> | <Id> ;

<Expr1>    ::= <Expr0>
             | <Expr1> "|" <Expr0>
             | <Expr1> "*" <Expr0> ;
```

```
<Expr>     ::= <Expr1>
             | <Expr> "U" <Expr1>
             | <Expr> "+" <Expr1>
             | <Expr> "-" <Expr1> ;
```

Extending BNF grammar

Real-world example from COMP3043

```
<Expr0>     ::= "{" <Id> ":" <Predicate> "}"
              | "(" <Expr> ")" |  <Num> | <Id> ;

<Expr1>     ::= <Expr0>
              | <Expr1> "|" <Expr0>
              | <Expr1> "*" <Expr0> ;
```

```
<Expr0: "set">   ::= "{" <Id> ":" <Predicate> "}" ;

<Expr0: "int">   ::= "(" <Expr: "int"> ")"
                   | <Num> | <Id> ;

<Expr1: "int">   ::= <Expr0: "int">
                   | <Expr1: "int"> "*" <Expr0: "int"> ;

<Expr1: "set">   ::= <Expr0: "set">
                   | <Expr1: "set"> "U" <Expr0: "set"> ;
```

Extending BNF grammar

Properties of Typed BNF grammar

The entire typing feature are **opt-in** and **progressive**, which means:

- You can use other parts of the extension without typing, vice versa, you can also just use typing and don't use weighted alternative or invoke limits.

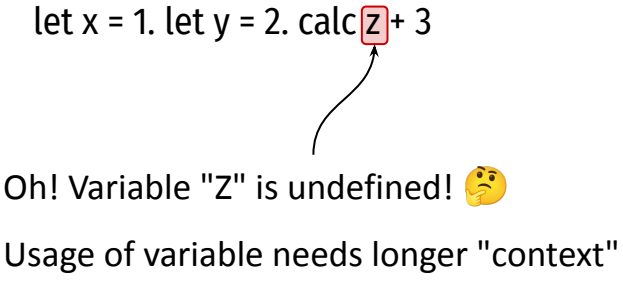
- Typed BNF generator does not required everything to be typed to make it works. Instead, how much you typed, how much it can guarantee. **(Partially typed)**

Partially typed is extremely useful, because in our experiment on Set-Algebra language, we only need to type a small proportion of grammar, then everything is typed !

Last mile to perfection: undefined variable

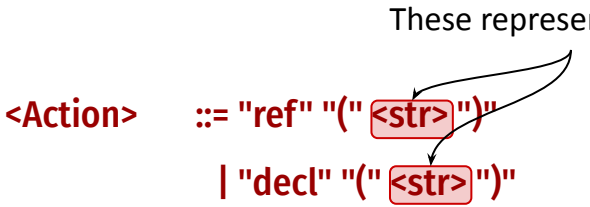
When variable comes in, everything becomes complicated!

```
<Prog>      ::= <Decl> "calc" <Expr>
<Decls>     ::= <Decl> | <Decls> <Decl>
<Decl>      ::= "let" <id> "=" <num> "."
<Expr>      ::=  <Expr> "+" <Expr>
                | <Expr> "*" <Expr>
                | <num> | <id>
<id>        ::= re("[a-zA-Z]*")
```



Last mile to perfection: undefined variable

Introduce "Action" to solve complex context related dependencies



"decl" action will insert an identifier with type ***t*** into the symbol table

```
<NonTerm> ::= "<" <id> ">"
            | "<" <id> <Typed> ">"
            | "<" <id> <Action> ">"
```

"ref" action will randomly select an identifier with type ***t*** from the symbol table

Last mile to perfection: undefined variable

Introduce "Action" to solve complex context related dependencies

```
<Decl>      ::= "let" <id: decl("int")> "=" <num> ";"
<Expr>      ::= <id: ref("int")>
<id>        ::= re("[a-zA-Z]*")
```

When reducing <id>, the generator will use re("[a-zA-Z]*") to generate a random string ***s***. Then, insert ***s*** to the symbol table with type ***int*** and symbol name ***id***

Looks good! But do we actually solve the problem? 🤔

When reducing <id>, the generator will retrieve symbol with type ***int*** and symbol name ***id*** from the table

Last mile to perfection: undefined variable

What's the problem in this example?

```
<Decl>      ::= "let" <id: decl("int")> "=" <Expr>"."
<Expr>      ::= <id: ref("int")>
<id>        ::= re("[a-zA-Z]*")
```

Cyclic Reference may happen!

If we reduce from right to left, it fine

But if we reduce from left to right? It may produce:

let x = x. calc x

Because **<id: decl(..)>** goes before the reduction of **<Expr>**

```
<Action>    ::= "ref" "(" <str> ")"
              | "decl" "(" <str> ")"
              | "decl_defer" "(" <str> ")"
```

The semantic we want:

The declaration of variable happens **at the end of the reduction of the entire production**

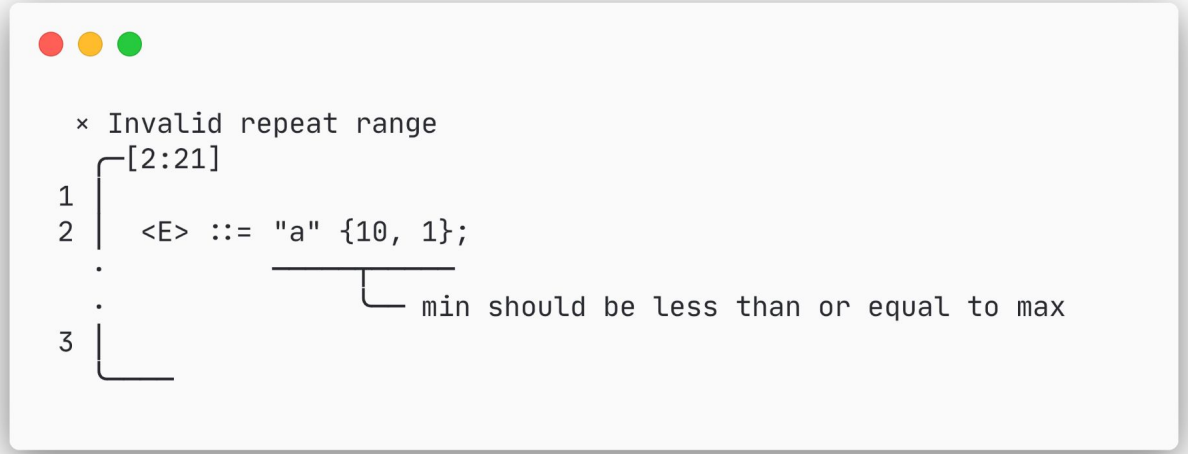
Finalized BNF grammar

```
<Grammar> ::= <Rule> | <Grammar> <Rule>                ;
<Rule>    ::= "<" <id> ">"  "::=" <Alts> ";"
            | "<" <id> <Typed> ">" "::=" <Alts> ";"    ;
<Alts>    ::= <Alt> | <Alts> <Alt>                     ;
<Alt>     ::=  <Symbols>  | <num> <Symbols>
             | <Symbols> <Limit>
             | <num> <Symbols> <Limit>                 ;
<Symbols> ::= <Symbol> | < Symbols> <Symbol>           ;
<Symbol>  ::= <str> | <NonTerm> | <Regex>              ;
<NonTerm> ::= "<" <id> ">"  | "<" <id> <Suffix> ">"    ;
```

```
<Regex>   ::= "re" "(" <str> ")"                       ;
<Limit>   ::= "{" <num> "}" | "{" <num> "," "}"
            | "{" <num> "," <num> "}"                  ;
<Suffix>  ::= <Typed> | ":" <Action>                   ;
<Typed>   ::= ":" <str>                                ;
<Action>  ::= "ref" "(" <str> ")"
            | "decl" "(" <str> ")"
            | "decl_defer" "(" <str> ")"               ;
```

Beyond the grammar and generation

To achieve the goal of making it ergonomic to user, we make great effort in semantic analysis



This is the real input from our tool! We use ASCII Art to maximize the readability of error message

Beyond the grammar and generation

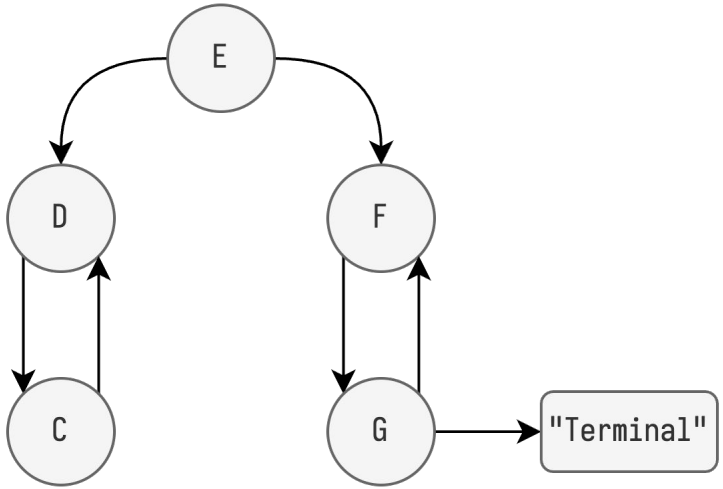
Solve the problem of "trap loop"

BNF grammar, in fact, is a directed graph

```
<E> ::= <D> | <F>;
<C> ::= <D> ;
<D> ::= <C> ;
<F> ::= <G> ;
<G> ::= <F> | "Terminal" ;
```

Each non-terminal becomes a vertex

Each production rule creates edges between vertices



Beyond the grammar and generation

Steps to identify the trap loop

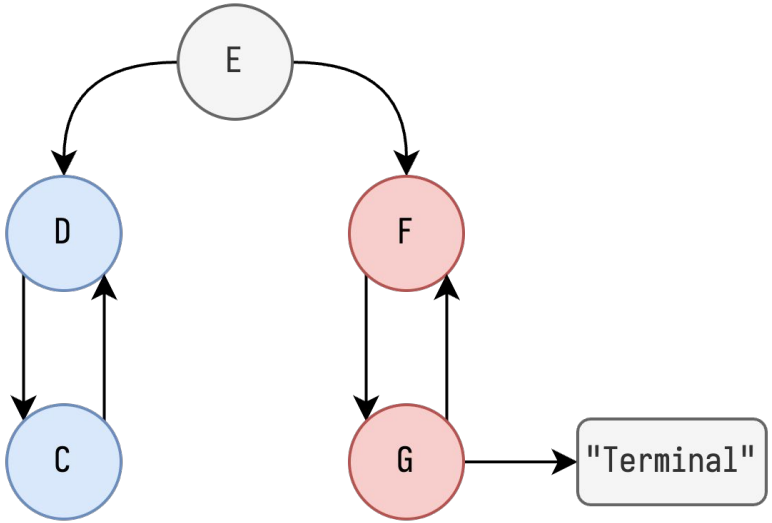
1. Find the strongly connected components in the graph
2. For each SCC, check if:

   a) It contains more than one vertex (forms a cycle)

   b) Has no escape to a terminal

{F, G} is not a trap loop because:

- ❏ Although it forms a cycle
- ❏ It has an escape to "Terminal" through G

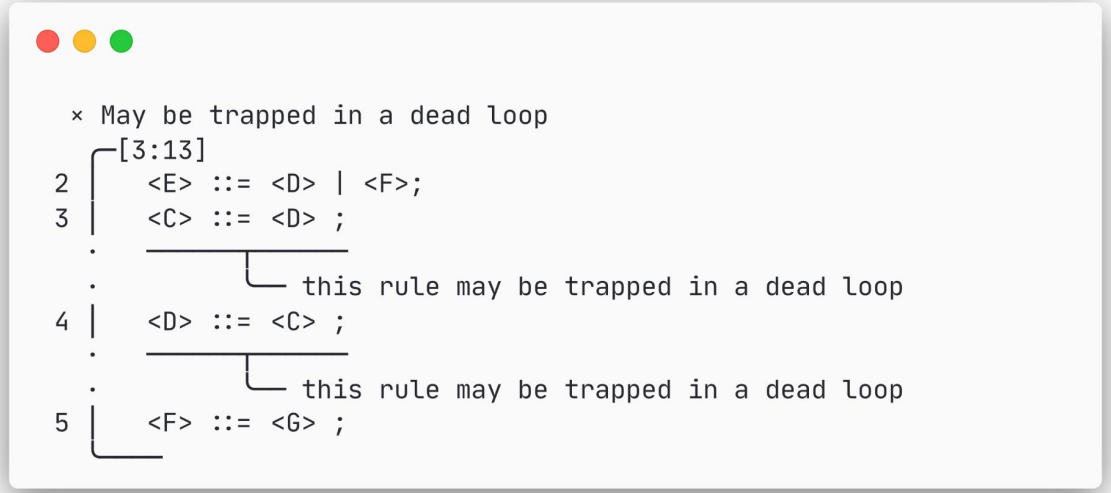


{C, D} is a trap loop because:

- ❏ Forms a cycle (C → D → C)
- ❏ Has no path to a terminal

Beyond the grammar and generation

The error message is clean, neat and human-readable



This is the real input from our tool! We use ASCII Art to maximize the readability of error message

Beyond the grammar and generation

A List of currently supported semantic analysis:

- ❑ Invalid invoke limit range detection
- ❑ Undefined rule detection (via DFS)
- ❑ Duplicated rule detection
- ❑ Unreachable rule detection (via DFS)
- ❑ Dead loop detection (which avoid the possible infinite loop in the generation)

Design philosophy: Catch errors as earlier as possible, instead of panic in runtime

Each of these problems may lead to serious panic in string generation!

Trophies

- The tool has been adopted by COMP3043 Compiler Construction to generate parser / typechecker test cases in its final project.

- The tool is public available on Github under MIT License. We hear from communities!

  https://github.com/Devin-Yeung/bnfgen

Thank you for listening