

–contents of Regular Expression Grammars–

Definition
 Self-describing Regular Expression Grammar
 Rewriting Regular Expression Grammars
 Fusion
 Removal
 Back-substituting
 Regular Expression to CFG
 Using Regular Expressions

–requires–

Context-free Grammars
 Regular Expressions
 Finite Automata

Regular Expression Grammars

In a lot of mathematical work
 the use of an appropriate notation
 makes all the difference.
 — *Edsger Dijkstra*

Definition of Regular Expression Grammars

Extending the CFG to allow regular expressions of symbols and phrase names on the right hand side of reductions makes the grammar more compact and often more readable; it does not change the set of languages that can be defined. The regular expression meta-operators for intersection and complement are excluded because they are somewhere between inefficient and impossible to implement. The extension is called a regular expression grammar (REG).

Self-describing REG

As in the self-describing CFG, ignoring whitespace, and using grammatical builtins `Letter`, `Digit`, and `Any`

```
Grammar = Rule*;
Rule    = Name '=' RegExp ';';
RegExp  = Alt* ('|' Alt)*;
Alt     = Rep ('*' | '+' | '?')?;
Rep     = Name | Input | '(' RegExp ')';
Name    = Letter (Letter | Digit)*;
Input   = ''' Any ''';
```

Table 1: A Self-describing REG

Rewriting Regular Expression Grammars

Like CFGs, REGs can be rewritten without changing the language. One use of the following rewriting rules is to transform any CFG into a REG and *vice-versa*.

<u>transform</u>	<u>condition</u>	<u>$\mathcal{T}(\Pi)$</u>
substitute	$A \leftarrow \alpha B \gamma \in \Pi \wedge B \leftarrow \beta \in \Pi$	$\Pi \cup \{A \leftarrow \alpha(\beta)\gamma\}$
combine	$A \leftarrow \alpha \in \Pi \wedge A \leftarrow \beta \in \Pi$	$\Pi - \{A \leftarrow \alpha, A \leftarrow \beta\} \cup \{A \leftarrow \alpha \beta\}$
left recursion	$A \leftarrow \alpha \in \Pi \wedge A \leftarrow A\beta \in \Pi$	$\Pi - \{A \leftarrow \alpha, A \leftarrow A\beta\} \cup \{A \leftarrow \alpha(\beta)^*\}$
right recursion	$A \leftarrow \alpha \in \Pi \wedge A \leftarrow \beta A \in \Pi$	$\Pi - \{A \leftarrow \alpha, A \leftarrow \beta A\} \cup \{A \leftarrow (\beta)^*\alpha\}$
meta-parens	$A \leftarrow (\alpha) \in \Pi$	$\Pi - \{A \leftarrow (\alpha)\} \cup \{A \leftarrow \alpha\}$

Substitute

The substitution rule from CFGs can be applied to REGs so long as meta-parentheses are added.

Combine (general form)

The rules

$$A \leftarrow \alpha_1 \quad A \leftarrow \alpha_1 \dots A \leftarrow \alpha_m$$

can be replaced with a regular expression rule

$$A \leftarrow \alpha_1|\alpha_2|\dots|\alpha_m$$

Recursion

Both of the recursion rules can also be applied in reverse to remove the Kleene '*'.

Left Recursion Removal

The rules

$$A \leftarrow \alpha_1 \quad A \leftarrow \alpha_2; \dots A \leftarrow \alpha_m$$

and the rules

$$A \leftarrow A\beta_1; A \leftarrow A\beta_2; \dots A \leftarrow A\beta_n$$

can be turned into the single rule

$$A \leftarrow (\alpha_1|\alpha_2\dots|\alpha_m)(\beta_1|\beta_2\dots|\beta_n)^*$$

REG to CFG

Given a regular expression R , the regular expression grammar (REG)

$$G \leftarrow R$$

describes the same language as the regular expression R . Using the REG transformations, the regular expression meta-symbols can be systematically removed, resulting in a CFG that still describes the same language.

Continuing, the CFG can be transformed into a left-linear form describing a DFA.

The process can be reversed. In the general case, starting from an arbitrary CFG, the reverse process will not end in a single rule.

Using REGs

The REG notation will be applied to the two main components of the compiler front end, parsing and scanning. Its obvious descriptive power is offset by not providing an obvious syntax tree. Because of this we usually need both a CFG and a REG for each computer language.

Exercise

1. Describe the X language using an REG.
2. What is the shortest self describing grammar you can devise? You may want to exclude some of the meta-symbols.