Why aren’t regular expressions enough?

review: what is a language

- A *language* is a (possibly infinite) set of strings
- Regular expressions define *regular languages*
  - All regular languages can be defined by regular expressions (and anything you can define with a regular expression is a regular language)
  - All regular languages can be recognized by a finite automaton (and anything you can recognize with a finite automaton is a regular language)

so what's beyond regular languages?

- Key consequence of correspondence between regular languages and finite automaton:
  - If a language is regular it *must be* recognizable by a finite automaton
  - If a language *cannot* be recognized by a finite automaton, it *cannot* be regular

- Consider the following piece of C code:

```
{{{ int x; }}}
```

- Need to make sure there are as many '{' as '}' — and no limit to nesting depth
  - Can you do this with a finite automaton? Can you do this with a regular expression?

key challenge

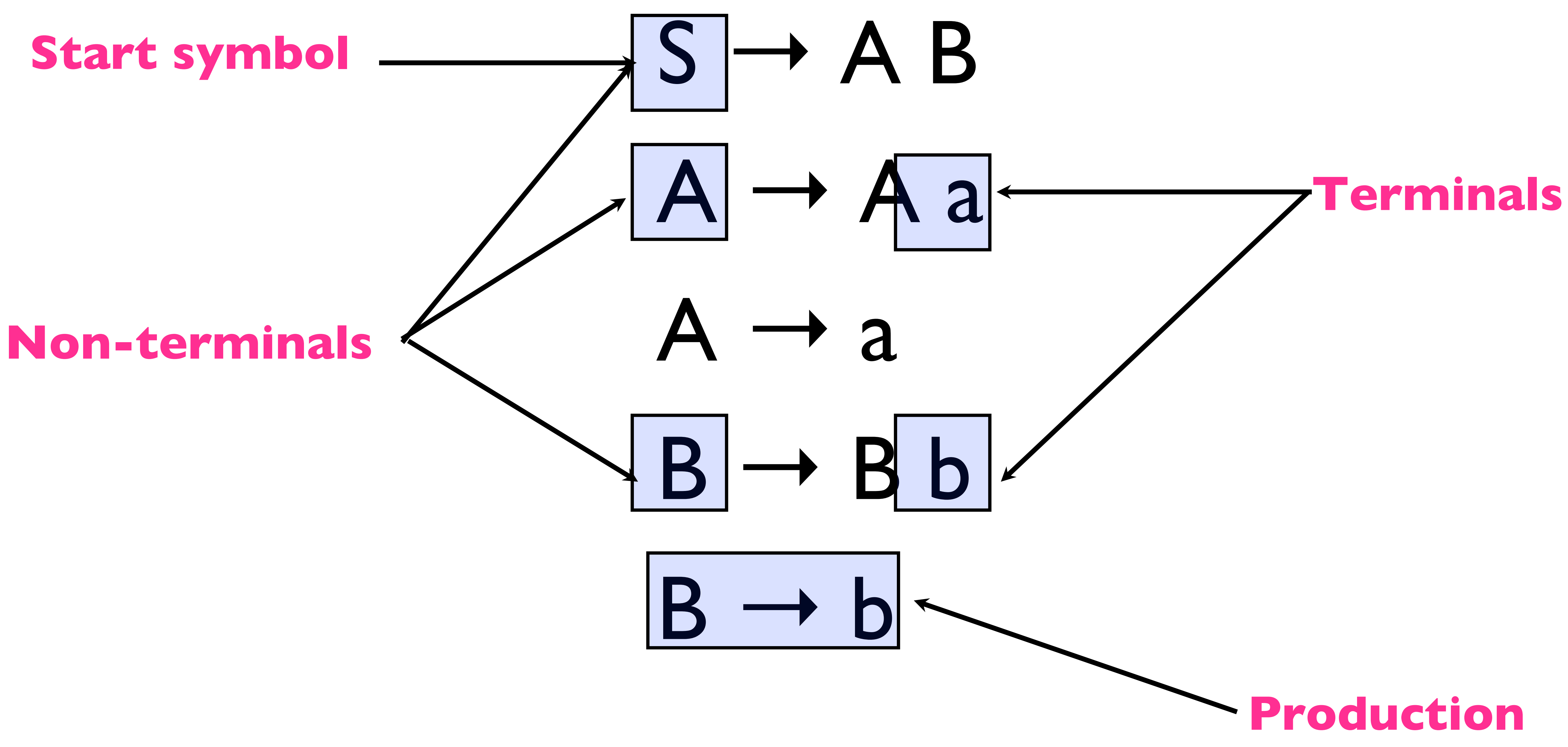
- The **structure** of a program is recursive:
  - If statements nested inside while loops nested inside for loops nested inside if statements nested inside …
  - Nesting can be arbitrarily deep
- Accounting for this kind of recursive nesting is beyond what regular expressions can do — need to keep track of how deep you are in nesting to make sure everything lines up
- Need a new kind of language formalism for specifying these types of languages: **context-free grammars**

a simple grammar

- Grammar $G = (V_t, V_n, S, P)$
- V_t is the set of terminals
- V_n is the set of non-terminals
- $S \in V_n$ is the start symbol
- P is the set of productions
- Each production takes the form:

  $$V_n \to \lambda | (V_n | V_t)^+$$

- Grammar is **context-free** (why?)



how does a grammar define a language?

- Given a start rule, productions tell us how we can *rewrite* non-terminals into other strings
- Some productions rewrite into λ. That just removes the non-terminal
- To derive the string "a a b b b" we can do the following rewrites:

$S \Rightarrow A\ B \Rightarrow A\ a\ B \Rightarrow a\ a\ B \Rightarrow a\ a\ B\ b \Rightarrow a\ a\ B\ b\ b \Rightarrow a\ a\ b\ b\ b$

$S \rightarrow A\ B$

$A \rightarrow A\ a$

$A \rightarrow a$

$B \rightarrow B\ b$

$B \rightarrow b$

terminology

- **Strings** are composed of symbols
  - A A a a B b b A a is a string
- We will use Greek letters to represent strings composed of both terminals and non-terminals
- *L(G)* is the language produced by the grammar *G*
  - All strings consisting of only terminals that can be produced by *G*
  - In our example, $L(G) = a^+b^+$
  - The language of a context-free grammar is a **context-free language**
  - All regular languages are context-free, but not vice versa

matching { and }

- So how can we use a CFG to define a language for matching braces?

$$S \rightarrow \{\, S \,\}$$
$$S \rightarrow \lambda$$

- Note that we can rewrite a non-terminal to λ to make it “disappear”

programming language syntax

- Programming language syntax is defined with CFGs
- Constructs in language become non-terminals
- May use auxiliary non-terminals to make it easier to define constructs

if_stmt → if (cond_expr) then statement else_part

else_part → else statement

else_part → λ

- Tokens in language become terminals

problem: how do we tell if a string matches a CFG?