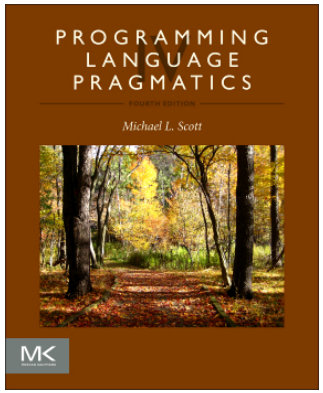
Syntax and Lexical Analysis

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 2 through section 2.2

Prof. Ben Titzer

ELSEVIER

Specifying Syntax

- Let's start by specifying the idea of a *digit*:

$$digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- From this we can build *natural numbers*:

$$non\_zero\_digit \longrightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$natural\_number \longrightarrow non\_zero\_digit\ digit*$$

- Simple concepts like these can be expressed with *regular expressions*

Regular Expressions

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
  - Two regular expressions separated by |, i.e. ***or***
  - A regular expression R followed by the Kleene star *****, i.e. ***zero or more of R***
  - We use parentheses ( ) to disambiguate grouping

Regular Expressions

- Numerical constants accepted by a simple hand-held calculator:

$$
\begin{aligned}
\textit{number} &\longrightarrow \textit{integer} \mid \textit{real} \\
\textit{integer} &\longrightarrow \textit{digit}\ \textit{digit}* \\
\textit{real} &\longrightarrow \textit{integer}\ \textit{exponent} \mid \textit{decimal}\ (\textit{exponent} \mid \epsilon) \\
\textit{decimal} &\longrightarrow \textit{digit}*\ (\ .\ \textit{digit} \mid \textit{digit}\ .\ )\ \textit{digit}* \\
\textit{exponent} &\longrightarrow (\texttt{e} \mid \texttt{E})\ (\texttt{+} \mid \texttt{-} \mid \epsilon)\ \textit{integer} \\
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

Practice with Regular Expressions

- Define a regular expression for C-style comments
  - You may use abbreviations like *non-** or *newline*
  - You may use Kleene + (1 or more) in addition to Kleene *

Practice with Regular Expressions

- Define a regular expression for C-style comments
  - You may use abbreviations like *non-** or *newline*
  - You may use Kleene + (1 or more) in addition to Kleene *

- One solution (from the textbook)

comment → /* (*non-** | * *non-*/)* *+ /

| // (*non-newline*)* *newline*

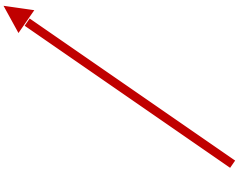
From Tokens to Grammar

- Regular expressions are great for describing *tokens*
  - The smallest meaningful units of syntax – numbers, symbols, keywords, and identifiers
  - These constructs have no interesting recursive structure
- But real programs have recursive structure, even in expressions like 2 * (x + (y /3))
- To capture higher-level syntax we need *context-free grammars*

Context-Free Grammars

- A calculator expression grammar is recursive:

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

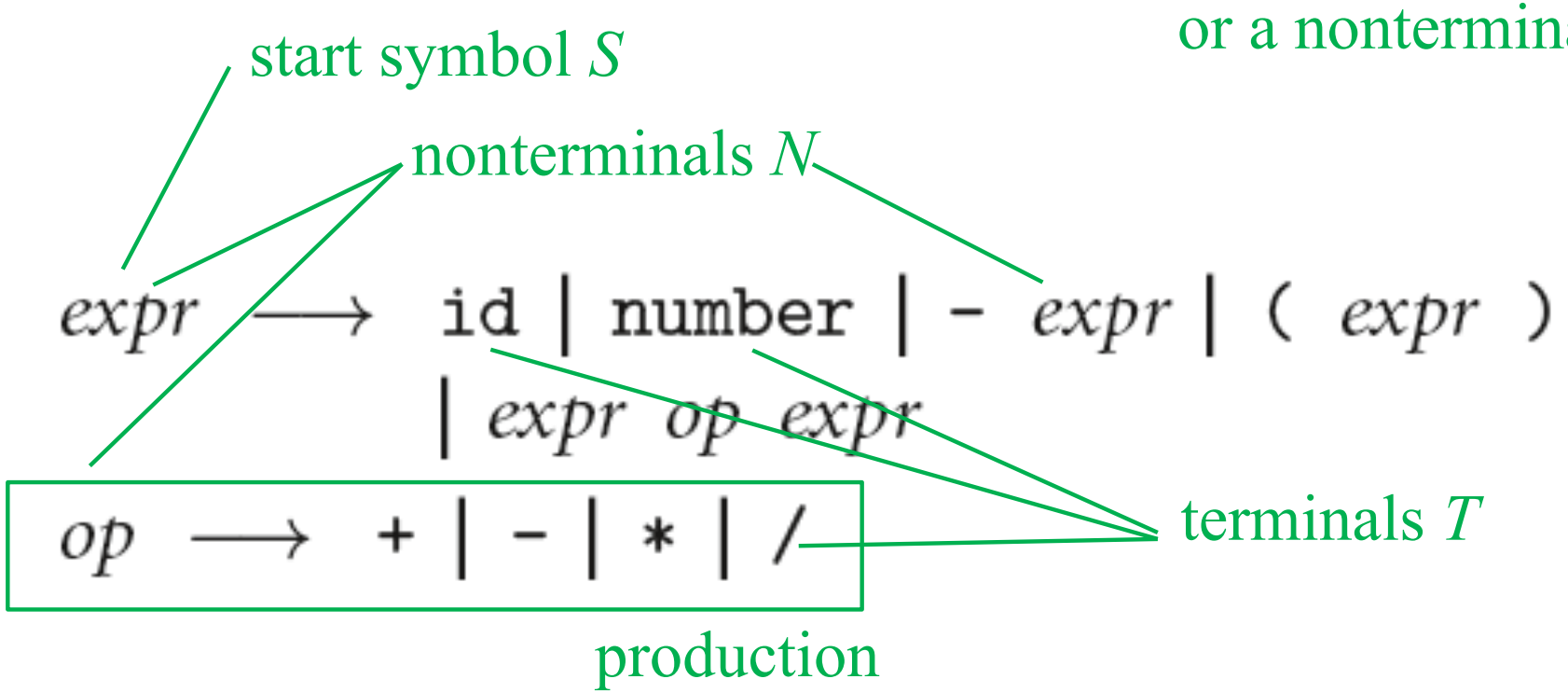


expr is defined in terms of itself!

Context-Free Grammars (CFGs)

- Anatomy of a CFG
  - In Backus-Naur Form (BNF)

A *symbol* is a terminal or a nonterminal



Context-Free Grammars

- In this grammar, generate the string **"slope * x + intercept"**

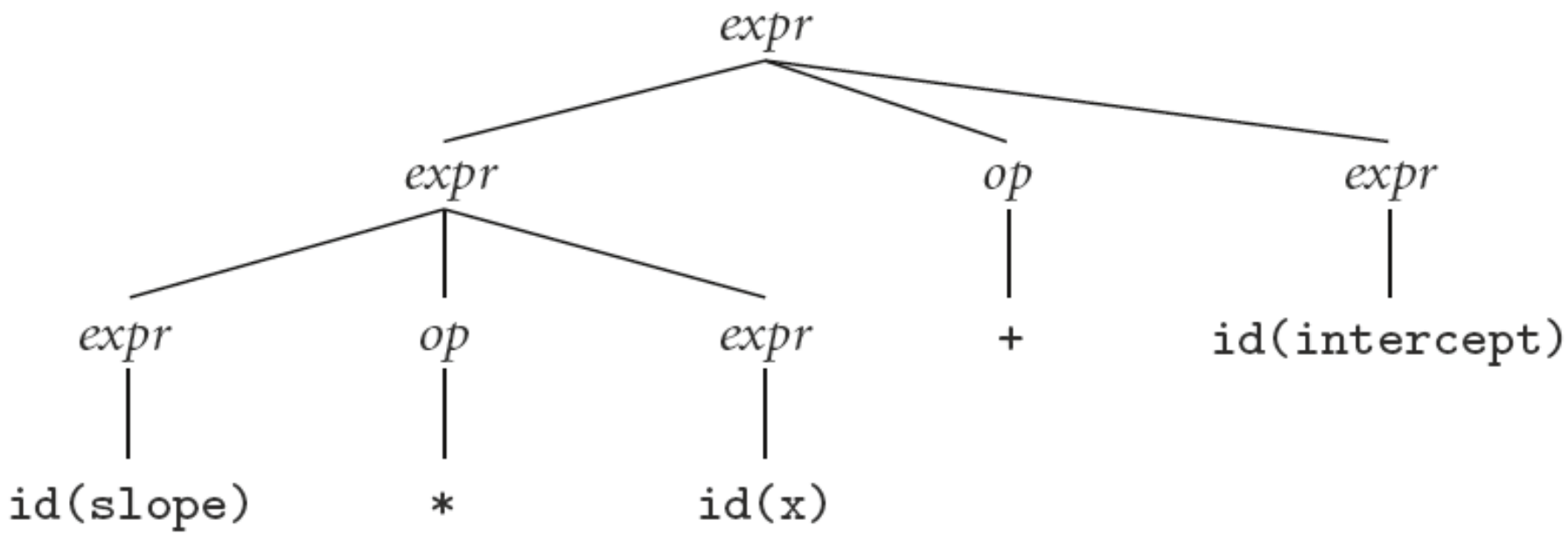
$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
expr &\Longrightarrow expr\ op\ \underline{expr} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \texttt{id}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\qquad \texttt{(slope)}\quad \texttt{(x)}\quad \texttt{(intercept)}
\end{aligned}
$$

This is called a *derivation*

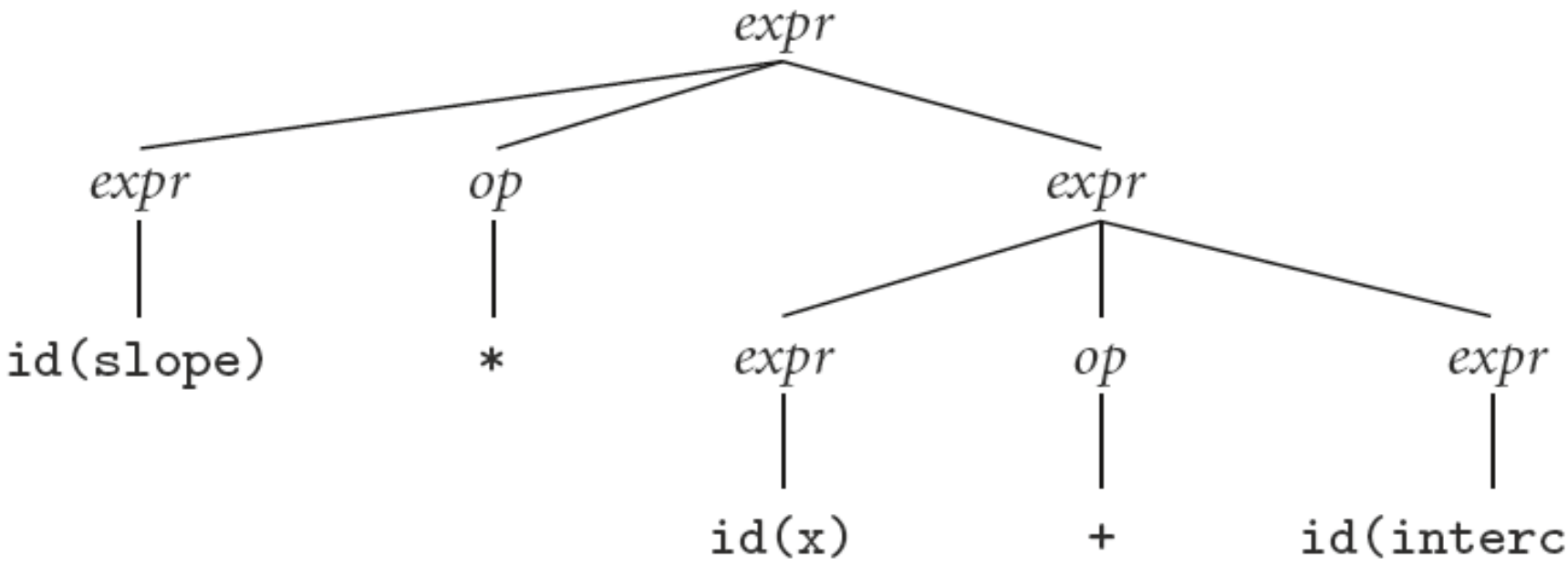
Context-Free Grammars

- Parse tree for expression grammar for **`"slope * x + intercept"`**



Context-Free Grammars

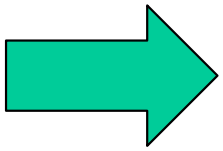
- Alternate (Incorrect) Parse tree for **`"slope * x + intercept"`**
- Our grammar is *ambiguous*



Lexical Analysis (or “Scanning”)

- Divides source code into tokens
- Removes comments
- Saves text of identifiers, strings, numbers
- Tags tokens with line numbers, for error messages

```
y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y – 1
od
```



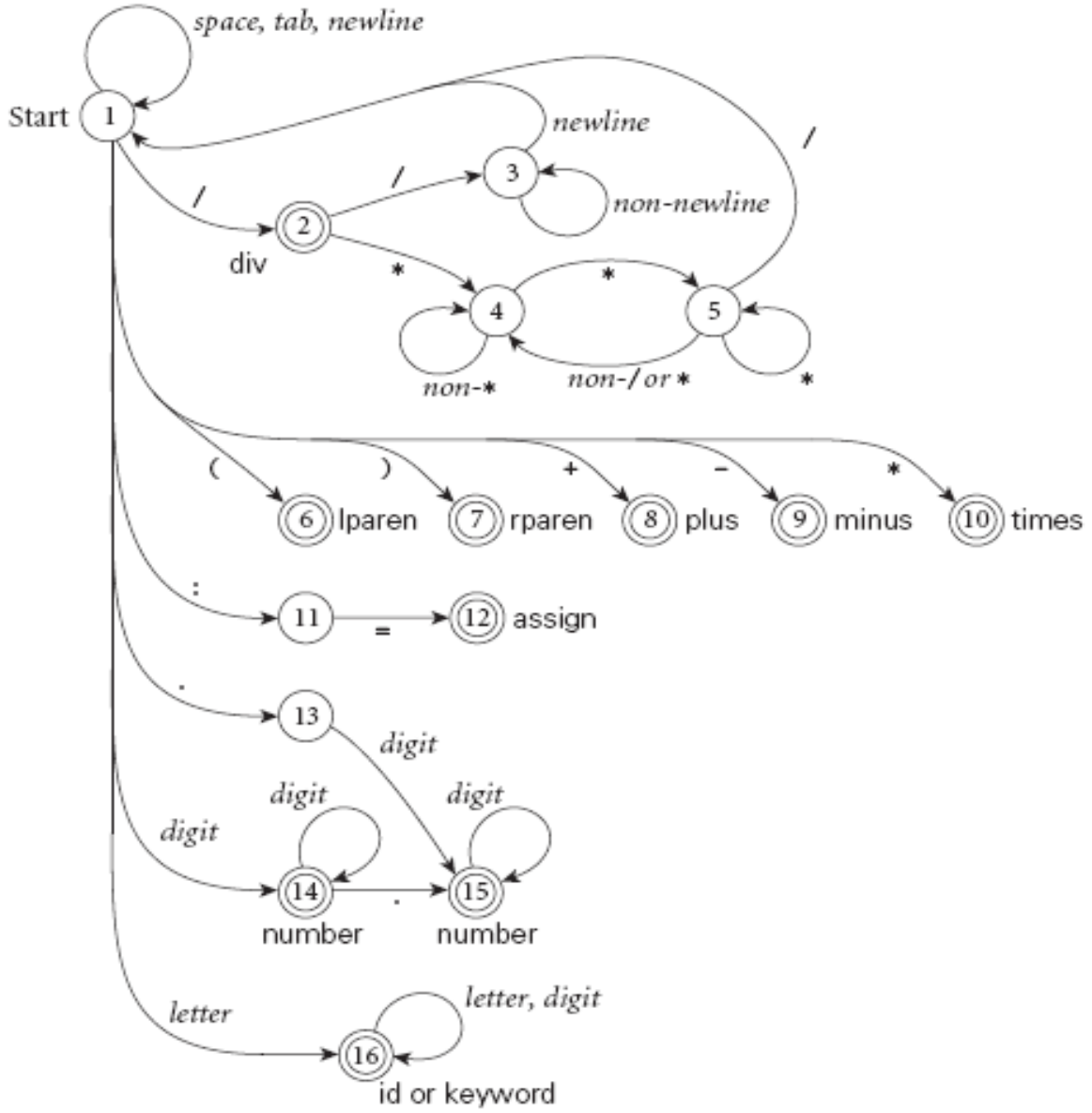
```
y := x ; z := 1 ; while y
> 1 do z := z * y ; y :=
y – 1 od
```

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for a calculator language:
  - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens
  ( ) + - * /
  we recognize it as a **token**
- If it is a digit, we keep reading digits until we can't anymore, then recognize it as a **number**
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore, then recognize it as an **identifier**

Scanning

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Scanning with floating point

- If it is a digit, we keep reading until we find a non-digit
  - if that is not a '.' we recognize an **integer**
  - otherwise, we keep looking for a real number
  - if the character after the '.' is not a digit we announce an integer and reuse the '.' and the look-ahead

Scanning

- This is a deterministic finite automaton (DFA)
  - Lex, scangen, etc. build these things automatically from a set of regular expressions
  - Specifically, they construct a machine that accepts the language

```
identifier | int const
| real const | comment | symbol
| ...
```

Scanning

- We run the machine over and over to get one token after another
  - Longest match rule:
    - take the longest complete token from the input
    - Thus `foobar` and never `f` or `foo` or `foobar`
    - And `3.14159` is a real const and never `3, ., ` and `14159`
- Regular expressions "generate" a regular language; DFAs "recognize" it

Scanning

- Scanners tend to be built three ways
  - ad-hoc: hand-written code
  - semi-mechanical pure DFA
    (usually realized as nested case statements)
  - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use perl, awk, sed
  - for details see Example 2.16
- Table-driven DFA is what lex and scangen produce
  - lex/ocamllex in the form of C/OCaml code
  - scangen in the form of numeric tables and a separate driver (for details see Figure 2.11-2.12)

Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - In Pascal, for example, when you have a 3 and you a see a dot
    - do you proceed (in hopes of getting 3.14)?
      or
    - do you stop (in fear of getting 3..5)?

Scanning

- In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have

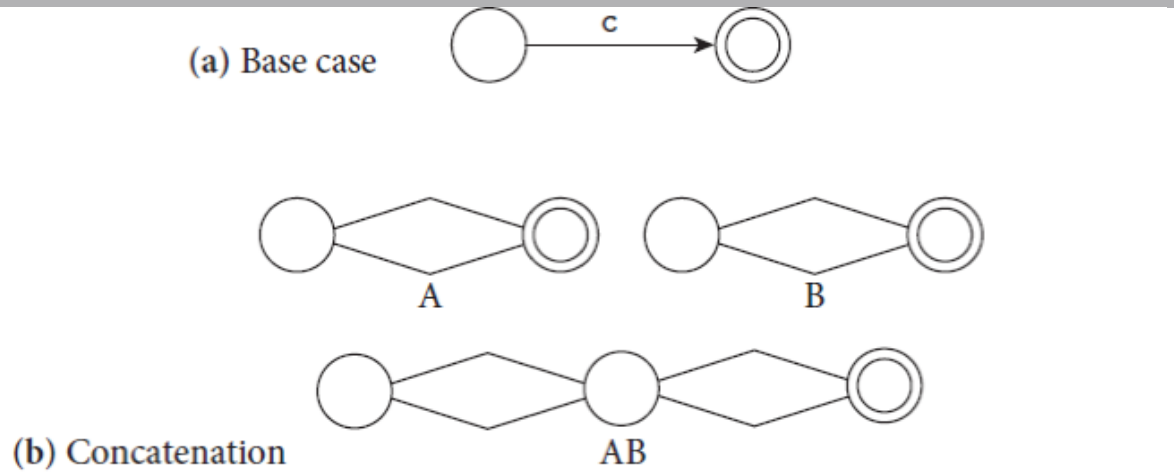
```
DO 5 I = 1,25   loop
DO 5 I = 1.25   assignment
                (to DO5I)
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Converting a RE to a DFA

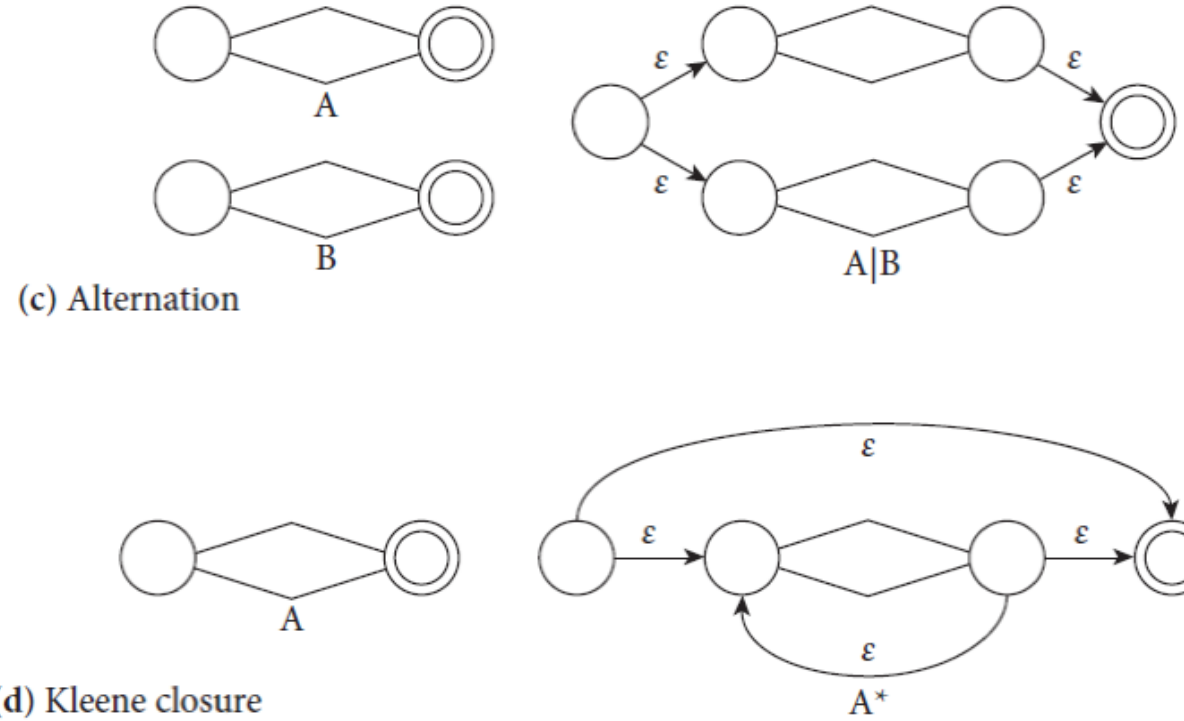
1. Write regular expressions for each construct
   - Except keywords – special case of identifiers
2. Construct NFA from REs
3. Convert NFA to a DFA (set of subsets)
4. Minimize DFA (find equivalence classes)
5. Fix up the result
   - Longest-possible token rule
   - Discard whitespace and comments
   - Distinguish keywords from identifiers
   - Save text, token location
   - Return a special EOF token at end of file

RE to NFA Construction

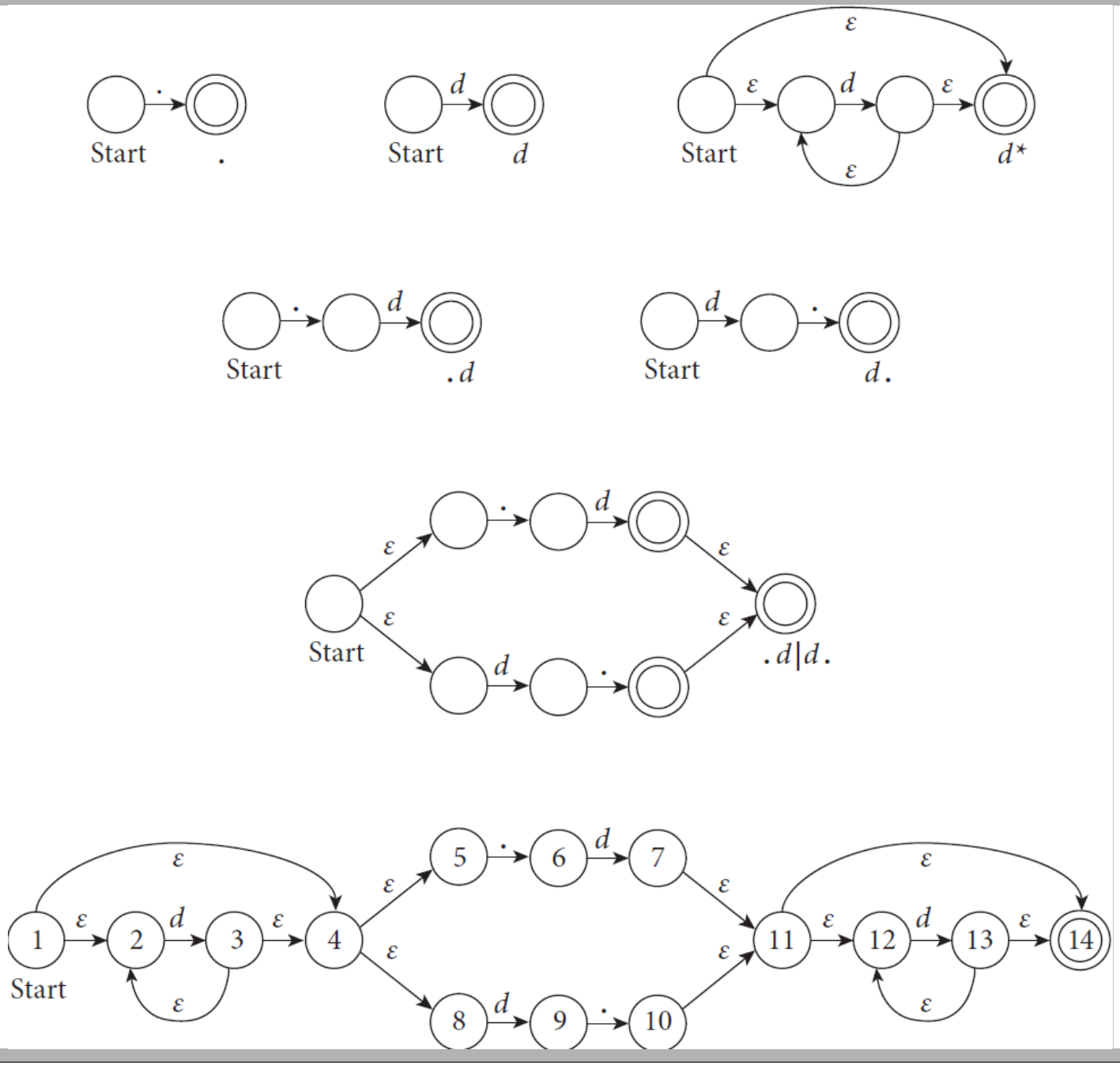


Let's apply this to

d* (. d | d .) d*



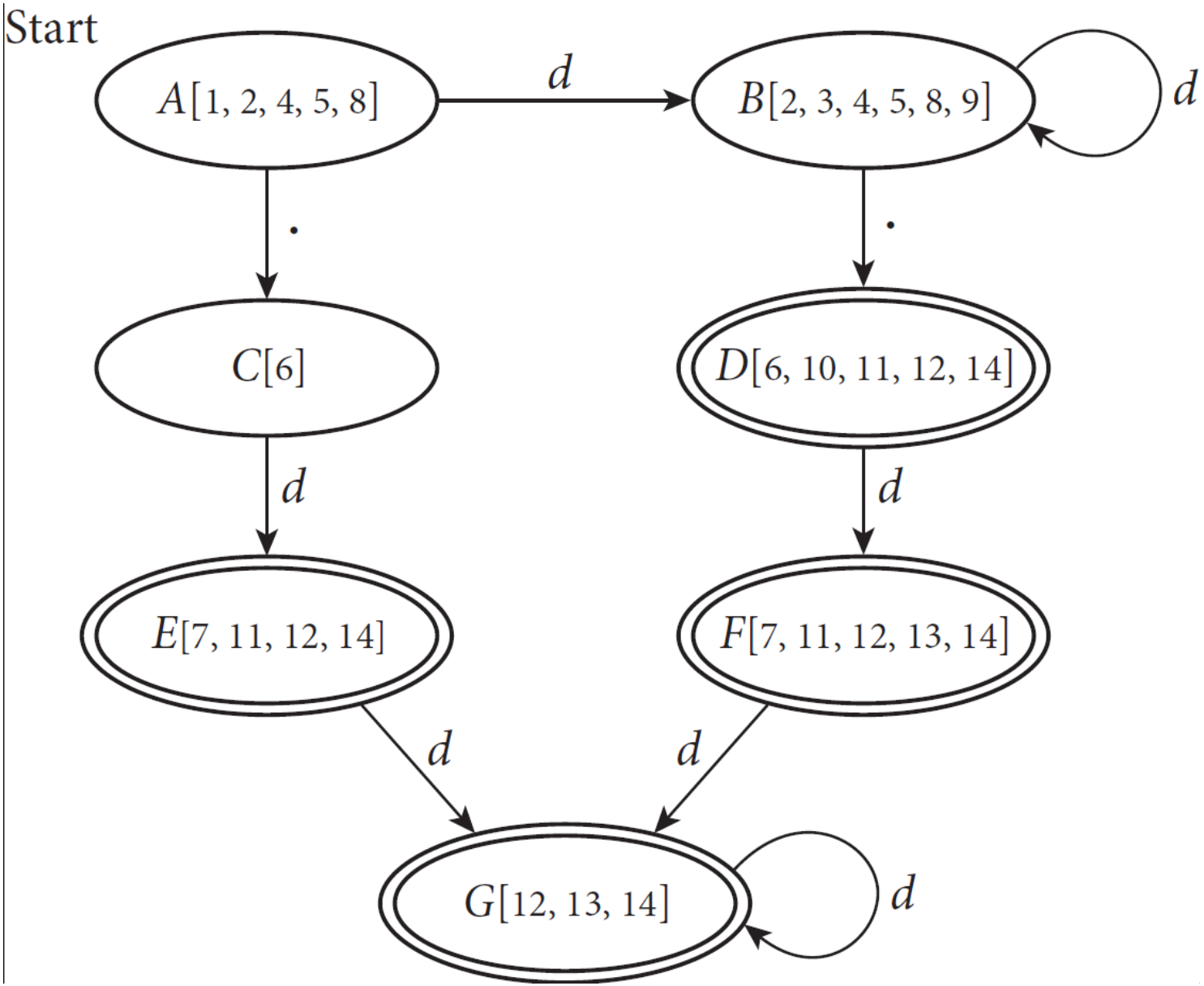
RE to NFA Construction



NFA to DFA Construction

- Each state in the DFA is a set of NFA states
  - "Set of subsets"
- The start DFA state contains the start NFA state, plus all states reachable through ²-transitions
- For each input that can be consumed from one of those NFA states, we create another DFA state with the set of destination states (plus states from ²-transitions)

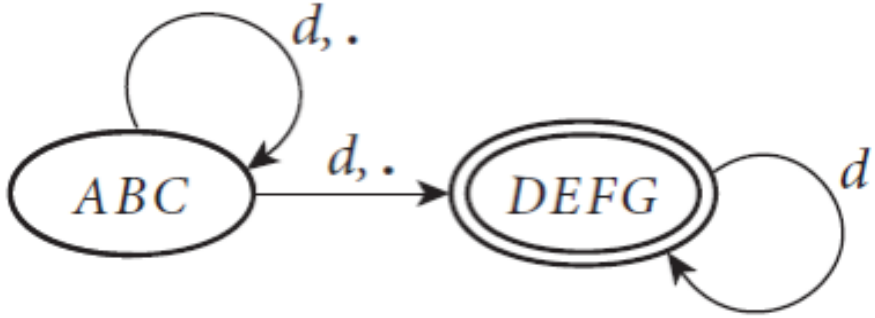
NFA to DFA Construction (example)

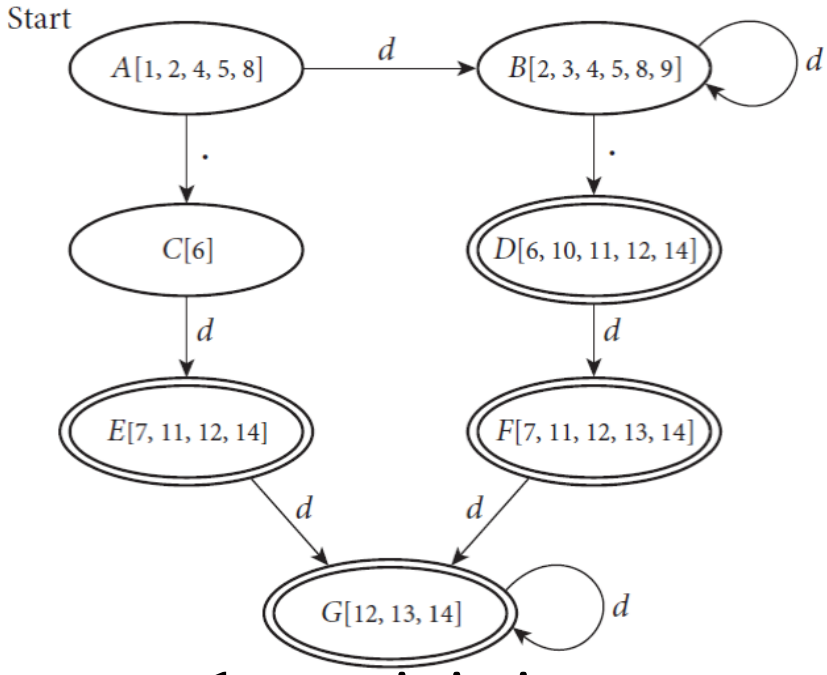


DFA Minimization

- Start by merging all DFA states into two equivalence classes: final and non-final
- Iteratively identify nondeterministic transitions and split states to avoid them

DFA Minimization





- Example: Consider the diagram on the left, derived by merging states from the one on the right.
- Transitions from ABC on both *d* and . are nondeterministic
- We can make the *d* transition deterministic by splitting into a state representing A&B and a state representing C
- Conversely, we could make the . transition deterministic by splitting into AC and B
- Let's take the first (AB and C) and proceed.

DFA Minimization (example)

- From state (b) we can now make the . transition deterministic by splitting AB into A and B.