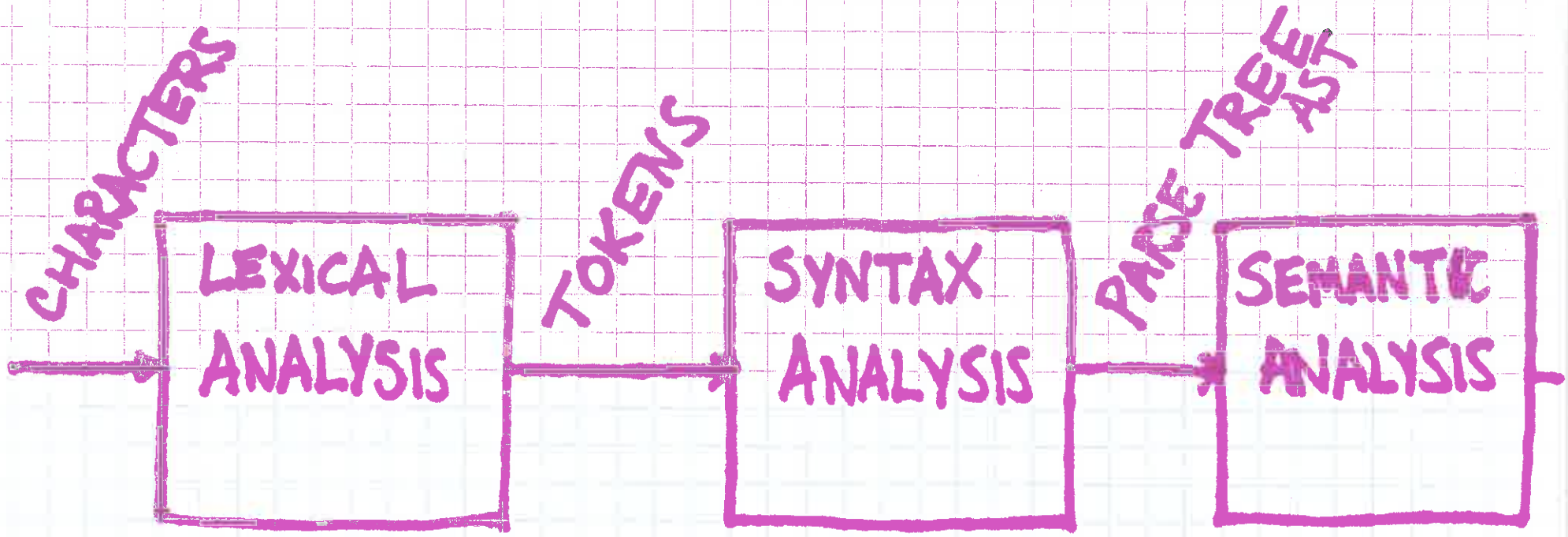
- Review Regular Expressions
- Overview of compiler construction
- Dive into Go spec
  - EBNF
  - UTF8, letters, digits
  - Lexical Elements
    - Comments
    - Semicolons
    - Identifiers
    - Keywords, operators, & punctuation
    - Literals

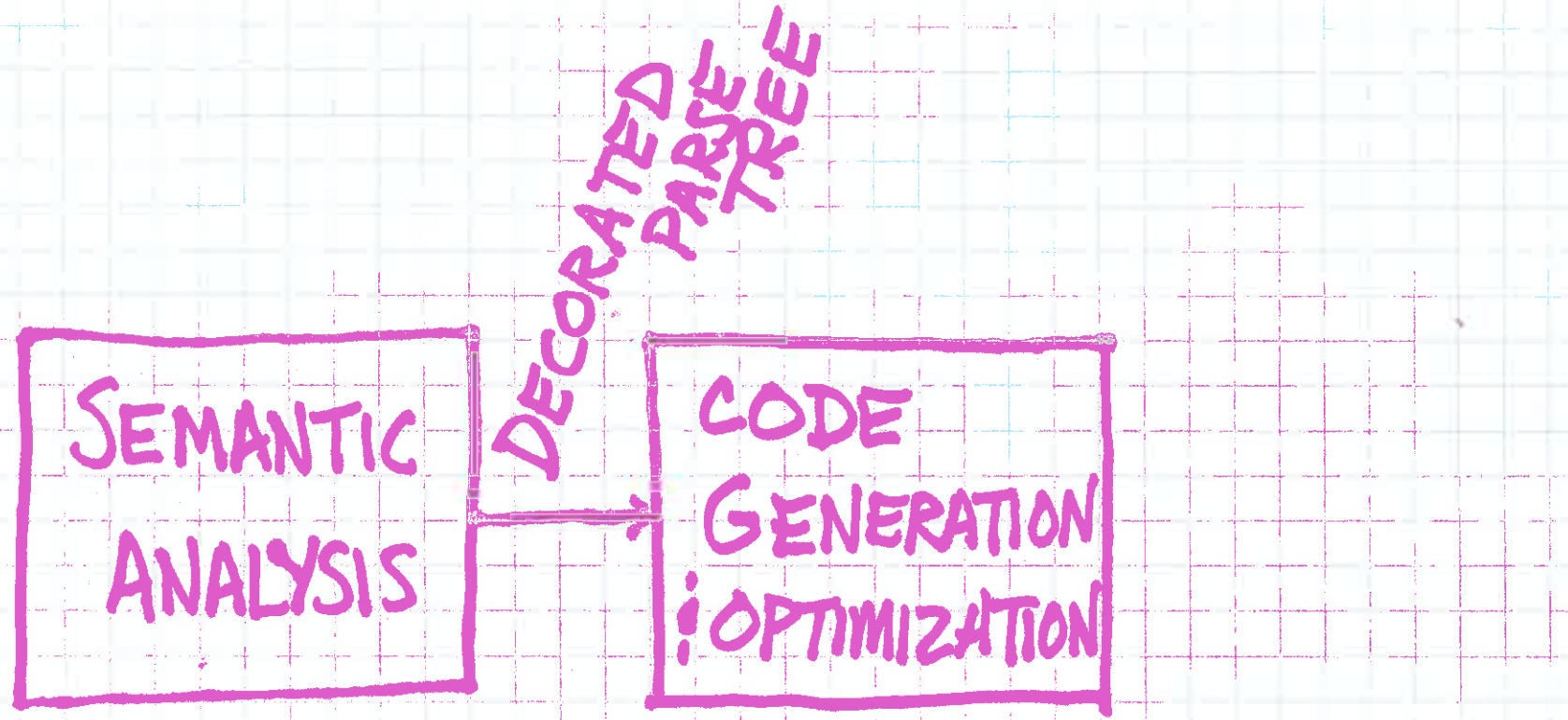
Reg. Exp.	Denotes
ε	$\{\varepsilon\}$
a	$\{a\}$
$\alpha\beta$	$\{ab \mid a \in \alpha \; b \in \beta\}$
$\alpha \mid \beta$	$\{s \mid s \in \alpha \text{ or } s \in \beta\}$
α^*	$\{\varepsilon \cup \alpha \cup \alpha\alpha \cup \alpha\alpha\alpha \ldots$
α^+	$\alpha\alpha^*$
$\alpha?$	$\alpha \mid \varepsilon$
$[a..bc]$	$a \mid .. \mid b \mid c$
$[\hat{}a..bc]$	any $d \in \Sigma$ where d is not in $[a..bc]$
$(\;)$	grouping



SCANNING
TOKENIZATION

PARSING

TYPE-
CHECKING


DECORATED PARSE TREE
SEMANTIC ANALYSIS
CODE GENERATION & OPTIMIZATION

Productions are expressions constructed from terms and the following operators, in increasing precedence:

`|`	alternation	**same as regular expressions**
`()`	grouping	**same as regular expressions**
`[]`	option (0 or 1 times)	**like ? in regular expressions**
`{}`	repetition (0 to n times)	**like * in reg. expressions**

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes `""` or back quotes ``` `` ```.

The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis … is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character … (as opposed to the three characters ...) is not a token of the Go language.

Source code representation

Source code is Unicode text encoded in UTF-8.
But you can think in and use ASCII.

The text is not canonicalized, ...

... A byte order mark may be disallowed anywhere else in the source.

The syntax is specified using Extended Backus-Naur Form (EBNF):

```
Production  = production_name "=" [ Expression ] "." .

Expression  = Alternative { "|" Alternative } .

Alternative = Term { Term } .

Term        = production_name | token [ "…" token ]
            | Group | Option | Repetition .

Group       = "(" Expression ")" .

Option      = "[" Expression "]" .

Repetition  = "{" Expression "}" .
```

Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
```

```
a
_x9
ThisVariableIsExported
αβ
```

Some identifiers are predeclared.

follow the handy link. You'll see that things like *int* and *println* are predeclared.

Characters

The following terms are used to denote specific Unicode character classes:

```
newline        = /* the Unicode code point U+000A */ .

unicode_char   = /* an arbitrary Unicode code point
                    except newline */ .

unicode_letter = /* a Unicode code point classified as
                    "Letter" */ .

unicode_digit  = /* a Unicode code point classified as
                    "Number, decimal digit" */ .
```

In The Unicode Standard 8.0, Section 4.5 "General Category" defines a set of character categories. Go treats all characters in any of the Letter categories Lu, Ll, Lt, Lm, or Lo as Unicode letters, and those in the Number category Nd as Unicode digits.

Letters and digits

The underscore character _ (U+005F) is considered a letter.

```
letter        = unicode_letter | "_" .
decimal_digit = "0" … "9" .
octal_digit   = "0" … "7" .
hex_digit     = "0" … "9" | "A" … "F" | "a" … "f" .
```

Comments

Comments serve as program documentation. There are two forms:

1. *Line comments* start with the character sequence // and stop at the end of the line.

   //[^\n]*\n alphabet in red

   regular expression *metacharacters* in green

2. *General comments* start with the character sequence /* and stop with the first subsequent character sequence */.

   /* [^*]* ([*]+ [^/] [^*]*)* [*]+/

A comment cannot start inside a rune or string literal, or inside a comment.

A general comment containing no newlines acts like a space.

```
        a := foo/* hello */+bar
becomes a := foo +bar
```

(there's an old C trick:

```
        #define real_add(x, y)      x/* */_real + y/* */_real

then    z_real = real_add(a, b);
becomes z_real = a/* */_real + b/* */_real;
which becomes  z_real = a_real + b_real;
```

because C comments act like no characters.)

Any other comment acts like a newline.

(Important for their semicolon-elimination strategy.)

Tokens

Tokens form the vocabulary of the Go language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*.

White space, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token.

Also, a newline or end of file may trigger the insertion of a semicolon.

yikes!

While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

usual rule. You don't want the program text

134.2

to yield the three numbers

1, 3, and 4.2

Semicolons

The formal grammar uses semicolons ";" as terminators in a number of productions. Go programs may omit most of these semicolons using the following two rules:

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is
   - an identifier
   - an integer, floating-point, imaginary, rune, or string literal
   - one of the keywords `break`, `continue`, `fallthrough`, or `return`
   - one of the operators and punctuation `++`, `--`, `)`, `]`, or `}`

2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing ")" or "}".

This seems to be a parsing rule ("omission allowed") vs. the previous rule as a lexical analysis rule ("automatically inserted").

To reflect idiomatic use, code examples in this document elide semicolons using these rules.

Keywords

The following keywords are reserved and may not be used as identifiers.

`break`	`default`	`func`	`interface`	`select`
`case`	`defer`	`go`	`map`	`struct`
`chan`	`else`	`goto`	`package`	`switch`
`const`	`fallthrough`	`if`	`range`	`type`
`continue`	`for`	`import`	`return`	`var`

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation:

`+`	`&`	`+=`	`&=`	`&&`	`==`	`!=`	`(`	`)`
`-`	`|`	`-=`	`|=`	`||`	`<`	`<=`	`[`	`]`
`*`	`^`	`*=`	`^=`	`<-`	`>`	`>=`	`{`	`}`
`/`	`<<`	`/=`	`<<=`	`++`	`=`	`:=`	`,`	`;`
`%`	`>>`	`%=`	`>>=`	`--`	`!`	`...`	`.`	`:`
	`&^`		`&^=`					

Integer literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: `0` for octal, `0x` or `0X` for hexadecimal. In hexadecimal literals, letters `a-f` and `A-F` represent values 10 through 15.

```
int_lit     = decimal_lit | octal_lit | hex_lit .

decimal_lit = ( "1" … "9" ) { decimal_digit } .
octal_lit   = "0" { octal_digit } .
hex_lit     = "0" ( "x" | "X" ) hex_digit { hex_digit } .
```

```
42
0600
0xBadFace
170141183460469231731687303715884105727
```

Note that negative numbers are handled by the unary – operator.

Floating-point literals

A floating-point literal is a decimal representation of a floating-point constant. It has an integer part, a decimal point, a fractional part, and an exponent part.

The integer and fractional part comprise decimal digits; the exponent part is an `e` or `E` followed by an optionally signed decimal exponent.

One of the integer part or the fractional part may be elided; one of the decimal point or the exponent may be elided.

```
float_lit = decimals "." [ decimals ] [ exponent ] |
            decimals exponent |
            "." decimals [ exponent ] .
decimals  = decimal_digit { decimal_digit } .
exponent  = ( "e" | "E" ) [ "+" | "-" ] decimals .
```

```
0.
72.40                       6.67428e-11
072.40  // == 72.40         1E6
2.71828                     .25
```

```
1.e+0                    .12345E+5
```

Imaginary literals

An imaginary literal is a decimal representation of the imaginary part of a complex constant. It consists of a floating-point literal or decimal integer followed by the lower-case letter `i`.

```
imaginary_lit = (decimals | float_lit) "i" .
```

```
0i
011i  // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i
```

Rune literals

A rune literal represents a rune constant, an integer value identifying a Unicode code point.

A rune literal is expressed as one or more characters enclosed in single quotes, as in `'x'` or `'\n'`.

Within the quotes, any character ...

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

Raw string literals are character sequences between back quotes, as in `` `foo` ``... backslashes have no special meaning and the string may contain newlines. Carriage return characters ('\r') inside raw string literals are discarded from the raw string value.

Interpreted string literals are character sequences between double quotes, as in `"bar"`... with backslash escapes interpreted as they are in rune literals (except that `\'` is illegal and `\"` is legal), with the same restrictions. The three-digit octal (`\`*nnn*) and two-digit hexadecimal (`\x`*nn*) escapes represent individual *bytes* of the resulting string; all other escapes represent the (possibly multi-byte) UTF-8 encoding of individual *characters*. Thus inside a string literal `\377` and `\xFF` represent a single byte of value `0xFF`=255, while ÿ, `\u00FF`, `\U000000FF` and `\xc3\xbf` represent the two bytes `0xc3 0xbf` of the UTF-8 encoding of character U+00FF.

```
string_lit             = raw_string_lit | interpreted_string_lit .

raw_string_lit         = "`" { unicode_char | newline } "`" .

interpreted_string_lit = `"` { unicode_value | byte_value } `"` .
```

This is a modern string constant definition, having both raw and interpreted styles.

CHOMSKY HIERARCHY

			Productions
0	DECIDABLE (R.E.)	TM	...
1	CONTEXT-SENSITIVE	LBA	$\alpha N \beta \to \gamma$
2	CONTEXT-FREE	PDA	$N \to abMLK$
3	REGULAR LANG.	DFA	$N \to a$, $N \to aM$, $N \to bK$

DETERMINISTIC FINITE AUTOMATON

PUSHDOWN AUTOMATON

LINEAR BOUNDED AT.

TURING MACHINE

BY REPLACING A NONTERMINAL WITH THE R.H.S. OF A PRODUCTION FOR THAT NONTERMINAL.

S=E

~~E → T+T~~
~~E → E~~
~~E → ...~~

$E \to T+T$
$E \to T$
$T \to F*F$
$T \to F$
$F \to i$
$F \to d$

$N = \{E, T, F\}$

$T = \{+, *, i, d\}$

33+foo*7

$E \to T+T \to T+F*F \to T+i*F \to F+i*F \to d+i*F$
$\alpha_0 \quad \alpha_1 \quad \alpha_2$
$\to d+i*d$

CONTEXT-FREE GRAMMAR (N, T, S, P)

N - Set of nonterminals

T - Set of terminals $\quad N \cap T = \emptyset$

S - Start symbol $\quad S \in N$

P - Productions

$N \rightarrow \alpha$

cap. letters - nonterminals
lowercase - terminals

α is a string on $N \cup T$.

DERIVATION:

starts with $\alpha_0 = S$

CREATE α_{i+1} FROM α_i

$a^n b^n$

ε

ab

$aabb$

$aaabbb$

$aaaabbbb$

$N \to aNb$

$N \to \varepsilon$