Computer Mathematics

Week 15

Context-free languages and parsing

KUAS 京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

last week

origins of computational grammars

grammars describe languages

FSMs describe languages

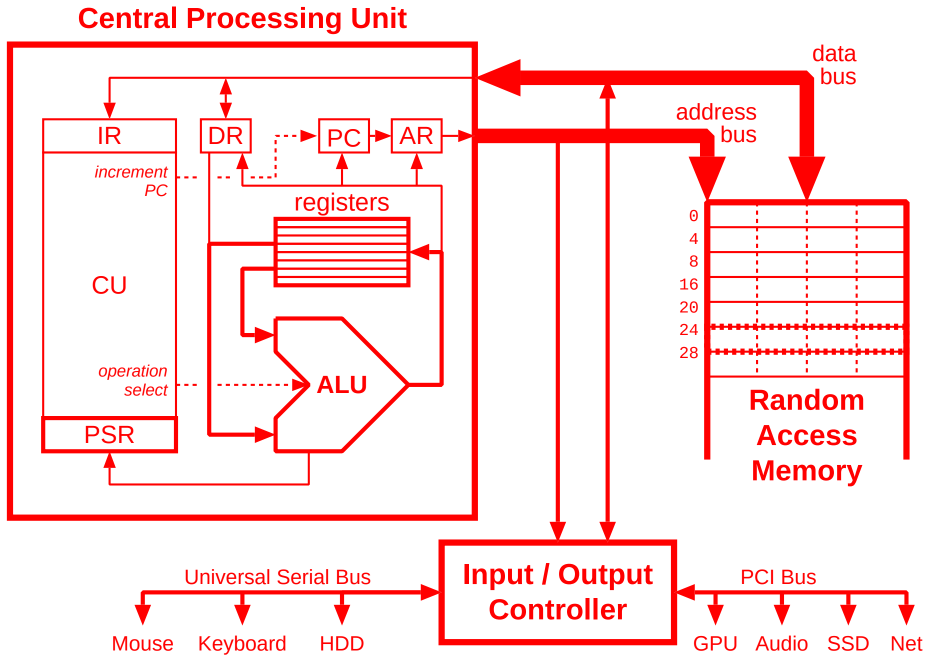
grammars corresponding to FSMs

the hierarchy of grammar types

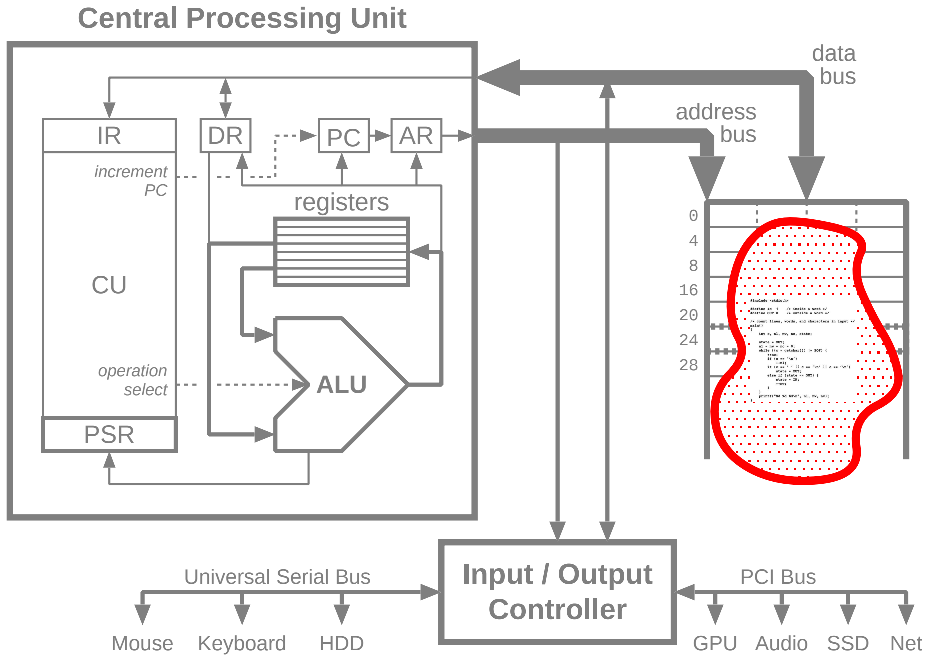
regular grammars

context-free grammars

- and their importance to computing



this week



context-free grammars

- how they behave
- how to describe languages with them
- how to parse sequences with them

the remaining grammar types

- type 1 grammars
- type 0 grammars

and the machines that correspond to them

regular and context-free grammars

regular grammars (and expressions) good for patterns

- identifiers, integers, floating-point numbers, etc.
- the sequence of operations at a traffic light, in the CPU/ALU, etc.

right-hand sides must have no more than one non-terminal and one terminal

$$
\begin{aligned}
<\textit{non-terminal}> \;\to\;& \epsilon \\
\mid\;& \texttt{terminal} \\
\mid\;& \qquad\qquad\quad <\textit{non-terminal}> \\
\mid\;& \texttt{terminal} \;\; <\textit{non-terminal}>
\end{aligned}
$$

corresponding to a finite-state machine

to describe anything recursive (things nested inside themselves)

- parenthesised expressions, HTML tags, etc.

we need a context-free grammar (CFG)

$$<\textit{non-terminal}> \;\to\; \alpha \qquad \text{(where } \alpha \text{ is any sequence of terminal or non-terminal symbols)}$$

corresponding to a finite-state machine with a stack

context

in a CFG, the left-hand side of a rule is a single non-terminal symbol

e.g., the rules

$$
\begin{aligned}
<\!P\!> \;&\rightarrow\; \mathsf{a}\ <\!P\!>\ \mathsf{a} \\
<\!P\!> \;&\rightarrow\; \mathsf{b}
\end{aligned}
$$

are both context-free (and generate palindromes “$\mathsf{a}^n\mathsf{b}\mathsf{a}^n$”)

the rule

$$\mathsf{a}\ <\!P\!> \;\rightarrow\; \mathsf{a}\ <\!P\!>\ \mathsf{a}$$

is *not* context-free

- because $<\!P\!>$ can only be replaced when it is preceded by an a
- $<\!P\!>$ is *sensitive* to the *context* in which it appears within the sentence

other grammar types

context-sensitive grammars contain rules of the form

$$\alpha \ \textit{<non-terminal>} \ \beta \rightarrow \alpha \ \gamma \ \beta$$

(where α and β can be empty, but γ must contain at least one symbol)

- when replacing a non-terminal, the sentential form cannot shrink

the machine corresponding to this kind of grammar is

- a finite-state machine
- plus a memory, that can be accessed in *any* order
  - not limited to stack (LIFO) behaviour, but
  - can be limited to the size of the input sentence

this kind of machine is called a *linear-bounded automaton* (LBA)

- because its memory is linear, and bounded by the size of the input

other grammar types

unrestricted (or *phrase-structured*) grammars can contain rules of the form

$$\alpha \;\rightarrow\; \beta$$

(where α must contain at least one symbol, and β can be anything)

- when replacing a sentential form, the sentence can shrink

$\Rightarrow$ non-terminals can vanish, or appear from nowhere, without warning

the machine corresponding to this kind of grammar is

- a finite-state machine
- plus an infinitely-large, linear memory

this kind of machine is called a *Turing machine*

- because its inventor was the mathematician Alan Turing
- it is at least as powerful as the most powerful computer we know how to build

grammar hierarchy

type 0
unrestricted (phrase-structured)
unbounded linear automaton (Turing machine)

type 1
context-sensitive
linear-bounded automaton

type 2
context-free
push-down automaton

type 3
regular
finite state machine

note that each type of grammar includes all those with higher numbers

- e.g., you can implement a finite state machine (a regular language)
- using a program that generates parsers for context-free languages
  - provided you only use rules that are regular

but let's return to context-free languages, since most of computing is made from them...

syntax and semantics are (often) separated

syntax [mass noun]

- the arrangement of words and phrases to create well-formed sentences in a language

semantics [plural noun]

- the meaning of a word, phrase or text

easy to create sentences that appear meaningless

<sentence> → *<noun-phrase>* *<verb-phrase>*
<noun-phrase> → *<adjective>* *<noun>*
<verb-phrase> → *<verb>* *<adverb>*

<adjective>	*<noun>*	*<verb>*	*<adverb>*
`considerate`	`tea`	`drinks`	`rapidly`

syntax conveys meaning about roles and relationships

<adjective> *<noun>* *<verb>* *<adverb>*

```
considerate  tea  drinks  rapidly
```

even here, we can identify that

something	tea
that has a property	considerate
does something	drinks
in a particular way	rapidly

what does this have to do with computer languages?

- syntax can tell us about the roles of symbols and their relationships

to understand how, let's look at the syntactic structure of expressions

grammars describe patterns of symbols

recall our BNF example of addition from last week

$$
\begin{array}{rcl}
<\mathit{sum}> & \rightarrow & <\mathit{identifier}> \; \texttt{+} \; <\mathit{identifier}> \\
<\mathit{identifier}> & \rightarrow & <\mathit{letter}> \mid <\mathit{letter}> \; <\mathit{identifier}> \\
<\mathit{letter}> & \rightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{x} \mid \texttt{y} \mid \texttt{z}
\end{array}
$$

repetition was via recursion

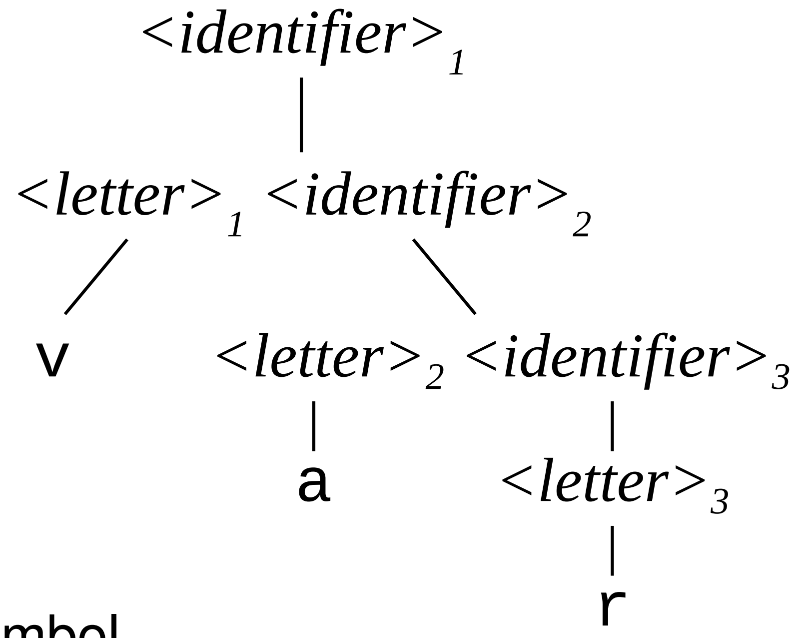
$$
\begin{array}{lll}
 & \text{sentential forms} & \text{replacements} \\
 & <\mathit{identifier}> & (<\mathit{identifier}> \rightarrow <\mathit{letter}> \; <\mathit{identifier}>) \\
\rightarrow & <\mathit{letter}> \; <\mathit{identifier}> & (<\mathit{letter}> \rightarrow \text{“}\texttt{v}\text{”}) \\
\rightarrow & \texttt{v} \; <\mathit{identifier}> & (<\mathit{identifier}> \rightarrow <\mathit{letter}> \; <\mathit{identifier}>) \\
\rightarrow & \texttt{v} \; <\mathit{letter}> \; <\mathit{identifier}> & (<\mathit{letter}> \rightarrow \text{“}\texttt{a}\text{”}) \\
\rightarrow & \texttt{v} \; \texttt{a} \; <\mathit{identifier}> & (<\mathit{identifier}> \rightarrow <\mathit{letter}>) \\
\rightarrow & \texttt{v} \; \texttt{a} \; <\mathit{letter}> & (<\mathit{letter}> \rightarrow \text{“}\texttt{r}\text{”}) \\
\rightarrow & \texttt{v} \; \texttt{a} \; \texttt{r} &
\end{array}
$$

the *derivation* of “`var`” is the sequence of replacements that we made

- from the start symbol $<\mathit{identifier}>$ to the final sentence “`var`”
- numbering distinct ‘appearances’ of a non-terminal will help us see how it evolves

derivations form a ‘tree’ structure

sentential forms	replacements
$<\textit{identifier}>$	$(<\textit{identifier}>_1 \rightarrow <\textit{letter}>_1\ <\textit{identifier}>_2)$
$\rightarrow <\textit{letter}>\ <\textit{identifier}>$	$(<\textit{letter}>_1 \rightarrow$ “v”$)$
$\rightarrow$ v $<\textit{identifier}>$	$(<\textit{identifier}>_2 \rightarrow <\textit{letter}>_2\ <\textit{identifier}>_3)$
$\rightarrow$ v $<\textit{letter}>\ <\textit{identifier}>$	$(<\textit{letter}>_2 \rightarrow$ “a”$)$
$\rightarrow$ v a $<\textit{identifier}>$	$(<\textit{identifier}>_3 \rightarrow <\textit{letter}>_3)$
$\rightarrow$ v a $<\textit{letter}>$	$(<\textit{letter}>_3 \rightarrow$ “r”$)$
$\rightarrow$ v a r	



within the derivation *tree*

- the *root* is the start symbol
- the *leaves* are the terminal symbols
- the *branches* are the sequences of non-terminal replacements that were made to reach each terminal symbol from the start symbol

derivations form a ‘tree’ structure

$$
\begin{aligned}
\langle \textit{expression} \rangle \;\rightarrow\;& \langle \textit{expression} \rangle \ \texttt{+} \ \langle \textit{expression} \rangle \\
\mid\;& \langle \textit{expression} \rangle \ \texttt{*} \ \langle \textit{expression} \rangle \\
\mid\;& \langle \textit{number} \rangle
\end{aligned}
$$

$$\langle \textit{number} \rangle \rightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{8} \mid \texttt{9}$$

let’s try to find a derivation for:

$$\texttt{1 + 2 * 3}$$

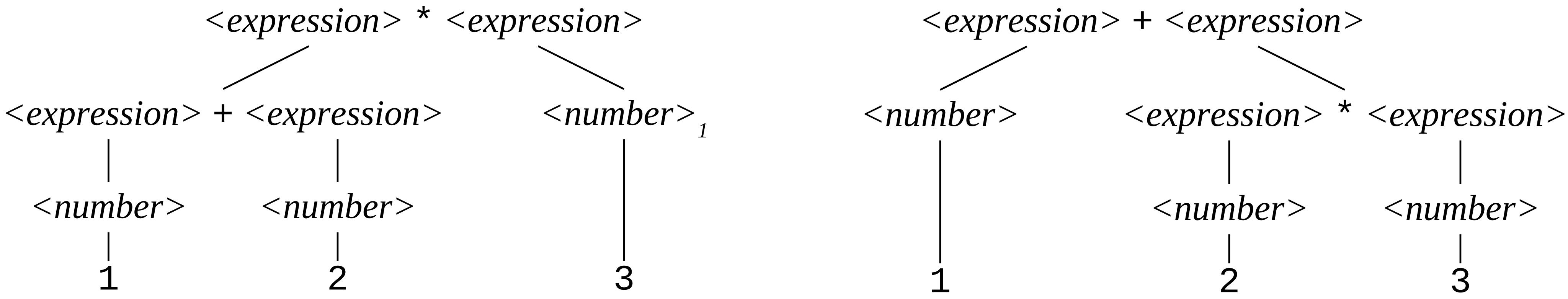
(remember: ‘ | ’ does not imply an order, and *any* alternative that works is acceptable)

syntactic structure can be ambiguous

the sentence can be generated in two ways

- the choice makes *no difference* to the language defined by the grammar

$$1 + 2 * 3$$



however, the *meaning* of the sentence is its value (as an arithmetic expression)

before applying an operator, all its operand values must be calculated

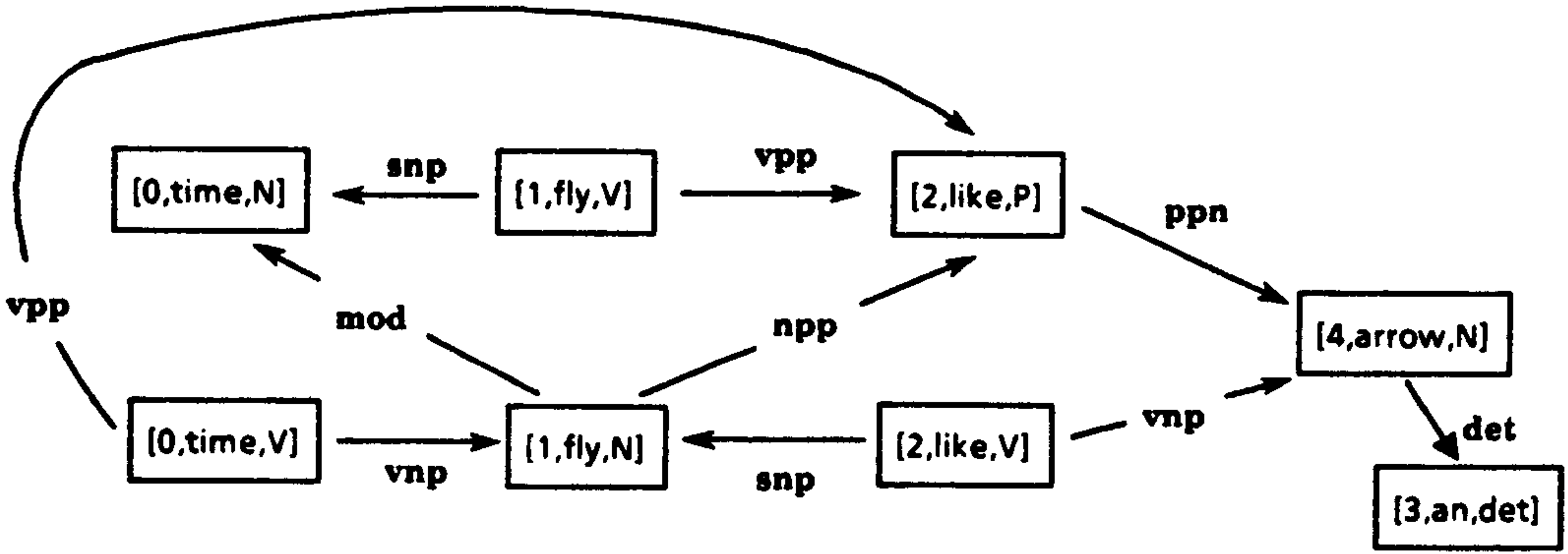
⇒ values are calculated *bottom-up*

- from the leaves of the tree up towards the root

the two derivations apply operators in a different order

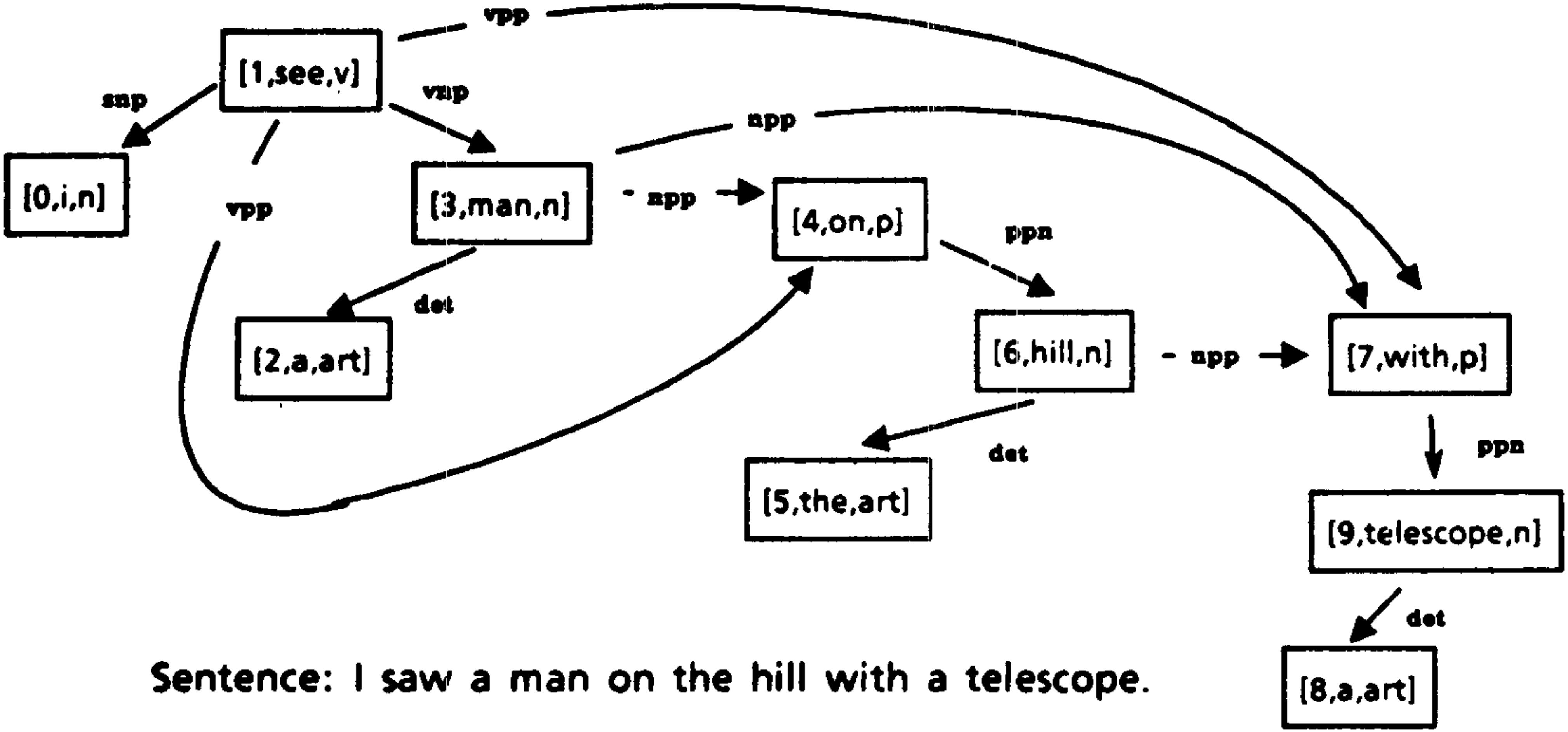
- the choice of derivation makes a *big difference* to the meaning of the sentence

ambiguity creates uncertainty of meaning



Sentence: Time flies like an arrow.

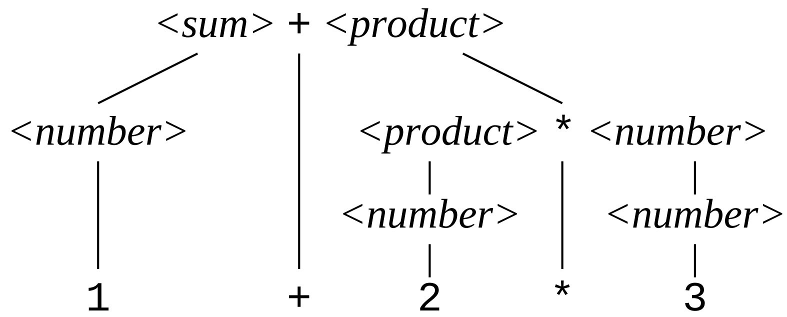
ambiguity creates uncertainty of meaning



Sentence: I saw a man on the hill with a telescope.

grammar can impose correct syntactic structure

$$\begin{aligned}
\langle sum\rangle &\rightarrow \langle sum\rangle \; \texttt{+} \; \langle product\rangle \\
&\;\mid\; \langle product\rangle \\
\langle product\rangle &\rightarrow \langle product\rangle \; * \; \langle number\rangle \\
&\;\mid\; \langle number\rangle \\
\langle number\rangle &\rightarrow \langle digit\rangle \mid \langle digit\rangle \; \langle number\rangle \\
\langle digit\rangle &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{...} \mid \texttt{8} \mid \texttt{9}
\end{aligned}$$



ambiguity has been removed

- any given expression has only one derivation
- the language did not change
  - the exact same set of sentences can be derived
- higher-precedence operators are placed ‘lower’ in the grammar
- recursion provides repetition of same-precedence operators

does it matter whether we use left- or right-recursion for the repetition?

correct syntactic structure imposes correct semantics

$$
\begin{array}{rcl}
\langle \textit{expression} \rangle & \rightarrow & \langle \textit{assignment} \rangle \mid \langle \textit{sum} \rangle \\
\langle \textit{assignment} \rangle & \rightarrow & \langle \textit{id} \rangle \; \texttt{=} \; \langle \textit{assignment} \rangle \\
\langle \textit{sum} \rangle & \rightarrow & \langle \textit{sum} \rangle \; \texttt{+} \; \langle \textit{product} \rangle \\
 & \mid & \langle \textit{sum} \rangle \; \texttt{-} \; \langle \textit{product} \rangle \\
 & \mid & \langle \textit{product} \rangle \\
\langle \textit{product} \rangle & \rightarrow & \langle \textit{product} \rangle \; * \; \langle \textit{number} \rangle \\
 & \mid & \langle \textit{product} \rangle \; \texttt{/} \; \langle \textit{number} \rangle \\
 & \mid & \langle \textit{product} \rangle \; \texttt{\%} \; \langle \textit{number} \rangle \\
 & \mid & \langle \textit{number} \rangle
\end{array}
$$

using left recursion for "+" . . . "%" makes them all left-associative

- "7−3−1" means 7−3 = 4, then 4−1 = 3 (<u>not</u> 3−1 = 2, then 7−2 = 5)

using right recursion for assignment makes it right-associative

- "a = b = 42" means b = 42, then a = b (<u>not</u> a = b, then b = 42)

abstract syntax trees

parse trees contain too much information

we don’t really care that

- an <*expression*> made a <*sum*>, and
- the <*sum*> made a <*sum*> + <*product*>, and
- the <*sum*> made a <*number*>, and the <*number*> made a `3`, and
- the <*product*> made a <*number*>, and the <*number*> made a `4`

all we really wanted to know was

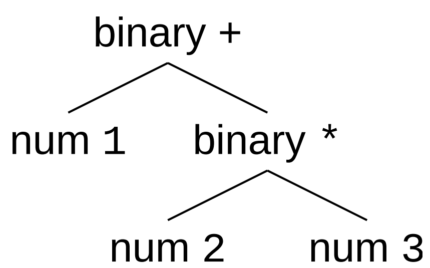
- <*number=3*> + <*number=4*>

this kind of tree is called an *abstract syntax tree* (AST)

- it encodes only what is required to understand the meaning of the sentence

given an AST we could (e.g.) *evaluate* it, or *compile* it into machine code

“`1+2*3`” $\longrightarrow$



parsing

generating sentences from a grammar is easy

- replace non-terminals, choosing arbitrary right-hand sides
- eventually a sentence appears, along with its derivation (the set of choices)

parsing goes in the opposite direction

- given a grammar, and a sentence...
- find a derivation that produces the sentence

this is a far harder problem

there are many (many) algorithms for parsing

- and a different classification system for grammars
- based on which parsing algorithms work with them

enough for an entire course (and beyond)

let’s look at one way to write parsers by hand

- which works well for many kinds of computer languages and applications

recursive-descent parsing

the basic idea:

- write functions that follow the structure of the grammar

each function has

- input: a string and a position within the string
- output: true/false (was the input recognised)
  - advancing the input position if it was

```
text = "your input sentence goes here"
position = 0

def parseSomething():
    if cannot recognise Something at text[position]: return False
    position += size of Something that was recognised
    return True
```

recursive-descent parsing

identifiers revisited

$$\langle \textit{identifier} \rangle \rightarrow \langle \textit{letter} \rangle \mid \langle \textit{letter} \rangle \; \langle \textit{identifier} \rangle$$

$$\langle \textit{letter} \rangle \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{x} \mid \texttt{y} \mid \texttt{z}$$

to parse a letter

- recognise a valid alternative

```
def parseLetter(): # a | b | ... | y | z
    global text, position
    if text[position] < "a" or text[position] > "z": return False
    position += 1
    return True
```

to parse an identifier, we can ‘left factor’ the rule to simplify it

- parse a letter, then an optional identifier

```
def parseIdentifier(): # letter | letter identifier ≡ letter identifier*
    global text, position
    if not parseLetter(): return False
    parseIdentifier()
    return True
```

returning values from parser rules

```
text = "hello+there$" # $ is explicit end-of-text marker
position = 0
lval = None

def parseLetter(): # a | b | ... | y | z
    global text, position, lval
    if text[position] < "a" or text[position] > "z": return False
    lval = text[position]
    position += 1
    return True

def parseIdentifier(): # letter | letter identifier ≡ letter identifier*
    global text, position, lval
    start = position
    if not parseLetter(): return False
    parseIdentifier()
    lval = text[start:position]
    return True

print(parseIdentifier())
print(position)
print(lval)
```

constructing an AST during parsing

```
text = "hello+there$" # $ is explicit end-of-text marker

def parseIdentifier(): # letter | letter identifier
    global text, position, lval
    start = position
    if not parseLetter(): return False
    parseIdentifier()
    lval = ["identifier", text[start:position]]
    return True

def parseSum(): # sum + identifier | identifier ≡ identifier ( + identifier )*
    global text, position, lval
    if not parseIdentifier(): return False
    saved = position
    result = lval
    while text[position] == "+":
        position += 1
        if not parseIdentifier():
            print("identifier expected at position " + str(position))
            exit()
        result = ["add", result, lval]
        saved = position
    lval = result
    return True
```

homework

reinforce your understanding

- see if you can write a parser for addition and multiplication
  - in Python
  - the input should be a string, e.g., "`1+2*3`"
- produce an AST for the input
  - for a numbers N, make a list: `['number',` N`]`
  - for additions, make a list: `['add',` *left-child* `,` *right-child* `]`
  - for multiplications, make a list: `['mul',` *left-child* `,` *right-child* `]`
- verify you produce the correct ASTs for different expressions
  - including things like "`1+2+3*4`" and "`1+2*3+4`"
- write a function that can evaluate your AST and print the result

ask about anything you do not understand

- from any of the classes so far this semester (or the lecture notes)
- it will be too late for you to try to catch up later!
- I am always happy to explain things differently and practice examples with you

glossary

abstract syntax tree — one possible result of parsing, similar to a parse tree but containing only the information that is necessary to extract the meaning of the input.

bottom-up — (tree algorithms) an algorithm that begins at the leaves of the tree, and proceeds up the tree towards the root. At each step, processing of a node in the tree is performed only when all of its children have been completely processed.

branches — the paths through a tree that connect the root to the leaves.

compile — convert a source code program into executable machine code.

context-sensitive grammars — grammars in which rules specify the context in which a non-terminal must occur in order to be replaced by the right-hand side.

context — the symbols in a sentence (or sentential form) preceding or following a non-terminal symbol.

derivation — the sequence of replacement steps performed to convert the start symbol of a grammar into a valid sentence.

evaluate — calculate a value for a structure, such as a numerical value for a tree representing an arithmetic expression.

leaves — the nodes at the ends of the branches in a tree, which have no children.

linear-bounded automaton — a finite state automaton with randomly accessible memory bounded by some upper limit, such as the size of the input data.

parse tree — a tree showing the parent-child relationships produced by the sequence of replacements made in a derivation of a sentence.

phrase-structured — (grammar) another name for an unrestricted grammar.

root — (of a tree) the top-most node in a tree, which has no parent.

top-down — (tree algorithms) an algorithm that begins at the root of the tree and proceeds down the tree towards the root.

tree — a structure in which nodes have parent-child relationships, each node having exactly one parent (except for the root, which has no parent).

Turing machine — an automaton that has an infinitely large memory.

unrestricted — a grammar in which anything goes, the only restriction being that the left-hand side of a rule cannot be empty.