Programming Abstractions

Week 5: A quick introduction to grammars

Stephen Checkoway

Alphabets and words

An alphabet Σ is a finite, nonempty set of symbols
‣ {0, 1} is a binary alphabet
‣ The set of emoji is an alphabet
‣ The set of English words is an alphabet

A word (also called a string) *w* over an alphabet Σ is a finite (possibly-empty) sequence of symbols from the alphabet
‣ The empty word, ε, consisting of no symbols is a word over every alphabet
‣ `001101` is a word over {0, 1}
‣ ⭐🐈🐈🐉💥🤦‍♀️ is a word over the emoji alphabet
‣ `functional programming is great` is a word over English

Languages

A language is a (possibly infinite) set of words over an alphabet

There's a whole lot we can do studying languages as mathematical objects

We're not going to do that in this course, take theory of computation to find out more!

For a given programming language (like Scheme) the alphabet is the set of keywords, identifiers, and symbols in the language
‣ This is a bit of a simplification because there are infinitely many possible identifiers but alphabets must be finite

A word (or string) over this alphabet is in the programming language if it is a syntactically valid program

Syntactically valid?

Consider the invalid Scheme program
```
(let ([x 5]
      [y 32])
  (+ z 2))
```

This is *syntactically* valid (i.e., it's a word in the Scheme language) but *semantically* meaningless as we don't have a binding for the identifier z

Grammars

A grammar for a language is a (mathematical) tool for specifying which words over the alphabet belong to the language

Grammars are often sued to determine the meaning of words in the language

For example, consider the arithmetic expression a+b*c as a word over the alphabet consisting of variables and arithmetic operators
‣ We can write many different grammars that will let us determine if a given word is a valid expression (i.e., is in the language of valid expressions)
‣ With a careful choice of grammars we can determine that this means a+(b*c) and not (a+b)*c

Grammars are very old, dating back to at least Yāska the 4th c. BCE

Mathematical representation of grammars

A grammar G is a 4-tuple $G = (V, \Sigma, S, R)$ where

‣ V is a finite, nonempty set of *nonterminals*, also called variables
‣ Σ is an alphabet of *terminal* symbols
‣ $S \in V$ is the *start* nonterminal
‣ R is a finite set of *production rules*

(Terminal symbols are distinct from nonterminals)

In English, we might have nonterminals like *NOUN*, *VERB*, *NP*, etc.

We often write nonterminals in upper-case and terminals in lower-case

Production rules

Nonterminals are expanded using production rules to sequences of terminals and nonterminals

A production rule looks has the form
$A \rightarrow \alpha$

where A is a nonterminal and α is a (possibly-empty) word over $\Sigma \cup V$

Here's an example for Scheme
$EXP \rightarrow$ (`if` *EXP EXP EXP*)

This says that wherever we have an expression, we can expand it to an if-then-else expression which starts with `(` followed by `if` and then three more expressions and lastly `)`

Example grammar for arithmetic

EXP → *EXP* + *TERM*
EXP → *TERM*
TERM → *TERM* * *FACTOR*
TERM → *FACTOR*
FACTOR → (*EXP*)
FACTOR → number

Compact form：

EXP → *EXP* + *TERM* | *TERM*
TERM → *TERM* * *FACTOR* | *FACTOR*
FACTOR → (*EXP*) | number

Derivations

A *derivation* with a grammar starts with a nonterminal and replaces nonterminals one at a time until only a sequence of terminals remains

Derivation example

EXP ⇒ EXP + TERM

⇒ TERM + TERM

⇒ FACTOR + TERM

⇒ 3 + TERM

⇒ 3 + TERM * FACTOR

⇒ 3 + FACTOR * FACTOR

⇒ 3 + 4 * FACTOR

⇒ 3 + 4 * 50

EXP → EXP + TERM | TERM
TERM → TERM * FACTOR | FACTOR
FACTOR → (EXP) | number

Parse tree

EXP ⇒ EXP + TERM

    ⇒ TERM + TERM

    ⇒ FACTOR + TERM

    ⇒ 3 + TERM

    ⇒ 3 + TERM * FACTOR

    ⇒ 3 + FACTOR * FACTOR

    ⇒ 3 + 4 * FACTOR

    ⇒ 3 + 4 * 50
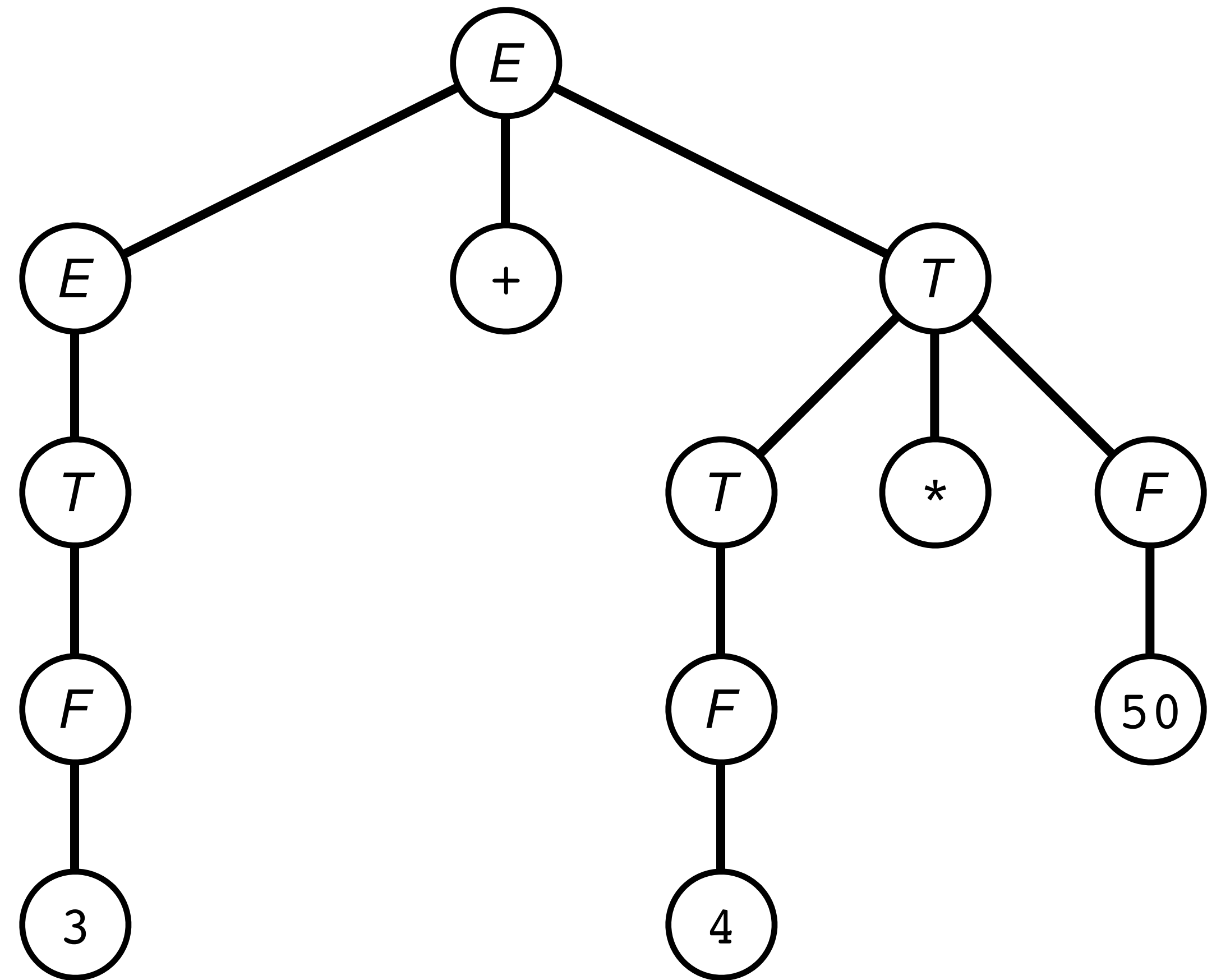
Note that the derived expression is a left-to-right traversal of the leaves

The language generated by a grammar

One nonterminal is designated as the start nonterminal
‣ Typically, this is the nonterminal on the left-hand side of the first production rule

The language *generated* by the grammar is the set of words over the terminal alphabet which can be derived by the production rules, starting with the start nonterminal

A convenient shorthand

It's often useful to say that a particular terminal or nonterminal can appear 0 or more times

$A \rightarrow xA \mid \varepsilon$

where x is either a terminal or nonterminal and ε represents the empty word

Similarly, it's often useful to say that a particular terminal or nonterminal can appear 1 or more times

$A \rightarrow xA \mid x$

We write x^* or x^+ as a shorthand for these constructs

A full grammar for Minischeme

EXP → number

 | symbol

 | (`if` *EXP EXP EXP*)

 | (`let` (*LET-BINDINGS*) *EXP*)

 | (`letrec` (*LET-BINDINGS*) *EXP*)

 | (`lambda` (*PARAMS*) *EXP*)

 | (`set!` symbol *EXP*)

 | (`begin` *EXP**)

 | (*EXP*$^+$)

LET-BINDINGS → *LET-BINDING**

LET-BINDING → `[` symbol *EXP* `]`

PARAMS → symbol*