CS 320: Principles of Programming Languages

Katie Casamento, Portland State University
Based on slides by Mark P. Jones, Portland State University, Winter 2017

Winter 2018
Week 3: Describing Syntax

Language = syntax + semantics

concrete syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines

abstract syntax: the representation of a program structure, independent of written form

syntax analysis: transformation from concrete syntax to abstract syntax

---

Syntax analysis

Usually a two-step process:

- *Tokenization* or *lexing* turns a character stream into a token stream
- *Parsing* turns a token stream into an abstract syntax structure

---

Syntax analysis

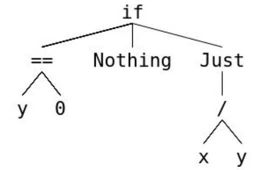
Character stream

```
"if y == 0 then Nothing else Just (x / y)"
```

→ Token stream

```
[IF, ID(y), ==, NUM(0), THEN, ID(Nothing), ELSE, ID(Just),
L_PAREN, ID(x), /, ID(y), R_PAREN]
```

→ Abstract syntax tree



Formal languages

How do we describe syntax?

We want methods that are:

- clear, precise, and unambiguous
- expressive (e.g. finite descriptions of infinite languages)
- suitable for use in the implementation of syntax analysis tools (lexers, parsers, …)

Formal languages provides such a foundation:

- *Regular languages* describe lexical syntax (grouping characters into tokens)
- *Context-free languages* describe more complex syntactic structure (parsing token streams into expressions)

Formal languages

- Pick a set of *symbols*, A, to be the *alphabet*
  - For lexical analysis, "symbols" are typically characters
  - For parsing, "symbols" are typically tokens
- The set of all finite strings of symbols in A is written A*
- A *language* over A is a subset of A*

Examples

A = `{0, 1}`
so A* = `{"", "0", "1", "00", "01", "10", "11", "000", …}`

The set of bytes is a finite language over A:
Bytes = $\{b_0b_1b_2b_3b_4b_5b_6b_7 \mid b_i \in A\}$

The set of even-length bitstreams is an infinite language over A:
Evens = `{"00", "01", "10", "11"}*`

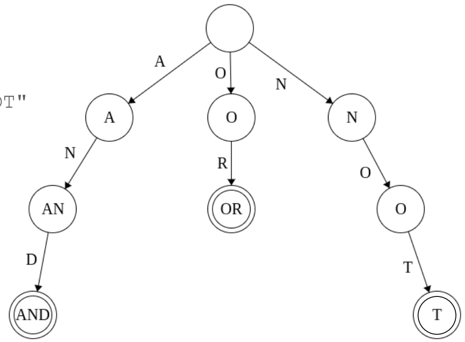
Prop as a formal language

- Alphabet:
  - `{"A", "B", "C", …, "(", ")", …}`
- Tokens:
  - Keywords: `AND, OR, NOT`
  - Literals: `TRUE, FALSE`
  - Punctuation: `L_PAREN, R_PAREN`
  - Input names: all nonempty subsets of A* containing only letters
- Expressions:
  - The subset of Tokens* corresponding to valid circuits

How do we specify these details?

Keywords in Prop

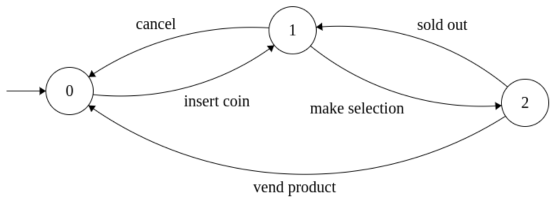
PropKW = `"AND" | "OR" | "NOT"`



Finite automata and regular languages

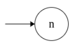
Terminology

A *finite automaton* (or *finite state machine*) describes a system that can *transition* (or *move*) between different *states* in response to particular *inputs*.



Finite automata building blocks

state, labeled n



start state

accept (final) state

$\xrightarrow{a}$ transition on input "a"

$\xrightarrow{\varepsilon}$ transition without consuming any input

Determinism

- A machine is non-deterministic (an *NFA*, or *non-deterministic finite automaton*) if it has a state with either more than one transition on the same symbol or if it has any ε-transitions.
- Otherwise the machine is deterministic (a *DFA*, or *deterministic finite automaton*)

Regular languages

A *regular language* is a language specified by:

- A deterministic finite automaton
- A nondeterministic finite automaton
- A regular expression

These are all equivalent: a DFA can be converted to a regex, a regex can be converted to an NFA, etc.

Regular languages

- `{}` and `{""}` are languages over any alphabet
- if $c \in A$, then `{c}` is a language over A
- if L_1 and L_2 are regular languages, then so are
  - $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$
  - $L_1 \bullet L_2$ (or just L_1L_2) $= \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- If L is a language, then so is
  - $L^* = \{""\} \cup \{xy \mid x \in L,\ y \in L^*\}$

Regular expressions

- A language for describing regular languages
- Often used in:
  - text editors/programming languages, for describing patterns in text strings (e.g. email address validation)
  - lexers, for describing the lexical structure of a language

Regular expressions

- ε

  Empty: matches the empty string
- c

  Constant: matches the single character 'c'
- $r_1 \mid r_2$

  Alternation: strings matching either r_1 or r_2
- $r_1 \bullet r_2$ (or just r_1r_2)

  Sequencing/concatenation: a string matching r_1 followed by a string matching r_2
- r*

  Repetition: a sequence of zero or more strings, each matching r
- (r)

  Grouping: string matching r

Derived forms

- r+ = r • r*
  - Repetition: a sequence of one or more strings, each matching r
- r? = ε | r
  - Option: an empty string or a string matching r
- [abc] = a | b | c
  - Character class: any listed character (also allows ranges, e.g. [a-zA-Z])
- .
  - Wildcard: matches any character
- ^
  - Line start: matches the empty string, but only at the start of a line
- $
  - Line end: matches the empty string, but only at the end of a line

Compiler-related examples

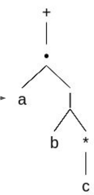
- Decimal integer literals: `[0-9]+`
- Keywords: `i • f`, `e • l • s • e`
- Haskell variables: `[a-z][A-Za-z0-9'_]*`
- Haskell types/constructors: `[A-Z][A-Za-z0-9'_]*`
- Whitespace: `[ \t\n]*`
- C-style comment: `//.*$`

Abstract syntax of regexes

How do we write programs that operate on regexes?

- The same way we operate on any kind of syntax: with an abstract syntax data structure

`(a • (b | c*))+` →



Denotational semantics of regexes

Each regular expression describes a regular language, where L(r) is the language denoted by r:

- $L(\varepsilon) = \{""\}$
- $L(c) = \{c\}$
- $L(r_1 \mid r_2) = L(r_1) \cup L(r_2) = \{x \mid x \in L(r_1) \text{ or } x \in L(r_2)\}$
- $L(r_1 \bullet r_2) = L(r_1) \bullet L(r_2) = \{xy \mid x \in L(r_1) \text{ and } y \in L(r_2)\}$
- $L(r^*) = L(r)^* = \{""\} \cup \{xy \mid x \in L(r),\ y \in L(r^*)\}$
- $L((r)) = L(r)$

This function is an interpreter, mapping a regex (syntax) to a set of strings (semantics).

Basics of lexical analysis

- Lexical analysis is carried out by a lexer/scanner/tokenizer
- Goal: to recognize and identify the sequence of tokens represented by the characters in a program's text
- The lexical structure (definition of tokens) is an important part of many language specifications

Lexemes

- A *lexeme* is a string that might represent a single atomic syntactic unit
- Examples of lexemes in Haskell:
  - `"0.0"`
  - `"String"`
  - `"True"`
  - `"if"`
  - `"("`
  - `"eval"`

Tokens

- A *token type* classifies lexemes; a *token* is a lexeme tagged with a token type
- Examples of tokens in Haskell:
  - `"0.0"    = NUM(0.0)`
  - `"String" = ID("String")`
  - `"True"   = ID("True")`
  - `"if"     = IF`
  - `"("      = L_PAREN`
  - `"eval"  = ID("eval")`
- When a token type contains only one lexeme (e.g. `IF`), we usually leave out the lexeme and just write the type
- The tokens and lexemes for a language are usually chosen so that each valid lexeme is a member of exactly one token set

Patterns

- A *pattern* is a description of the way that a set of lexemes are written
- Informally, in natural language
  - e.g. from the Java spec: "An identifier is an unlimited-length sequence of Java letters and Java digits…'
- Formally, in the language of regular expressions:
  - `ID = letter • (letter | digit)*`

Common token types

- Keywords, symbols, punctuation
  - `for, if, then, <=, +, (, ;, .`
- Literals/constants
  - integers
  - floating point numbers
  - characters
  - strings
- Identifiers
  - `String, True, eval`

Other input elements

Other elements that might appear in the input stream (but are not tokens):

- Whitespace (space, tab, newline, etc.)
  - Except in whitespace-sensitive languages (e.g. Python)
- Comments

These are filtered out during lexing and not passed as tokens to the parser.

Lexical analysis summary

- Lexing breaks input streams of characters into output streams of tokens, usually filtering out whitespace and comments
- Regular expressions, regular languages, and finite automata provide a solid (but not mandatory) foundation for lexical analysis
  - Precise and concise notions for describing syntax
  - Expressive enough for the lexical syntax of many languages
  - Algorithms and practical tools exist to construct efficient lexers

Context-free languages

Matching brackets

- Brackets = `{""} ∪ {[b] | b ∈ Brackets}`
- So the words in Brackets are:
  - `"", "[]", "[[]]", "[[[]]]", "[[[[]]]]", …`
- In other words, nested pairs of bracket characters:
  - A sequence of N open brackets followed by N close brackets
- A subset of any language that uses brackets
- Is it regular?
  - Is there a regular expression r such that L(r) = Brackets?

Brackets is not regular

Remember the pumping lemma?

- In short, intuitively: DFAs can't count
- If s_n is the state that we reach after n open brackets and $n \neq m$, then $s_n \neq s_m$
- So there needs to be one state for each possible number of open brackets
- There are an infinite possible numbers of open brackets!
- So any machine to match Brackets must have infinite states, and therefore is not a **finite** state machine
- So Brackets can't be regular

Repetition vs. recursion

- Regular expressions allow iteration (repetition) (e.g. with the * operator) but don't allow recursion (self-reference)
- A recursive characterization of Brackets is straightforward:
```
B → ε
B → [B]
```
- Meaning: an element of Brackets is either the empty string, or an element of Brackets surrounded by brackets

Context-free grammars

Formally: a context-free grammar G = (T, N, P, S) consists of

- A set T of *terminal* symbols (tokens)
- A set N of *nonterminal* symbols
- A set P of *productions* (elements of `N × (T ∪ N)*`)
  - Usually written "n → w" where n ∈ N and w ∈ (T ∪ N)*
- A *start symbol* S ∈ N

Brackets CFG

```
T = {'[', ']'}
N = {B}
P = {B → ε, B → [B]}
S = B
```
Brackets = `(T, N, P, S)`

In practice, we usually just write the productions; the start symbol is either denoted with S or assumed to be the left-hand symbol in the first production.

Example CFGs

```
Prop (without inputs) =
    P → TRUE
    P → FALSE
    P → (P)
    P → AND P P
    P → OR P P
    P → NOT P
```

```
Regex =
    R → c
    R → ε
    R → R • R
    R → R | R
    R → R*
```

```
Arithmetic =
    E → n
    E → (E)
    E → E + E
    E → E * E
```

Derivations

Formally: A *derivation* of a CFG is a sequence of strings $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$ where each string s_{i+1} is obtained from the previous string s_i by choosing a production n→w and replacing an occurrence of n in s_i with w.

In Brackets:

B → [B] → [[B]] → [[[B]]] → [[[]]]

In Prop:

P → AND P P → AND (P) P → AND (NOT P) P → AND (NOT TRUE) P → AND (NOT TRUE) FALSE

We say that a CFG *generates* the language that contains all strings that can be derived from the start symbol; any language generated by a CFG is a *context-free language*.

Why "context-free"?

- The productions in a CFG can be expanded anywhere in a derivation, regardless of surrounding symbols
- In contrast to a context-sensitive grammar, which can have productions with the left hand side restricted to certain contexts - e.g. `[B] → (B)`

EBNF grammars

Extended Backus-Naur form (EBNF) is a shorthand syntax for CFGs, very often used in programming language specifications.

Definition: `… = …`
Terminal string: `"…"`
Alternation: `… | …`
Zero or one: `[ … ]`
Zero or more: `{ … }`

```
digit = "0" | "1" | "2" | … | "9"
expr = [digit]{digit} {"+" expr}
```

Parse trees

Multiple derivations

Often, there are multiple choices for a step in a derivation:

- In a *right-most* derivation, replace the right-most nonterminal at each step
- In a *left-most* derivation, replace the left-most nonterminal at each step
- Any other arbitrary ordering

Does it matter which derivation we use?

Example: 1 + 2 * 3

Arithmetic =
  E → n
  E → (E)
  E → E + E
  E → E * E

Left-most:

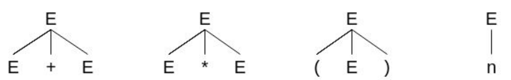
  E
→ **E** + E
→ 1 + **E**
→ 1 + **E** * E
→ 1 + 2 * **E**
→ 1 + 2 * 3

Right-most:

  E
→ E + **E**
→ E + E * **E**
→ E + **E** * 3
→ **E** + 2 * 3
→ 1 + 2 * 3

Parse trees

To capture the structure of a derivation, we use a graphical tree notation:



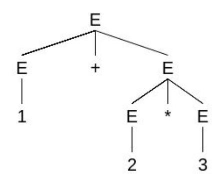
- These are called *parse trees*, or sometimes *concrete syntax trees* (*CSTs*)
  - More information than an AST (e.g. parens)
- In theory, token stream → parse tree → AST
- In practice, parse trees are often left implicit (token stream → AST)
  - A parse tree is the call graph of a recursive descent parser

Example: 1 + 2 * 3

Arithmetic =
  E → n
  E → (E)
  E → E + E
  E → E * E

Leftmost:

  E
→ **E** + E
→ 1 + **E**
→ 1 + **E** * E
→ 1 + 2 * **E**
→ 1 + 2 * 3



Rightmost:

  E
→ E + **E**
→ E + E * **E**
→ E + **E** * 3
→ **E** + 2 * 3
→ 1 + 2 * 3

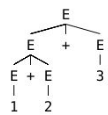
Both derivation orders produce the same parse tree - the only difference is the order in which the nodes are constructed.

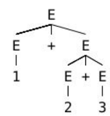
Example: 1 + 2 + 3

```
Arithmetic =
    E → n
    E → (E)
    E → E + E
    E → E * E
```

Leftmost:

```
  E
→ E + E
→ E + E + E
→ 1 + E + E
→ 1 + 2 + E
→ 1 + 2 + 3
```





Rightmost:

```
  E
→ E + E
→ E + E + E
→ E + E + 3
→ E + 2 + 3
→ 1 + 2 + 3
```

The two orders produce different trees!

Ambiguity

- A grammar is *ambiguous* if the language it generates contains a string with more than one parse tree
- e.g. our simple arithmetic grammar is ambiguous because "1+2+3" has multiple parse trees
  - There are many other expressions in the language with multiple parse trees, but one is enough to demonstrate ambiguity
- Ambiguity is a property of a grammar, not a language
  - We can have multiple grammars describing the same language, some ambiguous and some unambiguous

Dealing with ambiguity

Does it matter?

- If all parse trees for a string are semantically equivalent, it doesn't
  - e.g. for regexes, $r_1 \bullet (r_2 \bullet r_3)$ describes exactly the same language as $(r_1 \bullet r_2) \bullet r_3$, so we can parse $r_1 \bullet r_2 \bullet r_3$ either way arbitrarily without issue
- If different trees have different meanings, we need to choose between them
  - Disambiguating rules (e.g. operator precedence)
  - Rewrite the grammar to avoid ambiguity

Precedence and associativity

For two arbitrary infix operators, $\boxplus$ and $\oplus$:

- If $\boxplus$ has *higher precedence* than $\oplus$, then "a $\boxplus$ b $\oplus$ c" parses as "(a $\boxplus$ b) $\oplus$ c"
- If $\boxplus$ is *left-associative*, then "a $\boxplus$ b $\boxplus$ c" parses as "(a $\boxplus$ b) $\boxplus$ c"
- If $\boxplus$ is *right-associative*, then "a $\boxplus$ b $\boxplus$ c" parses as "a $\boxplus$ (b $\boxplus$ c)"
- If $\boxplus$ is *non-associative*, then "a $\boxplus$ b $\boxplus$ c" is a syntax error

Fixity = precedence + associativity

Order of operations

- There are widely used conventions for the precedence of arithmetic operators (PEMDAS)
- What about less traditional operators?
  - Ternary conditionals (`x ? y : z`) in C/C++/Java
  - User-defined operators (`.@.`) in Haskell
- Rules vary by language
  - C/C++/Java have a table in the spec with the fixity of each operator
  - Haskell allows user-specified fixity (e.g. `infixl 2 (.@.)`)
    - `infix`: non-associative
    - `infixl`: left-associative
    - `infixr`: right-associative

An unambiguous grammar for expressions

- Three nonterminals:
  - **E**xpressions: sum of products
  - **P**roducts: product of atoms
  - **A**toms: parenthesized expressions and numbers
- * has higher precedence than +, and both associate to the left
- Choose (arbitrarily) to only allow left-most derivations

```
Arithmetic =

  E → P
  E → E + P

  P → A
  P → P * A

  A → (E)
  A → n
```

Example: 1 + 2 * 3

```
  E
→ E + P
→ P + P
→ A + P
→ 1 + P
→ 1 + P * A
→ 1 + A * A
→ 1 + 2 * A
→ 1 + 2 * 3
```

```
Arithmetic =

  E → P
  E → E + P

  P → A
  P → P * A

  A → (E)
  A → n
```

Context-free languages summary

- Context-free grammars describe a significantly larger family of languages than regular expressions
  - Including most programming languages
- Parse trees are graphical descriptions of CFG derivations
  - Reflect the grammatical structure of the input
  - Highlight ambiguities in the grammar
  - Include more detail than ASTs
- Operator precedence and associativity can reduce/eliminate ambiguity