3. Analysis of Context-free Languages

Emil Sekerinski, **McMaster University, January 2022**

> This notebook contains type hints that allow type-checking with mypy. See also this introduction, the Python typing library, and this cheat sheet. The nb_mypy notebook extension type-checks notebook cells with mypy as they are executed. The extension can be installed by `python3 -m pip install nb_mypy`, which also installs mypy, and then has to be enabled by running the line magic below.

In [ ]:
```
%load_ext nb_mypy
```

Pushdown Automata

Context-free languages can contain nested structures, e.g.

```
S → a S c | b
```

Recognizing this language requires matching an unbounded number of `a` symbols with the same number of `c` symbols, which cannot be done by finite state automata.

Context-free languages can be recognized by *pushdown automata* that operate on a stack: transitions can push on and pop from the stack. The size of the stack is not bounded.

A pushdown automaton `P = (T, S, R, s₀)` is specified by

- a finite set `T` of *input symbols*,
- a finite set `S` of *stack symbols*,
- a finite set `R` of *transitions*,
- an *initial stack symbol* `s₀ ∈ S ∪ {ε}`,

where `T` is also called the *vocabulary* and each transition is a triple with a sequence `σ ∈ S*`, a symbol `t ∈ T ∪ {ε}`, and a sequence `σ' ∈ S*`, written:

```
σ t → σ'
```

The pushdown automaton starts with just `s₀` on the stack. A transition `σ t → σ'` can be taken if the top of the stack is `σ` and `t` can be consumed from the input: the transition will pop `σ` from the stack and push `σ'` on the stack. If there are no more input symbols when the stack is empty, the input string is accepted, otherwise rejected.

For example, an automaton accepting sequences over `T = {a, b}` with the same number of occurrences of `a` and `b`, formally `{τ ∈ T* | a#τ = b#τ}`, is `P = (T, S, R, s₀)` where `S = {a, b}`, `s₀ = ε`, and the transitions `R` are:

```
ε a → a      (1)
b a → ε      (2)
ε b → b      (3)
a b → ε      (4)
```

transition	stack	input
	ε	abba
(1)	a	bba
(4)	ε	ba
(3)	b	a
(2)	ε	ε

Here, the stack symbols coincide with the input symbols. The input `abba` is accepted by `P₀` as in the table. By convention, the stack grows to the left.

Question. What is a pushdown automaton for accepting the language generated by `S → a S c | b`?

Answer. The automaton is `P = (T, S, R, s₀)` where `T = {a, b, c}`, `S = {s, .}`, `s₀ = s`, and the transitions `R` are:

```
s a → s.     (1)
s b → ε      (2)
. c → ε      (3)
```

The input `aabcc` is accepted as in the table.

transition	stack	input
	s	aabcc
(1)	s.	abcc
(1)	s..	bcc
(2)	..	cc
(3)	.	c
(3)	ε	ε

Exercise: What is the pushdown automaton for accepting palindromes over `{a, b, c}`, i.e. sequences that read backward the same as forward?

For every finite state automaton an equivalent pushdown automaton can be constructed. For example, the finite state automaton accepting `E₁ = ab|ac` is `A₁ = (T, Q, R, q₀, F)` with `T = {a, b, c}`, `Q = {0, 1, 2, 3, 4}`, `q₀ = 0`, `F = {3, 4}`, and transitions `R`:

```
0 a → 1
0 a → 2
1 b → 3
2 c → 4
```

The equivalent pushdown automaton `P₁ = (T, S, R, s₀)` has the same vocabulary `T`, has `S = {0, 1, 2}`, `s₀ = 0`, and has transitions `R`:

`0 a → 1` (1)
`0 a → 2` (2)
`1 b → ε` (3)
`2 c → ε` (4)

That is, the stack is initialized with the initial state of `A₁`, the transitions of `A₁` and `P₁` are the same, except that transitions in `A₁` to final states pop the state from the stack in `P₁`.

Question. What are the steps to accept `ab` with `P₁`?

Answer.

transition	stack	input
	`0`	`ab`
(1)	`1`	`b`
(3)	`ε`	`ε`

For every context-free grammar, an equivalent pushdown automaton can be constructed and vice versa.

As with finite state automata, pushdown automata can be deterministic or nondeterministic. Unlike with finite state automata, it is not possible in general to make a pushdown automaton deterministic and running in linear time. The best one can achieve in general is to accept in approximately `n³` time, where `n` is the length of the input.

For accepting in linear time, restrictions on the languages have to be imposed and therefore on the grammars generating these languages. There are different ways of constructing a pushdown automaton given a grammar, each with different restrictions on the grammars. Since our ultimate goal is to determine the meaning of a sentence through its parse tree and not just to accept it, we have to be careful with modification of the grammar to suit the construction of the pushdown automaton.

Top-down and Bottom-up Parsing

Top-down parsing starts to build the parse tree with the start symbol as the goal, which is split into subgoals for each non-terminal according the grammar rules, until the terminals match the input.

Consider parsing the sentence `x × (y + z)` with grammar `G₂`:

```
E → T | E + T
T → F | T × F
F → id | ( E )
```

The equivalent top-down pushdown automaton `P₂ = (T, S, R, s₀)` has the same vocabulary `T = {+, ×, id, (, )}`, has stack symbols `S = {E, T, F, +, ×, id, (, )}`, `s₀ = E`, and has transitions `R`:

`E ε → T` (1)
`E ε → E + T` (2)
`T ε → F` (3)
`T ε → T × F` (4)
`F ε → id` (5)
`F ε → ( E )` (6)
`id id → ε` (7)

step	stack	input
	`E`	`x × (y + z)`
P	`T`	`x × (y + z)`
P	`T × F`	`x × (y + z)`
P	`F × F`	`x × (y + z)`
P	`id × F`	`x × (y + z)`
M	`× F`	`× (y + z)`
M	`F`	`(y + z)`
P	`(E)`	`(y + z)`
M	`E)`	`y + z)`
P	`E + T)`	`y + z)`
P	`T + T)`	`y + z)`
P	`F + T)`	`y + z)`
P	`id + T)`	`y + z)`
M	`+ T)`	`+ z)`
M	`T)`	`z)`
P	`F)`	`z)`

`+ + → ε` (8)
`× × → ε` (9)
`( ( → ε` (10)
`) ) → ε` (11)

P	`id)`	`z)`
M	`)`	`)`
M		

A top-down parser takes *produce (expand) steps*, for transitions (1) to (6) above, and *match steps*, for transitions (7) to (11) above:

Because of the direct correspondence between a context-free grammar and its bottom-up pushdown automaton, we omit from now on the explicit definition of the automaton.

Bottom-up parsing builds the parse tree successively bottom-up by consecutively processing input symbols; it ends when the start symbol of the grammar is reached.

Consider parsing the sentence `x × (y + z)` with grammar `G₂`:

```
E → T | E + T
T → F | T × F
F → id | ( E )
```

step	stack	input
		`x × ( y + z )`
S	`x`	`× ( y + z )`
R	`F`	`× ( y + z )`
R	`T`	`× ( y + z )`
S	`× T`	`( y + z )`
S	`( × T`	`y + z )`
S	`y ( × T`	`+ z )`
R	`F ( × T`	`+ z )`
R	`T ( × T`	`+ z )`
R	`E ( × T`	`+ z )`
S	`+ E ( × T`	`z )`
S	`z + E ( × T`	`)`
R	`F + E ( × T`	`)`
R	`T + E ( × T`	`)`
R	`E ( × T`	`)`
S	`) E ( × T`	
R	`F × T`	
R	`T`	
R	`E`	

The equivalent bottom-up pushdown automaton `P₂' = (T, S, R, s₀)` has the same vocabulary `T = {+, ×, id, (, )}`, has stack symbols `S = {E, T, F, +, ×, id, (, )}`, `s₀ = ε`, and has transitions `R`:

`T ε → E` (1)
`T + E ε → E` (2)
`F ε → T` (3)
`F × T ε → T` (4)
`id ε → F` (5)
`) E ( ε → F` (6)
`ε id → id` (7)
`ε + → +` (8)
`ε × → ×` (9)
`ε ( → (` (10)
`ε ) → )` (11)

A bottom-up parser takes *shift steps*, for transitions (7) to (11) above, and *reduce steps*, for transitions (1) to (6) above:

Bottom-up parsing proceeds without a specific goal; the parse tree grows from bottom to top; the input is accepted if it is reduced to the start symbol by two kinds to step:

- Shift steps shift the next input symbol on the stack.
- Reduce steps reduce a sequence of symbols on the stack according to a transition.

As in pushdown automata an input is accepted if the stack is empty, this can be achieved by adding one more transition, `E ε → ε`.

Conditions for Top-down Parsing

We continue with top-down parsing, which is also called *predictive parsing* since at each P step we have to predict which production to expand. The key for deterministic parsing is to select the right production. For this, we only allow that the parser selects a production with *one symbol lookahead*.

Consider parsing `xxxz` in grammar `G₃`:

```
S → A | B
A → x A | y
B → x B | z
```

step	stack	input
	`S`	`xxxz`
P	`A`	`xxxz`
P	`xA`	`xxxz`
M	`A`	`xxz`
P	`xA`	`xxz`
M	`A`	`xz`
P	`xA`	`xz`
M	`A`	`z`

After an unfortunate initial P step, we get stuck. Here, one would need to look ahead to the last input symbol to prevent that. However, with this grammar there is no bound on the number of symbols one would need to look ahead in general.

The required restrictions on the grammar are expressed in terms of the *first* and *follow sets*.

The set `first(ω)` is the set of all terminals that can appear in the first position of sentences derived from `ω`:

```
first(ω) = {t ∈ T | ω ⇒* t ν, for some ν ∈ V*}
```

The set `follow(ω)` is the set of all terminal symbols that may follow `ω` in any sentence:

```
follow(ω) = {t ∈ T | S ⇒* μ ω t ν, for some μ, ν ∈ V*}
```

Question. What are `first(A)`, `first(B)`, `first(S)`, `first(xA)`, `follow(x)`, `follow(xA)`, `follow(A)` for G_3 with `S → A | B`, `A → x A | y`, `B → x B | z`?

Answer.

- `first(A) = {x, y}`, `first(B) = {x, z}`, `first(S) = {x, y, z}`, `first(xA) = {x}`
- `follow(x) = {x, y, z}`, `follow(xA) = {}`, `follow(A) = {}`

Two conditions are required to ensure that a deterministic, one symbol lookahead top-down parser can be constructed.

Condition 1. If `A` is defined by the production

$$A \to \chi_1 \mid \chi_2 \mid \ldots$$

then the initial symbols of all sentences that can be derived from all χ_i must be distinct, i.e.:

$first(\chi_i) \cap first(\chi_j) = \{\}$ for all $i \neq j$

Question. What does this imply for productions `S → A | B`, `A → x A | y`, `B → x B | z` in G_3?

Answer. As `first(A) = {x, y}` and `first(B) = {x, z}`, production `S → A | B` does not satisfy Condition 1. An equivalent grammar with production `S → x S | y | z` satisfies the condition.

Consider parsing `x` in grammar G_4; we again may get stuck:

```
S → A x
A → x | ε
```

step	stack	input
	S	x
P	Ax	x
P	xx	x
M	x	

Condition 2. For every nonterminal `A` from which the empty sequence can be derived, the set of initial symbols must be disjoint from the set of symbols that may follow any sequence generated from `A`:

`first(A) ∩ follow(A) = {}` for all `A` such that `A ⇒* ε`

If `A ⇒* ε`, then `A` is called *nullable*.

Question. What are `first(A)` and `follow(A)` and from which nonterminals can `ε` be derived in G_4 with `S → A x`, `A → x | ε`?

Answer. Only from `A` can `ε` be derived, `A ⇒* ε`, and `first(A) = {x} = follow(A)`; hence Condition 2 is violated.

Recursive Descent Parsing

The appeal of top-down parsing is that an acceptor can be directly expressed in a programming language with mutually recursive procedures by a parsing technique known as *recursive descent*. There is no need to explicitly represent the stack of the pushdown automaton, the stack of the programming language is sufficient. Recursive descent parsing assumes that the grammar is in EBNF.

For each production `p`, a parsing procedure `pr(p)` is constructed. For production `B → E`, the name of the procedure is `B` and its body is `pr(E)`, a parser recognizing EBNF expression `E`:

p	pr(p)
B → E	procedure B() pr(E)

Assume that procedure `next` reads and assigns the next input symbol to global variable `sym`. The rules for constructing `pr(E)` for recognizing `E` are:

E	pr(E)
"a"	if sym = "a" then next else error
B	B()
(E_1)	$pr(E_1)$
$[E_1]$	if sym ∈ $first(E_1)$ then $pr(E_1)$
$\{E_1\}$	while sym ∈ $first(E_1)$ do $pr(E_1)$

`E₁ E₂ …`	`pr(E₁) ; pr(E₂) ; …`

`E₁ | E₂ | …`	`case sym of`
	`  first(E₁): pr(E₁)`
	`  first(E₂): pr(E₂)`
	`  …`
	`otherwise error`

The procedure recognizing the start symbol has to be called for recognizing a sentence of the language.

For example, consider grammar G_5:

```
A → a A c | b
```

Condition 1 requires `first(a A c) ∩ first(b) = {}`, which holds. Condition 2 applies only for nullable nonterminals, but `A` is not nullable; both conditions are satisfied. Applying above rules to `A` results in:

```
procedure A()
    case sym of
        "a": next; A() ; if sym = "c" then next else error
        "b": next
    otherwise error
```

Above, we have already applied a number of simplifications. Generally useful transformations are:

`case sym of`
`  L: if sym ∈ L then S else error`
`  …`
	`=`	`case sym of`
		`  L: S`
		`  …`

`while sym ∈ L do` `  if sym ∈ L then S else error`	`=`	`while sym ∈ L do` `  S`

`case sym of`
`  L1: S1`
`  L2: S2`
`  …`
`otherwise error`
	`=`	`if sym = L1 then S1`
		`else if sym = L2 then S2`
		`…`
		`else error`

The implementation below of the parser for G_5 consists of a single recursive parsing procedure `A`. The input are characters that are stored in the global variable `src` and an index to the next symbol is maintained. An end-of-input symbol that does not occur otherwise is appended to the input. In case that symbol is encountered prematurely, the parser exists from the recursion without consuming further input symbols. After successfully returning from the recursion, there may still input symbols be left, which is checked for:

In [3]:
```
src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos + 1
    else: sym = chr(0) # end of input symbol

def A(): # A → a A c | b
    if sym == 'a':
        nxt(); A();
        if sym == 'c': nxt()
        else: raise Exception("'c' expected at " + str(pos))
    elif sym == 'b': nxt()
    else: raise Exception("'a' or 'b' expected at " + str(pos))

def parse(s: str):
    global src, pos;
    src, pos = s, 0; nxt(); A()
    if sym != chr(0): raise Exception("unexpected characters at " + str(pos))

parse("aabcc")
```

Now consider the grammar G_6:

```
A → A a | b
```

Condition 1 for recursive descent parsing requires that `first(A a) ∩ first(b) = {}`:

```
first(A a) = {b}
```

```
first(b) = {b}
```

Hence a recursive descent parser cannot be constructed for this grammar. More generally, a recursive descent parser cannot be constructed for any grammar with left-recursive productions.

However, any grammar with left-recursive productions can be rewritten into it an equivalent grammar with right-recursive productions. Grammar G_6 can be rewritten with EBNF without recursion at all:

```
A → b {a}
```

For an EBNF grammar, the two conditions for recursive descent parsing can be phrased as follows:

E	condition(E)
$[E_1]$	$first(E_1) \cap follow(E) = \{\}$
$\{E_1\}$	$first(E_1) \cap follow(E) = \{\}$
$E_1\ E_2\ \ldots$	$first(E_i) \cap first(E_{i+1}\ E_{i+2}\ \ldots) = \{\}$ for any nullable E_i, provided E is not nullable
$E_1 \mid E_2 \mid \ldots$	$first(E_i) \cap first(E_j) = \{\}$ for all $i \neq j$

For the production `A → b {a}`, the conditions are

1. `first(b) ∩ first({a}) = {}` if `b` is nullable, which it is not, so the condition holds vacuously,
2. `first(a) ∩ follow({a}) = {}`, which holds as `first(a) = {a}` and `follow(A) = {}`.

As both conditions hold, a parser can be constructed:

```
procedure A()
    if sym = "b" then next else error
    while sym = "a" do next
```

Question. What is an implementation in Python?

In [ ]:
```
src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos+1
    else: sym = chr(0) # end of input symbol

def A(): # A → b {a}
    if sym == 'b': nxt()
    else: raise Exception("'a' expected at " + str(pos))
    while sym == 'a': nxt()

def parse(s: str):
    global src, pos;
    src, pos = s, 0; nxt(); A()
    if sym != chr(0): raise Exception("unexpected characters at " + str(pos))

parse("baa")
```

Let us construct a parser for regular expressions. The symbols are characters:

```
expression  →  term { '|' term }
term  →  factor { factor }
factor  →  atom [ '*' | '+' | '?' ]
atom  →  plainchar | escapedchar | '(' expression ')'
plainchar  →  ' ' | '!' | '"' | '#' | '$' | '%' | '&' | '\'' | ',' | '-' | '.' | '/' |
     '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | ':' | ';' | '<' | '=' | '>' |
     '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
'O' |
     'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '[' | ']' | '^' | '_' |
     '`' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
'o' |
     'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '{' | '}' | '~'
escapedchar  →  '\' ( '(' | ')' | '*' | '+' | '?' | '\' | '|' )
```

Question. What are the conditions for recursive descend parsing for this grammar?

- `term` is not nullable, so `first(term) ∩ first({ '|' term }) = {}` is not needed
- `first('|' term) ∩ follow(expression) = {}`
- `first(factor) ∩ follow(factor) = {}`
- `atom` is not nullable, so `first(atom) ∩ first([ '*' | '+' | '?' ]) = {}` is not needed

- `first` of any two of `'*'`, `'+'`, `'?'` are disjoint
- `first` of any two of `plainchar`, `escapedchar`, `'(' expression ')'` are disjoint
- `first` of any two in `plainchar` are disjoint
- `'\'` is not nullable, so a condition of `escapedchar` is not needed
- `first` of any two of `'('`, `')'`, `'*'`, `'+'`, `'?'`, `'\\'`, `'|'` are disjoint

As the symbols are characters, the implementation uses strings rather than sets of characters for the `first` sets.

In [2]:
```
PlainChars = ' !"#$%&\',-./0123456789:;<=>@ABCDEFGHIJKLMNO' + \
             'PQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{}~'
EscapedChars = '()*+?\\|'
FirstFactor = PlainChars + '\\('

src: str; pos: int; sym: str

def nxt():
    global pos, sym
    if pos < len(src): sym, pos = src[pos], pos+1
    else: sym = chr(0) # end of input symbol

def expression(): # expression → term { '|' term }
    term()
    while sym == '|': nxt(); term()

def term(): # term → factor { factor }
    factor()
    while sym in FirstFactor: factor()

def factor(): # factor → atom [ '*' | '+' | '?' ]
    atom()
    if sym in '*+?': nxt()

def atom(): # atom → plainchar | escapedchar | '(' expression ')'
    if sym in PlainChars: nxt()
    elif sym == '\\':
        nxt()
        if sym in EscapedChars: nxt()
        else: raise Exception("invalid escaped character at " + str(pos))
    elif sym == '(':
        nxt(); expression()
        if sym == ')': nxt()
        else: raise Exception("')' expected at " + str(pos))
    else: raise Exception("invalid character at " + str(pos))

def parse(s: str):
    global src, pos;
    src, pos = s, 0; nxt(); expression()
    if sym != chr(0): raise Exception("unexpected character at " + str(pos))

#parse("a\$") # Exception: invalid escaped character at 3
#parse("a(b") # Exception: ')' expected at 3
#parse("a(" + chr(5) + ")") # invalid character at 3
#parse("a" + chr(5)) # unexpected character at 2
parse("(a*)*abcc")
```

In [ ]:
```
```