8. Generalized Parsing

Emil Sekerinski, McMaster University, March 2021

General Context-free Parsing

Earley's parser works with an arbitrary context-free grammar without backtracking. If the grammar is unambiguous, it produces a parse tree in quadratic time; if the grammar is ambiguous, it produces all parse trees in cubic time (in the length of the input). For most "practical" grammars, it produces a parse tree in linear time.

We assume that the start symbol S appears only on the left-hand side of one rule, $S \to \pi$; if that is not the case, a rule $S' \to S$ with a new start symbol S' can be added. Earley's parser is a top-down parser that constructs all possible derivations simultaneously: starting with S, nonterminals are eagerly expanded according to the all possible productions, rather than just a single production.

Let P be the set of productions and let the input be given by $x_1, \ldots, x_n$. Assume $x_{n+1} = \$$, where $\$$ is a symbol that does not occur anywhere in the grammar. For each position of the input a set s_i of *Earley items* is maintained. An (Earley) item is a grammar rule with the right-hand side split, visualized by $\bullet$, together with an index into the input string. An item $(A \to \sigma \bullet \omega, j)$ at position i means that A is attempted to be recognized at input position $j + 1$ and up to i the input $x_{j+1} \ldots x_i$ can be derived from σ, formally $\sigma \Rightarrow^* x_{j+1} \ldots x_i$. At each position i, the algorithm adds items to s_i in *predict* and *complete* steps and to s_{i+1} in *match* steps. The algorithm iterates over all items at one position. Since items are being added, a set, v, of visited items is maintained.

```
s_0 := {(S → • π, 0)}; for i = 1 to n do s_i := {}
for i = 0 to n do
    v := {}
    while v ≠ s_i do
        e :∈ s_i - v; v := v ∪ {e}
        case e of
            (A → σ • a ω, j) and a = x_{i+1}:         -- match (M)
                s_{i+1} := s_{i+1} ∪ {(A → σ a • ω, j)}
            (A → σ • B ω, j):                          -- predict (P)
                for B → μ ∈ P do
                    s_i := s_i ∪ {(B → • μ, i)}
            (A → σ •, j):                              -- complete (C)
                for (B → μ • A ξ, k) ∈ s_j do
                    s_i := s_i ∪ {(B → μ A • ξ, k)}
accept := (S → π •, 0) ∈ s_n
```

Consider the grammar:

$$E \to T \mid E + T$$
$$T \to F \mid T \times F$$
$$F \to a$$

The input $a + a \times a$ is accepted as $(S \to E \bullet, 0) \in s_5$.

Lines in bold correspond to the derivation.

	item	step
s_0:	$S \to \bullet E, 0$	
$(x_1 = a)$	$E \to \bullet T, 0$	P
	$E \to \bullet E + T, 0$	P
	$T \to \bullet F, 0$	P
	$T \to \bullet T \times F, 0$	P
	$F \to \bullet a, 0$	P
s_1:	$\mathbf{F \to a \bullet, 0}$	**M at $\mathbf{0}$**
$(x_2 = +)$	$\mathbf{T \to F \bullet, 0}$	C
	$\mathbf{E \to T \bullet, 0}$	C
	$T \to T \bullet \times F, 0$	C
	$S \to E \bullet, 0$	C
	$E \to E \bullet + T, 0$	C
s_2:	$E \to E + \bullet T, 0$	M at 1
$(x_3 = a)$	$T \to \bullet T \times F, 2$	P
	$T \to \bullet F, 2$	P
	$F \to \bullet a, 2$	P
s_3:	$\mathbf{F \to a \bullet, 2}$	**M at $\mathbf{2}$**
$(x_4 = \times)$	$\mathbf{T \to F \bullet, 2}$	**C**
	$E \to E + T \bullet, 0$	C
	$T \to T \bullet \times F, 2$	C
	$S \to E \bullet, 0$	C

	E → E • + T, 0	C
s_4 :	T → T × • F, 2	M at 3
(x_5 = a)	F → • a, 4	P
s_5 :	**F → a •, 4**	**M at 4**
(x_6 = $)	**T → T × F •, 2**	**C**
	E → E + T •, 0	C
	T → T • × F, 2	C
	S → E •, 0	**C**
	E → E • + T, 0	C

The Python implementation below assumes that each terminal and nonterminal is a single character, the grammar is represented by a tuple of productions, and each production is a string of the form `A→τ` where `A` is a nonterminal. The first production, `g[0]` in the implementation, defines the start symbol. Since in Python strings are indexed starting from `0`, an extra character, `^`, is prepended to the input. The sequence `a ω` in the algorithm corresponds to `τ` in the implementation and `A ξ` corresponds to `ν`.

In [ ]:
```
def parse(g: "grammar", x: "input"):
    global s
    n = len(x); x = '^' + x + '$'; S, π = g[0][0], g[0][2:]
    s = [{(S, '', π, 0)}] + [set() for _ in range(n)]#; print('   s[ 0 ]:', S, '→ •', π, ', 0')
    for i in range(n + 1):
        v = set() # visited items
        while v != s[i]:
            e = (s[i] - v).pop(); v.add(e) # pick an arbirary un-visited item
            A, σ, τ, j = e
            if len(τ) > 0 and τ[0] == x[i + 1]: # match, a == τ[0]
                f = (A, σ + τ[0], τ[1:], j)
                s[i + 1].add(f)#; print('M  s[', i + 1, ']:', f[0], '→', f[1], '•', f[2], ',', f[3])
            elif len(τ) > 0: # predict, B == ω[0]
                for f in ((r[0], '', r[2:], i) for r in g if r[0] == τ[0]):
                    s[i].add(f)#; print('P  s[', i, ']:', f[0], '→', f[1], '•', f[2], ',', f[3])
            else: # complete, len(τ) == 0
                for f in ((B, μ + ν[0], ν[1:], k) for (B, μ, ν, k) in s[j] if len(ν) > 0 and ν[0] == A):
                    s[i].add(f); #print('C  s[', i, ']:', f[0], '→', f[1], '•', f[2], ',', f[3])
    return (S, π, '', 0) in s[n]
```

In [ ]:
```
G1 = ("S→E", "E→a", "E→E+E")
```

In [ ]:
```
parse(G1, "a+a+a")
```

In [ ]:
```
grammar = ("S→E", "E→T", "E→E+T", "T→F", "T→T×F", "F→a")
```

In [ ]:
```
parse(grammar, "a+a×a")
```

The algorithm can be "animated" by uncommenting the `print` statements; the resulting set of items can also be observed:

In [ ]:
```
s
```

For efficiency, instead of using a set of items for an Earley state, a lists with a marker separating the items that have been visited and that still need to be visited can be used.

The number of items in s_i is proportional to i in the worst case. Matching and predicting need at most i steps for s_i, but completing may need i^2 steps, as adding an item may cause a previous set to be revisited. Summing i^2 for i from 0 to n is n^3, thus Earley's parser needs n^3 steps in the worst case.