Functional lexing and parsing

Prabhakar Ragde
Cheriton School of Computer Science
University of Waterloo

Outline of talk

- Functional vs imperative programming
- Bunch notation
- Finite state machines and regular expressions
- Context-free grammars
- Recursive descent and LL parsing
- Recursive ascent and LR parsing

Functional vs imperative programming

Functional programming:

- Based on the computation of new values by applying functions to old values
- Closer to mathematics (more conceptual)
- Typically uses recursion, lists, trees

Imperative programming:

- Based on the accretion of small changes to values
- Closer to machines (more efficient)
- Typically uses loops, arrays, frequently-modified variables

Lexing and parsing

- Major results worked out in ’60’s
- Computers were slow, memory and disk space very limited
- Dominant programming languages were low-level and imperative
- Things have changed (somewhat)
- But we still teach highly-optimized low-level algorithms!
- Now that we’re not scared of functions, recursion, lists, trees. . .

Bunches

Bunches are a variant of sets.

Intention: to simplify notation used in algorithms.

- A singleton bunch is identified with its only element.
- Bunches are "flat".

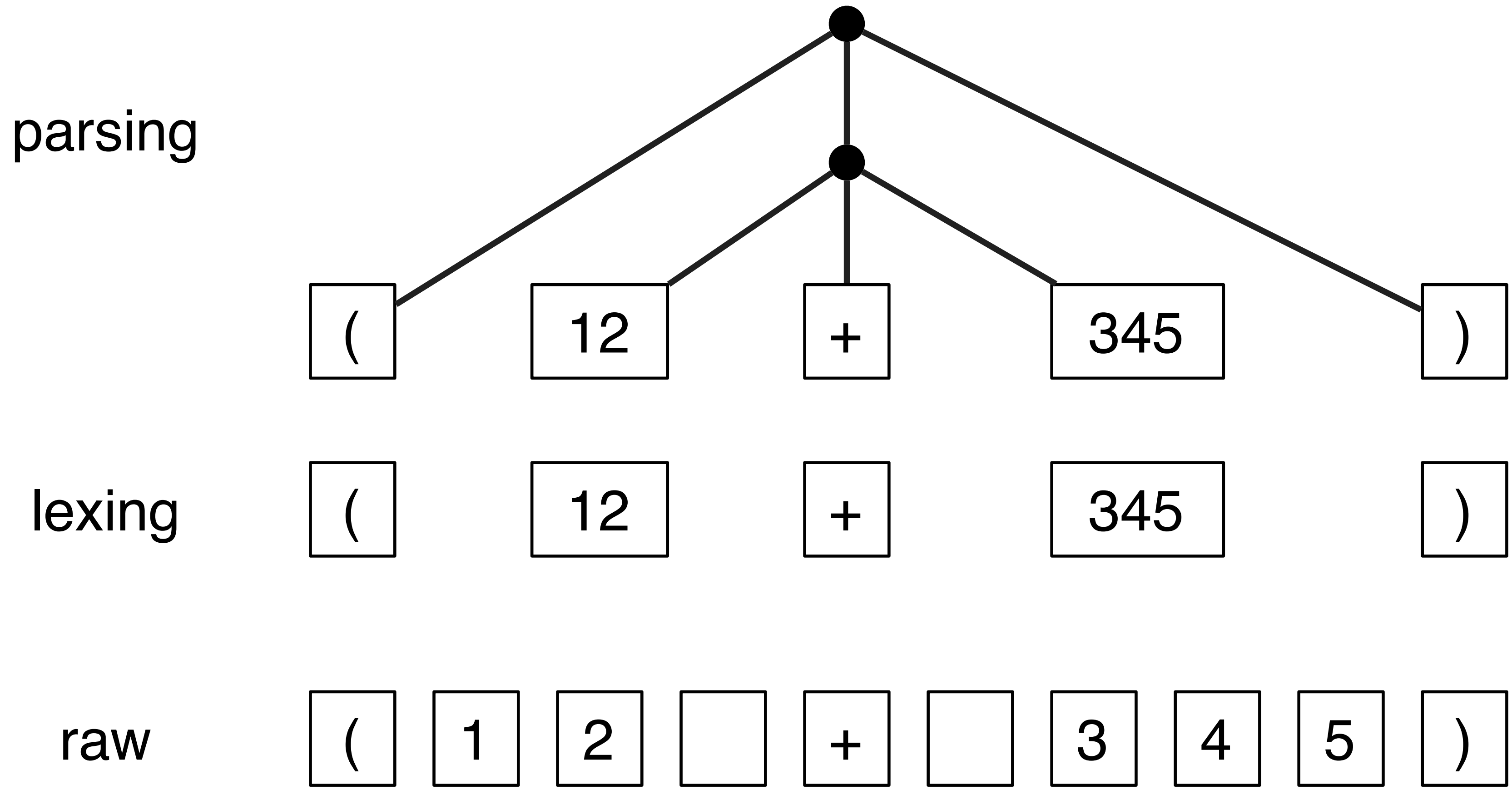
  They contain "atomic" values (not other bunches).
- Functions distribute over bunches.

  (The value of a function applied to a bunch is the bunch of the function applied to each value in the bunch.)

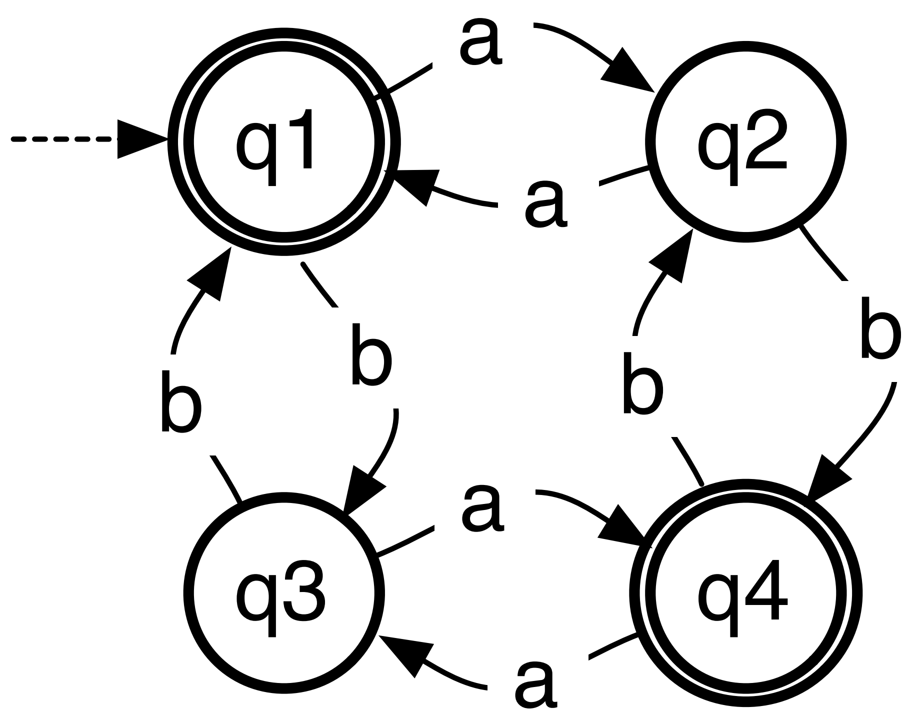
Bunch notation

- $\in$ and $\subseteq$ subsumed in $\longleftarrow$
- For union: use $\mid$ and ,
- Guards: $P \triangleright x$ means **if** P **then** x **else** ϕ.
- Implied "there exists" in guards;
  Instead of $f(x, y) = \{A(x, y) \mid \exists z\ P(x, y, z)\}$
  we write $P(x, y, z) \triangleright A(x, y)$.

Lexing and parsing



Finite state machines (FSMs)



Formally, a finite state machine M is:

- A set of states Q;
- A set of final states F;
- A start state s;
- An alphabet Σ;
- A transition function $\delta : Q \times \Sigma \rightarrow Q$.

The language L accepted by M is a subset of Σ^* (strings over the alphabet Σ).

We define a function $[q] : \Sigma^* \longrightarrow Q$ for each state q in Q.

$$[q](\sigma) = \begin{cases} q & \sigma = \epsilon \text{ (empty string)} \\ [\delta(q, \mathit{first}(\sigma))](\mathit{rest}(\sigma)) & \text{otherwise} \end{cases}$$

$$\sigma \leftarrow L \;\equiv\; [s](\sigma) \leftarrow F$$

“Interpreted”: implement $[\;]$ as a function of two arguments (q, σ)

“Compiled”: implement each $[q]$ as a separate function

```
;; "interpreted"
(define (run q sigma)
  (cond
    [(empty? q) q]
    [else (run (delta q (first sigma))
               (rest sigma))]))
```

```
// "interpreted"
q = s;
c = getchar();
while (c != EOF) {
    q = delta(q,c);
    c = getchar();
}
```

```
(define machine
  (local [(define (q1 sigma)
            (cond
              [(empty? sigma) true] ;; final state
              [else
                (case (first sigma)
                  [(a) (q2 (rest stream))]
                  [(b) (q3 (rest stream))]
                  [else false])]))
        (define (q2 sigma) . . . ) . . . ] ;; tedious repetition omitted
    q1))
```

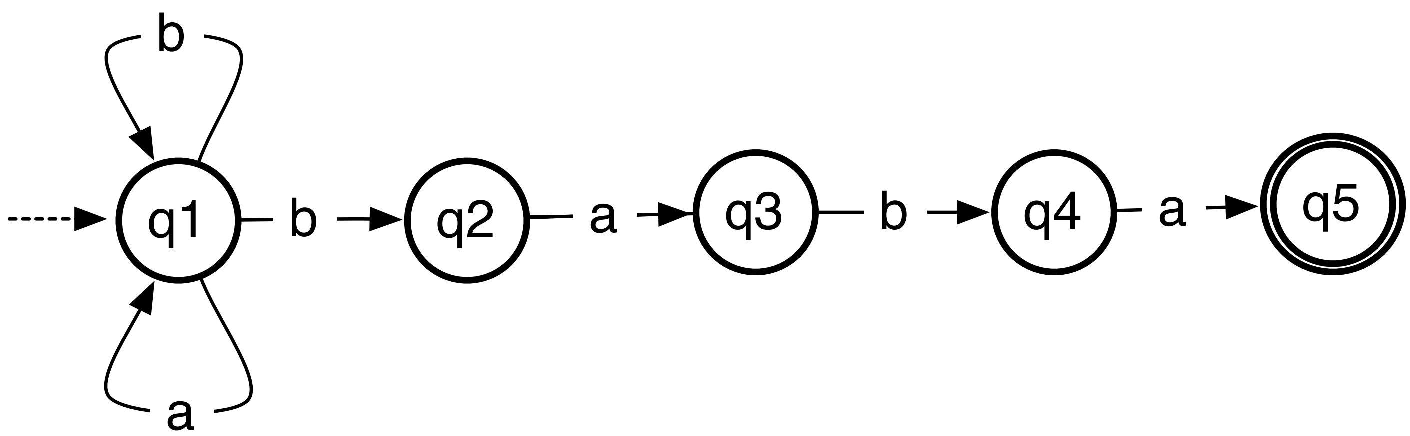
Desired syntax:

```
(define machine
  (automaton q1
    (q1 true  : (a -> q2)
                (b -> q3))
    (q2 false : (a -> q1)
                (b -> q4))
    (q3 false : (a -> q4)
                (b -> q1))
    (q4 true  : (a -> q3)
                (b -> q2))))
```

Using a macro (no omissions):

```
(define-syntax automaton
  (syntax-rules (: ->)
    [(_ init-state (state : result (symbol -> next) ...) ...)
      (local [(define (state sigma)
                  (cond
                    [(empty? sigma) result]
                    [else
                      (case (first sigma)
                        [(symbol) (next (rest sigma))] ...
                        [else false])])) ... ]
         init-state)]))
```

Nondeterministic finite state machines (NFSM)



Change: make δ a set-valued (or bunch-valued function).

Example: $\delta(q1, b) = q1, q2$.

In our definitions, we view q as a bunch (no changes needed).

We must look for a final state in the final bunch.

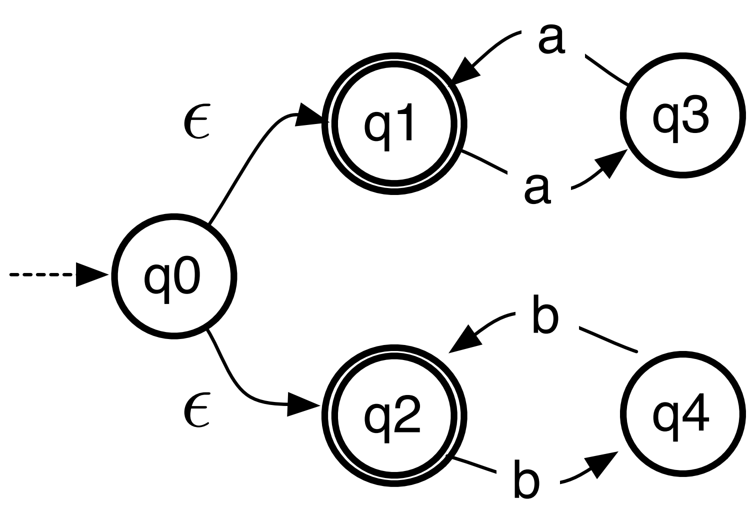
$$[q](\sigma) = \begin{cases} q & \sigma = \epsilon \\ [\delta(q, \mathit{first}(\sigma))](\mathit{rest}(\sigma)) & \text{otherwise} \end{cases}$$

$$\sigma \leftarrow L \;\equiv\; [s](\sigma) \cap F \text{ is nonempty}$$

“Interpreted”: Classical “simulation” of an NFA.

“Compiled”: the subset construction (NFA to DFA).

Adding ϵ-transitions



Change: add $eps(q) \leftarrow Q$

Example: $eps(q0) = q1, q2$.

To fix our definitions: define the $reach$ function.

$$reach(q) = q \mid eps(reach(q))$$

$$[q](\sigma) = \begin{cases} q & \sigma = \epsilon \\ [\delta(reach(q), first(\sigma))](rest(\sigma)) & \text{otherwise} \end{cases}$$

$$\sigma \leftarrow L \;\equiv\; [s](\sigma) \cap F \text{ is nonempty}$$

But how do we compute $reach(q)$?

Fixed-point computation

$reach(q)$ is a solution of $b = f(b)$ for:

$$f(b) = q \mid b \mid eps(b)$$

Here f is monotone: if $x \leftarrow y$, then $f(x) \leftarrow f(y)$.

One solution is

$$b = f(\phi) \mid f(f(\phi)) \mid f(f(f(\phi))) \ldots = \bigcup_{i=0}^{\infty} f^{(i)}(\phi)$$

This is the smallest solution, and it is a finite computation if the size of b is bounded. We say b is a **fixed point** of f.

Regular expressions (REs)

Examples: $(a+b)^*baba$, $1(0+1)^*$.

A RE R is either ϕ or ϵ or t ($t \leftarrow \Sigma$) or R_1R_2 or $R_1 + R_2$ or R_1^*.

$$
\begin{aligned}
L(R) = \;& R = \phi \triangleright \phi \mid R = \epsilon \triangleright \epsilon \mid R = t \triangleright t \\
& \mid R = R_1R_2 \triangleright L(R_1)L(R_2) \\
& \mid R = R_1 + R_2 \triangleright (L(R_1) \mid L(R_2)) \\
& \mid R = R_1^* \triangleright L(R_1)^*
\end{aligned}
$$

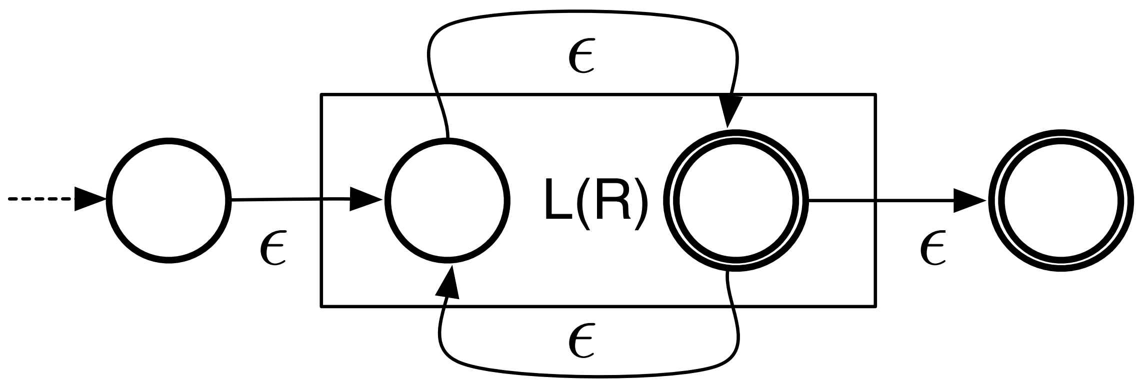
where

$$
\begin{aligned}
L_1L_2 = \;& x_1 \leftarrow L_1 \wedge x_2 \leftarrow L_2 \triangleright x_1x_2 \\
L^* = \;& x \leftarrow L \wedge y \leftarrow L^* \triangleright xy \text{ (fixed-point)}
\end{aligned}
$$

A RE has a recursive structure that is easily represented by a tree.

Various simplifications ($\epsilon R = R$, $\phi + R = R$, $\epsilon^* = \epsilon$) can be implemented with "smart constructors".

The traditional approach: convert an RE to an ϵ-NFA, then to an NFA, then to a DFA (or simulate the NFA).



Problem: adding some operators (e.g. $\neg$) becomes difficult.

A functional approach to REs

Goal: define the RE-valued $[R](\sigma)$, with specification $\gamma \leftarrow L([R](\sigma))$ if and only if $\sigma\gamma \leftarrow L(R)$.

First: define the RE-valued $nbl(R)$ (meaning "R is nullable").

$$nbl(R) \equiv \begin{cases} \epsilon & \epsilon \leftarrow L(R) \\ \phi & \text{otherwise} \end{cases}$$

$$\begin{aligned} nbl(R) = \quad & R = \phi \triangleright \phi \mid R = \epsilon \triangleright \epsilon \mid R = t \triangleright \phi \\ & \mid R = R_1 R_2 \triangleright nbl(R_1) nbl(R_2) \\ & \mid R = R_1 + R_2 \triangleright nbl(R_1) + nbl(R_2) \\ & \mid R = R_1^* \triangleright \epsilon \end{aligned}$$

Next: define $\partial_t(R)$, the "derivative with respect to t of R", with specification $t\alpha \leftarrow L(R)$ if and only if $\alpha \leftarrow L(\partial_t(R))$.

To compute $\partial_t(R)$:

$$
\begin{aligned}
\partial_t(R) = \quad & R = \phi \triangleright \phi \mid R = \epsilon \triangleright \phi \mid R = t \triangleright \epsilon \mid R = t' \triangleright \phi \\
& \mid R = R_1 R_2 \triangleright \partial_t(R_1) R_2 + nbl(R_1)\partial_t(R_2) \\
& \mid R = R_1 + R_2 \triangleright \partial_t(R_1) + \partial_t(R_2) \\
& \mid R = R_1^* \triangleright \partial_t(R_1) R_1
\end{aligned}
$$

Example: $\partial_b((a+b)^* baba) = aba + (a+b)^* baba$.

Now it is easy to define $[R](\sigma)$.

$$[R](\sigma) = \begin{cases} R & \sigma = \epsilon \\ [\partial_{\mathit{first}(\sigma)}(R)](\mathit{rest}(\sigma)) & \text{otherwise} \end{cases}$$

$$\sigma \leftarrow L(R) \;\equiv\; nbl([R](\sigma)) = \epsilon$$

“Interpreted”: structural recursion on R, tail recursion on σ.

“Compiled”: another DFA construction.

Adding new operators is much simpler.

Context-free grammars

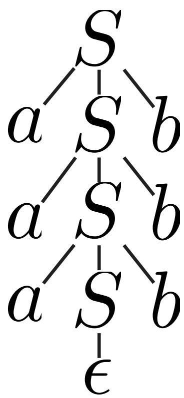
A grammar G consists of:

- A set of terminals T (here $a, b, c \ldots$);
- A set of nonterminals N (here $A, B, C \ldots$ or $\langle X \rangle$);

  (here, strings of the above are $\alpha, \beta, \ldots$)
- A set of rules R (e.g. $A \rightarrow aBa$);
- A starting nonterminal S.

Example grammar: $S \rightarrow aSb$, $S \rightarrow \epsilon$.

Rewriting: $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$.



Recognition: can a given string be produced by the grammar?

Parsing: produce the parse tree[s] for a given string.

Traditionally: a rewriting step is $\beta A\gamma \to \beta\alpha\gamma$ where $A \to \alpha$ is a rule.

$$\alpha \xrightarrow{*} \beta \;\equiv\; (\alpha = \beta) \vee (\alpha \xrightarrow{+} \beta)$$

$$\alpha \xrightarrow{+} \beta \;\equiv\; \exists\gamma(\alpha \to \gamma \wedge \gamma \xrightarrow{*} \beta)$$

$$L_G = \{\alpha \in T^* \mid S \xrightarrow{*} \alpha\}$$

Nontraditionally: define $L_G(\,\bullet\,)$ on strings from $(T|N)^*$.

$$L_G(t) = t$$

$$L_G(\epsilon) = \epsilon$$

$$L_G(\alpha\beta) = L_G(\alpha)L_G(\beta)$$

For $A \leftarrow N$, $L_G(A) = (A \to \alpha) \leftarrow R \triangleright L_G(\alpha)$

These equations in the unknowns $L_G(A)$ can be solved by a (possibly infinite) fixed-point computation, and $L_G = L_G(S)$.

Grammars and state machines

We can simulate an ϵ-NFSM using a grammar.

A state q corresponds to a nonterminal $\langle q \rangle$.

The start state s yields the rule $S \longrightarrow \langle s \rangle$.

A transition $\delta(q, c) = q'$ yields the rule $\langle q \rangle \longrightarrow c\langle q' \rangle$.

An ϵ-transition $q' \leftarrow eps(q)$ yields the rule $\langle q \rangle \longrightarrow \langle q' \rangle$.

A final state f yields the rule $\langle f \rangle \longrightarrow \epsilon$.

We will be using this idea later on.

Recognition of context-free languages

We define functions $[\gamma](\,\bullet\,)$ for $\gamma \leftarrow (T|N)^*$ with the specification
$[\gamma](\sigma) \equiv (\sigma = \sigma_1\sigma_2) \wedge \sigma_1 \leftarrow L_G(\gamma) \triangleright \sigma_2$.

$$
\begin{aligned}
[\epsilon](\sigma) &= \sigma \\
[t](\sigma) &= (\mathit{first}(\sigma) = t) \triangleright \mathit{rest}(\sigma) \\
[X\beta](\sigma) &= [\beta]([X](\sigma)) \\
[A](\sigma) &= (A \rightarrow \alpha) \leftarrow R \triangleright [\alpha](\sigma)
\end{aligned}
$$

This is a **recursive descent** parser.

$\sigma \leftarrow L \equiv \epsilon \leftarrow [S](\sigma)$

Example: $S \rightarrow aSb$, $S \rightarrow \epsilon$.

$$
\begin{aligned}
[S](aabb) &= [aSb](aabb) \mid [\epsilon](aabb) \\
&= [Sb]([a](aabb)) \mid aabb \\
&= [Sb](abb) \mid aabb \\
&= \epsilon, aabb \quad \text{because:} \\
[Sb](abb) &= [b]([S](abb)) \\
&= [b]([aSb](abb) \mid [\epsilon](abb)) \\
&= [b]([Sb](bb) \mid abb) \\
&= [b]([b]([S](bb))) \\
&= [b]([b]([aSb](bb) \mid [\epsilon](bb))) \\
&= [b]([b](bb)) = \epsilon
\end{aligned}
$$

Problem: left recursion

Example: $S \rightarrow Sa$, $S \rightarrow \epsilon$.

$$[S](\sigma) = [a]([S](\sigma)) \mid [\epsilon](\sigma)$$

Solution: Rewrite the grammar to eliminate left recursion.

Problem: it's less natural.

Problem: parse trees have the "wrong shape".

Left recursion arises naturally from left-associative operators.

Example: $a + b + c + d$ means $((a + b) + c) + d$.

We will come back to this problem.

For the time being, we avoid left recursion.

Problem: running time

Recursive descent is slow for some grammars without left recursion.

Example: $S \rightarrow aSS$, $S \rightarrow \epsilon$.

Recursive descent on a string of n a's takes exponential time.

Solution: memoization.

Create a table of previously computed function values.

There are $O(1)$ function "names" (nonterminals, suffixes of rule RHSs).

There are $O(n)$ arguments (suffixes of input).

A table entry (bunch) could be of size $O(n)$, and computing it could take $O(n^2)$ time.

Time complexity $O(n^3)$, space complexity $O(n^2)$.

Problem: still too much time/space used

Idea: use the next character in the input to eliminate unnecessary recursion (perhaps to the point of eliminating bunches).

$$[A](\sigma) = (A \to \alpha) \leftarrow R \triangleright [\alpha](\sigma)$$

If nothing in $L_G(\alpha)$ starts with $\mathit{first}(\sigma)$, don't call $[\alpha]$.

Complication: what if $\sigma \leftarrow [\alpha](\sigma)$ (i.e., $\epsilon \leftarrow L_G(\alpha)$)?

Then we must check if $\mathit{first}(\sigma)$ can follow A in some rewriting of S.

As a utility predicate, we define the Boolean-valued $nbl(\alpha) \equiv \epsilon \leftarrow L_G(\alpha)$ for α a suffix of a rule RHS.

$$
\begin{aligned}
nbl(\epsilon) &= true \\
nbl(t) &= false \\
nbl(X\beta) &= nbl(X) \wedge nbl(\beta) \\
nbl(A) &= (A \to \epsilon) \leftarrow R \triangleright true \mid (A \to \alpha) \leftarrow R \triangleright nbl(\alpha)
\end{aligned}
$$

This is a finite fixed-point computation.

For use in the "first" condition, we define $\mathit{first}(\alpha) \equiv t\beta \leftarrow L_G(\alpha) \triangleright t$ for α a suffix of a rule RHS.

$$
\begin{aligned}
\mathit{first}(\epsilon) &= \phi \\
\mathit{first}(t) &= t \\
\mathit{first}(X\beta) &= \mathit{first}(X) \mid \mathit{nbl}(X) \triangleright \mathit{first}(\beta) \\
\mathit{first}(X) &= (X \rightarrow t\alpha) \leftarrow R \triangleright t \\
&\mid (X \rightarrow Y\alpha) \leftarrow R \triangleright \mathit{first}(Y)
\end{aligned}
$$

This is a finite fixed-point computation.

For use in the "follow" condition, we define $follow(X)$ for $X \leftarrow N$.

Specification:

$follow(X) \equiv \alpha X \beta \leftarrow L_G(S) \wedge \mathit{first}(\beta) \leftarrow T \triangleright \mathit{first}(\beta)$.

$$\begin{aligned} follow(X) = {} & (A \rightarrow \alpha X \beta) \leftarrow R \wedge \mathit{first}(\beta) \leftarrow T \triangleright \mathit{first}(\beta) \\ & \mid (A \rightarrow \alpha X \beta) \leftarrow R \wedge nbl(\beta) \triangleright follow(A) \end{aligned}$$

This is a finite fixed-point computation.

We are finally ready to modify our recursive descent parser.

$$
\begin{aligned}
[\epsilon](\sigma) &= \sigma \\
[t](\sigma) &= (\mathit{first}(\sigma) = t) \triangleright \mathit{rest}(\sigma) \\
[X\beta](\sigma) &= [\beta]([X](\sigma)) \\
[A](\sigma) &= (A \to \alpha) \leftarrow R \;\wedge \\
&\qquad ((\mathit{first}(\alpha) = \mathit{first}(\sigma)) \vee (\mathit{nbl}(\alpha) \wedge \mathit{first}(\sigma) \leftarrow \mathit{follow}(A))) \\
&\qquad \triangleright [\alpha](\sigma)
\end{aligned}
$$

A grammar is **LL(1)** iff this is “deterministic” (there is at most one rule making the guard true).

For **LL(k)**, we define first_k and follow_k (k symbols of lookahead).

To obtain the conventional algorithm: make the recursion stack explicit.

```
push S
while (stack nonempty) {
    if (top is terminal t) {
        if (input symbol is t) {
            pop t, consume t
        } else {
            pop A
            push RHS of rule rewriting A
        }
}
accept iff input empty
```

Before we move towards LR parsing. . .

Some alternatives:

ANTLR and LL(*) parsing

Parsing expression grammars and packrat parsing

Parser combinators

A grammar transformation

Aim: to ensure at most two nonterminals on RHS of any rule.

Idea: create new nonterminals which are **items** of the form $\langle A \to \alpha \bullet \beta \rangle$, where $(A \to \alpha\beta) \leftarrow R$.

Given a grammar G, create E_G with the following rules:

$A \to \langle A \to \ \bullet \alpha \rangle$ for $(A \to \alpha) \leftarrow R$

$\langle A \to \alpha \bullet X\beta \rangle \to X\langle A \to \alpha X \bullet \beta \rangle$ for $(A \to \alpha X \beta) \leftarrow R$

$\langle A \to \alpha \bullet \rangle \to \epsilon$ for $(A \to \alpha) \leftarrow R$

G and E_G define the same language.

Apply recursive descent to E_G.

$$
\begin{aligned}
[t](\sigma) &= (\mathit{first}(\sigma) = t) \triangleright \mathit{rest}(\sigma) \\
[A](\sigma) &= (A \to \alpha) \leftarrow R \triangleright [A \to \ \bullet \alpha](\sigma) \\
[A \to \alpha \bullet X\beta](\sigma) &= [A \to \alpha X \bullet \beta]([X](\sigma)) \\
[A \to \alpha \bullet](\sigma) &= \sigma
\end{aligned}
$$

Inline $[t]$ and $[A]$, so all remaining functions have “item names”.

$$
\begin{aligned}
[A \to \alpha \bullet t\beta](\sigma) &= (\mathit{first}(\sigma) = t) \triangleright [A \to \alpha \bullet t\beta](\mathit{rest}(\sigma)) \\
[A \to \alpha \bullet B\beta](\sigma) &= (B \to \gamma) \leftarrow R \triangleright [A \to \alpha B \bullet \beta]([B \to \ \bullet \gamma](\sigma)) \\
[A \to \alpha \bullet](\sigma) &= \sigma
\end{aligned}
$$

If we add the rule $S' \rightarrow S$ to the grammar, then
$\sigma \leftarrow L_G(S) \equiv \epsilon \leftarrow [S' \rightarrow \bullet S](\sigma)$.

This is just a variation on recursive descent.

Memoized, it is still an $O(n^3)$ algorithm.

And it still has problems with left recursion.

A better grammar transformation can deal with left recursion.

We say A is a **left corner** of α if by rewriting the leftmost symbol repeatedly, we get from α to $A\beta$.

We'll abbreviate this as $lc(A, \alpha)$.

$$lc(A, \alpha) = (A = \mathit{first}(\alpha)) \vee ((first(\alpha) \to \gamma) \leftarrow R \wedge lc(A, \gamma))$$

This is another finite fixed-point computation.

We add nonterminals of the form $\langle X, A \to \alpha \bullet \beta \rangle$, meaning, intuitively, that we've seen α, we hope to see β, and $lc(X, \beta)$.

We use this idea to create a grammar F_G equivalent to G, with rules of the five types listed on the next slide.

Type 1: $S \rightarrow \langle S \rightarrow \ \bullet \alpha \rangle$ for $(S \rightarrow \alpha) \leftarrow R$

Type 2: $\langle X, A \rightarrow \alpha \bullet X\beta \rangle \rightarrow \langle A \rightarrow \alpha X \bullet \beta \rangle$ for $(A \rightarrow \alpha X\beta) \leftarrow R$

Type 3: $\langle A \rightarrow \alpha \bullet \beta \rangle \rightarrow t\langle t, A \rightarrow \alpha \bullet \beta \rangle$ for $(A \rightarrow \alpha\beta) \leftarrow R \wedge lc(t, \beta)$.

Type 4: $\langle X, A \rightarrow \alpha \bullet \beta \rangle \rightarrow \langle B \rightarrow X \bullet \delta \rangle \langle B, A \rightarrow \alpha \bullet \beta \rangle$ for $(B \rightarrow X\delta), (A \rightarrow \alpha\beta) \leftarrow R \wedge lc(B, \beta)$.

Type 5: $\langle A \rightarrow \alpha \bullet \rangle \rightarrow \epsilon$ for $(A \rightarrow \alpha) \leftarrow R$

Claim: this is not left-recursive if G is not cyclic (we cannot rewrite A and get A) and has no ϵ-rules (that can be fixed with a sixth type of rule).

Example: $S \rightarrow Sx$, $S \rightarrow y$.

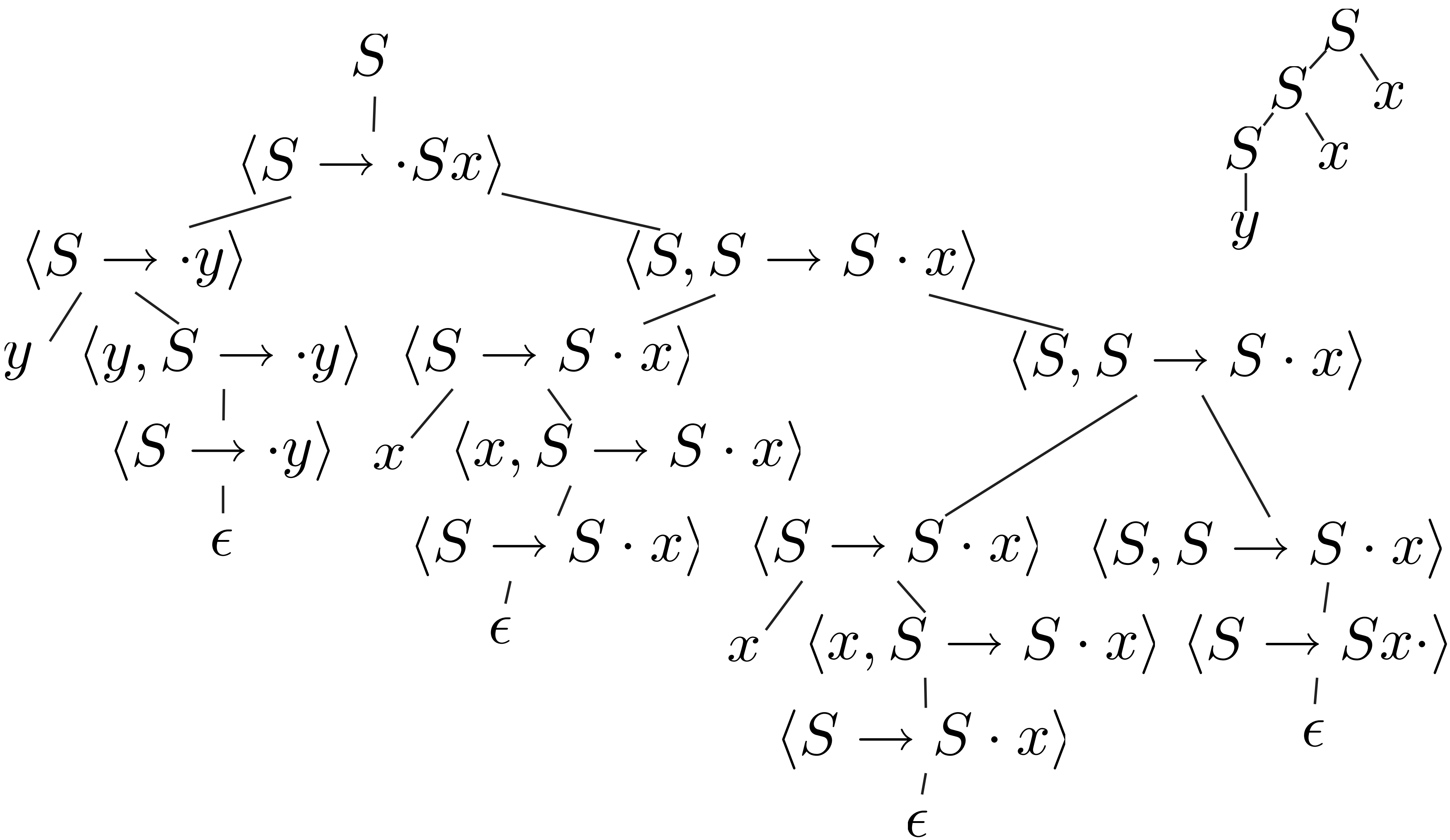
Type 1: $S \rightarrow \langle S \rightarrow \bullet Sx \rangle$, $S \rightarrow \langle S \rightarrow \bullet y \rangle$.

Type 2: $\langle S, S \rightarrow \bullet Sx \rangle \rightarrow \langle S \rightarrow S \bullet x \rangle$,
$\langle x, S \rightarrow S \bullet x \rangle \rightarrow \langle S \rightarrow Sx \bullet \rangle$, $\langle y, S \rightarrow \bullet y \rangle \rightarrow \langle S \rightarrow y \bullet \rangle$.

Type 3: $\langle S \rightarrow S \bullet x \rangle \rightarrow x \langle x, S \rightarrow S \bullet x \rangle$,
$\langle S \rightarrow \bullet y \rangle \rightarrow y \langle y, S \rightarrow y \bullet \rangle$.

Type 4: $\langle S \rightarrow \bullet Sx \rangle \rightarrow \langle S \rightarrow S \bullet x \rangle \langle S, S \rightarrow S \bullet x \rangle$,
$\langle S \rightarrow \bullet y \rangle \rightarrow \langle S \rightarrow \bullet y \rangle \langle S, S \rightarrow S \bullet x \rangle$.

Type 5: $\langle S \rightarrow Sx \bullet \rangle \rightarrow \epsilon$, $\langle S \rightarrow y \bullet \rangle \rightarrow \epsilon$.



S
$\langle S \rightarrow \cdot Sx \rangle$
$\langle S \rightarrow \cdot y \rangle$
$\langle S, S \rightarrow S \cdot x \rangle$
y
$\langle y, S \rightarrow \cdot y \rangle$
$\langle S \rightarrow S \cdot x \rangle$
$\langle S, S \rightarrow S \cdot x \rangle$
$\langle S \rightarrow \cdot y \rangle$
x
$\langle x, S \rightarrow S \cdot x \rangle$
ϵ
$\langle S \rightarrow S \cdot x \rangle$
$\langle S \rightarrow S \cdot x \rangle$
$\langle S, S \rightarrow S \cdot x \rangle$
ϵ
x
$\langle x, S \rightarrow S \cdot x \rangle$
$\langle S \rightarrow Sx \cdot \rangle$
$\langle S \rightarrow S \cdot x \rangle$
ϵ
ϵ
S
S
x
S
x
y

We apply recursive descent to F_G.

We'll have functions of the form $[A \rightarrow \alpha.\beta](\sigma)$ which, intuitively, removes from σ something obtainable by rewriting β.

We will represent the $[X, A \rightarrow \alpha \bullet \beta](\sigma)$ functions as $[\overline{A \rightarrow \alpha \bullet \beta}](X, \sigma)$.

The resulting parser is shown on the next slide.

$$[A \to \alpha \bullet \beta](\sigma) = lc(\mathit{first}(\sigma), \beta) \triangleright [\overline{A \to \alpha \bullet \beta}](\mathit{first}(\sigma)), \mathit{rest}(\sigma))$$

$$\mid lc(B, \beta) \triangleright [\overline{A \to \alpha \bullet \beta}](B, \sigma)$$

$$\mid \beta = \epsilon \triangleright \sigma$$

$$[\overline{A \to \alpha \bullet \beta}](X, \sigma) = (\beta = X\gamma) \triangleright [A \to \alpha X \bullet \gamma](\sigma)$$

$$\mid lc(B, \beta) \wedge (B \to X\delta) \leftarrow R$$

$$\triangleright [\overline{A \to \alpha \bullet \beta}](B, [B \to X \bullet \delta](\sigma))$$

This is a **recursive ascent** parser.

Memoized, the recursive ascent parser still has $O(n^3)$ time complexity and $O(n^2)$ space complexity when parsing strings of length n.

It can handle left-recursive grammars, and it can be augmented to produce a compact representation of all possible parse trees of the parsed string.

We need to add one more idea in order to design LR parsers with $O(n)$ time and space complexity (for a restricted set of grammars).

Recall our simulation of a finite-state machine by a grammar.

Let's examine some of the rules in F_G.

Type 3: $\langle A \to \alpha \bullet \beta \rangle \to t \langle t, A \to \alpha \bullet \beta \rangle$ for $(A \to \alpha\beta) \leftarrow R \wedge lc(t, \beta)$.

This looks like a simulated state transition on t.

Type 2: $\langle X, A \to \alpha \bullet X\beta \rangle \to \langle A \to \alpha X \bullet \beta \rangle$ for $(A \to \alpha X \beta) \leftarrow R$

This could be viewed as a state transition on X.

Type 4: $\langle X, A \to \alpha \bullet \beta \rangle \to \langle B \to X \bullet \delta \rangle \langle B, A \to \alpha \bullet \beta \rangle$ for $(B \to X\delta), (A \to \alpha\beta) \leftarrow R \wedge lc(B, \beta)$.

This is like an ϵ-transition from working on X to working on B.

The analogy is not perfect, but if:

- a rule is like a transition, and
- not knowing what rule to apply is like not knowing what transition to make,

then we can use a variant on our definition of the meaning of a nondeterministic finite state machine (NFSM).

We will write functions $[q]$ where q is no longer just an item, but a bunch of items.

Just as our NFSM functions could be thought of as “trying all transitions in parallel”, so our parsing functions will try all possible “transitions” defined by F_G “in parallel”.

A bunch of items is called a **state** in the classic presentation.

LR parsing

For each state q, we'll define $[q](\sigma)$ with specification

$(A \to \alpha \bullet \beta) \leftarrow q \wedge \sigma = \sigma_1\sigma_2 \wedge \sigma_1 \leftarrow L_G(\beta) \triangleright (A \to \alpha \bullet \beta, \sigma_2)$.

Here's how we recognize strings generated by our grammar:

$$\sigma \leftarrow L_G(S) \equiv (S' \to S, \epsilon) \leftarrow [S' \to \bullet S](\sigma)$$

Our "ϵ-transitions" will be:

$$\begin{aligned} eps(q) = {} & (A \to \alpha \bullet B\beta) \leftarrow q \wedge (B \to \nu) \leftarrow R \\ & \triangleright B \to \bullet \nu \end{aligned}$$

As before, $reach(q) = q \mid eps(reach(q'))$.

(This is called $predict$ in the classical presentation, and has a description in terms of left corners.)

We then get the transition function:

$$goto(q, X) = (A \to \alpha \bullet X\beta) \leftarrow reach(q) \triangleright A \to \alpha X \bullet \beta$$

This defines the **LR(0) automaton** of the grammar.

We now apply the recursive ascent idea.

We define auxiliary functions $[\overline{q}]$ with specification:

$$\begin{aligned}[\overline{q}](X, \sigma) = (A \to \alpha \bullet \beta) \leftarrow R \wedge lc(X, \beta) \wedge \\ \sigma = \sigma_1\sigma_2 \wedge \sigma_1 \leftarrow L_G(rest(\beta)) \\ \triangleright [A \to \alpha.\beta](\sigma_2)\end{aligned}$$

Working out the details, we get the **LR(0)** parser on the next slide.

$$
\begin{aligned}
[q](\sigma) = {} & [\overline{q}](\mathit{first}(\sigma), \mathit{rest}(\sigma)) \\
& \quad \mid (B \rightarrow \bullet) \leftarrow \mathit{reach}(q) \triangleright [\overline{q}](B, \sigma) \\
& \quad \mid (A \rightarrow \alpha \bullet) \leftarrow q \triangleright (A \rightarrow \alpha \bullet, \sigma) \\
[\overline{q}](X, \sigma) = {} & (A \rightarrow \alpha \bullet X\gamma) \leftarrow q \wedge \\
& \quad (A \rightarrow \alpha X \bullet \gamma, \sigma') \leftarrow [\mathit{goto}(q, X)](\sigma) \\
& \quad \triangleright (A \leftarrow \alpha X \bullet \gamma, \sigma') \\
& \quad \mid C \rightarrow \bullet X\delta \leftarrow \mathit{reach}(q) \wedge \\
& \quad (C \rightarrow X \bullet \delta, \sigma') \leftarrow [\mathit{goto}(q, X)](\sigma) \\
& \quad \triangleright [\overline{q}](C, \sigma')
\end{aligned}
$$

If $[q]$ is deterministic (single-valued) for all q, the grammar is **LR(0)**.

Possible sources of nondeterminism:

- if a state q has more than one item of the form $A \to \alpha \bullet$
  (this is a **reduce-reduce** conflict)
- if a state q has an item $A \to \alpha \bullet$ but also $goto(q, t)$ is nonempty, which will be a problem if $t = \mathit{first}(\sigma)$
  (this is a **shift-reduce** conflict)

For **LR(k)**, add lookahead k as with LL(k).

This only vaguely resembles the classical description of an LR parser.

To get the classical presentation:

- make the recursion stack explicit (the “state stack”), allowing the use of a while loop
- view the input argument as a stack (the “symbol stack”) augmented by items in the case of $[q]$ and the extra argument in the case of $[\overline{q}]$, allowing input to be read a character at a time
- implement various optimizations (e.g. items never need to be pushed onto the symbol stack)

In summary

- LR parsing is hard to understand
- It gets harder when you start from the wrong end
- There are easier lexers and parsers for learning and experiment
- A functional approach facilitates understanding
  of both lexing and parsing

References

Lex Augusteijn, John Brzozowski, E.C.H. Hehner, Shriram Krishnamurthi, Frank Kruseman Aretz, René Leermakers, Peter Norvig, Scott Owens, John Reppy, Michael Sperber, Peter Thiemann, Aaron Turon.