Introduction to Parsing

Detmar Meurers: Intro to Computational Linguistics I
OSU, LING 684.01, February 3., 2004

Parsers and criteria to evaluate them

- Function of a parser:

  – grammar + string → analysis trees

- Main criteria for evaluating parsers:

  – correctness
  – completeness
  – efficiency

Overview

- What is a parser?

- Under what criteria can they be evaluated?

- Parsing strategies

  – top-down vs. bottom-up
  – left-right vs. right-left
  – depth-first vs. breadth-first

- Implementing different types of parsers:

  – Basic top-down and bottom-up
  – More efficient algorithms

Correctness

A parser is **correct** iff for every grammar and for every string, every analysis returned by parser is an actual analysis.

Correctness is nearly always required (unless simple post-processor could eliminate wrong analyses)

Completeness

A parser is **complete** iff for every grammar and for every string, every correct analysis is found by the parser.

- In theory, always desirable.

- In practice, essential to find the 'relevant' analysis first (possibly using heuristics).

- For grammars licensing an infinite number of analyses this means: there is no analysis that the parser could not find.

Complexity classes

If n is the length of the string to be parsed, one can distinguish the following complexity classes:

- **constant**: amount of work does not depend on n

- **logarithmic**: amount of work behaves like $log_k(n)$ for some constant k

- **polynomial**: amount of work behaves like n^k, for some constant k. This is sometimes subdivided into the cases
  - **linear** $(k = 1)$
  - **quadratic** $(k = 2)$
  - **cubic** $(k = 3)$
  - . . .

- **exponential**: amount of work behaves like k^n, for some constant k.

Efficiency

- One can reason about complexity of (parsing) algorithms by considering how it will deal with bigger and bigger examples.

- For practical purposes, the factors ignored by such analyses are at least as important.
  - profiling using typical examples important
  - finding the (relevant) first parse vs. all parse

- Memoization of complete or partial results is essential to obtain efficient parsing algorithms.

Complexity and the Chomsky hierarchy

Grammar type	Worst-case complexity of recognition
regular (3)	linear
context-free (2)	cubic (n^3)
context-sensitive (1)	exponential
general rewrite (0)	undecidable

Recognition with type 0 grammars is **recursively enumerable**: if a string x is in the language, the recognition algorithm will succeed, but it will not return if x is not in the language.

Parsing strategies

1. What do we start from?

   - top-down vs. bottom-up

2. In what order is the string or the RHS of a rule looked at?

   - left-to-right, right-to-left, island-driven, . . .

3. How are alternatives explored?

   - depth-first vs. breadth-first

Direction of processing II: Bottom-up

Data-driven processing is Bottom-up:

- Start with the sentence.

- For each substring σ of each sentential form $\alpha\sigma\beta$, find each grammar rule $N \rightarrow \sigma$ to obtain all sentential forms $\alpha N \beta$.

- If the start symbol is among the sentential forms obtained, the sentence is part of the language.

Problem: Epsilon rules ($N \rightarrow \epsilon$).

Direction of processing I: Top-down

Goal-driven processing is Top-down:

- Start with the start symbol

- Derive sentential forms.

- If the string is among the sentences derived this way, it is part of the language.

The order in which one looks at a RHS

Left-to-Right

- Use the leftmost symbol first, continuing with the next to its right

How are alternatives explored? I. Depth-first

- At every choice point: Pursue a single alternative completely before trying another alternative.

- State of affairs at the choice points needs to be remembered. Choices can be discarded after unsuccessful exploration.

- Depth-first search is not necessarily complete.

Problem for top-down, left-to-right, depth-first processing:

- left-recursion
  For example, a rule like N' → N' PP leads to non-termination.

How are alternatives explored? II. Breadth-first

- At every choice point: Pursue every alternative for one step at a time.

- Requires serious bookkeeping since each alternative computation needs to be remembered at the same time.

- Search is guaranteed to be complete.

Compiling and executing DCGs in Prolog

- DCGs are a grammar formalism supporting any kind of parsing regime.

- The standard translation of DCGs to Prolog plus the proof procedure of Prolog results in a parsing strategy which is

  – top-down
  – left-to-right
  – depth-first

Implementing parsers

- Data structures: a parser configuration

- Top-down parsing

  – formal characterization
  – Prolog implementation

- Bottom-up parsing

  – formal characterization
  – Prolog implementation

- Towards more efficient parsers:

  – Left-corner
  – Remembering subresults

An example grammar (parser/simple/grammar.pl)

```
% defining grammar rule operator
:- op(1100,xfx,'--->').

% lexicon:              % syntactic rules:
vt  ---> [saw].         s  ---> [np, vp].
det ---> [the].         vp ---> [vt, np].
det ---> [a].           np ---> [det, n].
n   ---> [dragon].      n  ---> [adj, n].
n   ---> [boy].
adj ---> [young].
```

Top-down parsing

- **Start configuration** for recognizing a string ω: $<S, \omega>$

- **Available actions**:
  - **consume**: remove an expected terminal a from the string
    $<a\alpha, a\tau> \mapsto <\alpha, \tau>$
  - **expand**: apply a phrase structure rule
    $<A\beta, \tau> \mapsto <\alpha\beta, \tau>$ if $A \to \alpha \in P$

- **Success configuration**: $<\epsilon, \epsilon>$

A parser configuration

Assuming a left-to-right order of processing, a **configuration** of a parser can be encoded by a pair of

- a stack as auxiliary memory

- the string remaining to be recognized

More formally, for a grammar $G = (N, \Sigma, S, P)$, a parser configuration is a pair $<\alpha, \tau>$ with $\alpha \in (N \cup \Sigma)^*$ and $\tau \in \Sigma^*$

A top-down parser in Prolog (parser/simple/td_parser.pl)

```
:- op(1100,xfx,'--->').

% Start
td_parse(String) :- td_parse([s],String).

% Success
td_parse([],[]).

% Consume
td_parse([H|T],[H|R]) :-
        td_parse(T,R).

% Expand
td_parse([A|Beta],String) :-
        (A ---> Alpha),
        append(Alpha,Beta,Stack),
        td_parse(Stack,String).
```
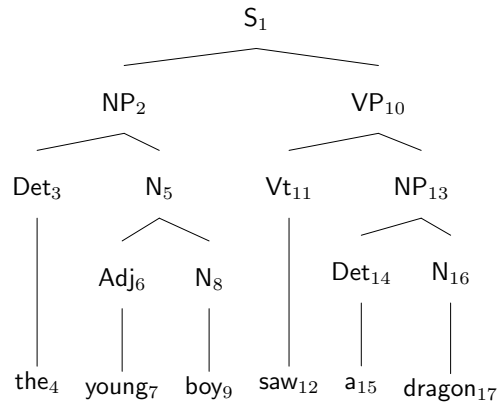
Top-Down, left-right, depth-first tree traversal

$S \rightarrow NP \; VP$
$VP \rightarrow Vt \; NP$
$NP \rightarrow Det \; N$
$N \rightarrow Adj \; N$

$Vt \rightarrow saw$
$Det \rightarrow the$
$Det \rightarrow a$
$N \rightarrow dragon$
$N \rightarrow boy$
$Adj \rightarrow young$

```
< [vt, np], [saw, the, dragon] >
< [saw, np], [saw, the, dragon] >
< [np], [the, dragon] >
< [det, n], [the, dragon] >
< [the, n], [the, dragon] >
< [n], [dragon] >
< [dragon], [dragon] >
< [], [] >
```

A trace (parser/simple/grammar.pl, parser/simple/td_parser_trace.pl)

```
?- td_parse([the,young,boy,saw,the,dragon]).
< [s], [the, young, boy, saw, the, dragon] >
< [np, vp], [the, young, boy, saw, the, dragon] >
< [det, n, vp], [the, young, boy, saw, the, dragon] >
< [the, n, vp], [the, young, boy, saw, the, dragon] >
< [n, vp], [young, boy, saw, the, dragon] >
< [dragon, vp], [young, boy, saw, the, dragon] >
< [boy, vp], [young, boy, saw, the, dragon] >
< [adj, n, vp], [young, boy, saw, the, dragon] >
< [young, n, vp], [young, boy, saw, the, dragon] >
< [n, vp], [boy, saw, the, dragon] >
< [dragon, vp], [boy, saw, the, dragon] >
< [boy, vp], [boy, saw, the, dragon] >
< [vp], [saw, the, dragon] >
```

Bottom-up parsing

- **Start configuration** for recognizing a string ω: $< \epsilon, \omega >$

- **Available actions**:

  - **shift**: turn to the next terminal a of the string
    $< \alpha, a\tau > \mapsto < \alpha a, \tau >$
  - **reduce**: apply a phrase structure rule
    $< \beta\alpha, \tau > \mapsto < \beta A, \tau >$ if $A \rightarrow \alpha \in P$

- **Success configuration**: $< S, \epsilon >$

A shift-reduce parser in Prolog (parser/simple/sr_parser.pl)

```prolog
:- op(1100,xfx,'--->').

sr_parse(String) :- sr_parse([],String).    % Start

sr_parse([s],[]).                            % Success

sr_parse(Stack,String) :-                    % Reduce
        append(Beta,Alpha,Stack),
        (A ---> Alpha),
        append(Beta,[A],NewStack),
        sr_parse(NewStack,String).

sr_parse(Stack,[Word|String]) :-             % Shift
        append(Stack,[Word],NewStack),
        sr_parse(NewStack,String).
```
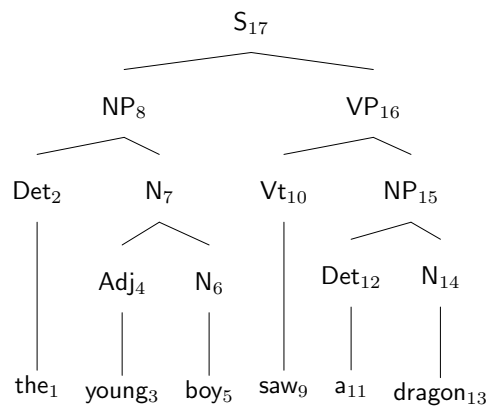
25

Bottom-up, left-right, depth-first tree traversal

S → NP VP
VP → Vt NP
NP → Det N
N → Adj N

Vt → saw
Det → the
Det → a
N → dragon
N → boy
Adj → young

26

A trace (parser/simple/grammar.pl, parser/simple/sr_parser_trace.pl)

```
| ?- sr_parse([the,young,boy,saw,the,dragon]).
START: <[],[the,young,boy,saw,the,dragon]>
     Reduce []? no
     Shift "the"
<[the],[young,boy,saw,the,dragon]>
     Reduce [the] => det
<[det],[young,boy,saw,the,dragon]>
     Reduce [det]? no
     Reduce []? no
     Shift "young"
<[det,young],[boy,saw,the,dragon]>
     Reduce [det,young]? no
     Reduce [young] => adj
```

27

```
<[det,adj],[boy,saw,the,dragon]>
     Reduce [det,adj]? no
     Reduce [adj]? no
     Reduce []? no
     Shift "boy"
<[det,adj,boy],[saw,the,dragon]>
     Reduce [det,adj,boy]? no
     Reduce [adj,boy]? no
     Reduce [boy] => n
<[det,adj,n],[saw,the,dragon]>
     Reduce [det,adj,n]? no
     Reduce [adj,n] => n
<[det,n],[saw,the,dragon]>
     Reduce [det,n] => np
<[np],[saw,the,dragon]>
     Reduce [np]? no
     Reduce []? no
     Shift "saw"
```

28

```
<[np,saw],[the,dragon]>
     Reduce [np,saw]? no
     Reduce [saw] => vt
<[np,vt],[the,dragon]>
     Reduce [np,vt]? no
     Reduce [vt]? no
     Reduce []? no
     Shift "the"
<[np,vt,the],[dragon]>
     Reduce [np,vt,the]? no
     Reduce [vt,the]? no
     Reduce [the] => det
<[np,vt,det],[dragon]>
     Reduce [np,vt,det]? no
     Reduce [vt,det]? no
     Reduce [det]? no
     Reduce []? no
     Shift "dragon"
```

```
<[np,vt,det,dragon],[]>
     Reduce [np,vt,det,dragon]? no
     Reduce [vt,det,dragon]? no
     Reduce [det,dragon]? no
     Reduce [dragon] => n
<[np,vt,det,n],[]>
     Reduce [np,vt,det,n]? no
     Reduce [vt,det,n]? no
     Reduce [det,n] => np
<[np,vt,np],[]>
     Reduce [np,vt,np]? no
     Reduce [vt,np] => vp
<[np,vp],[]>
     Reduce [np,vp] => s
<[s],[]>
SUCCESS!
```

A shift-reduce parser for grammars in CNF
using difference lists to encode the string (parser/simple/cnf_sr.pl)

```
:- op(1100,xfx,'--->').

recognise(String) :- recognise([],String) % Start

recognise([s],[]).            % Success

recognise([Y,X|Rest],S) :-    % Reduce
    (LHS ---> [X,Y]),
    recognise([LHS|Rest],S).

recognise(Stack,[Word|S]) :- % Shift
    Cat ---> [Word],
    recognise([Cat|Stack],S).
```

A trace (parser/simple/grammar.pl, parser/simple/cnf_sr_trace.pl)

```
| ?- recognise([the,young,boy,saw,the,dragon]).
START: <[],[the,young,boy,saw,the,dragon]>
     Shift "the" as "det"
<[det],[young,boy,saw,the,dragon]>
     Shift "young" as "adj"
<[adj,det],[boy,saw,the,dragon]>
     Reduce [det,adj]? no
     Shift "boy" as "n"
<[n,adj,det],[saw,the,dragon]>
     Reduce [adj,n] => n
<[n,det],[saw,the,dragon]>
     Reduce [det,n] => np
<[np],[saw,the,dragon]>
     Shift "saw" as "vt"
```

```
<[vt,np],[the,dragon]>
     Reduce [np,vt]? no
     Shift "the" as "det"
<[det,vt,np],[dragon]>
     Reduce [vt,det]? no
     Shift "dragon" as "n"
<[n,det,vt,np],[]>
     Reduce [det,n] => np
<[np,vt,np],[]>
     Reduce [vt,np] => vp
<[vp,np],[]>
     Reduce [np,vp] => s
<[s],[]>
SUCCESS!
```