LL(k): Deterministic top-down parsing

Parsing
ISCL-BA-06

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2020/21

version: 6e4a5d2+ @2021-03-07

So far ...

- Formal languages and automata
- General parsing techniques
  - Top-down – Bottom-up
  - Directional – non-directional
- Chart parsing
  - CKY
  - Early

Coming next:

- Deterministic context-free parsing
- Probabilistic context-free parsing
- Dependency parsing

Recap: top-down parsing

- General idea: try to generate the input using the grammar rules
  - Initialize with the start symbol
  - Rewrite each non terminal, replacing them with matching RHS in the grammar
  - When there are multiple options, follow one, backtrack and follow others when done
  - Repeat until input sentence is generated (or failed)
- If we always expand the left-most symbol first, the parser is directional, the resulting derivation is the left-most derivation
- Parsing proceeds with two actions:

predict expanding all RHS of the left-most non-terminal
match if the left-most item is a terminal, it has to match the next input symbol

Top-down parsing: an example

S → NP VP	NP → d AN	NP → AN
VP → v NP	AN → a AN	AN → n

Matched	Sent. Form	Input	Action
d n v a n	$		match n
n	a AN $	n	match ✗
	a AN VP $	d n v a n	P: match ✗

Top-down parsing

- If we follow the predicted productions, we obtain a *leftmost* derivation
- Lots of unnecessary work, backtracking because of useless predictions
- Most of the unnecessary work is done in *predict*
- In this lecture we will look at ways to reduce this
- For some grammars, the unnecessary predictions can be completely avoided, resulting in a *deterministic* parser

Recursive descent parser

- *Recursive descent* parsers are top-down, recursive parsers where each non-terminal is implemented as a procedure
- For each symbol on a RHS, we either
  - call the sub-procedure (another nonterminal)
  - or match the input symbol

```
procedure A( )
    select a rule A → X1, ..., Xk
    for i = 1 to k do
        if Xi is a nonterminal then
            call Xi()
        else if Xi = current input then
            advance the input pointer
        else
            return error
```


Recursive descent parser

some remarks

- The interesting idea is that now the parser is a program in a(ny) programming language
- In its general form a recursive descent parser is a backtracking parser
- If we can select a rule deterministically, then we can get a deterministic parser
- Deterministic parsing generally requires a *lookahead* mechanism:
  - Given the non-terminal to expand/rewrite, and the next input symbol(s), for some grammars, we can build a table that can deterministically guide a parser

Table driven parsing

S → NP VP	NP → d AN	NP → AN
VP → v NP	AN → a AN	AN → n

non-term.	input (lookahead)				
	d	a	n	v	$
S	S → NP VP	S → NP VP	S → NP VP	S → NP VP	
NP	NP → d AN	NP → AN	NP → AN		
VP				VP → v NP	
AN		AN → a AN	AN → n		

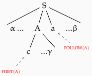
Table driven parsing: example

non-term.	input (lookahead)				
	d	a	n	v	$
S	S → NP VP	S → NP VP	S → NP VP	S → NP VP	
NP	NP → d AN	NP → AN	NP → AN		
VP				VP → v NP	
AN		AN → a AN	AN → n		

Matched	Sent. Form	Input	Action
d n v a n	$		match n

FIRST and FOLLOW sets

- FIRST and FOLLOW sets are useful for both top-down and bottom-up table driven parsers
- FIRST set of a non-terminal A, FIRST(A), is the set of initial terminal symbols of all strings generated by A
- FOLLOW set of a non-terminal A, FOLLOW(A), is the set of initial terminals that may follow any A according to the grammar
- Both sets generalize to any sentential form
- FIRST and FOLLOW sets are also useful for error recovery during parsing



Computing the FIRST set

- The FIRST set of a terminal symbol contains only itself
- To compute the FIRST sets of nonterminals, repeat the following until no new symbols are added to any of the sets
  1. For each rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ in the grammar,
     - place all terminals in FIRST(Y_i) if $Y_1 Y_2 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$
     - if ϵ is in all FIRST(Y_i) for all i = 1, ..., k, add ϵ to FIRST(X)
  2. if the rule processed is $X \rightarrow \epsilon$, add ϵ to FIRST(X)
- Then, FIRST set of any sentential form, FIRST($X_1 X_2 \ldots X_k$) can be computed:
  - For $i = 1, \ldots, k$
    1. Add all non-ϵ symbols from X_i to FIRST($X_1 X_2 \ldots X_k$)
    2. If $\epsilon \notin$ FIRST(X_i), stop
  - if $\epsilon \in$ FIRST(X_i) for all $i = 1, \ldots, k$, add ϵ to FIRST($X_1 X_2 \ldots X_k$)

Computing the FOLLOW set

- Calculate the FIRST sets
  1. Place $ in the FOLLOW(S)
  2. For a production $A \rightarrow \alpha B \beta$, add everything in FIRST(β) except ϵ to FOLLOW(B)
  3. For a production $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, add all items in FOLLOW(A) to FOLLOW(B)
  4. Repeat 3 until no more items are added to any of the FOLLOW sets

LL(1) grammars

- A grammar is called and LL(1) grammar, if we can find a table similar to our example:
  - If there is only a single prediction for each (non-terminal, lookahead) pair, then the grammar is an LL(1) grammar
- L's stand for ***Left-to-right*** and ***Leftmost** derivation*, (1) indicates the number of lookahead symbols needed
- If we increase the number of lookahead symbols, we get LL(k) grammars
- LL(k) grammar can be parsed with a top-down parser without backtracking
- Not every context free grammar is LL(k)
- But, programming language grammars are mostly LL(1)

LL(1) grammars
formal definition

- If a grammar is LL(1) then whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two rules in the grammar, then
  - The sets of non-terminals of strings derived from α and β are disjoint
  - Only one (or none) of α and β can derive the empty string
  - If $\beta \overset{*}{\Rightarrow} \epsilon$, α cannot start with a terminal that may follow A
- In other words:
  - FIRST(α) and FIRST(β) are disjoint
  - if ϵ is in FIRST(α), then FIRST(β) and FOLLOW(A) are disjoint sets

Construction of LL(1) table

- If there are no ϵ productions, the table can be easily constructed from the FIRST sets
- Otherwise, after computing FIRST and FOLLOW sets, the following procedure fills the LL(1) table
  - For each rule $A \rightarrow \alpha$ in the grammar
    1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to table cell [A, a]
    2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A) add $A \rightarrow \alpha$ to table cell [A, b]

Example
calculating FIRST sets

$S \rightarrow BA \quad A \rightarrow aBA \mid \epsilon \quad B \rightarrow CD \quad D \rightarrow bCD \mid \epsilon \quad C \rightarrow cSc \mid d$

- Repeat until no additions
  1. For each $X \rightarrow Y_1 Y_2 \ldots Y_k$
     - place all terminals in FIRST(Y_i) if $Y_1 Y_2 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$
     - if ϵ is in all FIRST(Y_i) for all i = 1, ..., k, add ϵ to FIRST(X)
  2. if the rule processed is $X \rightarrow \epsilon$, add ϵ to FIRST(X)

FIRST(S) = {c,d}
FIRST(A) = {a,ϵ}
FIRST(B) = {c,d}
FIRST(C) = {c,d}
FIRST(D) = {b,ϵ}

Example
calculating FOLLOW sets

$S \rightarrow BA \quad A \rightarrow aBA \mid \epsilon \quad B \rightarrow CD \quad D \rightarrow bCD \mid \epsilon \quad C \rightarrow cSc \mid d$

1. Place \$ in the FOLLOW(S)
2. For a production $A \rightarrow \alpha B \beta$, add everything in FIRST(β) except ϵ to FOLLOW(B)
3. For $A \rightarrow \alpha B$, or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, add items in FOLLOW(A) to FOLLOW(B)

	S	A	B	C	D
FIRST	{c,d}	{a,ϵ}	{c,d}	{c,d}	{b,ϵ}

FIRST(S) = {c,\$}
FIRST(A) = {c,\$}
FIRST(B) = {a,c,\$}
FIRST(C) = {a,b,c,\$}
FIRST(D) = {a,c,\$}

Example
constructing the LL(1) table

$S \rightarrow BA \quad A \rightarrow aBA \mid \epsilon \quad B \rightarrow CD \quad D \rightarrow bCD \mid \epsilon \quad C \rightarrow cSc \mid d$

	S	A	B	C	D
FIRST	{c,d}	{a,ϵ}	{c,d}	{c,d}	{b,ϵ}
FOLLOW	{c,\$}	{c,\$}	{a,c,\$}	{a,b,c,\$}	{a,c,\$}

- For each rule $A \rightarrow \alpha$ in the grammar
  1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to table cell [A, a]
  2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A) add $A \rightarrow \alpha$ to table cell [A, b]

	a	b	c	d	\$
S			BA	BA	
A	aBA		ϵ		ϵ
B			CD	CD	
C			cSc	d	
D	ϵ	bCD	ϵ		ϵ

Summary

- LL(1) grammars can be parsed deterministically (without backtracking) using top-down parsers
- Like any top-down parser, left-recursion needs additional care
- Not every context free grammar is LL(k), but programming language grammars are mostly LL(1)
- LL(k) parsing is intuitive and relatively easy to construct by hand, but LR(k) grammars (bottom-up, deterministic) are more powerful (next lecture)
- Suggested reading: Grune and Jacobs (2007, ch.8), Aho et al. (2007, Section 4.4)

Next:

- Deterministic bottom-up parsing
- Suggested reading: Grune and Jacobs (2007, ch.9), Aho et al. (2007, Section 4.5–4.7)

Acknowledgments, references, additional reading material

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2007). *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley. ISBN: 9780321486813.

Grune, Dick and Ceriel J.H. Jacobs (2007). *Parsing Techniques: A Practical Guide*. second. Monographs in Computer Science. The first edition is available at `http://dickgrune.com/Books/PTAPG_1st_Edition/BookBody.pdf`. Springer New York. ISBN: 9780387689548.

Exercise
compute the FIRST and FOLLOW sets, and LL(1) table for $S \rightarrow iEtSQ \mid a \quad Q \rightarrow eS \mid \epsilon \quad E \rightarrow b$

blank

blank

blank

blank

blank

blank