Follow sets. LL(1) Parsing Table

Exercise Introducing Follow Sets

Compute nullable, first for this grammar:

stmtList ::= ε | stmt stmtList
stmt ::= assign | block
assign ::= **ID = ID ;**
block ::= **beginof ID** stmtList **ID ends**

Describe a parser for this grammar and explain how it behaves on this input:

beginof myPrettyCode
x = u;
y = v;
myPrettyCode **ends**

How does a recursive descent parser look like?

```
def stmtList =
  if (???) {}                what should the condition be?
  else { stmt; stmtList }
def stmt =
  if (lex.token == ID) assign
  else if (lex.token == beginof) block
  else error(“Syntax error: expected ID or beginonf”)
…
def block =
  { skip(beginof); skip(ID); stmtList; skip(ID); skip(ends) }
```

Problem Identified

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Problem parsing stmtList:

- **ID** could start alternative stmt stmtList
- **ID** could **follow** stmt, so we may wish to parse **ε** that is, do nothing and return

- For nullable non-terminals, we must also compute what **follows** them

LL(1) Grammar - good for building recursive descent parsers

- Grammar is LL(1) if for each nonterminal X
  - first sets of different alternatives of X are disjoint
  - if nullable(X), first(X) must be disjoint from follow(X) and only one alternative of X may be nullable
- For each LL(1) grammar we can build recursive-descent parser
- Each LL(1) grammar is unambiguous
- If a grammar is not LL(1), we can sometimes transform it into equivalent LL(1) grammar

Computing if a token can **follow**

first$(B_1 \ldots B_p) = \{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

follow$(X) = \{a \in \Sigma \mid S \Rightarrow \ldots \Rightarrow \ldots Xa \ldots\}$

There exists a derivation from the start symbol that produces a sequence of terminals and nonterminals of the form ...Xa...
(the token a follows the non-terminal X)

Rule for Computing Follow

Given $X ::= YZ$ (for reachable X)

then **first**$(Z) \subseteq$ **follow**(Y)

and **follow**$(X) \subseteq$ **follow**(Z)

now take care of nullable ones as well:

For each rule $X ::= Y_1 \dots Y_p \dots Y_q \dots Y_r$

follow(Y_p) should contain:

- **first**$(Y_{p+1}Y_{p+2}\dots Y_r)$
- also **follow**(X) if **nullable**$(Y_{p+1}Y_{p+2}Y_r)$

Compute nullable, first, follow

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID** stmtList **ID ends**

Is this grammar LL(1)?

Conclusion of the Solution

The grammar is not LL(1) because we have

- nullable(stmtList)
- first(stmt) ∩ follow(stmtList) = {**ID**}

- If a recursive-descent parser sees **ID**, it does not know if it should
  - finish parsing stmtList or
  - parse another stmt

Table for LL(1) Parser: Example

S ::= B **EOF**
(1)

B ::= ε | B **(**B**)**
(1) **(2)**

empty entry:
when parsing S,
if we see) ,
report error

nullable: B

first(S) = { **(**, **EOF** }

follow(S) = {}

first(B) = { **(** }

follow(B) = { **)**, **(**, **EOF** }

Parsing table:

	EOF	(	)
S	{1}	{1}	{}
B	{1}	{1,2}	{1}

parse conflict - choice ambiguity:
grammar not LL(1)

1 is in entry because **(** is in follow(B)
2 is in entry because **(** is in first(B(B))

Table for LL(1) Parsing

Tells which alternative to take, given current token:

choice : Nonterminal x Token -> Set[Int]

A ::= **(1)** $B_1 \ldots B_p$
| **(2)** $C_1 \ldots C_q$
| **(3)** $D_1 \ldots D_r$

if $t \in \text{first}(C_1 \ldots C_q)$ add 2 to choice(A,t)

if $t \in \text{follow}(A)$ add K to choice(A,t) where K is nullable

For example, when parsing A and seeing token t

choice(A,t) = {2} means: parse alternative 2 ($C_1 \ldots C_q$)

choice(A,t) = {3} means: parse alternative 3 ($D_1 \ldots D_r$)

choice(A,t) = {} means: report syntax error

choice(A,t) = {2,3} : not LL(1) grammar

General Idea when parsing nullable(A)

A ::= $B_1 \dots B_p$
| $C_1 \dots C_q$
| $D_1 \dots D_r$

```
def A =
  if (token ∈ T1) {
    $B_1 \dots B_p$
  else if (token ∈ (T2 U $T_F$)) {
    $C_1 \dots C_q$
  } else if (token ∈ T3) {
    $D_1 \dots D_r$
  } // no else error, just return
```

where:

T1 = **first**($B_1 \dots B_p$)
T2 = **first**($C_1 \dots C_q$)
T3 = **first**($D_1 \dots D_r$)
T_F = **follow**(A)

Only one of the alternatives can be nullable (here: 2nd)
T1, T2, T3, T_F should be pairwise **disjoint** sets of tokens.

Algorithm for parsing arbitrary grammars
Parse trees, syntax trees
Ambiguity and priorities

Chomsky's Classification of Grammars

On Certain Formal Properties of Grammars

(N. Chomsky, INFORMATION AND CONTROL 9., 137-167 (1959)

- type 0: arbitrary string-rewrite rules
  - equivalent to Turing machines!
  - e X b => e X   e X => Y
- type 1: context sensitive, RHS always larger
  - O(n)-space Turing machines
  - a X b => a c X b
- **type 2: context free - one LHS nonterminal**
- type 3: regular grammars (regular languages)

Parsing Context-Free Grammars

Decidable even for type 1 grammars,
(by eliminating epsilons - Chomsky 1959)

We choose $O(n^3)$ CYK algorithm - simple

Better complexity possible:

General Context-Free Recognition in Less than Cubic Time, JOURNAL OF COMPUTER AND SYSTEM SCIENCES 10, 308--315 (1975)

- problem reduced to matrix multiplication - n^k for k between 2 and 3

More practical algorithms known:

J. Earley **An efficient context-free parsing algorithm,** Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA (1968)

can be adapted so that it automatically works in quadratic or linear time for better-behaved grammars

CYK Parsing Algorithm

C:

John **Cocke** and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

Y:

Daniel H. **Younger** (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2): 189–208.

K:

T. **Kasami** (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.

CYK Algorithm Can Handle Ambiguity

Why Parse General Grammars

- General grammars can be ambiguous: for some strings, there are multiple parser trees
- Can be impossible to make grammar unambiguous
- Some languages are more complex than simple programming languages
  - mathematical formulas:

    $x = y \land z$? $(x=y) \land z$ $x = (y \land z)$
  - natural language:

    I saw the man with the telescope.
  - future programming languages

Ambiguity 1

1)

2)

I saw the man with the telescope.

Ambiguity 2

Time flies like an arrow.

Indeed, time passes by quickly.

Those special “time flies” have an “arrow” as their favorite food.

You should regularly measure how fast the flies are flying, using a process that is much like an arrow.

…

Two Steps in the Algorithm

1) Transform grammar to normal form

   called Chomsky Normal Form

2) Parse input using transformed grammar

   dynamic programming algorithm

“a method for solving complex problems by breaking them down into simpler steps.
It is applicable to problems exhibiting the properties of overlapping subproblems”

Dynamic Programming to Parse Input

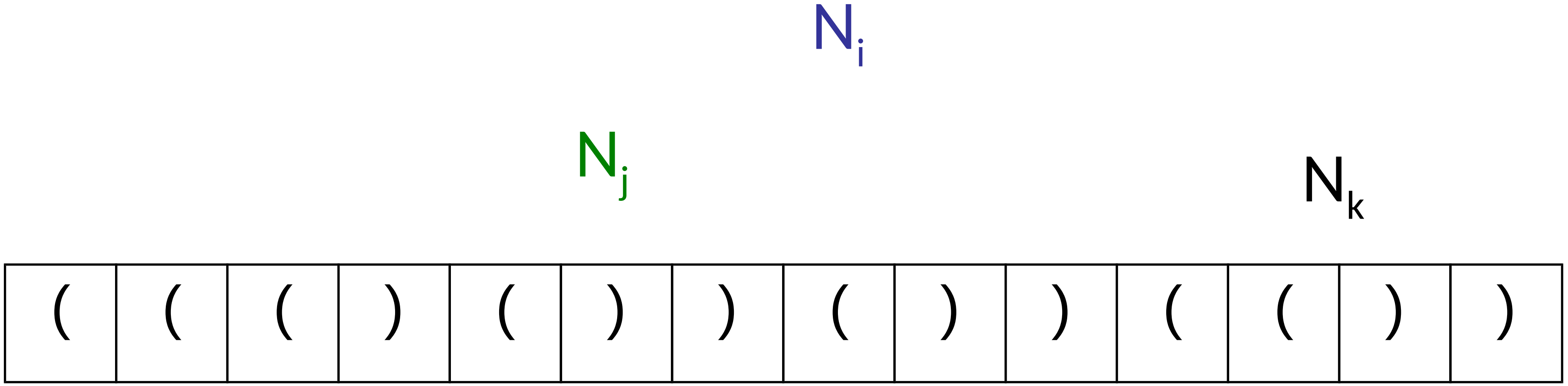
Assume Chomsky Normal Form, 3 types of rules:

$S' \rightarrow \varepsilon \mid S$ (only for the start non-terminal)

$N_i \rightarrow t$ (names for terminals)

$N_i \rightarrow N_j\ N_k$ (just **2** non-terminals on RHS)

Decomposing long input:



N_i

N_j

N_k

(	(	(	)	(	)	)	(	)	)	(	(	)	)

find all ways to parse substrings of length 1,2,3,…

Balanced Parentheses Grammar

Original grammar G

B → ε | B B | (B)

Modified grammar in Chomsky Normal Form:

B1 → ε | B B | O M | O C
B → B B | O M | O C
M → B C
O → '('
C → ')'

Terminals: ( )

Nonterminals: B, B1, O, C, M, B

Parsing an Input

B1 → ε | B B | O M | O C
B → B B | O M | O C
M → B C
O → '('
C → ')'

6								
5								
4								
3								
2								
1	O	O	C	O	C	O	C	C
	(	(	)	(	)	(	)	)
	1	2	3	4	5	6	8	9

Algorithm Idea

w_{pq} – substring from p to q

d_{pq} – all non-terminals that could expand to w_{pq}

Initially d_{pp} has $N_{w(p,p)}$

key step of the algorithm:

if $X \rightarrow Y\ Z$ is a rule,
Y is in $d_{p\,r}$, and
Z is in $d_{(r+1)q}$

then put X into d_{pq}

($p <= r < q$),

in increasing value of (q-p)

Algorithm

```
INPUT:  grammar G in Chomsky normal form
             word w to parse using G
OUTPUT: true iff (w in L(G))
N = |w|
var d : Array[N][N]
for p = 1 to N {
   d(p)(p) = {X | G contains X->w(p)}
   for q in {p + 1 .. N} d(p)(q) = {} }
for k = 2 to N // substring length
  for p = 0 to N-k // initial position
    for j = 1 to k-1 // length of first half
      val r = p+j-1; val q = p+k-1;
      for (X::=Y Z) in G
        if Y in d(p)(r) and Z in d(r+1)(q)
           d(p)(q) = d(p)(q) union {X}
return  S in d(0)(N-1)
```

What is the running time as a function of grammar size and the size of input?

O(   )

(	(	)	(	)	(	)	)

Number of Parse Trees

Let w denote word ()()()

- it has two parse trees

Give a lower bound on number of parse trees of the word w^n (n is positive integer)

w^5 is the word

()()() ()()() ()()() ()()() ()()()

CYK represents all parse trees compactly

- can re-run algorithm to extract first parse tree, or enumerate parse trees one by one