Vertalerbouw HC 9:

Parsing: Top-Down & LL(1)

do 3 mei 2001 5+6

Theo Ruys

INF 5037 - tel. 3716
ruys@cs.utwente.nl

Overview HC9

- Ch. 8 - Parsing

  See also [Aho, Sethi & Ullman 1986] for a more thorough discussion.

  8.1 Context-free Grammars
  8.2 Top-Down Parsing
  8.3 LL(1) Grammars

Introduction

- Parser (= syntax analyser)
  - checks whether the input program is syntactically correct
  - usually specified by a context-free grammar
    - regular expressions ⇒ finite-state automaton
    - context-free grammar ⇒ stack automaton
  - is usually augmented with actions for
    - context constraints
    - code optimisation and generation
  - not only for "programming languages", but for all programs that process structured data
  - parsing strategies:
    - top-down parsing
    - bottom-up parsing

Context-Free Grammars (1)

A Context-Free Grammar (CFG) G is defined by a 4-tuple (N, T, P, S)

- S: start symbol
- P: production rules
- T: finite set of terminals (tokens that occur)
- N: finite set of non-terminals (define structure)

$S \in N$

$A \Rightarrow \beta$

$A \in N$

$\beta \in (N \cup T)^*$

$V = N \cup T$

- *Example:*

  G = ({A,B}, {a,b,c}, A, P)
  where P: A ⇒ aA
  A ⇒ B
  B ⇒ b
  B ⇒ c

regexp: a*(b|c)

Notational conveniences:
- only provide the production rules
- use choice operator: A ⇒ aA | B

Context-Free Grammars (2)

- A CFG is a specification of a rewrite system.
  - CFGs are used to derive strings of terminals.
- Notation:

α, β, γ, δ	$\in (N \cup T)^* = V^*$	string of symbols
u, v, w	$\in T^*$	string of terminals
X, Y, Z	$\in (N \cup T)$	single grammar symbol
A, B, C, D	$\in N$	single non-terminal
a, b, c	$\in T$	single terminal

- 1-step derivation:
  $\alpha A \gamma \Rightarrow \alpha\beta\gamma$ using production rule: A ⇒ B

CFGs (3)

$\alpha, \beta, \gamma \in (N \cup T)^*$
$u, v, w \in T^*$
$X, Y, Z \in (N \cup T)$
$A, B, C \in N$
$a, b, c \in T$

- Derivation $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
  - left-most derivation
    if $\alpha = w \in T^*$, then $wA\gamma \Rightarrow_l \alpha\beta\gamma$
  - right-most derivation
    if $\gamma = w \in T^*$, then $\alpha Aw \Rightarrow_r \alpha\beta w$
  - zero or more steps $\alpha \Rightarrow^* \beta$
  - one or more steps $\alpha \Rightarrow^+ \beta$
- Recursion
  - left-recursive derivation
    if $A \Rightarrow^+ A\alpha$ then the CFG is left-recursive
  - right-recursive derivation
    if $A \Rightarrow^+ \alpha A$ then the CFG is right-recursive

CFGs (4)

α, β, γ ∈ (N ∪ T)*
u, v, w ∈ T*
X, Y, Z ∈ (N ∪ T)
A, B, C ∈ N
a, b, c ∈ T

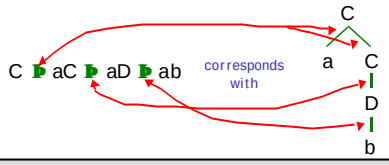
- Terminology (cont.)
  - if S ⇒* β then β is a sentential form
  - if S ⇒+ w then w is a sentence
- *Example:*
  C → aC | D
  D → b | c

C ⇒ aC ⇒ aaC ⇒ aaaC ⇒ aaaD ⇒ aaab

sentential forms | sentence

Context-Free Grammars (5)

- Context-Free Language (CFL)
  - CFL = the set of all sentences derived from a CFG
    CFL(G) = { w | S ⇒+ w }
  - Previous example (C → aC | D D → b | c):
    CFL = { b, c, ab, ac, aab, aac, aaab, aaac, … }
- Parse tree: another representation of a derivation



Context-Free Grammars (6)

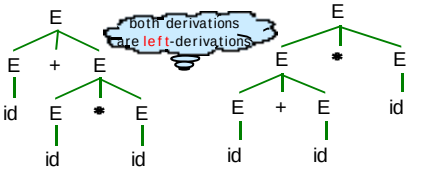
- *Example:*
  E → E + E
  E → E * E
  E → id

In this case, G does not define the relative priorities of + and *

Two ways of deriving a sentence in the CFL corresponding to G: G is ambiguous!

id+id*id

both derivations are left-derivations



E ⇒ E+E ⇒ id+E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id

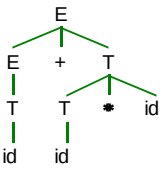
E ⇒ E*E ⇒ E+E*E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id

Context-Free Grammars (7)

- *Unambiguous grammar:*
  E → E + T
  E → T
  T → T * id
  T → id

T (*) has priority over E (+)

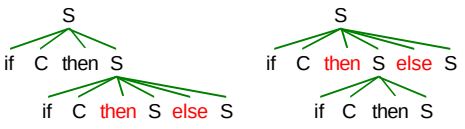
An extra nonterminal T is used to solve the priority/ambiguity problem



Context-Free Grammars (8)

- Infamous dangling-else problem:
  S → if C then S
  S → if C then S else S

if C then if C then S else S



Context-Free Grammars (9)

- Problem:
  verifying that the language L is generated by a grammar G
  i.e. to prove that: L(G) = L
  - → verify: if S ⇒* w then w ∈ L
  - ¬ verify: if w ∈ L then S ⇒* w
- *Example:*
  L is the language consisting of balanced parantheses.
  G: S → '(' S ')' S
  S → ε

Proof sketch: use induction on the number of derivation steps and the length of the sentence

CFGs (10)

S → '(' S ')' S
S → ε

→ *verify that every generated string is balanced*

n=1 (one step derivation)
- ε is balanced

n>1
- assume that all strings are balanced for <n-step derivations
- consider an "n-step derivation", which will be of the form:
  S ⇒ (S) S ⇒* (x) S ⇒* (x) y
- x and y must be balanced (both are cases of <n derivations), hence (x)y is balanced.

CFGs (11)

S → '(' S ')' S
S → ε

¬ *verify that all "balanced-parenthesis" strings can be generated from S*

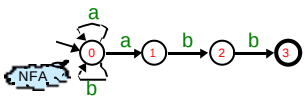
n=0 (length of sentence)
- ε is derivable from S

n>0
- assume that every string of length <2n is derivable
- consider a balanced string of length 2n (for n≥1)
  - let (x) be the shortest prefix of the balanced string
  - the balanced string can be written as (x)y where x and y are both balanced, and are both <2n in length; therefore they are derivable;
  - Hence, we can find
    S ⇒ (S) S ⇒* (x) S ⇒* (x) y

Context-Free Grammars (12)

- REs vs CFGs
  A RE can always be expressed as a CFG.
  Algorithm:
  1. ∀ state s, create a nonterminal A_s
  2. ∀ transition labelled a, write $A_s \rightarrow aA_t$
  3. For accept states s, write $A_s \rightarrow \varepsilon$
  4. The start state is the begin symbol.

RE: {a|b}*abb



CFG:
$A_0 \rightarrow aA_0$
$A_0 \rightarrow bA_0$
$A_0 \rightarrow aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \varepsilon$

Context-Free Grammars (13)

- So:
  - A RL is always context-free.
  - A CFL is usually not regular.
- *Examples:*
  $L_1 = \{a^n \mid n \geq 1\}$
  regular: aa*
  $L_2 = \{a^n b^n \mid n \geq 1\}$
  not regular
  context free: S → aSb | ab
  $L_3 = \{a^n b^n c^n \mid n \geq 1\}$
  not regular
  not context free

Idea: a RL/CFL can always be written in a form so that a substring/state is repeated

The "Pumping Lemmas" for regular expressions and grammars should be used to prove that a language L is not a RL or CFL. (see [Sudkamp 1991])

A finite automaton cannot keep count

A grammar can count two items, but not three

Top-Down Parsing (1)

- RE
  - Use element construction + subset construction to generate a DFA (= scanner)
  - DFA = finite-state automaton
- CFG
  - Can we also generate a parser for a CFG?
  - SA = stack automaton
    A SA is a NFA (or DFA) with an extra stack.
    The stack gives the FA the extra 'power'.
  - A parser is an algorithm based on a SA that begins with a start symbol of a CFG and derives a sentence.

Top-Down Parsing (2)

- Recall recursive-descent parsing (Ch.1)
  A procedure is associated with each nonterminal N in the grammar. The body of the procedure may contain
  - statements that match terminals;
  - statements that call procedures for each nonterminal in the right-hand side of the production of N;
  - semantic actions.

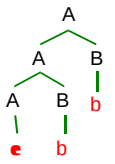
  Recursive-descent parsers implicitly use a stack, i.e. the call-stack of the procedures.
- Top-Down parsing: building the parse tree from the root (i.e. the start symbol).

Top-Down Parsing (3)

- *Example* (using an explicit stack):

  A → A B
  A → ε
  B → b

string = bb



A ⇒ AB ⇒ AAB ⇒ BB ⇒ bB ⇒ bb
all left-derivations

stack	input
A	bb$
AB	bb$
ABB	bb$
BB	bb$
bB	bb$
B	bb$
b	bb$
-	bb$

For the nonterminal on top of a production rule is executed.

Terminals on top of the stack get popped, while advancing the look-ahead pointer in the input.

Top-Down Table-Driven TD Parsing (4)

TD-TD Algorithm:

```
bool TDTD()
{
  Stack s;
  bool  accept=true;
  s.init();
  s.push(S);
  while (accept && (look_ahead!=$ || !s.empty())) {
    top = s.pop();
    if (top ∈ T) {
      if (top != look_ahead) accept=false;
      else look_ahead=read_input();
    } else if (top ∈ N) { // assume top == A
      Select some production A → X1 ... Xn
      s.push(X1,...,Xn);
    } else accept=false;
  }
  return accept;
}
```

Note that in each iteration a symbol is popped from the stack

might be nondeterministic...

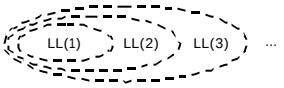
In the TD-TD algorithm, the selection of a production rule is driven by a table.

LL(1) (1)

- So we may have a choice of production rules
  A → α
  A → β
- Choosing a production rule
  - non-predictive: randomly (requires backtracking!)
  - predictive: using the look-ahead symbols in the input
- LL(k)
  If by looking ahead k symbols in the input stream, we can always choose the right production rule, the given grammar is (strong) LL(k).
  - L: left-to-right scanning through the input stream
  - L: left-derivation

LL(1) (2)

- strong LL(k) vs. (normal) LL(k)
  - strong LL(k): we only consider the look-ahead tokens in the input stream when choosing a production rule.
  - LL(k): apart from the look-ahead symbols in the input, we may also use the input tokens that have already been read to choose a production rule.
  - class of strong LL(k) grammars ⊂ class of LL(k) grammars
- *Example:*
  p_1: A → aA
  p_2: A → a



  - If k=1, we cannot tell if p_1 or p_2 should be applied.
  - Therefore, the grammar is not LL(1); it is LL(2).

LL(1) (3)

- Consider k=1
  - LL(1) = strong LL(1)
  - LL(1) grammars are sufficient to describe most programming constructs
- Define:

New definition of LL(1):
Given G and productions
A → α and A → β, then if
FIRST(α.FOLLOW(A)) ∩ FIRST(β.FOLLOW(A)) = ∅
then G is LL(1).

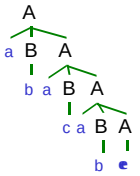
α and β might be ε

prefix(w)
= first terminal of w
FIRST(α)
= { terminals that are first in a sentence w derived from α }
= { a | α ⇒* w and a=prefix(w), for some w ∈ T* }
FOLLOW(A)
= { terminals that are in FIRST(γ) in some sentential form βAγ }
= { a | S ⇒* βAγ and a ∈ FIRST(γ), for some β,γ ∈ V* }

LL(1) (4)

- *Example:* G_1 is defined by
  p_1 : A → aBA
  p_2 : A → ε
  p_3 : B → b
  p_4 : B → c

L(G) = { ab, ac, abab, abac, acab, acac, ... }



- LL(1)-test for G_1:
  - p_1 and p_2
    FIRST(aBA.FOLLOW(A)) ∩ FIRST(ε.FOLLOW(A))
    = {a} ∩ {$} = ∅
  - p_3 and p_4
    FIRST(b.FOLLOW(B)) ∩ FIRST(c.FOLLOW(B))
    = {b} ∩ {c} = ∅

Hence, G_1 is LL(1)

LL(1) (5)

- *Example:* G_2 is defined by

  p_1 : A → aBAa
  p_2 : A → ε
  p_3 : B → b
  p_4 : B → c

  L(G) = { aba, aca, ababa, abaca, acaba, acaca, ... }

- LL(1)-test for G_2:
  - p_1 and p_2

    FIRST(aBAa.FOLLOW(A)) ∩ FIRST(ε.FOLLOW(A))
    = {a} ∩ {FOLLOW(A)} = {a} ∩ {a} ≠ ∅

    G_2 is not LL(1)

LL(1) (6)

- Notational convenience:
  - Instead of using the expression FIRST(α.FOLLOW(A)) for a production rule A → α, we define

    DIRSET(A → α)
    = FIRST(α) , if ¬EMPTY(α)
    = FIRST(α) ∪ FOLLOW(A) , otherwise

- The DIRSET set can be used to compute the parse table

  DIRSET(A → α) = { a_1, a_2, ... }
  DIRSET(A → β) = { b_1, b_2, ... }
  Now: M(A, a_i) = A → α
  M(A, b_i) = B → β

- In general: M(A,a) = A → α, if a ∈ DIRSET(A → α)

LL(1) (7)

When G is not LL(1), can it *be made LL(1)*?

Answer: sometimes

It is not decidable, though

- ... we know how to check whether G is LL(1)...
- Left factorisation

  A → αβ
  A → αγ
  cannot be LL(1)

  becomes

  A → αB
  B → β
  B → γ
  which is LL(1) if FIRST(β) ∩ FIRST(γ) = ∅

- Eliminate left recursion

  A → Aα
  A → β
  cannot be LL(1)

  becomes

  A → BC
  B → β
  C → αC
  C → ε
  which is LL(1) if FIRST(αC.FOLLOW(C)) ∩ FIRST(ε.FOLLOW(C)) = ∅

LL(1) (8)

- ... it does not always work (e.g. left factorisation)

D is not important

A → B
A → C
B → DB
B → α
C → DC
C → β

A → DB
A → DC
A → α
A → β
B → DB
B → α
C → DC
C → β

A → DE
E → B
E → C
A → α
A → β
B → DB
B → α
C → DC
C → β

It may not directly clear why this G is not LL(1)

The nonterminal E now assumes the role of the original A: this method will not terminate!

LL(1) (9)

- Concluding remarks:
  - Section 8.3.3 of the book/reader contains an extensive and formal discussion on the LL(1)-test using the function EMPTY and the sets LEADING (generalisation of FIRST), TRAILING, FOLLOW and DIRSET.
  - Algorithms are presented to automatically calculate these sets to perform the LL(1)-test; and if the grammar is LL(1), the DIRSET can directly be used to construct the parse table.
  - We will briefly discuss these sets (and algorithms) in HC10 when presenting ECFGs.