A formalisation of the Cocke-Younger-Kasami algorithm

Maksym Bortin

March 19, 2025

Abstract

The theory provides a formalisation of the Cocke-Younger-Kasami algorithm [1] (CYK for short), an approach to solving the word problem for context-free languages. CYK decides if a word is in the languages generated by a context-free grammar in Chomsky normal form. The formalized algorithm is executable.

Contents

theory *CYK*
imports *Main*
begin

The theory is structured as follows. First section deals with modelling of grammars, derivations, and the language semantics. Then the basic properties are proved. Further, CYK is abstractly specified and its underlying recursive relationship proved. The final section contains a prototypical implementation accompanied by a proof of its correctness.

1 Basic modelling

1.1 Grammars in Chomsky normal form

A grammar in Chomsky normal form is here simply modelled by a list of production rules (the type CNG below), each having a non-terminal symbol on the lhs and either two non-terminals or one terminal symbol on the rhs.

datatype ('n, 't) *RHS* = *Branch* 'n 'n
| *Leaf* 't

type-synonym ('n, 't) *CNG* = ('n × ('n, 't) *RHS*) *list*

Abbreviating the list append symbol for better readability

abbreviation *list-append* :: 'a *list* ⇒ 'a *list* ⇒ 'a *list* (**infixr** ‹·› *65*)
where *xs* · *ys* ≡ *xs* @ *ys*

1.2 Derivation by grammars

A *word form* (or sentential form) may be built of both non-terminal and terminal symbols, as opposed to a *word* that contains only terminals. By the usage of disjoint union, non-terminals are injected into a word form by *Inl* whereas terminals – by *Inr*.

type-synonym ('n, 't) *word-form* = ('n + 't) *list*
type-synonym 't *word* = 't *list*

A single step derivation relation on word forms is induced by a grammar in the standard way, replacing a non-terminal within a word form in accordance to the production rules.

definition *DSTEP* :: ('n, 't) *CNG* ⇒ (('n, 't) *word-form* × ('n, 't) *word-form*) *set*
where *DSTEP G* = {(*l* · [*Inl N*] · *r*, *x*) | *l N r rhs x*. (*N*, *rhs*) ∈ *set G* ∧
(*case rhs of*
Branch A B ⇒ *x* = *l* · [*Inl A*, *Inl B*] · *r*
| *Leaf t* ⇒ *x* = *l* · [*Inr t*] · *r*)}

abbreviation *DSTEP'* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (‹- −-→ -› [*60*, *61*, *60*] *61*)
where *w* −*G*→ *w'* ≡ (*w*, *w'*) ∈ *DSTEP G*

abbreviation *DSTEP-reflc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (‹- −-→$^=$ -› [*60*, *61*, *60*] *61*)
where w −G→$^=$ w' ≡ (w, w') ∈ (*DSTEP G*)$^=$

abbreviation *DSTEP-transc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (‹- −-→$^+$ -› [*60*, *61*, *60*] *61*)
where w −G→$^+$ w' ≡ (w, w') ∈ (*DSTEP G*)$^+$

abbreviation *DSTEP-rtransc* :: ('n, 't) *word-form* ⇒ ('n, 't) *CNG* ⇒ ('n, 't) *word-form* ⇒ *bool* (‹- −-→* -› [*60*, *61*, *60*] *61*)
where w −G→* w' ≡ (w, w') ∈ (*DSTEP G*)*

1.3 The generated language semantics

The language generated by a grammar from a non-terminal symbol comprises all words that can be derived from the non-terminal in one or more steps. Notice that by the presented grammar modelling, languages containing the empty word cannot be generated. Hence in rare situations when such languages are required, the empty word case should be treated separately.

definition *Lang* :: ('n, 't) *CNG* ⇒ 'n ⇒ 't *word set*
where *Lang G S* = {*w*. [*Inl S*] −G→$^+$ *map Inr w* }

So, for instance, a grammar generating the language $a^n b^n$ from the non-terminal ''S'' might look as follows.

definition *G-anbn* =
[(''S'', *Branch* ''A'' ''T''),
(''S'', *Branch* ''A'' ''B''),
(''T'', *Branch* ''S'' ''B''),
(''A'', *Leaf* ''a''),
(''B'', *Leaf* ''b'')]

Now the term *Lang G-anbn* ''S'' denotes the set of words of the form $a^n b^n$ with $n > 0$. This is intuitively clear, but not straight forward to show, and a lengthy proof for that is out of scope.

2 Basic properties

lemma *prod-into-DSTEP1* :
(*S*, *Branch A B*) ∈ *set G* ⟹
L · [*Inl S*] · *R* −G→ *L* · [*Inl A*, *Inl B*] · *R*
by(*simp add*: *DSTEP-def*, *rule-tac x*=*L* **in** *exI*, *force*)

lemma *prod-into-DSTEP2* :
(*S*, *Leaf a*) ∈ *set G* ⟹
L · [*Inl S*] · *R* −G→ *L* · [*Inr a*] · *R*

by(*simp add*: *DSTEP-def*, *rule-tac x*=*L* **in** *exI*, *force*)

lemma *DSTEP-D* :
s −*G*→ *t* ⟹
∃ *L N R rhs*. *s* = *L* · [*Inl N*] · *R* ∧ (*N*, *rhs*) ∈ *set G* ∧
(∀ *A B*. *rhs* = *Branch A B* ⟶ *t* = *L* · [*Inl A*, *Inl B*] · *R*) ∧
(∀ *x*. *rhs* = *Leaf x* ⟶ *t* = *L* · [*Inr x*] · *R*)
by(*unfold DSTEP-def*, *clarsimp*, *simp split*: *RHS.split-asm*, *blast*+)

lemma *DSTEP-append* :
assumes *a*: *s* −*G*→ *t*
shows *L* · *s* · *R* −*G*→ *L* · *t* · *R*
proof −
from *a* **have** ∃ *l N r rhs*. *s* = *l* · [*Inl N*] · *r* ∧ (*N*, *rhs*) ∈ *set G* ∧
(∀ *A B*. *rhs* = *Branch A B* ⟶ *t* = *l* · [*Inl A*, *Inl B*] · *r*) ∧
(∀ *x*. *rhs* = *Leaf x* ⟶ *t* = *l* · [*Inr x*] · *r*) (**is** ∃ *l N r rhs*. *?P l N r rhs*)
by(*rule DSTEP-D*)
then obtain *l N r rhs* **where** *?P l N r rhs* **by** *blast*
thus *?thesis*
by(*simp add*: *DSTEP-def*, *rule-tac x*=*L* · *l* **in** *exI*,
rule-tac x=*N* **in** *exI*, *rule-tac x*=*r* · *R* **in** *exI*,
simp, *rule-tac x*=*rhs* **in** *exI*, *simp split*: *RHS.split*)
qed

lemma *DSTEP-star-mono* :
s −*G*→* *t* ⟹ *length s* ≤ *length t*
proof(*erule rtrancl-induct*, *simp*)
fix *t u*
assume *s* −*G*→* *t*
assume *a*: *t* −*G*→ *u*
assume *b*: *length s* ≤ *length t*
show *length s* ≤ *length u*
proof −
from *a* **have** ∃ *L N R rhs*. *t* = *L* · [*Inl N*] · *R* ∧ (*N*, *rhs*) ∈ *set G* ∧
(∀ *A B*. *rhs* = *Branch A B* ⟶ *u* = *L* · [*Inl A*, *Inl B*] · *R*) ∧
(∀ *x*. *rhs* = *Leaf x* ⟶ *u* = *L* · [*Inr x*] · *R*) (**is** ∃ *L N R rhs*. *?P L N R rhs*)
by(*rule DSTEP-D*)
then obtain *L N R rhs* **where** *?P L N R rhs* **by** *blast*
with *b* **show** *?thesis*
by(*case-tac rhs*, *clarsimp*+)
qed

qed

lemma *DSTEP-comp* :
assumes *a*: $l \cdot r -G\rightarrow t$
shows $\exists\, l'\ r'.\ l -G\rightarrow^{=} l' \wedge r -G\rightarrow^{=} r' \wedge t = l' \cdot r'$
proof −
from *a* **have** $\exists\, L\ N\ R\ rhs.\ l \cdot r = L \cdot [Inl\ N] \cdot R \wedge (N, rhs) \in set\ G \wedge$
$(\forall\, A\ B.\ rhs = Branch\ A\ B \longrightarrow t = L \cdot [Inl\ A, Inl\ B] \cdot R) \wedge$
$(\forall\, x.\ rhs = Leaf\ x \longrightarrow t = L \cdot [Inr\ x] \cdot R)$ (**is** $\exists\, L\ N\ R\ rhs.\ ?T$ $L\ N\ R\ rhs$)
by(*rule DSTEP-D*)
then obtain *L N R rhs* **where** *b*: *?T L N R rhs* **by** *blast*
hence $l \cdot r = L \cdot Inl\ N \# R$ **by** *simp*
hence $\exists\, u.\ (l = L \cdot u \wedge u \cdot r = Inl\ N \# R) \vee (l \cdot u = L \wedge r = u \cdot Inl\ N \# R)$ **by**(*rule append-eq-append-conv2*[*THEN iffD1*])
then obtain *xs* **where** *c*: $l = L \cdot xs \wedge xs \cdot r = Inl\ N \# R \vee l \cdot xs = L \wedge r = xs \cdot Inl\ N \# R$ (**is** *?C1* ∨ *?C2*) **by** *blast*
show *?thesis*
proof(*cases rhs*)
case (*Leaf x*)
with *b* **have** *d*: $t = L \cdot [Inr\ x] \cdot R \wedge (N, Leaf\ x) \in set\ G$ **by** *simp*
from *c* **show** *?thesis*
proof
assume *e*: *?C1*
show *?thesis*
proof(*cases xs*)
case *Nil* **with** *d* **and** *e* **show** *?thesis*
by(*clarsimp*, *rule-tac x=L* **in** *exI*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
next
case (*Cons z zs*) **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x=L · Inr x # zs* **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
qed
next
assume *e*: *?C2*
show *?thesis*
proof(*cases xs*)
case *Nil* **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x=L* **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
next
case (*Cons z zs*) **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x=l* **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*,

rule-tac x=z#zs **in** *exI*, *rule-tac x=N* **in** *exI*, *rule-tac x=R* **in** *exI*, *simp*, *rule-tac x=Leaf x* **in** *exI*, *simp*)
qed

qed
next
case (*Branch A B*)
with *b* **have** *d*: $t = L \cdot [Inl\ A,\ Inl\ B] \cdot R \wedge (N,\ Branch\ A\ B) \in set\ G$ **by** *simp*
from *c* **show** *?thesis*
proof
assume *e*: *?C1*
show *?thesis*
proof(*cases xs*)
case *Nil* **with** *d* **and** *e* **show** *?thesis*
by(*clarsimp*, *rule-tac x=L* **in** *exI*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
next
case (*Cons z zs*) **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x*=$L \cdot [Inl\ A,\ Inl\ B] \cdot zs$ **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
qed
next
assume *e*: *?C2*
show *?thesis*
proof(*cases xs*)
case *Nil* **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x=L* **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*, *blast*)
next
case (*Cons z zs*) **with** *d* **and** *e* **show** *?thesis*
by(*rule-tac x=l* **in** *exI*, *clarsimp*, *simp add*: *DSTEP-def*, *simp split*: *RHS.split*,

rule-tac x=z#zs **in** *exI*, *rule-tac x=N* **in** *exI*, *rule-tac x=R* **in** *exI*, *simp*, *rule-tac x=Branch A B* **in** *exI*, *simp*)
qed
qed
qed
qed

theorem *DSTEP-star-comp1* :
assumes *A*: $l \cdot r\ {-G{\rightarrow}^*}\ t$
shows $\exists\, l'\ r'.\ l\ {-G{\rightarrow}^*}\ l' \wedge r\ {-G{\rightarrow}^*}\ r' \wedge t = l' \cdot r'$
proof −
have $\bigwedge s.\ s\ {-G{\rightarrow}^*}\ t \Longrightarrow$
$\forall\, l\ r.\ s = l \cdot r \longrightarrow (\exists\, l'\ r'.\ l\ {-G{\rightarrow}^*}\ l' \wedge r\ {-G{\rightarrow}^*}\ r' \wedge t = l' \cdot r')$ (**is** $\bigwedge s.$ *?P s t* $\Longrightarrow$ *?Q s t*)
proof(*erule rtrancl-induct*, *force*)
fix *s t u*
assume *?P s t*
assume *a*: $t\ {-G\rightarrow}\ u$

assume b: *?Q s t*
show *?Q s u*
proof(*clarify*)
fix $l\ r$
assume $s = l \cdot r$
with b **have** $\exists\, l'\ r'.\ l\ -G\rightarrow^*\ l' \wedge r\ -G\rightarrow^*\ r' \wedge t = l' \cdot r'$ **by** *simp*
then obtain $l'\ r'$ **where** c: $l\ -G\rightarrow^*\ l' \wedge r\ -G\rightarrow^*\ r' \wedge t = l' \cdot r'$ **by** *blast*
with a **have** $l' \cdot r'\ -G\rightarrow\ u$ **by** *simp*
hence $\exists\, l''\ r''.\ l'\ -G\rightarrow^=\ l'' \wedge r'\ -G\rightarrow^=\ r'' \wedge u = l'' \cdot r''$ **by**(*rule DSTEP-comp*)
then obtain $l''\ r''$ **where** $l'\ -G\rightarrow^=\ l'' \wedge r'\ -G\rightarrow^=\ r'' \wedge u = l'' \cdot r''$ **by** *blast*
hence $l'\ -G\rightarrow^*\ l'' \wedge r'\ -G\rightarrow^*\ r'' \wedge u = l'' \cdot r''$ **by** *blast*
with c **show** $\exists\, l'\ r'.\ l\ -G\rightarrow^*\ l' \wedge r\ -G\rightarrow^*\ r' \wedge u = l' \cdot r'$
by(*rule-tac x*=l'' **in** *exI*, *rule-tac x*=r'' **in** *exI*, *force*)
qed
qed
with A **show** *?thesis* **by** *force*
qed

theorem *DSTEP-star-comp2* :
assumes A: $l\ -G\rightarrow^*\ l'$
and B: $r\ -G\rightarrow^*\ r'$
shows $l \cdot r\ -G\rightarrow^*\ l' \cdot r'$
proof −
have $l\ -G\rightarrow^*\ l' \Longrightarrow$
$\forall\, r\ r'.\ r\ -G\rightarrow^*\ r' \longrightarrow l \cdot r\ -G\rightarrow^*\ l' \cdot r'$ (**is** *?P l l'* $\Longrightarrow$ *?Q l l'*)
proof(*erule rtrancl-induct*)
show *?Q l l*
proof(*clarify*, *erule rtrancl-induct*, *simp*)
fix $r\ s\ t$
assume a: $s\ -G\rightarrow\ t$
assume b: $l \cdot r\ -G\rightarrow^*\ l \cdot s$
show $l \cdot r\ -G\rightarrow^*\ l \cdot t$
proof −
from a **have** $l \cdot s\ -G\rightarrow\ l \cdot t$ **by**(*drule-tac L*=l **and** *R*=[] **in** *DSTEP-append*, *simp*)
with b **show** *?thesis* **by** *simp*
qed
qed
next
fix $s\ t$
assume a: $s\ -G\rightarrow\ t$
assume b: *?Q l s*
show *?Q l t*
proof(*clarsimp*)
fix $r\ r'$
assume $r\ -G\rightarrow^*\ r'$
with b **have** c: $l \cdot r\ -G\rightarrow^*\ s \cdot r'$ **by** *simp*

from a **have** $s \cdot r' -G\rightarrow t \cdot r'$ **by**(*drule-tac* L=[] **and** R=r' **in** *DSTEP-append*, *simp*)
with c **show** $l \cdot r -G\rightarrow^* t \cdot r'$ **by** *simp*
qed
qed
with A **and** B **show** *?thesis* **by** *simp*
qed

lemma *DSTEP-trancl-term* :
assumes A: [*Inl S*] $-G\rightarrow^+$ t
and B: *Inr x* $\in$ *set t*
shows $\exists N$. (N, *Leaf x*) $\in$ *set G*
proof −
have [*Inl S*] $-G\rightarrow^+$ $t \Longrightarrow$
$\forall x$. *Inr x* $\in$ *set t* $\longrightarrow$ ($\exists N$. (N, *Leaf x*) $\in$ *set G*) (**is** *?P t* $\Longrightarrow$ *?Q t*)
proof(*erule trancl-induct*)
fix t
assume a: [*Inl S*] $-G\rightarrow$ t
show *?Q t*
proof −
from a **have** $\exists$ *rhs*. (S, *rhs*) $\in$ *set G* $\wedge$
($\forall A$ B. *rhs* = *Branch A B* $\longrightarrow$ t = [*Inl A*, *Inl B*]) $\wedge$
($\forall x$. *rhs* = *Leaf x* $\longrightarrow$ t = [*Inr x*]) (**is** $\exists$ *rhs*. *?P rhs*)
by(*simp add*: *DSTEP-def*, *clarsimp*, *simp split*: *RHS.split-asm*, *case-tac l*, *force*, *simp*,
clarsimp, *simp split*: *RHS.split-asm*, *case-tac l*, *force*, *simp*)
then obtain *rhs* **where** *?P rhs* **by** *blast*
thus *?thesis*
by(*case-tac rhs*, *clarsimp*, *force*)
qed
next
fix s t
assume a: $s -G\rightarrow t$
assume b: *?Q s*
show *?Q t*
proof −
from a **have** $\exists L$ N R *rhs*. $s = L \cdot$ [*Inl N*] $\cdot R \wedge$ (N, *rhs*) $\in$ *set G* $\wedge$
($\forall A$ B. *rhs* = *Branch A B* $\longrightarrow$ $t = L \cdot$ [*Inl A*, *Inl B*] $\cdot R$) $\wedge$
($\forall x$. *rhs* = *Leaf x* $\longrightarrow$ $t = L \cdot$ [*Inr x*] $\cdot R$) (**is** $\exists L$ N R *rhs*. *?P L N R rhs*)
by(*rule DSTEP-D*)
then obtain *L N R rhs* **where** *?P L N R rhs* **by** *blast*
with b **show** *?thesis*
by(*case-tac rhs*, *clarsimp*, *force*)
qed
qed
with A **and** B **show** *?thesis* **by** *simp*

qed

2.1 Properties of generated languages

lemma *Lang-no-Nil* :
$w \in Lang\ G\ S \Longrightarrow w \neq []$
by(*simp add*: *Lang-def*, *drule trancl-into-rtrancl*, *drule DSTEP-star-mono*, *force*)

lemma *Lang-rtrancl-eq* :
$(w \in Lang\ G\ S) = [Inl\ S]\ -G\rightarrow^*\ map\ Inr\ w$ (**is** $?L = (?p \in ?R^*)$)
proof(*simp add*: *Lang-def*, *rule iffI*, *erule trancl-into-rtrancl*)
 assume $?p \in ?R^*$
 hence $?p \in (?R^+)^=$ **by**(*subst rtrancl-trancl-reflcl*[*THEN sym*], *assumption*)
 hence $[Inl\ S] = map\ Inr\ w \vee ?p \in ?R^+$ **by** *force*
 thus $?p \in ?R^+$ **by**(*case-tac w*, *simp-all*)
qed

lemma *Lang-term* :
$w \in Lang\ G\ S \Longrightarrow$
$\forall x \in set\ w.\ \exists N.\ (N, Leaf\ x) \in set\ G$
by(*clarsimp simp add*: *Lang-def*, *drule DSTEP-trancl-term*,
  simp, *erule imageI*, *assumption*)

lemma *Lang-eq1* :
$([x] \in Lang\ G\ S) = ((S, Leaf\ x) \in set\ G)$
proof(*simp add*: *Lang-def*, *rule iffI*, *subst* (*asm*) *trancl-unfold-left*, *clarsimp*)
 fix *t*
 assume *a*: $[Inl\ S]\ -G\rightarrow\ t$
 assume *b*: $t\ -G\rightarrow^*\ [Inr\ x]$
 note *DSTEP-star-mono*[*OF b*, *simplified*]
 hence *c*: $length\ t \leq 1$ **by** *simp*
 have $\exists z.\ t = [z]$
 proof(*cases t*)
  assume $t = []$
  with *b* **have** *d*: $[]\ -G\rightarrow^*\ [Inr\ x]$ **by** *simp*
  have $\bigwedge s.\ ([], s) \in (DSTEP\ G)^* \Longrightarrow s = []$
  by(*erule rtrancl-induct*, *simp-all*, *drule DSTEP-D*, *clarsimp*)
  note *this*[*OF d*]
  thus *?thesis* **by** *simp*
 next
  fix *z zs*
  assume $t = z\#zs$

with *c* **show** *?thesis* **by** *force*
qed
with *a* **have** $\exists$ *z*. (*S*, *Leaf z*) $\in$ *set G* $\wedge$ *t* = [*Inr z*]
by(*clarsimp simp add*: *DSTEP-def*, *simp split*: *RHS.split-asm*, *case-tac l*, *simp-all*)

with *b* **show** (*S*, *Leaf x*) $\in$ *set G*
proof(*clarsimp*)
fix *z*
assume *c*: (*S*, *Leaf z*) $\in$ *set G*
assume [*Inr z*] $-G\rightarrow^*$ [*Inr x*]
hence ([*Inr z*], [*Inr x*]) $\in$ ((*DSTEP G*)$^+$)$^=$ **by** *simp*
hence [*Inr z*] = [*Inr x*] $\vee$ [*Inr z*] $-G\rightarrow^+$ [*Inr x*] **by** *force*
hence *x* = *z*
proof
assume [*Inr z*] = [*Inr x*] **thus** *?thesis* **by** *simp*
next
assume [*Inr z*] $-G\rightarrow^+$ [*Inr x*]
hence $\exists$ *u*. [*Inr z*] $-G\rightarrow$ *u* $\wedge$ *u* $-G\rightarrow^*$ [*Inr x*] **by**(*subst* (*asm*) *trancl-unfold-left*, *force*)
then obtain *u* **where** [*Inr z*] $-G\rightarrow$ *u* **by** *blast*
thus *?thesis* **by**(*clarsimp simp add*: *DSTEP-def*, *case-tac l*, *simp-all*)
qed
with *c* **show** *?thesis* **by** *simp*
qed
next
assume *a*: (*S*, *Leaf x*) $\in$ *set G*
show [*Inl S*] $-G\rightarrow^+$ [*Inr x*]
by(*rule r-into-trancl*, *simp add*: *DSTEP-def*, *rule-tac x*=[] **in** *exI*,
rule-tac x=*S* **in** *exI*, *rule-tac x*=[] **in** *exI*, *simp*, *rule-tac x*=*Leaf x* **in** *exI*,
simp add: *a*)
qed

theorem *Lang-eq2* :
(*w* $\in$ *Lang G S* $\wedge$ *1* < *length w*) =
($\exists$ *A B*. (*S*, *Branch A B*) $\in$ *set G* $\wedge$ ($\exists$ *l r*. *w* = *l* $\cdot$ *r* $\wedge$ *l* $\in$ *Lang G A* $\wedge$ *r* $\in$ *Lang G B*))
(**is** *?L* = *?R*)
proof(*rule iffI*, *clarify*, *subst* (*asm*) *Lang-def*, *simp*, *subst* (*asm*) *trancl-unfold-left*, *clarsimp*)
have *map-Inr-split* : $\bigwedge$*xs*. $\forall$ *zs w*. *map Inr w* = *xs* $\cdot$ *zs* $\longrightarrow$
($\exists$ *u v*. *w* = *u* $\cdot$ *v* $\wedge$ *xs* = *map Inr u* $\wedge$ *zs* = *map Inr v*)
by(*induct-tac xs*, *simp*, *force*)
fix *t*
assume *a*: *Suc 0* < *length w*
assume *b*: [*Inl S*] $-G\rightarrow$ *t*
assume *c*: *t* $-G\rightarrow^*$ *map Inr w*
from *b* **have** $\exists$ *A B*. (*S*, *Branch A B*) $\in$ *set G* $\wedge$ *t* = [*Inl A*, *Inl B*]

proof(*simp add*: *DSTEP-def*, *clarify*, *case-tac l*, *simp-all*, *simp split*: *RHS.split-asm*, *clarify*)
fix *x*
assume $t = [\mathit{Inr}\ x]$
with *c* **have** *d*: $[\mathit{Inr}\ x]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ w$**by** *simp*
have $\bigwedge x\ s.\ [\mathit{Inr}\ x]\ {-G\rightarrow^*}\ s \Longrightarrow s = [\mathit{Inr}\ x]$
by(*erule rtrancl-induct*, *simp-all*, *drule DSTEP-D*, *clarsimp*, *case-tac L*, *simp-all*)
note *this*[*OF d*]
hence $w = [x]$ **by**(*case-tac w*, *simp-all*)
with *a* **show** *False* **by** *simp*
qed
then obtain *A B* **where** *d*: $(S,\ \mathit{Branch}\ A\ B) \in \mathit{set}\ G \wedge t = [\mathit{Inl}\ A,\ \mathit{Inl}\ B]$ **by** *blast*
with *c* **have** *e*: $[\mathit{Inl}\ A] \cdot [\mathit{Inl}\ B]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ w$ **by** *simp*
note *DSTEP-star-comp1*[*OF e*]
then obtain $l'\ r'$ **where** *e*: $[\mathit{Inl}\ A]\ {-G\rightarrow^*}\ l' \wedge [\mathit{Inl}\ B]\ {-G\rightarrow^*}\ r' \wedge$
$\mathit{map}\ \mathit{Inr}\ w = l' \cdot r'$ **by** *blast*
note *map-Inr-split*[*rule-format*, *OF e*[*THEN conjunct2*, *THEN conjunct2*]]
then obtain *u v* **where** *f*: $w = u \cdot v \wedge l' = \mathit{map}\ \mathit{Inr}\ u \wedge r' = \mathit{map}\ \mathit{Inr}\ v$ **by** *blast*
with *e* **have** *g*: $[\mathit{Inl}\ A]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ u \wedge [\mathit{Inl}\ B]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ v$ **by** *simp*
show *?R*
by(*rule-tac x=A* **in** *exI*, *rule-tac x=B* **in** *exI*, *simp add*: *d*,
rule-tac x=u **in** *exI*, *rule-tac x=v* **in** *exI*, *simp add*: *f*,
(*subst Lang-rtrancl-eq*)+, *rule g*)
next
assume *?R*
then obtain *A B l r* **where** *a*: $(S,\ \mathit{Branch}\ A\ B) \in \mathit{set}\ G \wedge w = l \cdot r \wedge l \in \mathit{Lang}\ G\ A \wedge r \in \mathit{Lang}\ G\ B$ **by** *blast*
have $[\mathit{Inl}\ A] \cdot [\mathit{Inl}\ B]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ l \cdot \mathit{map}\ \mathit{Inr}\ r$
by(*rule DSTEP-star-comp2*, *subst Lang-rtrancl-eq*[*THEN sym*], *simp add*: *a*,
subst Lang-rtrancl-eq[*THEN sym*], *simp add*: *a*)
hence *b*: $[\mathit{Inl}\ A] \cdot [\mathit{Inl}\ B]\ {-G\rightarrow^*}\ \mathit{map}\ \mathit{Inr}\ w$ **by**(*simp add*: *a*)
have *c*: $w \in \mathit{Lang}\ G\ S$
by(*simp add*: *Lang-def*, *subst trancl-unfold-left*, *rule-tac b*=[*Inl A*] · [*Inl B*] **in** *relcompI*,
simp add: *DSTEP-def*, *rule-tac x*=[] **in** *exI*, *rule-tac x=S* **in** *exI*, *rule-tac x*=[] **in** *exI*,
simp, *rule-tac x=Branch A B* **in** *exI*, *simp add*: *a*[*THEN conjunct1*], *rule b*)
thus *?L*
proof
show $1 < \mathit{length}\ w$
proof(*simp add*: *a*, *rule ccontr*, *drule leI*)
assume $\mathit{length}\ l + \mathit{length}\ r \leq \mathit{Suc}\ 0$
hence $l = [] \vee r = []$ **by**(*case-tac l*, *simp-all*)
thus *False*
proof
assume $l = []$
with *a* **have** $[] \in \mathit{Lang}\ G\ A$ **by** *simp*

note *Lang-no-Nil*[*OF this*]
thus *?thesis* **by** *simp*
next
assume $r = []$
with *a* **have** $[] \in$ *Lang G B* **by** *simp*
note *Lang-no-Nil*[*OF this*]
thus *?thesis* **by** *simp*
qed
qed
qed
qed

3 Abstract specification of CYK

A subword of a word w, starting at the position i (first element is at the position 0) and having the length j, is defined as follows.

definition *subword w i j = take j* (*drop i w*)

Thus, to any subword of the given word w CYK assigns all non-terminals from which this subword is derivable by the grammar G.

definition *CYK G w i j* = {*S. subword w i j* ∈ *Lang G S*}

3.1 Properties of *subword*

lemma *subword-length* :
$i + j \leq$ *length w* ⟹ *length*(*subword w i j*) = j
by(*simp add*: *subword-def*)

lemma *subword-nth1* :
$i + j \leq$ *length w* ⟹ $k < j$ ⟹
(*subword w i j*)!k = w!($i + k$)
by(*simp add*: *subword-def*)

lemma *subword-nth2* :
assumes *A*: $i + 1 \leq$ *length w*
shows *subword w i 1* = [*w*!*i*]
proof −
note *subword-length*[*OF A*]
hence ∃ *x. subword w i 1* = [*x*] **by**(*case-tac subword w i 1*, *simp-all*)
then obtain *x* **where** *a*:*subword w i 1* = [*x*] **by** *blast*
note *subword-nth1*[*OF A*, **where** *k*=(*0* :: *nat*), *simplified*]
with *a* **have** *x* = *w*!*i* **by** *simp*
with *a* **show** *?thesis* **by** *simp*
qed

lemma *subword-self* :
subword w 0 (*length w*) = *w*
by(*simp add*: *subword-def*)

lemma *take-split*[*rule-format*] :
∀ *n m*. *n* ≤ *length xs* ⟶ *n* ≤ *m* ⟶
take n xs · *take* (*m* − *n*) (*drop n xs*) = *take m xs*
by(*induct-tac xs*, *clarsimp*+, *case-tac n*, *simp-all*, *case-tac m*, *simp-all*)

lemma *subword-split* :
i + *j* ≤ *length w* ⟹ *0* < *k* ⟹ *k* < *j* ⟹
subword w i j = *subword w i k* · *subword w* (*i* + *k*) (*j* − *k*)
by(*simp add*: *subword-def*, *subst take-split*[**where** *n*=*k*, *THEN sym*], *simp-all*,
rule-tac f=λ*x*. *take* (*j* − *k*) (*drop x w*) **in** *arg-cong*, *simp*)

lemma *subword-split2* :
assumes *A*: *subword w i j* = *l* · *r*
and *B*: *i* + *j* ≤ *length w*
and *C*: *0* < *length l*
and *D*: *0* < *length r*
shows *l* = *subword w i* (*length l*) ∧ *r* = *subword w* (*i* + *length l*) (*j* − *length l*)
proof −
have *a*: *length*(*subword w i j*) = *j* **by**(*rule subword-length*, *rule B*)
note *arg-cong*[**where** *f*=*length*, *OF A*]
with *a* **and** *D* **have** *b*: *length l* < *j* **by** *force*
with *B* **have** *c*: *i* + *length l* ≤ *length w* **by** *force*
have *subword w i j* = *subword w i* (*length l*) · *subword w* (*i* + *length l*) (*j* − *length l*)
by(*rule subword-split*, *rule B*, *rule C*, *rule b*)
with *A* **have** *d*: *l* · *r* = *subword w i* (*length l*) · *subword w* (*i* + *length l*) (*j* − *length l*) **by** *simp*
show *?thesis*
by(*rule append-eq-append-conv*[*THEN iffD1*], *subst subword-length*, *rule c*, *simp*, *rule d*)
qed

3.2 Properties of *CYK*

lemma *CYK-Lang* :
(*S* ∈ *CYK G w 0* (*length w*)) = (*w* ∈ *Lang G S*)
by(*simp add*: *CYK-def subword-self*)

```
lemma CYK-eq1 :
i + 1 ≤ length w ⟹
 CYK G w i 1 = {S. (S, Leaf (w!i)) ∈ set G}
by(simp add: CYK-def, subst subword-nth2[simplified], assumption,
   subst Lang-eq1, rule refl)


theorem CYK-eq2 :
assumes A: i + j ≤ length w
    and B: 1 < j
shows CYK G w i j = {X | X A B k. (X, Branch A B) ∈ set G ∧ A ∈ CYK G
w i k ∧ B ∈ CYK G w (i + k) (j − k) ∧ 1 ≤ k ∧ k < j}
proof(rule set-eqI, rule iffI, simp-all add: CYK-def)
 fix X
 assume a: subword w i j ∈ Lang G X
 show ∃A B. (X, Branch A B) ∈ set G ∧ (∃k. subword w i k ∈ Lang G A ∧
subword w (i + k) (j − k) ∈ Lang G B ∧ Suc 0 ≤ k ∧ k < j)
 proof −
  have b: 1 < length(subword w i j) by(subst subword-length, rule A, rule B)
  note Lang-eq2[THEN iffD1, OF conjI, OF a b]
  then obtain A B l r where c: (X, Branch A B) ∈ set G ∧ subword w i j = l ·
r ∧ l ∈ Lang G A ∧ r ∈ Lang G B by blast
  note Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct1]]
  hence d: 0 < length l by(case-tac l, simp-all)
  note Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct2]]
  hence e: 0 < length r by(case-tac r, simp-all)
  note subword-split2[OF c[THEN conjunct2, THEN conjunct1], OF A, OF d, OF
e]
  with c show ?thesis
  proof(rule-tac x=A in exI, rule-tac x=B in exI, simp,
        rule-tac x=length l in exI, simp)
   show Suc 0 ≤ length l ∧ length l < j (is ?A ∧ ?B)
   proof
    from d show ?A by(case-tac l, simp-all)
   next
     note arg-cong[where f=length, OF c[THEN conjunct2, THEN conjunct1],
THEN sym]
    also have length(subword w i j) = j by(rule subword-length, rule A)
    finally have length l + length r = j by simp
    with e show ?B by force
   qed
  qed
 qed
next
 fix X
 assume ∃A B. (X, Branch A B) ∈ set G ∧ (∃k. subword w i k ∈ Lang G A ∧
subword w (i + k) (j − k) ∈ Lang G B ∧ Suc 0 ≤ k ∧ k < j)
 then obtain A B k where a: (X, Branch A B) ∈ set G ∧ subword w i k ∈ Lang
G A ∧ subword w (i + k) (j − k) ∈ Lang G B ∧ Suc 0 ≤ k ∧ k < j by blast
```

show *subword w i j* ∈ *Lang G X*
proof(*rule Lang-eq2*[*THEN iffD2*, *THEN conjunct1*], *rule-tac x=A* **in** *exI*, *rule-tac x=B* **in** *exI*, *simp add*: *a*,
rule-tac x=subword w i k **in** *exI*, *rule-tac x=subword w* (*i* + *k*) (*j* − *k*) **in** *exI*, *simp add*: *a*,
rule subword-split, *rule A*)
from *a* **show** *0* < *k* **by** *force*
next
from *a* **show** *k* < *j* **by** *simp*
qed
qed

4 Implementation

One of the particularly interesting features of CYK implementation is that it follows the principles of dynamic programming, constructing a table of solutions for sub-problems in the bottom-up style reusing already stored results.

4.1 Main cycle

This is an auxiliary implementation of the membership test on lists.

fun *mem* :: *'a* ⇒ *'a list* ⇒ *bool*
where
mem a [] = *False* |
mem a (*x*#*xs*) = (*x* = *a* ∨ *mem a xs*)

lemma *mem*[*simp*] :
mem x xs = (*x* ∈ *set xs*)
by(*induct-tac xs*, *simp*, *force*)

The purpose of the following is to collect non-terminals that appear on the lhs of a production such that the first non-terminal on its rhs appears in the first of two given lists and the second non-terminal – in the second list.

fun *match-prods* :: (*'n*, *'t*) *CNG* ⇒ *'n list* ⇒ *'n list* ⇒ *'n list*
where *match-prods* [] *ls rs* = [] |
match-prods ((*X*, *Branch A B*)#*ps*) *ls rs* =
(*if mem A ls* ∧ *mem B rs then X* # *match-prods ps ls rs*
else match-prods ps ls rs) |
match-prods ((*X*, *Leaf a*)#*ps*) *ls rs* = *match-prods ps ls rs*

lemma *match-prods* :
(*X* ∈ *set*(*match-prods G ls rs*)) =
(∃ *A* ∈ *set ls*. ∃ *B* ∈ *set rs*. (*X*, *Branch A B*) ∈ *set G*)
by(*induct-tac G*, *clarsimp*+, *rename-tac l r ps*, *case-tac r*, *force*+)

The following function is the inner cycle of the algorithm. The parameters i and j identify a subword starting at i with the length j, whereas k is used to iterate through its splits (which are of course subwords as well) all having the length greater 0 but less than j. The parameter T represents a table containing CYK solutions for those splits.

```
function inner :: ('n, 't) CNG ⇒ (nat × nat ⇒ 'n list) ⇒ nat ⇒ nat ⇒ nat ⇒
'n list
where inner G T i k j =
(if k < j then match-prods G (T(i, k)) (T(i + k, j − k)) @ inner G T i (k + 1) j
 else [])
by pat-completeness auto
termination
by(relation measure(λ(a, b, c, d, e). e − d), rule wf-measure, simp)


declare inner.simps[simp del]

lemma inner :
(X ∈ set(inner G T i k j)) =
 (∃ l. k ≤ l ∧ l < j ∧ X ∈ set(match-prods G (T(i, l)) (T(i + l, j − l))))
(is ?L G T i k j = ?R G T i k j)
proof(induct-tac G T i k j rule: inner.induct)
 fix G T i k j
 assume a: k < j ⟹ ?L G T i (k + 1) j = ?R G T i (k + 1) j
 show ?L G T i k j = ?R G T i k j
 proof(case-tac k < j)
  assume b: k < j
  with a have c: ?L G T i (k + 1) j = ?R G T i (k + 1) j by simp
  show ?thesis
  proof(subst inner.simps, simp add: b, rule iffI, erule disjE, rule-tac x=k in exI,
simp add: b)
   assume X ∈ set(inner G T i (Suc k) j)
   with c have ?R G T i (k + 1) j by simp
   thus ?R G T i k j by(clarsimp, rule-tac x=l in exI, simp)
  next
   assume ?R G T i k j
   then obtain l where d: k ≤ l ∧ l < j ∧ X ∈ set(match-prods G (T(i, l)) (T(i
+ l, j − l))) by blast
   show X ∈ set(match-prods G (T(i, k)) (T(i + k, j − k))) ∨ ?L G T i (Suc k)
j
    proof(case-tac Suc k ≤ l, rule disjI2, subst c[simplified], rule-tac x=l in exI,
simp add: d,
         rule disjI1)
    assume ¬ Suc k ≤ l
    with d have l = k by force
    with d show X ∈ set(match-prods G (T(i, k)) (T(i + k, j − k))) by simp
   qed
  qed
```

next
assume $\neg\ k < j$
thus *?thesis* **by**(*subst inner.simps*, *simp*)
qed
qed

Now the main part of the algorithm just iterates through all subwords up to the given length *len*, calls *inner* on these, and stores the results in the table T. The length j is supposed to be greater than 1 – the subwords of length 1 will be handled in the initialisation phase below.

function *main* :: $('n, 't)$ *CNG* $\Rightarrow$ $(nat \times nat \Rightarrow 'n\ list) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \times nat \Rightarrow 'n\ list)$
where *main G T len i j* = (*let* $T' = T((i, j) := inner\ G\ T\ i\ 1\ j)$ *in*
if $i + j < len$ *then main G* T' *len* $(i + 1)$ j
else if $j < len$ *then main G* T' *len 0* $(j + 1)$
else T')
by *pat-completeness auto*
termination
by(*relation inv-image* (*less-than* <*lex*> *less-than*) $(\lambda(a, b, c, d, e).\ (c - e, c - (d + e)))$, *rule wf-inv-image*, *rule wf-lex-prod*, (*rule wf-less-than*)+, *simp-all*)

declare *main.simps*[*simp del*]

lemma *main* :
assumes $1 < j$
and $i + j \le length\ w$
and $\bigwedge i'\ j'.\ j' < j \Longrightarrow 1 \le j' \Longrightarrow i' + j' \le length\ w \Longrightarrow set(T(i', j')) = CYK\ G\ w\ i'\ j'$
and $\bigwedge i'.\ i' < i \Longrightarrow i' + j \le length\ w \Longrightarrow set(T(i', j)) = CYK\ G\ w\ i'\ j$
and $1 \le j'$
and $i' + j' \le length\ w$
shows $set((main\ G\ T\ (length\ w)\ i\ j)(i', j')) = CYK\ G\ w\ i'\ j'$
proof –
have $\forall len\ T'\ w.\ main\ G\ T\ len\ i\ j = T' \longrightarrow length\ w = len \longrightarrow 1 < j \longrightarrow i + j \le len \longrightarrow$
$(\forall j' < j.\ \forall i'.\ 1 \le j' \longrightarrow i' + j' \le len \longrightarrow set(T(i', j')) = CYK\ G\ w\ i'\ j') \longrightarrow$
$(\forall i' < i.\ i' + j \le len \longrightarrow set(T(i', j)) = CYK\ G\ w\ i'\ j) \longrightarrow$
$(\forall j' \ge 1.\ \forall i'.\ i' + j' \le len \longrightarrow set(T'(i', j')) = CYK\ G\ w\ i'\ j')$ (**is** $\forall len.$ *?P G T len i j*)
proof(*rule allI*, *induct-tac G T len i j rule*: *main.induct*, (*drule meta-spec*, *drule meta-mp*, *rule refl*)+, *clarify*)
fix $G\ T\ i\ j\ i'\ j'$
fix w :: *'a list*
assume a: $i + j < length\ w \Longrightarrow$ *?P G* $(T((i, j) := inner\ G\ T\ i\ 1\ j))$ $(length\ w)$ $(i + 1)\ j$

assume *b*: ¬ *i* + *j* < *length w* ⟹ *j* < *length w* ⟹ *?P G* (*T*((*i*, *j*) := *inner G T i 1 j*)) (*length w*) *0* (*j* + *1*)
assume *c*: *1* < *j*
assume *d*: *i* + *j* ≤ *length w*
assume *e*: (*1*::*nat*) ≤ *j'*
assume *f*: *i'* + *j'* ≤ *length w*
assume *g*: ∀ *j'* < *j*. ∀ *i'*. *1* ≤ *j'* ⟶ *i'* + *j'* ≤ *length w* ⟶ *set*(*T*(*i'*, *j'*)) = *CYK G w i' j'*
assume *h*: ∀ *i'* < *i*. *i'* + *j* ≤ *length w* ⟶ *set*(*T*(*i'*, *j*)) = *CYK G w i' j*

have *inner*: *set* (*inner G T i* (*Suc 0*) *j*) = *CYK G w i j*
proof(*rule set-eqI*, *subst inner*, *subst match-prods*, *subst CYK-eq2*, *rule d*, *rule c*, *simp*)
fix *X*
show (∃ *l*≥*Suc 0*. *l* < *j* ∧ (∃ *A* ∈ *set*(*T*(*i*, *l*)). ∃ *B* ∈ *set*(*T*(*i* + *l*, *j* − *l*)). (*X*, *Branch A B*) ∈ *set G*)) =
(∃ *A B*. (*X*, *Branch A B*) ∈ *set G* ∧ (∃ *k*. *A* ∈ *CYK G w i k* ∧ *B* ∈ *CYK G w* (*i* + *k*) (*j* − *k*) ∧ *Suc 0* ≤ *k* ∧ *k* < *j*)) (**is** *?L* = *?R*)
proof
assume *?L*
thus *?R*
proof(*clarsimp*, *rule-tac x*=*A* **in** *exI*, *rule-tac x*=*B* **in** *exI*, *simp*, *rule-tac x*=*l* **in** *exI*, *simp*)
fix *l A B*
assume *i*: *Suc 0* ≤ *l*
assume *j*: *l* < *j*
assume *k*: *A* ∈ *set*(*T*(*i*, *l*))
assume *l*: *B* ∈ *set*(*T*(*i* + *l*, *j* − *l*))
note *g*[*rule-format*, **where** *i'*=*i* **and** *j'*=*l*]
with *d i j* **have** *A*: *set*(*T*(*i*, *l*)) = *CYK G w i l* **by** *force*
note *g*[*rule-format*, **where** *i'*=*i* + *l* **and** *j'*=*j* − *l*]
with *d i j* **have** *set*(*T*(*i* + *l*, *j* − *l*)) = *CYK G w* (*i* + *l*) (*j* − *l*) **by** *force*
with *k l A* **show** *A* ∈ *CYK G w i l* ∧ *B* ∈ *CYK G w* (*i* + *l*) (*j* − *l*) **by** *simp*
qed
next
assume *?R*
thus *?L*
proof(*clarsimp*, *rule-tac x*=*k* **in** *exI*, *simp*)
fix *A B k*
assume *i*: *Suc 0* ≤ *k*
assume *j*: *k* < *j*
assume *k*: *A* ∈ *CYK G w i k*
assume *l*: *B* ∈ *CYK G w* (*i* + *k*) (*j* − *k*)
assume *m*: (*X*, *Branch A B*) ∈ *set G*
note *g*[*rule-format*, **where** *i'*=*i* **and** *j'*=*k*]
with *d i j* **have** *A*: *CYK G w i k* = *set*(*T*(*i*, *k*)) **by** *force*
note *g*[*rule-format*, **where** *i'*=*i* + *k* **and** *j'*=*j* − *k*]
with *d i j* **have** *CYK G w* (*i* + *k*) (*j* − *k*) = *set*(*T*(*i* + *k*, *j* − *k*)) **by** *force*
with *k l A* **have** *A* ∈ *set*(*T*(*i*, *k*)) ∧ *B* ∈ *set*(*T*(*i* + *k*, *j* − *k*)) **by** *simp*

with *m* **show** $\exists A \in set(T(i, k)).\ \exists B \in set(T(i + k, j - k)).\ (X, Branch\ A\ B) \in set\ G$ **by** *force*
qed
qed
qed

show $set((main\ G\ T\ (length\ w)\ i\ j)(i', j')) = CYK\ G\ w\ i'\ j'$
proof(*case-tac* $i + j = length\ w$)
assume *i*: $i + j = length\ w$
show *?thesis*
proof(*case-tac* $j < length\ w$)
assume *j*: $j < length\ w$
show *?thesis*
proof(*subst main.simps*, *simp add*: *Let-def i j*,
rule b[*rule-format*, **where** $w{=}w$ **and** $i'{=}i'$ **and** $j'{=}j'$, *OF - - refl*, *simplified*],

simp-all add: *inner*)
from *i* **show** $\neg\ i + j < length\ w$ **by** *simp*
next
from *c* **show** $0 < j$ **by** *simp*
next
from *j* **show** $Suc\ j \le length\ w$ **by** *simp*
next
from *e* **show** $Suc\ 0 \le j'$ **by** *simp*
next
from *f* **show** $i' + j' \le length\ w$ **by** *assumption*
next
fix $i''\ j''$
assume *k*: $j'' < Suc\ j$
assume *l*: $Suc\ 0 \le j''$
assume *m*: $i'' + j'' \le length\ w$
show $(i'' = i \longrightarrow j'' \ne j) \longrightarrow set(T(i'',j'')) = CYK\ G\ w\ i''\ j''$
proof(*case-tac* $j'' = j$, *simp-all*, *clarify*)
assume *n*: $j'' = j$
assume $i'' \ne i$
with *i m n* **have** $i'' < i$ **by** *simp*
with *n m h* **show** $set(T(i'', j)) = CYK\ G\ w\ i''\ j$ **by** *simp*
next
assume $j'' \ne j$
with *k* **have** $j'' < j$ **by** *simp*
with *l m g* **show** $set(T(i'', j'')) = CYK\ G\ w\ i''\ j''$ **by** *simp*
qed
qed
next
assume $\neg\ j < length\ w$
with *i* **have** *j*: $i = 0 \land j = length\ w$ **by** *simp*
show *?thesis*
proof(*subst main.simps*, *simp add*: *Let-def j*, *intro conjI*, *clarify*)
from *j* **and** *inner* **show** $set\ (inner\ G\ T\ 0\ (Suc\ 0)\ (length\ w)) = CYK\ G\ w\ 0$

($length\ w$) **by** *simp*
next
show $0 < i' \longrightarrow set(T(i', j')) = CYK\ G\ w\ i'\ j'$
proof
assume $0 < i'$
with j **and** f **have** $j' < j$ **by** *simp*
with $e\ g\ f$ **show** $set(T(i', j')) = CYK\ G\ w\ i'\ j'$ **by** *simp*
qed
next
show $j' \neq length\ w \longrightarrow set(T(i', j')) = CYK\ G\ w\ i'\ j'$
proof
assume $j' \neq length\ w$
with j **and** f **have** $j' < j$ **by** *simp*
with $e\ g\ f$ **show** $set(T(i', j')) = CYK\ G\ w\ i'\ j'$ **by** *simp*
qed
qed
qed
next
assume $i + j \neq length\ w$
with d **have** i: $i + j < length\ w$ **by** *simp*
show *?thesis*
proof(*subst main.simps*, *simp add*: *Let-def i*,
rule a[*rule-format*, **where** $w{=}w$ **and** $i'{=}i'$ **and** $j'{=}j'$, *OF i*, *OF refl*, *simplified*])
from c **show** $Suc\ 0 < j$ **by** *simp*
next
from i **show** $Suc(i + j) \leq length\ w$ **by** *simp*
next
from e **show** $Suc\ 0 \leq j'$ **by** *simp*
next
from f **show** $i' + j' \leq length\ w$ **by** *assumption*
next
fix $i''\ j''$
assume $j'' < j$
and $Suc\ 0 \leq j''$
and $i'' + j'' \leq length\ w$
with g **show** $set(T(i'', j'')) = CYK\ G\ w\ i''\ j''$ **by** *simp*
next
fix i'' **assume** j: $i'' < Suc\ i$
show $set(if\ i'' = i\ then\ inner\ G\ T\ i\ (Suc\ 0)\ j\ else\ T(i'', j)) = CYK\ G\ w\ i''\ j$
proof(*simp split*: *if-split*, *rule conjI*, *clarify*, *rule inner*, *clarify*)
assume $i'' \neq i$
with j **have** $i'' < i$ **by** *simp*
with $d\ h$ **show** $set(T(i'', j)) = CYK\ G\ w\ i''\ j$ **by** *simp*
qed
qed
qed
qed
with *assms* **show** *?thesis* **by** *force*

qed

4.2 Initialisation phase

Similarly to *match-prods* above, here we collect non-terminals from which the given terminal symbol can be derived.

fun *init-match* :: ('n, 't) *CNG* ⇒ 't ⇒ 'n *list*
where *init-match* [] *t* = [] |
    init-match ((*X*, *Branch A B*)#*ps*) *t* = *init-match ps t* |
    init-match ((*X*, *Leaf a*)#*ps*) *t* = (*if a* = *t then X* # *init-match ps t*
        else init-match ps t)

lemma *init-match* :
(*X* ∈ *set*(*init-match G a*)) =
((*X*, *Leaf a*) ∈ *set G*)
by(*induct-tac G a rule*: *init-match.induct*, *simp-all*)

Representing the empty table.

definition *emptyT* = (λ(*i*, *j*). [])

The following function initialises the empty table for subwords of length 1, i.e. each symbol occurring in the given word.

fun *init'* :: ('n, 't) *CNG* ⇒ 't *list* ⇒ *nat* ⇒ *nat* × *nat* ⇒ 'n *list*
where *init'* *G* [] *k* = *emptyT* |
    init' *G* (*t*#*ts*) *k* = (*init'* *G ts* (*k* + *1*))((*k*, *1*) := *init-match G t*)

lemma *init'* :
assumes *i* + *1* ≤ *length w*
shows *set*(*init'* *G w 0* (*i*, *1*)) = *CYK G w i 1*
proof −
 have ∀ *i*. *Suc i* ≤ *length w* ⟶
    (∀ *k*. *set*(*init'* *G w k* (*k* + *i*, *Suc 0*)) = *CYK G w i* (*Suc 0*)) (**is** ∀ *i*. *?P i w* ⟶ (∀ *k*. *?Q i k w*))
 proof(*induct-tac w*, *clarsimp*+, *rule conjI*, *clarsimp*, *rule set-eqI*, *subst init-match*)
  fix *x w S*
   show ((*S*, *Leaf x*) ∈ *set G*) = (*S* ∈ *CYK G* (*x*#*w*) *0* (*Suc 0*)) **by**(*subst CYK-eq1*[*simplified*], *simp-all*)
 next
  fix *x w i*
  assume *a*: ∀ *i*. *?P i w* ⟶ (∀ *k*. *?Q i k w*)
  assume *b*: *i* ≤ *length w*
  show *0* < *i* ⟶ (∀ *k*. *set*(*init'* *G w* (*Suc k*) (*k* + *i*, *Suc 0*)) = *CYK G* (*x*#*w*) *i* (*Suc 0*))
  proof(*clarify*, *case-tac i*, *simp-all*, *subst CYK-eq1*[*simplified*], *simp*, *erule subst*, *rule b*, *simp*)
   fix *k j*

assume *c*: *i* = *Suc j*
note *a*[*rule-format*, **where** *i*=*j* **and** *k*=*Suc k*]
with *b* **and** *c* **have** *set*(*init′ G w* (*Suc k*) (*Suc k* + *j*, *Suc 0*)) = *CYK G w j* (*Suc 0*) **by** *simp*
also with *b* **and** *c* **have** ... = {*S*. (*S*, *Leaf* (*w* ! *j*)) ∈ *set G*} **by**(*subst CYK-eq1*[*simplified*], *simp-all*)
finally show *set*(*init′ G w* (*Suc k*) (*Suc* (*k* + *j*), *Suc 0*)) = {*S*. (*S*, *Leaf* (*w* ! *j*)) ∈ *set G*} **by** *simp*
qed
qed
with *assms* **have** ∀ *k*. *?Q i k w* **by** *simp*
note *this*[*rule-format*, **where** *k*=*0*]
thus *?thesis* **by** *simp*
qed

The next version of initialization refines *init′* in that it takes additional account of the cases when the given word is empty or contains a terminal symbol that does not have any matching production (that is, *init-match* is an empty list). No initial table is then needed as such words can immediately be rejected.

fun *init* :: (′*n*, ′*t*) *CNG* ⇒ ′*t list* ⇒ *nat* ⇒ (*nat* × *nat* ⇒ ′*n list*) *option*
where *init G* [] *k* = *None* |
init G [*t*] *k* = (*case* (*init-match G t*) *of*
[] ⇒ *None*
| *xs* ⇒ *Some*(*emptyT*((*k*, *1*) := *xs*))) |
init G (*t*#*ts*) *k* = (*case* (*init-match G t*) *of*
[] ⇒ *None*
| *xs* ⇒ (*case* (*init G ts* (*k* + *1*)) *of*
None ⇒ *None*
| *Some T* ⇒ *Some*(*T*((*k*, *1*) := *xs*))))

lemma *init1*:
‹*init′ G w k* = *T*› **if** ‹*init G w k* = *Some T*›
using *that* **by** (*induction G w k arbitrary*: *T rule*: *init.induct*)
(*simp-all split*: *list.splits option.splits*)

lemma *init2* :
(*init G w k* = *None*) =
(*w* = [] ∨ (∃ *a* ∈ *set w*. *init-match G a* = []))
by(*induct-tac G w k rule*: *init.induct*, *simp*, *simp split*: *list.split*,
simp split: *list.split option.split*, *force*)

4.3 The overall procedure

definition *cyk G S w* = (*case init G w 0 of*
None ⇒ *False*
| *Some T* ⇒ *let len* = *length w in*
if len = *1 then mem S* (*T*(*0*, *1*))

else let T′ = main G T len 0 2 in
mem S (T′(0, len)))

theorem *cyk* :
cyk G S w = (*w* ∈ *Lang G S*)
proof(*simp add*: *cyk-def split*: *option.split*, *simp-all add*: *Let-def*,
rule conjI, *subst init2*, *simp*, *rule conjI*)
show *w* = [] ⟶ [] ∉ *Lang G S* **by**(*clarify*, *drule Lang-no-Nil*, *clarify*)
next
show (∃ *x*∈*set w*. *init-match G x* = []) ⟶ *w* ∉ *Lang G S* **by**(*clarify*, *drule Lang-term*, *subst* (*asm*) *init-match*[*THEN sym*], *force*)
next
show ∀ *T*. *init G w 0* = *Some T* ⟶
((*length w* = *Suc 0* ⟶ *S* ∈ *set*(*T*(*0*, *Suc 0*))) ∧
(*length w* ≠ *Suc 0* ⟶ *S* ∈ *set*(*main G T* (*length w*) *0 2* (*0*, *length w*)))) =
(*w* ∈ *Lang G S*) (**is** ∀ *T*. *?P T* ⟶ *?L T* = *?R*)
proof *clarify*
fix *T*
assume *a*: *?P T*
hence *b*: *init′ G w 0* = *T* **by**(*rule init1*)
note *init2*[*THEN iffD2*, *OF disjI1*]
have *c*: *w* ≠ [] **by**(*clarify*, *drule init2*[**where** *G*=*G* **and** *k*=*0*, *THEN iffD2*, *OF disjI1*], *simp add*: *a*)
have *?L* (*init′ G w 0*) = *?R*
proof(*case-tac length w* = *1*, *simp-all*)
assume *d*: *length w* = *Suc 0*
show *S* ∈ *set*(*init′ G w 0* (*0*, *Suc 0*)) = *?R*
by(*subst init′*[*simplified*], *simp add*: *d*, *subst CYK-Lang*[*THEN sym*], *simp add*: *d*)
next
assume *length w* ≠ *Suc 0*
with *c* **have** *1* < *length w* **by**(*case-tac w*, *simp-all*)
hence *d*: *Suc*(*Suc 0*) ≤ *length w* **by** *simp*
show (*S* ∈ *set*(*main G* (*init′ G w 0*) (*length w*) *0 2* (*0*, *length w*))) = (*w* ∈ *Lang G S*)
proof(*subst main*, *simp-all*, *rule d*)
fix *i′ j′*
assume *j′* < *2* **and** *Suc 0* ≤ *j′*
hence *e*: *j′* = *1* **by** *simp*
assume *i′* + *j′* ≤ *length w*
with *e* **have** *f*: *i′* + *1* ≤ *length w* **by** *simp*
have *set*(*init′ G w 0* (*i′*, *1*)) = *CYK G w i′ 1* **by**(*rule init′*, *rule f*)
with *e* **show** *set*(*init′ G w 0* (*i′*, *j′*)) = *CYK G w i′ j′* **by** *simp*
next
from *d* **show** *Suc 0* ≤ *length w* **by** *simp*
next
show (*S* ∈ *CYK G w 0* (*length w*)) = (*w* ∈ *Lang G S*) **by**(*rule CYK-Lang*)

```
   qed
  qed
  with b show ?L T = ?R by simp
 qed
qed

value [code]
  let G = [(0::int, Branch 1 2), (0, Branch 2 3),
           (1, Branch 2 1), (1, Leaf ''a''),
           (2, Branch 3 3), (2, Leaf ''b''),
           (3, Branch 1 2), (3, Leaf ''a'')]
  in map (cyk G 0)
     [[''b'',''a'',''a'',''b'',''a''],
      [''b'',''a'',''b'',''a'']]

end
```

References

[1] D. H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.