

A formalisation of the Cocke-Younger-Kasami algorithm

Maksym Bortin

February 6, 2026

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

1	Basic modelling	2
1.1	Grammars in Chomsky normal form	2
1.2	Derivation by grammars	2
1.3	The generated language semantics	3
2	Basic properties	3
2.1	Properties of generated languages	9
3	Abstract specification of CYK	12
3.1	Properties of <i>subword</i>	12
3.2	Properties of <i>CYK</i>	13
4	Implementation	15
4.1	Main cycle	15
4.2	Initialisation phase	21
4.3	The overall procedure	22

```
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 $('n, 't) \text{ RHS} = \text{Branch } 'n \ 'n$
 $\quad \quad \quad | \text{Leaf } 't$

type-synonym $('n, 't) \text{ CNG} = ('n \times ('n, 't) \text{ RHS}) \text{ list}$

Abbreviating the list append symbol for better readability

abbreviation $\text{list-append} :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ (**infixr** $\langle \cdot \rangle$ 65)
where $xs \cdot ys \equiv xs @ ys$

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 $('n, 't) \text{ word-form} = ('n + 't) \text{ list}$

type-synonym $'t \text{ word} = 't \text{ list}$

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 $\text{DSTEP} :: ('n, 't) \text{ CNG} \Rightarrow (('n, 't) \text{ word-form} \times ('n, 't) \text{ word-form})$
 set

where $\text{DSTEP } G = \{ (l \cdot [\text{Inl } N] \cdot r, x) \mid l N r \text{ rhs } x. (N, \text{rhs}) \in \text{set } G \wedge$
 $(\text{case rhs of}$
 $\quad \text{Branch } A B \Rightarrow x = l \cdot [\text{Inl } A, \text{Inl } B] \cdot r$
 $\quad | \text{Leaf } t \Rightarrow x = l \cdot [\text{Inr } t] \cdot r) \}$

abbreviation $\text{DSTEP}' :: ('n, 't) \text{ word-form} \Rightarrow ('n, 't) \text{ CNG} \Rightarrow ('n, 't) \text{ word-form}$
 $\Rightarrow \text{bool} (\langle \cdot \rangle \text{ ---} \rightarrow \rightarrow [60, 61, 60] 61)$

where $w -G \rightarrow w' \equiv (w, w') \in \text{DSTEP } G$

abbreviation $DSTEP\text{-reflc} :: ('n, 't) \text{ word-form} \Rightarrow ('n, 't) \text{ CNG} \Rightarrow ('n, 't) \text{ word-form} \Rightarrow \text{bool}$ ($\langle - \dashrightarrow^= \rightarrow [60, 61, 60] 61$)
where $w - G \dashrightarrow^= w' \equiv (w, w') \in (DSTEP\ G)^=$

abbreviation $DSTEP\text{-transc} :: ('n, 't) \text{ word-form} \Rightarrow ('n, 't) \text{ CNG} \Rightarrow ('n, 't) \text{ word-form} \Rightarrow \text{bool}$ ($\langle - \dashrightarrow^+ \rightarrow [60, 61, 60] 61$)
where $w - G \dashrightarrow^+ w' \equiv (w, w') \in (DSTEP\ G)^+$

abbreviation $DSTEP\text{-rtransc} :: ('n, 't) \text{ word-form} \Rightarrow ('n, 't) \text{ CNG} \Rightarrow ('n, 't) \text{ word-form} \Rightarrow \text{bool}$ ($\langle - \dashrightarrow^* \rightarrow [60, 61, 60] 61$)
where $w - G \dashrightarrow^* w' \equiv (w, w') \in (DSTEP\ G)^*$

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 $Lang :: ('n, 't) \text{ CNG} \Rightarrow 'n \Rightarrow 't \text{ word set}$
where $Lang\ G\ S = \{w. [Inl\ S] - G \dashrightarrow^+ \text{ map } Inr\ w\}$

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

definition $G\text{-anbn} =$
 $[(\text{"S"}, \text{Branch } \text{"A"}\ \text{"T"}),$
 $(\text{"S"}, \text{Branch } \text{"A"}\ \text{"B"}),$
 $(\text{"T"}, \text{Branch } \text{"S"}\ \text{"B"}),$
 $(\text{"A"}, \text{Leaf } \text{"a"}),$
 $(\text{"B"}, \text{Leaf } \text{"b"})]$

Now the term $Lang\ G\text{-anbn}\ \text{"S"}$ 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 $prod\text{-into-}DSTEP1 :$
 $(S, \text{Branch } A\ B) \in \text{set } G \implies$
 $L \cdot [Inl\ S] \cdot R - G \rightarrow L \cdot [Inl\ A, Inl\ B] \cdot R$
by($simp\ add: DSTEP\text{-def}, rule\text{-tac } x=L\ \mathbf{in}\ exI, force$)

lemma $prod\text{-into-}DSTEP2 :$
 $(S, \text{Leaf } a) \in \text{set } G \implies$
 $L \cdot [Inl\ S] \cdot R - G \rightarrow L \cdot [Inr\ a] \cdot R$

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

lemma *DSTEP-D* :

$s - G \rightarrow t \implies$

$\exists L N R rhs. s = L \cdot [Inl N] \cdot R \wedge (N, rhs) \in set G \wedge$

$(\forall A B. rhs = Branch A B \longrightarrow t = L \cdot [Inl A, Inl B] \cdot R) \wedge$

$(\forall x. rhs = Leaf x \longrightarrow t = L \cdot [Inr x] \cdot R)$

by(*unfold DSTEP-def, clarsimp, simp split: RHS.split-asm, blast+*)

lemma *DSTEP-append* :

assumes $a: s - G \rightarrow t$

shows $L \cdot s \cdot R - G \rightarrow L \cdot t \cdot R$

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*

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* :

$s - G \rightarrow^* t \implies length s \leq length t$

proof(*erule rtrancl-induct, simp*)

fix $t u$

assume $s - G \rightarrow^* t$

assume $a: t - G \rightarrow u$

assume $b: length s \leq length t$

show $length s \leq length u$

proof –

from a **have** $\exists L N R rhs. t = L \cdot [Inl N] \cdot R \wedge (N, rhs) \in set G \wedge$

$(\forall A B. rhs = Branch A B \longrightarrow u = L \cdot [Inl A, Inl B] \cdot R) \wedge$

$(\forall x. rhs = Leaf x \longrightarrow u = L \cdot [Inr x] \cdot R)$ (**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+*)

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** $?C1 \vee ?C2$) **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 \cdot 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 · [Inl A, Inl B] · R ∧ (N, Branch A B) ∈ 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 · [Inl A, Inl B] · 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 · r -G→* t
shows ∃ l' r'. l -G→* l' ∧ r -G→* r' ∧ t = l' · r'
proof -
have ∧s. s -G→* t ⇒
  ∀ l r. s = l · r ⇒ (∃ l' r'. l -G→* l' ∧ r -G→* r' ∧ t = l' · r') (is ∧s.
?P s t ⇒ ?Q s t)
proof(erule rtrancl-induct, force)
fix s t u
assume ?P s t
assume a: t -G→ 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 $l -G\rightarrow^* l' \implies$
 $\forall r r'. r -G\rightarrow^* r' \implies l \cdot r -G\rightarrow^* l' \cdot r'$ (**is** $?P\ l\ l' \implies ?Q\ l\ l'$)
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 $[Inl\ S] - G \rightarrow^+ t \implies$
 $\forall x. Inr\ x \in set\ t \longrightarrow (\exists N. (N, Leaf\ x) \in set\ G)$ (**is** *?P t \implies ?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* :

$w \in \text{Lang } G \ S \implies w \neq []$

by(*simp add: Lang-def, drule trancl-into-rtrancl, drule DSTEP-star-mono, force*)

lemma *Lang-rtrancl-eq* :

$(w \in \text{Lang } G \ S) = [\text{Inl } S] -G \rightarrow^* \text{map } \text{Inr } w \quad (\text{is } ?L = (?p \in ?R^*))$

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 $[\text{Inl } S] = \text{map } \text{Inr } w \vee ?p \in ?R^+$ **by** *force*

thus $?p \in ?R^+$ **by**(*case-tac w, simp-all*)

qed

lemma *Lang-term* :

$w \in \text{Lang } G \ S \implies$

$\forall x \in \text{set } w. \exists N. (N, \text{Leaf } x) \in \text{set } G$

by(*clarsimp simp add: Lang-def, drule DSTEP-trancl-term, simp, erule imageI, assumption*)

lemma *Lang-eq1* :

$([x] \in \text{Lang } G \ S) = ((S, \text{Leaf } x) \in \text{set } G)$

proof(*simp add: Lang-def, rule iffI, subst (asm) trancl-unfold-left,clarsimp*)

fix t

assume $a: [\text{Inl } S] -G \rightarrow t$

assume $b: t -G \rightarrow^* [\text{Inr } x]$

note *DSTEP-star-mono[OF b, simplified]*

hence $c: \text{length } t \leq 1$ **by** *simp*

have $\exists z. t = [z]$

proof(*cases t*)

assume $t = []$

with b **have** $d: [] -G \rightarrow^* [\text{Inr } x]$ **by** *simp*

have $\wedge s. ([], s) \in (\text{DSTEP } G)^* \implies s = []$

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($clarsimp\ simp\ add: DST\ EP-def, simp\ split: RHS.split-asm, case-tac\ l, simp-all$)

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**($clarsimp\ simp\ add: DST\ EP-def, case-tac\ l, simp-all$)
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($rule\ r-into-trancl, simp\ add: DST\ EP-def, rule-tac\ x=[]$ **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 $t = [Inr\ x]$
with c **have** $d: [Inr\ x] -G \rightarrow^* \text{map } Inr\ w$ **by** *simp*
have $\bigwedge x\ s. [Inr\ x] -G \rightarrow^* s \implies s = [Inr\ x]$
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** $d: (S, \text{Branch } A\ B) \in \text{set } G \wedge t = [Inl\ A, Inl\ B]$ **by**
blast
with c **have** $e: [Inl\ A] \cdot [Inl\ B] -G \rightarrow^* \text{map } Inr\ w$ **by** *simp*
note *DSTEP-star-comp1[OF e]*
then obtain $l'\ r'$ **where** $e: [Inl\ A] -G \rightarrow^* l' \wedge [Inl\ B] -G \rightarrow^* r' \wedge$
 $\text{map } Inr\ w = l' \cdot r'$ **by** *blast*
note *map-Inr-split[rule-format, OF e[THEN conjunct2, THEN conjunct2]]*
then obtain $u\ v$ **where** $f: w = u \cdot v \wedge l' = \text{map } Inr\ u \wedge r' = \text{map } Inr\ v$ **by**
blast
with e **have** $g: [Inl\ A] -G \rightarrow^* \text{map } Inr\ u \wedge [Inl\ B] -G \rightarrow^* \text{map } Inr\ v$ **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** $a: (S, \text{Branch } A\ B) \in \text{set } G \wedge w = l \cdot r \wedge l \in \text{Lang}$
 $G\ A \wedge r \in \text{Lang } G\ B$ **by** *blast*
have $[Inl\ A] \cdot [Inl\ B] -G \rightarrow^* \text{map } Inr\ l \cdot \text{map } Inr\ r$
by(*rule DSTEP-star-comp2, subst Lang-rtrancl-eq[THEN sym], simp add: a,*
subst Lang-rtrancl-eq[THEN sym], simp add: a)
hence $b: [Inl\ A] \cdot [Inl\ B] -G \rightarrow^* \text{map } Inr\ w$ **by**(*simp add: a*)
have $c: w \in \text{Lang } G\ S$
by(*simp add: Lang-def, subst trancl-unfold-left, rule-tac b=[Inl A] \cdot [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 $1 < \text{length } w$
proof(*simp add: a, rule ccontr, drule leI*)
assume $\text{length } l + \text{length } r \leq \text{Suc } 0$
hence $l = [] \vee r = []$ **by**(*case-tac l, simp-all*)
thus *False*
proof
assume $l = []$
with a **have** $[] \in \text{Lang } G\ A$ **by** *simp*

```

note Lang-no-Nil[OF this]
thus ?thesis by simp
next
assume  $r = []$ 
with  $a$  have  $[] \in \text{Lang } G \ B$  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* $w \ i \ j = \text{take } j \ (\text{drop } i \ w)$

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* $G \ w \ i \ j = \{S. \text{subword } w \ i \ j \in \text{Lang } G \ S\}$

3.1 Properties of *subword*

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

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

lemma *subword-nth2* :
assumes $A: i + 1 \leq \text{length } w$
shows $\text{subword } w \ i \ 1 = [w!i]$
proof –
note *subword-length*[*OF A*]
hence $\exists x. \text{subword } w \ i \ 1 = [x]$ **by**(*case-tac subword w i 1, simp-all*)
then obtain x **where** $a: \text{subword } w \ i \ 1 = [x]$ **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*] :
 $\forall n m. n \leq \text{length } xs \longrightarrow n \leq m \longrightarrow$
 $\text{take } n \text{ } xs \cdot \text{take } (m - n) (\text{drop } n \text{ } xs) = \text{take } m \text{ } xs$
by(*induct-tac xs, clarsimp+, case-tac n, simp-all, case-tac m, simp-all*)

lemma *subword-split* :
 $i + j \leq \text{length } w \implies 0 < k \implies k < j \implies$
 $\text{subword } w \text{ } i \text{ } j = \text{subword } w \text{ } i \text{ } k \cdot \text{subword } w \text{ } (i + k) \text{ } (j - k)$
by(*simp add: subword-def, subst take-split[where n=k, THEN sym], simp-all,*
rule-tac f= $\lambda x. \text{take } (j - k) (\text{drop } x \text{ } w)$ in arg-cong, simp)

lemma *subword-split2* :
assumes *A*: $\text{subword } w \text{ } i \text{ } j = l \cdot r$
and *B*: $i + j \leq \text{length } w$
and *C*: $0 < \text{length } l$
and *D*: $0 < \text{length } r$
shows $l = \text{subword } w \text{ } i (\text{length } l) \wedge r = \text{subword } w \text{ } (i + \text{length } l) (j - \text{length } l)$
proof –
have *a*: $\text{length}(\text{subword } w \text{ } i \text{ } j) = j$ **by**(*rule subword-length, rule B*)
note *arg-cong*[**where** $f = \text{length}$, *OF A*]
with *a* **and** *D* **have** *b*: $\text{length } l < j$ **by** *force*
with *B* **have** *c*: $i + \text{length } l \leq \text{length } w$ **by** *force*
have $\text{subword } w \text{ } i \text{ } j = \text{subword } w \text{ } i (\text{length } l) \cdot \text{subword } w \text{ } (i + \text{length } l) (j - \text{length } l)$
by(*rule subword-split, rule B, rule C, rule b*)
with *A* **have** *d*: $l \cdot r = \text{subword } w \text{ } i (\text{length } l) \cdot \text{subword } w \text{ } (i + \text{length } l) (j - \text{length } l)$ **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* :
 $(S \in \text{CYK } G \text{ } w \text{ } 0 (\text{length } w)) = (w \in \text{Lang } G \text{ } S)$
by(*simp add: CYK-def subword-self*)

lemma *CYK-eq1* :

$i + 1 \leq \text{length } w \implies$

$\text{CYK } G \ w \ i \ 1 = \{S. (S, \text{Leaf } (w!i)) \in \text{set } G\}$

by(*simp add: CYK-def, subst subword-nth2[simplified], assumption,*
subst Lang-eq1, rule refl)

theorem *CYK-eq2* :

assumes $A: i + j \leq \text{length } w$

and $B: 1 < j$

shows $\text{CYK } G \ w \ i \ j = \{X \mid X \ A \ B \ k. (X, \text{Branch } A \ B) \in \text{set } G \wedge A \in \text{CYK } G \ w \ i \ k \wedge B \in \text{CYK } G \ w \ (i + k) \ (j - k) \wedge 1 \leq k \wedge k < j\}$

proof(*rule set-eqI, rule iffI, simp-all add: CYK-def*)

fix X

assume $a: \text{subword } w \ i \ j \in \text{Lang } G \ X$

show $\exists A \ B. (X, \text{Branch } A \ B) \in \text{set } G \wedge (\exists k. \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j)$

proof –

have $b: 1 < \text{length}(\text{subword } w \ i \ j)$ **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** $c: (X, \text{Branch } A \ B) \in \text{set } G \wedge \text{subword } w \ i \ j = l \cdot r \wedge l \in \text{Lang } G \ A \wedge r \in \text{Lang } G \ B$ **by** *blast*

note *Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct1]]*

hence $d: 0 < \text{length } l$ **by**(*case-tac l, simp-all*)

note *Lang-no-Nil[OF c[THEN conjunct2, THEN conjunct2, THEN conjunct2]]*

hence $e: 0 < \text{length } r$ **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 $\text{Suc } 0 \leq \text{length } l \wedge \text{length } l < j$ (**is** *?A* \wedge *?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 $\text{length}(\text{subword } w \ i \ j) = j$ **by**(*rule subword-length, rule A*)

finally have $\text{length } l + \text{length } r = j$ **by** *simp*

with e **show** *?B* **by** *force*

qed

qed

next

fix X

assume $\exists A \ B. (X, \text{Branch } A \ B) \in \text{set } G \wedge (\exists k. \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j)$

then obtain $A \ B \ k$ **where** $a: (X, \text{Branch } A \ B) \in \text{set } G \wedge \text{subword } w \ i \ k \in \text{Lang } G \ A \wedge \text{subword } w \ (i + k) \ (j - k) \in \text{Lang } G \ B \wedge \text{Suc } 0 \leq k \wedge k < j$ **by** *blast*

```

show subword  $w$   $i$   $j \in \text{Lang } G \ X$ 
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  $x=\text{subword } w \ i \ k$  in  $exI$ , rule-tac  $x=\text{subword } w \ (i + k) \ (j - k)$  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  $\Rightarrow$  'a list  $\Rightarrow$  bool
where
  mem  $a$  [] = False |
  mem  $a$  ( $x\#xs$ ) = ( $x = a \vee$  mem  $a$   $xs$ )

```

```

lemma mem[simp] :
  mem  $x$   $xs$  = ( $x \in$  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  $\Rightarrow$  'n list  $\Rightarrow$  'n list  $\Rightarrow$  'n list
where match-prods []  $ls$   $rs$  = [] |
  match-prods (( $X$ , Branch  $A$   $B$ )# $ps$ )  $ls$   $rs$  =
    (if mem  $A$   $ls \wedge$  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 :
  ( $X \in$  set(match-prods  $G$   $ls$   $rs$ )) =
  ( $\exists A \in$  set  $ls$ .  $\exists B \in$  set  $rs$ . ( $X$ , Branch  $A$   $B$ )  $\in$  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  $i + j \leq \text{length } w$ 
    and  $\bigwedge i' j'. j' < j \implies 1 \leq j' \implies i' + j' \leq \text{length } w \implies \text{set}(T(i', j')) = \text{CYK } G w i' j'$ 
  and  $\bigwedge i'. i' < i \implies i' + j \leq \text{length } w \implies \text{set}(T(i', j)) = \text{CYK } G w i' j$ 
    and  $1 \leq j'$ 
    and  $i' + j' \leq \text{length } w$ 
  shows  $\text{set}((\text{main } G T (\text{length } w) i j)(i', j')) = \text{CYK } G w i' j'$ 
proof –
  have  $\forall \text{len } T' w. \text{main } G T \text{len } i j = T' \longrightarrow \text{length } w = \text{len} \longrightarrow 1 < j \longrightarrow i + j \leq \text{len} \longrightarrow$ 
     $(\forall j' < j. \forall i'. 1 \leq j' \longrightarrow i' + j' \leq \text{len} \longrightarrow \text{set}(T(i', j')) = \text{CYK } G w i' j')$ 
   $\longrightarrow$ 
     $(\forall i' < i. i' + j \leq \text{len} \longrightarrow \text{set}(T(i', j)) = \text{CYK } G w i' j) \longrightarrow$ 
     $(\forall j' \geq 1. \forall i'. i' + j' \leq \text{len} \longrightarrow \text{set}(T'(i', j')) = \text{CYK } G w i' j')$  (is  $\forall \text{len}. ?P G T \text{len } i j$ )
  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:  $i + j < \text{length } w \implies ?P G (T((i, j) := inner G T i 1 j)) (\text{length } w) (i + 1) j$ 

```

```

assume b:  $\neg i + j < \text{length } w \implies j < \text{length } w \implies ?P G (T((i, j) := \text{inner } G$ 
T i 1 j)) (length w) 0 (j + 1)
assume c:  $1 < j$ 
assume d:  $i + j \leq \text{length } w$ 
assume e:  $(1::\text{nat}) \leq j'$ 
assume f:  $i' + j' \leq \text{length } w$ 
assume g:  $\forall j' < j. \forall i'. 1 \leq j' \longrightarrow i' + j' \leq \text{length } w \longrightarrow \text{set}(T(i', j')) = \text{CYK}$ 
G w i' j'
assume h:  $\forall i' < i. i' + j \leq \text{length } w \longrightarrow \text{set}(T(i', j)) = \text{CYK } G w i' j$ 

have inner:  $\text{set}(\text{inner } G T i (\text{Suc } 0) j) = \text{CYK } G w i j$ 
proof(rule set-eqI, subst inner, subst match-prods, subst CYK-eq2, rule d, rule
c, simp)
fix X
show  $(\exists l \geq \text{Suc } 0. l < j \wedge (\exists A \in \text{set}(T(i, l)). \exists B \in \text{set}(T(i + l, j - l)). (X,$ 
Branch A B) \in \text{set } G)) =
 $(\exists A B. (X, \text{Branch } A B) \in \text{set } G \wedge (\exists k. A \in \text{CYK } G w i k \wedge B \in \text{CYK}$ 
G w (i + k) (j - k) \wedge \text{Suc } 0 \leq k \wedge 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:  $\text{Suc } 0 \leq l$ 
assume j:  $l < j$ 
assume k:  $A \in \text{set}(T(i, l))$ 
assume l:  $B \in \text{set}(T(i + l, j - l))$ 
note g[rule-format, where i'=i and j'=l]
with d i j have A:  $\text{set}(T(i, l)) = \text{CYK } G w i l$  by force
note g[rule-format, where i'=i + l and j'=j - l]
with d i j have  $\text{set}(T(i + l, j - l)) = \text{CYK } G w (i + l) (j - l)$  by force
with k l A show  $A \in \text{CYK } G w i l \wedge B \in \text{CYK } G w (i + l) (j - l)$  by simp
qed
next
assume ?R
thus ?L
proof(clarsimp, rule-tac x=k in exI, simp)
fix A B k
assume i:  $\text{Suc } 0 \leq k$ 
assume j:  $k < j$ 
assume k:  $A \in \text{CYK } G w i k$ 
assume l:  $B \in \text{CYK } G w (i + k) (j - k)$ 
assume m:  $(X, \text{Branch } A B) \in \text{set } G$ 
note g[rule-format, where i'=i and j'=k]
with d i j have A:  $\text{CYK } G w i k = \text{set}(T(i, k))$  by force
note g[rule-format, where i'=i + k and j'=j - k]
with d i j have  $\text{CYK } G w (i + k) (j - k) = \text{set}(T(i + k, j - k))$  by force
with k l A have  $A \in \text{set}(T(i, k)) \wedge B \in \text{set}(T(i + k, j - k))$  by simp

```

```

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

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

```

```

(length w) by simp
next
show  $0 < i' \longrightarrow \text{set}(T(i', j')) = \text{CYK } G \text{ w } i' j'$ 
proof
  assume  $0 < i'$ 
  with j and f have  $j' < j$  by simp
  with e g f show  $\text{set}(T(i', j')) = \text{CYK } G \text{ w } i' j'$  by simp
qed
next
show  $j' \neq \text{length } w \longrightarrow \text{set}(T(i', j')) = \text{CYK } G \text{ w } i' j'$ 
proof
  assume  $j' \neq \text{length } w$ 
  with j and f have  $j' < j$  by simp
  with e g f show  $\text{set}(T(i', j')) = \text{CYK } G \text{ w } i' j'$  by simp
qed
qed
next
assume  $i + j \neq \text{length } w$ 
with d have  $i: i + j < \text{length } w$  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  $\text{Suc } 0 < j$  by simp
next
  from i show  $\text{Suc}(i + j) \leq \text{length } w$  by simp
next
  from e show  $\text{Suc } 0 \leq j'$  by simp
next
  from f show  $i' + j' \leq \text{length } w$  by assumption
next
  fix  $i'' j''$ 
  assume  $j'' < j$ 
  and  $\text{Suc } 0 \leq j''$ 
  and  $i'' + j'' \leq \text{length } w$ 
  with g show  $\text{set}(T(i'', j'')) = \text{CYK } G \text{ w } i'' j''$  by simp
next
  fix  $i''$  assume  $j: i'' < \text{Suc } i$ 
  show  $\text{set}(if\ i'' = i\ then\ inner\ G\ T\ i\ (\text{Suc } 0)\ j\ else\ T(i'', j)) = \text{CYK } G \text{ w } i'' j$ 
  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  $\text{set}(T(i'', j)) = \text{CYK } G \text{ w } i'' j$  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  $c: i = \text{Suc } j$ 
note  $a[\text{rule-format, where } i=j \text{ and } k=\text{Suc } k]$ 
with  $b$  and  $c$  have  $\text{set}(\text{init}' G w (\text{Suc } k) (\text{Suc } k + j, \text{Suc } 0)) = \text{CYK } G w j$ 
 $(\text{Suc } 0)$  by simp
also with  $b$  and  $c$  have  $\dots = \{S. (S, \text{Leaf } (w ! j)) \in \text{set } G\}$  by(subst
CYK-eq1[simplified], simp-all)
finally show  $\text{set}(\text{init}' G w (\text{Suc } k) (\text{Suc } (k + j), \text{Suc } 0)) = \{S. (S, \text{Leaf } (w !$ 
 $j)) \in \text{set } G\}$  by simp
qed
qed
with assms have  $\forall k. ?Q i k w$  by simp
note  $\text{this}[\text{rule-format, where } k=0]$ 
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  $\Rightarrow$  't list  $\Rightarrow$  nat  $\Rightarrow$  (nat  $\times$  nat  $\Rightarrow$  'n list) option
where init  $G [] k = \text{None}$  |
  init  $G [t] k = (\text{case } (\text{init-match } G t) \text{ of}$ 
     $[] \Rightarrow \text{None}$ 
    |  $xs \Rightarrow \text{Some}(\text{emptyT}((k, 1) := xs)))$  |
  init  $G (t\#ts) k = (\text{case } (\text{init-match } G t) \text{ of}$ 
     $[] \Rightarrow \text{None}$ 
    |  $xs \Rightarrow (\text{case } (\text{init } G ts (k + 1)) \text{ of}$ 
       $\text{None} \Rightarrow \text{None}$ 
      |  $\text{Some } T \Rightarrow \text{Some}(T((k, 1) := xs)))$ )

```

```

lemma init1:
   $\langle \text{init}' G w k = T \rangle$  if  $\langle \text{init } G w k = \text{Some } T \rangle$ 
using that by (induction  $G w k$  arbitrary: T rule: init.induct)
  (simp-all split: list.splits option.splits)

```

```

lemma init2 :
  (init  $G w k = \text{None}$ ) =
  ( $w = [] \vee (\exists a \in \text{set } w. \text{init-match } G a = [])$ )
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  $G S w = (\text{case } \text{init } G w 0 \text{ of}$ 
   $\text{None} \Rightarrow \text{False}$ 
  |  $\text{Some } T \Rightarrow \text{let } \text{len} = \text{length } w \text{ in}$ 
   $\text{if } \text{len} = 1 \text{ then mem } S (T(0, 1))$ )

```

else let $T' = \text{main } G \ T \ \text{len } 0 \ 2$ in
 mem $S \ (T'(0, \text{len}))$)

theorem *cyk* :

cyk $G \ S \ w = (w \in \text{Lang } G \ S)$

proof(*simp* add: *cyk-def split: option.split, simp-all* add: *Let-def,*
rule conjI, subst init2, simp, rule conjI)

show $w = [] \longrightarrow [] \notin \text{Lang } G \ S$ **by**(*clarify, drule Lang-no-Nil, clarify*)

next

show $(\exists x \in \text{set } w. \text{init-match } G \ x = []) \longrightarrow w \notin \text{Lang } G \ S$ **by**(*clarify, drule*
Lang-term, subst (asm) init-match[THEN sym], force)

next

show $\forall T. \text{init } G \ w \ 0 = \text{Some } T \longrightarrow$
 $((\text{length } w = \text{Suc } 0 \longrightarrow S \in \text{set}(T(0, \text{Suc } 0))) \wedge$
 $(\text{length } w \neq \text{Suc } 0 \longrightarrow S \in \text{set}(\text{main } G \ T \ (\text{length } w) \ 0 \ 2 \ (0, \text{length } w)))) =$
 $(w \in \text{Lang } G \ S) \ (\text{is } \forall T. \ ?P \ T \longrightarrow \ ?L \ T = \ ?R)$

proof *clarify*

fix T

assume $a: \ ?P \ T$

hence $b: \text{init}' \ G \ w \ 0 = T$ **by**(*rule init1*)

note *init2[THEN iffD2, OF disjI1]*

have $c: w \neq []$ **by**(*clarify, drule init2[where G=G and k=0, THEN iffD2, OF*
disjI1], simp add: a)

have $\ ?L \ (\text{init}' \ G \ w \ 0) = \ ?R$

proof(*case-tac length w = 1, simp-all*)

assume $d: \text{length } w = \text{Suc } 0$

show $S \in \text{set}(\text{init}' \ G \ w \ 0 \ (0, \text{Suc } 0)) = \ ?R$

by(*subst init'[simplified], simp* add: $d, \text{subst CYK-Lang[THEN sym], simp add:
 d)$

next

assume $\text{length } w \neq \text{Suc } 0$

with c **have** $1 < \text{length } w$ **by**(*case-tac w, simp-all*)

hence $d: \text{Suc}(\text{Suc } 0) \leq \text{length } w$ **by** *simp*

show $(S \in \text{set}(\text{main } G \ (\text{init}' \ G \ w \ 0) \ (\text{length } w) \ 0 \ 2 \ (0, \text{length } w))) = (w \in$
 $\text{Lang } G \ S)$

proof(*subst main, simp-all, rule d*)

fix $i' \ j'$

assume $j' < 2$ **and** $\text{Suc } 0 \leq j'$

hence $e: j' = 1$ **by** *simp*

assume $i' + j' \leq \text{length } w$

with e **have** $f: i' + 1 \leq \text{length } w$ **by** *simp*

have $\text{set}(\text{init}' \ G \ w \ 0 \ (i', 1)) = \text{CYK } G \ w \ i' \ 1$ **by**(*rule init', rule f*)

with e **show** $\text{set}(\text{init}' \ G \ w \ 0 \ (i', j')) = \text{CYK } G \ w \ i' \ j'$ **by** *simp*

next

from d **show** $\text{Suc } 0 \leq \text{length } w$ **by** *simp*

next

show $(S \in \text{CYK } G \ w \ 0 \ (\text{length } w)) = (w \in \text{Lang } G \ S)$ **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 n^3 . *Information and Control*, 10(2):189 – 208, 1967.