Pushdown Automata

Hendrik Jan Hoogeboom
Leiden, NL

April 19, 2007

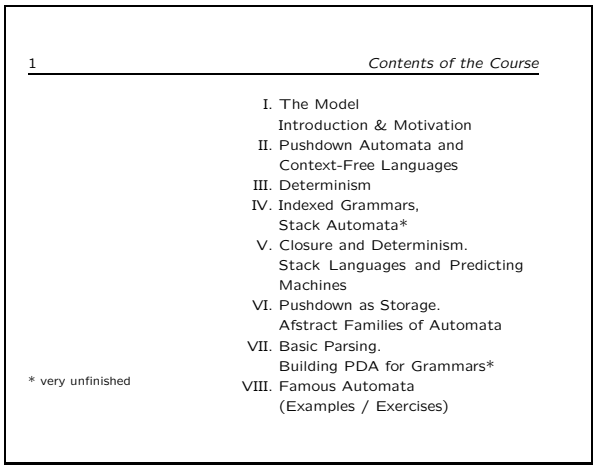


Figure 1: Contents of the Course

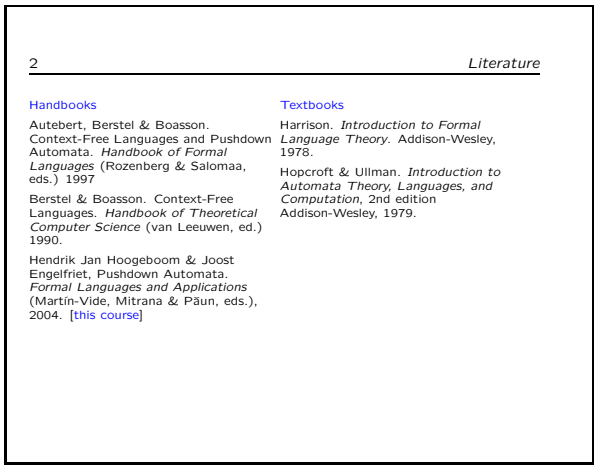


Figure 2: Literature

Transparencies made for a course at the International PhD School in Formal Languages and Applications, Rovira i Virgili University, Tarragona, Spain. With notes.

(*Fig. 1*) These transparencies are designed to replace my handwritten slides, and were not yet put to classroom test. They are very 'under construction', in particular near the end.

Home for the slides `http://www.liacs.nl/home/hoogeboo/praatjes/tarragona/`

(*Fig. 2*) Based on the ideas for this the course in Tarragona we (Hoogeboom & Engelfriet) have written a chapter on pushdown automata. Most of the topics presented here are discussed in that chapter.

The chapters from the other two handbooks also cover the standard material on PDA, each focussing on specific aspects.

1 The Model, Introduction & Motivation

(*Fig. 3*) The four levels of the Chomsky hierarchy. Four families of languages, with the grammatical models and machine models generating/recognizing them. This course focusses the pushdown automata from level 2.

(*Fig. 4*) The 'Tower of Hanoi' is a classic for Computer Science students as it is used by many to introduce the concepts of recursion. Here we use it to show two sides of this medal: the recursive ('grammatical') solution of the problem, and its implementation (using a stack = push-down).

"*The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a legend about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. The priests of Brahma, acting out the command of an ancient prophecy, have been mov-*

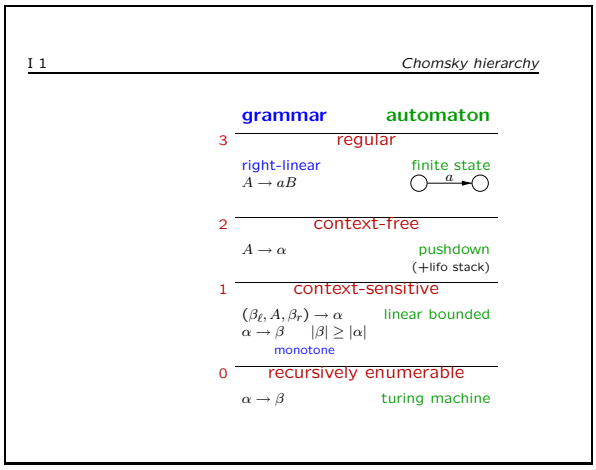


Figure 3: Chomsky hierarchy

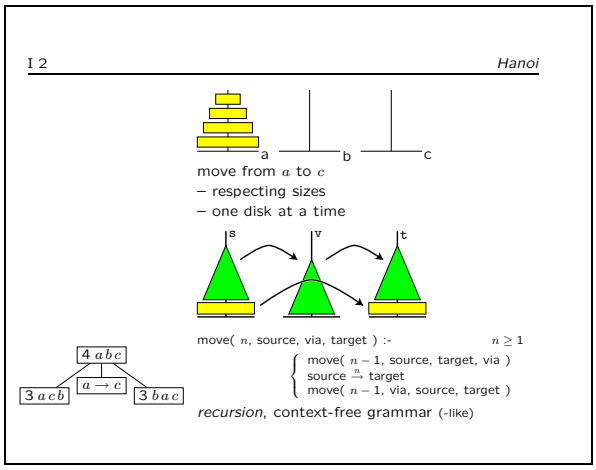


Figure 4: Hanoi

ing these disks, in accordance with the rules of the puzzle. According to the legend, when the last move of the puzzle is completed, the world will end."

wikipedia: Tower of Hanoi

(*Fig. 5*) Moving three disks (from a via b to c) in the recursive way leads to the following tree of recursive function calls. It is close to a derivation tree of a context-free grammar, but not really one: the parameter for the depth (a natural number) cannot be modelled by a cfg.

Another slide shows the stack of function calls as they are processed.

(*Fig. 7*) The PDA is a finite state device that reads its input from a (one-way) tape. Additionally it is

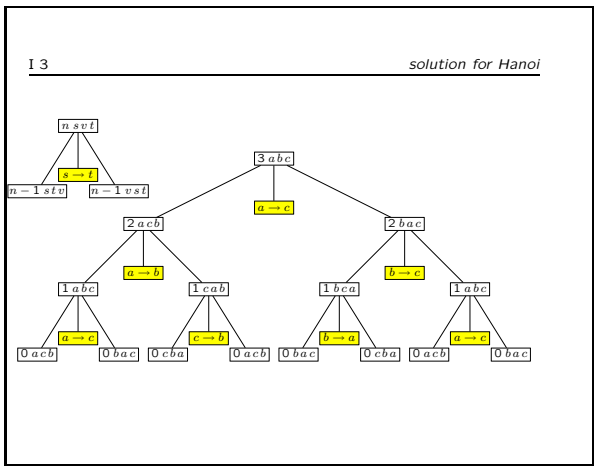


Figure 5: solution for Hanoi

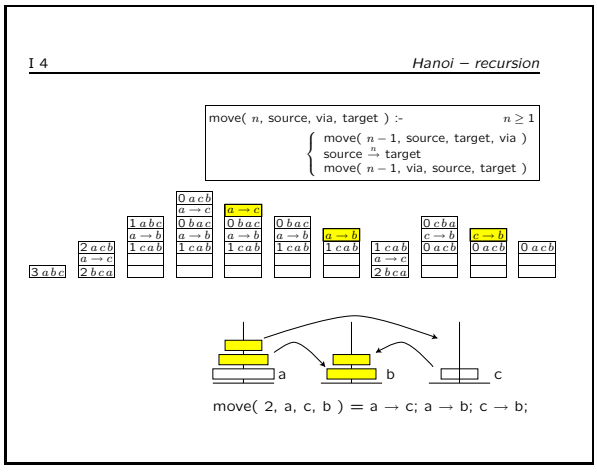


Figure 6: Hanoi – recursion

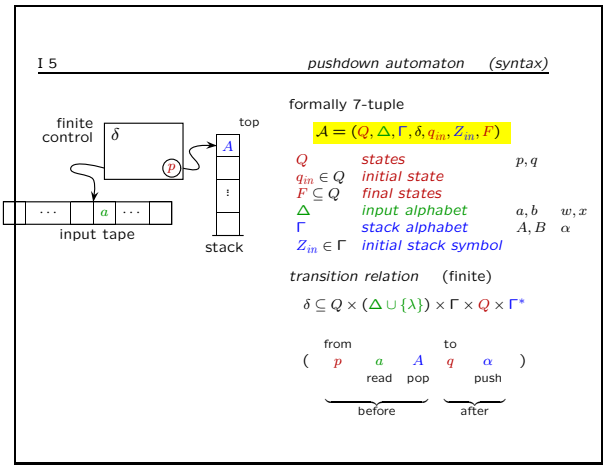


Figure 7: pushdown automaton (syntax)

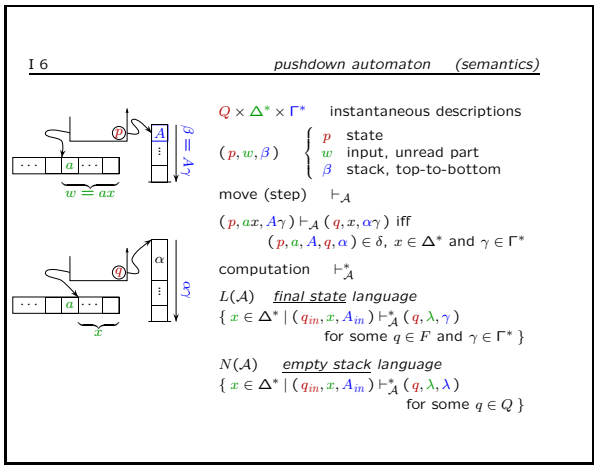


Figure 8: pushdown automaton (semantics)

equipped with external memory in the form of a LIFO stack to store information.

The formal specification is a 7-tuple, much like that of a finite state automaton, but additionally including the stack alphabet, and an initial stack symbol.

The transition relation consists of 5-tuples: based on the current state p, input symbol a, and topmost stack symbol A, it specifies a new state q and a new sequence of stack symbols α.

Our choice, representing the transitions as a relation $\delta \subseteq Q \times (\Delta \cup \{\lambda\}) \times \Gamma \times Q \times \Gamma^*$ is one of the standard choices. Alternatively one has a function $\delta : Q \times (\Delta \cup \{\lambda\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$, where $\mathcal{P}$ denotes the power set. This is just a notational variant. Where we write $(p, a, A, q, \alpha) \in \delta$, the other formalism has $(q, \alpha) \in \delta(p, a, A)$.

We like the relation better because it enables to talk about separate transitions. The function however clearly separates input (before) from output (after). In our 'chapter' we tend to write $(p, a, A) \mapsto (q, \alpha)$ as a compromise.

(*Fig. 8*) The 'global configuration' of a PDA consists of state, the part of the tape not yet read, and the stack. This is formalized as instantaneous description (ID).

Single moves of the PDA are the result of transitions applicable to the ID; sequences of moves form a computation.

There are two natural ways to accept a computation of a PDA: either by internal state, or by external storage. This leads to two languages for each

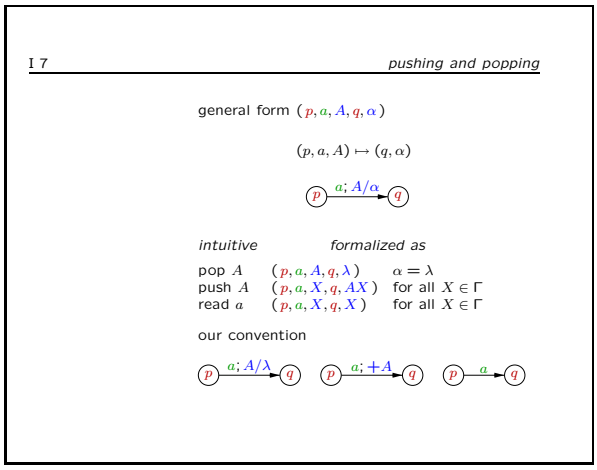


Figure 9: pushing and popping

PDA, $L(\mathcal{A})$ and $N(\mathcal{A})$, and two families of PDA languages, PDℓ and PDn.

(*Fig. 9*) A transition may read λ from the input, but it always pops a specific symbol A from the stack.

The intuitive actions 'push X' (regardless the contents of the stack) and 'read a' have to be coded as a set of transitions, one for each value of the top of the stack.

We introduce a personal pictorial representation for PDA, including shortcuts for 'push A' (regardless the topmost symbol) and for 'read a'.

(*Fig. 10*) Three different automata for a single language: final state, empty stack, and finally (on an-

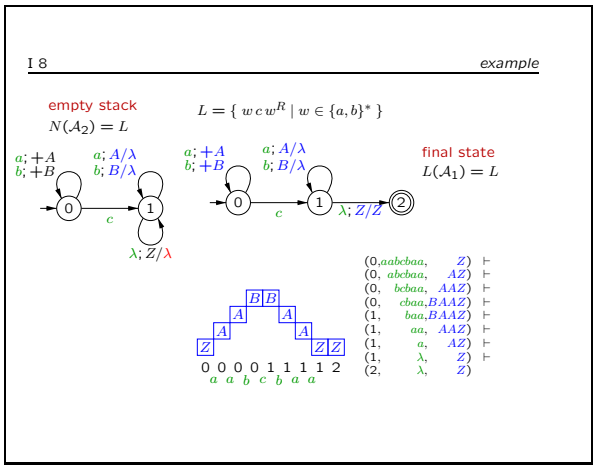


Figure 10: example

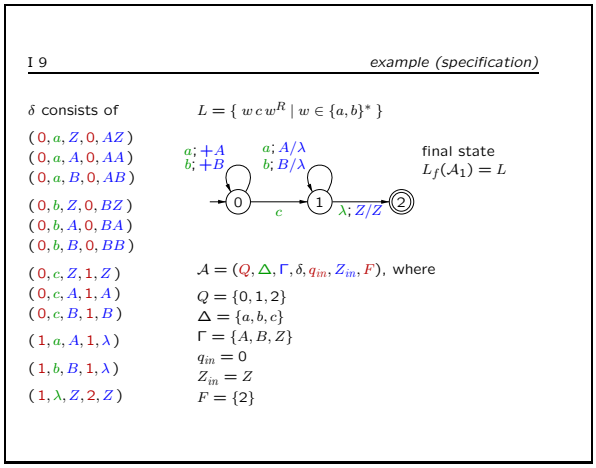


Figure 11: example (specification)

other slide) empty stack with only a single state.

(*Fig. 11*) The number of arrows in the diagram does not equal the number of transitions, due to the shortcuts we have introduced. This is notationally convenient but we should always avoid the introduction of shortcuts that are not supported by the original formal model.

(*Fig. 13*) A challenge: the coin exchange language. Can you construct a CFG for it, or (equivalently) a single state PDA?

(*Fig. 14*) We note some peculiar properties of the model.

Input is only accepted if it is completely read. A

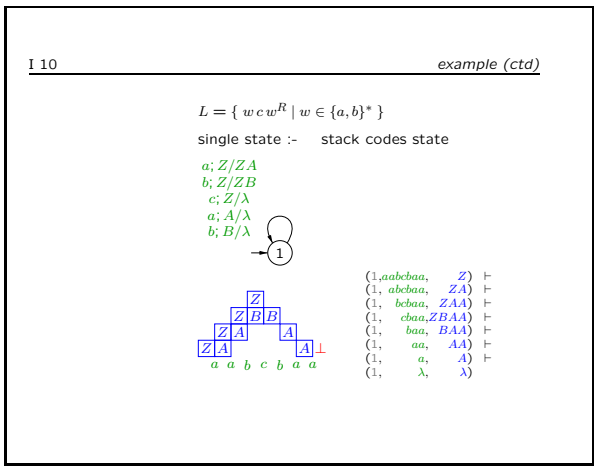


Figure 12: example (ctd)

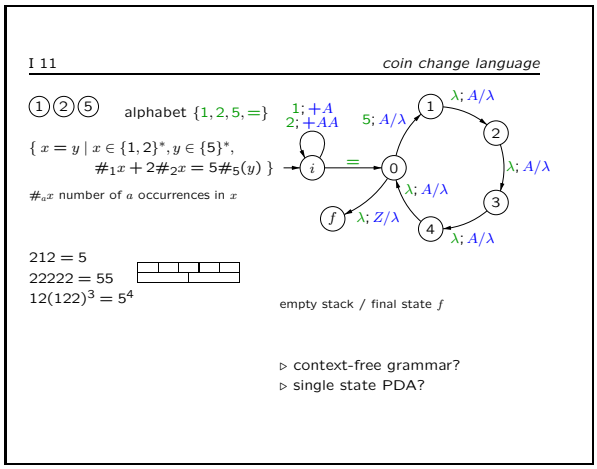


Figure 13: coin change language

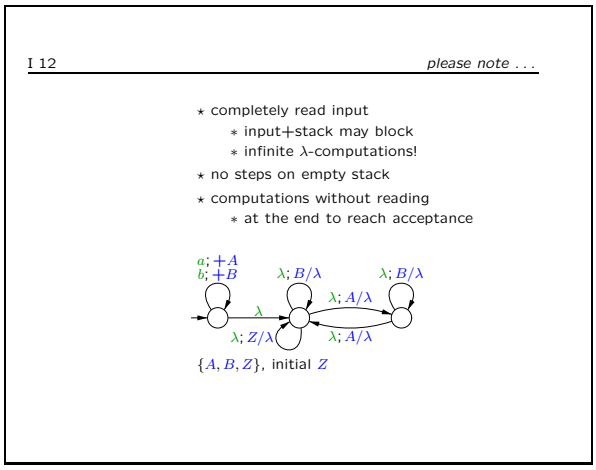


Figure 14: please note ...

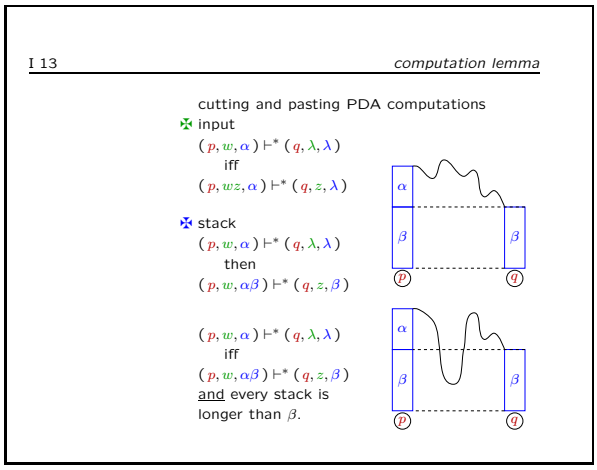


Figure 15: computation lemma

computation does not lead to acceptance when it blocks, i.e., there is no applicable transition, or when it is stuck in an infinite computation reading λ.

Every transition pops a stack symbol, hence no steps on the empty stack are allowed. The automaton is blocked (this is important when we deal with determinism).

Even after all symbols have been read the PDA can continue computing (and reach acceptance only several steps later). We show a very simple example that makes all its computations after the input has been read. What is its empty stack language?

(*Fig. 15*) Context-free grammars are 'context-free', its productions can be applied independently of the context: $A \Rightarrow^* \alpha$ iff $\beta_1 A \beta_2 \Rightarrow^* \beta_1 \alpha \beta_2$ (in context β_1, β_2). Moreover we can glue derivations: if $A \Rightarrow^* \alpha_1 B \alpha_2$, and $B \Rightarrow^* \beta$, then $A \Rightarrow^* \alpha_1 \beta \alpha_2$.

For PDA we have similar ways to combine computations, both involving input and stack. However, when considering the stack we have to be careful and avoid the empty stack.

2 Pushdown Automata and Context-Free Languages

(*Fig. 16*) We study the families of languages accepted by pushdown automata.

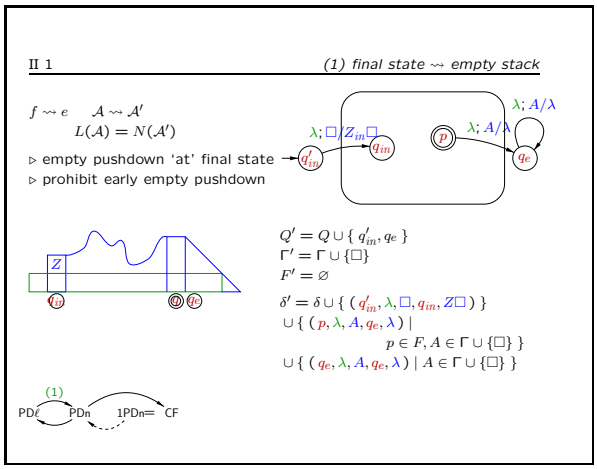


Figure 16: (1) final state ⇝ empty stack

First we show that the two ways of accepting are equivalent. Given an automaton and one acceptance type it is possible to construct another, equivalent, automaton with the other acceptance type (1,2).

Then we argue that single state PDA are actually more or less the same as context-free grammars (3).

The main result is the equivalence of PDA and CFG (4).

As an 'application' we look at closure properties of CF and prove them using PDA (and finite state transductions).

(*Fig. 18*) Any CFG can be simulated by a single state PDA using the expand & match technique. In case we start with a CFG satisfying a mild nor-

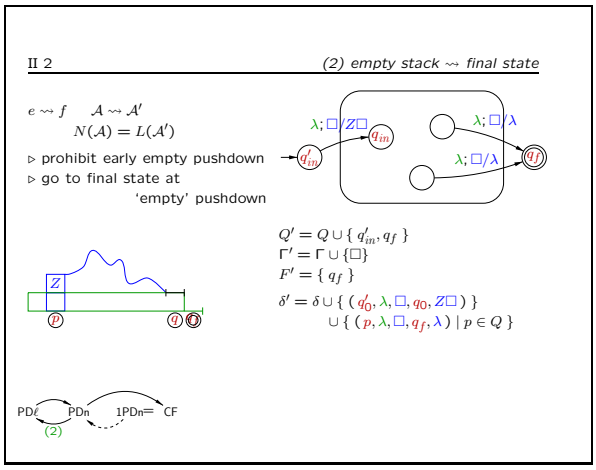


Figure 17: (2) empty stack ⇝ final state

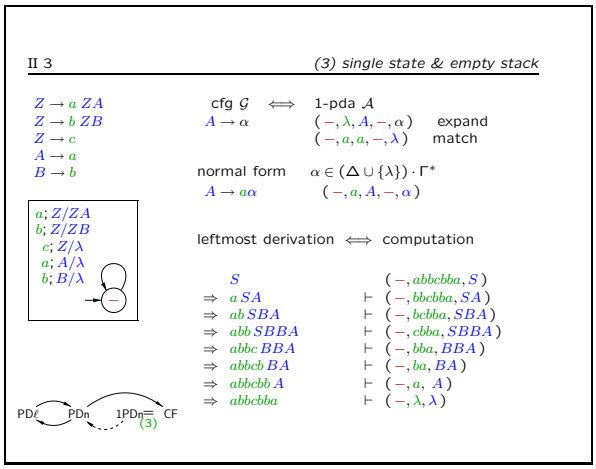


Figure 18: (3) single state & empty stack

mal form property, the two steps can be combined into one, and we obtain a one-to-one correspondence between productions and transitions.

This correspondence also works backwards: single state PDA are just like grammars. Their computations are like the leftmost derivations of the corresponding CFG.

Hence, to prove the equivalence of PDA with CFG the problem is to deal with the states of the PDA.

(*Fig. 19*) The basic theorem of context-free languages: the equivalence of CFG and PDA.

It is due to Chomsky ('Context Free Grammars and Pushdown Storage'), Evey ('Application of Pushdown Store Machines'), and Schützenberger

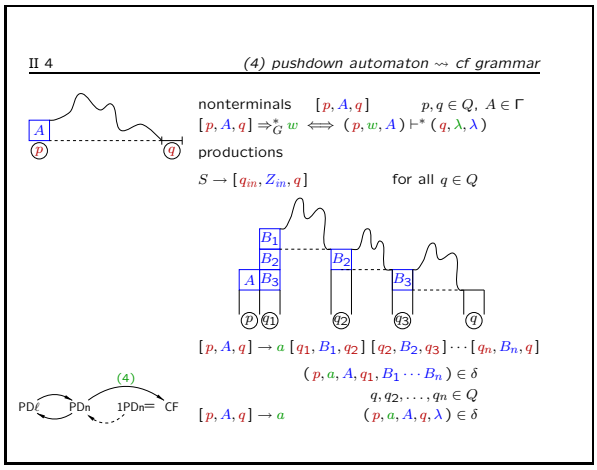


Figure 19: (4) pushdown automaton ⇝ cf grammar

('On Context Free Languages and Pushdown Automata') in 1962/3.

Starting with a pda under empty stack acceptance we construct an equivalent cfg. Its nonterminals are triplets $[p, A, q]$ representing computations of the pda. Productions result from recursively breaking down computations. A single instruction yields many productions, mainly because intermediate states of the computations have to be guessed.

(*Fig. 20*) The previous slide 'almost' is the proof for the equivalence of the PDA $\mathcal{A}$ and constructed CFG G. Use induction on the length of the derivation/ computation to prove
$[p, A, q] \Rightarrow_G^* w \iff (p, w, A) \vdash_{\mathcal{A}}^* (q, \lambda, \lambda)$
$\Leftarrow$: Assume $(p, w, A) \vdash^* (q, \lambda, \lambda)$. Consider the first instruction ι used by $\mathcal{A}$.

$*$ $\iota = (p, a, A, p', \lambda)$. This empties the stack; hence we have a single step computation, $(p, a, A) \vdash (p', \lambda, \lambda)$, so $p' = q$, $w = a$. By construction there exists the rule $[p, A, p'] \to a$. This yields the derivation $[p, A, q] \Rightarrow_G w = a$ as required.

$*$ $\iota = (p, a, A, q_1, B_1 \ldots B_n)$, $n \geq 1$. The computation starts like $(p, w, A) \vdash (q_1, w', B_1 \ldots B_n) \vdash^* (q, \lambda, \lambda)$, $w = aw'$. These B_i must be popped from the stack. We can split the computation
$(q_1, w_1 w_2 \ldots w_n, B_1 \ldots B_n) \vdash^*$
$(q_2, w_2 \ldots w_n, B_2 \ldots B_n) \vdash^* (q_n, w_n, B_n) \vdash^*$
$(q_{n+1}, \lambda, \lambda)$, such that q_i is the first position where B_i appears as top of stack, $q_{n+1} = q$, and and $w' = w_1 w_2 \ldots w_n$.

By the Computation Lemma [previous section], we can remove common parts of the input and stack at the start and end of a computation. We obtain separate computations $(q_i, w_i, B_i) \vdash^* (q_{i+1}, \lambda, \lambda)$.

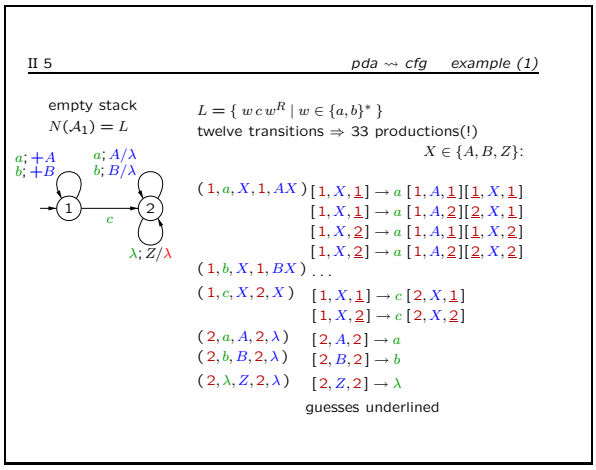


Figure 20: pda ⇝ cfg example (1)

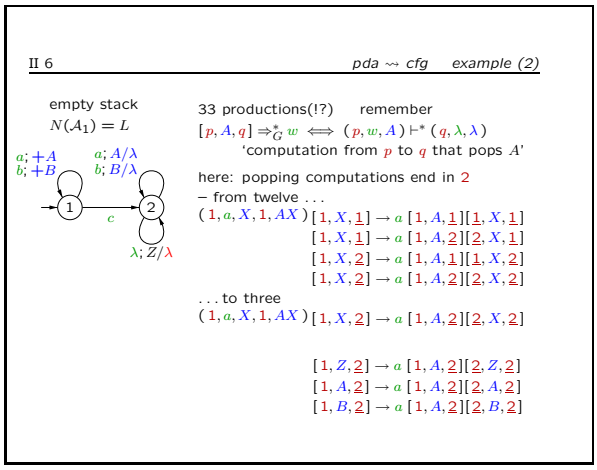


Figure 21: pda ⇝ cfg example (2)

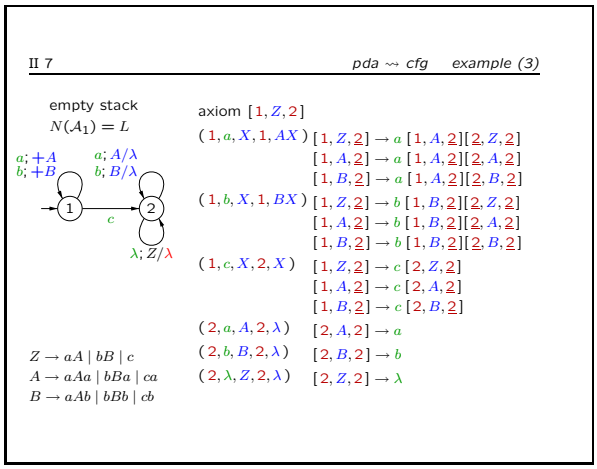


Figure 22: pda ⇝ cfg example (3)

As these are shorter than the original computation (even when $n = 1$ we separated the first step) we know that $[q_i, B_i, q_{i+1}] \Rightarrow_G^* w_i$.

Given ι we know G has the rule $[p, A, q] \to a[q_1, B_1, q_2][q_1, B_1, q_2] \dots [q_n, B_n, q_{n+1}]$, which can be combined with the computations we found $[p, A, q] \Rightarrow a[q_1, B_1, q_2][q_1, B_1, q_2] \dots [q_n, B_n, q_{n+1}] \Rightarrow^* aw_1w_2 \dots w_n = aw' = w$. As $q_{n+1} = q$ this is as required.

$\Rightarrow$: For the other direction use a similar technique. Given a derivation in G consider its first production rule, use induction on the remaining subtrees of the derivation to find computations, and join the pieces into a full computation.

(*Fig. 20*) We apply the construction from pda to cfg to one of our earlier examples. This leads to a huge grammar.

One can reduce such grammars in the standard way, by pruning symbols that are not productive, i.e., do not produce terminals, or are not reachable from the axiom. Alternatively one can be clever in advance, eliminating symbols that correspond to computations of the pda that do not exist. (Perhaps the example will clarify this a little.)

(*Fig. 21*) There are algorithms that will reduce a CFG, but here we directly use the 'meaning' of the triples $[p, A, q]$: they represent computations from state p to state q finally popping the stack that started with A. If we look carefully at the PDA we can see several cases for which such computations do not exist. We can delete the corresponding symbols $[p, A, q]$ and all productions containing them.

(*Fig. 23*) There are two natural ways to turn the finite state automaton into a PDA. We can ignore the stack, using a single stack symbol that is replaced each step, under final state acceptance. Alternatively we can store the state on the stack, and get a single state PDA, under empty store acceptance, popping the final state as last move.

(*Fig. 24*) We have shown that CFG and PDA are equivalent. There are also restricted forms of both that are still equivalent. One example: linear grammars vs. one-turn PDA. The standard construction applied to a one-turn PDA does not always yield a linear grammar.

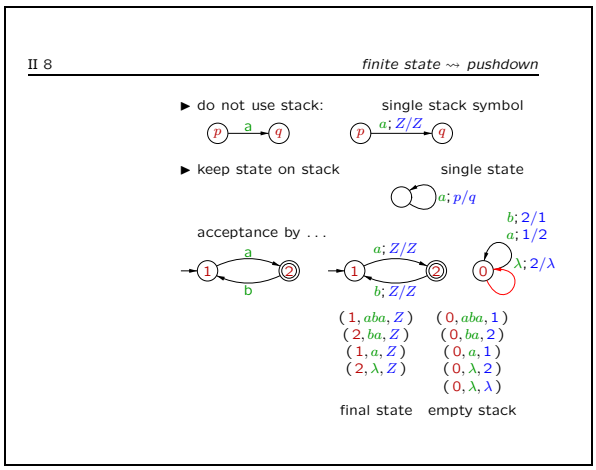


Figure 23: finite state ⇝ pushdown

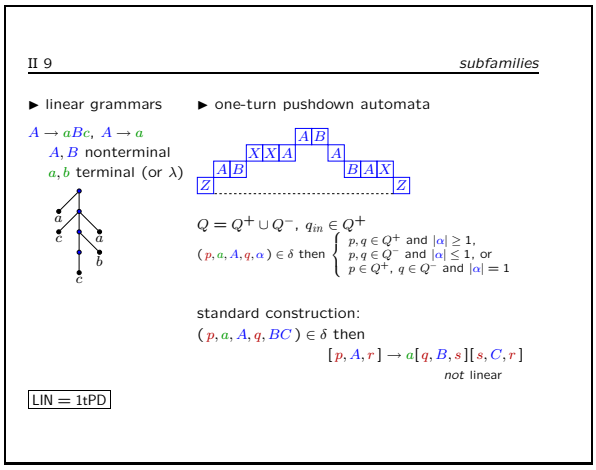


Figure 24: subfamilies

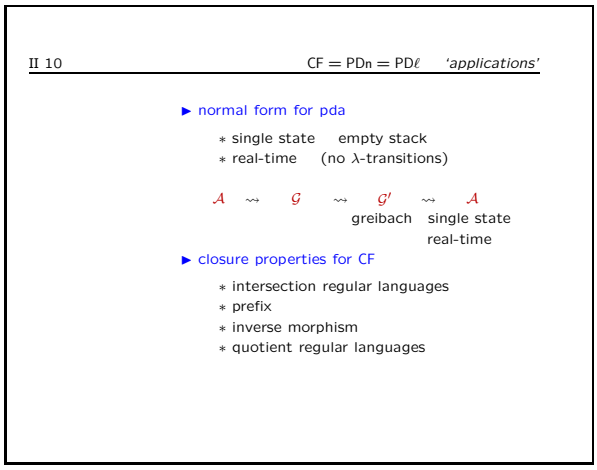


Figure 25: CF = PDn = PDℓ 'applications'

(*Fig. 25*) We can use the equivalence between CFG and PDA in two ways.

First we may transfer the Greibach normal form to PDA: every PDA can be transformed into an equivalent PDA that is real-time: it reads a symbol in every step. As it results from a grammar, it additionally has only a single state.

Second we may use the PDA to show closure properties for the context-free languages. For some operations this seems to give intuitively simple constructions. (Depending on your taste.)

(*Fig. 26*) In the Tarragona course context-free grammars are presented by prof. Kudlek. Two of the closure results for CF he proves are closure under intersection with regular languages (triplet construction) and closure under inverse morphisms. The latter is shown using other closure properties and arguments from AFL theory.

We will use PDA to obtain these and other closure properties. First we introduce finite state transducers, and we argue that many operations can be performed by these FST. Then we show that CF is closed under all finite state transductions by a direct product construction of PDA and FST.

(*Fig. 30*) A pictorial representation of the direct product construction of a PDA and a FST, showing the image of a PDA language under a transduction is again accepted by a PDA. This proves closure of CF under several operations.

Same construction is given on another transparency, but now in a clear specification. No formal

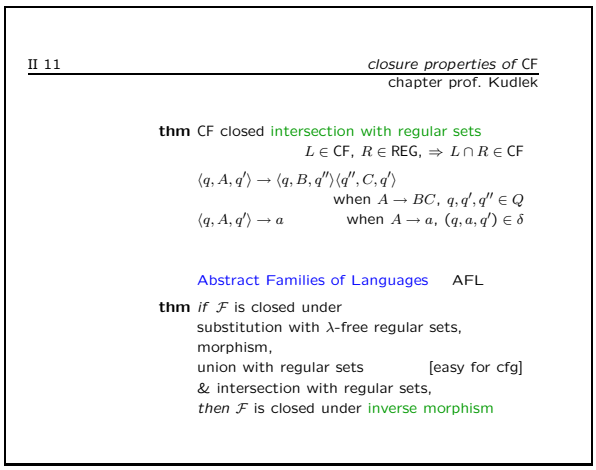


Figure 26: closure properties of CF

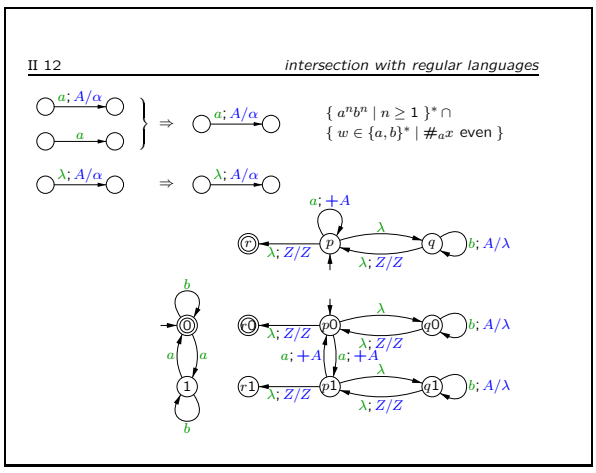


Figure 27: intersection with regular languages

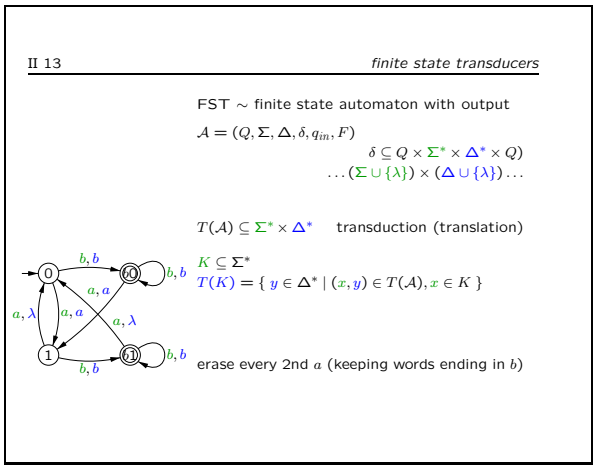


Figure 28: finite state transducers

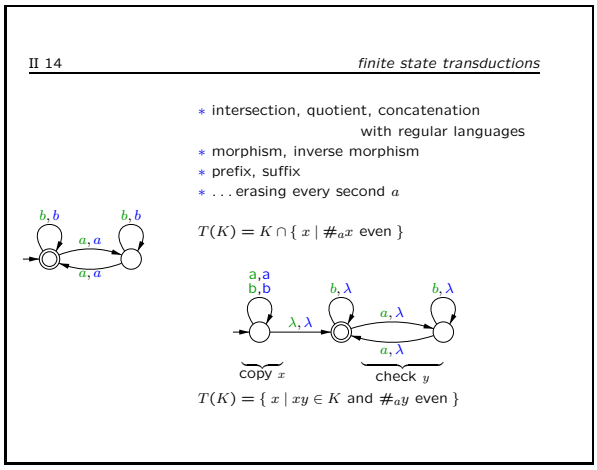


Figure 29: finite state transductions

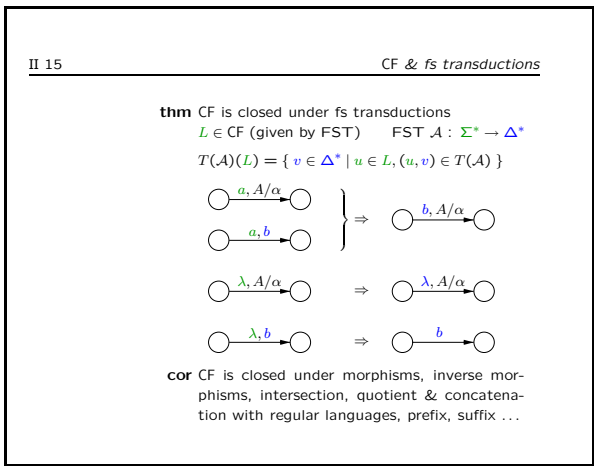


Figure 30: CF & fs transductions

proof (induction on computations) is given.

(*Fig. 32*) As an example of finite state transducers and the closure construction: the inverse morphism.

For a morphism h we construct a FST that realizes h^{-1}. Then for the context-free language $K = \{(100)^n(10)^n \mid n \geq 0\}$ we construct PDA for K and $h^{-1}(K)$.

(*Fig. 33*) As promised, the CF languages are closed under right quotient with regular languages, since for every regular language R we can transform the FSA for R into a FST that performs the quotient by R as its function.

The next slide implements this construction. Given

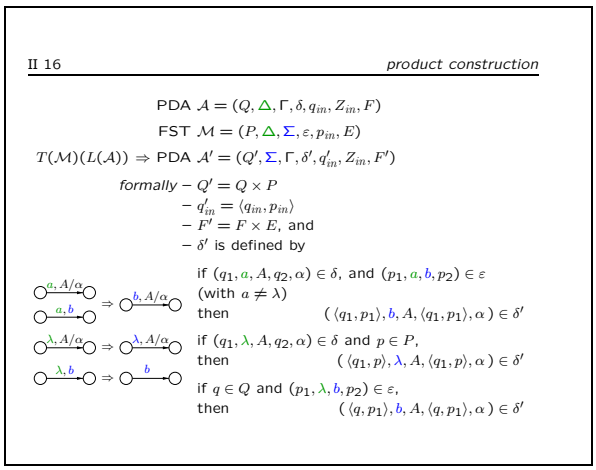


Figure 31: product construction

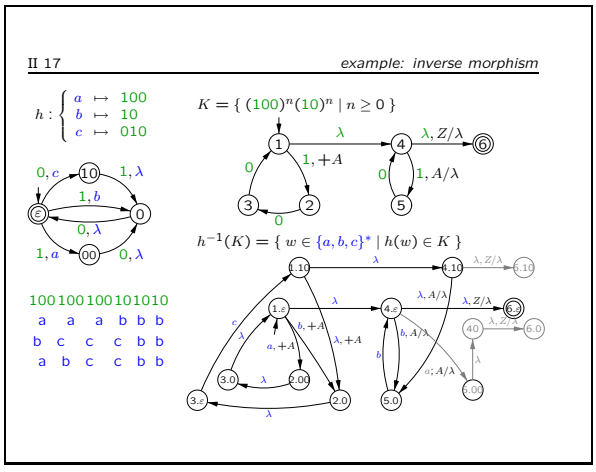


Figure 32: example: inverse morphism

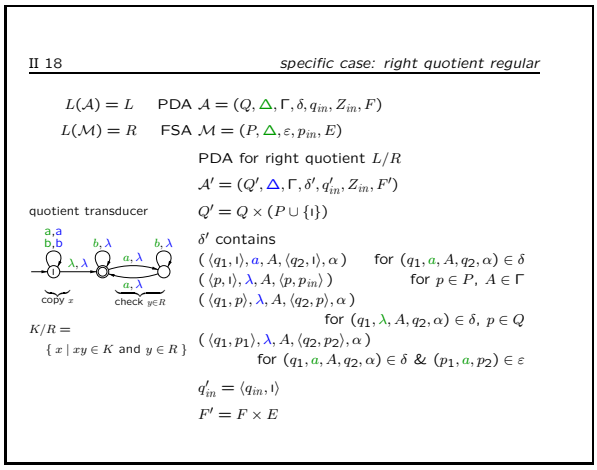


Figure 33: specific case: right quotient regular

a PDA $\mathcal{A}$ and a FSA $\mathcal{M}$ it directly constructs the PDA for the quotient of the languages. It uses the general format for transductions from previous slides, as if the transducer for the quotient had been given. In fact, is has been implicitly derived from the FSA, by adding a single state ı, see sketch to the left.

3 Determinism

(*Fig. 34*) For finite state automata determinism is simple: it is a normal form, equivalent to the general model. Every nondet FSA can be made deterministic by the subset construction, perhaps at the cost of an exponentially larger state set.

We are not that lucky for PDA. First both language definitions (final state and empty stack) are no longer equivalent. Final state acceptance is stronger than empty state acceptance. Unfortunately even for this acceptance PDA are no longer equivalent to CFG. There is a simple intuitive example that makes this clear. A full formal proof however takes some more hard work.

(*Fig. 35*) Determinism means the automaton has no choice: at each moment it can take at most one step to continue its computation. To translate this intuition to a restriction on the instructions for PDA is nontrivial, as the next step is determined both by input letter and by topmost stack symbol. Additionally this is complicated by the choice

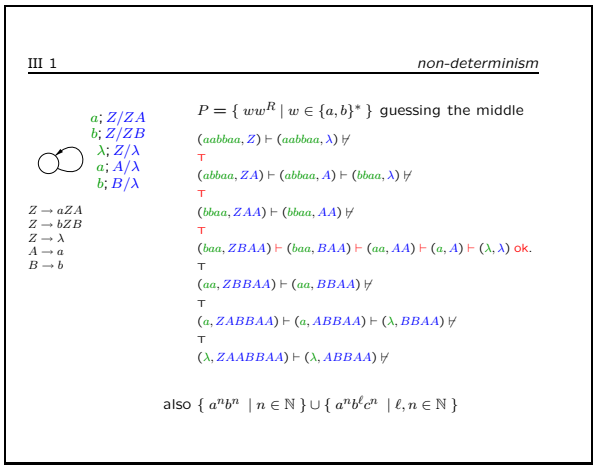


Figure 34: non-determinism

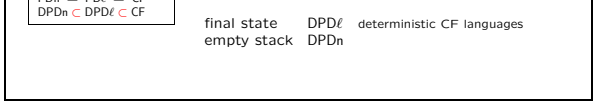


Figure 35: definition

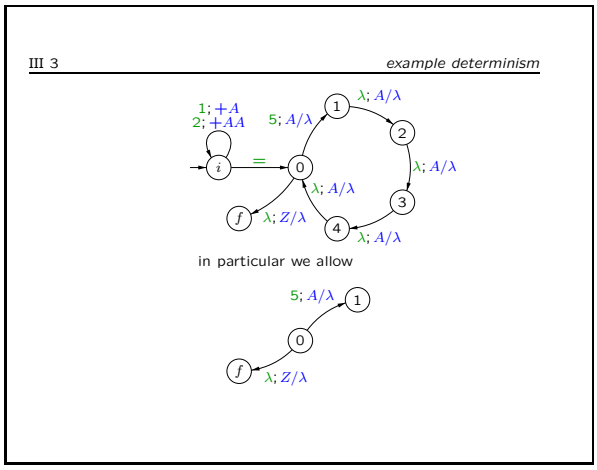


Figure 36: example determinism

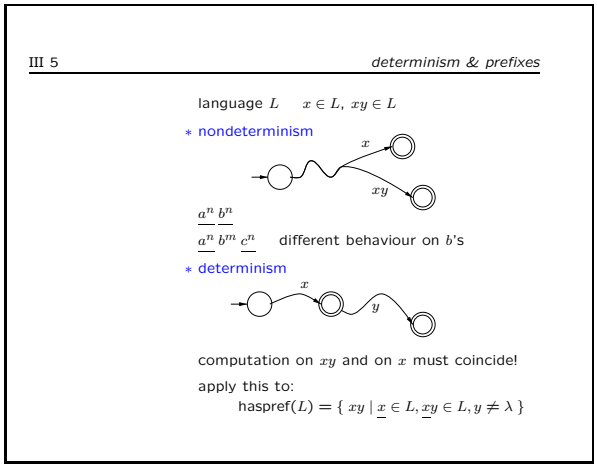


Figure 37: determinism & prefixes

between reading an input letter and following a λ-instruction.

We quote from our chapter:

The PDA $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_{in}, A_{in}, F)$ is *deterministic* if

- for each $p \in Q$, each $a \in \Delta$, and each $A \in \Gamma$, δ does not contain both an instruction $(p, \lambda, A, q, \alpha)$ and an instruction (p, a, A, q', α').
- for each $p \in Q$, each $a \in \Delta \cup \{\lambda\}$, and each $A \in \Gamma$, there is at most one instruction (p, a, A, q, α) in δ.

(*Fig. 37*) Consider a language that both includes string x and an extension xy of it. Nondeterministic automata may have quite different accepting computations on both strings. For deterministic automata we know that the computation that accepts xy must start with the accepting computation on x.

(*Fig. 38*) In the nondeterministic case the two types of acceptance for PDA (empty stack, final state) were shown to be equivalent. This is no longer true in the deterministic case; only one of the inclusion constructions carries over. Empty stack acceptance is particularly weak: it cannot accept a word together with one of its prefixes. Hence it cannot accept even a^*.

(*Fig. 39*) In order to rigorously show that DPD$\ell \subset$

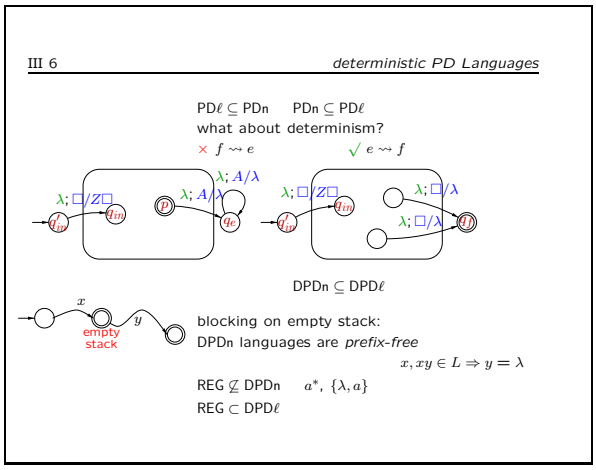


Figure 38: deterministic PD Languages

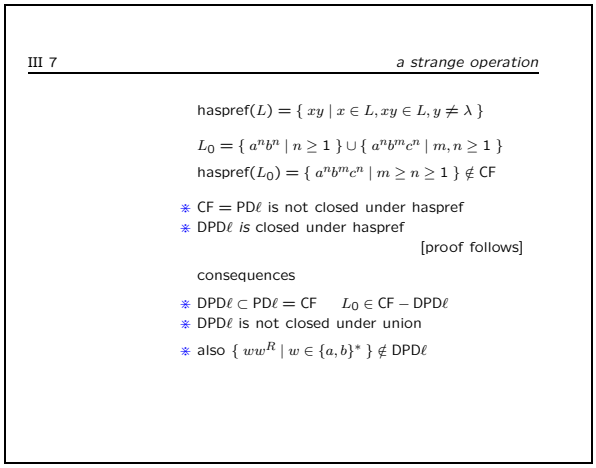


Figure 39: a strange operation

PDℓ = CF we define a 'strange operation' haspref. We show that DPDℓ and CF behave differently with respect to this operator. See properties on the slide.

(*Fig. 40*) The rigorous proof that DPDℓ is closed under haspref. Well, the construction and the intuition behind it.

The construction is illustrated on the next transparency.

(*Fig. 42*) Some ramifications.

What we know: DPDn ⊂ DPDℓ, and languages in DPDn are prefix-free.

Now we observe that for prefix-free languages the two families are equal.

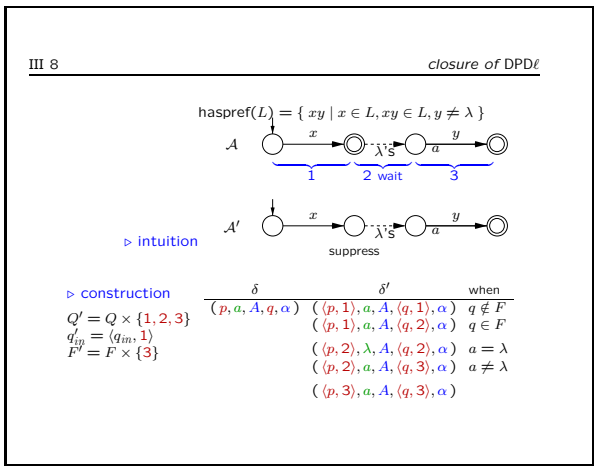


Figure 40: closure of DPDℓ

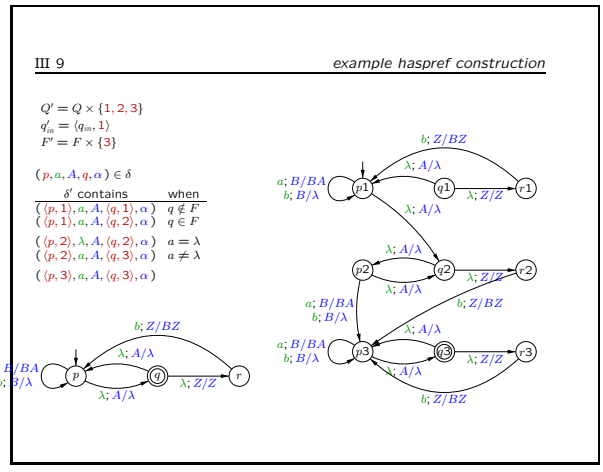


Figure 41: example haspref construction

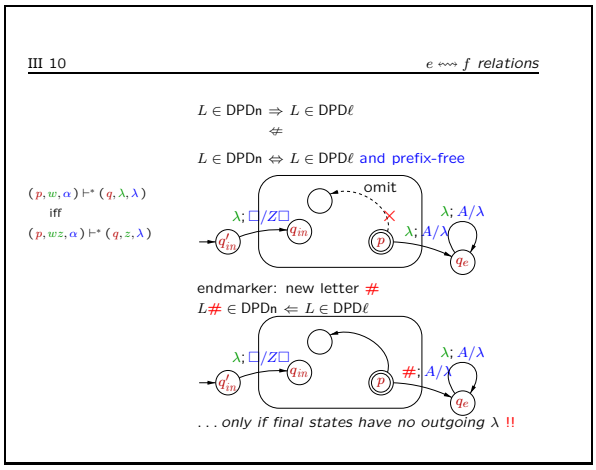


Figure 42: $e \leftrightsquigarrow f$ relations

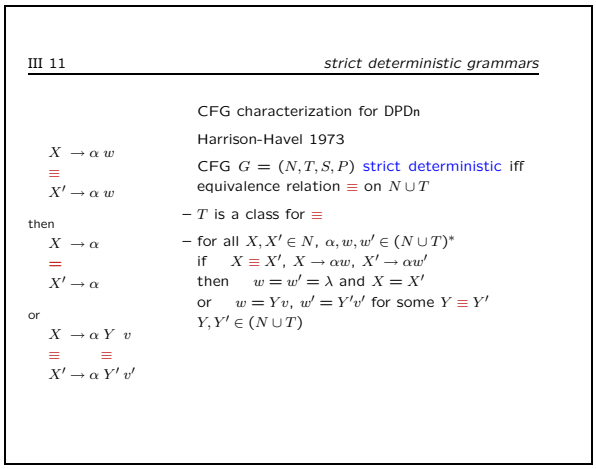


Figure 43: strict deterministic grammars

Choosing a new letter #, appending # to a language L one obtains a prefix-free language $L\#$. Do we have $L \in$ DPD$\ell \Rightarrow L\# \in$ DPDn? Yes, if we may assume that final states have no outgoing λ-instructions.

Indeed, we cán assume this. How can we obtain that nornal form?

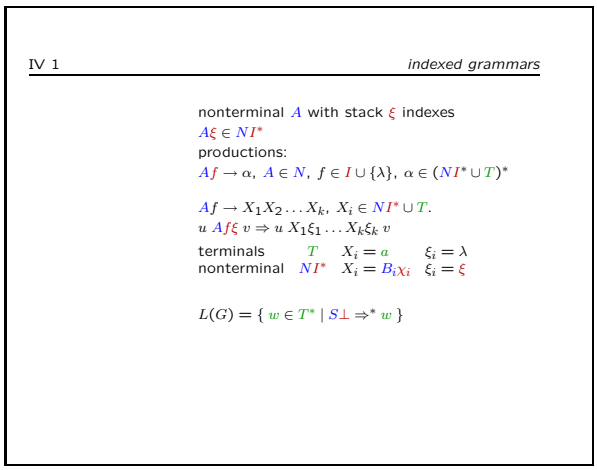


Figure 44: indexed grammars

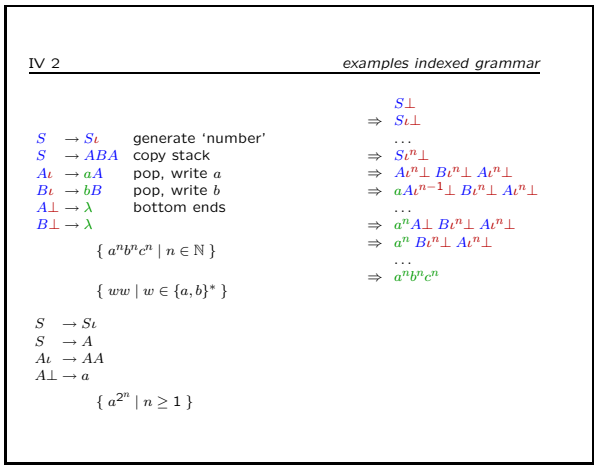


Figure 45: examples indexed grammar

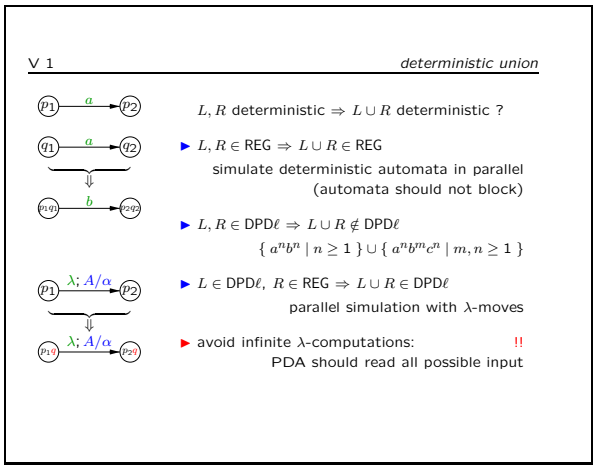


Figure 46: deterministic union

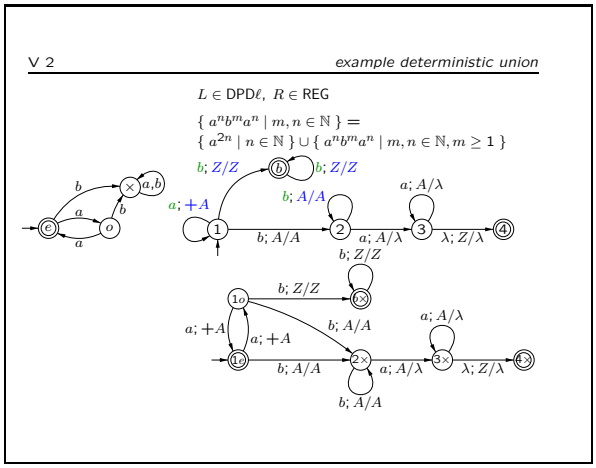


Figure 47: example deterministic union

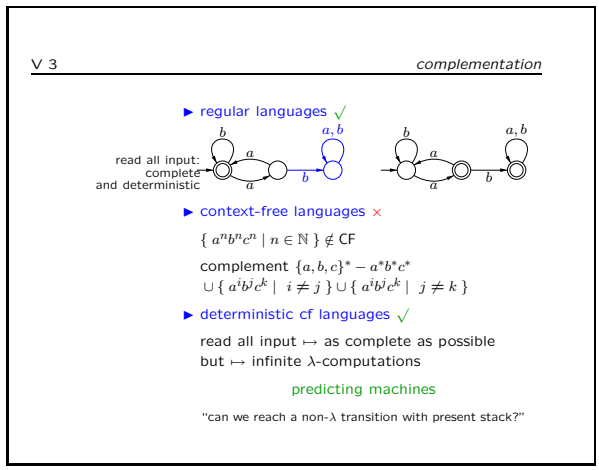


Figure 48: complementation

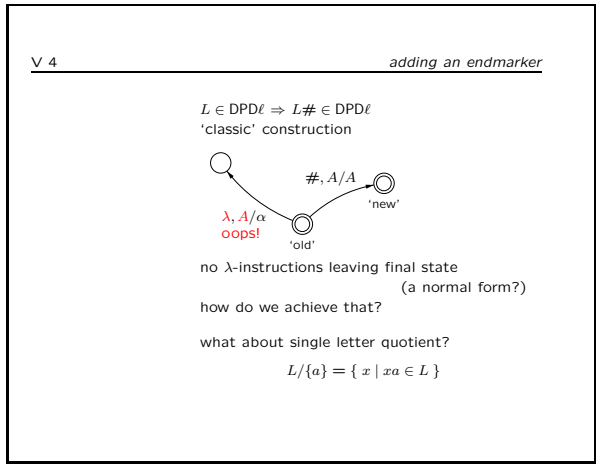


Figure 49: adding an endmarker

4 Indexed Grammars, Stack Automata

5 Closure and Determinism Stack Languages and Predicting Machines

(*Fig. 50*) We study the language of stacks during computations of a PDA. This language is regular! The proof is a simple consequence of the $[p, A, q]$-construction! Exclamation mark!

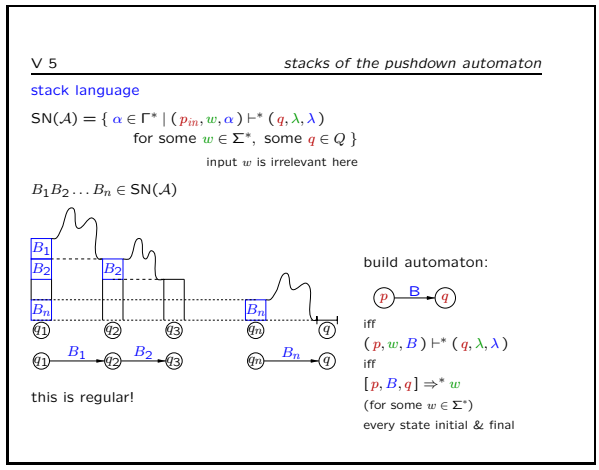
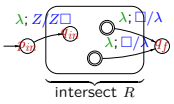


Figure 50: stacks of the pushdown automaton

V 6 *stack language variants*



$SN(\mathcal{A}) = \{\ \alpha \in \Gamma^* \mid (p_{in}, w, \alpha) \vdash^* (q, \lambda, \lambda)$
for some $w \in \Sigma^*$, some $q \in Q\ \}$

variant [also regular]

$\{\ \ldots \mid \quad \ldots$ for some $w \in R$, some $q \in F\ \}$

$SF(\mathcal{A}) = \{\ \alpha \in \Gamma^* \mid (p_{in}, w, Z_{in}) \vdash^* (q, \lambda, \alpha)$
for some $w \in \Sigma^*$, some $q \in F\ \}$

Figure 51: stack language variants

V 9 *predicting machines*

- stack language $SN(\mathcal{A})$ is regular
- a (deterministic) PDA can keep regular info on its stack

$\Rightarrow$ a (deterministic) PDA can predict future behaviour using present stack by inspecting top

predicting machines

"reaches $\mathcal{A}$ the empty stack from *my* stack?"

B	yes	no	yes
A	yes	yes	no
B	no	yes	yes
A	no	no	yes
A	yes	no	no
	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$

Figure 54: predicting machines

V 7 *application Buchi*

$SN(\mathcal{A}) = \{\ \alpha \in \Gamma^* \mid (p_{in}, w, \alpha) \vdash^* (q, \lambda, \lambda)$
for some $w \in \Sigma^*$, some $q \in F\ \}$

Buchi: regular canonical systems
type-0 productions $\alpha \to \beta$
prefix rewriting $\alpha\,| \Rightarrow \beta\,|$

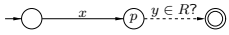
$L(\text{rcs}) = \{\ w \mid w \Rightarrow^* \lambda\ \}$

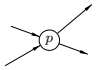
rcs defines regular language
simulate prefix $\alpha \to \beta$ by PDA [use F]

Figure 52: application Buchi

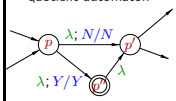
V 10 *application: quotient*

DPDℓ closed quotient with REG





quotient automaton



Y: $\alpha \in SN(\mathcal{A}_{p,R})$
N: $\alpha \notin SN(\mathcal{A}_{p,R})$

can we accept extending with $y \in R$?
stack α satisfies:
$(p, y, \alpha) \vdash^*_{\mathcal{A}} (q, \lambda, \beta)$, $q \in F$, $y \in R$, some β

$SN(\mathcal{A}_{p,R}) = \{\ \alpha \in \Gamma^* \mid (p, w, \alpha) \vdash^*_{\mathcal{A}_{p,R}} (q, \lambda, \lambda)$
for some $w \in \Sigma^*\ \}$

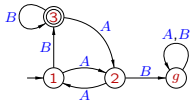
$\mathcal{A}_{p,R}$ constructed from $\mathcal{A}$
– initial state p
– intersection with R (product construction)
– change to empty stack acceptance

Figure 55: application: quotient

V 8 *regular properties of the stack*

stack belongs to regular language R?

e.g. $R = B(AA + B)^*$
deterministic automaton for *reverse*



update stack
B/AB
$\langle B,1\rangle/\langle A,1\rangle\langle B,2\rangle$
$\langle B,2\rangle/\langle A,2\rangle\langle B,1\rangle$
$\langle B,3\rangle/\langle A,3\rangle\langle B,2\rangle$
$\langle B,g\rangle/\langle A,g\rangle\langle B,g\rangle$
success $\in R$
$\langle B,1\rangle$, $\langle B,3\rangle$ on top

add state info to stack

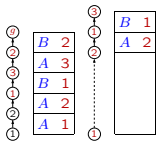


Figure 53: regular properties of the stack

V 11 *application: complement*

as promised:

- deterministic cf languages $\surd$

read all input $\mapsto$ as complete as possible
but $\mapsto$ infinite λ-computations

predicting machines

"can we reach a non-λ transition with present stack?"

Figure 56: application: complement

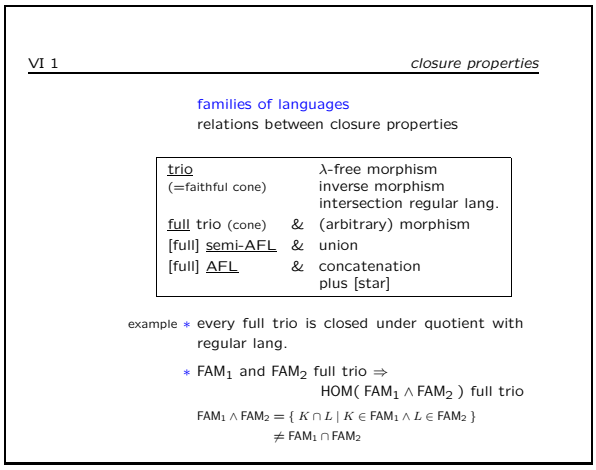


Figure 57: closure properties

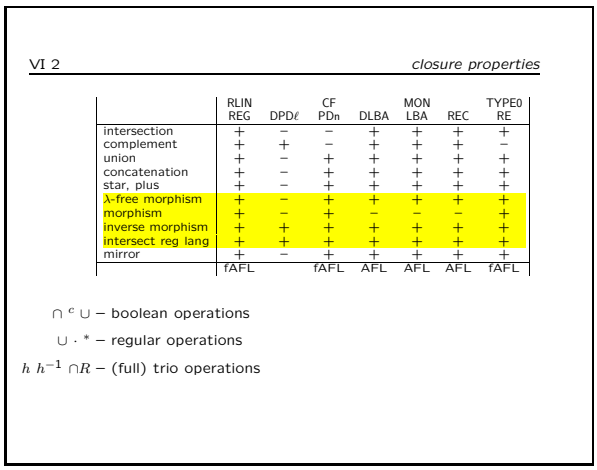


Figure 58: closure properties

6 Pushdown as Storage Abstract Families of Automata

(*Fig. 57*) AFL theory ('abstract family of languages') is all about language operations, and closure properties of language families.

In an earlier lecture we have seen that a direct product construction with PDA can be used to show that CF is closed under the FST's. Many known operations are in fact FST's. We return to finite state transductions in this lecture.

(*Fig. 57*) Basic closure properties for the families in the Chomsky Hierarchy, and deterministic variants. We distinguish three natural sets of operations: boolean, regular, and trio operations.

(*Fig. 61*) Thus we have an alternative formalization of the notion of (full) trio as family closed under arbitrary/λ-free FTS's, rather than in terms of the basic trio operations.

(*Fig. 65*) Here another example of a similar AFA result. We consider blind multi-counter automata. Each transition has an associated k-dimensional integer vector, and a computation is accepted if it runs from initial state to final state, while additonally the vectors of the transitions in the computation add up to the zero-vector.

The automata are 'blind' as they cannot perform a zero-test during the computation. There is only one test at the end of the computation.

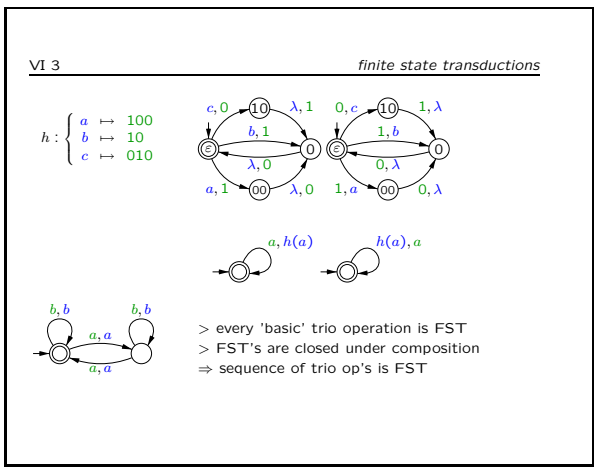


Figure 59: finite state transductions

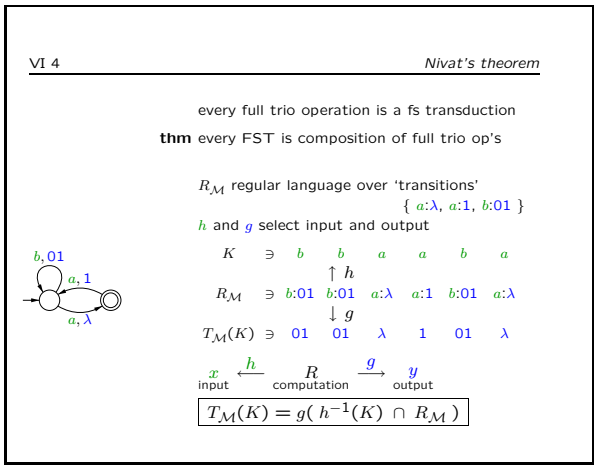


Figure 60: Nivat's theorem

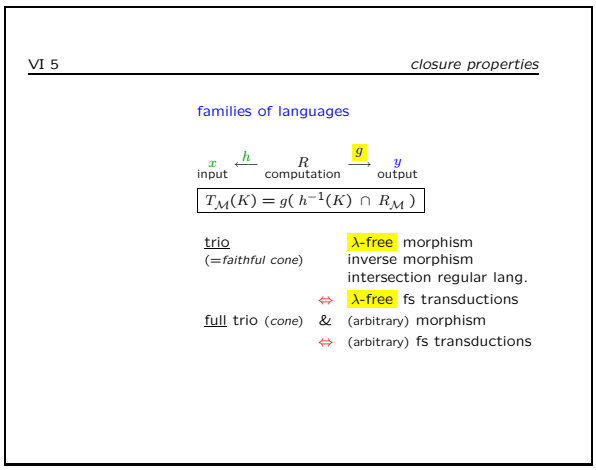


Figure 61: closure properties

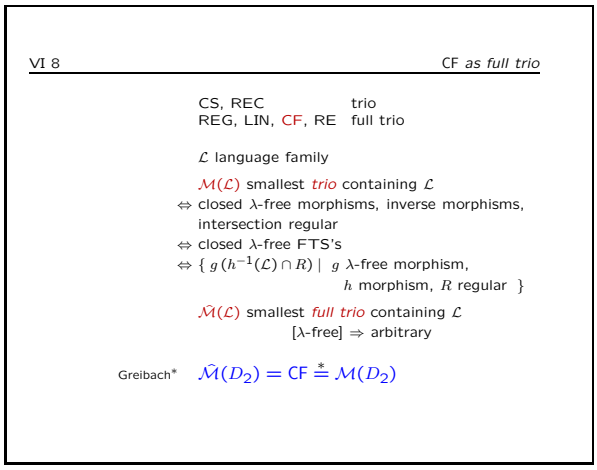


Figure 64: CF as full trio

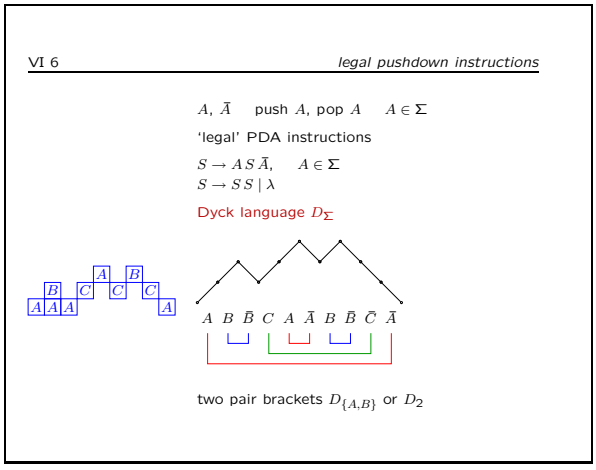


Figure 62: legal pushdown instructions

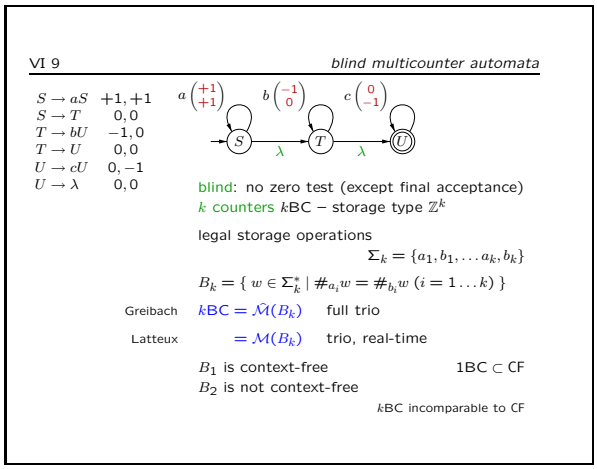


Figure 65: blind multicounter automata

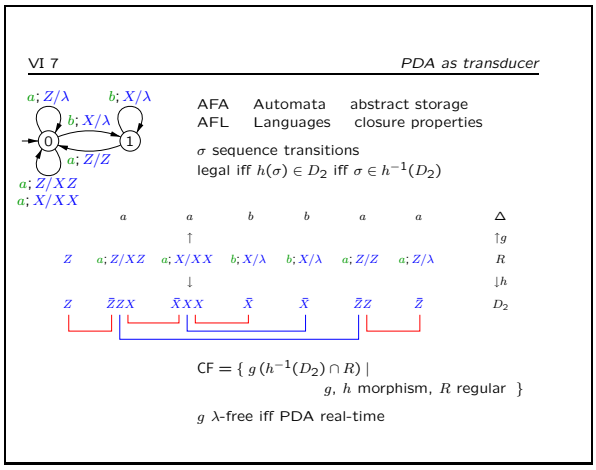


Figure 63: PDA as transducer

We give the language B_k associated to this storage type. As for PDA these automata accept the smallest full trio containing B_k. By a result of Latteux (algebraic, and more general) this equals the smallest trio containing B_k. Translating this back to the automata, we obtain a real-time normal form (no λ-transitions).

(*Fig. 67*) We present some counter-like families, each family with the associated language(s) of legal storage operations.

A blind counter has as native storage $\mathbb{Z}$, can add and substract one from the counter, but has no zero-test. There is only the final test on the storage as each computation leads from 0 to 0.

A *partially* blind counter is exactly like the blind

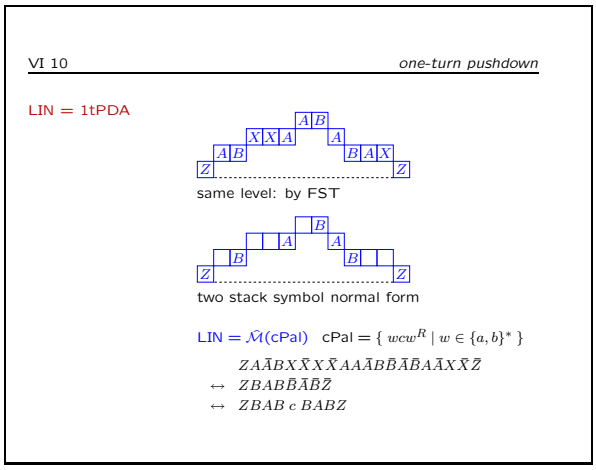


Figure 66: one-turn pushdown

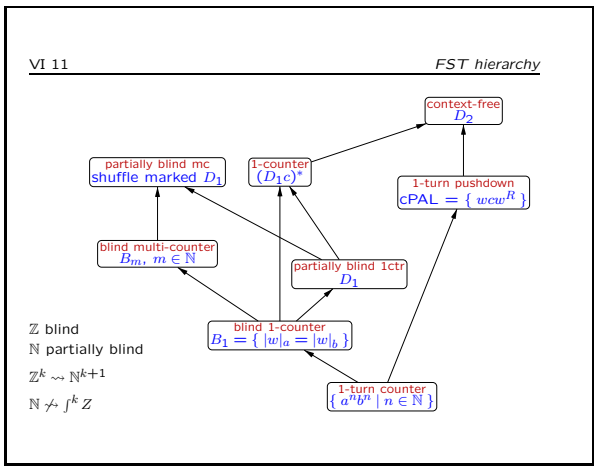


Figure 67: FST hierarchy

counter, except that the native storage is $\mathbb{N}$. This means that the computation blocks (halts unsuccessfully) if the automaton tries to decrease the counter when zero. This is like a half test on the sign of the counter.

We can also consider k (partially) blind counters with storage $\mathbb{N}^k$ or $\mathbb{Z}^k$. Quite surprisingly k blind counters can be simulated by $k+1$ partially blind counters, while even a single partially blind counter cannot be simulated by blind multi-counters.

Why is the slide called 'FST hierarchy'? The family of 1-turn counter languages is included in the blind 1-counter languages because the exists a FST that outputs $\{a^n b^n \mid n \in \mathbb{N}\}$ on input B_1, and similarly for all other inclusions in the diagram.

As inspiration for the diagram I used the Habilitations Thesis of K. Reinhardt, which contains many more families. I hope I got all the arrows right ...

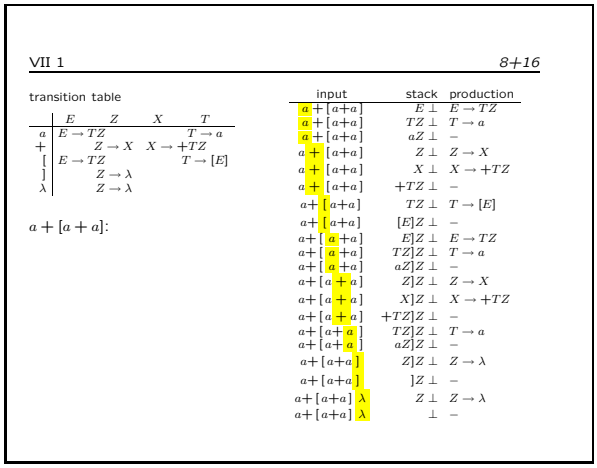


Figure 68: 8+16

7 Basic Parsing Building sPDA for Grammars

8 Famous Automata (Examples / Exercises)

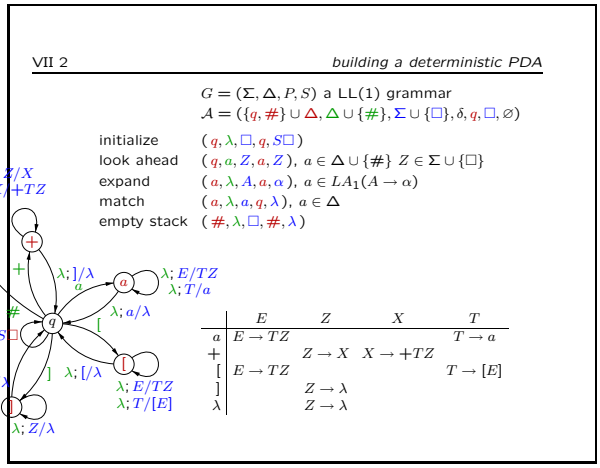


Figure 69: building a deterministic PDA

(*Fig. 71*) We convert the PDA for palindromes to a CFG for the same language. We start by noting

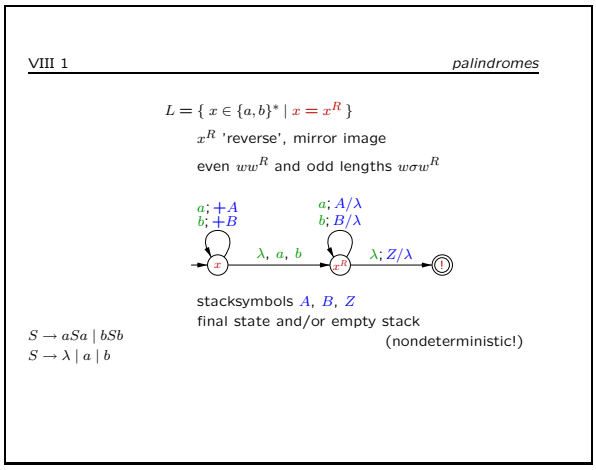


Figure 70: palindromes

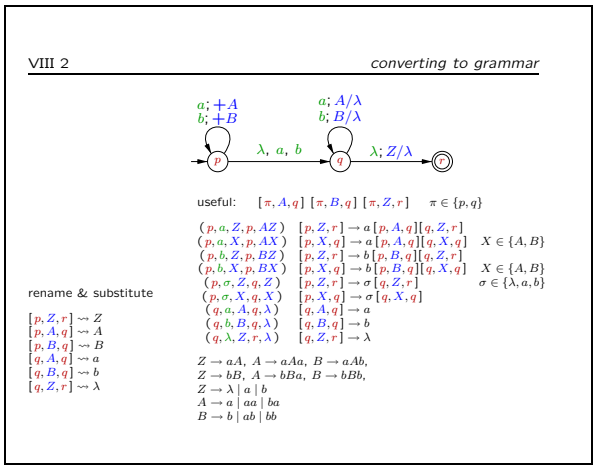


Figure 71: converting to grammar

that not all triples $[p, X, q]$ are useful in the grammar, but only those that represent existing computations starting in state p with X on the stack end ending in q with empty stack. Here computations starting in p with A can only end in q, etc. We can drop the other triples from the set of nonterminals, as well as the productions introducing those symbols.

(*Fig.* **??**) The pictures in these transparencies were put into LaTeX using GasTeX: *Graphs and Automata Simplified in TeX* by Paul Gastin

`http://www.lsv.ens-cachan.fr/~gastin/gastex/gastex.html`

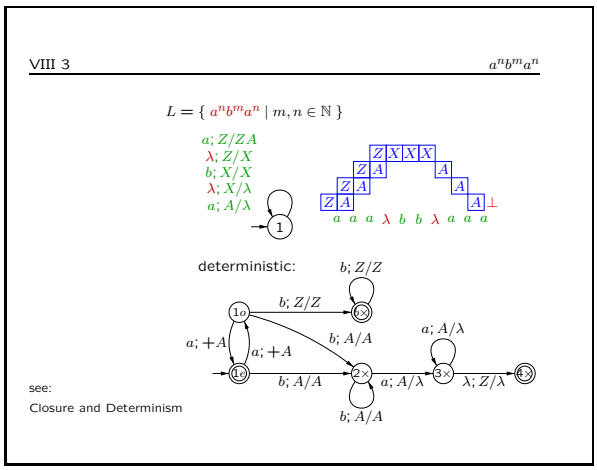


Figure 72: $a^n b^m a^n$

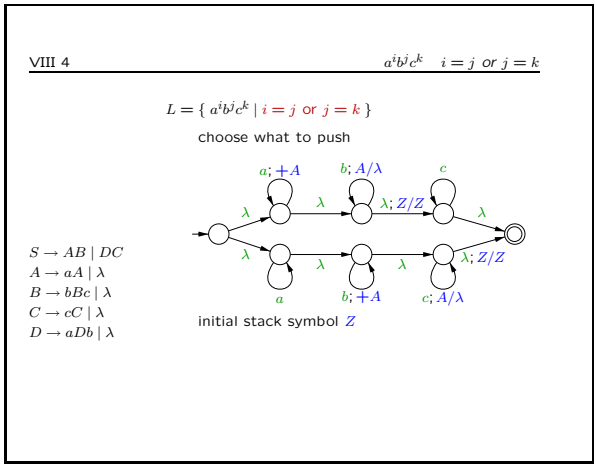


Figure 73: $a^i b^j c^k \quad i = j$ or $j = k$

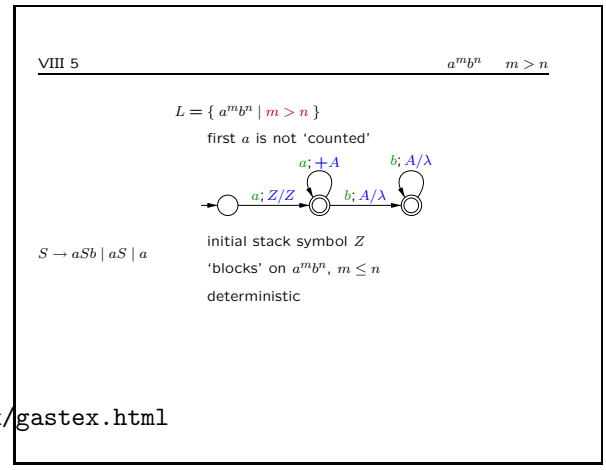


Figure 74: $a^m b^n \quad m > n$

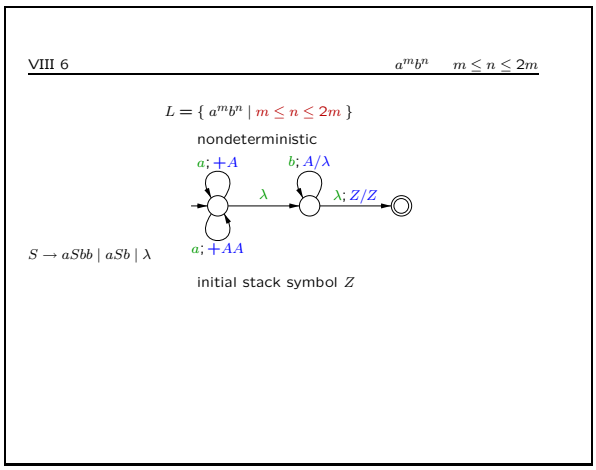


Figure 75: $a^m b^n \quad m \le n \le 2m$

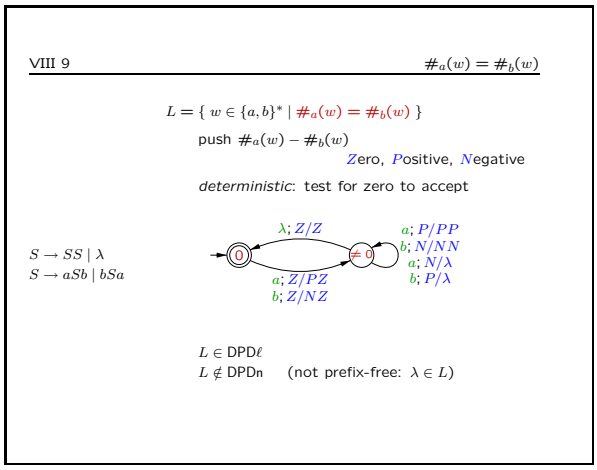


Figure 78: $\#_a(w) = \#_b(w)$

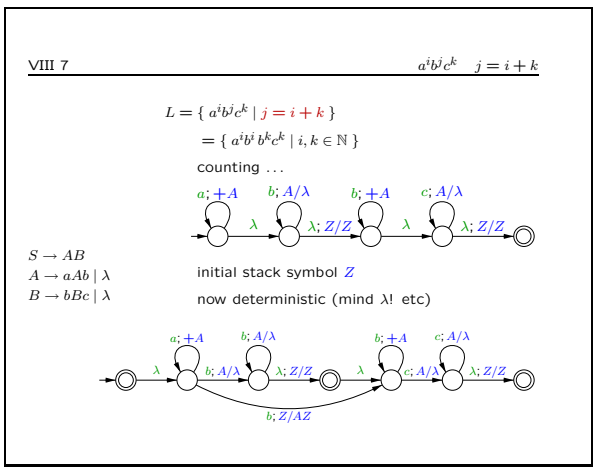


Figure 76: $a^i b^j c^k \quad j = i + k$

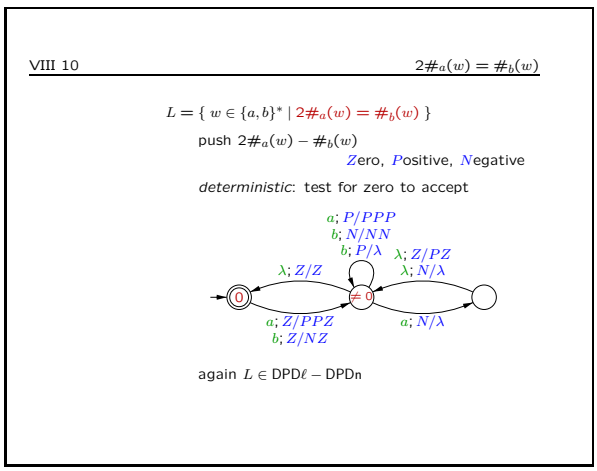


Figure 79: $2\#_a(w) = \#_b(w)$

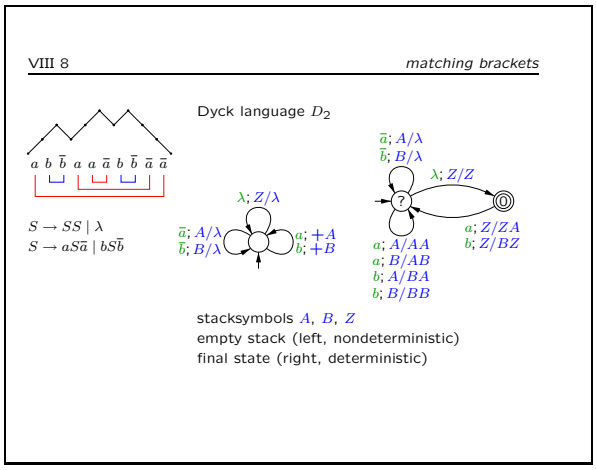


Figure 77: matching brackets

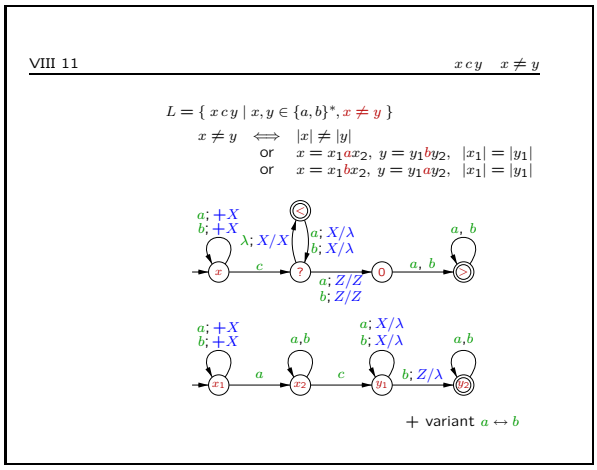


Figure 80: $x\,c\,y \quad x \neq y$