Deterministic Context-Free Languages

Huan Long

Shanghai Jiao Tong University

Acknowledgements

Part of the slides comes from a similar course given by Prof. Yijia Chen.

http://basics.sjtu.edu.cn/~chen/

Textbook

Introduction to the theory of computation
Michael Sipser, MIT
Third edition, 2012

Outline

Deterministic pushdown automata

Deterministic Context-Free Languages

Deterministic Context-Free Grammars

Relationship of DPDAs AND DCFGs

Review

Pushdown automata

Definition

A pushdown automata (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite set of input alphabet,
3. Γ is a finite set of stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $F \subseteq Q$ is the set of accept states.

Formal definition of computation

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a pushdown automata. M accepts input w if w can be written as $w = w_1 \ldots w_m$, where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \epsilon$.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$.
3. $r_m \in F$.

Theorem

A language is context free if and only if some pushdown automaton recognizes it.

Deterministic pushdown automata

Definition

A deterministic pushdown automata (DPDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite set of input alphabet,
3. Γ is a finite set of stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to (Q \times \Gamma_\epsilon) \cup \{\emptyset\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $F \subseteq Q$ is the set of accept states.

For every $q \in Q$, $a \in \Sigma$ and $x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x) \text{ and } \delta(q, \epsilon, \epsilon)$$

is not $\emptyset$.

Acceptance of DPDA

Given a DPDA and a string, we say the machine will

- Accept: If a DPDA enters an accept state after it has read the last input symbol of an input string, then it accepts that string.
- Reject: In all other cases, it reject that string.

It could be one of the following cases:

1. the DPDA reads the entire input but does not enter an accept state when it is at the end, or
2. the DPDA fails to read the entire input string,
   (1) the DPDA tries to *pop an empty stack*, or
   (2) the DPDA makes an *endless sequence of* $\epsilon-$*input moves* without reading any new inputs.

The language of a DPDA is a deterministic context-free language (DCFL).

Lemma

Every DPDA has an equivalent DPDA that always reads the entire input string.

Proof (1)

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a DPDA. P may fail to read the entire input for two reasons:

1. It tries to pop an empty stack - hanging.
2. It makes an endless sequence of $\epsilon-$input moves - looping.

To solve hanging, we initialize the stack with \$. if \$ is popped from the stack before the end of the input, P reads to the end of the input and rejects.

For looping, we identify the looping situations, i.e., those from which no further input symbol is ever read, and reprogramming P so that it reads and rejects the input instead of looping.

Proof (2)

Add new states:

$$q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}.$$

Then modify P in three steps.
(i) Once P enters an accept state, it remains in accepting states until it reads the next input symbol:

- Add a new accept state q_a for every $q \in Q$.
- For every $q \in Q$, if $\delta(q, \epsilon, x) = (r, y)$, then set $\delta(q_a, \epsilon, x) = (r_a, y)$. If $q \in F$, also change $\delta(q, \epsilon, x) = (r_a, y)$.
- For each $q \in Q$ and $a \in \Sigma$, if $\delta(q, a, x) = (r, y)$, then $\delta(q_a, a, x) = (r, y)$.
- Let F' be the set of new and old accept states.

Proof (3)

(ii) P rejects if it tries to pop an empty stack.

- P initializes the stack with the symbol \$ by $\delta(q_{\text{start}}, \epsilon, \epsilon) = (q_0, \$)$.
- If P detects \$ while in a non-accepting state, it enters q_{reject} and scans the input to the end.
  More precisely, if $q \notin F'$, then set $\delta(q, a, \$) = (q_{\text{reject}}, \epsilon)$. For $a \in \Sigma$, set $\delta(q_{\text{reject}}, a, \epsilon) = (q_{\text{reject}}, \epsilon)$.
- If P detects \$ while in an accept state, it enters q_{accept}. Then if any input remains unread, it enters q_{reject} and scans the input to the end.
  More precisely, if $q \in F'$, then set $\delta(q, \epsilon, \$) = (q_{\text{accept}}, \epsilon)$. For $a \in \Sigma$, set $\delta(q_{\text{accept}}, a, \epsilon) = (q_{\text{reject}}, \epsilon)$.

Proof (4)

(iii) Modify P to reject instead of making an endless sequence of $\epsilon-$input moves.

- For every $q \in Q$ and $x \in \Gamma$, call (q, x) a looping situation if, when P is started in state q with $x \in \Gamma$ on the top of the stack, if it never pops anything below x and it never reads an input symbol.
- A loop situation is accepting, if P enters an accept state during its subsequent moves, and otherwise rejecting.
- If (q, x) is an accepting looping situation, set $\delta(q, \epsilon, x) = (q_{\text{accept}}, \epsilon)$.
- If (q, x) is a rejecting looping situation, set $\delta(q, \epsilon, x) = (q_{\text{reject}}, \epsilon)$.

Theorem

The class of DCFLs is closed under complementation. That is, if A is a DCFL, then

$$\Sigma^* - A = \{s \in \Sigma^* \mid s \notin A\}$$

is also a DCFL.

Proof (1)

We cannot simply swap the accept non-accept states of a DPDA. The DPDA may accept its input by entering both accept and non-accept states in a sequence of moves at the end of the input string.

Assume $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a DPDA for A which always reads the whole input string. Moreover, once P enters an accept state, it remains in accept states until it reads the next input symbol.

Proof (2)

We designate some states as reading states.

- If P in a state q reads an $a \in \Sigma$ without popping the stack, i.e., $\delta(q, a, \epsilon) \neq \emptyset$, then q is a reading state.
- If $\delta(q, a, x) = (r, y)$, then add a new state q_x and modify δ as

  $$\delta(q, \epsilon, x) = (q_x, \epsilon) \text{ and } \delta(q_x, a, \epsilon) = (r, y).$$

  Let q_x be a reading state, and it is an accept state if $q \in F$.

Remove the accepting state designation from any state which isn't a reading state.

Proof (3)

The modified DPDA is equivalent to P, but it enters an accept state at most once per input symbol, when it is about to read the next symbol.

Now, invert which reading states are classified as accepting. The resulting DPDA recognizes the complementary language.

Corollary

Any CFL whose complement is not a CFL is not a DCFL.

Example

$\{a^i b^j c^k \mid i \neq j$ or $j \neq k\}$ is a CFL but not a DCFL.

Endmarked languages

Definition

For any language A the endmarked language $A \dashv$ is defined by

$$\{w \dashv \mid w \in A\}.$$

Here $\dashv$ is the special endmarker symbol.

Theorem

A is a DCFL if and only if $A \dashv$ is a DCFL.

Proof (1)

Let P be a DPDA recognizing A. Then DPDA P' recognizes $A \dashv$ by:

1. Simulating P until P' reads $\dashv$.
2. P' accepts if P had entered an accept state during the previous symbol.
3. P' does not read any symbols after $\dashv$.

Proof (2)

Let DPDA $P = (Q, \Sigma \cup \{\dashv\}, \Gamma, \delta, q_0, F)$ recognize $A \dashv$. Modify P so that each of its moves does exactly one of the following operations:

- read an input symbol;
- push a symbol onto the stack;
- or pop a symbol from the stack.

We construct a P' to simulate P. Every time P' pushes a stack symbols of P', then it pushes a symbol representing a subset of Ps states. i.e., $\Gamma' = \Gamma \cup \mathcal{P}(Q)$.

Initially, P' pushes the set R_0 defined by

$R_0 = \{q \in Q \mid$ when P is started in q with an empty stack
it eventually accepts without reading any input symbols.$\}$

Proof (3)

Then P' begins simulating P.

- To simulate a pop move, P' first pops and discards the set of states on the top of the stack, then it pops again to obtain the symbol in order to simulate P.
- To simulate a push move $\delta(q, \epsilon, \epsilon) = (r, x)$, P' examines the set of states R on the top of its stack, and then it pushes x and the set of states

  $$S = \{q \mid q \in F \text{ or } \delta(q, \epsilon, x) = (r, \epsilon) \text{ and } r \in R\}.$$
- P' simulates a read move $\delta(q, a, \epsilon) = (r, \epsilon)$ by examining the set R on the top of the stack and entering an accept state if $r \in R$.
  - If P' is at the end of the input string, then it accepts.
  - Otherwise, it will continue simulating P, so this accept state must also record Ps state.

    Thus we create this state as a second copy of Ps original state, marking it as an accept state in P'.

Our goal is to define deterministic context-free grammars (DCFG), the counterpart to deterministic pushdown automata.

We will show that these two models are equivalent on endmarked languages.

From derivations to reductions

Derivation

Derivations in CFGs begin with the start variable and proceed top down with a series of substitutions according to the grammar rules, until the derivation obtains a string of terminals.

Reduction

For defining DCFGs we take a bottom up approach, by starting with a string of terminals and processing the derivation in reverse, employing a series of reduce steps until reaching the start variable.

- Each reduce step is a reversed substitution, whereby the string of terminals and variables on the right-hand side of a rule is replaced by the variable on the corresponding left-hand side.
- The string replaced is the reducing string.
- The entire reversed derivation is a reduction.

Formal definition of reductions

If u and v are strings of variables and terminals, write $u \rightarrowtail v$ to mean that v can be obtained from u by a reduce step, i.e., $v \Rightarrow u$.

A reduction from u to v is a sequence

$$u = u_1 \rightarrowtail u_2 \rightarrowtail \ldots \rightarrowtail u_k = v$$

and we say that u is reducible to v, written $u \overset{*}{\rightarrowtail} v$, equivalently $v \overset{*}{\Rightarrow} u$.

A reduction from u is a reduction from u to the start variable.

In a leftmost reduction, each reducing string is reduced only after all other reducing strings that lie entirely to its left.

Handle

Let w be a string in the language of CFG G, and let u_i appear in a leftmost reduction of w. In the reduce step $u_i \rightarrowtail u_{i+1}$, say that the rule $T \to h$ was applied in reverse.

Therefore ,

$$u_i = xhy \text{ and } u_{i+1} = xTy,$$

where h is the reducing string.

$$u_i = \overbrace{x_1 \cdots x_j}^{x} \overbrace{h_1 \cdots h_k}^{h} \overbrace{y_1 \cdots y_\ell}^{y} \rightarrowtail \overbrace{x_1 \cdots x_j}^{x} \overbrace{T}^{T} \overbrace{y_1 \cdots y_\ell}^{y} = u_{i+1}$$

We call h, together with its reducing rule $T \to h$, a handle of u_i. A string that appears in a leftmost reduction of some string in $L(G)$ is called a valid string. We define handles only for valid strings.

Valid string may have several handles, but only if the grammar is ambiguous. Unambiguous grammars may generate strings by one parse tree only, and therefore the leftmost reductions, and hence the handles, are also unique. In that case, we may refer to the handle of a valid string.

Observe that y, the portion of u_i following a handle, is always a string of terminals because the reduction is leftmost. Otherwise, y would contain a variable symbol and that could arise only from a previous reduce step whose reducing string was completely to the right of h. But then the leftmost reduction should have reduced the handle at an earlier step.

$$u_i = \overbrace{x_1 \cdots x_j}^{x} \overbrace{h_1 \cdots h_k}^{h} \overbrace{y_1 \cdots y_\ell}^{y} \rightarrowtail \overbrace{x_1 \cdots x_j}^{x} \overbrace{\ T\ }^{T} \overbrace{y_1 \cdots y_\ell}^{y} = u_{i+1}$$

Example (1)

Consider the grammar G:

$$\begin{array}{rcl} R & \rightarrow & S \mid T \\ S & \rightarrow & aSb \mid ab \\ T & \rightarrow & aTbb \mid abb \end{array}$$

Then

$$\begin{array}{rcl} L(G) & = & B \cup C \\ \text{where } B & = & \{a^m b^m \mid m \geq 1\} \\ \text{and } C & = & \{a^m b^{2m} \mid m \geq 1\} \end{array}$$

Some leftmost reductions:

$$aa\underline{ab}bb \rightarrowtail a\underline{aSb}b \rightarrowtail \underline{aSb} \rightarrowtail \underline{S} \rightarrowtail R,$$

$$aa\underline{abb}bbbb \rightarrowtail a\underline{aTbb}bb \rightarrowtail \underline{aTbb} \rightarrowtail \underline{T} \rightarrowtail R.$$

Example (2)

Consider the grammar G:

$$\begin{array}{lcl} S & \rightarrow & T \dashv \\ T & \rightarrow & T(T) \mid \epsilon \end{array}$$

A leftmost reductions:

$$\_()()\dashv \;\rightarrowtail\; T(\_)()\dashv \;\rightarrowtail\; \underline{T(T)}()\dashv \;\rightarrowtail\; T(\_)\dashv \;\rightarrowtail\; \underline{T(T)}\dashv \;\rightarrowtail\; \underline{T\dashv} \;\rightarrowtail\; S.$$

Forced handles

Definition

A handle h of valid string $v = xhy$ is a forced handle if h is the unique handle in every valid sting $xh\hat{y}$ where $\hat{y} \in \Sigma^*$.

Definition

A deterministic context-free grammar (DCFG) is a context-free grammar such that every valid string has a forced handle.

The above definition does not tell us how to decide whether a given CFG is a DCFG.

DK-test

For any CFG we can construct an associated DFA DK that can identify handles. DK accepts its input z if

1. z is the prefix of some valid string $v = zy$, and
2. z ends with a handle of v.

Moreover, each accept state of DK indicates the associated reducing rule(s). In a general CFG, multiple reducing rules may apply, depending on which valid v extends z. But in a DCFG, each accept state corresponds to exactly one reducing rule.

Two provisos

1. The start variable of a CFG does not appear on the right-hand side of any rule.
2. Every variable appears in a reduction of some string in the grammar's language.

The plan

To construct DFA DK, we will construct an equivalent NFA K and convert K to DK via the subset construction.

To understand K, we introduce an NFA J which

> *accepts every input string that ends with the right-hand side of any rule.*

The NFA J

1. guesses a rule $B \to u$ to use,
2. guess at which point to start matching the input with u,
3. keeps track of its progress through u.

We represent this progress by placing a dot in the corresponding point in the rule. yielding the following dotted rules:

$$\begin{array}{rcl}
B & \to & {.}u_1 u_2 \cdots u_{k-1} u_k \\
B & \to & u_1 {.} u_2 \cdots u_{k-1} u_k \\
 & \vdots & \\
B & \to & u_1 u_2 \cdots u_{k-1} {.} u_k \\
B & \to & u_1 u_2 \cdots u_{k-1} u_k {.}
\end{array}$$

Dotted rules

1. Each dotted rule corresponds to one state of J, i.e., for $B \to u.v$ we have a state $\boxed{B \to u.v}$.
2. The accept states $\boxed{\boxed{B \to u.}}$, which correspond to the complete rules.
3. We add a separate start state with a self-loop on all symbols and an $\epsilon-$move to $\boxed{B \to .u}$ for each rule $B \to u$.

Thus J accepts if the match completes successfully at the end of the input. If a mismatch occurs or if the end of the match does not coincide with the end of the input, this branch of Js computation rejects.

The NFA K

- Like J, the states of K correspond to all dotted rules.
- It has a special start state that has an $\epsilon-$move to $\boxed{S_1 \to .u}$ for every rule with S_1 being the start variable.
- Shift-moves: for a rule $B \to uav$ where a can be a terminal or variable we have

$$\boxed{B \to u.av} \xrightarrow{a} \boxed{B \to ua.v}$$

- $\epsilon-$moves: for a rule $B \to uCV$ and $C \to r$ we have

$$\boxed{B \to u.Cv} \xrightarrow{\epsilon} \boxed{C \to .r}$$

- The accept states are all $\boxed{\boxed{B \to u.}}$.

Lemma

K may enter state $\boxed{T \to u.v}$ on reading input z if and only if $z = xu$ and $xuvy$ is a valid string with handle uv and reducing rule $T \to uv$, for some $y \in \Sigma^*$.

Proof (1)

Assume K enters state $\boxed{T \to u.v}$ on reading input z, whose path from the start state is viewed as runs of shift-moves separated by $\epsilon-$ moves.

- The shift-moves are transitions between states sharing the same rule, shifting the dot rightward over symbols read from the input.
- In the ith run, the rule is $S_i \to u_i S_{i+1} v_i$, where S_{i+1} is the variable expanded in the next run.
- The penultimate run is for rule $S_\ell \to u_\ell T v_\ell$, and the final run has rule $T \to uv$.
- So the input $z = u_1 u_2 \ldots u_\ell u = xu$, because the strings u_i and u were the shift-move symbols read from the input.

Proof (2)

Let

$$y' = v_\ell \dots v_2 v_1$$

then $xuvy'$ is derivable in G.

- Fully expand all variables that appear in y' until each variable derives some string of terminals, and let y be the resulting string.
- The string $xuvy$ is valid because it occurs in a leftmost reduction of $w \in L(G)$, a string of terminals obtained by fully expanding all variables in $xuvy$.
- uv is the handle in the reduction and its reducing rule is $T \to uv$.

Proof (3)

Assume that string $xuvy$ is a valid string with handle uv and reducing rule $T \to uv$. We need to show that K may enter $\boxed{T \to u.v}$ on reading input xu.

1. The parse tree is rooted at the start variable S_1 and it must contain the variable T because $T \to uv$ is the first reduce step in the reduction of $xuvy$.
2. Let $S_2, \ldots, S_\ell$ be the variables on the path from S_1 to T. All variables in the parse tree that appear leftward of this path must be unexpanded, or else uv would not be the handle.
3. Each S_i leads to S_{i+1} by some rule $S_i \to u_i S_{i+1} v_i$:

$$\begin{array}{rcl} S_1 & \to & u_1 S_2 v_1 \\ S_2 & \to & u_2 S_3 v_2 \\ & \vdots & \\ S_\ell & \to & u_\ell T v_\ell \\ T & \to & uv. \end{array}$$

Proof (4)

On the input $z = xu$, the path from K's start state to $\boxed{T \to u.v}$ is:

1. K makes an $\epsilon-$moves to $\boxed{S_1 \to .u_1 S_2 v_1}$.
2. Reading the symbols of u_1, it performs the corresponding shift-moves until it enters $\boxed{S_1 \to u_1 . S_2 v_1}$ at the end of u_1.
3. It makes an $\epsilon-$move to $\boxed{S_2 \to .u_2 S_3 v_2}$ and continues with shift-moves on reading u_2 until it reaches $\boxed{S_2 \to u_2 . S_3 v_2}$.
   $\dots$
4. After reading u_ℓ it enters $\boxed{S_\ell \to u_\ell . T v_\ell}$ which leads by an $\epsilon-$move to $\boxed{T \to .uv}$.
5. Finally after reading u it is in $\boxed{T \to u.v}$.

Corollary

K may enter state $\boxed{\boxed{T \to h.}}$ on reading input z if and only if $z = xh$ and h is a handle of some valid string xhy with reducing rule $T \to h$.

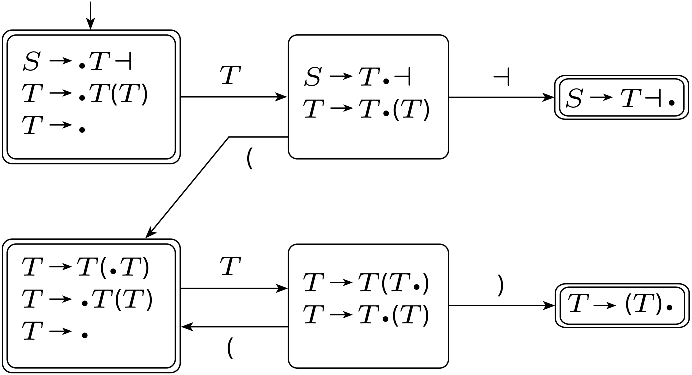
From K to DK

We convert NFA k to DFA DK by using the subset construction. All states unreachable from the start state are removed.

Thus, each of DK's status thus contains one or more dotted rules. Each accept state contains at least one completed rule.

An example

$$
\begin{array}{rcl}
S & \rightarrow & T\dashv \\
T & \rightarrow & T(T) \mid \epsilon
\end{array}
$$



$S \rightarrow \bullet T\dashv$
$T \rightarrow \bullet T(T)$
$T \rightarrow \bullet$
T
$S \rightarrow T\bullet\dashv$
$T \rightarrow T\bullet(T)$
$\dashv$
$S \rightarrow T\dashv\bullet$
(
$T \rightarrow T(\bullet T)$
$T \rightarrow \bullet T(T)$
$T \rightarrow \bullet$
T
$T \rightarrow T(T\bullet)$
$T \rightarrow T\bullet(T)$
)
$T \rightarrow (T)\bullet$
(

The $DK-$test

Starting with a CFG G, construct the associated DFA DK. Determine whether G is deterministic by examining DKs accept states. G passes the $DK-$test if every accept state contains

1. exactly one completed rule, and
2. no dotted rule in which a terminal symbol immediately follows the dot, i.e., no dotted rule of the form $B \rightarrow u.av$ for $a \in \Sigma$.

Theorem

G passes the $DK-$test if and only if G is a DCFG.

Proof (1)

Assume that G is not deterministic and show that it fails the DK-test.

- Take a valid string xhy that has an unforced handle h.
- Some valid string xhy' has a different handle $\hat{h} \neq h$, where y' is a string of terminals, i.e., $y' \in \Sigma^*$. Thus $xhy' = \hat{x}\hat{h}\hat{y}$.
- If $xh = \hat{x}\hat{h}$, then input xh sends DK to a state with two completed rules, failing the DK-test.
- If $xh \neq \hat{x}\hat{h}$, by symmetry, we assume that xh is a proper prefix of $\hat{x}\hat{h}$. Then $y' \in \Sigma^+$.
  - Let q be the state that DK enters on input xh, which must be accepting for h is a handle of xhy.
  - A transition arrow must exit q as $\hat{x}\hat{h}$ sends DK to an accept state via q. That transition arrow is labeled with a terminal symbol for $y' \in \Sigma^+$.

  Hence q contains a dotted rule with a terminal symbol immediately following the dot, violating the DK-test.

Proof (2)

Assume G fails the DK-test at some accept state q. We show that G has an unforced handle.

q has a complete rule $T \rightarrow h$., for it is accepting. Let z be a string that leads DK to q. Then $z = xh$ where some valid string xhy has handle h with reducing rule $T \rightarrow h$, for $y \in \Sigma^*$

- If q has another completed rule $B \rightarrow \hat{h}$.. Then some valid string xhy' must have a different handle $\hat{h}$ with reducing rule $B \rightarrow \hat{h}$. Hence, h is not a forced handle.

Proof (3)

q has a complete rule $T \rightarrow h.$, for it is accepting. Let z be a string that leads DK to q. Then $z = xh$ where some valid string xhy has handle h with reducing rule $T \rightarrow h$, for $y \in \Sigma^*$

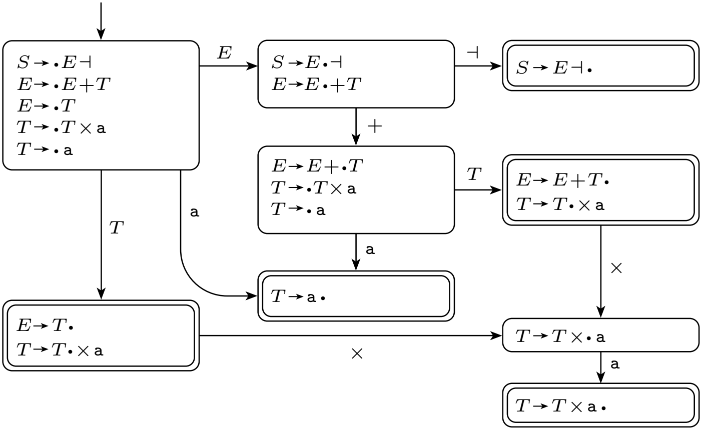
- q contains a rule $B \rightarrow u.av$ with $a \in \Sigma$.
  - xh takes DK to q, so $xh = \hat{x}u$, where $\hat{x}uav\hat{y}$ is valid and has a handle uav with reducing rule $B \rightarrow uav$, for some $\hat{y} \in \Sigma^*$.
  - Expand all variables in v to get $v' \in \Sigma^*$, and let $y' = av'\hat{y}$ with $y' \in \Sigma^*$.
  - The following is a leftmost reduction

$$xhy' = xhav'\hat{y} = \hat{x}uav'\hat{y} \overset{*}{\rightarrowtail} \hat{x}uav\hat{y} \rightarrowtail \hat{x}B\hat{y} \overset{*}{\rightarrowtail} S.$$

  - $\hat{x}uav\hat{y}$ is valid, and we can obtain $\hat{x}uav'\hat{y}$ from it using a rightmost derivation, so $\hat{x}uav'\hat{y}$ is also valid.
  - The handle of $\hat{x}uav'\hat{y}$ either lies inside v' (if $v \neq v'$) or is uav (if $u = v$). In either case, the handle includes a or follows a and thus cannot be h because h fully precedes a. Hence h is not a forced handle.

An example failing the $DK-$test

$$\begin{aligned} S &\rightarrow E\dashv \\ E &\rightarrow E+T \mid T \\ T &\rightarrow T\times a \mid a \end{aligned}$$



$S\to\bullet E\dashv$
$E\to\bullet E+T$
$E\to\bullet T$
$T\to\bullet T\times\mathtt{a}$
$T\to\bullet\,\mathtt{a}$

$S\to E\bullet\dashv$
$E\to E\bullet+T$

$S\to E\dashv\bullet$

$E\to E+\bullet T$
$T\to\bullet T\times\mathtt{a}$
$T\to\bullet\,\mathtt{a}$

$E\to E+T\bullet$
$T\to T\bullet\times\mathtt{a}$

$T\to\mathtt{a}\,\bullet$

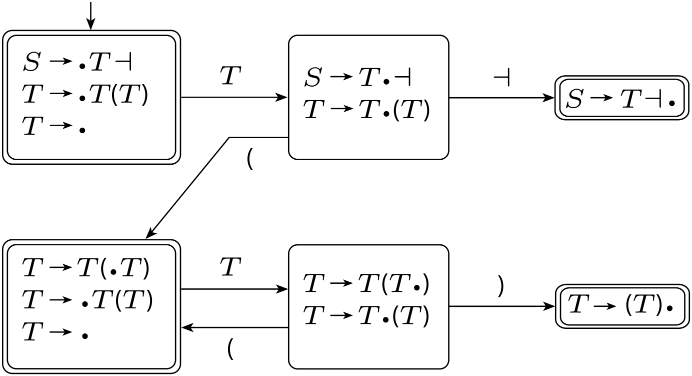
$E\to T\bullet$
$T\to T\bullet\times\mathtt{a}$

$T\to T\times\bullet\,\mathtt{a}$

$T\to T\times\mathtt{a}\,\bullet$

An example passing the DK–test

$$\begin{array}{rcl} S & \to & T\dashv \\ T & \to & T(T) \mid \epsilon \end{array}$$



Relationship of DPDAs AND DCFGs

Theorem

An endmarked language is generated by a deterministic context-free grammar if and only if it is deterministic context free.

Endmarked languages

Definition

For any language A the endmarked language $A \dashv$ is defined by

$$\{w \dashv \; | \; w \in A\}$$

Here $\dashv$ is the special endmarker symbol.

Theorem

A is DCFL if and only if $A \dashv$ is a DCFL.

Prefix-free languages

Definition

$A \subseteq \Sigma^*$ is prefix-free if for every $w \in A$ and every proper prefix w' of w we have $w' \notin A$.

Lemma

Every $A \dashv$ is prefix-free.

Lemma

Every DCFG generates a prefix-free language.

Prefix-free languages (cont'd)

The grammar

$$\begin{array}{rcl} S & \to & T\dashv \\ T & \to & T(T) \mid \epsilon \end{array}$$

is DCFG, which recognizes some $A \dashv$.

However, A cannot be generated by DCFG, since it is not prefix-free.

Lemma

Every DCFG has an equivalent DPDA.

Lemma

Every DPDA that recognizes an endmarked language has an equivalent DCFG.

Proof (1)

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ be a DPDA. Recall the previous construction: For each pair of states p and q, the grammar has a variable A_{pq} which generates all strings taking P from p with an empty stack to q with an empty stack.

We modify P such that:

1. It has a single accept state q_{accept}.
2. It empties its stack before accepting, which cannot be deterministic without reading the endmark $\dashv$.
3. Each transition either pushes a symbol onto the stack or pops one off the stack, but it does not do both at the same time.

Proof (2)

1. For every $p, q, r, s \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon) = (r, u)$ and $\delta(s, b, u) = (q, \epsilon)$, then add $A_{pq} \to aA_{rs}b$.
2. For every $p, q, r, s \in Q$, add $A_{pq} \to A_{pr}A_{rq}$.
3. For every $p \in Q$, add $A_{pp} \to \epsilon$.

To avoid ambiguity, we combine rules of type 1 and 2 into:

1-2 For every $p, q, r, s, t \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(r, a, \epsilon) = (s, u)$ and $\delta(t, b, u) = (q, \epsilon)$, add the rule $A_{pq} \to A_{pr}aA_{st}b$.

Let G be re resulting grammar.

Proof (3)

In a derivation of the original grammar, for each substitution due to a type 2 rule $A_{pq} \to A_{pr} A_{rq}$, we can assume that

> *r is P's state when it is at the rightmost point where the stack becomes empty midway.*

Then the subsequent substitution of A_{rq} must expand it using a type 1 rule $A_{rq} \to a A_{st} b$.
We can combine these two substitutions into a single type 1-2 rule

$$A_{pq} \to A_{pr} a A_{st} b$$

Proof (4)

In a derivation using the modified grammar G, if we replace each type 1-2 rule $A_{pq} \to A_{pr} a A_{st} b$ by the type 2 rule $A_{pq} \to A_{pr} A_{rq}$ followed by the type 1 rule $A_{rq} \to a A_{st} b$, we get the same result.

Proof (5)

We use DK-test to show that G is deterministic

- We need to analyze how P operates on valid strings by extending its input alphabet and transition function to process variable symbols in addition to terminal symbols.
- Add all symbols A_{pq} to P's input alphabet and extend its δ by defining
$$\delta(p, A_{pq}, \epsilon) = (q, \epsilon).$$
Set all other transitions involving A_{pq} to $\emptyset$.
- To preserve P' deterministic behavior, if P reads A_{pq} from the input then disallow an ϵ-move.

Proof (6)

Consider the derivation:

$$A_{q_0,q_{\text{accept}}} = V_0 \Rightarrow v_1 \Rightarrow \cdots v_i \Rightarrow \cdots \Rightarrow v_k = w.$$

Claim

If P reads v_i containing a variable A_{pq}, it enters state p just prior to reading A_{pq}.

If $i = 0$, then $v_i = A_{q_0,q_{\text{accept}}}$ and P starts in q_0.

Assume the claim is true for some $i \geq 0$.

- $v_i = xA_{pq}y$ and A_{pq} is the variable substituted in the step $v_i \Rightarrow v_{i+1}$. By IH, P enters state p after reading x, prior to reading A_{pq}. By the construction of G, the substitution rules may of two types:
  1. $A_{pq} \rightarrow A_{pr}aA_{st}b$ or
  2. $A_{pp} \rightarrow \epsilon$.

Proof (7)

- Thus either $v_{i+1} = xA_{pr}aA_{st}$ or $v_{i+1} = xy$.
- In the first case, when P reads $A_{pr}aA_{st}b$ in v_{i+1}, we know it starts in state p, because it has just finished reading x.
  As P reads $A_{pr}aA_{st}b$ in v_{i+1}, it enters the sequence of states r, s, t, and q, due to the substitution rule's construction.
  Therefore, it enters state p just prior to reading A_{pr} and it enters state s just prior to reading A_{st}.
  The claim holds on variables in the y part because, after reading b, P enters state q and then it reads y. On input v_i, it also enters q just before reading y, so the computation agree on the y parts of v_i and v_{i+1}.
  Obviously, the computations agree on the x parts.
- In the second case, no new variables are introduced, so we only need to observe that the computations agree on the x and y parts of v_i and v_{i+1}.

Proof (8)

Claim

G passes the DK-test.

Select one of the accept states of DK for G. It contains a completed rule R which can be

1. $A_{pq} \to A_{pr} a A_{st} b$, or
2. $A_{pp} \to$.

We need to show in both cases that the state cannot contain

a. another complete rule, and

b. a dotted rule that has a terminal symbol immediately after the dot.

In each case, we start by considering a string z on which DK goes to the above accept state.

Case 1a.

R is a completed type 1-2 rule. For any rule in this accept state, z must end with the symbols preceding the dot in that rule because DK goes to that state on z. Hence the symbols preceding the dot must be consistent in all such rules.

- These symbols are $A_{pr}aA_{st}b$ in R so any other type 1-2 completed rule must have exactly the same symbols on the right-hand side.
- The variables on the left-hand side must also agree, so the rules must be the same.

Case 1a. (cont'd)

- Assume that the accept state contains R and some type 3 completed ϵ-rule $T = A_{ss} \rightarrow .$.
- From R we know that z ends with $A_{pr}aA_{st}b$.
- P pops its stack at the very end of z because a pop occurs at that point in R, due to G's construction.
- According to the way we build DK, a completed ϵ-rule in a state must derive from a dotted rule that resides in the same state, where the dot isn't at the very beginning and the dot immediately precedes some variable.
- An exception occurs at DK's start state, where this dot may occur at the beginning of the rule, but this accept state cannot be the start state because it contains a completed type 1-2 rule.
- In G, that means T derives from a type 1-2 dotted rule where the dot precedes the second variable. From G's construction a push occurs just before the dot.
- This implies that P does a push move at the very end of z, contradicting our previous statement.