

Push-Down Automata

COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University

Semester 2, 2011

Languages and Automata

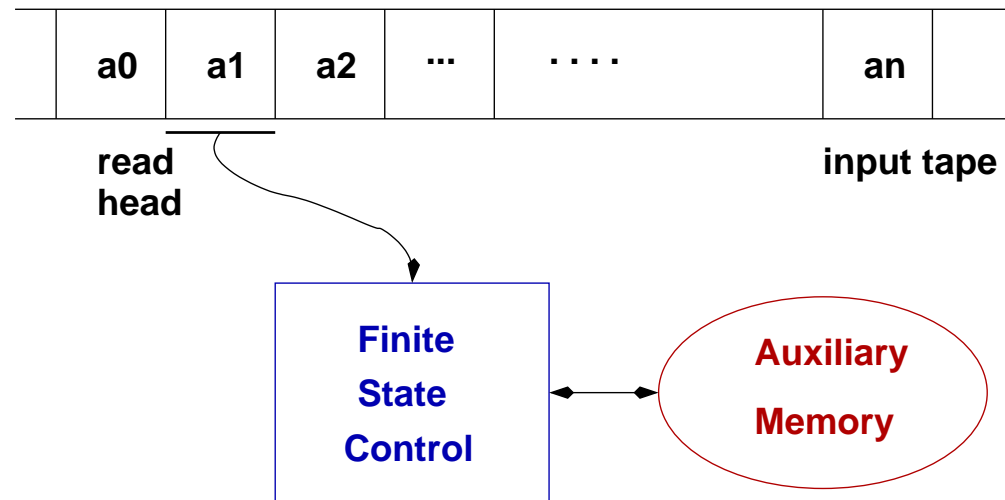
Recall that to define a language we can either:

1. Give a *set of rules* (i.e. a grammar) to produce all the legal strings (sentences) of the language.
2. Provide a *machine* (i.e. an algorithm) to recognise all the sentences of the language.

There is a close relationship between the two approaches. Commonly we *define* a language by giving a grammar and then base *parsers* (or compilers) on the corresponding machine.

The machines are automata like Turing machines, but constrained in the same sense as the Chomsky hierarchy.

General Structure of Automata



The **input tape** is a sequence of tokens.

Each time a symbol is processed the read head advances.

The **auxiliary memory** is usually a linear organisation (e.g. a stack).

The memory alphabet is usually $V_t \cup V_n$.

The **finite state control** can be in any one of a finite number of states.

General Automata ctd

Each **action** of the machine may change the FSC *state*, change the auxiliary *memory*, advance to the next *input* symbol.

The action of the machine **depends on** the current FSC *state*, the current *input* symbol, the current *memory* symbol(s).

The machine **starts** in some particular start state (q_0), with the read head at the first input symbol (a_0), with the memory empty.

A machine **accepts** an input string as a sentence of the language if it reaches a final state with the input exhausted.

Automata and Grammars

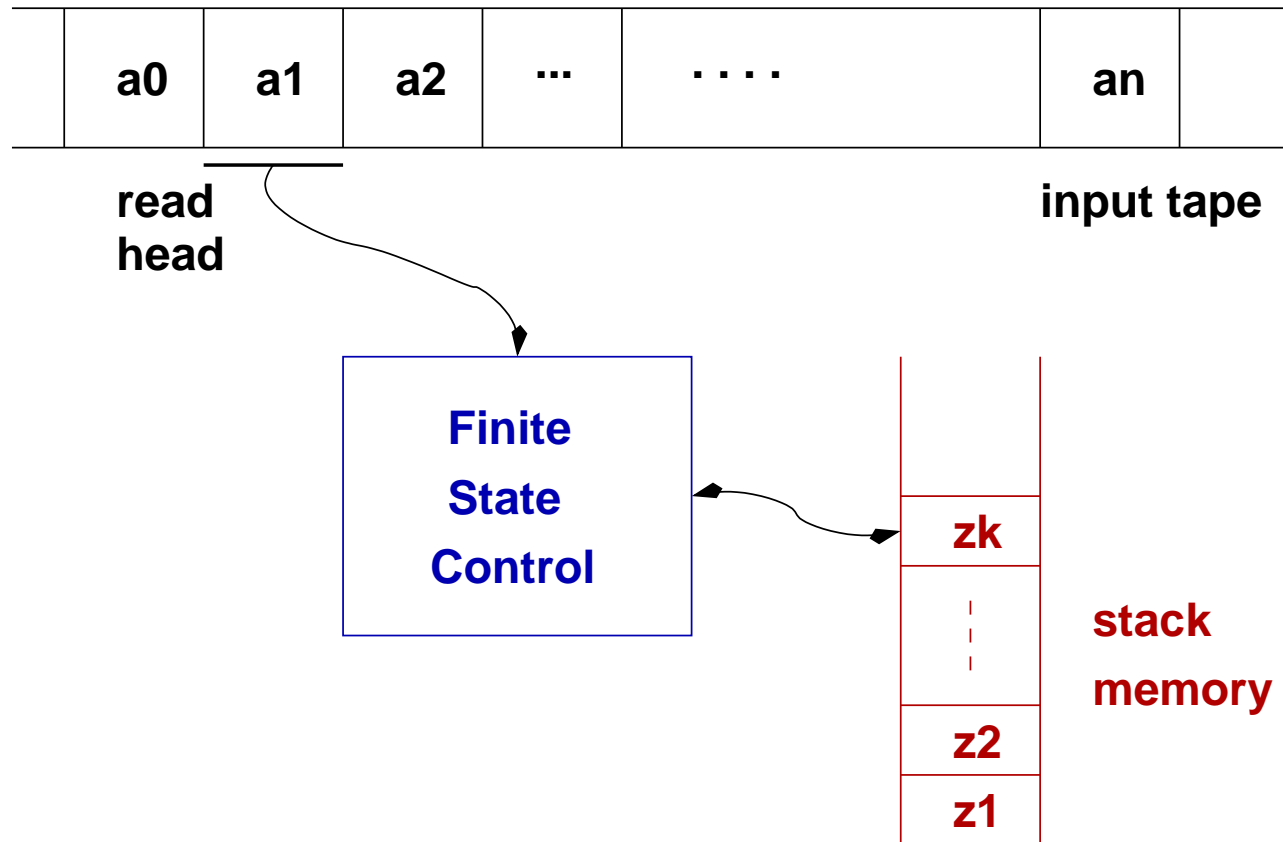
The kind of auxiliary memory in a machine determines the class of languages that the machine can recognise:

| Language Class | Memory |
|---------------------|--------------------------------|
| <i>regular</i> | <i>none</i> |
| <i>context-free</i> | <i>stack</i> |
| context-sensitive | tape (bounded by input length) |
| unrestricted | unbounded tape |

We have already looked at Finite State Automata (i.e. automata without memory and their relation to regular languages.

We now consider *Push-Down Automata* (i.e. automata with stack memory) and their relation to context-free grammars and languages.

Push-down Automata — PDA



PDA's ctd

Each **action** of the machine may involve change to the FSC state, pushing or popping the stack, advance to the next input symbol.

The action of the machine may **depend on** the current FSC state, the current input symbol, the current top-of-stack symbol.

The machine **accepts** an input string if it reaches a final state, with the input exhausted *and the stack empty*.

NOTE: There are other definitions of a PDA, where

- an input string is accepted if the PDA has an empty stack with the input exhausted
- an input string is accepted if the PDA is in a final state with the input exhausted

Example

$$\{a^n b^n \mid n \in \mathbb{N}\}$$

Recall that this language cannot be recognised by a FSA (because there can only be a finite number of states). But it can be recognised by a PDA.

Ad hoc design:

- **phase 1:** (state q_1) *push as* from the input onto the stack
- **phase 2:** (state q_2) *pop as* from the stack, if there is a b on input
- **finalise:** if the stack is empty and the input is exhausted in the final state (q_3), accept the string.

Example ctd

PDA transitions modify the stack as well as change the FSC state.

For **deterministic PDAs** transitions are a function δ of type:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$$

$$\delta : (\text{state, input token or } \varepsilon, \text{top-of-stack}) \rightarrow (\text{new state, stack string})$$

- Q is the set of states
- $\Sigma \cup \{\varepsilon\}$ is the alphabet of input symbols, with also the symbol ε
- Γ is the alphabet of stack symbols

Notation: we write $\delta(q, a, s) = q' / \sigma$ Then the **string** σ in the result is the symbols with which to **replace** the top-of-stack symbol (**tos**) s .

(Doing this makes it simple to specify pushes and pops in a uniform way.)

Example ctd

To simplify (the notation for) testing for empty stack, we assume a marker symbol Z is initially on the stack.

PDA to recognise $a^n b^n$ (start state q_0 , final state(s) underlined):

| | | |
|---|-----|----------------------|
| $\delta(q_0, a, Z) = q_1/aZ$ | ... | push first a |
| $\delta(q_1, a, a) = q_1/aa$ | ... | push a 's |
| $\delta(q_1, b, a) = q_2/\varepsilon$ | ... | start popping a 's |
| $\delta(q_2, b, a) = q_2/\varepsilon$ | ... | pop a 's |
| $\delta(q_2, \varepsilon, Z) = \underline{q_3}/\varepsilon$ | ... | accept |

The transition $\delta(q_2, \varepsilon, Z) = \underline{q_3}/\varepsilon$ does not use or consume an input symbol

Note that this definition of δ makes it a **partial function**. It is undefined for many arguments.

Example ctd — PDA Trace

PDA configurations can be written as a triple $(state, remaining\ input, stack)$ with the top of stack to the *left*.

$$\begin{aligned}(q_0, aaabbb, Z) &\Rightarrow (q_1, aabbb, aZ) \\ &\Rightarrow (q_1, abbb, aaZ) \\ &\Rightarrow (q_1, bbb, aaaZ) \\ &\Rightarrow (q_2, bb, aaZ) \\ &\Rightarrow (q_2, b, aZ) \\ &\Rightarrow (q_2, \varepsilon, Z) \\ &\Rightarrow (\underline{q_3}, \varepsilon, \varepsilon)\end{aligned}$$

The machine halts in the final state with input exhausted and empty stack, so the string is accepted.

Example ctd — Rejection

The string *aaba* should be rejected by the PDA:

$$\begin{aligned}(q_0, aaba, Z) &\Rightarrow (q_1, aba, aZ) \\ &\Rightarrow (q_1, ba, aaZ) \\ &\Rightarrow (q_2, a, aZ) \\ &\Rightarrow ???\end{aligned}$$

No transition applies, and the PDA is “stuck” without reaching a final state.

Rejection happens when the transition function is undefined for the current configuration (that is *state*, *input symbol* and *top of stack symbol*).

PDA transitions

- when δ contains $(q_1, x, s) \mapsto q_2/\sigma$,

| | state | input | stack |
|---------------------|--------------|--------------------|--------------------------------|
| the PDA can go from | q_1 | next symbol is x | s on top-of-stack |
| to | q_2 | x is consumed | σ replaces s on stack |

- and when δ contains $(q_1, \varepsilon, s) \mapsto q_2/\sigma$,

| | state | input | stack |
|---------------------|--------------|--------------------|--------------------------------|
| the PDA can go from | q_1 | any input, or none | s on top-of-stack |
| to | q_2 | input is ignored | σ replaces s on stack |

Such transitions **do not look at or consume** an input symbol

Non-deterministic PDAs

For **deterministic PDAs** transitions are a (partial) **function**:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$$

$$\delta : (\text{state, input token or } \varepsilon, \text{top-of-stack}) \rightarrow (\text{new state, stack string})$$

For **non-deterministic PDAs** transitions are a **relation**

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \leftrightarrow Q \times \Gamma^*$$

There may be configurations where there are multiple choices of transition; note also that transitions $(q_1, x, s) \mapsto \dots$ and $(q_1, \varepsilon, s) \mapsto \dots$ will make a PDA non-deterministic, as there would be a choice.

Grammars and PDAs

Theorem

The class of languages recognised by non-deterministic PDA's is *exactly* the class of **context-free languages**.

We will only justify this result in one direction: for any CFG, there is a corresponding PDA.

This is the most interesting direction since it is the basis of automatically deriving parsers from grammars.

From CFG to PDA

The translation uses three states: q_0 (initial), q_1 (processing), q_2 (final).

1. For all terminal symbols t , pop the stack if it matches the input:

$$\delta(q_1, t, t) \mapsto q_1/\varepsilon$$

2. If a *non-terminal* is on top of stack, expand it to one of its right-hand sides. For all productions $A \rightarrow \alpha$:

$$\delta(q_1, \varepsilon, A) \mapsto q_1/\alpha$$

continued...

From CFG to PDA, ctd

3. Initialise the process (start state q_0) by pushing S onto the stack.

For start symbol S :

$$\delta(q_0, \varepsilon, Z) \mapsto q_1/SZ$$

4. For termination, add the transition, with final state q_2 :

$$\delta(q_1, \varepsilon, Z) \mapsto \underline{q_2}/\varepsilon$$

In general we get a **non-deterministic** PDA since there may be several productions for each non-terminal.

Unfortunately, there is **no algorithm** for obtaining a deterministic PDA from a non-deterministic one.

Example — Derive a PDA for a CFG

$$E \rightarrow T \mid E + T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow \mathbf{id} \mid (E)$$

1. Match and pop terminals:

$$\delta(q_1, +, +) \mapsto q_1/\varepsilon$$

$$\delta(q_1, *, *) \mapsto q_1/\varepsilon$$

$$\delta(q_1, \mathbf{id}, \mathbf{id}) \mapsto q_1/\varepsilon$$

$$\delta(q_1, (, () \mapsto q_1/\varepsilon$$

$$\delta(q_1,),)) \mapsto q_1/\varepsilon$$

CFG to PDA ctd

2. Expand non-terminals:

$$\delta(q_1, \varepsilon, E) \mapsto q_1/T$$

$$\delta(q_1, \varepsilon, E) \mapsto q_1/E + T$$

$$\delta(q_1, \varepsilon, T) \mapsto q_1/F$$

$$\delta(q_1, \varepsilon, T) \mapsto q_1/T * F$$

$$\delta(q_1, \varepsilon, F) \mapsto q_1/\mathbf{id}$$

$$\delta(q_1, \varepsilon, F) \mapsto q_1/(E)$$

3,4. Initiate and terminate:

$$\delta(q_0, \varepsilon, Z) \mapsto q_1/EZ$$

$$\delta(q_1, \varepsilon, Z) \mapsto \underline{q_2}/\varepsilon$$

Example Parse

$$\begin{aligned} (q_0, \mathbf{id * id}, Z) &\Rightarrow (q_1, \mathbf{id * id}, EZ) \\ &\Rightarrow (q_1, \mathbf{id * id}, TZ) \\ &\Rightarrow (q_1, \mathbf{id * id}, T * FZ) \\ &\Rightarrow (q_1, \mathbf{id * id}, F * FZ) \\ &\Rightarrow (q_1, \mathbf{id * id}, \mathbf{id * FZ}) \\ &\Rightarrow (q_1, \mathbf{*id}, *FZ) \\ &\Rightarrow (q_1, \mathbf{id}, FZ) \\ &\Rightarrow (q_1, \mathbf{id}, \mathbf{idZ}) \\ &\Rightarrow (q_1, \varepsilon, Z) \\ &\Rightarrow (q_2, \varepsilon, \varepsilon) \\ &\Rightarrow \textit{accept} \end{aligned}$$

Example Parse, ctd

Notes:

- The parse was guided through the non-determinism (by me, the *Oracle*) to always make the correct choice towards a successful parse.
- In practical terms states q_0 and q_2 and the initialisation and termination transitions are unnecessary.
- The stack always contains the *unmatched part of the sentential form*.

A Context-Sensitive Language

In case you're curious, a brief look at context-sensitive languages.

The following language is not context-free:

$$\{a^n b^n c^n \mid n \in \mathbb{N}\}$$

Intuitively, we can imagine a CFG generating either the ab pairs or the bc pairs, but this language requires us to keep the generation process in step, *in two different points in the sentential forms*.

A context-sensitive grammar is on the next slide. Each production is of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

That is, A can go to γ provided it is *in the context* $\alpha _ \beta$.

$$\begin{aligned}
S &\rightarrow aRc \\
R &\rightarrow aRT \mid b \\
bTc &\rightarrow bbcc \\
bTT &\rightarrow bbUT \\
UT &\rightarrow UU \\
UUc &\rightarrow VUc \rightarrow Vcc \\
UV &\rightarrow VV \\
bVc &\rightarrow bbcc \\
bVV &\rightarrow bbWV \\
WV &\rightarrow WW \\
WWc &\rightarrow TWc \rightarrow Tcc \\
WT &\rightarrow TT
\end{aligned}$$

The trick is to use non-terminals as markers and to shift and convert them to tokens. For example:

$$\begin{aligned}
S &\rightarrow aRc \\
&\rightarrow aaRTc \\
&\rightarrow aaaRTTc \\
&\rightarrow aaabTTc \\
&\rightarrow aaabbUTc \\
&\rightarrow aaabbUUc \\
&\rightarrow aaabbVUc \\
&\rightarrow aaabbVcc \\
&\rightarrow aaabbbccc
\end{aligned}$$

To see how this works observe that

$$S \xRightarrow{*} aa^n bT^n c \quad (n \geq 0)$$

$$bT^n \xRightarrow{*} bbU^n \quad (n \geq 2)$$

$$U^n c \xRightarrow{*} V^{n-1} cc \quad (n \geq 2)$$

$$bV^n \xRightarrow{*} bbW^n \quad (n \geq 2)$$

$$W^n c \xRightarrow{*} T^{n-1} cc \quad (n \geq 2)$$

$$bTc \rightarrow bbcc$$

$$bVc \rightarrow bbcc$$

CSGs and Automata

The automata that recognise CSGs have a tape memory, of length bounded by a linear function of the length of the input . . .