DISSERTATION TITLE, THAT CAN SPAN OVER

MULTIPLE LINES IF NEEDED

SUBTITLE (OPTIONAL)

By

First Last

Submitted to the Faculty of The Graduate School

of Augusta University in partial fulfillment

of the Requirements of the Degree of

Doctor of Philosophy

December

2024

Acknowledgements

This part serves two purposes.

To write the acknowledgments (as a "*Thank you note*"). You can look for inspiration Chisholm, 2015 if you need some.

To include a detailed summary of the work performed by other authors on published or accepted manuscripts used in the thesis / dissertation, if applicable.

Abstract

FIRST LAST

Dissertation Title, that can span over multiple lines if needed

(Under the direction of DR. ADVISOR)

The abstract must not exceed 350 words. It consists of a succinct summary of the thesis/dissertation and the conclusions reached. Opinions should be omitted.

KEYWORDS: Key1· Key2· A longer keyword

Table of Contents

References   **26**

Appendix A (Optional)   **27**

List of Tables

List of Figures

List of Listings

1 Introduction

This document is a guide on how to use it ("how meta!"), and its structure does not reflect the structure of a Thesis: you will need to erase (almost) all of its body and fill it with your own, organized in a coherent manner respectful of your reader's expectations, of your fields guidelines, and in agreement with your advisor. Note that The Graduate School recommends using APA (American Psychological Association, 2023), that this document is set-up to use it, and that using another style requires approval of the Dean of The Graduate School prior to writing the thesis or dissertation.

It is very important that you comply with all of the graduate school's policies (Augusta University's Graduate School, 2021). This template was carefully crafted with highest standards in mind, and respects all of the graduate schools requirements. You can find additional information in the "Thesis/Dissertation Preparation Booklet", in the "Electronic Theses and Dissertations (ETD)" help page, and in this template's repository.

Normally, what you can and cannot edit is clearly labeled in the source code, either at the beginning of the file, or with

⚠ Do not edit ⚠

Markdown only The comments applicable only to the markdown version of this document are indicated in such environments.

1.1 Title Levels

As indicated in the Section 3.6 of the koma-script manual, the `scrbook` class that is used for this document has access to 6 levels of titles:

```
1  \chapter{Test}
2  \section{Test}
3  \subsection{Test}
4  \subsubsection{Test}
5  \paragraph{Test}
6  \subparagraph{Test}
```

Only Chapters, Sections and Subsections will appear in the table of contents, by design.

Markdown only pandoc follows the ATX heading convention, and has # correspond to Chapter, and that increasing the number of # increases the level of heading.

1.1.1 Subsection

This is a subsection.

1.1.1.1 Subsubsection

This is a subsubsection.

1.1.1.1.1 Paragraph This is a paragraph.

2

1.1.1.1.1.1 Sub-Paragraph   This is a sub-paragraph.

1.2 Debugging

If this template does not "work" as expected, refer to the source code of the present document, read `aux/input.log` (md version) or `main.log` (tex version), then feel free to open an issue or reach out to caubert@augusta.edu.

2 References and Bibliography

Prepare your references using LaTeX's bibliography system BibTeX: this template uses by default biblatex, but you can alter this behaviour to use natbib if you prefer.

The references are stored in the .bib file located at `references/references.bib`: it contains examples of various entries. In computer science, a good source of bibliographical references is the dblp computer science bibliography. Make sure to include the digital object identifier (DOI) whenever possible, and note that this identifier can be used to obtain the corresponding .bib entry. Finally, you can "tidy" your .bib file using bibtex-tidy.

The list of references is automatically inserted in the list of references, p. 26. Use LaTeX's `\cite` command to insert references.

Links are only underlined *on screen* (and not in print), and with colors that should be colour-blind safe.

Markdown only You can use various syntaxes to integrate references: on top of LaTeX's `\cite` command, pandoc's `[@key]`, as well as more complex commands, such as `\↵ citeauthor` or pandoc's prefix, locator, and suffix, such as in `[see @key1, pp. 33↵ -35and *passim*; @key2, chap. 1]`.

You can insert hyperlinks in different ways, including hyperlinks to this document[1] using e.g. the link automatically added to all chapters, following the convention described in pandoc's manual.

---

[1] You may note that the footnote number is itself a link.

3 Writing Mathematics

LaTeXcan be used to render complex mathematics expressions in a relatively simple manner. Note that thanks to XeLaTeX, you can insert mathematical symbols directly in unicode, as follows: $\forall y \in \mathbb{N}, \exists x \in \mathbb{N}, y = x^2$, but of course you can always fall back to usual LaTeX notation, using e.g. \forall to produce $\forall$.

You can add additional unicode symbols that may not be supported by this template or its font using the model

---

1 \newunicodechar{<unicode symbol>}{\ensuremath{<latex command>}}

(in head_c.tex in the markdown version), in this case additionally forcing the symbol <↵ unicode symbol> to be rendered in math mode using \ensuremath.

3.1 Theorem, Proof, and Others Environments

Markdown only You can state e.g. theorems and proofs using pandoc's built-in "*Definition list*", that are rendered as description environments in LaTeX.

Theorem Every $n \in \mathbb{N}, n > 1$ has a unique prime factorization.

Proof Carl Friedrich Gauss told me so. □

To insert numbered theorems, definitions, and the like, and be able to reference them or add automatically the "qed" (□) symbol, you need to use LaTeX's theorem environment, label

commands, etc. Note that, by default, proofs are unnumbered environments, but that there
are ways to reference them if you want to.

Theorem 1 (Pythagoras theorem). $\forall a, b, c, a^2 + b^2 = c^2$.

Proof. Proving Theorem 1 is not that easy. $\square$

Markdown only If you would rather keep the "pure" markdown syntax but improve pandoc using a filter, you can look at the pandoc filter "statement" and its discussion on related filters, but it may be more difficult to install and use properly.

3.2 Formal Proofs

You can easily represent formal proofs using LaTeX's ebproof or bussproof packages:

$$
\cfrac{A \vee B \quad \begin{matrix}[A]\\ \vdots\\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots\\ C\end{matrix}}{C} \vee\mathrm{E}
$$

4 Inserting PDFs, Figures, Tables and (Code) Listings

4.1 Inserting PDFs

PDF documents can be inserted using `pdfpages`'s `\includepdf` command. For commodity, a `\modifiedincludepdf` is provided:

```
1  \modifiedincludepdf{options for includepdf}%
2      {label}%
3      {full path to the document}%
4      {title of the document}%
5      {"level" (e.g., section, subsection, etc.)}
```

Note that using `label.x` will refer to the page `x` of the inserted document (starting with 1): refer to the source code of this current document for an example usage. We insert in the following pages (pp. 8–16) an article as an example of PDF insertion.

Simulation of Two-Way Pushdown Automata Revisited

Robert Glück

DIKU, Dept. of Computer Science, University of Copenhagen

glueck@acm.org

Dedicated to David A. Schmidt on the Occasion of his 60th Birthday

The linear-time simulation of *2-way deterministic pushdown automata* (2DPDA) by the Cook and
Jones constructions is revisited. Following the semantics-based approach by Jones, an interpreter
is given which, when extended with random-access memory, performs a linear-time simulation of
2DPDA. The recursive interpreter works without the dump list of the original constructions, which
makes Cook's insight into linear-time simulation of exponential-time automata more intuitive and the
complexity argument clearer. The simulation is then extended to *2-way nondeterministic pushdown
automata* (2NPDA) to provide for a cubic-time recognition of context-free languages. The time
required to run the final construction depends on the degree of nondeterminism. The key mechanism
that enables the polynomial-time simulations is the sharing of computations by memoization.

1 Introduction

We revisit a result from theoretical computer science from a programming perspective. Cook's surprising
theorem [4] showed that *two-way deterministic pushdown automata* (2DPDA) can be simulated faster on
a random-access machine (in linear time) than they may run natively (in exponential time). This insight
was utilized by Knuth [8] to find a linear-time solution for the left-to-right pattern-matching problem,
which can easily be expressed as a 2DPDA:

> "*This was the first time in Knuth's experience that automata theory had taught him how to
> solve a real programming problem better than he could solve it before.*" [8, p. 339]

Cook's original construction in 1971 is obscured by the fact that it does not follow the control flow of
a pushdown automaton running on some input. It traces all possible flows backward thereby examining
many unreachable computation paths, which makes the construction hard to follow. Jones clarified the
essence of the construction using a semantics-based simulator that interprets the automaton in linear
time while following the control flow forward thereby avoiding unreachable branches [6]. The simulator
models the *symbol stack* of the automaton on its *call stack* using recursion in the meta-language and
maintains a local list of surface configurations (dump list) to record their common terminator in a table
when a pop-operation is simulated.

We follow Jones' semantics-based approach and give a simplified recursive simulator that does not
require a local dump list and captures the essence of Cook's speedup theorem in a (hopefully) intuitive
and easy to follow form. Furthermore, we then extend the construction from a simulation of deterministic
automata to a simulation of *two-way nondeterministic pushdown automata* (2NPDA). The simulations
are all realized by deterministic computation on a random-access machine.

Even though some time has passed since the theorem was originally stated, it continues to inspire
studies in complexity theory and on the computational power of more practical programming paradigms,

R. Glück 251

such as subclasses of imperative and functional languages (*e.g.* [2, 3, 7, 10]). It therefore appears worthwhile to capture the computational meaning of this classic result in clear and simple terms from a programming perspective. It is hoped that the progression from simple interpreters to simulators with memoization and termination detection makes these fundamental theoretical results more accessible.

We begin with a simple interpreter for two-way deterministic pushdown automata (Sect. 2) that we extend to simulate deterministic PDA in linear time (Sect. 3). We then introduce a nondeterministic choice operator (Sect. 4) and show the simulation of nondeterministic PDA (Sect. 5).

2   Deterministic PDA Interpreter

A *two-way deterministic pushdown automaton* (2DPDA) consists of a finite-state control attached to a stack and an input tape with one two-way read-only head [4]. The state p, the symbol read at head position i, and the symbol A on top of the stack determine the next action for a given tape, which is the automaton's input. Only when the stack top is popped does the symbol below the top become relevant for the following computation. The set of states, the set of input symbols and the set of stack symbols are fixed for an automaton. A transition function chooses the next action depending on the current *surface configuration* $c = (p, i, A)$, shortly referred to as *configuration*. The *instantaneous description* $(c, stack\text{-}rest)$ of an automaton includes the current configuration c and the stack below the top symbol A.

The automaton can **push** and **pop** stack symbols, and perform an operation **op** that modifies the current configuration without pushing or popping (*e.g.*, move to a new tape position). The stack bottom and the left and right tape ends are marked by distinguished symbols. The head position i in a configuration (p, i, A) is always kept within the tape bounds and one can determine an empty stack. The automaton answers decision problems. It is said to *accept* an input if, when started in initial state p_0 with an empty stack and the head at the left endmarker, it terminates with **accept**, an empty stack and the head at the right endmarker. It can just **halt** with an empty stack without accepting an input. In the exposition below we tacitly assume some fixed input tape.

Termination.   A configuration in which a pop-operation occurs is a *terminator* [4]. Every configuration c in a terminating computation has a unique terminator, that is the earliest terminator reached from c that returns the stack *below the height* at c. This case is illustrated below (i): d is the terminator of c. Terminator d can be viewed as the result of configuration c. Configuration c will always lead to d regardless of what is on the stack below.

If a configuration c is met again *before* the terminator is reached, which means that the stack never returned below the level at which c occurred for the first time, then the automaton is in an *infinite loop*. The second occurrence of c will eventually lead to a third occurrence of c, ad infinitum. The only two possible situations are illustrated below: either c repeats at the same level of the stack (ii) or at a higher level after some stack-operations have been performed (iii). In both cases, the contents of the stack below c (shaded) is untouched and irrelevant to the computation: c will always lead to an infinite loop.
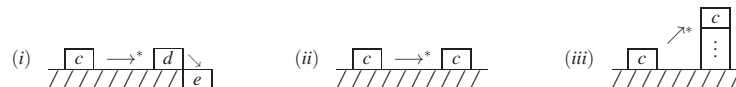
Running Time.   The *number of configurations* that an automaton can enter during a computation depends on the input tape. The states and symbols are fixed for an automaton. The number of head positions

A paper proving concisely a result in automata theory that helped solve a real programming problem Glück, 2013 (p. 3 / 9)

4 Inserting PDFs, Figures, Tables and (Code) Listings

252   *Simulation of Two-Way Pushdown Automata Revisited*

on the input tape is bounded by the *length of the input tape*. The number of configurations is therefore linear in the length of the input tape, $n = O(|tape|)$. We remark that the number of configurations of an automaton with k independent heads on the input tape is $n = O(|tape|^k)$. The k head positions are easily accommodated by configurations of the form $c = (p, i_1, \ldots, i_k, A)$. An automaton can carry out an *exponential number of steps* before it terminates. For example, an automaton that during its computation forms all stacks consisting of n zeros and ones takes $O(2^n)$ steps.

Interpreter. Figure 1 shows the interpreter for 2DPDA written in the style of an imperative language with recursion and call-by-value semantics. The interpreter Int can be run on a *random-access machine* (RAM). A call $Int(c) = d$ computes the terminator d of a configuration c, where $pop(d)$. There is no symbol stack and no loop in the interpreter. All operations are modeled on the *call stack* of the implementation language by recursive calls to the interpreter. A recursive call takes constant time, thus a call stack just adds a constant-time overhead compared to a data stack. Statements **accept** and **halt** stop the interpreter and report whether the input was accepted or not. The automaton is assumed to be correct and no special checks are performed by the interpreter. We will now discuss the interpreter in more detail. It is the basis for the three interpreters and simulators in the following sections.

In the interpreter we abstract from the concrete push-, op- and pop-operations. We define predicates $push(c)$, $op(c)$, $pop(c)$, $accept(c)$, $halt(c)$ to be true if a configuration c causes the corresponding operation in the automaton. Their actual effect on a configuration is not relevant as long as the next configuration can be determined by the built-in operations next and follow. We let $next(c)$ be the operation that yields in one step the next configuration, if $op(c)$ or $push(c)$, and $follow(c,d)$ be the operation that yields in one step the next configuration given c and d, if $pop(d)$.[1] Each of these operations takes constant time, including next and follow that calculate the next configuration.

In case a configuration c causes a pop-operation, that is $pop(c)$ is true in the cond-statement (Fig. 1), c is a terminator and the interpreter returns it as result. If a configuration c causes a push-operation, that is $push(c)$ is true, first the terminator of the next configuration is calculated by $Int(next(c)) = d$. The terminator always causes a pop-operation and interpretation continues at configuration $follow(c,d)$ which follows from c and terminator d. In case $op(c)$ is true, that is the operation neither pushes nor pops, the terminator of c is equal to the terminator $Int(next(c))$ of the next configuration.

The effect of the operations on the configurations and the call stack can be summarized as follows.

$$c = \boxed{\begin{array}{c} (p,i,A) \\ \hline \cdots \end{array}} \quad \xrightarrow{\text{push}(c)} \quad \boxed{\begin{array}{c} (q,j,B) \\ \hline (p,i,A) \\ \hline \cdots \end{array}} = next(c)$$

$$c = \boxed{\begin{array}{c} (p,i,A) \\ \hline \cdots \end{array}} \quad \xrightarrow{\text{op}(c)} \quad \boxed{\begin{array}{c} (q,j,B) \\ \hline \cdots \end{array}} = next(c)$$

$$\begin{array}{c} d = \boxed{(q,j,B)} \\ c = \boxed{\begin{array}{c} (p,i,A) \\ \hline \cdots \end{array}} \end{array} \quad \xrightarrow{\text{pop}(d)} \quad \boxed{\begin{array}{c} (r,k,C) \\ \hline \cdots \end{array}} = follow(c,d)$$

---

[1] The conventional 'pop' just removes the top symbol from the stack. Our generalization that defines the next configuration by $follow(c,d)$ does not affect the complexity arguments later and is convenient from a programming language perspective.

A paper proving concisely a result in automata theory that helped solve a real programming problem Glück, 2013 (p. 4 / 9)

4 Inserting PDFs, Figures, Tables and (Code) Listings

```
procedure Int(c: conf): conf;
   cond
      push(c):    d := Int(follow(c,Int(next(c))));
      op(c):      d := Int(next(c));
      pop(c):     d := c;
      halt(c):    halt;
      accept(c):  accept;
   end;
   return d
```

Figure 1: A recursive interpreter for deterministic PDA.

A push-operation may, for example, push a constant symbol B onto the stack or duplicate the current top A. Likewise, an op-operation may replace the current top A by a new top B, but without pushing or popping the stack, and move the tape head by changing position i into j. A pop-operation may just remove the stack top A to uncover B below or replace the uncovered symbol by a symbol C depending on A and B. The abstract pop-operation covers many common binary stack-operations familiar from stack programming languages (*e.g.*, it may choose from symbols A and B the one that is smaller according to some order). Depending on the concrete set of binary operators and stack symbols this allows to express a number of interesting functions as pushdown automata.

Properties. The body of the interpreter contains no loop, only sequential statements. The time it takes to execute each of the statements is bounded by a constant (ignoring the time to evaluate a recursive call to a result). No side-effects are performed and no additional storage is used except for the local variable d. Even though written in an imperative style, the interpreter is purely functional. It terminates if and only if the pushdown automaton terminates on the same input. The correctness of the interpreter should be evident as it merely interprets the automaton one step at a time. Note the simplicity of the construction by recursively calling the interpreter for each action of the automaton. Also an op-operation that does not change the height of the symbol stack converts into a (tail-recursive) call on the call stack.

In a terminating computation, no call Int(c) can lead to a second call Int(c) as long as the first call has not returned a result, which means that it is still on the call stack. If a second call Int(c) occurs while the first one is still on the call stack, the interpreter is in an infinite recursion.

As a consequence, in a terminating computation the height of the call stack is bounded by n, the number of configurations, and the same call stack cannot repeat during the interpretation. After exhausting all possible call stacks of height up to n, that is all permutations of up to n configurations, the interpreter must terminate, that is within $O(n^n)$ steps. The interpreter can have a running time exponential in the number of configurations.

3   Linear-Time Simulation of Deterministic PDA

The 2DPDA-interpreter in Fig. 1 is purely functional and has no persistent storage. Each time the terminator d of a configuration c is computed and the same configuration is reached again, the terminator has to be recomputed by a call Int(c), which means the entire subcomputation is repeated. To store known terminators and to share them across different procedure invocations, we extend the interpreter with *memoization* [9]. This straightforward extension gives linear-time simulation of 2DPDA. The sharing of terminators is the reason why Cook's speedup theorem works.

254   *Simulation of Two-Way Pushdown Automata Revisited*

```
procedure Sim(c: conf): conf;
  if defined(T[c]) then return T[c];   /* find shortcut */
  cond
    push(c):    d := Sim(follow(c,Sim(next(c))));
    op(c):      d := Sim(next(c));
    pop(c):     d := c;
    halt(c):    halt;
    accept(c):  accept;
  end;
  T[c] := d;                           /* memoize result */
  return d
```

Figure 2: A linear-time simulator for deterministic PDA.

RAM extension. Figure 2 shows the interpreter with memoization, called *simulator*. It works in the same way as the interpreter except that each time before a call Sim(c) returns the terminator d of c, the terminator is stored in a table T by assignment T[c] := d. Next time the terminator is needed, it can be retrieved from T, avoiding its recomputation. Terminators are now shared dynamically at run time and over the entire simulation. Table T can be implemented as a one-dimensional array indexed by configurations and can hold one terminator for each of the n configurations that can occur during a computation. All table entries are initially undefined. It is easy to see that the shortcut (if-statement) and the memoization (table assignment) do not change the result of the automaton. Storing and retrieving a terminator takes constant time on a RAM (see Cook for a charged RAM model instead of a unit-cost model [4]). An "automatic storage management" also means that many terminators are recorded during a computation that are not needed later, but we shall see that this does not affect the linearity argument. A more thorough analysis would surely reveal that memoization points are only required at a few places in an automaton (*cf.* [3, 10]).

Linear-time simulation. In a terminating computation, before a second call Sim(c) is made, the first call must have returned and stored the terminator d of c at T[c]. Once the terminator is known, it need not be recomputed and can be fetched from the table. Hence, the cond-statement, which is guarded by a lookup in T, is executed *at most once* for any c. Recursive calls to the simulator occur only from within the cond-statement, namely one call if op(c) and two calls if push(c). Consequently, Sim can be called at most $2n$ times, where n is the number of possible configurations. This also limits how often the if-statement guarding the cond-statement is executed. Hence, the total number of execution steps during a terminating simulation is bounded linearly by n. Recall that n is linear in the length of the input tape, $n = O(|tape|)$. This concludes the argument for the linear-time simulation of a 2DPDA on a RAM.

Discussion. Deterministic pushdown automata are the accepting device for *deterministic context-free languages*. More precisely, they are exactly recognized by *1-way* deterministic pushdown automata (1DPDA), that is, deterministic pushdown automata that never move their head to the left on the input. The LR grammar of a deterministic context-free language is easy to convert into a 1DPDA (*e.g.* [5]). Thus, recognition of this subclass of context-free languages using the memoizing simulator Sim (Fig. 2) takes at most linear time (as does the classic LR-parsing algorithm by Knuth). In the following we extend the simulator to recognize all context-free languages in cubic time.

The method by Aho *et al.* [1] requires $O(n^2)$ for simulating 2DPDA, a result which was then strength-

A paper proving concisely a result in automata theory that helped solve a real programming problem Glück, 2013 (p. 6 / 9)

4 Inserting PDFs, Figures, Tables and (Code) Listings

R. Glück 255

```
procedure Int(c: conf): confset;
  if visited(T[c]) then return {};        /* detect infinite branch */
  T[c] := Visited;                         /* mark configuration */
  cond
    push(c):    d := ⋃ Int(follow(c,e)) where e ∈ Int(next(c));
    op(c):      d := Int(next(c));
    choose(c):  d := Int(nextleft(c)) ∪ Int(nextright(c));
    pop(c):     d := {c};
    halt(c):    d := {};
    accept(c):  accept;
  end;
  T[c] := Undef;                           /* unmark configuration */
  return d
```
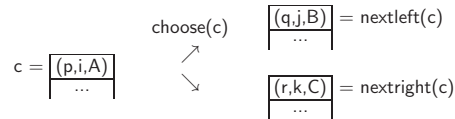
Figure 3: A recursive interpreter for nondeterministic PDA.

ened to $O(n)$ by Cook [4]. Both methods work bottom-up. In contrast, the simulator Sim works top-down following the forward control flow as does the one by Jones [6]. It clearly shows that the key mechanism that turns a recursive pushdown interpreter into a linear-time simulator is memoization.

4   Interpretation of Nondeterministic PDA

In a two-way *nondeterministic* pushdown automaton (2NPDA) the computation path is not uniquely determined. A deterministic automaton can be made *nondeterministic* by introducing an operation **choose** that allows the automaton to select any of two computation paths in a configuration c (*cf.* [7]). This means that a configuration no longer has a unique terminator, but a set of possible terminators. We let nextleft(c) and nextright(c) be the abstract operations that yield in one step the two next configurations that are possible if choose(c). For simplicity, the new operation can neither push nor pop stack symbols. With a choose-operation two transitions are possible:

$$c = \boxed{\begin{array}{c}(p,i,A)\\ \cdots\end{array}} \quad \begin{array}{c}\text{choose}(c)\\ \nearrow\\ \searrow\end{array} \quad \begin{array}{c}\boxed{\begin{array}{c}(q,j,B)\\ \cdots\end{array}} = \text{nextleft}(c)\\[1em]\boxed{\begin{array}{c}(r,k,C)\\ \cdots\end{array}} = \text{nextright}(c)\end{array}$$

A nondeterministic automaton is said to *accept* an input if it has at least one accepting computation when started in the initial state p_0 with an empty stack and the head at the left tape end. It has the ability to guess the right choice that leads to the shortest accepting computation. In an interpreter this "angelic nondeterminism" can be thought of as searching through a tree of all possible computation sequences, some of which may be infinite or non-accepting, to find at least one accepting sequence. Branching in the computation tree is due to nondeterministic choose-operations in the automaton.

Interpreter.   The interpreter for *nondeterministic* PDA that can be run on a RAM is shown in Fig. 3. Two main changes to the original interpreter in Fig. 1 are necessary to accommodate the "guessing": (1) a *set of terminators* instead of a single terminator is returned, and (2) a *termination check* ("seen before") that stops interpretation along an infinite computation sequence. We detail the two modifications below.

13

256 *Simulation of Two-Way Pushdown Automata Revisited*

1. *Terminator sets*: A choose-operation requires the union of the terminator sets obtained from the two next configurations, nextleft(c) and nextright(c). In case of a push-operation, and this is the most involved modification, each configuration e in the terminator set obtained by the inner call Int(next(c)) must be followed by an outer call. The big set union used for this purpose is a shorthand for a while-loop over the inner terminator set. A pop-operation now returns a singleton set {c} instead of c. Finally, instead of making a full stop at a halt-operation, an empty set is returned in order not to miss an accepting computation along another possible branch.

2. *Termination check*: As discussed before, non-termination occurs when the interpreter is called a second time with the same configuration c as argument while the first call has not yet returned. This situation can be detected by marking c in a table when a call Int(c) is made and unmarking c when the call returns. If a call with a marked c as argument is made, an infinite computation is detected and the interpreter returns an empty terminator set. The same table T as before can be used, but can now hold the additional value Visited. Initially all table entries are set to Undef.

The cardinality of a terminator set is bounded by n, the number of configurations that can occur in a computation. The most costly set operation in the interpreter is the union of terminator sets. Assuming a suitable choice of data structures, a union operation takes time linear in the total cardinality of the sets, that is the union of two sets with cardinalities u and v takes time $O(u+v)$. All remaining set-operations needed in the interpreter are straightforward and take constant time: creating a set (empty, singleton), and picking and removing an arbitrary element from a set (in the set comprehension). In the discussion below we assume such an implementation of the set operations.[2]

A choose-operation, which unites two terminator sets each of cardinality up to n, takes linear time $O(n)$. A push-operation, where the inner call Int(next(c)) returns a set of at most n terminators, each of which, when followed by the outer call Int(follow(c,e)), can again return a set of up to n terminators, requires the union of n sets each of cardinality up to n, which then takes quadratic time $O(n^2)$. This is the most expensive set-operation in the cond-statement.

In the case of a deterministic automaton, that is, an automaton *without* choose-operation, the new interpreter in Fig. 3 operates with singleton sets only, and the set-operations introduce at most a constant-time overhead compared to the original interpreter in Fig. 1. This is useful because the new interpreter "falls back" to its original behavior and, except for a constant time overhead in the new interpreter, there is no penalty in using it to run deterministic PDA and, as an extra benefit, it always terminates.

There is a major pitfall. If a nondeterministic automaton is left-recursive, then the termination check may stop left-recursion too early and miss useful branches contributing to a terminator set. In the case of 1NPDA there always exists a non-left-recursive version (presumably the same for 2NPDA). Alternatively, one might bound the unfolding of a left-recursion in terms of the input assuming some normal-form automaton (the termination check in Fig. 3 limits left-recursion unfolding to one).

5   Cubic-Time Simulation of Nondeterministic PDA

To turn the new interpreter (Fig. 3) into a fast simulator (Fig. 4) we use the same memoization method as in Sect. 3. The use of table T parallels its use in the deterministic case except that for each of the n possible configurations the table can now hold a set of up to n terminators and the value Visited. The body of the simulator is again guarded by an if-statement (first line) that returns the terminator set of a

---

[2] A straightforward implementation of such a set data structure might be a Boolean array of length n to indicate membership of a configuration c in a set together with an unsorted list of all configurations contained in that set.

R. Glück 257

```
procedure Sim(c: conf): confset;
  if defined(T[c]) then return T[c];   /* find shortcut  */
  if visited( T[c]) then return { };    /* detect infinite branch */
  T[c] := Visited;                      /* mark configuration */
  cond
    push(c):    d := ⋃ Sim(follow(c,e)) where e ∈ Sim(next(c));
    op(c):      d := Sim(next(c));
    choose(c):  d := Sim(nextleft(c)) ∪ Sim(nextright(c));
    pop(c):     d := { c };
    halt(c):    d := { };
    accept(c):  accept;
  end;
  T[c] := d;                            /* memoize result */
  return d
```

Figure 4: A cubic-time simulator for nondeterministic PDA.

configuration c, if it is available in table T. Otherwise, and if no infinite computation path is detected, c is marked as Visited in T and its terminator set is computed.

Before returning, terminator set d of c is stored in T, which overwrites the mark Visited. The cond-statement is executed at most once for each configuration. The mark Visited is only needed the first time the procedure is called, when the table does not yet contain a terminator set for c. Thus, the same table can be used for marking configurations and for storing their terminator sets. A terminator set may be empty if none of the branches rooted in c is accepting. Otherwise, the interpreter is unchanged.

Cubic-time simulation. As before, the cond-statement is executed at most once for each of the n configuration due to the guards at the beginning of Sim. Up to $n+1$ calls to Sim may occur in the case of a push-operation, namely one inner call and at most n outer calls, one for each $e \in$ Sim(next(c)). Hence, Sim can be called at most $O(n^2)$ times during a simulation. This also limits how often the if-statements guarding the cond-statement are executed.

In the cond-statement, as before, the simulation of the op-, pop-, halt-, accept-operations takes constant time, $O(1)$. The union of two sets of at most n terminators in case of a choose-operation may take linear time, $O(n)$. The union of the terminator sets in a push-operation is the most costly operation and may take quadratic time, $O(n^2)$. A push is simulated at most once per execution of a cond-statement, which is at most n times. Hence, the total number of execution steps during a simulation is cubic in the number of configurations, $O(n^3)$. Recall that n is linear in the length of the input tape, $n = O(|tape|)$. This ends the argument for the cubic-time simulation of (non-left-recursive) 2NPDA on a RAM.

Discussion. We observe that the "complexity generator" in the cond-statement is *not* the choose-operation, even though it introduces two computation branches, rather the handling of up to n continuations and the union of their terminator sets in case of a push-operation. If the cardinality of each terminator set that can occur during a simulation is bounded by a *constant*, that is, not dependent on the input, the simulation time is *linear* in the input as before. Deterministic automata, where the cardinality of each terminator set is at most one, and a class of nondeterministic automata, where the cardinality is bounded by some k, are all simulated in linear time by Sim. The top-down method is useful because the same simulator runs them in the time corresponding to their degree of nondeterminism.

15

258   *Simulation of Two-Way Pushdown Automata Revisited*

One-way nondeterministic pushdown automata (1NPDA) are the accepting device for *context-free languages*. Every context-free language has a grammar without left recursion and it is straightforward to convert the grammar into a 1NPDA. This means that recognition of context-free languages using the simulator (Fig. 4) has the same worst-case time complexity as the classic parsing algorithms that can handle the full class of context-free languages (Earley, Cocke-Younger-Kasami), that is $O(|string|^3)$. As discussed before, the performance of the simulator is determined by the degree of nondeterminism in the automaton. Recognition of deterministic context-free languages using the simulator takes, again, at most linear time. In practice, of course, specialized parsing algorithms will have better run times (due to the constant term hidden in the *O*-notation) and use less space than the recursive simulator. Again, the mechanism that enables polynomial-time simulation is the sharing of computations by memoization.

References

[1] Alfred V. Aho, John E. Hopcroft & Jeffrey D. Ullman (1968): *Time and tape complexity of pushdown automaton languages*. Information and Control 13(3), pp. 186–206, doi:10.1016/S0019-9958(68)91087-5.

[2] Torben Amtoft & Jesper Larsson Träff (1992): *Partial memoization for obtaining linear time behavior of a 2DPDA*. Theoretical Computer Science 98(2), pp. 347–356, doi:10.1016/0304-3975(92)90008-4.

[3] Nils Andersen & Neil D. Jones (1994): *Generalizing Cook's transformation to imperative stack programs*. In J. Karhumäki, H. Maurer & G. Rozenberg, editors: *Results and Trends in Theoretical Computer Science*, LNCS 812, Springer-Verlag, pp. 1–18, doi:10.1007/3-540-58131-6_33.

[4] Stephen A. Cook (1972): *Linear time simulation of deterministic two-way pushdown automata*. In C. V. Freiman, J. E. Griffith & J. L. Rosenfeld, editors: *Information Processing 71*, North-Holland, pp. 75–80.

[5] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[6] Neil D. Jones (1977): *A note on linear time simulation of deterministic two-way pushdown automata*. Information Processing Letters 6(4), pp. 110–112, doi:10.1016/0020-0190(77)90022-9.

[7] Neil D. Jones (1997): *Computability and Complexity: From a Programming Language Perspective*. Foundations of Computing, MIT Press, Cambridge, Massachusetts.

[8] Donald E. Knuth, James H. Morris & Vaughan R. Pratt (1977): *Fast pattern matching in strings*. SIAM Journal on Computing 6(2), pp. 323–350, doi:10.1137/0206024.

[9] Donald Michie (1968): *"Memo" functions and machine learning*. Nature 218(5136), pp. 19–22, doi:10.1038/218019a0.

[10] Torben Æ. Mogensen (1994): *WORM-2DPDAs: an extension to 2DPDAs that can be simulated in linear time*. Information Processing Letters 52(1), pp. 15–22, doi:10.1016/0020-0190(94)90134-1.

4.2 Inserting Figures

Markdown only You can easily insert images and figures using Pandoc, as in Figure 1, a painting by Jérôme Minard under copyleft.

Figure 1: *D'un autre âge*

4.3 Inserting Tables

Markdown only You can write tables using pandoc's syntax*es*, as in Tables 1, 2 and 3 (all borrowed from https://www.flutterbys.com.au/stats/tut/tut17.3.html).

17

Table 1: The price of categories

Column A	Column B	Column C
Category 1	High 95.00	100.00
High		
Category 2	High 82.50	80.50
High		

Table 2: Illustrating how to align entries in a table

Default	Left	Center	Right
High	Cat 1	A	100.00
High	Cat 2	B	85.50
Low	Cat 3	C	80.00

Table 3: The price and advantages of fruits

Fruit	Price	Advantages
Bananas	$1.34	• built-in wrapper
		• bright color
Oranges	$2.10	• cures scurvy
		• tasty

4.4 Inserting Code Listings

Code is displayed using the listings package. Check the "Table 1: Predefined languages" of the listings package documentation to see the list of supported languages by default.

Markdown only Your can display code using various possible syntaxes.

4 Inserting PDFs, Figures, Tables and (Code) Listings

As a fenced block:

```
1  public class HelloWorld {
2      public static void main(String[] args) {
3          System.out.println("Hello, World");
4      }
5  }
```

In a figure, as in Listings 1, 2 or 3 (that uses respectively the backtick, the tildes, and listinputlisting to display the code – this latter option allows to load a file directly).

```
1  (** Courtesy of https://coq.inria.fr/a-short-introduction-to-coq. **)
2  Inductive even : N → Prop :=
3  | even_0 : even 0
4  | even_S n : odd n → even (n + 1)
5  with odd : N → Prop :=
6  | odd_S n : even n → odd (n + 1).
```

Listing 1: An inductive definition in Coq

```
1  # Courtesy of https://stackoverflow.com/a/2188369
2  for num in {000..2}; do echo "$num"; done
```

Listing 2: How to use braces ({and }) in bash

```
1  /* Courtesy of Brian Kernighan and https://en.wikipedia.org/wiki/%22Hello,↵
       _World!%22_program#C */

2  #include <stdio.h>

3  int main(void)

4  {

5      printf("Hello, world\n");

6      return 0;

7  }
```

Listing 3: "*Hello World*" in C

5 Landscape Pages

You can obtain landscape pages using the landscape package in LaTeX.

Markdown only This feature is not accessible in pure markdown: if you want to have

landscape pages, you need to use LaTeXcommands in your document.

Note that the drawing presented in Figure 2 was obtained using LaTeX's package tiKz, and
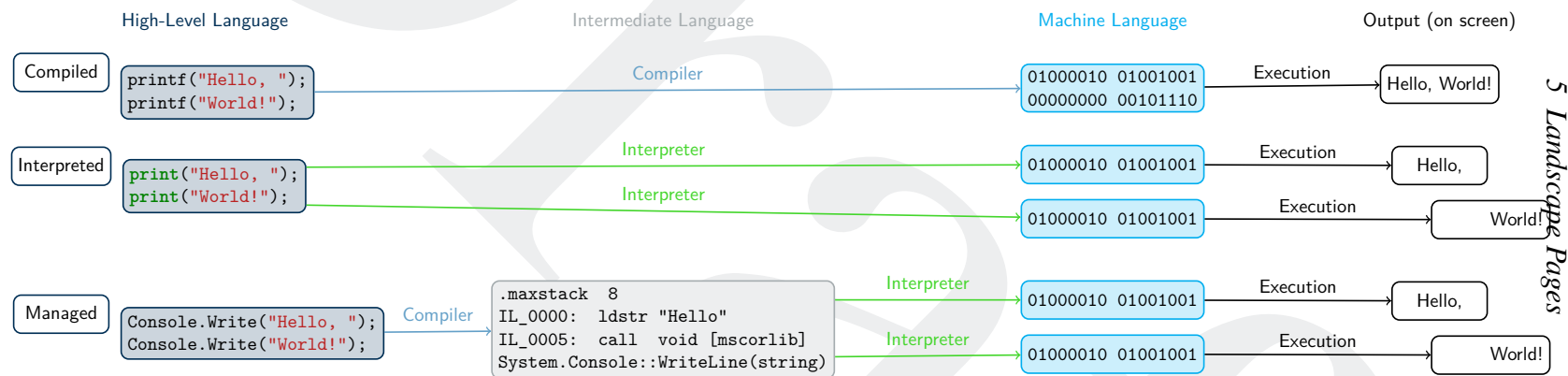
that the source code is shared in the pictures folder.

High-Level Language	Intermediate Language	Machine Language	Output (on screen)

Compiled
```
printf("Hello, ");
printf("World!");
```
Compiler
```
01000010 01001001
00000000 00101110
```
Execution → Hello, World!

Interpreted
```
print("Hello, ");
print("World!");
```
Interpreter
```
01000010 01001001
```
Execution → Hello,

Interpreter
```
01000010 01001001
```
Execution → World!

Managed
```
Console.Write("Hello, ");
Console.Write("World!");
```
Compiler
```
.maxstack  8
IL_0000:  ldstr "Hello"
IL_0005:  call  void [mscorlib]
System.Console::WriteLine(string)
```
Interpreter
```
01000010 01001001
```
Execution → Hello,

Interpreter
```
01000010 01001001
```
Execution → World!

Figure 2: Difference between programming languages (simplified)

6 Margins and Fonts

6.1 Margins

The margin have been set to fit the graduate school's requirements to:

_____

Actual page layout values.

_____

\paperheight = 11.00215in	\paperwidth = 8.50166in
\hoffset = 0in	\voffset = 0in
\evensidemargin = 0.50009in	\oddsidemargin = 0.50009in
\topmargin = 0in	\headheight = 0in
\headsep = 0in	\textheight = 9.00177in
\textwidth = 6.00117in	\footskip = 0.70236in
\marginparsep = 0in	\marginparpush = 0in
\columnsep = 0in	\columnseprule = 0in
1em = 0.16608in	1ex = 0.07472in

_____

Please, do not change those values.

6.2 Fonts

6.2.1 Body

The font used in the body of the document is "TeX Gyre Termes Font Family", which is an extension of the standard Times New Roman that is free for commercial use, and can be freely distributed. It is set to 12pt in all of the document, and adjusted when needed to the appropriate size (particularly in the cover page, where most attributes need to be set at 16pts).

The "usual" correspondence between points and LaTeX commands is as follows:

tiny is equivalent to 6pt

scriptsize is equivalent to 8pt

footnotesize is equivalent to 10pt

small is equivalent to 10.95pt

normalsize is equivalent to 12pt

large is equivalent to 14.4pt

Large is equivalent to 17.28pt

LARGE is equivalent to 20.74pt

huge is equivalent to 24.88pt

6.2.2 Symbols

For better unicode support, the Symbola font is also used. Starting with version 11, the licence of this font is too restrictive for non-personal use. As a consequence, users are asked to make sure they do not use a version greater than v.10.24, which is "free for any use" and archived on-line.

By default, the following symbols, not available in the TeX Gyre Termes Font Family, are displayed using Symbola: 🔐, ✗, ⚠, ❓, ⏩, ℕ, ℤ, ✔, ↵, ⏎, ♡, ℝ □. To declare other unicode symbols as having to be displayed using the Symbola font, use

```
1  \newunicodechar{<unicode symbol>}{\symb <unicode symbol>}
```

(in `head_c.tex` in the markdown version), so that `<unicode symbol>` will be rendered using the Symbola font.

References

American Psychological Association. (2023, December). *Publication manual of the american psychological association* (7th ed.).

Augusta University's Graduate School. (2021, July). *Forms, policies, and procedures*. Retrieved November 1, 2021, from https://www.augusta.edu/gradschool/student-resources.php

Chisholm, J. (2015, October). Writing acknowledgements: Saying "thank you". In *Cetl 8723: Writing for international graduate students*. https://esl.gatech.edu/sites/default/files/LI/li-how_to_write_acknowledgements_in_a_dissertation.pdf

Glück, R. (2013, September). Simulation of two-way pushdown automata revisited. In A. Banerjee, O. Danvy, K. Doh, & J. Hatcliff (Eds.), *Semantics, abstract interpretation, and reasoning about programs: Essays dedicated to David A. Schmidt on the occasion of his sixtieth birthday, Manhattan, Kansas, USA, 19-20th september 2013* (pp. 250–258, Vol. 129). https://doi.org/10.4204/EPTCS.129.15

Appendix A (Optional)

Insert here protocols, figures not included, larger listings, etc.