Generating Context-Free Grammars using Classical Planning

Javier Segovia-Aguas
Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
javier.segovia@upf.edu

Sergio Jiménez
Computing and Information Systems
University of Melbourne
Parkville, Victoria 3010, Australia
sjimenez@unimelb.edu.au

Anders Jonsson
Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
anders.jonsson@upf.edu

Abstract

This paper presents a novel approach for generating Context-Free Grammars (CFGs) from small sets of input strings (a single input string in some cases). Our approach is to compile this task into a classical planning problem whose solutions are sequences of actions that build and validate a CFG compliant with the input strings. In addition, we show that our compilation is suitable for implementing the two canonical tasks for CFGs, *string production* and *string recognition*.

1 Introduction

A formal grammar is a set of symbols and rules that describe how to form the strings of certain formal language. Usually two tasks are defined over formal grammars:

- *Production*: Given a formal grammar, generate strings that belong to the language represented by the grammar.
- *Recognition* (also known as *parsing*): Given a formal grammar and a string, determine whether the string belongs to the language represented by the grammar.

Chomsky defined four classes of formal grammars that differ in the form and generative capacity of their rules [Chomsky, 2002]. In this paper we focus on *Context-Free Grammars* (CFGs), where the left-hand side of a grammar rule is always a single non-terminal symbol. This means that the symbols on the left-hand side of CFG rules do not appear in the strings that belong to the corresponding language.

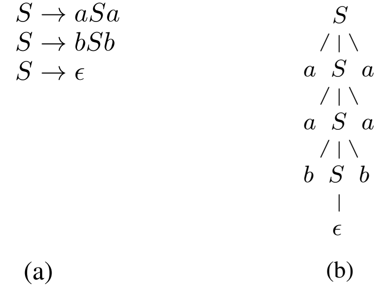

Figure 1: (a) Example of a context-free grammar; (b) the corresponding *parse tree* for the string $aabbaa$.

To illustrate this Figure 1(a) shows an example CFG that contains a single non-terminal symbol, S, and three terminal symbols (a, b and ϵ, where ϵ denotes the empty string). This CFG defines three production rules that can generate, for instance, the string $aabbaa$ by applying the first rule twice, then the second rule once and finally, the third rule once again. The *parse tree* in Figure 1(b) exemplifies the previous rule application and proves that the string $aabbaa$ belongs to the language defined by the grammar.

Learning the entire class of CFGs using only positive examples, i.e. strings that belong to the corresponding formal language, is not identifiable in the limit [Gold, 1967]. In this paper we address the generation of CFGs from positive examples but: (1) for a bounded maximum number of non-terminal symbols in the grammar and (2), a bounded maximum size of the rules in the grammar (i.e. a maximum number of symbols in the right-hand side of the grammar rules).

Our approach is compiling this inductive learning task into a classical planning task whose solutions are sequences of actions that build and validate a CFG compliant with the input strings. The reported empirical results show that our compilation can generate CFGs from small amounts of input data (even a single input string in some cases) using an off-the-shelf classical planner. In addition, we show that the compilation is also suitable for implementing the two canonical tasks of CFGs, *string production* and *string recognition*.

2 Background

This section defines the formalization of CFGs and the planning models that we adopt in this work.

2.1 Context-Free Grammars

We define a CFG as a tuple $\mathcal{G} = \langle V, v_0, \Sigma, R \rangle$ where:

- V is the finite set of non-terminal symbols, also called variables. Each $v \in V$ represents a sub-language of the language defined by the grammar.
- $v_0 \in V$ is the start non-terminal symbol that represents the whole grammar.
- Σ is the finite set of terminal symbols, which are disjoint from the set of non-terminal symbols, i.e. $V \cap \Sigma \neq \emptyset$. The set of terminal symbols is the alphabet of the language defined by $\mathcal{G}$ and can contain the empty string, which we denote by $\epsilon \in \Sigma$.

- $R : V \to (V \cup \Sigma)^*$ is the finite set of production rules in the grammar. By definition rules $r \in R$ always contain a single non-terminal symbol on the left-hand side.

Figure 1(a) shows a 1-variable CFG since it defines a single non-terminal symbol. In this example $V = \{S\}$, $v_0 = S$, $\Sigma = \{a, b, \epsilon\}$, and $R = \{S \to aSa, S \to bSb, S \to \epsilon\}$.

For any two strings $e_1, e_2 \in (V \cup \Sigma)^*$ we say that e_1 *directly yields* e_2, denoted by $e_1 \Rightarrow e_2$, iff there exists a rule $\alpha \to \beta \in R$ such that $e_1 = u_1 \alpha u_2$ and $e_2 = u_1 \beta u_2$ with $u_1, u_2 \in (V \cup \Sigma)^*$. Furthermore we say e_1 *yields* e_2, denoted by $e_1 \Rightarrow^* e_2$, iff $\exists k \geq 0$ and $\exists u_1, \ldots, u_k$ such that $e_1 = u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k = e_2$. For instance, Figure 1(b) shows how the string S yields the string $aabbaa$. The language of a CFG, $L(\mathcal{G}) = \{e \in \Sigma^* : v_0 \Rightarrow^* e\}$, is the set of strings that contain only terminal symbols and that can be yielded from the string that contains only the initial non-terminal symbol v_0.

Given a CFG $\mathcal{G}$ and a string $e \in L(\mathcal{G})$ that belongs to its language, we define a *parse tree* $t_{\mathcal{G},e}$ as an ordered, rooted tree that determines a concrete syntactic structure of e according to the rules in $\mathcal{G}$:

- Each node in a parse tree $t_{\mathcal{G},e}$ is either:
  - An *internal node* that corresponds to the application of a rule $r \in R$.
  - A *leaf node* that corresponds to a terminal symbol $\sigma \in \Sigma$ and has no outgoing branches.
- Edges in a parse tree $t_{\mathcal{G},e}$ connect non-terminal symbols to terminal or non-terminal symbols following the rules R in $\mathcal{G}$.

2.2 Classical Planning with Conditional Effects

We use the model of *classical planning with conditional effects* because conditional effects allow to adapt the execution of a sequence of actions to different initial states, as shown in conformant planning [Palacios and Geffner, 2009]. The support of conditional effects is now a requirement of the International Planning Competition [Vallati *et al.*, 2015] and most current classical planners natively cope with conditional effects without compiling them away [Helmert, 2006].

We use F to denote a set of propositional variables or *fluents* describing a state. A literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L on F represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents. Given L, let $\neg L = \{\neg l : l \in L\}$ be the complement of L.

A *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions with conditional effects. Each action $a \in A$ has a set of literals $\mathsf{pre}(a) \in \mathcal{L}(F)$ called the *precondition* and a set of conditional effects $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals $C \in \mathcal{L}(F)$ (the condition) and $E \in \mathcal{L}(F)$ (the effect).

An action $a \in A$ is applicable in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is a new state $\theta(s, a) = (s \setminus \neg \mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

Given a frame $\Phi = \langle F, A \rangle$, a *classical planning instance* is a tuple $P = \langle F, A, I, G \rangle$, where $I \in \mathcal{L}(F)$ is an initial state (i.e. $|I| = |F|$) and $G \in \mathcal{L}(F)$ is a goal condition. We consider the fragment of classical planning with conditional effects that includes negative conditions and goals.

A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the application of π in I.

2.3 Generalized Planning

Our approach for learning CFGs from positive examples is to model this task as a generalized planning problem that is eventually solved with an off-the-shelf classical planner using the compilation proposed by Jiménez and Jonsson (2015).

We define a generalized planning problem as a finite set of classical planning instances $\mathcal{P} = \{P_1, \ldots, P_T\}$ within the same planning frame Φ. Therefore, $P_1 = \langle F, A, I_1, G_1 \rangle, \ldots, P_T = \langle F, A, I_T, G_T \rangle$ share the same fluents and actions and differ only in the initial state and goals. Although actions are shared, each action can have different interpretations in different states due to conditional effects. Given a planning frame $\Phi = \langle F, A \rangle$, then $\Gamma(\Phi) = \{\langle F, A, I, G \rangle : I \in \mathcal{L}(F), |I| = |F|, G \in \mathcal{L}(F)\}$ denotes the set of all planning instances that can be instantiated from Φ by defining an initial state I and a goal condition G.

A solution Π to a generalized planning problem $\mathcal{P}$ is a generalized plan that solves every individual instance P_t, $1 \leq t \leq T$. Generalized plans may have diverse forms that range from *DS-planners* [Winner and Veloso, 2003] and *generalized polices* [Martín and Geffner, 2004] to finite state machines [Bonet *et al.*, 2010; Segovia-Aguas *et al.*, 2016b], AND/OR graphs or CFGs [Ramirez and Geffner, 2016].

In this work we assume that the solution to a generalized planning problem $\mathcal{P}$ is in the form of a *planning program* [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016a]. Briefly, a planning program is a set of *procedures*, each a sequence of planning actions enhanced with *goto instructions* and procedure calls. However, in this work there is no need to include goto instructions since they are not needed to represent CFGs.

Given a frame $\Phi = \langle F, A \rangle$, a *planning program with procedures* is a finite set of planning programs $\Pi = \{\Pi_0, \ldots, \Pi_m\}$ (we adopt the convention of designating Π_0 as the *main program*, and $\{\Pi_1, \ldots, \Pi_m\}$ as the *auxiliary procedures*). Every Π_j, $0 \leq j \leq m$ is then a sequence of instructions

```
Π0: 0. produce_a  Π1: 0. produce_b
    1. produce_a      1. produce_b
    2. call(1)        2. end
    3. produce_a
    4. produce_a
    5. end
```

Figure 2: Example of a planning program $\Pi = \{\Pi_0, \Pi_1\}$ with one auxiliary procedure that produces the string $aabbaa$.

$\Pi_j = \langle w_0, \ldots, w_n \rangle$. Each instruction w_i, $0 \leq i \leq n$, is associated with a *program line* i and is drawn from the instructions set $\mathcal{I} = A \cup \mathcal{I}_{call} \cup \{end\}$, where $\mathcal{I}_{call} = \{\texttt{call(j')} : 0 \leq j' \leq m\}$ is the set of *procedure calls*, allowing any procedure to call any other procedure on an arbitrary program line ($j = j'$ implies a recursive call). Figure 2 shows an example of a planning program with one auxiliary procedure that produces the string $aabbaa$. In this example, $A = \{\texttt{produce_a}, \texttt{produce_b}\}$, where `produce_a` has the effect of producing the terminal symbol `a`.

To define the execution model of a *planning program with procedures* we first introduce the notion of a *call stack* that keeps track of where control should return when the execution of a procedure terminates. Each element of the call stack is a tuple (j, i), where j is an index that refers to a procedure Π_j, $0 \leq j \leq m$ and i is a program line, $0 \leq i \leq |\Pi_j|$. In what follows we use $\Omega \oplus (j, i)$ to denote a call stack recursively defined by a call stack Ω and a top element (j, i).

The execution model for a planning program Π consists of a program state (s, Ω), where s is a planning state and Ω is a call stack. Given a program state $(s, \Omega \oplus (j, i))$, the execution of instruction w_i^j on line i of procedure Π_j is defined as:

- If $w_i^j \in A$, the new program state is $(s', \Omega \oplus (j, i+1))$, where $s' = \theta(s, w_i^j)$ is the state resulting from applying action w_i^j in state s.
- If $w_i^j = \texttt{call(j')}$, the new program state is $(s, \Omega \oplus (j, i+1) \oplus (j', 0))$. In other words, calling a procedure $\Pi_{j'}$ has the effect of (1) incrementing the program line at the top of the stack; and (2) pushing a new element onto the call stack to start the execution of $\Pi_{j'}$ on line 0.
- If $w_i^j = \texttt{end}$, the new program state is (s, Ω), i.e. has the effect of terminating a procedure by popping element (j, i) from the top of the call stack.

The execution of a planning program with procedures terminates when executing an end instruction empties the call stack, i.e. in program states $(s, \emptyset)$.

To execute a planning program with procedures Π on a planning problem $P = \langle F, A, I, G \rangle$, we set the initial program state to $(I, (0, 0))$, i.e. execution is in the initial planning state, on program line 0 of the main program Π_0. To ensure that the execution model remains bounded we impose an upper bound ℓ on the size of the call stack.

Executing Π on P can fail for any of these four reasons:

1. Execution terminates in program state $(s, \emptyset)$ but the goal condition does not hold, i.e. $G \not\subseteq s$.
2. When executing action $w_i \in A$ in program state (s, Ω), the precondition of w_i does not hold, i.e. $\mathsf{pre}(w_i) \not\subseteq s$.
3. Execution enters an infinite loop that never reaches a program state $(s, \emptyset)$.
4. Executing a `call(j')` instruction in program state (s, Ω) exceeds the stack size $|\Omega| = \ell$, i.e. causes a *stack overflow*.

We say that Π *solves* P iff the goal condition holds when the execution of Π on P terminates, i.e. $(s, \emptyset) \wedge G \subseteq s$. Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$, a planning program Π solves $\mathcal{P}$ iff Π solves each planning instance $P_t, 1 \leq t \leq T$.

2.4 Computing Planning Programs

A classical planner can detect the four failure conditions defined above, making it particularly suitable for generating planning programs. We briefly describe here the compilation from generalized planning to classical planning for computing planning programs [Jiménez and Jonsson, 2015; Segovia-Aguas *et al.*, 2016a].

Given a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$, the bounds on the number of program lines n, auxiliary procedures m and stack size ℓ, the compilation outputs a classical planning problem $P_{n,m}^\ell = \langle F_{n,m}^\ell, A_{n,m}^\ell, I_{n,m}^\ell, G_{n,m}^\ell \rangle$ such that any classical plan π that solves $P_{n,m}^\ell$ programs the instructions of a planning program Π and simulates the execution of Π on each $P_{1 \leq t \leq T}$, validating that Π solves $\mathcal{P}$.

In more detail, the set of fluents $F_{n,m}^\ell$ extends F with:

- $F_{pc}^\ell = \{\mathsf{pc}_i^k : 0 \leq i \leq n, 1 \leq k \leq \ell\} \cup \{\mathsf{proc}_j^k : 0 \leq j \leq m, 1 \leq k \leq \ell\}$ representing the current line and procedure.
- $F_{top}^\ell = \{\mathsf{top}^k\}_{0 \leq k \leq \ell}$ representing the top level of the stack at the current time.
- F_{ins}^m encoding the instructions in the main and auxiliary procedures, which is formally defined as $F_{ins}^m = \{\mathsf{ins}_{i,j,w} : 0 \leq i \leq n, 0 \leq j \leq m, w \in A \cup \mathcal{I}_{call} \cup \{\mathsf{nil}, \mathsf{end}\}\}$, where nil indicates that the instruction on line i of procedure j is *empty*, i.e. can be programmed.
- done, a fluent marking we are done programming and executing the planning program.

$A_{n,m}^\ell$ defines the following actions:

- An action $w_{i,j}^k$ for each action instruction $w \in A$, program line i, procedure j and stack level k:

$$\mathsf{pre}(w_{i,j}^k) = \mathsf{pre}(w) \cup \{\mathsf{pc}_i^k, \mathsf{top}^k, \mathsf{proc}_j^k\},$$
$$\mathsf{cond}(w_{i,j}^k) = \mathsf{cond}(w) \cup \{\emptyset \rhd \{\neg\mathsf{pc}_i^k, \mathsf{pc}_{i+1}^k\}\}.$$

- An action $\mathsf{call}_{i,j}^{j',k}$ for each $\texttt{call(j')} \in \mathcal{I}_{call}$ that pushes a new program line $(j', 0)$ onto the stack:

$$\mathsf{pre}(\mathsf{call}_{i,j}^{j',k}) = \{\mathsf{pc}_i^k, \mathsf{top}^k, \mathsf{proc}_j^k\},$$
$$\mathsf{cond}(\mathsf{call}_{i,j}^{j',k}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i^k, \mathsf{pc}_{i+1}^k, \neg\mathsf{top}^k, \mathsf{top}^{k+1}, \mathsf{pc}_0^{k+1}, \mathsf{proc}_{j'}^{k+1}\}\}.$$

- Actions $\mathsf{end}_{i,j}^{k+1}$ that simulate the termination on line i of procedure j on stack level $k+1$, $0 \leq k < \ell$:

$$\mathsf{pre}(\mathsf{end}_{i,j}^{k+1}) = \{\mathsf{pc}_i^{k+1}, \mathsf{top}^{k+1}, \mathsf{proc}_j^{k+1}\},$$
$$\mathsf{cond}(\mathsf{end}_{i,j}^{k+1}) = \{\emptyset \rhd \{\neg\mathsf{pc}_i^{k+1}, \neg\mathsf{top}^{k+1}, \neg\mathsf{proc}_j^{k+1}, \mathsf{top}^k\}\}.$$

The effect is to pop the program line (j,i) from the top of the stack.

The action set $A_{n,m}^{\ell}$ contains the following actions:

- For each instruction $w_{i,j}^k$, $w \in A \cup \mathcal{I}_{call} \cup \{\mathsf{end}\}$, an action $P(w_{i,j}^k)$ that programs w, and a repeat action $R(w_{i,j}^k)$ that executes w when already programmed:

$$\mathsf{pre}(P(w_{i,j}^k)) = \mathsf{pre}(w_{i,j}^k) \cup \{\mathsf{ins}_{i,j,\mathsf{nil}}\},$$
$$\mathsf{cond}(P(w_{i,j}^k)) = \{\emptyset \rhd \{\neg\mathsf{ins}_{i,j,\mathsf{nil}}, \mathsf{ins}_{i,j,w}\}\},$$
$$\mathsf{pre}(R(w_{i,j}^k)) = \mathsf{pre}(w_{i,j}^k) \cup \{\mathsf{ins}_{i,j,w}\},$$
$$\mathsf{cond}(R(w_{i,j}^k)) = \mathsf{cond}(w_{i,j}^k).$$

- For each planning problem P_t, $1 \leq t \leq T$, a termination action term_t that simulates the successful termination of the planning program on P_t when the stack is empty:

$$\mathsf{pre}(\mathsf{term}_t) = G_t \cup \{\mathsf{top}^0\}, t < T,$$
$$\mathsf{cond}(\mathsf{term}_t) = \{\emptyset \rhd \{\neg\mathsf{top}^0, \mathsf{top}^1, \mathsf{pc}_0^1, \mathsf{proc}_0^1\}\} \cup \{\{\neg f\} \rhd \{f\} : f \in I_{t+1}\} \cup \{\{f\} \rhd \{\neg f\} : f \notin I_{t+1}\}, t < T,$$
$$\mathsf{pre}(\mathsf{term}_T) = G_T \cup \{\mathsf{top}^0\},$$
$$\mathsf{cond}(\mathsf{term}_T) = \{\emptyset \rhd \{\mathsf{done}\}\}.$$

Note that the effect of term_t, $t < T$, is to reset the program state to the initial state of problem P_{t+1}.

The initial state sets all the program lines (`main` and auxiliary procedures) as *empty* and sets the procedure on stack level 1 to Π_0 (the `main` procedure) with the program counter pointing to the first line of that procedure. The initial state on fluents in F is I_1, hence $I_{n,m}^{\ell} = I_1 \cup \{\mathsf{ins}_{i,j,\mathsf{nil}} : 0 \leq i \leq n, 0 \leq j \leq m\} \cup \{\mathsf{top}^1, \mathsf{pc}_0^1, \mathsf{proc}_0^1\}$. The goal condition is defined as $G_{n,m}^{\ell} = \{\mathsf{done}\}$.

3 Generating Context-Free Grammars

This section explains our approach to generating CFGs from input strings using classical planning. We formalize this task as a tuple $\langle \Sigma, E, m \rangle$, where:

- Σ is the finite set of terminal symbols.
- $E = \{e_1, \ldots, e_T\}$ is the finite set of input strings containing only terminal symbols: $e_t \in \Sigma^*$, $1 \leq t \leq T$.
- m is a bound on the number of non-terminal symbols $V_m = \{v_0, \ldots, v_m\}$. As a consequence, m implicitly defines the space of possible rules, $V_m \to (V_m \cup \Sigma)^*$.

A solution to this inductive learning task is a CFG $\mathcal{G} = \langle V_m, v_0, \Sigma, R_m \rangle$ such that, for every $e \in E$, there exists a parse tree $t_{\mathcal{G},e}$.

```
Π0: 0. choose(1|5|8)
    1. parse_a
    2. call(0)
    3. parse_a
    4. end
    5. parse_b
    6. call(0)
    7. parse_b
    8. end
```

Figure 3: Planning program that represents the CFG in Figure 1(a).

3.1 CFG Generation as Generalized Planning

Our approach for solving $\langle \Sigma, E, m \rangle$ is modeling this task as a generalized planning task $\mathcal{P} = \{P_1, \ldots, P_T\}$ where each input string $e_t \in E$ corresponds to an individual classical planning task $P_t \in \mathcal{P}$ such that $1 \leq t \leq T$ and $P_t = \langle F, A, I_t, G_t \rangle$ is defined as follows:

- F comprises the fluents for modeling input strings as lists of symbols. These fluents are $\mathsf{string}(\mathsf{id}, \sigma)$ and $\mathsf{next}(\mathsf{id}, \mathsf{id2})$, where $0 \leq id, id2 \leq z$, $\sigma \in \Sigma$ and z is a bound on the string length. For instance, the string $abba$ is encoded as:

  $\mathsf{string(i0,a), string(i1,b), string(i2,b), string(i3,a), next(i0,i1), next(i1,i2), next(i2,i3), next(i3,i4)}$.

  In addition, F includes fluents $\mathsf{pos}(\mathsf{id})$, $0 \leq id \leq z$, to indicate the current string position, and $\mathsf{symb}(\sigma)$, $\sigma \in \Sigma$, to indicate the symbol at the current string position.

- A contains the actions for parsing the current symbol of an input string. There is a parse_σ action for each symbol $\sigma \in \Sigma$, e.g. $A = \{\mathsf{parse_a}, \mathsf{parse_b}\}$ for the CFG of Figure 1(a). Action parse_σ recognizes that σ is at the current position of the string and advances the position.

$$\mathsf{pre}(\mathsf{parse}_\sigma) = \{\mathsf{symb}(\sigma)\},$$
$$\mathsf{cond}(\mathsf{parse}_\sigma) = \{\{\mathsf{pos}(\mathsf{i_1}), \mathsf{next}(\mathsf{i_1}, \mathsf{i_2}), \mathsf{string}(\mathsf{i_2}, \sigma')\} \rhd \{\neg\mathsf{pos}(\mathsf{i_1}), \mathsf{pos}(\mathsf{i_2}), \neg\mathsf{symb}(\sigma), \mathsf{symb}(\sigma')\} : \forall \mathsf{i_1}, \mathsf{i_2}, \sigma'\}.$$

- I_t contains the fluents encoding the t-th string, $e_t \in E$, and its initial position $\mathsf{pos}(0)$.
- G_t requires that e_t is parsed, i.e. $G_t = \{\mathsf{pos}(|\mathsf{e_t}|)\}$.

According to this definition, a solution to a generalized planning problem $\mathcal{P}$ that models a CFG generation task $\langle \Sigma, E, m \rangle$ parses every $e_t \in E$.

3.2 Computing CFGs with Classical Planning

To compute CFGs with classical planning we first extend the *planning program* formalism, so that it can represent CFGs. Then we adapt the generalized planning compilation for computing the extended planning programs with an off-the-shelf classical planner.

Our extension of the planning program formalism augments its instruction set $\mathcal{I}$ with *choice instructions*. Choice instructions are intended to jump to a target line of a planning program and are defined as $\mathcal{I}_{choice} = \{choose(Target)\}$, where $Target \subseteq \{1, \ldots, n\}$ is a subset of possible target

program lines. Figure 3 shows a planning program with a *choice instruction* that encodes the CFG in Figure 1(a). In this example instruction `choose(1|5|8)` represents a jump to one of these three possible targets, line 1, line 5 or line 8.

The execution model for planning programs with choice instructions behaves as explained in Section 2.3. The conditions for *termination* and *success* are the same as in the original planning program formalism. The only new behavior that has to be defined is the execution of choice instructions. The execution of an instruction $w_i^j = choose(Target)$ on line i of procedure Π_j actively *chooses* to jump to a new line $i' \in Target$, changing the program state from $(s, \Omega \oplus (j, i))$ to $(s, \Omega \oplus (j, i'))$.

The representation of CFGs with planning programs is done associating each non-terminal symbol $v_j \in V_m$ with a planning program Π_j. *Choice instructions* always appear on the first line of Π_j and represent possible jumps to the lines coding the grammar rules, $v_j \to (v_j \cup \Sigma)^*$, associated to the corresponding non-terminal symbol. Initially, the subset of target program lines only includes 1 and n, i.e. $choose(1|8)$ for the example in Figure 3. Whenever we program an *end* instruction on a line i, we add $i+1$ to the subset of target lines, leading to the choice instruction $choose(1|5|8)$ in Figure 3.

The compilation takes as input a CFG generation task $\langle \Sigma, E, m \rangle$ such that $|e_t| \leq z$ for each $e_t \in E$, a number of program lines n and a stack size ℓ, and outputs a classical planning instance $P_{n,m}^{\ell,z} = \langle F_{n,m}^{\ell,z}, A_{n,m}^{\ell,z}, I_{n,m}^{\ell,z}, G_{n,m}^{\ell,z} \rangle$. The compilation is almost identical to the compilation described in the previous section; the only relevant difference is that $A_{n,m}^{\ell,z}$ includes actions for simulating the execution of choice instructions $\mathsf{choose(Target)}_{i',j}^{k}$, where $i' \in Target$:

$$\mathsf{pre}(\mathsf{choose(Target)}_{i',j}^{k}) = \{\mathsf{pc}_0^k, \mathsf{top}^k, \mathsf{proc}_j^k, \mathsf{ins}_{0,j,choose}\},$$
$$\mathsf{cond}(\mathsf{choose(Target)}_{i',j}^{k}) = \{\emptyset \rhd \{\neg \mathsf{pc}_0^k, \mathsf{pc}_{i'}^k\}\}.$$

Lemma 1. *Any classical plan π that solves $P_{n,m}^{\ell,z}$ induces a valid model $\mathcal{G} = \langle V_m, v_0, \Sigma, R_m \rangle$ for the CFG generation task $\langle \Sigma, E, m \rangle$.*

Proof sketch. Once the instructions of a planning program $\Pi = \{\Pi_0, \ldots, \Pi_m\}$ are programmed they can only be executed. The classical plan π has to program the instructions (if not yet programmed) of Π and simulate its execution, actively choosing the jumps defined by the choice instructions and their corresponding subsets of *target* lines. This simulation is done for the planning task P_t encoding the t-th string in E and the *active choice* corresponds exactly to the construction of the parse tree for the t-th string. If this is done for every $1 \leq t \leq T$, the CFG induced by π satisfies the solution condition for the solutions of the CFG generation task $\langle \Sigma, E, m \rangle$. □

3.3 Parsing and Production with Planning

String production and string parsing for arbitrary CFGs can also be addressed using our compilation and an off-the-shelf classical planner.

Parsing. Let $e \notin E$ be a new string, and let $P_e = \langle F_{n,m}^{\ell,z}, A_{n,m}^{\ell,z}, I_e, G_e \rangle$ be the classical planning instance for parsing the string e, i.e. instantiated on the planning frame as the problem $P_{n,m}^{\ell,z}$. In this case we can use a classical planner to determine whether $e \in L(\mathcal{G})$.

Our approach is to specify $\mathcal{G}$ in the initial state of P_e, making initially true the fluents $\mathsf{ins}_{i,j,w}$ that correspond to the planning program that encodes $\mathcal{G}$, and ignoring the actions $P(w_{i,j}^k)$ for programming instructions. A solution plan π_e to P_e is then constrained to the actions that execute the instructions specified by $\mathcal{G}$ and represents a parse tree $t_{\mathcal{G},e}$. Essentially, parsing consists in correctly choosing the target program line i' each time a choice instruction is executed.

Interestingly *parsing* can also be understood as activity recognition using plan libraries that are in the form of CFGs [Ramirez and Geffner, 2016].

Production. We can produce a string $e \in L(\mathcal{G})$ using our compilation and an off-the-shelf classical planner.

Again we use a classical planning instance P_e that represents the parsing of example e with $\mathcal{G}$ also specified in the initial state of P_e and ignoring the actions for programming instructions. The difference with the previous task is that here the linked list that encodes the string e is initially empty (the corresponding fluents are false at I_e). This list is filled up, symbol by symbol, by the actions that execute $\mathcal{G}$ until reaching the end of the string. To do so actions parse_σ are replaced with actions $\mathsf{produce}_\sigma$ that add symbol $\sigma \in \Sigma$ at the current position of the string. Formally, $\mathsf{produce}_\sigma$ is defined as:

$$\mathsf{pre}(\mathsf{produce}_\sigma) = \{\mathsf{active}\},$$
$$\mathsf{cond}(\mathsf{produce}_\sigma) = \{\{\mathsf{pos}(\mathsf{i}_1)\} \rhd \{\mathsf{string}(\mathsf{i}_1, \sigma)\} : \forall \mathsf{i}_1\}$$
$$\cup \ \{\{\mathsf{pos}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1, \mathsf{i}_2)\} \rhd \{\neg \mathsf{pos}(\mathsf{i}_1), \mathsf{pos}(\mathsf{i}_2)\} : \forall \mathsf{i}_1, \mathsf{i}_2\}$$
$$\cup \ \{\{\mathsf{pos}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1, z)\} \rhd \{\neg \mathsf{active}\} : \forall \mathsf{i}_1, \mathsf{i}_2\},$$

where active is a fluent that keeps track of whether we have reached the end of the string to be generated.

Text production using grammars and classical planning is a well studied task [Schwenger *et al.*, 2016; Koller and Hoffmann, 2010]. This task is also related to the compilation of domain-specific control knowledge for planning [Baier *et al.*, 2007; Alford *et al.*, 2009].

4 Evaluation

We designed two types of experiments: (1) *Generation*, for computing CFGs compliant with a set of input strings and (2), *Recognition* for parsing unseen input strings given a CFG. All the experiments are run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM, using the classical planner Fast Downward [Helmert, 2006], with the `LAMA-2011` configuration, and a planning time limit of 600 seconds.

We created six domains that correspond to CFGs with different structure and alphabet. The domain **AnBn** corresponds to the A^nB^n language. The **Parenthesis** domain corresponds to the strings that can be formed following one of two well-balanced parenthesis distributions, *sequential* $()()\ldots$ or *enclosing* $((\ldots))$. **Parenthesis Multiple** corresponds to the enclosing well-balanced parenthesis distribution but using a larger alphabet, $\Sigma = \{(, \{, [,], \},)\}$. In the **Binary Arithmetics** domain the alphabet contains two possible binary values $\{0, 1\}$ and two operators $\{+, -\}$, and corresponds to the

language of the arithmetic expressions composed by an arbitrary binary number, an operator and another binary number. For **Arithmetics** the alphabet includes the values $\{0, \ldots, 9\}$ and the operators $\{+, -, *, /\}$, and the language is the set of expressions formed as decimal number, operator and decimal number. Finally, a simplified **English Grammar** that includes *Sentence*, *Noun Phrase* and *Verb Phrase* as non-terminal symbols while *adjective*, *adverb*, *noun* and *verb* are terminal symbols.

Table 1 shows the results of the *Generation* tasks. For each domain we report the number m of non-terminal symbols, the size of the stack, the procedure lines (per non-terminal symbol), the number of input strings (per non-terminal symbol) and the planning time for computing each procedure corresponding to a non-terminal symbol.

	m	Stack	Lines	Strings	Time(s)
AnBn	1	5	5	1	0.3
Parenthesis	1	5	5	2	0.4
P. Multiple	1	5	12	3	53.1
Binary A.	2	4	(6,8)	(4,2)	(0.6,1.8)
Arithmetics	4	8	(20,8,3,4)	(10,4,1,4)	(10.3,3.7, 3.5,14.6)
E. Grammar	3	10	(6,3,3)	(2,1,1)	(1.4,0.3,1.9)

Table 1: Generation task results.

The *Generation* results show that non-terminal symbols are used with two aims: i) abstracting a set of terminal symbols, e.g. the first procedure of the **Arithmetics** domain (with 20 lines, learned from 10 strings in 10.3 seconds) processes any digit in the set $\{0, \ldots, 9\}$; ii) grouping multiple rules, e.g. in **English Grammar** one procedure represents a Noun Phrase (NP) that is composed of one or more adjectives (a) and a noun (n), so it computes the rules $NP \rightarrow an|aNP$.

Table 2 shows the results for the *Recognition* tasks. In these experiments the CFGs grammars are given so we explore the performance of our approach using larger stack sizes. For each domain we report the size of the stack (which limits the max depth of the possible parse trees), the number of strings, and the total planning time required for parsing the strings.

	Stack	Strings	Time(s)
AnBn	51	1	70.67
Parenthesis	52	1	19.29
P. Multiple	52	1	129.19
Binary A.	15	2	62.87
Arithmetics	25	4	137.76
E. Grammar	92	1	325.44

Table 2: Recognition task results.

5 Related work

The learning of CFGs can also be understood in terms of activity recognition, such that the library of activities is formalized as a CFG, the library is initially unknown, and the input strings encode observations of the activities to recognize. *Activity recognition* is traditionally considered independent of the research done on automated planning, using hand-crafted libraries of activities and specific algorithms [Ravi *et al.*, 2005]. An exception is the work by Ramírez and Geffner [2009; 2010] where goal recognition is formulated and solved with planning. As far as we know our work is the first that tightly integrates the tasks of (1) grammar learning, (2) recognition and (3) production using a common planning model and an off-the-shelf classical planner.

Hierarchical Task Networks (HTNs) is a powerful formalism for representing libraries of plans [Nau *et al.*, 2003]. HTNs are also defined at several levels such that the tasks at one level are decomposed into other tasks at lower levels with HTN decomposition methods sharing similarities with production rules in CFGs. There is previous work in generating HTNs [Hogg *et al.*, 2008; Lotinac and Jonsson, 2016] and an interesting research direction is to extend our approach for computing HTNs from flat sequences of actions. This aim is related to Inductive Logic Programming (ILP) [Muggleton, 1999] that learns logic programs from examples. Unlike logic programs (or HTNs) the CFGs that we generate are propositional and do not include variables. Techniques for learning high level state features that include variables are promising for learning lifted grammars [Lotinac *et al.*, 2016].

6 Conclusions

There is exhaustive previous work on learning CFGs given a corpus of correctly parsed input strings [Sakakibara, 1992; Langley and Stromsten, 2000] or using positive and negative examples [De la Higuera, 2010; Muggleton *et al.*, 2014]. This work addresses generating CFGs using only a small set of positive examples (in some cases even one single string that belongs to the language). Furthermore we follow a compilation approach that benefits straightforwardly from research advances in classical planning and that is also suitable for *production* and *recognition* tasks with arbitrary CFGs.

Our compilation bounds the number of rules m, the length of these rules n, the size of the stack ℓ and the length of the input strings z. If these bounds are too small, the classical planner used to solve the output planning task will not be able to find a solution. Larger values for these bounds do not formally affect to our approach, but in practice, the performance of classical planners is sensitive to the size of the input. Interestingly our approach can also follow an incremental strategy where we generate the CFG for a given sub-language and then encode this *sub-grammar* as an auxiliary procedure for generating more challenging CFGs [Segovia-Aguas *et al.*, 2016b].

The size of the compilation output also depends on the number of examples. Empirical results show that our approach is able to generate non-trivial CFGs from very small data sets. Another interesting extension would be to add negative input strings, which would require a mechanism for validating that a given CFG does *not* generate a given string, or to accept incomplete input strings that would require combining the generation and production mechanisms.

Acknowledgments

This work is partially supported by grant TIN2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain.

References

[Alford *et al.*, 2009] Ronald Alford, Ugur Kuter, and Dana S Nau. Translating htns to pddl: A small amount of domain knowledge can go a long way. In *IJCAI*, pages 1629–1634, 2009.

[Baier *et al.*, 2007] Jorge A Baier, Christian Fritz, and Sheila A McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, pages 26–33, 2007.

[Bonet *et al.*, 2010] Blai Bonet, Hector Palacios, and Hector Geffner. Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*, 2010.

[Chomsky, 2002] Noam Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[De la Higuera, 2010] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[Gold, 1967] E Mark Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[Helmert, 2006] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[Hogg *et al.*, 2008] Chad Hogg, Héctor Munoz-Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956, 2008.

[Jiménez and Jonsson, 2015] Sergio Jiménez and Anders Jonsson. Computing Plans with Control Flow and Procedures Using a Classical Planner. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS-15*, pages 62–69, 2015.

[Koller and Hoffmann, 2010] Alexander Koller and Jörg Hoffmann. Waking up a sleeping rabbit: On natural-language sentence generation with ff. In *ICAPS*, pages 238–241, 2010.

[Langley and Stromsten, 2000] Pat Langley and Sean Stromsten. Learning context-free grammars with a simplicity bias. In *European Conference on Machine Learning*, pages 220–228. Springer, 2000.

[Lotinac and Jonsson, 2016] Damir Lotinac and Anders Jonsson. Constructing Hierarchical Task Models Using Invariance Analysis. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI'16)*, 2016.

[Lotinac *et al.*, 2016] Damir Lotinac, Javier Segovia, Sergio Jiménez, and Anders Jonsson. Automatic generation of high-level state features for generalized planning. In *IJCAI*, 2016.

[Martín and Geffner, 2004] Mario Martín and Hector Geffner. Learning generalized policies from planning examples using concept languages. *Appl. Intell*, 20:9–19, 2004.

[Muggleton *et al.*, 2014] Stephen H Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine learning*, 94(1):25–49, 2014.

[Muggleton, 1999] Stephen Muggleton. Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114(1):283–296, 1999.

[Nau *et al.*, 2003] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404, 2003.

[Palacios and Geffner, 2009] Héctor Palacios and Héctor Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35:623–675, 2009.

[Ramírez and Geffner, 2009] Miquel Ramírez and Hector Geffner. Plan recognition as planning. In *IJCAI*, pages 1778–1783, 2009.

[Ramırez and Geffner, 2010] Miquel Ramırez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010)*, pages 1121–1126, 2010.

[Ramirez and Geffner, 2016] Miquel Ramirez and Hector Geffner. Heuristics for planning, plan recognition and parsing. *arXiv preprint arXiv:1605.05807*, 2016.

[Ravi *et al.*, 2005] Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael L Littman. Activity recognition from accelerometer data. In *AAAI*, volume 5, pages 1541–1546, 2005.

[Sakakibara, 1992] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[Schwenger *et al.*, 2016] Maximilian Schwenger, Alvaro Torralba, Jörg Hoffmann, David M Howcroft, and Vera Demberg. From openccg to ai planning: Detecting infeasible edges in sentence generation. In *International Conference on Computational Linguistics*, 2016.

[Segovia-Aguas *et al.*, 2016a] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generalized planning with procedural domain control knowledge. In *ICAPS*, 2016.

[Segovia-Aguas *et al.*, 2016b] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Hierarchical finite state controllers for generalized planning. In *IJCAI*, 2016.

[Vallati *et al.*, 2015] Mauro Vallati, Lukáš Chrpa, Marek Grzes, Thomas L McCluskey, Mark Roberts, and Scott Sanner. The 2014 international planning competition: Progress and trends. *AI Magazine*, 36(3):90–98, 2015.

[Winner and Veloso, 2003] Elly Winner and Manuela Veloso. Distill: Learning domain-specific planners by example. In *ICML*, pages 800–807, 2003.