Computational Thinking with Logic Programming

Gopal Gupta[1], Elmer Salazar[1] and Joaquín Arias[2]

[1]The University of Texas at Dallas, Richardson, TX, USA
[2]CETINIA, Universidad Rey Juan Carlos, Madrid, Spain

Abstract
Computational thinking is a problem-solving approach that involves breaking down complex problems into smaller, manageable parts, recognizing patterns, abstracting general principles, and devising algorithms to solve them. It can be thought of as an algorithmic method of thinking. Our thesis is that to learn to think computationally, one must first learn to think logically. In this work-in-progress paper we explore how logic programming, answer set programming (ASP) in particular, can aid in learning to think logically. We discuss how ASP can be used to help students organize their thoughts as well as make them understand various processes involved in human thinking. We believe that this will facilitate the teaching of computational thinking to students.

Keywords
Computational Thinking, Logic Programming, Answer Set Programming, s(CASP)

1. Introduction

Computational thinking involves "solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science" [1]. Computational thinking has been proposed as a crucial skill for everyone to learn because it helps individuals develop a structured and logical approach to solving complex problems. A pre-requisite, thus, to computational thinking is to learn to think logically. In this paper we show how logic programming, specifically answer set programming (ASP) and its realization in the goal-directed predicate ASP system called s(CASP), can be employed as an aid for developing logical thinking in students. Note that modeling human thinking using logic programming has been an object of intense study for a long time [2, 3]. Here, our objective is to show how logic programming can be used to teach logical thinking to students which, in turn, can lead to students learning computational thinking [1]. Logical thinking includes being able to precisely express knowledge that is involved in human thought as well as processing this knowledge to draw inferences, infer new knowledge, fill gaps in knowledge, etc.

Humans are capable of thought, of course. However, human thought process appears quite informal, and most people are not able to understand the steps they took, for example, to arrive at a conclusion. We argue that human thought relies on a small number of inference principles, and once a human understands these principles and how they are invoked, he/she can understand their thought processes involved in drawing conclusions, deriving new knowledge, and filling gaps in knowledge. This, in turn, can facilitate computational thinking, as humans can then understand the mapping between the steps of a task to be accomplished and their thought process. A pre-requisite to computational thinking is logical thinking. We show how logic programming can be used as a tool to formalize thought processes that appear informal otherwise. Once a person develops the ability to see the formalism underlying their thoughts, they will become better reasoners and hence better computational thinkers.

Formalizing the human thought process has been considered hard. The study of human thought process

XXX 2024: N^{xx} Workshop on XXX, October, 2024, Dallas, USA.

has been conducted over several millennia [4, 5]. In modern times this effort culminated in boolean logic [6], first order logic [7], and various other advanced logics. These logics, however, are limited and could not match the sophistication of human reasoning in the sense that it is hard to use these logics to faithfully model the human thought process in an elegant manner.

2. Computational Thinking

Computational thinking involves representing solutions to problems as computation steps (algorithm). The key points of computational thinking include:

- Decomposition: Breaking down complex problems into smaller, more manageable parts.
- Pattern Recognition: Identifying similarities, trends, or patterns in data or problems that can be reused in other situations.
- Abstraction: Simplifying a problem by focusing on the essential details and ignoring irrelevant information.
- Algorithm Design: Creating step-by-step instructions or rules (algorithms) to solve problems systematically and efficiently.
- Debugging: Identifying and fixing errors or flaws in an algorithm or solution.
- Generalization: Applying a solution to a broad set of similar problems.

Computational thinking can also be thought of as scientific thinking, design thinking, or model-based thinking. These have been discussed widely in the literature over decades [8]. Computational thinking, essentially, is algorithmic thinking. With this in mind, recall Kowalski's famous equation [2]:

Algorithm = Logic + Control

In the context of human thinking, 'logic' here stands for knowledge, and 'control' for methods for processing knowledge and reasoning over it. So to master algorithmic thinking, one must not only understand representation of (human) logic or knowledge, but *also the operational procedures involved in processing knowledge and reasoning over it*. Thus, to master algorithmic thinking, humans must become adept at both: knowledge representation *and* reasoning.

Knowledge (logic) that resides in the human mind can be represented as facts, rules, and goals. Control is represented by human mental processes that process knowledge. These mental processes correspond to operations such as deduction, abduction, induction, etc., realized through backward chaining, forward chaining, resolution, etc.

Significant number of articles and books have been written on modeling human knowledge in logic, and on performing reasoning. What we show in this paper is how this reasoning can be presented in a formal way, and how the proofs needed to establish a conclusion can be systematically generated. These proofs can help students understand the complex reasoning processes followed by humans.

3. Modeling Human Thinking

The representation of logic in the human mind (*human logic*) is quite different from what is espoused by classical logic. Human logic is non-monotonic, as it can draw inferences even if the knowledge is incomplete. Incomplete knowledge is elegantly modeled by negation-as-failure and the stable model semantics of logic programming [9]. Thus, to model human logic we will employ answer set programming (ASP). Seeing explicitly represented human thought in the form of an answer set program will lead to students being able to express their knowledge more precisely. *Human control* or reasoning, on the other hand, will by represented by the operational semantics of ASP. We will use the s(CASP) goal-directed predicate answer set programming system [10] for this purpose. The s(CASP) system is a query-driven ASP engine that executes an answer set program in a top-down, query-driven manner, very much how a human might think and reason [11]. The s(CASP) system supports predicates with arbitrary terms and it also supports constructive negation.

Human reasoning methods use techniques such as deduction, abduction, and induction to draw inference [12]. Humans also employ reasoning methods to simplify (human) knowledge by combining rules and facts to produce new information and infer missing knowledge.

Note that aside from negation-as-failure, ASP also includes *strong negation*. Negation-as-failure, represented as `not p` evaluates to true if `p` is false or unknown. It evaluates to false if `p` is true. Strong negation, denoted `-p`, evaluates to false if `p` is known to be true or unknown. It evaluates to true, if falsehood of `p` can be explicitly established.

3.1. Modeling Human Logic

Answer Set Programming (ASP) has been shown to be well-suited for knowledge representation and reasoning [13, 14]. ASP can model human thinking quite closely as it supports default rules with exceptions and preferences, integrity constraints, and *even loop over negation* [15]. ASP can be used to program various shades of truth as well as judgement calls through a combination of negation as failure and *strong negation* [12]. For example, given a proposition `p` (e.g., `p` $\equiv$ “it is raining now”):

1. `p`: denotes that `p` is unconditionally true.
2. `not -p`: denotes that `p` *maybe* true.
3. `not p` $\wedge$ `not -p`: denotes that `p` is unknown, i.e., there is no evidence of either `p` or `-p`.
4. `not p`: denotes that `p` *may* be false (no evidence that `p` is true).
5. `-p`: denotes that `p` is unconditionally false.

These shades of truth allow us to model judgement calls that humans make. Consider a physician who is about to prescribe a medicine to a patient. A physician who is willing to take risks may make the judgement call to immediately prescribe the medicine and not worry about the medicine’s side effects.

```
prescribe(M, D, P) :- cures(M, D), not contraindicated(M, P).   ...(1)
contraindicated(M, P) :- has_side_effects(M, P).                ...(2)
```

which states that medicine `M` can be prescribed to patient `P` for disease `D` if medicine `M` cures disease `D` and `M`’s contraindication for `P` maybe false. `M` is contraindicated for `P`, if `M` has side-effects on `P`. In contrast, a physician who is conservative may make the judgement call to first seek evidence that `M` is not contraindicated for patient `P`.

```
prescribe(M, D, P) :- cures(M, D), not contraindicated(M, P).    ...(3)
contraindicated(M, P) :- not -has_side_effects(M, P).            ...(4)
```

What has changed is the contraindication definition. Medicine `M` is now contraindicated for patient `P` if `M` having a side-effect on `P` *maybe true*. The net effect is that now `M` will be prescribed for patient `P` for disease `D` if we can prove that `M` has no side effects on `P`. Slight rewriting of (3) and (4) results in the rule below, which makes this logic explicit.

```
prescribe(M, D, P) :- cures(M, D), -has_side_effect(M, P).
```

Next, we theorize that a bulk of human knowledge can be simulated with three types of ASP rules: default rules with exceptions and preferences (defeasible rules), integrity constraints, and assumption-based (or abductive) reasoning [12, 15]. These three types of rules cover all of ASP, a Turing-complete language, so one can arguably draw the conclusion that human knowledge can be modeled with just these three constructs.

Default Rules: Given a goal, we establish it by decomposing it into subgoals, and then recursively proving those subgoals next, and so on. The dependence of goals on subgoals is described by logic programming rules. For example:

```
flies(X) :- bird(X), alive(X).
```

However, there might be conditions that might prevent us from reducing a goal into subgoals. These conditions represent exceptions. Thus, the above rule may be refined into:

```
flies(X) :- bird(X), alive(X), not exception(X).
exception(X) :- penguin(X).
```

where if `X` is a penguin it will prevent `flies(X)` from succeeding. There might be multiple rules that may be used to reduce a goal. We can add additional subgoals to the rules that will allow us to discriminate among these rules thereby prioritizing them [15].

Assumption-based Reasoning: While reducing a goal g, if no information is available for it, i.e., no rule is defined for g, we can perform *abductive* or *assumption-based reasoning*. That is, we now consider two alternate worlds, one in which g is true and another one in which g is false. This is achieved by adding the following rules (termed *even loop over negation* in ASP literature [14].

```
g :- not not_g.
not_g :- not g.
```

In the s(CASP) system, the above effect is achieved by simply declaring `g` as an abducible via the directive:

```
#abducible g/0.
```

An abducible can be an arbitrary predicate as well. The directive automatically generates the even loop described above.

Integrity Constraints: As we accumulate subgoals in the reduction process, we may encounter subgoals that are mutually inconsistent. These subgoals may be far apart in the proof process, and their inconsistency cannot be modeled using the exception mechanism described previously. In such a case, we impose these restrictions via integrity constraints. For example, a dead person cannot breathe:

```
false :- person(X), dead(X), breathe(X).
```

Note that integrity constraints are a special case of odd loops over negation [14, 15]. Also, these integrity constraints may be global or local. Above constraint is an example of a global constraint. A local constraint will be imposed on specific predicates, and thus will be included in the body of the rule specifying that predicate.

Logically organized thoughts should be expressible as answer set programs. Consider the following example [13]. A student is admitted to a program if he/she has a high GPA, or has a special talent but a reasonable GPA. A student is declined admission if he/she has low GPA or has no other talent. If it cannot be determined that a student must be admitted or rejected, the student will be interviewed. Note that `-admit` represents rejection.

```
admit(X) :- highGPA(X), not ab_eligible(X).
admit(X) :- special(X), fairGPA(X), not ab_eligible(X).
-admit(X) :- -special(X), -highGPA(X), not ab_ineligible(X).
interview(X) :- not eligible(X), not -eligible(X).
```

Note that `ab_eligible/1` and `ab_ineligible/1` represent realistic situations (a student with a high GPA but a criminal record, and a large donor's offspring with a low GPA, respectively, for example).

3.2. Modeling Human Control

So far we have shown how (possibly incomplete) knowledge that resides in human mind (human logic) can be faithfully and declaratively represented as an answer set program. Human thinking also involves drawing inferences from this knowledge, as well as modifying this knowledge to suit our purposes. We should be able to model this process of drawing conclusions as well as possibly transforming our knowledge to achieve a goal. Similarly, we may combine the knowledge that resides in our mind in all sorts of ways to produce new knowledge. We need to understand and automate these processes as well.

Next, we give a (non-exhaustive) list of concepts that are employed in *human control*. These concepts allow us to draw inferences from knowledge, simplify our knowledge, combine our knowledge, etc. These concepts are also invoked during computational thinking. These concepts must also be taught to students so they can understand their own thinking process. The idea will be to illustrate these concepts to students by applying them to modify and transform knowledge represented as an answer set program. Our goal is to use the s(CASP) ASP interpreter to illustrate this transformation and processing in a concrete way.

- **Deduction**: Deduction is a logical reasoning process where conclusions are drawn from premises. The conclusion logically follows from the premises. If the premises are true, the conclusion must also be true. Modus Ponens is an example of deductive reasoning: If `P` holds and $P \Rightarrow Q$ holds, we can conclude Q. Execution of a query against an answer set program involves performing deduction. Thus, given the program:

  ```
  flies(X) :- bird(X), not exception(X).
  exception(X) :- penguin(X).
  bird(tweety).
  ```

  we can deduce `flies(tweety)`.
- **Abduction**: Abduction refers to a form of reasoning that is concerned with the generation and evaluation of explanatory hypotheses. We could also think of abduction as assumptions based reasoning where we find the assumptions under which an observation would hold. Abductive reasoning leads back from facts to a proposed explanation of those facts or assumptions that will explain that fact. Abductive reasoning takes the following form [16]:

  The fact B is observed.
  But if A were true, B would be a matter of course.
  Hence, there is reason to suspect that A is true.

  Abductive reasoning can be modeled in ASP through even loops over negation, as explained earlier. As an example, consider the rule:

  ```
  flies(X) :- bird(X), not exception(X).
  exception(X) :- not -penguin(X).
  bird(chirpy)
  ```

  Suppose we are wondering whether the bird called `chirpy` flies. Notice that our judgement call if a bird can fly leans on the conservative side, meaning that we must demonstrate that `chirpy` is definitely not a penguin. We can add the rules:

  ```
  -penguin(chirpy) :- not auxpenguin(chirpy).
  auxpenguin(chirpy) :- not -penguin(chirpy).
  ```

  Now the query `?- flies(chirpy)` will succeed with the assumption `-penguin(chirpy)`, namely, `chirpy` should definitely not be a penguin.
- **Induction**: Induction is a reasoning process in which general conclusions are drawn from specific observations or examples. Induction generally refers to learning associations or dependency relations

between data features, in the sense of machine learning or inductive logic programming [17]. A useful idiom may be to explain to students that humans make inductive generalizations from data. Inductive generalization may also come from a single observation or a small number of observations, though generally this generalization relies on human knowledge of the world. Humans represent inductive generalizations as default rules. Much of expert knowledge is representable as default rules [15]. For example, a child may drop a ceramic mug on the floor which subsequently breaks. Perhaps after this happens a few times, the child may generalize and induce:

```
break(X) :- fragile(X), drop(X).
```

The child may subsequently notice that dropping the mug on a carpeted surface doesn't break it, and so may revise the rule above to add the exception.

```
break(X) :- mug(X), drop(X,Y), not exception(X,Y).
exception(X,Y) :- soft_surface(Y).
```

Algorithms that learn rule based machine learning models from large amount of data and represent them as default rules have been designed [18].
Finally, generalizing from just one or two instances of a predicate succeeding can be realized in s(CASP). Suppose we want to assert that if we can prove `p(1)` and `p(2)`, then we can induce that `p(X)` holds for all `X`. In such a case as soon as both `p(1)` and `p(2)` are proved, then the existing knowledge is extended with the following rules:

```
:- p(1), p(2), X .>. 2, not p(X).
p(X) :- X .>. 2, not np(X).
np(X) :- not p(X).
```

where `np(X)` is a dummy predicate. The constraint asserts that if `p(1)` and `p(2)` succeed, then `p(X)` must succeed for any given `X`. The even loop over negation states that for any given value of `X`, `p(X)` must be either true or false. Note that we could have represented this inductive inference much more simply using deduction as follows:

```
p(X) :- X .>. 2, p(1), p(2).
```

However, this representation is not making an inductive inference. In the former approach that involves a global constraint, we are closer to the human thought process, where if we discover that `p(1)` and `p(2)` hold, only then we infer that `p(X)` holds for all `X`. So if we encounter `p(5)` before we prove `p(1)` or `p(2)`, then `p(5)` will fail. In the latter approach, a call to `p(5)` will lead to an attempt to prove `p(1)` and `p(2)`, which is deduction not induction.

- **Resolution**: Resolution can be thought of as *hypothetical syllogism*: if $P \Rightarrow Q$ and $Q \Rightarrow R$, then $P \Rightarrow R$. That is, given $\neg P \vee Q$ and $\neg Q \vee R$, we infer that $\neg P \vee R$ is a their logical consequence. Resolution can be used to realize backward and forward chaining discussed later. In logic programming, resolution appears as SLD resolution [19]. Essentially, resolution corresponds to when a procedure call is made, and the call is matched to a rule head. Unification is used to achieve parameter passing. In s(CASP), coinductive success is another mechanism that is used alongside resolution. If a goal g is a variant of an ancestor goal in the search tree, then g succeeds *coinductively*. Coinductive success is critical for executing even loops over negation in a top-down, query driven manner. Resolution can also be thought of as a mechanism to derive new information (that was earlier implicit).
To illustrate resolution, we use the popularly-cited example: All Cretans are islanders and all islanders are liars, therefore, all Cretans are liars. Coded as:

```
liar(X) :- islander(X).         ...(5)
islander(X) :- cretan(X).       ...(6)
```

Simple reduction will generate the clause

```
liar(X) :- cretan(X).          ...(7)
```

- **Backward chaining**: Backward chaining is a reasoning technique that starts with a goal or conclusion and works backward to determine the conditions or premises that would allow that goal to be true. In a backward chaining system, we start with a query, which is expanded using resolution (SLD resolution in case of pure logic programming), until all goals are resolved. In s(CASP), goals also get resolved due to coinductive success. The s(CASP) system keeps track of all the subgoals that are proved via resolution or coinduction while executing a query. The set of subgoals constitutes a partial answer set. Backward chaining is essentially query-driven execution.
  Thus, using the Cretan example above, if we add the fact

  ```
  cretan(icarus).                ...(8)
  ```

  then the query `?- liar(icarus)` will succeed through backward chaining (`liar(icarus)` $\rightarrow$ `islander(icarus)` $\rightarrow$ `cretan(icarus)` $\rightarrow$ `success`.
- **Forward Chaining**: Forward chaining is a reasoning method that combines facts/rules to infer new information, progressing step by step until a specific goal or conclusion is reached. It can be thought of as a method for combining knowledge to generate new inferred knowledge. In the example above, clause (7) is obtained by forward chaining. In the context of logic programming, forward chaining can be thought of as partial evaluation [20]. In s(CASP), the partial evaluation process is more complex, as coinduction also has to be taken into account. In addition, integrity constraints can lead to failure if they are not satisfied at any instance. In the example above, given the fact (8), forward chaining applied to clauses (5), (6), and (8) will produce the conclusion `liar(icarus)`. Forward chaining is useful in making inferencing more efficient.
- **Craig Interpolation**: For two formulae P and Q such that $P \Rightarrow Q$, a Craig interpolant is a formula I such that $P \Rightarrow I$ and $I \Rightarrow Q$, and all the non logical symbols (i.e., atoms) in I occur in both P and Q. Craig interpolation can be thought of as the opposite of resolution. One can also define the reverse interpolant. For a contradiction $P \wedge Q$ a reverse interpolant is a formula I such that $P \Rightarrow I$, and $I \wedge Q$ is a contradiction, and the non-logical symbols in I occur in P and in Q. Intuitively, a reverse Craig interpolant of two inconsistent formulas P and Q is the smallest formula containing common atoms of P and Q that captures the inconsistency.
  It is important for students to understand the idea of inconsistency among statements. Craig Interpolant captures this idea. If we state "No one is in the kitchen," then next state that "John is in the kitchen," then these two statements are directly inconsistent. These mutually inconsistent statements can be part of larger set of statements. Reverse Craig interpolant corresponds to the smallest subset of statements that captures the inconsistency. Thus, given: `in(maria,study),-in(X, kitchen)` and `in(john,kitchen),in(jill,living_room)`, the reverse interpolant is `-in(X,kitchen)` (X is universally quantified).
- **Causality**: Causality is important for making sense of the world. We are constantly learning the causal relationships between events. We adopt the following definition of causality [21]: event Q causally depends on P if Q would have occurred if P occured, and Q would not have occurred if P had not occurred. It can be described by the relation: $occurs(P) \Leftrightarrow occurs(Q)$. We need to teach students to distinguish between causality and correlation.
  In answer set programming, program clauses are "completed" by default, i.e., given clauses

  ```
  p(X̄) :- B1.
  p(X̄) :- B2.
  ```

  it is assumed that subgoals in the body of `p`, i.e., $B_1 \vee B_2$ are the cause of $p(\bar{X})$. The program completion operation in s(CASP) results in the dual rule being added to the program:

```
not_p(X̄) :- not B1, not B2.
```

To illustrate, we know that a house floods if there is a water pipe leak, or that a house will also flood if there is torrential rain, then it follows that the house will not flood if there is *no* water leak *and no* torrential rain.

```
house_floods :- pipe_leak.
house_floods :- torrential_rain.
not_house_floods :- not pipe_leak, not torrential_rain.
```

However, note that while program completion may appear to make the rules defining causal relationships, they are only "weakly" causal. A negation-as-failure goal `not p` states that there is *no evidence for* `p` being true, and so `not p` can be assumed to be true. So in the above example, if there is no evidence of torrential rain or water pipe leak, then there will be no evidence of house flooding. However, it is up to the user to assume that lack of a pipe leak and lack of torrential rain is sufficient evidence for house not flooding.
If we want to express causality, we must use strong negation. Thus, to state that lack of rain and pipe leak is sufficient evidence, we must explicitly define the `-house_floods` predicate as follows:

```
-house_floods :- -pipe_leak, -torrential_rain.     .... (9)
```

If we have no information about pipe leak or torrential rain, then neither `house_floods` is provable nor `-house_floods`. Thus, whether the house got flooded remains unknown. However, we could make the close world assumption instead, by asserting the rule:

```
-house_floods :- not house_floods.
```

which amounts to:

```
-house_floods :- not pipe_leak, not torrential_rain.
```

If there is no evidence of rain and pipe leak, we can conclude that the house did not flood.
Note that these examples can be executed using s(CASP) and the proof traces shown to students, improving their understanding of causality.

- **Counterfactuals**: A counterfactual is a statement that considers what would have happened if a certain event or condition had been different from what actually occurred. It typically takes the form "If X had happened, Y would have occurred." Counterfactuals can also capture causality [22]. This is because if event `c` causes event `e`, and we state the counterfactual that if `c` had not occurred then `e` would not have occurred, then we are declaring that event `c` is the cause of event `e`.
Counterfactuals can be modeled in ASP and s(CASP). Suppose we have an answer set program with `p(`$\bar{X}$`)` as the top-level goal. If we execute the query `?- p(`$\bar{X}$`)` on s(CASP), then all possible worlds in which `p(`$\bar{X}$`)` holds will be generated by s(CASP) one by one. Note that s(CASP) generates just enough description of the world that is sufficient to prove the query (each partial world is guaranteed to be extensible to a complete consistent world). We can generate counterfactual worlds by issuing the query `?- not p(`$\bar{X}$`)`, which will systematically generate all the worlds in which `p(`$\bar{X}$`)` does not hold. These counterfactual worlds can be useful in improving student understanding of the program that they have develoepd.
Executing the negation of a query is very useful in case the positive query fails. The worlds generated for the negated query help one understand why the positive query failed, namely, what factor(s) contributed to the query's failure. This can help in debugging the logic developed for proving the query, as well as improve a student's understanding of the problem being modeled.
- **Constraint Relaxation**: Constraint relaxation is employed by humans to arrive at a solution if there are so many constraints that there is no solution (solution space is empty). Thus, if we model our knowledge as an answer set program that is over-constrained, and we pose a query against such a

program, then no answer will be produced. Note that there may be other reasons why an answer may not produced. For example, a query may fail due to incomplete knowledge. So over-constraining is not the only reason why a query may fail. If the query fails due to the program being over-constrained, then we may be interested in relaxing enough constraints such that the query will succeed.
The s(CASP) system has a feature where the constraint that caused the top-level query to fail is output. This feature can help the user to interactively figure out the constraints that are causing failure.

- **Consistency Maintenance**: Maintaining consistency is another strategy humans typically use in day to day life. Consistency is maintained by either relaxing constraints (discussed earlier), or making additional assumptions, or altering the premises (rules and facts).
  A system is consistent if it has at least one model (at least one possible world, or one answer set). Often, we want to model a situation and we write the rules, but then we find that the system is inconsistent. We have already discussed how the s(CASP) system can be used to find the constraints and premises that are making our system inconsistent: by relaxing constraints and/or by computing the counterfactual worlds and examining them. Consistency may also be restored by making assumptions. As illustrated earlier, abduction can be used for this purpose. That is, if the absence of a fact `p`($(\bar{t})$) causes inconsistency, we can alternately assume that it is true or false, by adding the following even loop over negation:

```
p(t̄) :- not n_p(t̄).
 n_p(t̄) :- not p(t̄).
```

4. Discussion

We argue in this paper that to teach logical thinking, not only the students must be taught knowledge representation in logic, but also the operational semantics of reasoning. Ideally, this operational semantics should be identical to that of human reasoning. We also argue that the goal-directed ASP system s(CASP) indeed follows an operational semantics that reflects how human reason. Most of the concepts that humans use for reasoning—deduction, abduction, induction, resolution, forward chaining, backward chaining, etc.—can be directly supported in s(CASP). The s(CASP) system supports both negation-as-failure and strong negation, as well as constructive negation. In fact, the s(CASP) system can produce a justification tree automatically for a given query [23] that reflects the reasoning steps followed. This justification tree is really a proof tree. This justification tree closely corresponds to how a human would reason in their mind to prove, for example, that a given query holds. The use of justification tree can be a good way to show students individual steps taken during human reasoning.

In addition, the modeling of various reasoning mechanisms used by humans can allow us to automate most human endeavors. For example, s(CASP)'s predecessor (s(ASP) [24]) has been used to build the CHeF system [25] that models a cardiologist's expertise for treating congestive heart failure. The reasoning used by cardiologists is expressed in about 100 page long guidelines developed by the American College of Cardiology. These guidelines [26] are represented in ASP. The CheF system outperforms cardiologists [27], primarily because it does not suffer from human weaknesses such as a human cardiologist not realizing that a specific lab result changed since patient's last visit. Similarly, ASP technology allows us to answer natural language questions against a textual passage or a graphical image by invoking common sense knowledge [28, 29]. The system has also been used for automation of legal reasoning [30] and to build reliable chatbots [31].

Significant amount of research has been invested in formalizing argumentation using logic programming and answer set programming [32]. However, the logic-based modeling in all these approaches is based on propositions. In contrast, our goal is to be as general as possible. This means that we should handle predicates as well—like the s(CASP) system does. The argumentation literature studies complex argument structures. Our goal is to keep the illustration of human thinking simple—confine it to only default rules, integrity constraints, and assumption-based reasoning.

Significant work has also been done in informal logic [33]. Similar to classical logic, work in informal logic is focused on inference rules (called arguments in the informal logic literature) and reasoning (proof). An argument is an attempt to present evidence for a conclusion (or a claim) and it relies on premises that support the conclusion [34]. Informal logic can be mapped to logic programming and answer set programming and modeled with s(CASP) also [35].

5. Conclusion and Future Work

Computational thinking is deemed as an important skill that every student should acquire. Computational thinking is essentially algorithmic thinking. Since "Algorithm = Logic + Control", we argue that to learn computational thinking, a student must learn *human logic* as well as *human control*. Human logic corresponds to knowledge. Thus, students must be taught how to represent knowledge. "Human control" corresponds to performing reasoning steps to draw conclusions, simplify/transform knowledge, fill in knowledge gaps, etc. We argue that ASP is an excellent formalism to represent "human logic", while the goal-directed predicate ASP system s(CASP) is excellent for illustrating "human control." The work reported here is still in progress. Future work involves developing an extensive set of examples from daily life represented in ASP and the reasoning processes involved illustrated via s(CASP) to teach logical thinking to students. As we emphasize in the paper, we want to focus both on teaching students "human logic" as well as "human control."

References

[1] J. M. Wing, Computational thinking, Commun. ACM 49 (2006) 33–35.

[2] R. A. Kowalski, Logic for Problem Solving, North Holland, 1979.

[3] R. A. Kowalski, Computational Logic and Human Thinking, Cambridge University Press, 2011.

[4] B. Gillon, Logic in Classical Indian Philosophy, in: The Stanford Encycl. of Philosophy, fall 2016 ed., Metaphysics Rsrch Lab, Stanford Univ., 2016.

[5] S. Bobzien, Ancient Logic, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, Metaphysics Rsrch Lab, Stanford Univ., 2020.

[6] G. Boole, An Investigation of the Laws of Thought, Walton & Maberly, 1854.

[7] G. Frege, Grundlagen der Arithmetik, Wilhelm Koebner, 1884.

[8] Wikipedia contributors, Computational thinking — Wikipedia, the free encyclopedia, 2024. URL: https://en.wikipedia.org/w/index.php?title=Computational_thinking&oldid=1246452653, [Online; accessed 10-July-2024].

[9] M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, in: 5th International Conference on Logic Programming, 1988, pp. 1070–1080.

[10] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, TPLP 18 (2018) 337–354.

[11] H. Jeong, A. Taylor, J. R. Floeder, M. Lohmann, S. Mihalas, B. Wu, M. Zhou, D. A. Burke, V. M. K. Namboodiri, Mesolimbic dopamine release conveys causal associations, Science 378 (2022) eabq6740.

[12] G. Gupta, Automating common sense reasoning, July 7, 2020. URL: http://utdallas.edu/~gupta/csg, tutorial talk.

[13] C. Baral, Knowledge representation, reasoning and declarative problem solving, Cambridge University Press, 2003.

[14] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103.

[15] M. Gelfond, Y. Kahl, Knowledge representation, reasoning, and the design of intelligent agents: Answer Set Programming approach, Cambridge Univ. Press, 2014.

[16] G. S. Halford, R. Baker, J. E. McCredden, J. D. Bain, How many variables can humans process?, Psychological Science 16 (2005) 70–76.

[17] A. Cropper, S. Dumancic, Inductive logic programming at 30: A new introduction, J. Artif. Intell. Res. 74 (2022) 765–850. URL: https://doi.org/10.1613/jair.1.13507. doi:10.1613/JAIR.1.13507.

[18] H. Wang, G. Gupta, FOLD-R++: A toolset for automated inductive learning of default theories from mixed data, CoRR abs/2110.07843 (2021).

[19] J. W. Lloyd, Foundations of Logic Programming, 2nd Edition, Springer, 1987.

[20] N. D. Jones, An introduction to partial evaluation, ACM Comput. Surv. 28 (1996) 480–503.

[21] Wikipedia contributors, Causality — Wikipedia, the free encyclopedia, https://en.wikipedia.org/w/index.php?title=Causality&oldid=1248883769, 2024. [Online; accessed 10-October-2024].

[22] W. Starr, Counterfactuals, in: E. N. Zalta, U. Nodelman (Eds.), The Stanford Encyclopedia of Philosophy, Winter 2022 ed., Metaphysics Research Lab, Stanford University, 2022.

[23] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for Goal-Directed Constraint Answer Set Programming, in: Proceedings 36th ICLP (Technical Communications), volume 325 of *EPTCS*, 2020, pp. 59–72. ArXiv:2009.09158.

[24] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, arXiv preprint arXiv:1709.00501 (2017).

[25] Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil, A physician advisory system for chronic heart failure management based on knowledge patterns, Theory Pract. Log. Program. 16 (2016) 604–618.

[26] C. W. Yancey, M. Jessup, et al., Accf/aha guideline for the management of heart failure, Circulation 28(16) (2013) e240–e327.

[27] Z. Chen, E. Salzar, et al., An ai-based heart failure treatment adviser system, IEEE J. of Trans. Engg in Health & Medicine 6, 2800810 (2018).

[28] K. Basu, S. C. Varanasi, F. Shakerin, J. Arias, G. Gupta, Knowledge-driven natural language understanding of english text and its applications, in: Proc. AAAI 2021, AAAI Press, ????, pp. 12554–12563.

[29] K. Basu, F. Shakerin, G. Gupta, Aqua: Asp-based visual question answering, in: Proc. PADL, Springer, Cham, 2020, pp. 57–72.

[30] J. Morris, Blawx: User-friendly goal-directed answer set programming for rules as code, in: Proc. Prog. Lang. and the Law (ProLaLa), 2023.

[31] Y. Zeng, A. Rajasekharan, P. Padalkar, K. Basu, J. Arias, G. Gupta, Automated interactive domain-specific conversational agents that understand human dialogs, in: M. Gebser, I. Sergey (Eds.), Practical Aspects of Declarative Languages - 26th International Symposium, PADL 2024, London, UK, January 15-16, 2024, Proceedings, volume 14512 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 204–222.

[32] P. Besnard, C. Cayrol, M.-C. Lagasquie-Schiex, Logical theories and abstract argumentation: A survey of existing works, Argumentation and Computation vol. 11, no. 1-2, pp. 41-102, 2020 (????).

[33] L. Groarke, Informal Logic, in: E. N. Zalta (Ed.), The Stanford Encycl. of Philosophy, spring 2020 ed., Metaphysics Rsrch Lab, Stanford Univ., 2020.

[34] C. M. Holloway, K. S. Wasson, A primer on argument (2020). URL: https://shemesh.larc.nasa.gov/people/cmh/cmhpubs.html.

[35] G. Gupta, S. Varnasi, K. Basu, Z. Chen, E. Salazar, F. Shakerin, S. Erbatur, F. Li, H. Wang, J. Arias, B. Hall, K. Driscoll, Formalizing informal logic and natural language deductivism, in: Proc. GDE'21 ICLP Workshop, volume 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2970/gdepaper3.pdf.