Puzzle Generation for Action-Adventure Games using Graph Grammars

Mark S. Durrheim
(570169)

Supervised by Prof. Sigrid Ewert and Prof. Abejide Ade-Ibijola

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements of the degree of Master of Science

Johannesburg, October 24, 2022

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University

(Signature of candidate)

2022-10-24

(Date)

To my family, friends, colleagues, and loved ones. Their encouragement has been key to my growth and perseverance.

Abstract

Action-Adventure video games such as *The Legend of Zelda* provide a variety of challenges to players. These include exploration, combat, and puzzles. The player's goal within these games is normally to reach a location, but various obstructions prevent the player from reaching it until they are overcome. By intertwining multiple challenges for the player, a puzzle can be made. We document the elements common to puzzles in action-adventure games. A game could be kept interesting indefinitely if the player was presented with new puzzles each time they played. This would be Procedural Content Generation. We design a system based on node replacement graph grammars capable of designing new puzzles like those seen in action-adventure games. Using this system we produce new lock-and-key puzzles. We also show that we can design a grammar that guarantees a solvable puzzle is produced.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The video game industry has dramatically increased in size in recent years. According to the Entertainment Software Association (2017) consumers in the United States spent $30.4 billion on video games in 2016. This has increased to $43.4 billion in 2018 (Entertainment Software Association, 2019). Making impressive new games requires large teams and years of development. This means that the cost of making games has also increased over time.

To keep players engaged a game designer needs to ensure that the game presents new challenges to the player to keep testing their skills. Designing content for games can be difficult and time consuming. Algorithms can be used to help design content. This makes it easier for game designers to present a variety of challenges to their players. Procedural Content Generation (PCG) can be used to reduce the number of human designers required to produce a game.

Video games are a diverse medium with many different types of games. Similarly content can take various forms. For this research we will be looking at games from the Action-Adventure genre. This research looks at challenges in the form of puzzles, specifically lock-and-key puzzles. These are based on the manipulation of keys to remove a series of locks that are blocking a player's path (Ashmore and Nitsche, 2007; Dormans and Bakkes, 2011). We will investigate an approach to generating puzzles based on breaking down the structures that we observe in sample games.

1.2 Research Aims

Our goal was to create a system that can generate puzzles for action-adventure games. There were three parts to achieving this goal: a puzzle model, a graph grammar, and an application. We explain each task in detail below. The methods that we used are described in Chapter 4.

Model: We have a method of describing a puzzle and all of its elements with a graph. We break puzzles down into three different types of elements: locks, keys, and traversals. These are discussed in more detail in Section 5.1. The method we used to document these puzzles is described in Section 4.2.2.

Grammar: We have designed a graph grammar for each game that we documented. These can generate the graphs that we captured in the data. They can also generate an infinite number of graphs that display similar designs to those in the dataset. Our method for designing graph grammars is discussed in Section 4.3.

We will prove that a well constructed grammar will always produce a puzzle that is solvable. This will require showing that at least one solution exists for any puzzle of that grammar. Note that most puzzles will have many ways of being solved.

Application: We created a computer program that generates graphs based on our graph grammars. This will demonstrate that our plan was effective. It is a command line application and was used to produce all of the diagrams in Chapters 5 and 6. The tools used for this application are discussed in Section 5.1.

With these aims in mind, we shall proceed to explain the foundational concepts that we will be working with around action-adventure games, Procedural Content Generation, and graph grammars.

1.3 Organisation

The rest of this thesis is structured as follows. Chapter 2 provides an overview of the type of games and puzzles that we cover in this research. It also provides background on graph grammars, which we use to create new puzzles. We then provide a review of work related to our research in Chapter 3. The

process that we followed to create our puzzle graphs is described in Chapter 4. Chapter 5 displays the puzzle graphs for the games we analysed and how they could be produced with our grammars. In Chapter 6 display original graphs created by our prototype and the same grammars. The full derivations for all of these graphs can be found in Appendix B. Chapter 7 summarises and concludes the content of our work.

Chapter 2

Background

In this chapter we will explain the core concepts that relate to our work, being the genre of action-adventure video games, Procedural Content Generation, lock-and-key puzzles, and graph grammars.

The chapter is structured as follows. In Section 2.1 we provide definitions for the terminology used in this work. We give an overview of the games that we have analysed in Section 2.2. Lock-and-key puzzles are discussed in Section 2.3. We give background on the field of Procedural Content Generation in Section 2.4. We explain graph grammars by first covering graphs in Section 2.5, then grammars in Section 2.6, and finally graph grammars in Section 2.7.

2.1 Definitions

Video games can be considered more of an art than a science. For this reason some concepts do not have consistent definitions. A definition may have to change from game to game. The definitions that we give here are what are relevant to the games that we covered in this research.

Definition 2.1.1 (Action-Adventure Games)**.** This is a genre of video games. These games typically involve a combination of combat, exploration, and puzzle solving (Adams, 2010). Some examples of action-adventure games include the *Prince of Persia*, *Darksiders*, *God of War*, and *Tomb Raider* series.

Definition 2.1.2 (Content)**.** Video game content is any piece that makes up the game (Hendrikx *et al.*, 2011). Some examples of content would be levels, music, scenery, and character behaviours.

Definition 2.1.3 (Procedural Content Generation (PCG))**.** This is the use of algorithms to generate content for games. In addition it may involve identifying interesting or entertaining instances of generated content (Hendrikx *et al.*, 2011).

Definition 2.1.4 (Logic Puzzles)**.** These are mental challenges that have at least one specific solution. A formal logic puzzle should be solvable through reasoning based on the information provided with the puzzle. Knowledge beyond the puzzle should not be required (Adams, 2010, p. 264).

Definition 2.1.5 (Quest)**.** These are phenomena where the player's character must travel through the game world to fulfil a goal, while overcoming a series of challenges (Ashmore and Nitsche, 2007). In this work the terms mission and quest can be used interchangeably.

Definition 2.1.6 (Level)**.** This definition will vary from game to game. In general a level refers to a distinct area within the game world. The area that constitutes a level may be constrained by the structure of the location, the restrictions of the gameplay, or the amount of content that can be kept in the device's memory at one time (Rouse III, 2005).

Definition 2.1.7 (Dungeon)**.** These are a particular type of level. These tend to have labyrinth-like structures, meaning that they have convoluted and twisting room layouts. Dungeons consist of a contained space containing inter-related puzzles and challenges. (van der Linden *et al.*, 2014). While the term dungeon refers to prison cells, it's meaning in games can refer to caves, castles, and other enclosed environments (Shaker *et al.*, 2016).

Definition 2.1.8 (Platforming)**.** This is the action of jumping between platforms in a game while avoiding obstacles (Adams, 2010).

Definition 2.1.9 (Linear)**.** A level designed with a linear structure requires the player to move through the game spaces in a fixed order (Adams, 2010, pg. 365). This does not mean that the spaces are necessarily arranged in a line. The game only allows the player to progress into the next area once they have completed the area that they are currently in.

Definition 2.1.10 (Lock)**.** A lock is an obstacle that prevents the player from making further progress through a level until a requirement is met. See the definition of *key* below. The player must get to the key, before they can pass through the lock.

Definition 2.1.11 (Key)**.** A key is an object in the level that will allow a lock to be opened. These come in two forms: items and switches. Items are keys that the character will pick up and take to the lock to make it open. Switches are objects that can be interacted with to open the lock, but do not need to be moved in the game world.

We have now covered the important terminology that we will be using in later sections. Below we will proceed with discussing the games and techniques that are used in this research.

2.2 Games of Interest

We have examined four games for our research. These are considered to be examples of the Action-Adventure genre. We give a brief overview of each game below.

2.2.1 Prince of Persia: The Sands of Time

This game was created by Ubisoft Montreal (2003). A screenshot from the game can be seen in Figure 2.1. The player controls a character known only as *Prince*.

The story of *Prince of Persia: The Sands of Time* follows a character only referred to as *prince*. The game starts with the prince and the Persian army sacking an Indian city. While attacking the city, the prince finds a dagger that can reverse time among the treasures of the city. The prince is tricked into using the dagger to unleash the *Sands of Time* from a mystical hourglass. The sands corrupt all people that they touch, turning them into monsters. Only the prince and Farah, the princess of the city which was sacked, are safe from the curse of the sands due to the prince carrying the *Dagger of Time* and Farah wearing the *Medallion of Time*. Together the prince and Farah must find a way to return the sands to the hourglass and break the curse.

Prince of Persia: The Sands of Time is focussed on combat and platforming. Most of the puzzles are based on using the Prince's amazing agility to

Figure 2.1: Prince of Persia: The Sands of Time (Minarik, 2011)

reach switches. The game is linear in its structure. The player moves from area to area completing challenges. There are no options about the path to take.

2.2.2 The Legend of Zelda: Link's Awakening

This game was released in 1993 for the *Game Boy* by Nintendo (1998a). It was re-released as *The Legend of Zelda: Link's Awakening DX* for the *Game Boy Color*. The *DX* version can be seen in Figure 2.2.

In *The Legend of Zelda: Link's Awakening* the player controls a character commonly known as Link (Wolf, 2012, pg. 360-365). The game starts when Link is washed onto the shore of an island after his boat was sunk in a storm. They cannot leave until they have woken the Wind Fish. This requires them to first gather the eight instruments of the siren.

The gameplay has a balance of combat, exploration, and puzzle solving. There is an *overworld* which the player is free to explore using the tools that they have gathered in their adventures. The instruments are hidden in dungeons (see Definition 2.1.7) scattered throughout the world.

Figure 2.2: The Legend of Zelda: Link's Awakening DX (Zena, 2013)

Figure 2.3: The Legend of Zelda: Ocarina of Time (Odow, 2015)

2.2.3 The Legend of Zelda: Ocarina of Time

This game was released in 1998 for the *Nintendo 64* by Nintendo (1998b). A screenshot is shown in Figure 2.3.

The player controls a young boy, who grows up during the course of the game. While each player can give their own name to the protagonist, he is commonly referred to as Link in popular culture (Wolf, 2012, pg. 360-365).

In the story of *The Legend of Zelda: Ocarina of Time*, Link must defeat the evil king and wizard known as *Gandorf*. Link is the *Hero of Time* who is destined to save the land of *Hyrule* from the evil of Gandorf. Link must gather the sages who will lend him their power to overcome Gandorf.

The gameplay of *The Legend of Zelda: Ocarina of Time* is a balance of combat, exploration, and puzzle solving. The player can freely explore the *overworld* using the tools that they have gathered in their adventures. The objectives of Link's quest are hidden in dungeons scattered throughout the world.

Figure 2.4: Tomb Raider (Square Enix Limited, 2012)

2.2.4 Tomb Raider

This game was made by Core Design (1996). A screenshot can be seen in Figure 2.4.

The player takes control of the character *Lara Croft*, a legendary archaeologist and adventurer. She is hired to find a mysterious artefact in an ancient Peruvian tomb. It turns out that the artefact is part of a larger and more powerful item. Lara must explore ancient ruins in South America, Greece, and Egypt to unravel the truth about Atlantis.

The gameplay of *Tomb Raider* is focused on combat and platforming. Over the adventure Lara has to navigate ancient tombs filled with traps and deadly creatures. Lara will fight dangerous animals, villainous rivals, and even Atlantean monstrosities.

2.3 Lock-and-Key Puzzles

The games that we have discussed possess some similarities in their level structure. We call this structure a lock-and-key puzzle. In this section we will describe the structure of these puzzles and how they are played.

2.3.1 Puzzle Structure

Lock-and-key puzzles give players the task of reaching a goal, but their way is blocked by one or more locks. Each lock has a matching key that needs to be found to open it (Ashmore and Nitsche, 2007; Dormans and Bakkes, 2011). Locks and keys can be interpreted as many different objects. The core concept is that the player cannot pass the lock until they acquire the key. Some examples of alternative objects are: a ladder to climb a high wall, a boat to cross a rushing river, or a disguise to sneak into an event. Games belonging to the *Metroid* and *Castlevania* series are great examples of the lock-and-key structure (Ashmore and Nitsche, 2007). These two games helped define a style of game; this resulted in the genre being called *Metroidvania*, as explained in the quote below.

"Let me define a Metroidvania. Back in the day most games were split up into levels and they followed one after another in order, then games like Metroid came along, squished the levels together into one contiguous world map and jumbled up the order. This means that during your adventure you might come across a door that you can't access yet until you unlock a new item

that will let you through. So you now have to retrace your steps to get back to that obstacle and bypass it." - Mark Brown (Brown, 2014, timestamp 0:25)

The layout of the elements of lock-and-key puzzles is important. In a backward puzzle the solution (key) is found by the player before they encounter the problem (lock) (Adams, 2010, pg. 564). Too many situations like this will cause a puzzle to become trivial and thus boring.

In an action-adventure game it is common to include challenges like traps and combat into a puzzle environment. For example, the switch that the player needs to flip is guarded by monsters. This does not change the structure of the puzzle, but adds variety to the player's experience. These are covered by traversals, which are discussed below.

2.3.2 Components of a Lock-and-Key Puzzle

Lock-and-key puzzles have three main components: locks, keys, and traversals. We discuss these in detail below. There are also components of levels that may resemble lock-and-key structures, but are not part of the navigational puzzle itself. We discuss these excluded challenges as well below.

Traversals

A traversal is a space in which the player can freely move their character to and from any point in that area using the tools available to them while playing. The space may consist of many rooms. We only mark boundaries that the player would be unable to cross when they first enter the traversal space. This is a variation on how Rees (2004) defines areas and Johansen (2016) defines state spaces.

Abstracting the physical layout of a level lets us focus on the structure of the player's mission. The physical space can be created later using other generative methods (Dormans, 2010).

This space will normally contain challenges for the player. Challenges can include fighting enemies and platforming sections. Platforming is form of challenging movement as defined in Definition 2.1.8 (pg. 8). These require the player to use their knowledge and skill at the game to pass.

Since the player can be expected to overcome these challenges with the in-game tools that they would already have at their disposal, these challenges are not considered to block the player's ability to move through the space.

A literal lock and key may be present in a space and considered to be part of the traversal if there is no significant obstacle between them. The lock does not impede the player's ability to traverse the space unless they have to overcome a challenge to open it.

An obstacle that you already have the item for is not included if you could not have reached the point without the item. Locks which have the key from another part of the game are not included.

Locks

Locks separate traversals. They represent an obstacle that the player cannot overcome by using the resources that they possess when they first enter the space that contains the lock.

The player can only pass through a lock when they have all of the required keys for that lock. This will allow the player to move into a new traversal space.

A lock has directionality. Once opened, the player is allowed to access a different traversal. Some locks operate in one direction only and will not allow a player to go back to the traversal that they came from after passing through.

Keys

A key allows the player to pass through its associated lock after it has been activated. Keys are situated within a traversal and generally in a different direction from the entrance as the lock is.

A key can be activated in different ways depending on how it is represented in the game world (Adams, 2010, p. 264). It could be activated by picking up a tool to use on the lock, otherwise it could be activated by pressing a button that opens a door or defeating a specific enemy which prevented the lock from opening.

A key is associated with a specific lock. In a game there may be situations where a single key can open multiple locks. This can often be simplified be treating them as a single lock. The implications on the puzzle structure remain the same with this abstraction.

In the game world, the keys and locks can take many forms, but their association remains the same as explained in the quote below.

"As we can see, a dungeon in Link's awakening is all about keys and locks. Small keys are many keys for many locks, the nightmare key is one key for one lock, and then the item is one key for many locks. You can even add in the crystal switches, those orbs that raise and lower all the blocks in the dungeon, which are like many keys for one giant lock" - Mark Brown (Brown, 2016a).

Excluded Structures

For the purpose of this research, we are only concerned with the critical path of the puzzle. This means that any section of the level that the player is not required to pass through can be ignored. We describe three types of concepts below that have lock-and-key structures, but that we exclude due to them not affecting the critical path.

Bonus areas are small parts of the level that provide extra resources that will help the player, but are not required for them to progress. These often require additional challenges to be overcome, or special keys to be found. These do fit a lock-and-key structure, but since they are not required for the completion of the level, they are excluded from our model.

Movement challenges are timed platforming obstacles. Generally the player activates a switch which temporarily opens a door, giving the player a limited amount of time to make a series of challenging platforming actions to reach the exit in time. Depending on the scale of the challenge and whether or not the player can pass back into the area afterwards, we will treat this as part of the traversal or as a one-way lock.

Short-cuts are paths that connect an area near the start of the level to a much later level. This aids the player's navigation in the space, but does not expand the area that they are able to access, so does not affect the critical path.

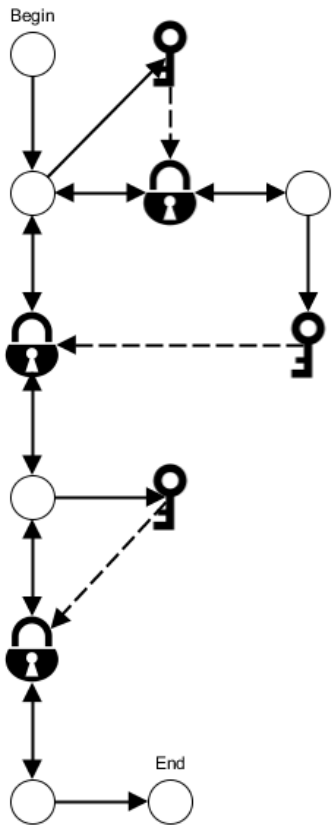


Figure 2.5: Puzzle graph for the level *Inside the Deku Tree* from *The Legend of Zelda: Ocarina of Time*

2.3.3 Interpreting Lock-and-Key Puzzle Diagrams

A puzzle graph is an abstract representation of the structure of a level using graphs. We discuss graphs in more detail in Section 2.5. An example puzzle graph can be seen in Figure 2.5. In this section we establish how to interpret these diagrams.

The player will begin at the Begin node and their goal is to reach the End node. There are rules for how the player can move through the puzzle graph. These are a variation of the rules used in PuzzleGraph (Johansen, 2016), which is discussed in Section 3.2.

There are five types of nodes in our representation: Begin, End, traversals, locks, and keys. These rules represent what the player can do at each type of node. It can be useful to visualise the puzzle graph as a board game. We represent the player as a token that can move between nodes.

In all cases the player must obey the edge direction. They can only move in the direction that the arrows indicate. In most cases there will be arrows in both directions.

- Begin node: The player will always start at this node. They will be able to move to the adjacent traversal.
- End node: When the player reaches this node, the puzzle has been completed.
- Traversals: These represent the space that the player is moving in. While at a traversal, the player can do an action with an adjacent node. The player can activate a key node, pass through an open lock to another traversal node, or move to the end node. In our diagrams, we represent them with circles.
- Locks: These restrict the movement of a player. The player can only move through a lock if it is open. A lock is open once all of the keys adjacent to it have been activated.
- Keys: These can be activated by the player when they are in an adjacent traversal, meaning that there is an edge between the traversal and the key. In our representation a key cannot be deactivated.

2.3.4 Solvable Lock-and-Key Puzzles

The player must be able to complete the game that they are playing. That means if a level can be represented by a lock-and-key puzzle, then there must be a solution to that puzzle. A lock-and-key puzzle is considered solvable if the player can reach the final traversal. In a game this typically contains the exit for the level and/or a treasure that is part of the game's larger quest.

We will consider two configurations that result in unsolvable lock-and-key puzzles, i.e, deadlocks and missing keys. These are to be avoided when designing a puzzle.

A deadlock occurs when you have two doors, A and B, with their respective keys, a and b. If key a can only be reached by passing through door B and key b can only be reached by passing through door A, then we have a deadlock.

The simplest case for a deadlock is with two locks, but it extends to any number or sequence of doors. In essence, this can be stated as: a deadlock occurs when the tools for unlocking a door are behind the locked door, making them are unreachable.

A missing key occurs when the key for a lock does not exist. This results in a lock that can never be opened, and makes progress impossible.

We prove that our grammar for *Prince of Persia: The Sands of Time* will always produce a solvable puzzle in Section 5.2.3.

2.4 Procedural Content Generation

In this research we will use algorithms to design puzzles like those that we described in Section 2.3. This is an example of Procedural Content Generation.

Procedural Content Generation is the use of algorithms to create game content (see Definition 2.1.3). Content (see Definition 2.1.2) can take many forms in video games (Hendrikx *et al.*, 2011).

PCG was first used in the video game Elite Acornsoft (1984) as a method to save space on disk. The full universe of this game would not fit on the disk that the game came on, so they devised a method to create the universe when the game was run on the user's computer.

2.5 Graphs

We have established that we want to generate lock-and-key puzzles. We need to represent these puzzles with a data structure that can be generated by an algorithm. Graphs are a versatile structure that have already been used to represent lock-and-key puzzles (Adams, 2002; Ashmore and Nitsche, 2007; Coxen and Baylis, 2012b; Ashmore, 2006). Examples of research where graphs have been used to represent lock-and-key puzzles can be found in Chapter 3.

Graphs are data structures that represent objects and the connections

between them (Dale and Lewis, 2011). A graph consists of nodes and edges. A node represents an object, while an edge corresponds to a connection between two nodes. An edge can be undirected or directed, indicating whether the connection is the same in both directions or represents a one way connection.

Graphs are suitable for representing any system where objects are connected in some way, for example, computers in a network, cities and the roads between them, and people's friendships on a social network.

2.6 Grammars

There is a wide variety of techniques that can be used to generate content, as was discussed in Section 2.4. We have chosen to use grammars for our research. They are well suited to generating the kind of content that we desire as is seen in Chapter 3.

In the field of Formal Languages a language is defined as the set of all strings that can be produced by a grammar (Martin, 1991). A grammar consists of an alphabet and a collection of production rules. The symbols of the alphabet are divided into two mutually exclusive sets: terminal and non-terminal. One non-terminal symbol is designated as the start symbol. The rules specify symbol substitutions from a non-terminal symbol to one or more symbols from either set. A string is formed by taking the start symbol and applying the production rules until only terminal symbols remain.

Grammars can be used to specify more than strings of text. For example, shape grammars have been used to generate buildings and cities (Müller *et al.*, 2006), and Lindenmayer systems have been used to simulate natural growth and can produce structures like trees. An example of this can be seen in Figure 2.6.

2.7 Graph Grammars

Graph grammars come in two primary forms: node replacement and hyperedge replacement. The latter is used for hypergraphs, which is a generalised form of a graph where an edge can connect multiple nodes.

A production rule for a graph grammar has three components: the mother and daughter subgraphs, and the embedding method (Rozenberg, 1997). The graph which we apply a production to is called the host graph. A production

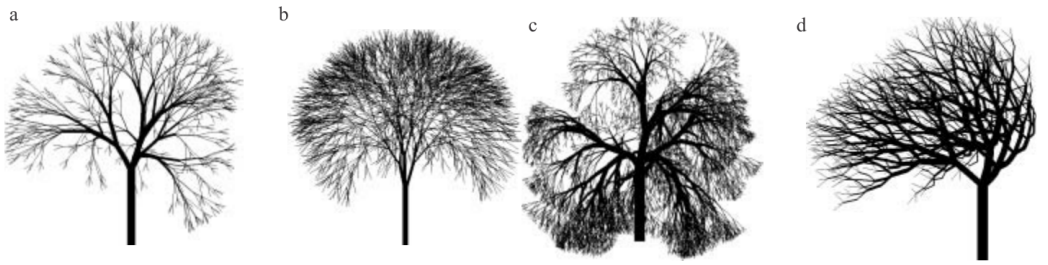


Figure 2.6: Trees generated by Lindenmayer systems (Prusinkiewicz and Lindenmayer, 1990, Figure 3.2, pg. 60)

can be applied to a graph if the mother subgraph is present in the host graph. The mother subgraph is removed from the host graph and the daughter subgraph is inserted in its place. The daughter subgraph is then connected to the remainder of the host graph according to the rules of the embedding technique.

Productions for graph grammars are defined as (M, D, E) (Rozenberg, 1997). Symbols M and D are both graphs and E is an embedding method. A production can be applied to a graph H if there is an occurrence of subgraph M in the host graph H. M is removed from H to make H^-. A copy of D is then added to H^-. E determines how to attach the nodes of D to the nodes of H^-.

There are two main embedding techniques used in graph grammars. These are known as *gluing* and *connecting*, or alternatively *algebraic* and *algorithmic*, respectively (Rozenberg, 1997).

2.7.1 Node Replacement using Connecting Embedding

In a node replacement graph grammar we replace a non-terminal node with a subgraph, which we then connect to the rest of the graph based on how the original node was connected. There are many types of these grammar classes. For the example below we use a simple type called Node Label Controlled (NLC) Grammar. This also demonstrates a connecting type of embedding mechanism.

Figure 2.7 displays a graph grammar production. The production states that the mother node, labelled S, will be replaced with the daughter subgraph, which consists of two nodes, a and c connected by an edge.

We will apply the production to the host graph as seen in Figure 2.8.

Figure 2.7: Sample graph grammar production

We take note of the nodes which are neighbours of S. Those are the set of nodes which have an edge connecting them to S. The host graph, with all the neighbours of S circled can be seen in Figure 2.8.

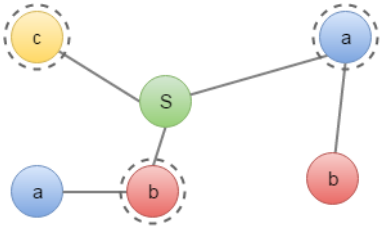


Figure 2.8: The host graph. Neighbours of the mother node S are circled.

To apply the production, we first remove the mother node and all edges that connect to it. Then the daughter subgraph is inserted into the host graph. This is shown in Figure 2.9.

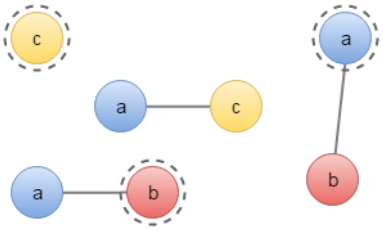


Figure 2.9: The node S has been replaced by the daughter subgraph.

We have embedding rules that determine how we connect the nodes of the daughter subgraph and the host graph. Each production has its own embedding rules. Let us say that for the production in Figure 2.7, they are $\{(b, a), (a, c)\}$. The rule (b, a) states that each node that was in the neighbourhood of S and has the label b should have a edge to each node in the daughter subgraph that has the label a. Similarly (a, c) connects all nodes that were previously connected to S with the label a to each node in the daughter subgraph with label c. The results of the embedding process can be seen in Figure 2.10.

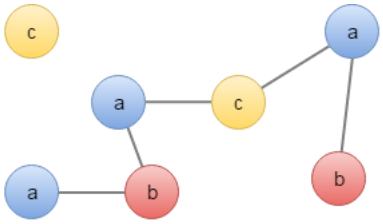


Figure 2.10: The subgraph is embedded.

There are more powerful types of node replacement graph grammars, most notably the edNCE grammar. The acronym means that they use Neighbourhood Controlled Embedding and make use of edge directions and labels in their connection rules. The process of applying an edNCE grammar is the same as that for an NLC grammar, but there are more conditions controlling the embedding mechanism.

2.7.2 Neighbourhood Controlled Embedding

In the preceding section we presented an NLC grammar. For this work we will be using a more complex type of grammar to describe lock-and-key puzzle graphs. We will use a Neighbourhood Controlled Embedding Directed graph grammar with the Boundary constraint (B-dNCE) (Rozenberg, 1997), which we describe below.

A B-dNCE graph grammar is a node replacement graph grammar like the NLC graph grammar described in Section 2.7.1 above. The substitution of the mother node with the daughter subgraph works the same as in NLC graph grammars. There are differences in the embedding rules.

Neighbourhood controlled embedding allows us to differentiate between nodes in the daughter subgraph that have the same label. In a NCE grammar rule we number the nodes in the daughter subgraph. An example of this is shown in Figure 2.11.

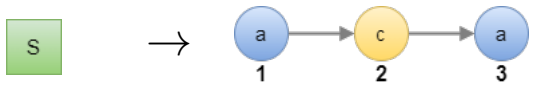
Embedding into a directed graph allow us to have different rules based on edge direction. We split the neighbours into two sets based on how they are connected to the mother subgraph M. If there is an edge from the neighbour node to M then that node is an in-neighbour. If there is an edge from M to the neighbour node then it is an out-neighbour.

In a boundary graph grammar there are no edges allowed between nodes with non-terminal labels. This ensures the context-freeness of the grammar.

That means that the order in which productions are applied does not affect the derivation tree.

An embedding rule of a dNCE graph grammar has the form $(\mu, d/d', x)$. Here, μ is the label neighbouring node of M in H^-. Moreover, x is the number of the node in the daughter sub-graph. Additionally, $d \in \{in, out\}$ is whether the neighbouring node is an in-neighbour or out-neighbour of M. Finally, $d' \in \{in, out\}$ is the direction of the edge created by the embedding rule.

We will present an example of a dNCE production being applied below. The host graph is shown in Figure 2.12. The neighbours of the daughter subgraph have been circled and labelled on whether they are in-neighbours or out-neighbours.The example production is shown in Figure 2.11.



(b, in/in, 1)
(c, in/in, 1)
(a, out/out, 2)
(c, in/in, 3)

Figure 2.11: Example dNCE production

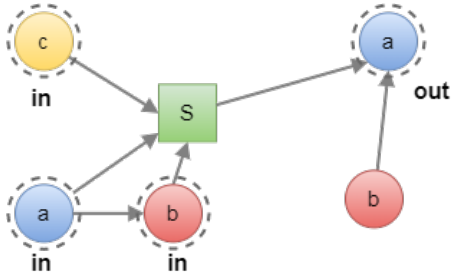


Figure 2.12: The host graph. Neighbours of the mother node S are circled.

To apply the production, the mother node and all edges connected to it are removed and the daughter subgraph is added as shown in Figure 2.13. This is exactly the same as described for NCE graph grammars in the previous section. The embedding rules are interpreted as follows.

Embedding rule *(b, in/in, 1)* creates an edge from an in-neighbour nodes with label b and connects it to subgraph node 1. Similarly, the embedding rules *(c, in/in, 1) and (c, in/in, 3)* create edges from all in-neighbour nodes with label c and connects them to subgraph nodes 1 and 3 respectively. Embedding rule *(a, out/out, 2)* creates an edge to an out-neighbour node

with label a. Note that the in-neighbour node with label a is not affected by this. The graph after the embedding rules are applied is shown in Figure 2.14.

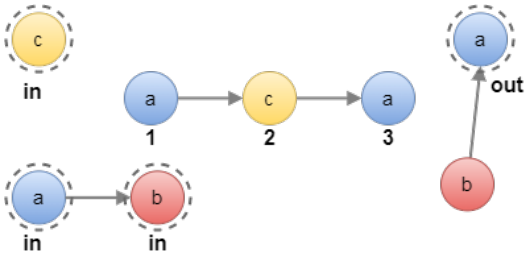


Figure 2.13: The node S has been replaced by the daughter subgraph.

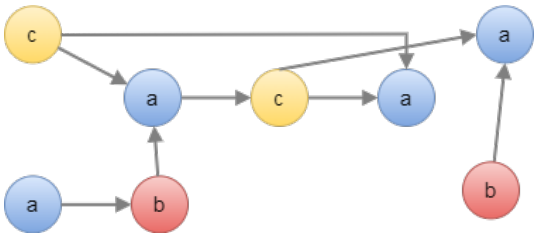


Figure 2.14: The subgraph is embedded.

The B-dNCE example shown above demonstrates the basics of the grammars that we created in Chapter 5. We will first cover some related work in the following chapter and more detail on our methodology in Chapter 4.

Chapter 3

Related Work

In this chapter we will review research that has been done on generating lock-and-key puzzles. A variety of techniques have been used by different authors. We have put a focus on works using grammars and research that also focusses on action-adventure video games.

3.1 Zelda Dungeon Generator

The *Zelda Dungeon Generator* is a system developed to produce missions (see the definition of a quest in Section 2.1) and maps in the style of the *Legend of Zelda* series of action-adventure video games (Lavender, 2015). The system generates missions using graph grammars. The maps that are created to fit these missions are then generated using shape grammars. This work expands research done previously by Dormans and Bakkes (2011).

3.2 PuzzleGraph

PuzzleGraph is an open source tool that allows for the design and analysis of puzzles as seen in games like *Lara Croft and the Guardian of Light* (Johansen, 2016).

The puzzle is represented as a graph. The player can traverse the graph and interact with the elements contained at each node. These interactions include pressing buttons, flipping switches, and even shooting switches from afar to flip them. The buttons and switches would be used to open and close doors as well as drawbridges.

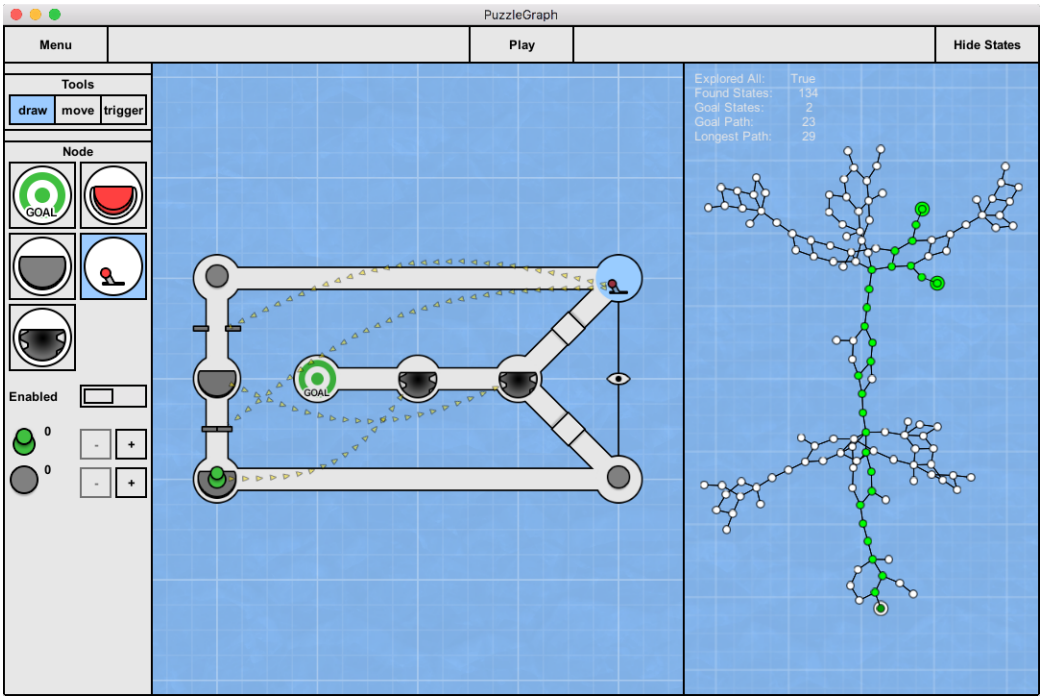


Figure 3.1: Screenshot of *PuzzleGraph* (Johansen, 2016)

PuzzleGraph also allows for the player to move boulders around to operate buttons as is done in the game *Lara Croft and the Guardian of Light*. This means that there can be many moving pieces in the puzzles that the application can render.

A screenshot of *PuzzleGraph* can be seen in Figure 3.1. The centre panel displays a puzzle that can be played. The left panel shows the tools used to construct the puzzle. The right panel displays the state-space graph for the puzzle, where green nodes indicate the path to the solution.

3.3 Dungeon Generation System

Adams (2002) created a *Dungeon Generation System*. This uses graph grammars to create levels for first-person shooter games. This type of game presents the player with a three-dimensional world as seen through the eyes of the character. Such games generally consist of moving through levels and shooting enemies. The final output of the system is still in graph form.

Content evaluation algorithms measure the size, difficulty, and expected fun of the level.

3.4 Machinations

Machinations is a framework that represents the rules of games with petri nets (Dormans, 2009). These are graphs where tokens are tracked at the nodes. Events can trigger tokens to transition between nodes along edges, or have tokens be created or destroyed. The tokens represent game resources that are moved between pools. This framework provides a model of the flow of the game based on its rules. This is particularly useful for analysing board games such as *Risk*.

3.5 Playtracer

Playtracer is a tool designed to visualise data gathered by people playing a game (Andersen *et al.*, 2010). The tool records each state that a player or players reach and the state that they reach next. Using a distance metric to determine the magnitude of the difference between states, a two-dimensional state transition graph is produced. These diagrams can provide insight into situations where players get confused while playing a game. By identifying such problematic states the designers of a game may identify what causes players to fail there. Modifying the game to reduce confusing situations will improve player experience.

3.6 Charbitat

Charbitat is an experimental game that produces an infinite 3D world. Ashmore and Nitsche (2007) prototyped a system that structures the placement of obstacles and the items needed to overcome them based on lock-and-key puzzles. Graphs describe the quest structure within *Charbitat*.

3.7 Metazelda

Metazelda is a tool developed by Coxen and Baylis (2012b). This tool generates lock-and-key puzzle graphs similar to those seen in *The Legend of Zelda* games. A screenshot of this tool can be seen in Figure 3.2.

This algorithm works by adding nodes onto the grid to create a spanning tree of the level, which is a sub-graph in the form of a tree that uses all the same nodes as the graph but can use fewer edges (Balakrishnan and Ranganathan, 2012, pg. 74). The locks and keys are then added to this tree. Each room can have up to four neighbours due to the graph being laid out on a grid. This aligns to how early *Legend of Zelda* games had their rooms laid out.

The algorithm uses the concepts of key-levels. This organises the level into sections based on a sequence of keys along the critical path. The critical path is the route that the player must travel to reach the end of the level, excluding all optional detours. Formally, the key n can only be reached by a player if they have all keys *1* to *n-1*. The rooms accessible to a player with keys *1* to *n-1* are labelled as key level n. Mark Brown uses a similar concept in his Boss Keys series, discussed in Section 3.8.

Metazelda models a switch mechanism. This means that there is a room where the player can alternate the switch between two states. Some edges in the puzzle graph can only be traversed while in a specific state. In Metazelda there can only be one such switch in the entire level.

The algorithm described in Metazelda was refined and integrated into a game developed by the authors (Coxen and Baylis, 2012a). The game was released as *Lenna's Inception* (Bytten Studio, 2020) in January 2020 (Coxen and Baylis, 2020).

3.8 Boss Keys

Mark Brown, a games journalist, has created a series of videos on YouTube called Boss Keys. These analyse the dungeon design in the *Legend of Zelda* games. The series has been expanded to include similar games, such as *Metroid* and *Castlevania*.

Lock-and-key graphs were used to analyse the design of the dungeons. An example dungeon graph is shown in Figure 3.3. In this representation, diamond symbols represent the keys and the squares indicate the locks. A

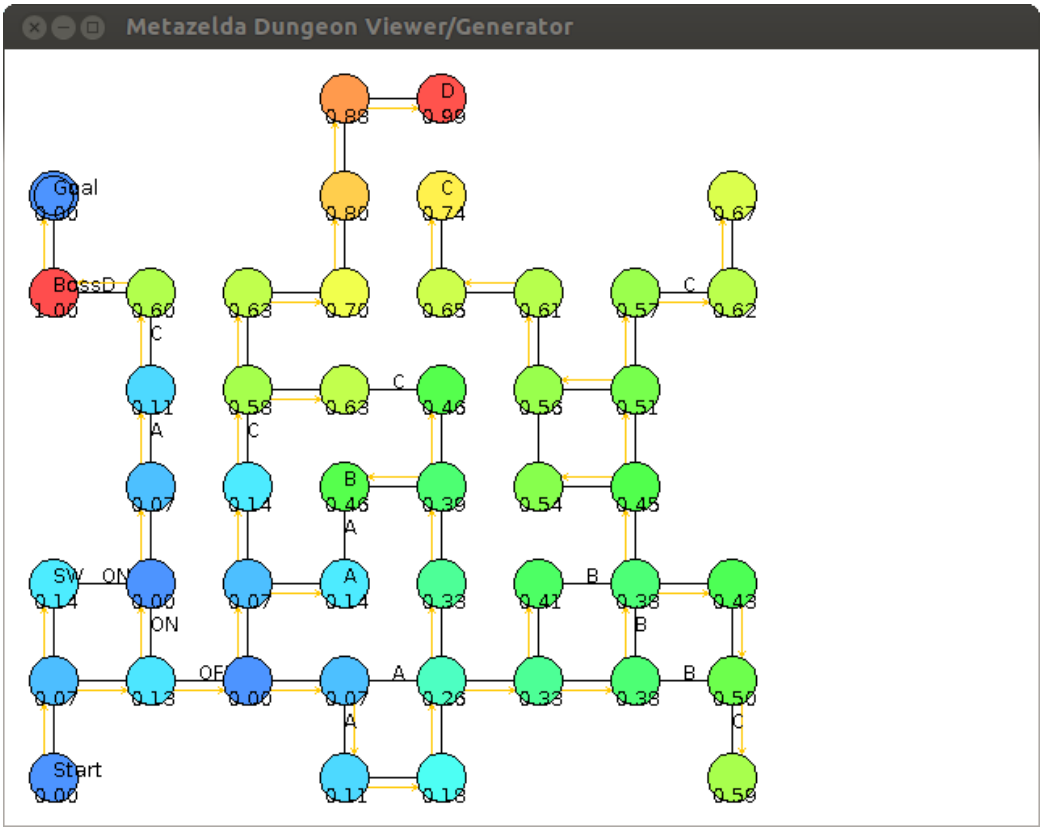


Figure 3.2: Metazelda screenshot taken from Coxen and Baylis (2012b)

lock is opened with the key that has a matching symbol.

We have covered a variety of previous works. In the next chapter we will discuss how we used the concepts discussed above in this research.

3.9 Evolutionary Algorithms for Locked-door Missions

Pereira *et al.* (2021) present a method of generating dungeons for action-adventure games using an evolutionary algorithm. This means that a population of candidate levels are created and evaluated by a fitness function. The low scoring levels are discarded and mixed together and some variations are introduced before a new round of evaluations is done. After many iterations it is expected that levels optimising the factors valued by the fitness function will be produced.

The evolutionary algorithm produces trees that represent the level, which are converted into a grid representation afterwards. The lock-and-key system that they use allows any key to open any lock, but each key is only usable once. This matches how small keys function in the *Legend of Zelda* game series.

Variations of the path finding A* algorithm (Heineman *et al.*, 2008, pg. 201) and depth-first search algorithm (Dale and Lewis, 2011, pg. 263) are used to simulate players navigating the generated levels. The results of these simulations feed into the fitness function that is used to determine how well the level fits the desired criteria of the authors.

The authors created a prototype game that would allow people to play through levels generated by their method. They got 70 players to play a set of levels generated by the evolutionary algorithm and some created by humans. Based on feedback they found the players perceived the procedurally generated levels as more difficult, fun, and human-made than the human-created levels.

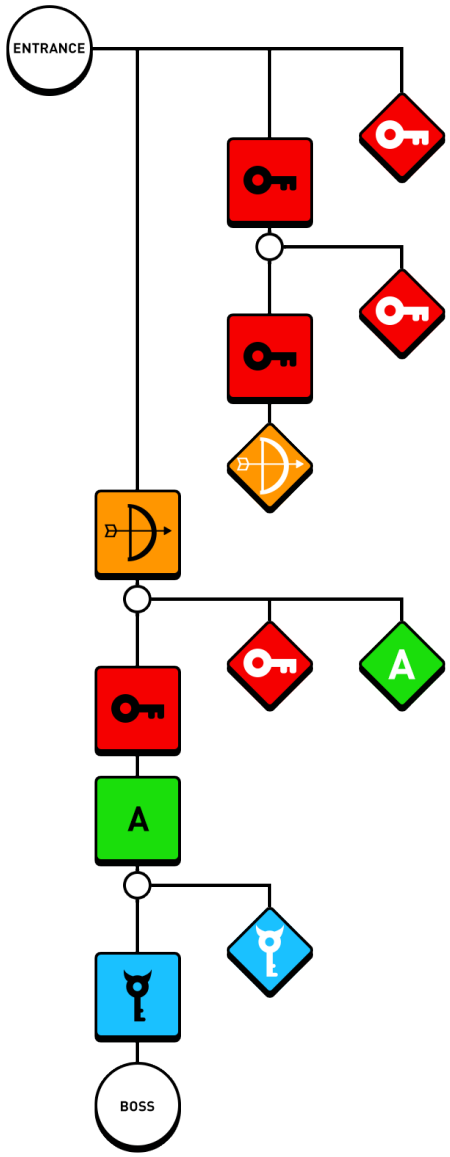


Figure 3.3: Dungeon Graph for the Earth Temple in *The Legend of Zelda: The Wind Waker*, taken from (Brown, 2016)

Chapter 4

Methodology

In this chapter, we explain how we achieved our goal of generating puzzle graphs like those from real Action-Adventure games. In Section 4.1 we explain the notation that we use for lock-and-key puzzle graphs. Section 4.2 gives an overview on how we documented the puzzle graphs from each game. Section 4.3 explains how we designed graph grammars for generating puzzles. The definitions and methods that we used evolved over the course of this research.

4.1 Notation

Our lock-and-key puzzle diagrams function similarly to those presented in PuzzleGraph (Johansen, 2016) and Boss Keys (Brown, 2021). We could treat the player as a token that can travel between traversals on the map. The player can activate a key that is adjacent to their current traversal. The player can only move through a lock if all the keys pointing to that lock have been activated. The player starts at the *Begin* node. They complete the puzzle when they reach the *End* node.

The layout of nodes was handled by the *Graphviz* package. We did not have complete control over the process. This means that sometimes the layout of a graph will change dramatically between steps.

Figure 4.1 shows the symbols that we use in lock-and-key diagrams.

The nodes have been numbered to make the process easier to follow. The node number is equal to how many nodes have been created so far. The example in Figure 4.1c shows a traversal if it was the first node created in

(a) Key. Icon by sbed (2013)

(b) Lock. Icon by Lorc (2009)

(c) Traversal

(d) Non-terminal

(e) Begin node

(f) End node

Figure 4.1: Lock-and-Key Puzzle Graph Grammar Notation

the graph.

The interpretation of puzzle graphs is discussed in Section 2.3.3. We will now present the grammars that we have created.

4.2 Game Analysis

We needed examples of puzzle graphs so that we could design the graph grammars. In this section we explain the process of extracting puzzle graphs from video games. We documented the puzzles by observing them while the game was being played.

Playing the games could prove to be challenging at times. A difficult fight could take many tries over an hour to win. Losing a fight would be a setback. The ability to save states in the emulators alleviated this issue. Another valuable resource was guides written to help players with the games. Both of these resources are discussed below.

Some video games were released exclusively on specialised video game platforms. Emulators are software that simulates that platform so that the games can be run on a regular computer. An additional benefit of using an emulator is being able to save the state of the game at any moment. Many

games will only allow the player to save their state at specific points. This means that failing a while after a save point can result in losing all progress gained in the intervening time. This can result in repeating the same section of the game for hours until succeeding. We could save at any point and this saved us many hours during our research.

4.2.1 The Games

We analysed four games. We needed to understand the layout and dynamics of each level so that we could draw lock-and-key graphs that represented them. In this section we explain the resources that we used to analyse each game.

Prince of Persia: The Sands of Time

See an overview of the game in Section 2.2.1. This game has a very linear level design. This means that the player has few options about the direction that they will go to proceed with the level. This made it easy to use a video of someone playing through the game to map the level layout. The game has long platforming (Definition 2.1.8) and combat sections. We could skip through these parts of the video and get to the parts that we needed. We used a recording of the full game being played. This was uploaded to YouTube by a user called HardlyAstounding (2012).

The Legend of Zelda: Link's Awakening

See an overview of the game in Section 2.2.2. To document the puzzles in *The Legend of Zelda: Link's Awakening* we played through the game. All of the puzzles took the form of dungeons. We played the *DX* version of the game. This does not affect the puzzle structures at all. The game was run using the emulator program *VisualBoyAdvance* (VBA Team, 2004). One of the dungeons from this game was used as an example of how we break down puzzles. This can be seen in Section 4.2.2. If we found that we were having difficulty making progress in the game we would look for help on www.ZeldaDungeon.net (Zelda Dungeon Team, 2001). The parts of the game that take place between the dungeons are not part of our analysis.

The Legend of Zelda: Ocarina of Time

See an overview of the game in Section 2.2.3. We followed the same procedure as was used for *The Legend of Zelda: Link's Awakening*. This is described above.

The Legend of Zelda: Ocarina of Time was released for the *Nintendo 64* gaming console. We did not have access to one of these devices and used a piece of software called *Project64* (Project64 Team, 2005) instead. This allowed us to run the game on a computer. One of the benefits to the emulator, like *VisualBoyAdvance* above, is the feature of full save states. This feature lets us save the state of our game at an exact moment, and restore the game to that point, effectively stepping back in time. Normally being defeated would set the player back a few minutes, even half an hour. This could become very frustrating during difficult parts of the game. We used save states to ensure that we only lost a minute or two at a time. This meant that we could progress much faster than a player normally would since we bypassed the time that would have been needed to recover from setbacks.

Tomb Raider

See an overview of the game in Section 2.2.4.

Watching videos of other people playing a game can be frustrating. They can waste time by making mistakes.

We used videos by a YouTube user called badassgamez (2013a,b,c, 2014a,b). These were very useful since the player made very few mistakes. The *[No Meds]* in the video title refers to the fact that he will not use any medical kit items. Typically, if the player makes a mistake and gets the character hurt this damage can be fixed with a medical kit. Hence the *[No Meds]* challenge is a symbol of mastery over the game.

Another issue with watching footage of a game is that it can sometimes be unclear which door has been opened by a switch. We use game footage from badassgamez (2013a,b,c, 2014a,b), in which they explain what is happening as they play.

4.2.2 Breaking Down a Puzzle

We needed to create lock-and-key graphs that represent the levels from action-adventure video games. To ensure we were consistent in how granular the

diagrams were, we developed a method of breaking down the game levels.

We used a method similar to that of Brown (2018) in that we played the game and marked the locks and keys on a game map, before laying it out into a graph. The method that we followed works as follows:

1. Play and map the level
2. Decide which items are significant
3. Convert the map to a lock-and-key diagram

We discuss each of these steps in detail below and work through an example of breaking down a level in Section 4.2.3.

Play and Map the Level

To create a lock-and-key diagram of the level we needed to understand the layout of the level and how the pieces were connected. We got this information either by playing the game or watching a video of someone else playing it. To use a video the levels in the game needed to be fairly straightforward. Playing the game ourselves allowed us to investigate parts of the level design that would have been unclear from watching someone play it.

While playing or watching the game we would draw a simple representation of the level by hand. If a map of the level was already available, such as the one in Figure 4.3 (pg. 43), then we would print and annotate the map instead.

We could build up a general understanding of a level the first time that we went through it. We could draw a rough map while going through the level. The map would indicate rooms, locks, and keys.

We want the level to be split into traversals, which are areas that the player can move about freely using all of their current tools. The full definition can be found in Section 2.3.2 (pg. 15). We would do this by exploring an area as far as possible without opening any locks. By doing this, we determine the areas that makes up a traversal. All of the rooms that we can move between are considered to be part of the same traversal. We mark all of the locks and keys that we find as attached to the traversal that we are currently exploring.

After we have fully explored a traversal we will open one lock to enter the next area. We mark the key that we used as attached to the lock that we opened. We explore this area to map its locks and keys. We repeat this process until we reach the end of the level.

We would sometimes go through the level a few times to refine the map. These refinements would generally be simplifications and the removal of structures that are not of interest to us in our research. This is discussed more in the next step of our method.

Decide which Items are Significant

For this research, we are only interested in lock-and-key combinations that are significant to how the player navigates the level to reach the final goal. This means that we must exclude bonus areas, moment challenges, and short-cuts from our maps. These elements are defined in Section 2.3.2 (pg. 17).

After completing a level and the rough map we would review if the locks and keys that we hand marked were significant. For a lock-and-key combination to be significant there must be some challenge between the lock and key. This challenge could be combat, navigating hazards, or just some exploration. The separation of the small-scale challenge from the larger navigation component is explained by Brown (2017, timestamp 0:16) below.

> "Sure there were enemies to fight and simple block pushing puzzles to solve, but those acted as quick-fire micro-challenges designed to keep you engaged while you solve the more important macro-challenge of actually finding your way to the boss room." - Mark Brown - The Legend of Zelda: Skyward Sword's dungeon design — Boss Keys (Brown, 2017, timestamp 0:16)

If a lock-and-key combination was determined to not be significant, we would remove it from the map. If that resulted in two traversals having no significant locks separating them, then we would merge them into one traversal.

Another refinement that we would make is merging redundant locks. Two locks are considered redundant if the one leads to the other without any challenge in between. In this case we would merge the locks on our map into one lock that requires all the keys of the original lock. This maintains the navigational complexity in a more compact form.

After reducing our map to the significant locks, keys, and traversals we would convert the rough map into a lock-and-key graph as discussed below.

Convert the Map to a Lock-and-Key Diagram

Once we had a good understanding of the structure of the level with our rough map we could create a lock-and-key graph to represent the level. We start with placing the *begin* node and connect it to the first traversal, a circle with the number one.

For every traversal we do the following. Add a node for every key found in that traversal and place an edge leading from the traversal to the key. For every lock, create a node and edge leading from the traversal to the lock. If the player is able to move back into the traversal after passing through the lock, then have the edge go in both directions.

Once you have completed this for one traversal, pick another traversal that is found on the other side of a lock and add it as a node with the next traversal number, with an edge connecting it to the lock. Repeat the above until all locks and keys are placed. Place an edge leading from every key to the locks that they open.

At the end of this process we will have a puzzle graph like the one seen in Figure 4.2. This diagram was created in the graph drawing software *yEd* graph editor software (yWorks, 2000), which includes functionality to help layout graphs.

The lock-and-key graph representation of the level can be used to understand the structure of the navigational puzzle. In Section 4.3 we will discuss how we used these graphs to find underlying rules for designing new puzzles. In the following section we give a practical example of how we created the lock-and-key diagram for a specific level.

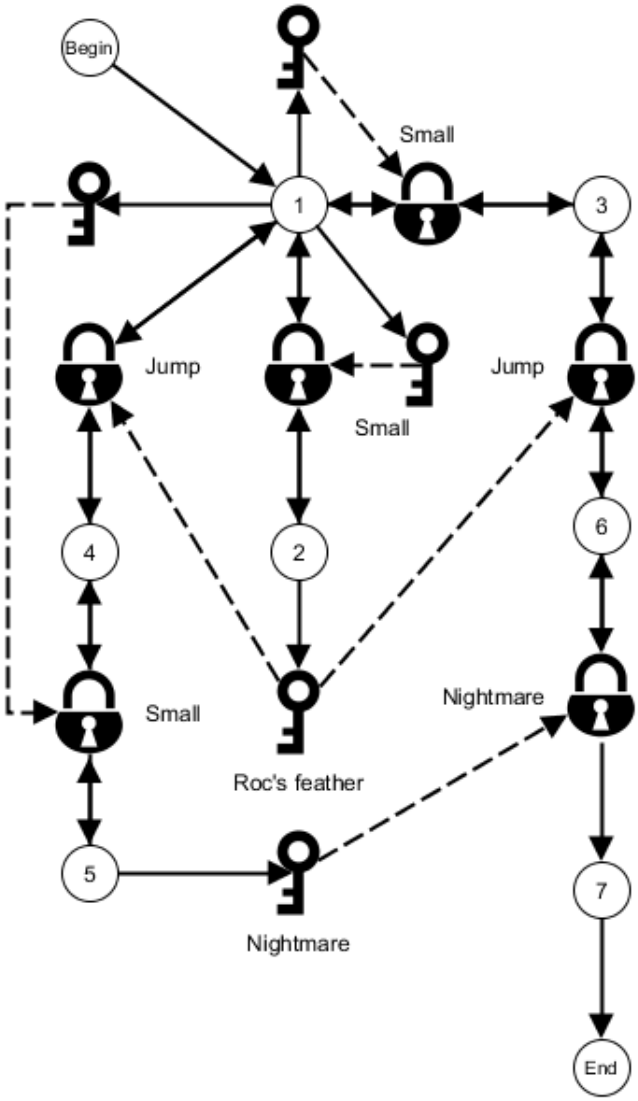


Figure 4.2: Tail Cave puzzle graph

4.2.3 Practical Example

We shall demonstrate the process we used to create the lock-and-key diagrams by going through one of the levels that we mapped. We will look at *Tail Cave* from *The Legend of Zelda: Link's Awakening* (Nintendo, 1998a). The game's overview is in Section 2.2.2. The derivation of the graph for *Tail Cave* is shown in Section 5.4.2.

We will represent our location with an image *Link*, the character controlled by the player. *Link*, as he appears in *The Legend of Zelda: Link's Awakening*, is shown in Figure 4.4a.

In preparation we would get a map of the level, if one is available. For *Tail Cave* a full map is shown in Figure 4.2. We have added a grid so that we can reference rooms easily in this section.

Tail Cave is the first dungeon in *The Legend of Zelda: Link's Awakening*. The only tools available to the player at the start of this level are the sword, shield, and magic powder. During the level we will acquire more tools important to our progress. These tools will be the *Roc's feather*, *small keys*, and the nightmare key.

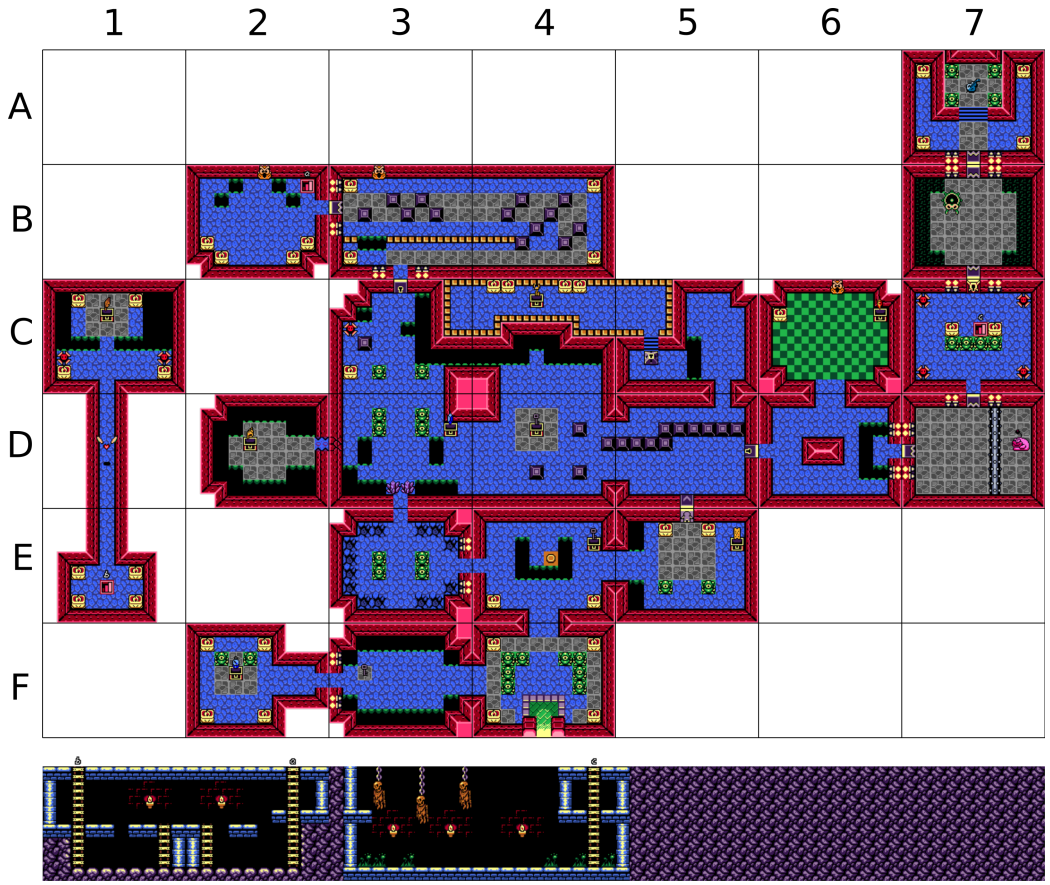
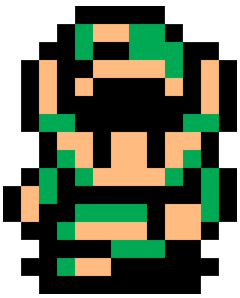
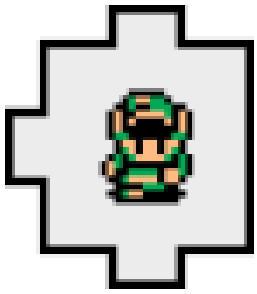


Figure 4.3: Tail Cave map



(a) Link character (Zelda Dungeon Team, 2012)

(b) F4



(c) Starting map

Figure 4.4: Tail Cave screenshots

We need to construct a map that splits the level into traversals. A traversal is the full area accessible with the tools that the player has available to them, as discussed in Section 2.3.2 (pg. 15). We represent our current location with an image of the main character *Link* as shown in Figure 4.4a.

We enter the dungeon at the bottom of room F4, shown in Figure 4.4b. The room is filled with statues and two exits. There are no obstacles or enemies in this room. Our starting map will just be a single room for traversal 1. This is shown in Figure 4.4c.

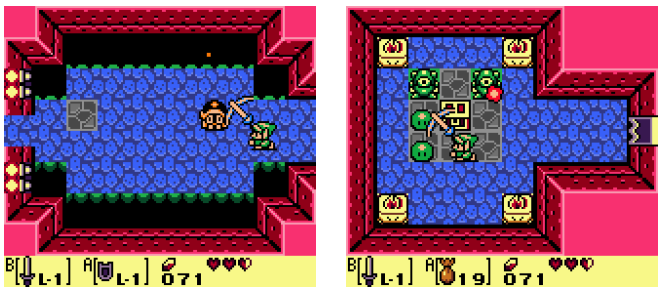
We choose to go left to room F3 first. The map in Figure 4.3 shows that there are only two rooms to the left, so it will be faster to explore that direction first.

Room F3, shown in Figure 4.5a, has two enemies which we defeat by knocking them off the bridge. Once we have done that, a *small key* is dropped. We mark this key on the map with a key symbol and an *S*, to indicate the type of key.

Room F2, shown in Figure 4.5b, has some enemies and contains the *compass*. The *compass* helps with finding items, but is not critical to completing the dungeon, so we do not need to record it on the map.

So far there has been nothing that restricts the passage between these three rooms, so they will all be part of traversal 1. We mark this on the map. The map so far is shown in Figure 4.6.

We return to room F4, where we first entered, and take the upward path. This leads to room E4, see Figure 4.7a. This room contains some enemies and a button. The button makes a chest containing a *small key* appear. We mark this key on the map. There is no obstacle between the button and chest, so



(a) F3 (b) F2

Figure 4.5: Tail Cave screenshots

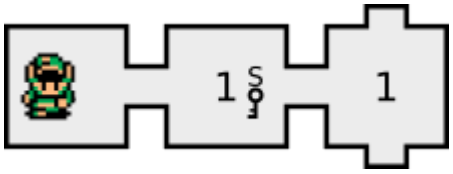


Figure 4.6: Map after exploring the first three rooms

we do not include this as a lock-and-key combination. This is discussed in Section 2.3.2 (pg. 15).

From room E4 we go right to room E5, seen in Figure 4.7b. There is a chest in this room that contains the *map*. This gives an outline of the rooms in the dungeon and indicates where the chests and the *boss* are. The *boss* is the final and toughest enemy of the level, their location would indicate the final room of the level. This item is useful, but not critical, like the *compass* discussed above. We do not mark it on the map.

We next go up to room D5, seen in Figure 4.7c. We can see a locked door on the right, which we mark on the map. One of the small keys will open it. Going through this door would lead into another traversal. We will only do this after we have fully explored the current traversal.

Exploring the rest of traversal 1 reveals one more *small key* in room D4 (Figure 4.8a). There is another lock at the top of room C3 (Figure 4.8b). Finally there are a lock and chasm in room C5 (Figure 4.8c). It is important to note that rooms C4 and C5 only have half of their space in traversal 1. We have marked this with dotted lines. The chasm in C5 acts as a lock as will be seen later. We cannot make any more progress without opening a lock. We mark this on the map. The map so far is shown in Figure 4.9. Rooms

(a) E4 (b) E5 (c) D5

Figure 4.7: Tail Cave screenshots

(a) D4 (b) C3 (c) C5

Figure 4.8: Tail Cave screenshots

C3, C4, and C5 have been subdivided with dotted lines. This is because part of that room is inaccessible to us currently, due to the chasms and walls in the way. We don't know which traversal they are part of yet, so we leave a question mark in their space for now.

We decide to go through the door at the top of room C3, which leads to room B3. This is marked as being part of traversal 2. All rooms that we enter will be labelled as traversal 2 until we pass through another lock.

We proceed into B3. There is a barrier that we cannot cross to reach the upper half of the room. To go around this obstacle we move into room B4 and then go up and back into B3. We can then move into room B2. There is a set of stairs in B2 that leads into a tunnel that emerges in room E1. This is shown in Figure 4.10b and the bottom left of Figure 4.3. After taking the tunnel to room E1, we move upwards through room D1 to room C1 (Figure 4.10c) where we find a chest containing the *Roc's feather*. We mark the *Roc's feather* on the map as a key with an *F*.

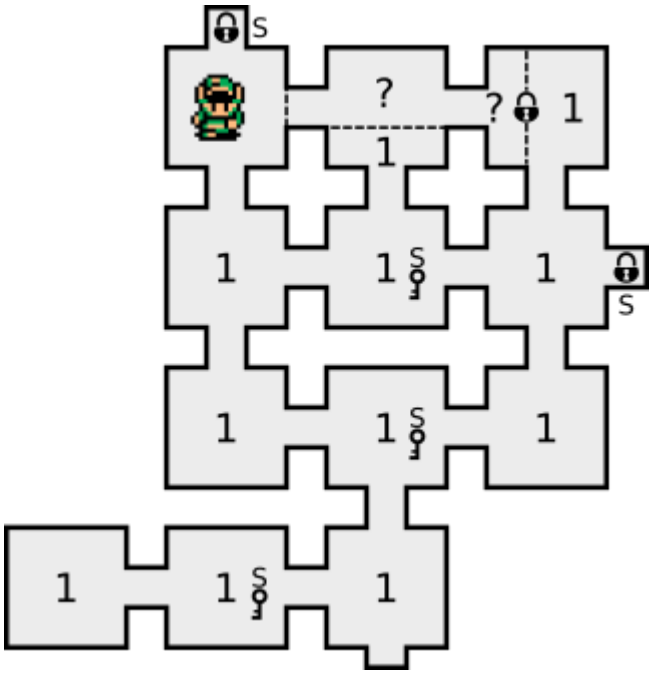
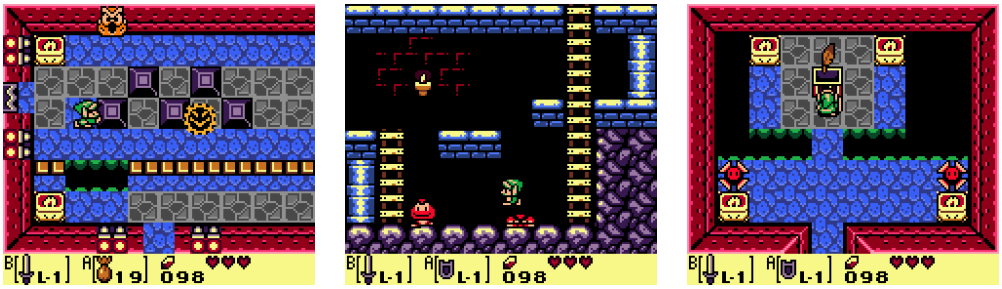


Figure 4.9: Map after fully exploring traversal 1



(a) B3 (b) Tunnel (c) C1

Figure 4.10: Tail Cave screenshots

The *Roc's feather* is the *key item* for this dungeon. It gives us the power to jump. We can consider the chasm in room C5 to be a lock and the *Roc's feather* to be the key. We will mark this as a key on our map with an *F* on the key as well as the lock in room C5.

We can proceed no further in traversal 2 so we will need to enter a new area. We have two options: jump across the chasm in room C5 (Figure 4.8c) or go through the door in room D5 (Figure 4.7c). We decided to use the door since we had access to the *small key* before the *Roc's feather*. The map so far is shown in Figure 4.11. We walk all the way back to room C5 by retracing our steps and use a *small key* to unlock the door to room D5.

The door leads into room D6 (Figure 4.13a). From here we will mark rooms as being in traversal 3. There is a chasm to the right. Since we could have entered this area before getting the *Roc's feather* we mark this as a lock with an *F*. If we go up to room C6 we will get another optional item, the *stone beak*, which provides clues. This is non-critical so we do not mark it on the map. These two rooms are all that make up traversal 3.

We decide to go back from room C6 to the chasm in room C5 (Figure 4.8c). Jumping across it using the *Roc's feather* puts us in a new area that we will label traversal 4. We immediately encounter a lock that requires one of the *small keys* that we picked up earlier. Unlocking this lets us move left into room C4.

We are in the upper half of room C4 (Figure 4.13b). We will mark this as traversal 5. We find a chest that contains the *Nightmare key* which unlocks the door to the final and most difficult enemy of the dungeon. We mark this with a key symbol annotated with an *N*. There is a small part of this traversal in room C3, but is does not affect the layout of the diagram. After this the

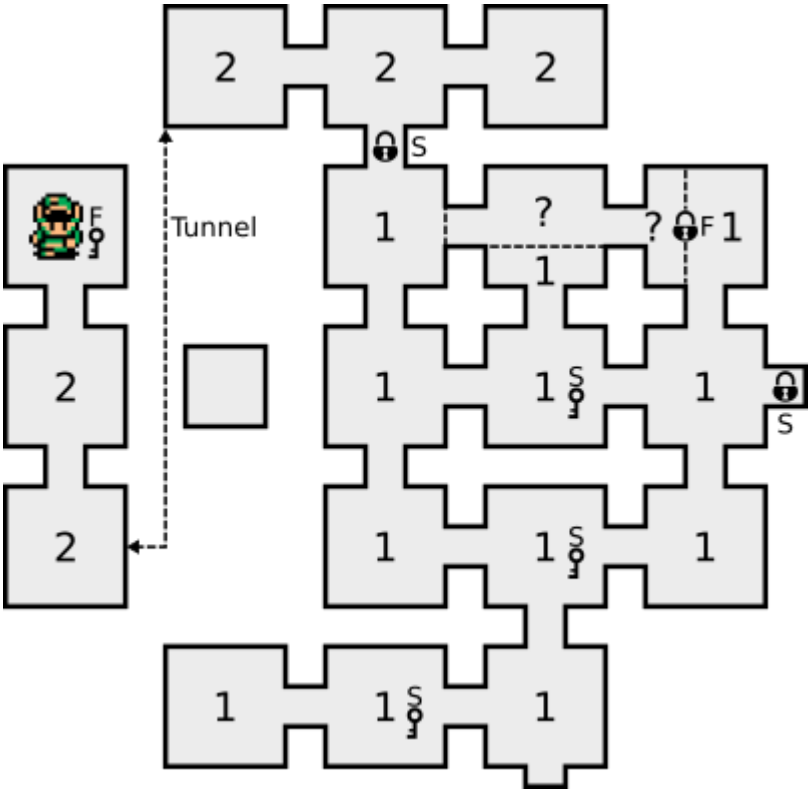


Figure 4.11: Map after fully exploring traversal 2

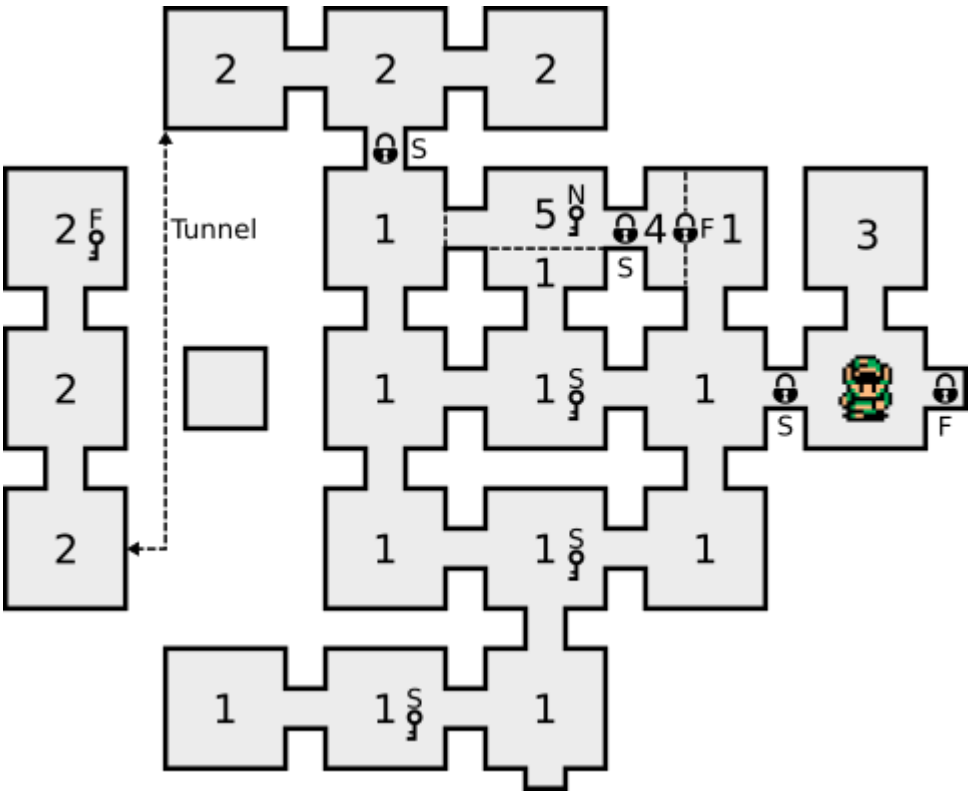


Figure 4.12: Map after fully exploring traversal 5

only option is to jump across the chasm in room D6. The map so far is shown in Figure 4.12.

We enter room D7 (Figure 4.13c). This is a new area so we will label it as traversal 6. In this room we fight a *mini-boss*. This is a challenging battle where we have to use the *Roc's feather* to win. The *mini-boss* pushes a large rolling beam at the player which we need to jump over.

Defeating the *mini-boss* opens up a portal that lets us teleport between rooms D7 and F4 (the start). This is a short-cut, so we do not mark this on the map due to it not affecting the puzzle structure. This is discussed in Section 2.3.2 (pg. 17).

We can move up into room C7 from D7 (Figure 4.14a). The stairs in the centre are not important to the map, but they will be explained later. The door at the top can be opened using the *Nightmare key*. This will take us into the final traversal, which we will label traversal 7.

(a) D6 (b) C4 (c) D7

Figure 4.13: Tail Cave screenshots

(a) C7 (b) B7 (c) A7

Figure 4.14: Tail Cave screenshots

In room B7 (Figure 4.14b) we face the *boss* of the dungeon. If the *boss* knocks the player off the platform they will fall into a tunnel that emerges at the stairs in room C7.

After a tough fight we can move up into the room A7. In this room we find the *Moon Cello*. Picking it up marks the end of the dungeon. The final map is shown in Figure 4.15.

Our annotated map is complete. The next step is to create a graph that represents the structure of the dungeon. We create a node for each traversal, lock, and key in the dungeon. For each traversal we draw an edge to and from each lock that we found in that traversal. We put edges both ways because the player is able to leave the area that they have entered after passing through the lock.

We put an edge to each key from the traversal that it is found in. For each key we draw an edge to the lock (or locks) that were unlocked by that key. We then lay out the nodes so that the map is easy to read. The final

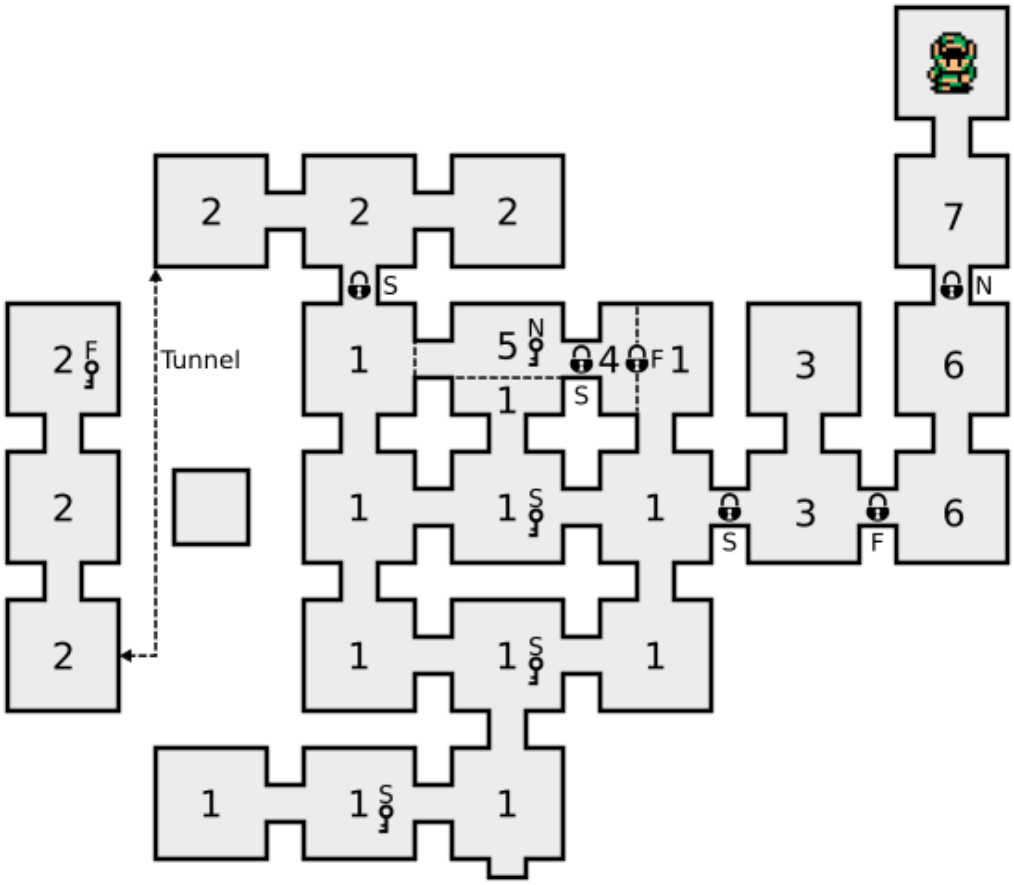


Figure 4.15: Final map

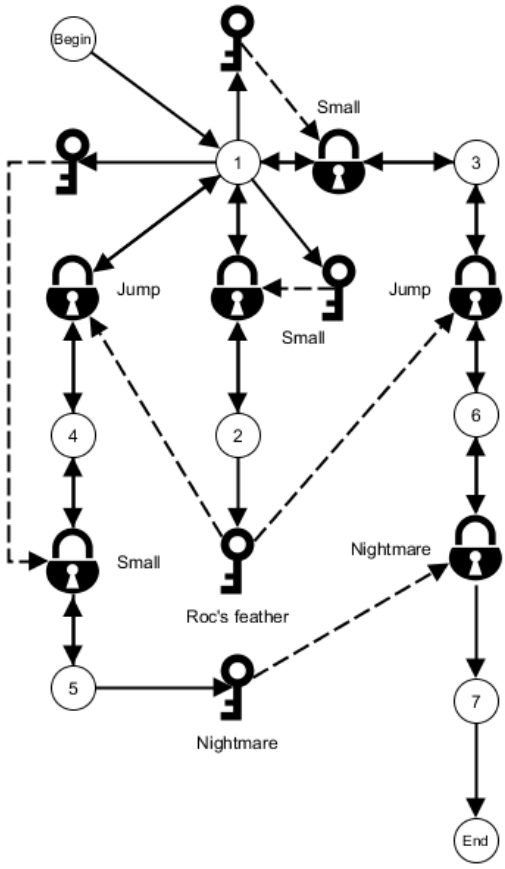


Figure 4.16: Tail Cave puzzle graph

result of this process is shown in Figure 4.16. We used the *yEd* graph editor software (yWorks, 2000) and its automatic layout algorithms, with some manual tweaks here and there.

This procedure was followed for all the levels that we have documented. Once we had the puzzle graphs for a game we could then design a grammar to describe those puzzles. Our method for doing so is discussed in the next section.

4.3 Grammar Design

We designed a grammar for each game. All of the lock-and-key graphs for a game would be contained in the language of that game's grammar. We have chosen to use B-dNCE graph grammars, as described in Section 2.7.2 (pg. 24).

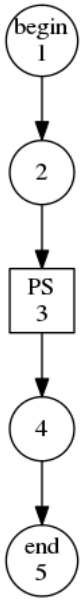
The logic behind the construction of the puzzles is that the final goal is established first. Then an obstacle is placed between it and the start. We consider reaching this final obstacle and its solution to be the new goal. As mentioned above, we will place more obstacles between that start and this goal until we have created a full puzzle.

4.3.1 Edge Directions

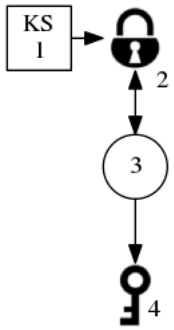
The direction of an edge leading into a non-terminal node indicates the direction of the player's objective. This can clearly be seen in Figure 5.1, where the edges point from the start node to the end. This also applies to the local objective, such as reaching the key as seen in Figure 5.6.

4.3.2 Maintaining Context Freeness

A graph grammar can be considered context free if no non-terminal nodes are adjacent to each other. A non-terminal node appears as a mother subgraph in at least one of the rules in the graph grammar. This condition can be maintained by having well designed productions.



(a) Daughter subgraph of grammar rule S1. Full rule shown in Figure 5.1 (pg. 58)



(b) Daughter subgraph of grammar rule KS2. Full rule shown in Figure 5.6 (pg. 60)

Figure 4.17: Edge directions

Chapter 5

Implementation

The results come in a few forms. For each game that we studied we will first show the grammar that we constructed to describe its puzzles. We then show the puzzle graphs for each level that we documented. The full derivations for each level are found in Appendix B.

We wrote some software to generate puzzle graphs using our grammars. This is discussed in Section 5.1.

The *Forest Temple* dungeon illustrates a limitation of the technique that we are using. This is discussed in Section 5.5.2.

For each of our games of interest we present the grammar that we derived from the game. All of these are B-dNCE graph grammars, as described in Section 2.7.2 (pg. 24). We show how a number of puzzles from that game can be productions of the grammar.

We also show examples of novel puzzles produced by the same grammars.

Additionally, in Section 5.2.3 we present a proof that a well designed grammar is guaranteed to always produce a solvable puzzle.

In Section 5.2 we present the grammar for *Prince of Persia: The Sands of Time*. In Section 5.3 we present the grammar for *Tomb Raider*. In Section 5.4 we present the grammar for *The Legend of Zelda: Link's Awakening*. In Section 5.5 we present the grammar for *The Legend of Zelda: Ocarina of Time*.

5.1 Application

We created a program that would use our grammar rules to create puzzle graphs. Additionally we needed a method to render the hundreds of graph figures that appear in this work. These are discussed below.

We render our graph diagrams using the Dot program from the Graphviz package (Graphviz Team, 2016). This takes in text files with the graphs described using the DOT language (Graphviz Team, 2017).

We stored our grammar rules in text files using JSON (JavaScript Object Notation). An example JSON file is given in Appendix A.

The application consists of some Python scripts that extend the Pygraph (Ciskey, 2016) library. This applies graph grammar rules until the graph reaches a terminal state. The resulting graph is transcribed into DOT language.

5.2 Prince of Persia: The Sands of Time

In this section we will present the grammar that we developed for *Prince of Persia: The Sands of Time* and demonstrate how it achieves our goals. An overview of this game is given in Section 2.2.1 (pg. 9).

First we will present the production rules in Section 5.2.1. We then show the levels that we observed in the game. The full derivations of these levels using our grammar are show in Appendix B.1 (pg. 58). In Section 5.2.3 we will present a proof that the grammar will always produce a solvable puzzle, meaning that there is a way to get from the beginning to the end of the puzzle. Finally we will show examples of puzzles that have been randomly generated using the grammar in Section 6.1.

This grammar is simpler than the other three that we produced. This is due to the straightforward obstacle layouts. The actual environments were fairly complex to navigate.

5.2.1 Rules

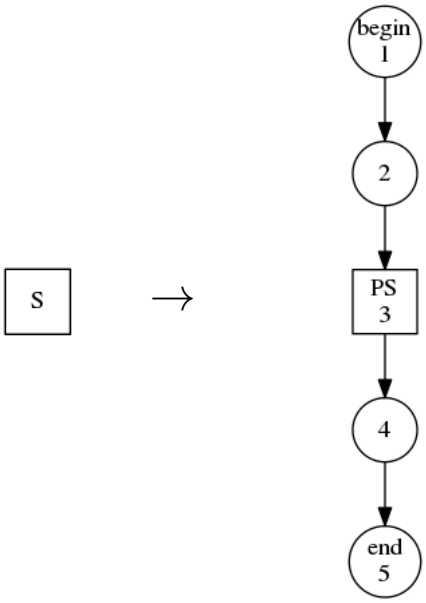


Figure 5.1: Prince of Persia rule: S1 - Start

S1 - Start: See Figure 5.1. This is the start production. This creates the start and end nodes as well as a puzzle section between them. Note that the edges indicate the direction that the player must go. This concept was discussed in Section 4.3.1.

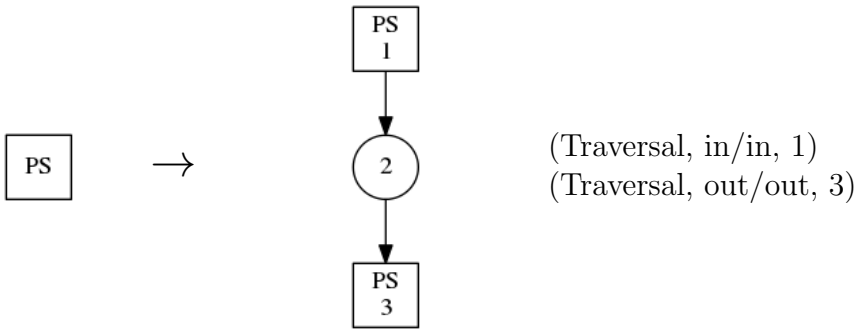


Figure 5.2: Prince of Persia rule: PS1 - Puzzle extension

PS1 - Puzzle extension: See Figure 5.2. This production extends the puzzle. One puzzle section is replaced by two successive ones. Applying this production repeatedly will make the puzzle longer.

We have ensured that the puzzle section non-terminal node will always found with a traversal leading in and a traversal leading out. It is guaranteed that a player who reaches the traversal leading in will be able to proceed to the traversal afterwards.

(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, out/out,2)

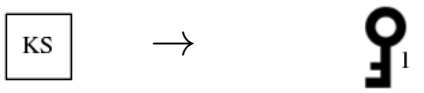
Figure 5.3: Prince of Persia rule: PS2 - One-way lock

PS2 - One-way lock: See Figure 5.3. This creates a one-way lock. Once the player passes through the lock they cannot return to the area that they left. These are used to stop players from getting lost and going the wrong way. These are often implemented as gates that open on a timer. The player can use the key in a form of a switch to open the gate for a few seconds. They must run through the gate quickly, after which it will shut behind them, with no way to open it from the area that they are now in.

(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, in/out,2)
(Traversal, out/in,2)
(Traversal, out/out,2)

Figure 5.4: Prince of Persia rule: PS3 - Two-way lock

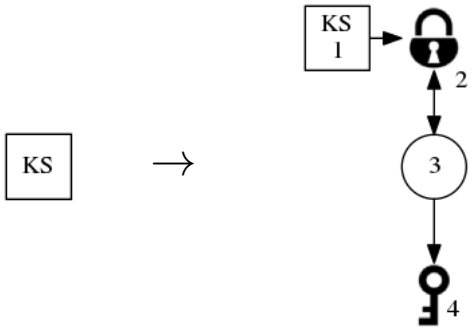
PS3 - Two-way lock: See Figure 5.4. This creates a two-way lock. It functions exactly the same as production PS2 above, but it does allow backtracking. That means that after passing through the lock into a new area, they will be able to move back into the previous area. Unlike the gates in rule PS2, these do not close after the player passes through. This difference is represented by the embedding rule adding in and out edges to traversals leading in and out of the puzzle section node.



(Traversal, in/in, 1)
(Lock, out/out, 1)

Figure 5.5: Prince of Persia rule: KS1 - Simple key

KS1 - Simple key: See Figure 5.5. This places a key that will unlock the adjacent lock. This is the terminal for all of the key search (KS) rules.



(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, in/out, 2)
(Lock, out/out, 4)

Figure 5.6: Prince of Persia rule: KS2 - Hidden key

KS2 - Hidden key: See Figure 5.6. This rule creates a stacked lock. That means that it places the key for the adjacent lock, but that key is behind another lock. The key for the new lock will be produced by the key search.

(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Lock, out/out, 1)
(Lock, out/out, 2)

Figure 5.7: Prince of Persia rule: KS3 - Multiple keys

KS3 - Multiple keys: See Figure 5.7. This increases the number of keys that the player has to find to open the lock connected to the original key search. Each key could be concealed using the other key search productions.

5.2.2 Levels

We analysed five levels of *Prince of Persia: The Sands of Time*. Not all levels in the game contain lock-and-key structures. Some are entirely combat and movement challenges. We show our lock-and-key representations of some levels below.

I'll Meet You at the Baths

The prince has been separated from his companion, Farah. She tells him to meet her at the baths of the palace. The prince must first open one of the palace gates by operating sets of smaller gates in the courtyard to get access to the main gate's mechanism. The graph of the level is shown if figure 5.8. The full derivation of this graph is shown in Appendix B.1.1 (pg. 142).

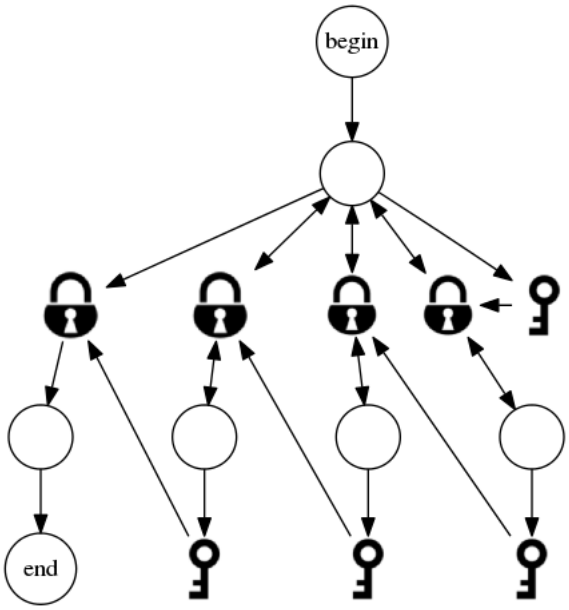


Figure 5.8: *I'll Meet You at the Baths* final graph

Hall of Learning Courtyards

Farah and the prince are trying to reach the Hourglass of Time in the Tower of Dawn and need to find a way up onto the palace ramparts. The graph of the level is shown if figure 5.9. The full derivation of this graph is shown in Appendix B.1.2 (pg. 145).

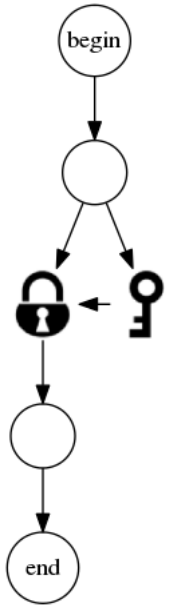


Figure 5.9: *Hall of Learning Courtyards* final graph

Out of the Well

The prince has just found his way out of an underground reservoir and makes his way towards what used to be the sultan's harem. He hopes to find her there. The graph of the level is shown if figure 5.10. The full derivation of this graph is shown in Appendix B.1.3 (pg. 146).

An Underground Reservoir & Atop a Bird Cage

The two levels *An Underground Reservoir* and *Atop a Bird Cage* have identical puzzle graphs. The environments are very different. In *An Underground Reservoir* the prince needs to find his way out of the reservoir feeding many of

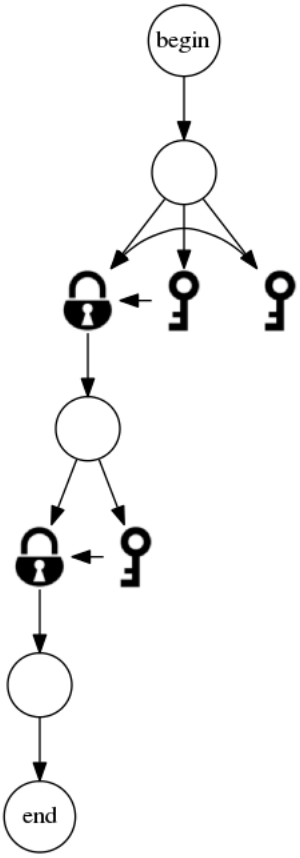


Figure 5.10: *Out of the Well* final graph

the palace wells. In *Atop a Bird Cage* the prince navigates the menagerie of the sultan. This requires ascending a giant bird cage filled with birds that were corrupted by the sands of time. The graph of the level is shown if figure 5.11. The full derivation of this graph is shown in Appendix B.1.4 (pg. 149).

5.2.3 Proof of Solvability

We wish to demonstrate that a well designed grammar can guarantee that all generated puzzles are solvable. This means that there exists a way to get from the start of the puzzle to the end. The grammar for *Prince of Persia: the Sands of Time* has this property. We will argue this for each production individually.

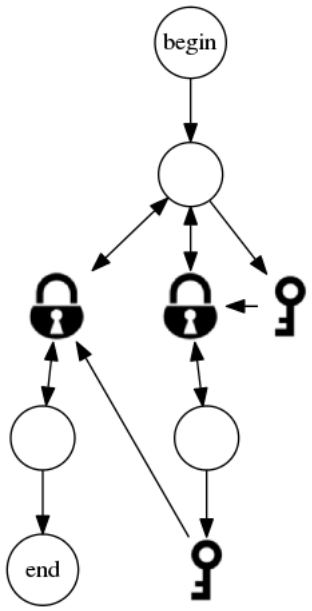


Figure 5.11: *An Underground Reservoir* final graph

All puzzles start with the production S1 being applied. See Figure 5.1. As we can see, the path from the start node goes directly to a single non-terminal node, labelled PS for puzzle section. The path to the end comes from this same non-terminal node.

For the puzzle to be solved the player must be able to move from the *begin* node to the *end* node. Note that the edges pointing in and out of the puzzle section indicate the direction that the player follows to reach the end of the puzzle. We will demonstrate that the player will always be able to reach the end node after any valid combination of productions has been applied to the graph.

Definition 5.2.1. If we can guarantee that any player reaching the node before the puzzle section can reach the node after then we can say with certainty that all puzzles will be solvable.

There are three possible productions that can be applied to a puzzle section. We will demonstrate that each one will allow the player to pass from the traversal before the puzzle section to the traversal after it.

PS1 (Figure 5.2) extends the puzzle by making two puzzle sections come one after another. The above argument holds for this situation as well. That

is: if we can guarantee that any player who reaches the traversal before the puzzle section will be able to reach the traversal after it then we can guarantee that the puzzle is solvable.

PS2 (Figure 5.3) and PS3 (Figure 5.4) can be argued together since being able to backtrack does not affect whether a puzzle section is solvable in this grammar. Backtracking means that after passing through a lock they can return to the original traversal.

The traversals before and after the puzzle sections are connected to the lock. That means that the player must pass through the lock to solve this puzzle section. For that to happen we require that the non-terminal node for a key search (KS) always produces a reachable key. We must argue that this happens for all three productions that apply to key searches.

Definition 5.2.2. We say that a key is reachable in a key search if the player can get to the key from the traversal that leads into the key search.

KS1 (Figure 5.5) replaces the key search with a key. The traversal leading into the key search connects directly to the key. This makes the key reachable. Thus this production rule always produces a reachable key.

KS2 (Figure 5.6) creates what we call a stacked lock. The key (node 4) is the one that we need to show is reachable. The requirements for this are the same as those discussed for PS2. We need the key search at node 1 to always produce a reachable key. Any of the three key search productions could be applied.

Applying KS1 would immediately satisfy the requirements for making the key reachable. Applying KS2 would result in the same circumstance as discussed in this paragraph. It can be applied an arbitrary number of times, but the final production used will have to be KS1. If we apply KS3 it creates two key searches, both of which need to produce reachable keys. From the above we know that applying KS1 or KS2 will do this. Applying KS3 will just produce even more key searches. This means that KS3 will also always produce reachable keys.

We have shown that a key search always produces a reachable key. This means that a puzzle section will always be solvable. Finally this implies that all puzzles produced by the grammar for *Prince of Persia: the Sands of Time* will be solvable.

This is useful to have demonstrated because it means that we do not waste computational resources by generating invalid puzzles as was seen in some of the related work (Coxen and Baylis, 2012b).

5.3 Tomb Raider

In this section we will present the grammar that we developed for *Tomb Raider*. Section 5.3.1 presents the grammar rules. Section 5.3.2 displays the levels that we analysed. An overview of *Tomb Raider* is given in Section 2.2.4 (pg. 14).

5.3.1 Rules

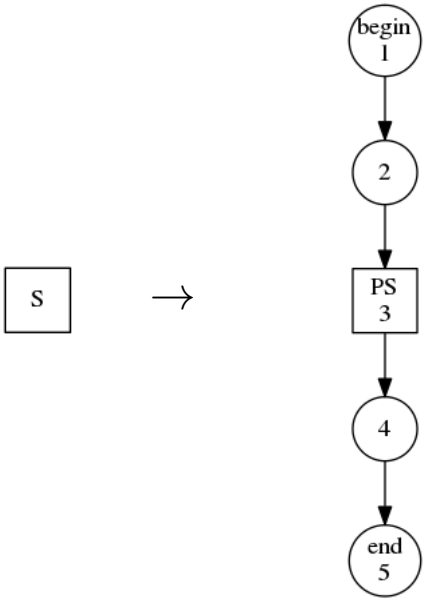


Figure 5.12: Tomb Raider rule: S1 - Start

S1 - Start: See Figure 5.12. This is the start production. This creates the start and end nodes as well as a puzzle section between them. Note that the edges indicate the direction that the player must go. This concept was discussed in Section 4.3.1.

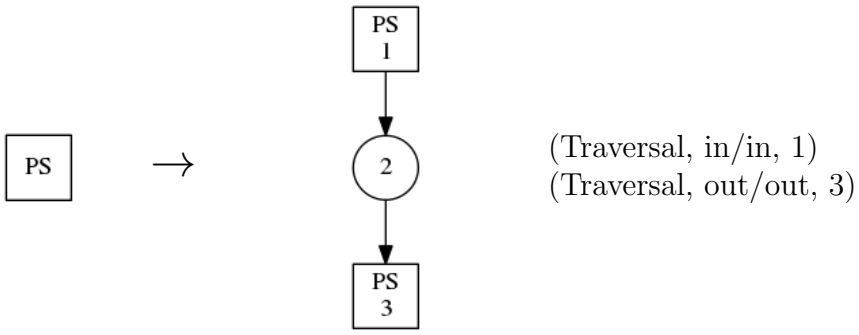


Figure 5.13: Tomb Raider rule: PS1 - Puzzle extension

PS1 - Puzzle extension: See Figure 5.13. This production extends the puzzle. One puzzle section is replaced by two successive ones. Calling this repeatedly will make the puzzle longer.

We have ensured that the puzzle section non-terminal is always found with a traversal leading in and a traversal leading out. It is guaranteed that a player who reaches the traversal leading in will be able to proceed to the traversal afterwards.

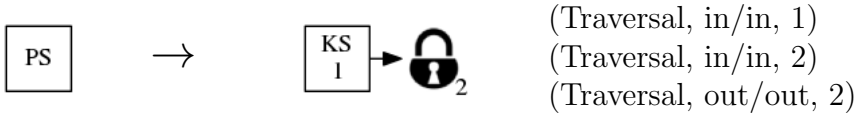


Figure 5.14: Tomb Raider rule: PS2 - One-way lock

PS2 - One-way lock: See Figure 5.14. This creates a one-way lock. Once the player passes through the lock they cannot return to the area that they left. These are used to stop players from getting lost and going the wrong way. These are often implemented as gates that open on a timer. The player can use the key in a form of a switch to open the gate for a few seconds. They must run through the gate quickly, after which it will shut behind them, with no way to open it from the area that they are now in.

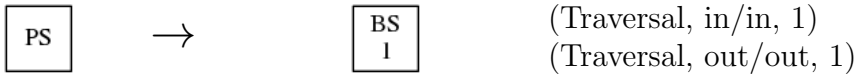


Figure 5.15: Tomb Raider rule: PS3 - Backtrackable section

PS3 - Backtrackable section: See Figure 5.15. This replaces the puzzle section with a backtrackable section. We do this if we want it to be possible for the player to return to a part of the level after passing through a lock.

All productions leading from this guarantee that the player will be able to return to a traversal with an edge leading in after they have travelled to a traversal with an edge leading out.

This provides the same guarantee of a puzzle section, that the player will be able to travel from the prior traversal to the subsequent traversal. In addition, the player will be able to return to the first traversal after passing through the lock.

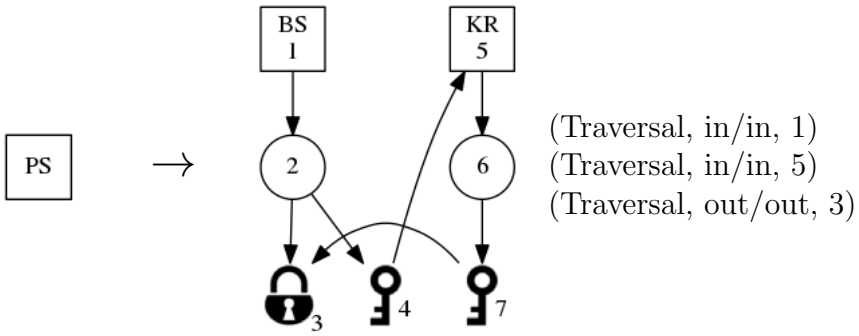


Figure 5.16: Tomb Raider rule: PS4 - Sideroom gate

PS4 - Sideroom gate: See Figure 5.16. This creates a pair of interlinked paths. This feature was observed in the *Tomb Raider* level *Colosseum* which is discussed in Section 5.3.2 (pg. 73). This rule will require the player to backtrack through the level multiple times. We will explain the required navigation below.

To complete this puzzle section, the player will need to pass through the backtrackable section 1 to traversal 2 and get key 4. They can then backtrack to the in-traversal and pass through the key-retrieval 6 and fetch the key 7. They will need to backtrack again to the in-traversal, pass through the backtrackable section 1 again and then through lock 3.

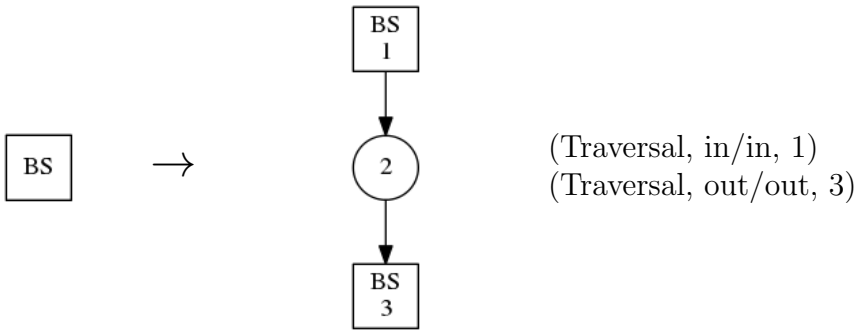


Figure 5.17: Tomb Raider rule: BS1 - Puzzle extension

BS1 - Puzzle extension: See Figure 5.17. This is similar to rule PS1 above, but for backtrackable sections. This will require the player to pass through more challenges before reaching the end of the section.

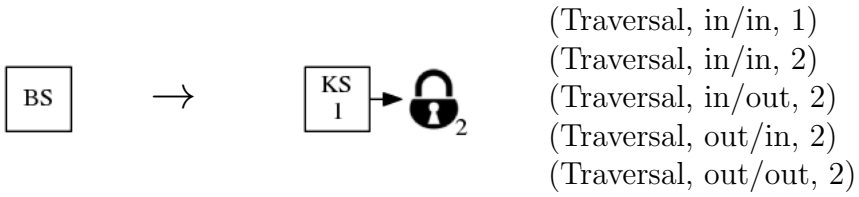


Figure 5.18: Tomb Raider rule: BS2 - Two-way lock

BS2 - Two-way lock: See Figure 5.18. This is similar to rule PS2 above, but for backtrackable sections. The player will be allowed to return to the in-traversal from the out-traversal.

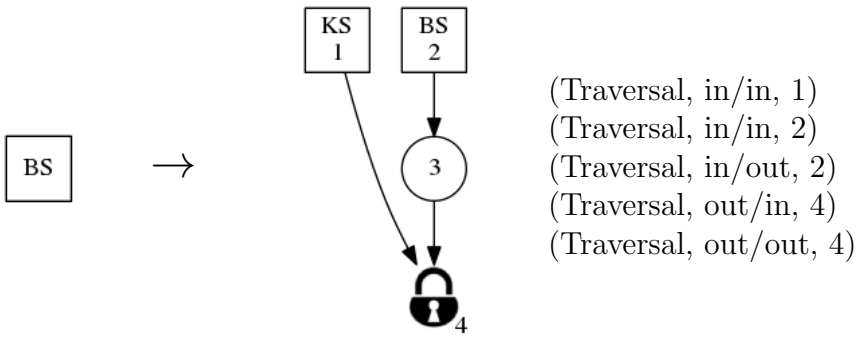


Figure 5.19: Tomb Raider rule: BS3 - Early key

BS3 - Early key: See Figure 5.19. The key for a lock is situated such that a player could find the key before they encounter the lock. Alternatively, they could find the lock and then have to backtrack a number of sections in search of the key.

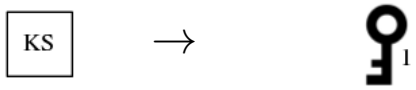


Figure 5.20: Tomb Raider rule: KS1 - Simple key

KS1 - Simple key: See Figure 5.20. This places a key that will unlock the adjacent lock. This is the terminal for all of the key search (KS) rules.

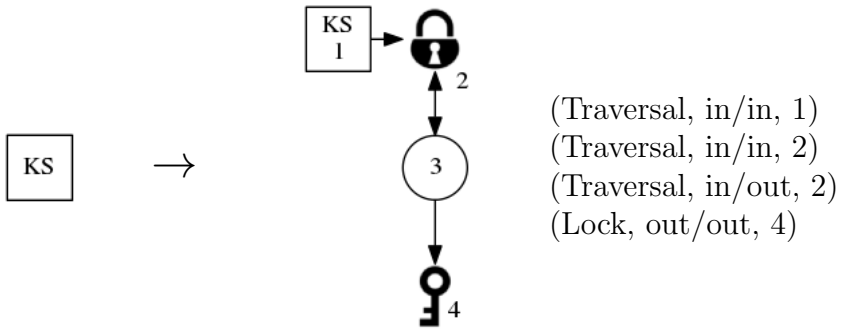


Figure 5.21: Tomb Raider rule: KS2 - Stacking key search

KS2 - Hidden key: See Figure 5.21. This rule creates a stacked lock. That means that it places the key for the adjacent lock, but that key is behind another lock. The key for the new lock will be produced by the key search.

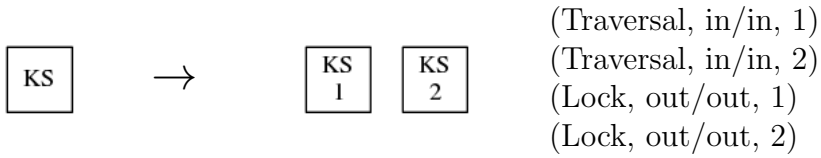
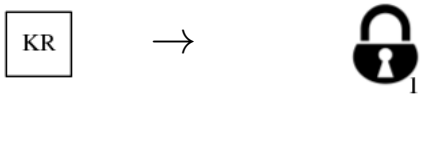


Figure 5.22: Tomb Raider rule: KS3 - Multiple keys

KS3 - Multiple keys: See Figure 5.22. This increases the number of keys that the player has to find to open the lock connected to the original key

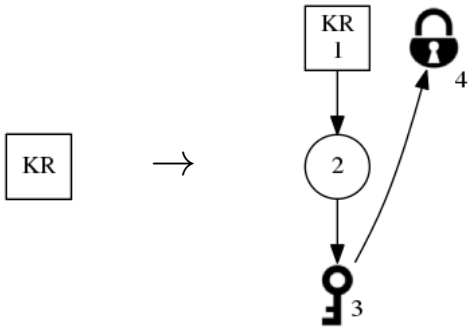
search. Each key could be concealed using the other key search productions.



(Key, in/in, 1)
(Traversal, in/in, 1)
(Traversal, in/out, 1)
(Traversal, out/in, 1)
(Traversal, out/out, 1)

Figure 5.23: Tomb Raider rule: KR1 - Simple key retrieval

KR1 - Simple key retrieval: See Figure 5.23. These are productions where we have already defined the location of the key and will be placing the associated lock. This is an inversion of the key search productions above. This non-terminal node is only created in the rule PS4 above. A key has been placed, and this covers how the lock that it opens is placed.



(Key, in/in, 1)
(Traversal, in/in, 1)
(Traversal, in/in, 4)
(Traversal, in/out, 4)
(Traversal, out/in, 4)
(Traversal, out/out, 4)

Figure 5.24: Tomb Raider rule: KR2 - Stacking key retrieval

KR2 - Stacking key retrieval: See Figure 5.24. Key retrievals are created as part of situations where the end of the branching path is a key. This creates a second layer to the branching path that will make the key retrieval longer.

5.3.2 Levels

We present a few of the levels that we mapped for *Tomb Raider* below. We skipped a few levels in the sequence where the puzzle structures were very similar.

Level 1: Caves

The protagonist, Lara Croft, has found her way to some long forgotten ruins in the Peruvian mountains in search of an ancient artefact. The graph of the level is shown if figure 5.25. The full derivation of this graph is shown in Appendix B.2.1 (pg. 151).

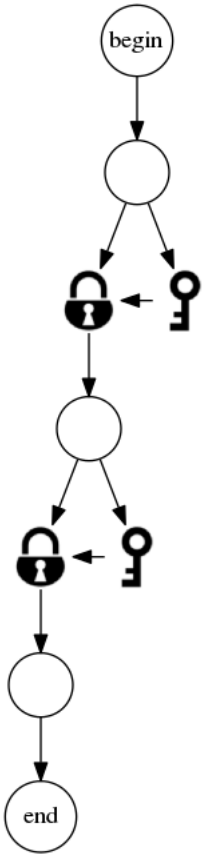


Figure 5.25: *Caves* final graph

Level 2: City of Vilcabamba

Lara Croft explores further into the ancient ruins of the lost civilization. She must overcome traps and dangerous wildlife that have been living there. The graph of the level is shown if figure 5.26. The full derivation of this graph is shown in Appendix B.2.2 (pg. 154).

Level 6: Colosseum

Lara Croft is pursuing the other parts of the artefact and is racing against another treasure hunter, Pierre DuPont, to reach them first. This time they are in some ancient ruins in Greece. The graph of the level is shown if figure 5.27. The full derivation of this graph is shown in Appendix B.2.3 (pg. 159).

Level 10: City of Khamoon

Lara Croft follows a vision given to her that hints at the location of the lost city of Atlantis. The next stop on her journey is in Egypt. Local wildlife and mummies provide obstacles to Lara's progress. The graph of the level is shown if figure 5.28. The full derivation of this graph is shown in Appendix B.2.4 (pg. 168).

Level 14: Atlantis

The rival treasure hunting team has activated the mystical devices of Atlantis unleashing strange monsters. Lara Croft must fight her way through the city to stop them. The graph of the level is shown if figure 5.29. The full derivation of this graph is shown in Appendix B.2.5 (pg. 175).

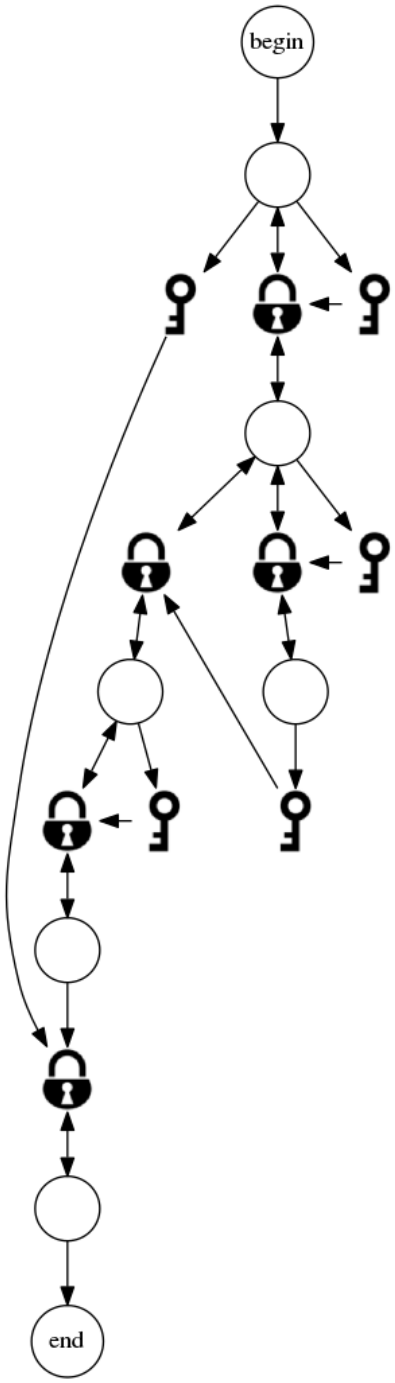


Figure 5.26: *City of Vilcabamba* final graph

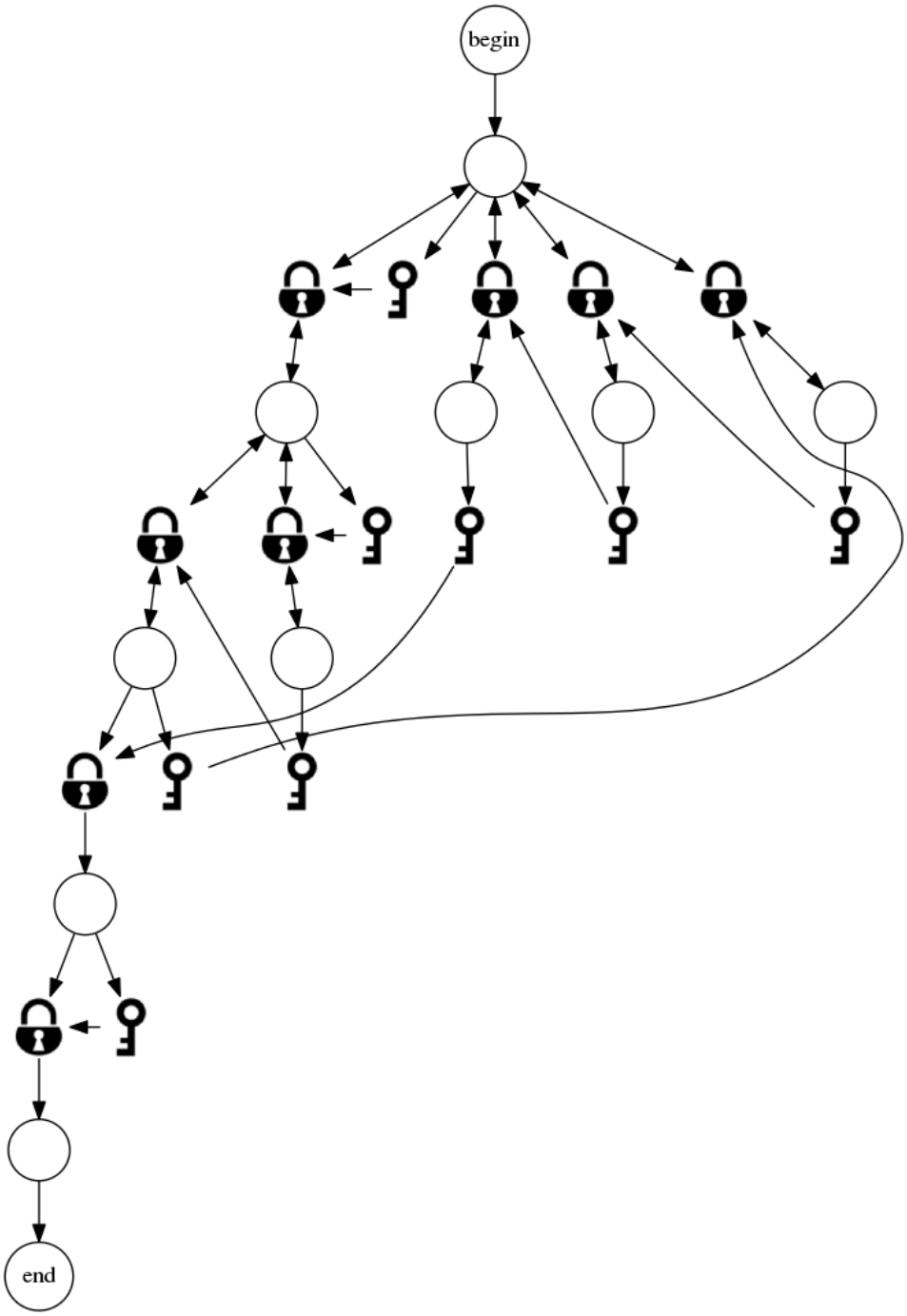


Figure 5.27: *Colosseum* final graph

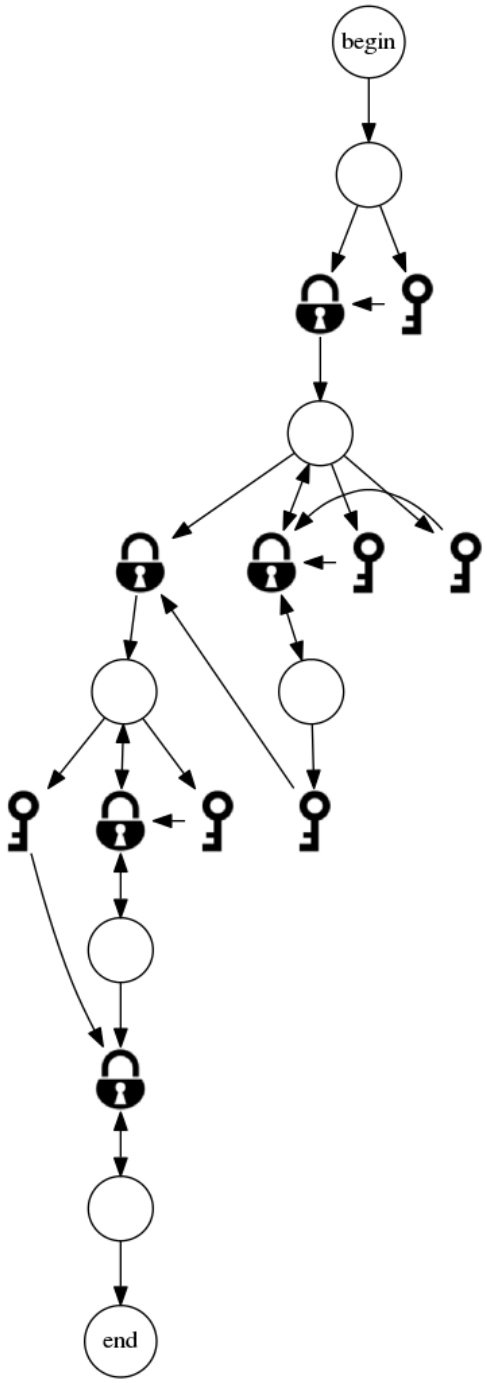


Figure 5.28: *City of Khamoon* final graph

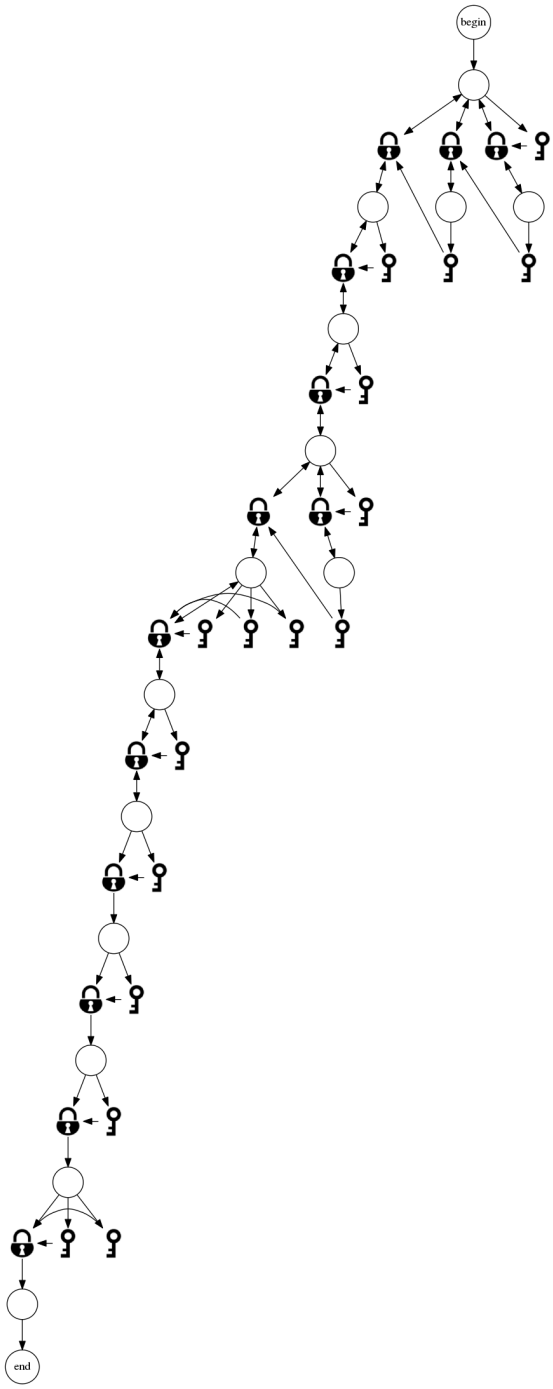


Figure 5.29: *Atlantis* final graph

5.4 Legend of Zelda: Link's Awakening

In this section we will present the grammar that we developed for *Legend of Zelda: Link's Awakening*. Section 5.4.1 presents the grammar rules. Section 5.4.2 displays the levels that we analysed. An overview of *Legend of Zelda: Link's Awakening* is given in Section 2.2.2 (pg. 10).

Legend of Zelda: Link's Awakening has a large world to explore. In this research we focus on the dungeons that are found across the world. The dungeons in *Legend of Zelda: Link's Awakening* tend to follow a similar pattern. The player must find the dungeon-item, which gives them new capabilities. The player can use these capabilities to then find the nightmare key, which will open the door that leads to the boss, and the end of the dungeon (Brown, 2016a).

We first present the levels that we have analysed and their lock-and-key diagrams in Section 5.4.2. We describe the grammar that we constructed for *Legend of Zelda: Link's Awakening* in Section 5.4.1. The full derivations of these levels are presented in Appendix B.3.

5.4.1 Rules

In this section we will present the grammar that we developed for *Legend of Zelda: Link's Awakening*. Section 5.4.1 presents the grammar rules. Section 5.4.2 displays the levels that we analysed. An overview of *Legend of Zelda: Link's Awakening* is given in Section 2.2.2 (pg. 10).

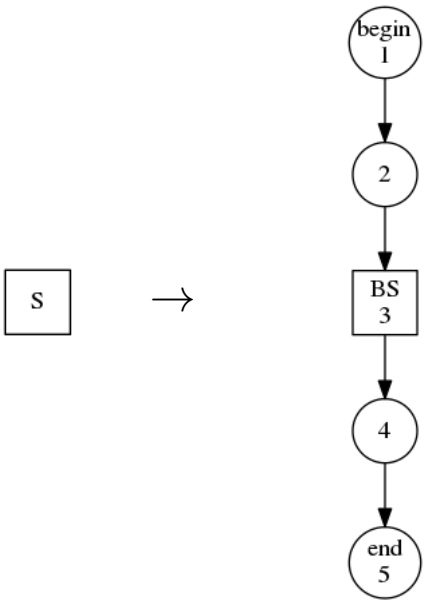


Figure 5.30: Legend of Zelda: Link's Awakening rule: S1 - Start

S1 - Start: See Figure 5.30. This is the start production. This creates the start and end nodes as well as a puzzle section between them. Note that the edges indicate the direction that the player must go. This concept was discussed in Section 4.3.1.

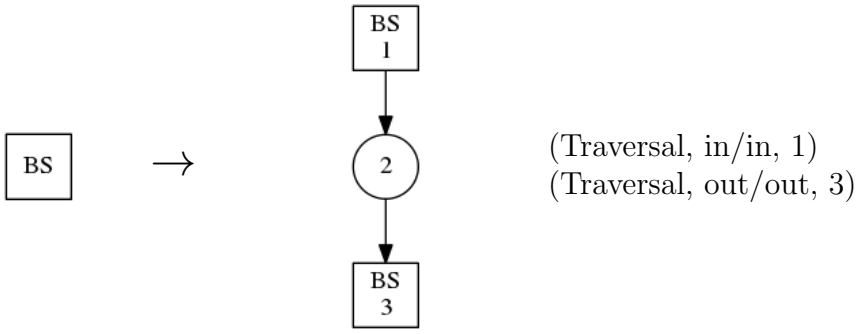
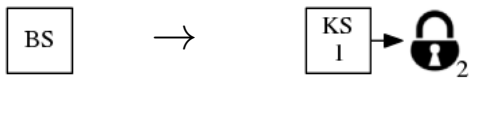


Figure 5.31: Legend of Zelda: Link's Awakening rule: BS1 - Puzzle Extension

BS1 - Puzzle extension: See Figure 5.31. This production extends the puzzle. One puzzle section is replaced by two successive ones. Calling this repeatedly will make the puzzle longer. These are backtrackable sections,

so the player will be allowed to return to the area that the came from after passing through the sections.

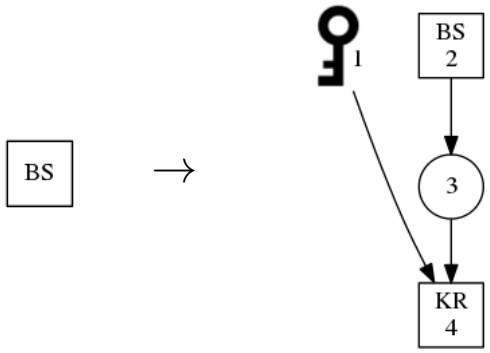
We have ensured that the puzzle section non-terminal is always found with a traversal leading in and a traversal leading out. It is guaranteed that a player who reaches the traversal leading in will be able to proceed to the traversal afterwards.



(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, in/out, 2)
(Traversal, out/in, 2)
(Traversal, out/out, 2)

Figure 5.32: Legend of Zelda: Link's Awakening rule: BS2 - Two-way lock

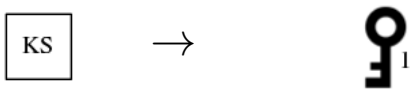
BS2 - Two-way lock: See Figure 5.32. This creates a lock where the player will be allowed to return to the in-traversal from the out-traversal.



(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, out/out, 4)

Figure 5.33: Legend of Zelda: Link's Awakening rule: BS3 - Early key

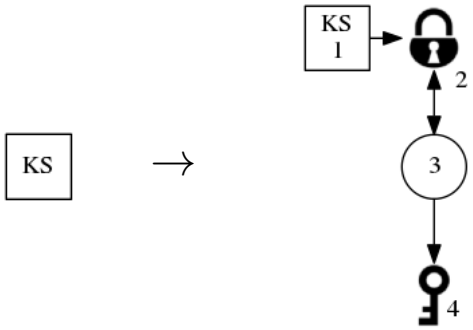
BS3 - Early key: See Figure 5.33. The key for a lock is situated such that a player could find it a number of sections before encountering the lock. Alternatively, they could find the lock and then have to backtrack a number of sections in search of the key.



(Traversal, in/in, 1)
(Lock, out/out, 1)

Figure 5.34: Legend of Zelda: Link's Awakening rule: KS1 - Simple key

KS1 - Simple key: See Figure 5.34. This places a key that will unlock the adjacent lock. This is the terminal for all of the key search (KS) rules.



(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, in/out, 2)
(Lock, out/out, 4)

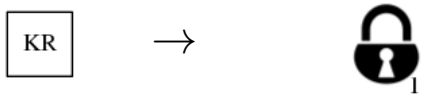
Figure 5.35: Legend of Zelda: Link's Awakening rule: KS2 - Stacking key search

KS2 - Hidden key: See Figure 5.35. This rule creates a stacked lock. That means that it places the key for the adjacent lock, but that key is behind another lock. The key for the new lock will be produced by the key search.

(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Lock, out/out, 1)
(Lock, out/out, 2)

Figure 5.36: Legend of Zelda: Link's Awakening rule: KS3 - Multiple keys

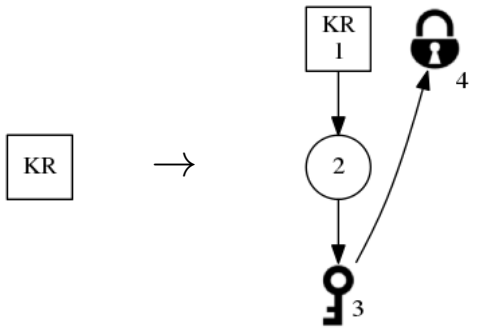
KS3 - Multiple keys: See Figure 5.36. This increases the number of keys that the player has to find to open the lock connected to the original key search. Each key could be concealed using the other key search productions.



(Key, in/in, 1)
(Traversal, in/in, 1)
(Traversal, in/out, 1)
(Traversal, out/in, 1)
(Traversal, out/out, 1)

Figure 5.37: Legend of Zelda: Link's Awakening rule: KR1 - Simple key retrieval

KR1 - Simple key retrieval: See Figure 5.37. These are productions where we have already defined the location of the key and will be placing the associated lock. This is an inversion of the key search productions above. This non-terminal node is only created in the rule PS4 above. A key has been placed, and this covers how the lock that it opens is placed.



(Key, in/in, 1)
(Traversal, in/in, 1)
(Traversal, in/in, 4)
(Traversal, in/out, 4)
(Traversal, out/in, 4)
(Traversal, out/out, 4)

Figure 5.38: Legend of Zelda: Link's Awakening rule: KR2 - Stacking key retrieval

KR2 - Stacking key retrieval: See Figure 5.38. Key retrievals are created as part of situations where the end of the branching path is a key. This creates a second layer to the branching path that will make the key retrieval longer.

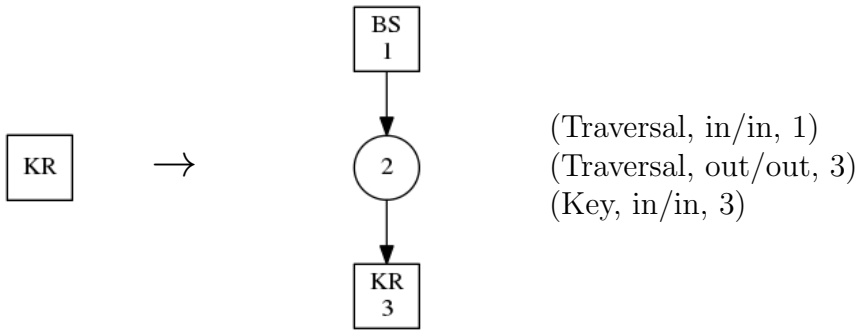


Figure 5.39: Legend of Zelda: Link's Awakening rule: KR3 - Key retrieval with preceding backtrackable section

KR3 - Key retrieval with preceding backtrackable section: See Figure 5.39. This extends the key retrieval by placing a backtrackable section before the key retrieval. This simply extends the distance of the path.

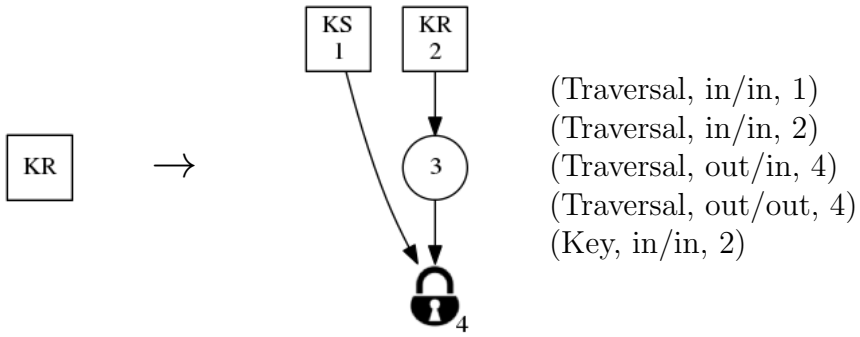
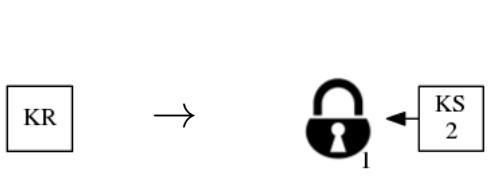


Figure 5.40: Legend of Zelda: Link's Awakening rule: KR4 - Pre-keyed key retrieval

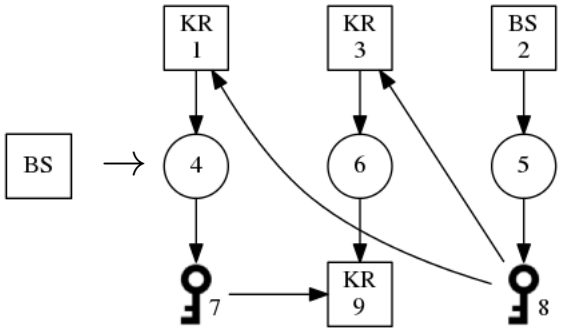
KR4 - Pre-keyed key retrieval: See Figure 5.40. This extends the key retrieval by placing a lock after it and a key search attached to the same in-traversal as the new key retrieval. The player will need to find the key in key search 1 before they can get through lock 4.



(Traversal, in/in, 1)
(Traversal, in/out, 1)
(Traversal, out/in, 1)
(Traversal, out/out, 1)
(Key, in/in, 1)
(Traversal, in/in, 2)

Figure 5.41: Legend of Zelda: Link's Awakening rule: KR5 - Simple key retrieval with extra key search

KR5 - Simple key retrieval with extra key search: See Figure 5.41. This adds at least one additional key to the lock within the key retrieval. This means that the player will need to overcome more challenges before passing through to the area beyond the lock.



(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Traversal, in/in, 3)
(Traversal, out/out, 9)

Figure 5.42: Legend of Zelda: Link's Awakening rule: BS4 - Simple key Item setup

BS4 - Simple key item setup: See Figure 5.42. In the *Legend of Zelda* games, there is a typical structure. The key item, represented by key 8 is required to access the area with the nightmare lock (key retrieval 9) as well as the nightmare key (key 7) which will open it. The paths to the key item, nightmare key, and nightmare lock all converge at the in-traversal.

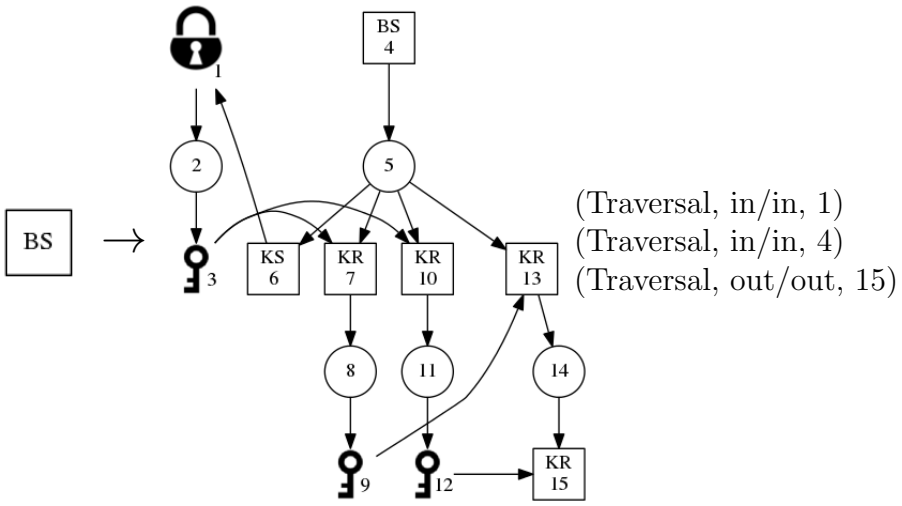


Figure 5.43: Legend of Zelda: Link's Awakening rule: BS5 - Key Item setup with Item at start

BS5 - Key Item setup with Item at start: See Figure 5.43. This is a similar configuration to that of *BS4*, but the key item is located close to the beginning (key 3). The key required to reach the key item is concealed later in the level (key search 6). This means that the player will need to explore and overcome some challenges before finding their way back to the start of the level.

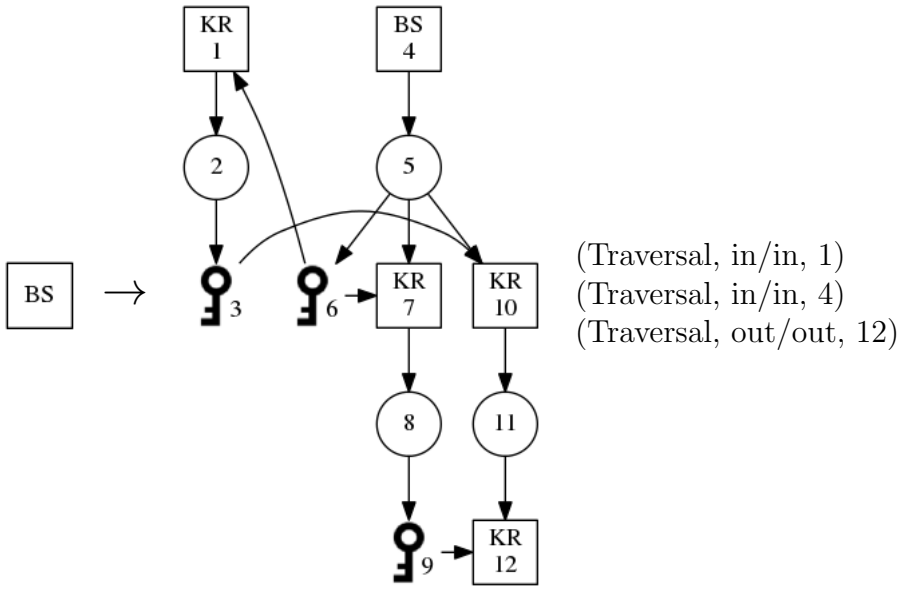


Figure 5.44: Legend of Zelda: Link's Awakening rule: BS6 - Key Item setup with Item in middle

BS6 - Key Item setup with Item in middle: See Figure 5.44. This is similar to *BS4* and *BS5* above. The difference is that the key item (key 6) is located in a central part of the level. It will give access to the nightmare key (key 9) and another key which they will need to backtrack to find (key 3).

5.4.2 Levels

The dungeons that we have analysed for *Legend of Zelda: Link's Awakening* all follow a similar pattern. At the end of each dungeon the player will find one of the eight instruments of the siren. The main quest of the game is to collect these instruments that will allow the hero, Link, to wake the Wind Fish and escape the island that he is trapped on.

Tail Cave

Tail Cave is the first dungeon in *Legend of Zelda: Link's Awakening*. The player will find the *Roc's Feather* item while exploring the cave, which allows them to jump across chasms. At the end of the dungeon they will find the *Moon Cello*. We used this level as an example of how we produce a lock-and-key puzzle graph for a level in Section 4.2.3. The graph of the level is shown if figure 5.45. The full derivation of this graph is shown in Appendix B.3.1 (pg. 190).

Bottle Grotto

Bottle Grotto is full of large jars that the player cannot move at first. After finding the *Power Bracelet* they are able to lift them up and move them out of the way. At the end of the dungeon they will find the *Conch Horn*. The graph of the level is shown if figure 5.46. The full derivation of this graph is shown in Appendix B.3.2 (pg. 196).

Key Cavern

Key Cavern has a section where the player can have many keys at once. The key item of the dungeon is the *Pegasus Boots* which allow the player to run quickly and break through certain rocks. At the end of the dungeon the player will find the *Sea Lily's Bell*. The graph of the level is shown if figure 5.47. The full derivation of this graph is shown in Appendix B.3.3 (pg. 201).

Angler's Tunnel

Angler's Tunnel has large sections of it filled with water. After the player finds the *flippers* they will be able to swim in sections of deep water. At the end of the dungeon the player will find the *Surf Harp*. The graph of the

level is shown if figure 5.48. The full derivation of this graph is shown in Appendix B.3.4 (pg. 210).

Catfish's Maw

Catfish's Maw takes place inside of a giant catfish. The key item is the *Hook Shot*, which is a grappling hook that allows the player to pull themselves across large gaps. At the end of the dungeon the player will find the *Wind Marimba*. The graph of the level is shown if figure 5.49. The full derivation of this graph is shown in Appendix B.3.5 (pg. 219).

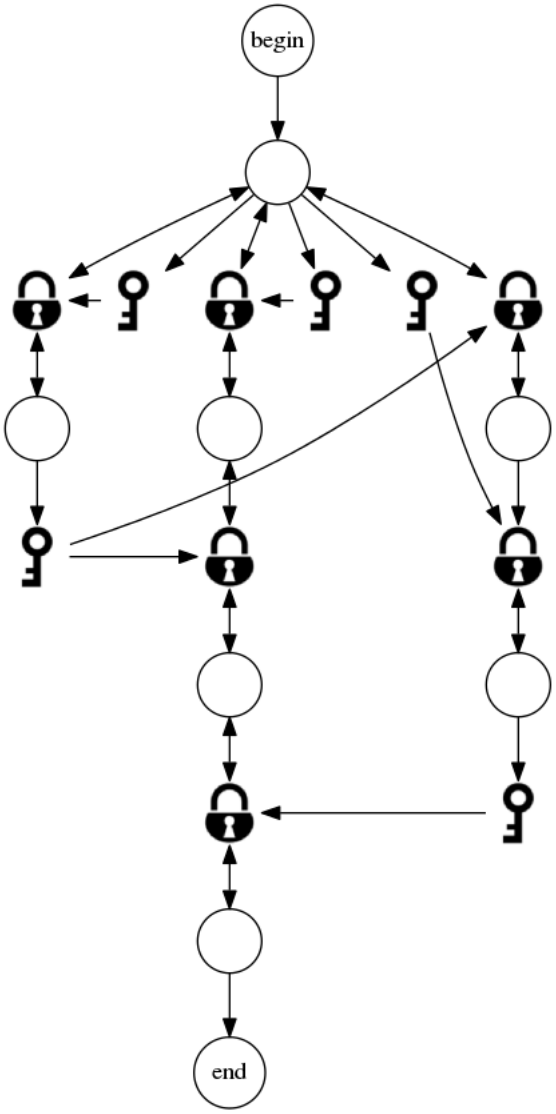


Figure 5.45: *Tail Cave* final graph

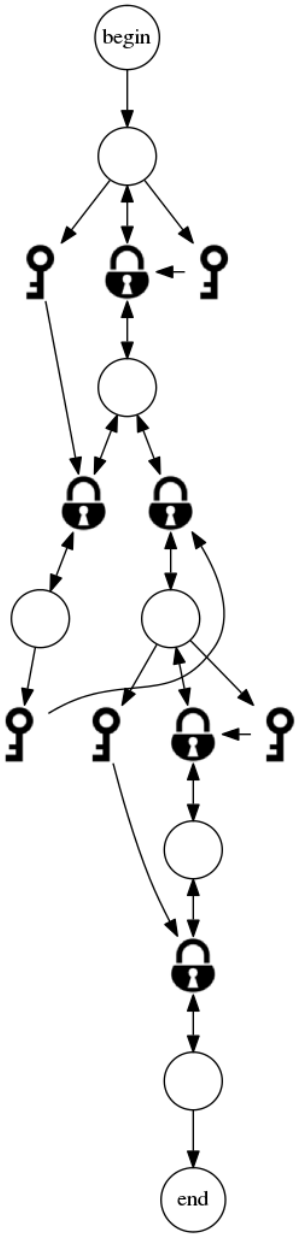


Figure 5.46: *Bottle Grotto* final graph

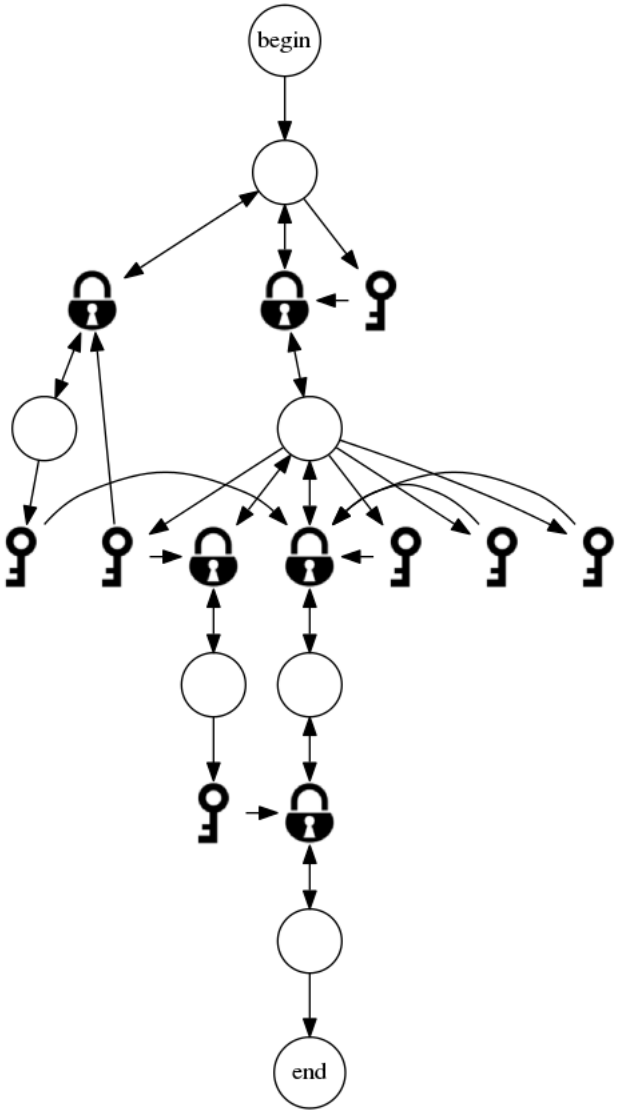


Figure 5.47: *Key Cavern* final graph

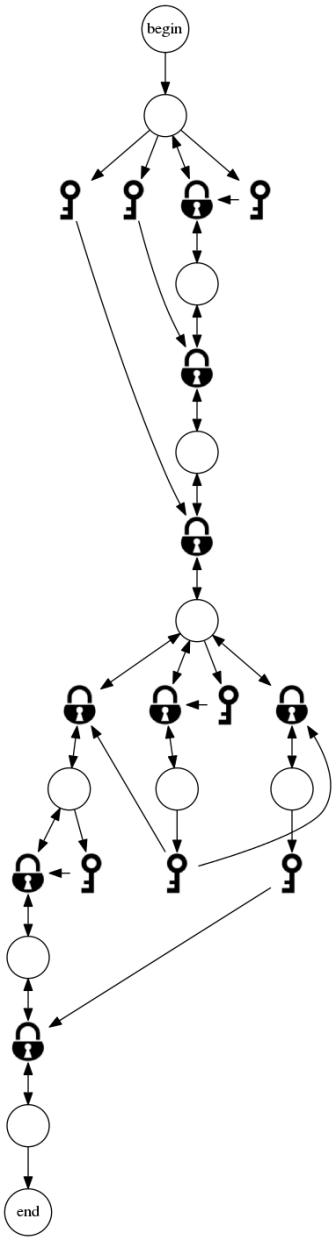


Figure 5.48: *Angler's Tunnel* final graph

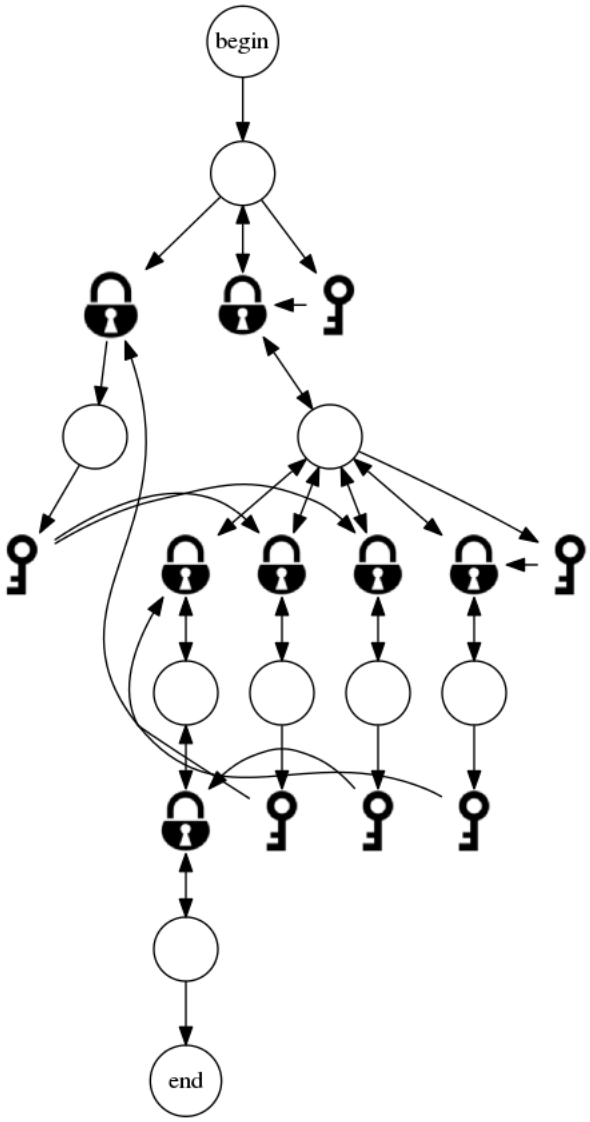


Figure 5.49: *Catfish's Maw* final graph

5.5 Legend of Zelda: Ocarina of Time

In this section we will present the grammar that we developed for *Legend of Zelda: Ocarina of Time*. Section 5.5.1 presents the grammar rules. Section 5.5.2 displays the levels that we analysed. An overview of *Tomb Raider* is given in Section 2.2.3 (pg. 12).

There are two notable analyses of *Legend of Zelda: Ocarina of Time* that we came across in our research. It was discussed by Brown (2016b) as part of their Boss Keys series. Rees (2004) analysed some dungeons and the overall game structure as lock-and-key structures. Their graph for the level *Inside the Great Deku Tree* is shown in Figure 5.50. We discuss this level in Section 5.5.2.

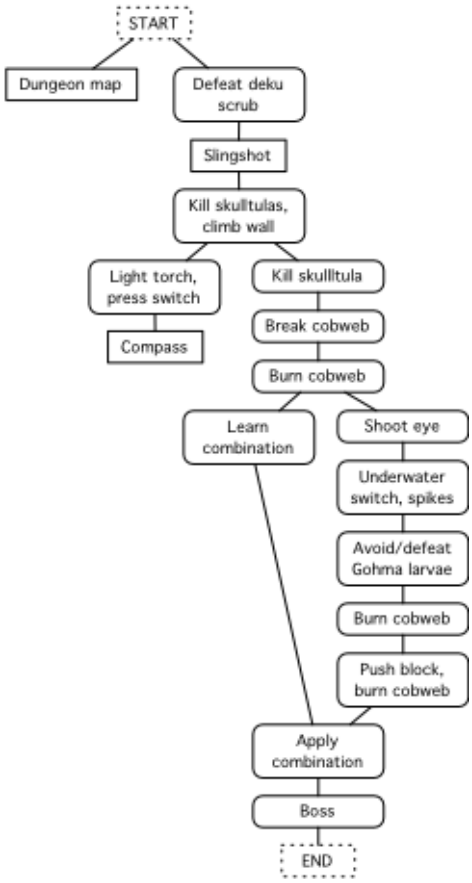


Figure 5.50: Puzzle structure of the Deku Tree (Rees, 2004)

5.5.1 Rules

In this section we will present the grammar that we developed for *Legend of Zelda: Ocarina of Time*. Section 5.5.1 presents the grammar rules. Section 5.5.2 displays the levels that we analysed. An overview of *Legend of Zelda: Ocarina of Time* is given in Section 2.2.3 (pg. 12).

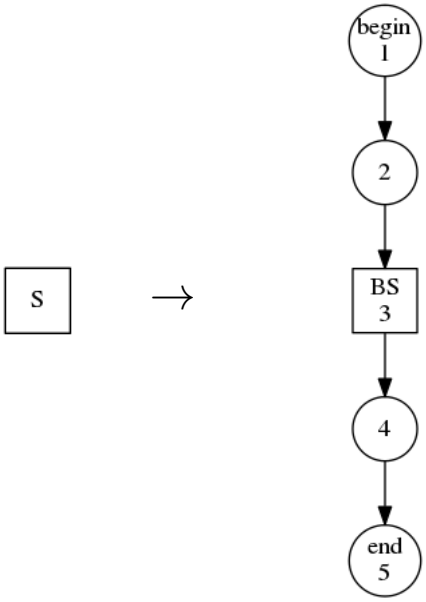


Figure 5.51: Legend of Zelda: Ocarina of Time rule: S1 - Start

S1 - Start: See Figure 5.51. This is the start production. This creates the start and end nodes as well as a puzzle section between them. Note that the edges indicate the direction that the player must go. This concept was discussed in Section 4.3.1.

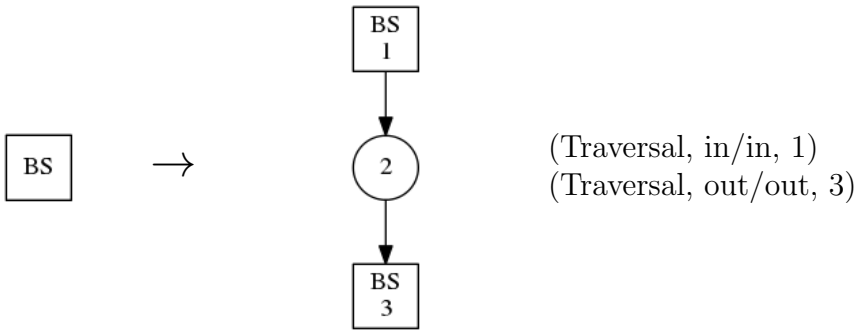


Figure 5.52: Legend of Zelda: Ocarina of Time rule: BS1 - Puzzle Extension

BS1 - Puzzle extension: See Figure 5.52. This production extends the puzzle. One puzzle section is replaced by two successive ones. Calling this repeatedly will make the puzzle longer. These are backtrackable sections, so the player will be allowed to return to the area that they came from after passing through the sections.

We have ensured that the puzzle section non-terminal is always found with a traversal leading in and a traversal leading out. It is guaranteed that a player who reaches the traversal leading in will be able to proceed to the traversal afterwards.

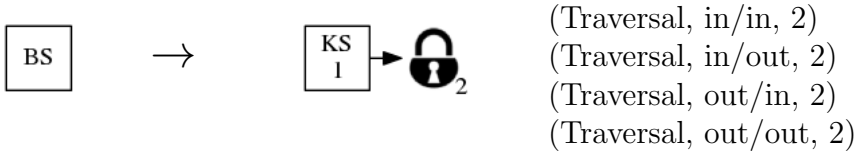


Figure 5.53: Legend of Zelda: Ocarina of Time rule: BS2 - Two-way lock

BS2 - Two-way lock: See Figure 5.53. This creates a lock where the player will be allowed to return to the in-traversal from the out-traversal.

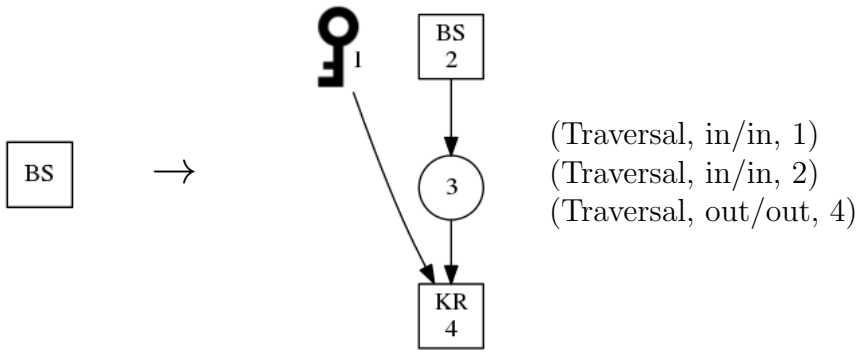


Figure 5.54: Legend of Zelda: Ocarina of Time rule: BS3 - Early key

BS3 - Early key: See Figure 5.54. The key for a lock is situated such that a player could find it a number of sections before encountering the lock. Alternatively, they could find the lock and then have to backtrack a number of sections in search of the key.

Figure 5.55: Legend of Zelda: Ocarina of Time rule: KS1 - Simple key

KS1 - Simple key: See Figure 5.55. This places a key that will unlock the adjacent lock. This is the terminal for all of the key search (KS) rules.

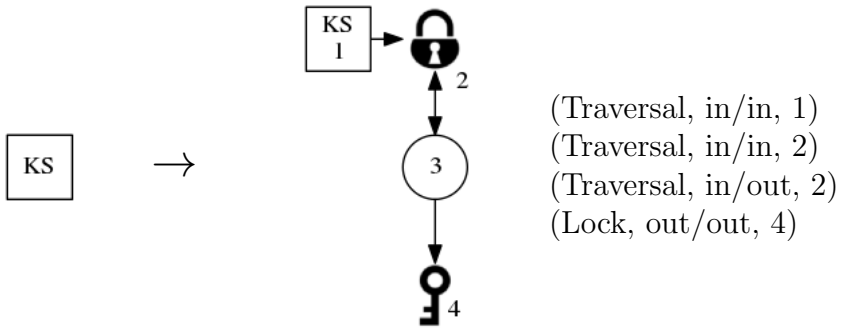


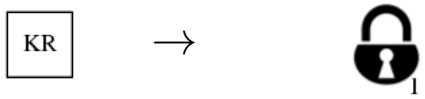
Figure 5.56: Legend of Zelda: Ocarina of Time rule: KS2 - Stacking key search

KS2 - Hidden key: See Figure 5.56. This rule creates a stacked lock. That means that it places the key for the adjacent lock, but that key is behind another lock. The key for the new lock will be produced by the key search.

(Traversal, in/in, 1)
(Traversal, in/in, 2)
(Lock, out/out, 1)
(Lock, out/out, 2)

Figure 5.57: Legend of Zelda: Ocarina of Time rule: KS3 - Multiple keys

KS3 - Multiple keys: See Figure 5.57. This increases the number of keys that the player has to find to open the lock connected to the original key search. Each key could be concealed using the other key search productions.



(Key, in/in, 1)
(Traversal, in/in, 1)
(Traversal, in/out, 1)
(Traversal, out/in, 1)
(Traversal, out/out, 1)

Figure 5.58: Legend of Zelda: Ocarina of Time rule: KR1 - Simple key retrieval

KR1 - Simple key retrieval: See Figure 5.58. These are productions where we have already defined the location of the key and will be placing the associated lock. This is an inversion of the key search productions above. This non-terminal node is only created in the rule PS4 above. A key has been placed, and this covers how the lock that it opens is placed.

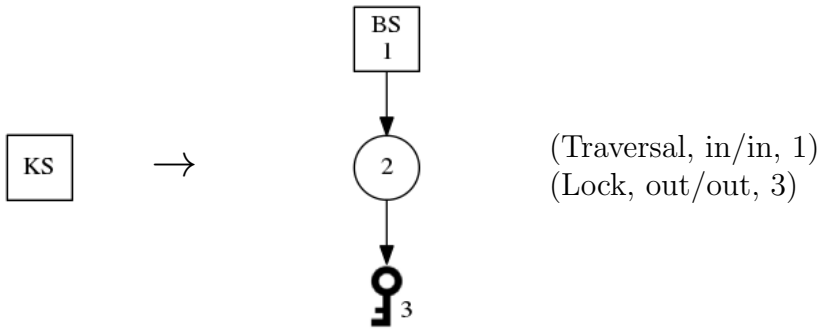


Figure 5.59: Legend of Zelda: Ocarina of Time rule: KS4 - Key with preceding BS

KS4 - Key with preceding BS: See Figure 5.59. This extends the distance that the player will need to travel to reach the key.

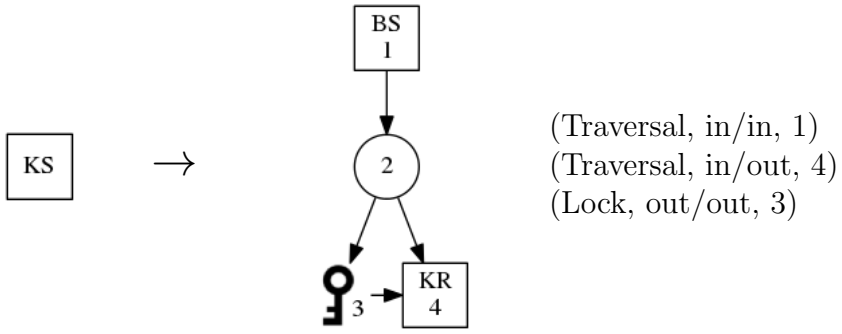


Figure 5.60: Legend of Zelda: Ocarina of Time rule: KS5 - Key retrieving loop

KS5 - Key retrieving loop: See Figure 5.60. Key 3 will open the out-lock as well as the one that will be produced by key retrieval 4. The route opened by key-retrieval leads back to the area before backtrackable section 1, the in-traversal.

5.5.2 Levels

The dungeons that we have analysed for *Legend of Zelda: Ocarina of Time* follow a similar pattern to those in *Legend of Zelda: Link's Awakening*. They have a key item that grants a new ability to the hero. We present the dungeons that we analysed below.

Inside the Great Deku Tree

Inside the Great Deku Tree is the first dungeon of *The Legend of Zelda: Ocarina of Time*. The level takes place inside a giant sentient tree. The player will find the *Fairy Slingshot* that allows them to shoot at enemies from afar. This is needed to pass a section where spiders were blocking the player from climbing a wall. After completing the dungeon the wise tree sends the hero, Link, on a quest to save a princess. The graph of the level is shown if figure 5.61. The full derivation of this graph is shown in Appendix B.4.1 (pg. 228).

Dodongo's Cavern

Link is trying to gather objects called the *Spiritual Stones*. The leader of the *Goron* people is willing to give Link the *Spiritual Stone of Fire* if he can defeat the beasts in *Dodongo's Cavern*. In the dungeon the player will find the *Bomb Bag* which allows them to carry and throw bombs. These let the player destroy things like cracked walls. The graph of the level is shown if figure 5.62. The full derivation of this graph is shown in Appendix B.4.2 (pg. 231).

Inside Jabu-Jabu's Belly

Link is asked to rescue the princess of the *Zora* people from inside *Jabu-Jabu*, a giant god-like fish. The player will find the *Boomerang* in the dungeon. It is a ranged weapon that is required for destroying some of the tentacles hurting *Jabu-Jabu*. After saving the princess Link is given the *Spiritual Stone of Water*. The graph of the level is shown if figure 5.63. The full derivation of this graph is shown in Appendix B.4.3 (pg. 235).

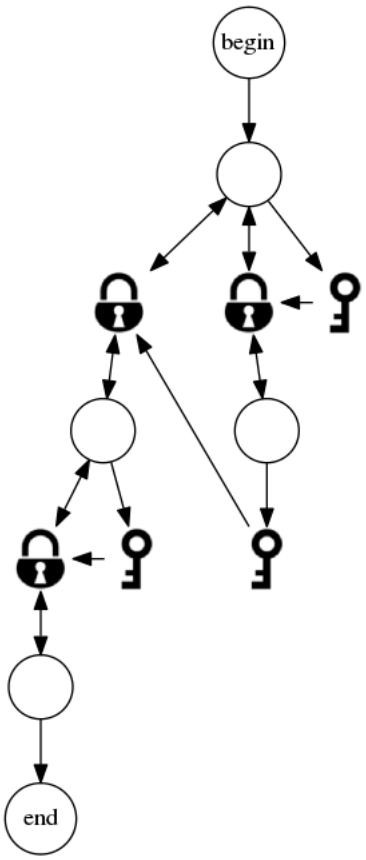


Figure 5.61: *Inside the Great Deku Tree* final graph

Forest Temple

A few years have passed since Link gathered the *Spiritual Stones*. He now needs to unite the sages to help save the kingdom. Link saves the *Sage of the Forest* by removing evil beings from the *Forest Temple*. In the dungeon the player finds the *Fairy Bow* which allows the player to shoot arrows that flip special switches.

We were able to represent the *Forest Temple* with a puzzle graph, but not generate it without an extremely complicated and specific production. The puzzle graph is shown with annotations in Figure 5.64.

Figures 5.16 (pg. 68), 5.42, 5.43, and 5.44 (pg. 84-86) are the most complicated rules that we produced in this research. We would need a rule of much greater complexity for this puzzle. This rule would also provide very little

flexibility in what would be produced from it.

As discussed in Section 4.3 a desirable rule is simple and reusable. The productions mentioned above barely satisfy these conditions.

If two nodes are adjacent, then they, or their ancestors, had to be created as adjacent in a production. All paths between two nodes have to be accounted for in a production. The sections of this puzzle are too intertwined to allow for a production rule to be designed simply.

Fire Temple

Link needs to go into the *Fire Temple* to prevent captured *Goron* people from being fed to a dragon. In the dungeon the player will find the *Megaton Hammer* which allows them to smash some rocks and rusty switches. By defeating the dragon you also save the *Sage of Fire*. The graph of the level is shown if figure 5.65. The full derivation of this graph is shown in Appendix B.4.4 (pg. 243).

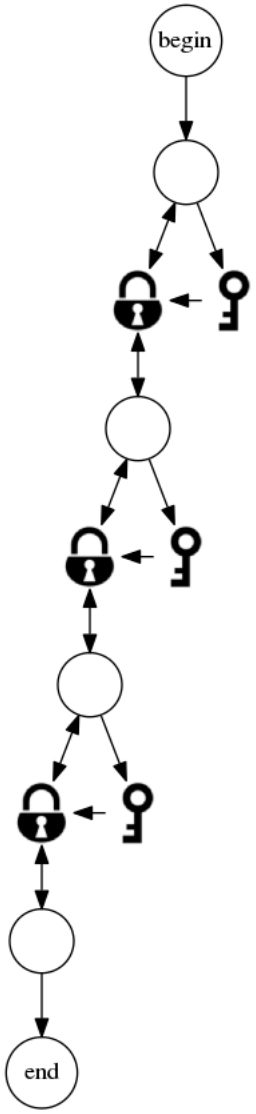


Figure 5.62: *Dodongo's Cavern* final graph

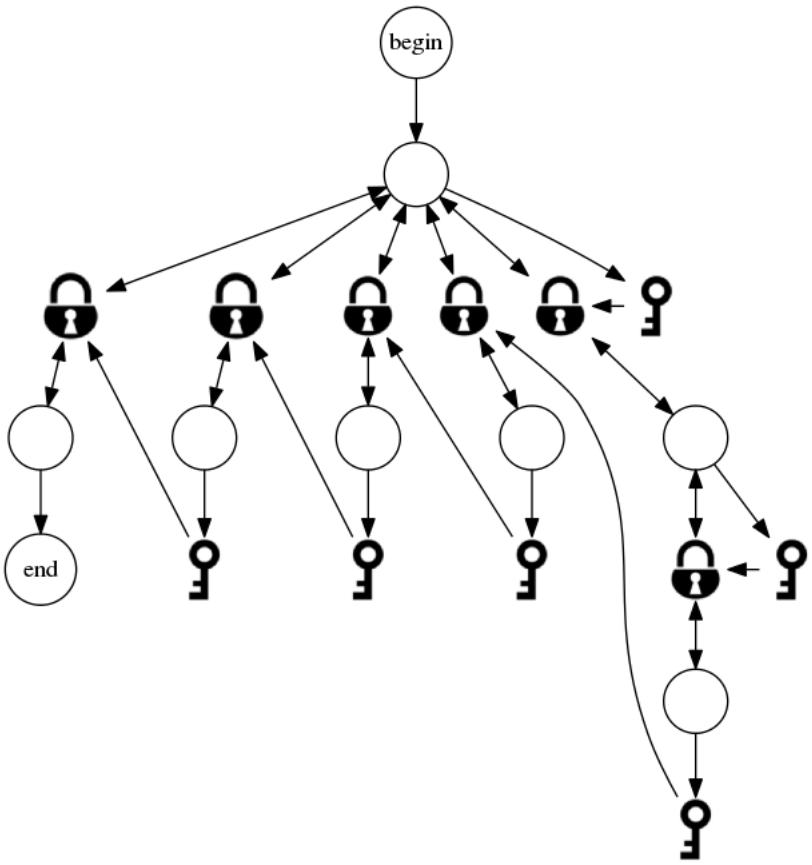


Figure 5.63: *Inside Jabu-Jabu's Belly* final graph

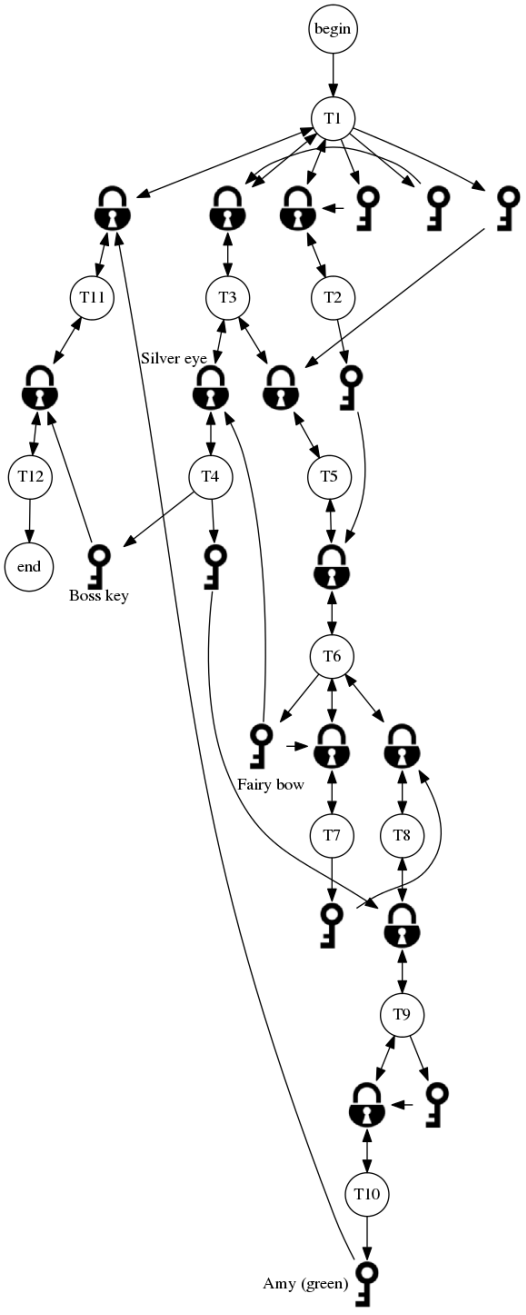


Figure 5.64: *Forest Temple* final graph

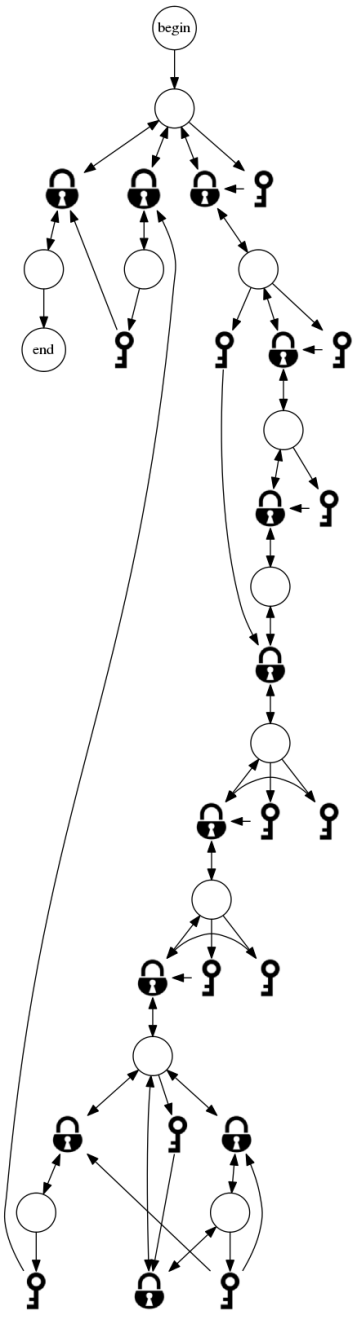


Figure 5.65: *Fire Temple* final graph

Chapter 6

Results

We can create new puzzles with the grammars that we have designed. These new puzzles have similar structures to those seen in the games that we have analysed. These specific puzzle layouts are not present in the games that we have analysed.

We wrote a program that generates puzzle graphs as discussed in Section 5.1 (pg. 57). We give it a grammar and it will randomly create puzzle graphs. The program does this by selecting a non-terminal node and a production to apply for that non-terminal subgraph. This repeats until there are no non-terminal nodes left.

All of the rules in our *Prince of Persia: The Sands of Time* grammar also exist in the other grammars, meaning that any graph created with the *Prince of Persia: The Sands of Time* grammar can also be created by one of the other grammars. There is a large overlap of rules between the grammars. As a result, some of the random graphs could have been generated by more than one of our grammars.

The largest lock-and-key structure that we had among the levels that we analysed was the *Tomb Raider* level *Atlantis* 5.3.2 (pg. 73). This level has 13 locks and 16 keys. Our grammars can expand to an infinite number of locks and keys. Results with hundreds of locks and keys are too large to practically use, so we have limited our results to puzzles with less than 100 nodes.

We present some examples of generated puzzles in the following sections.

6.1 Random Puzzles using the Prince of Persia: The Sands of Time Grammar

The following figures were generated using the *Prince of Persia: The Sands of Time* grammar, as presented in Section 5.2.1.

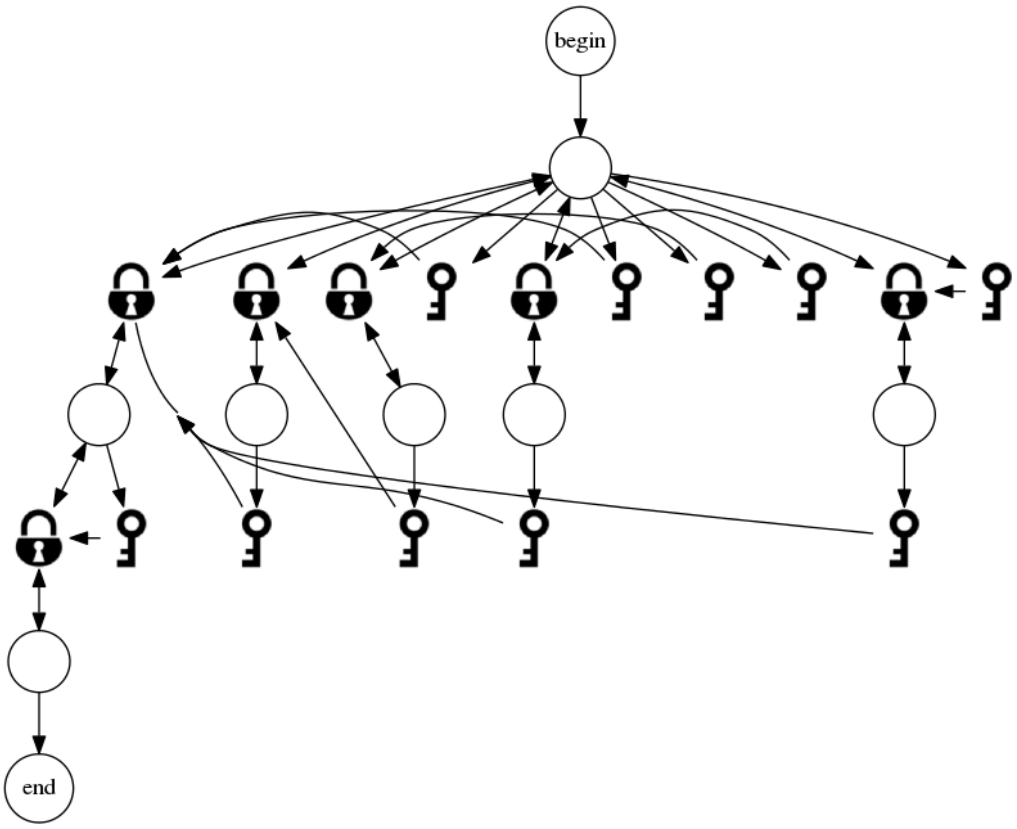


Figure 6.1: Puzzle graph created using the *Prince of Persia* grammar

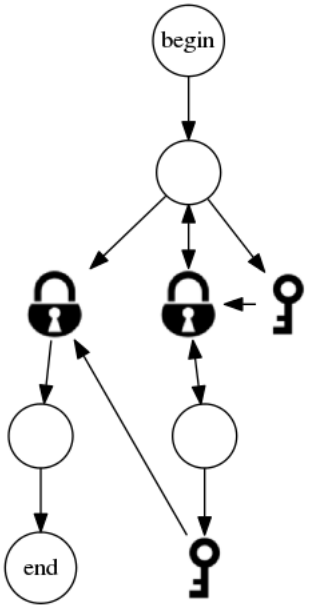


Figure 6.2: Puzzle graph created using the *Prince of Persia* grammar

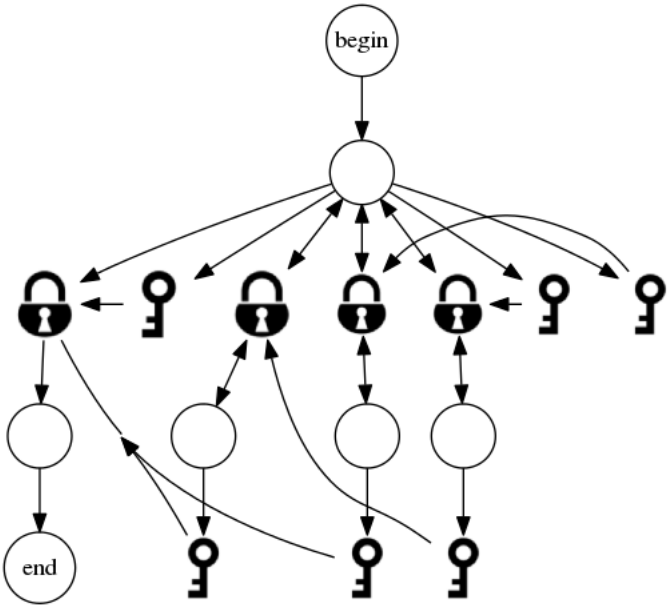


Figure 6.3: Puzzle graph created using the *Prince of Persia* grammar

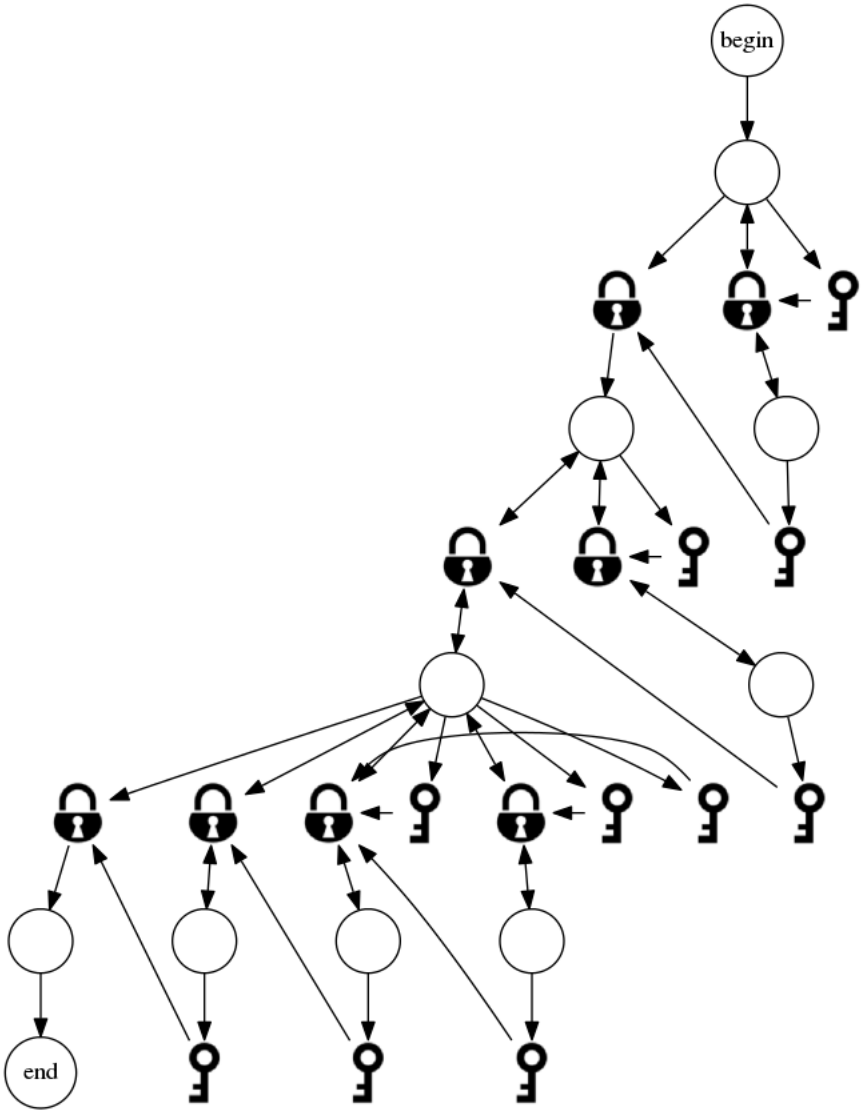


Figure 6.4: Puzzle graph created using the *Prince of Persia* grammar

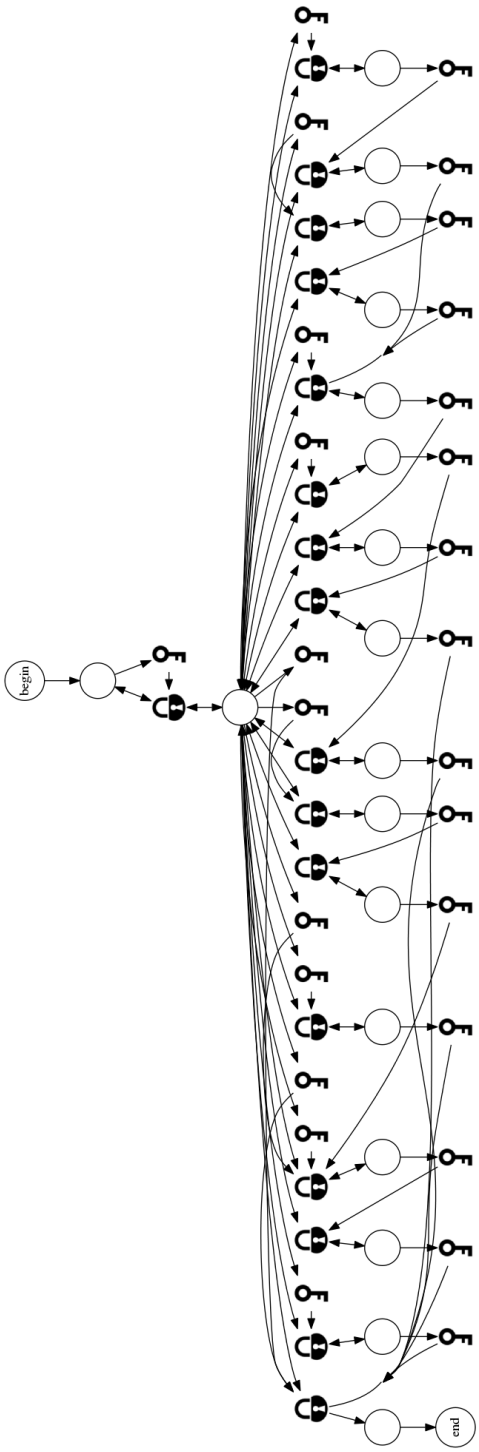


Figure 6.5: Puzzle graph created using the *Prince of Persia* grammar

6.2 Random Puzzles using the Tomb Raider Grammar

The following figures were generated using the *Tomb Raider* grammar as presented in Section 5.3.1.

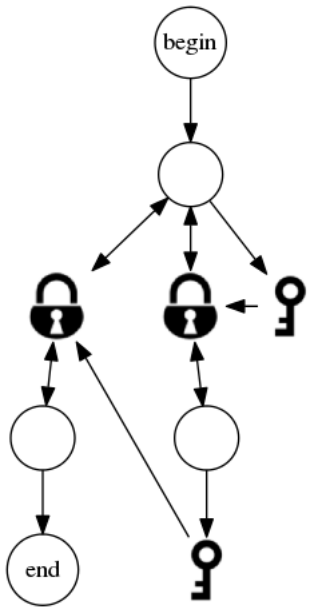


Figure 6.6: Puzzle graph created using the *Tomb Raider* grammar

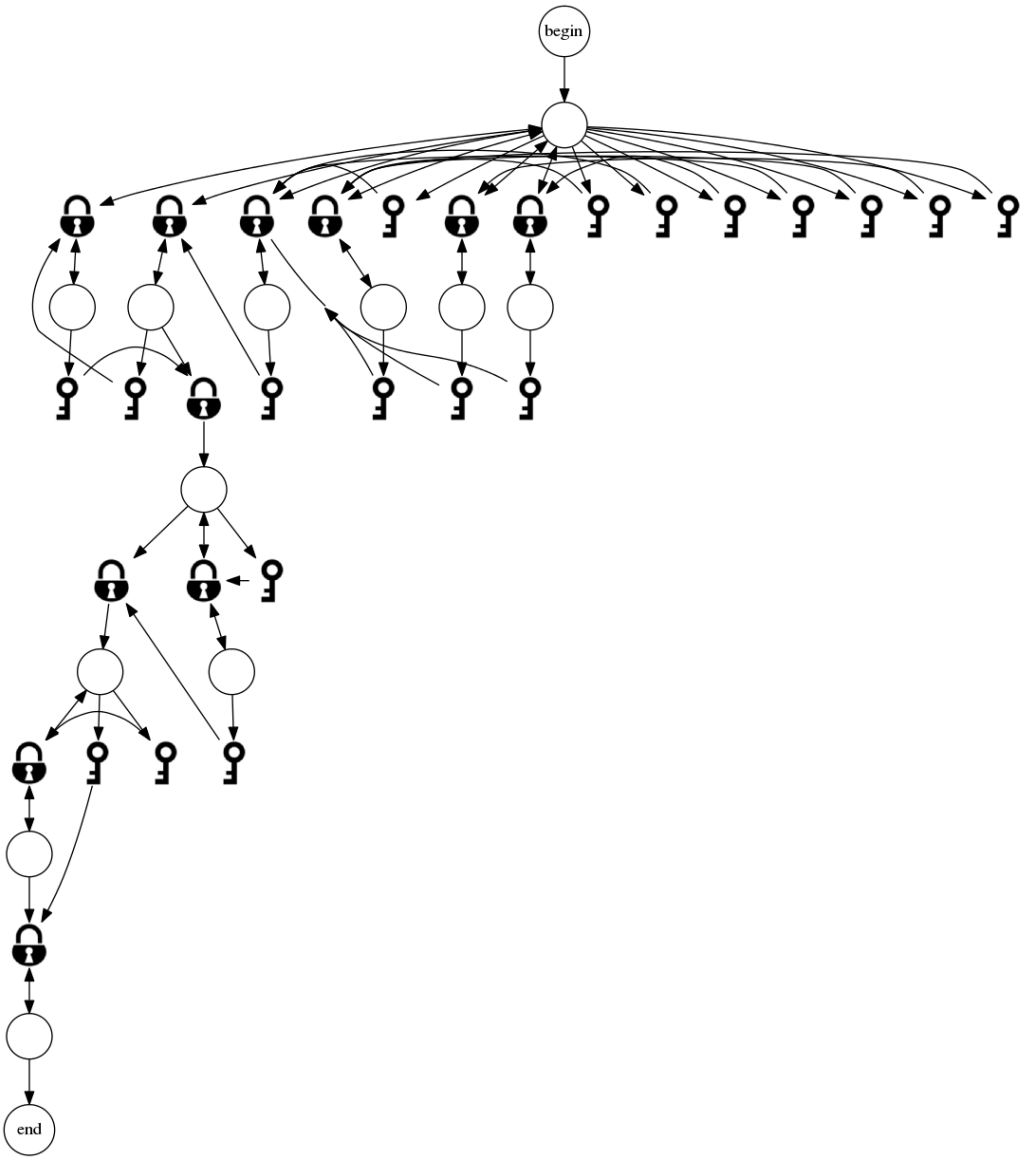


Figure 6.7: Puzzle graph created using the *Tomb Raider* grammar

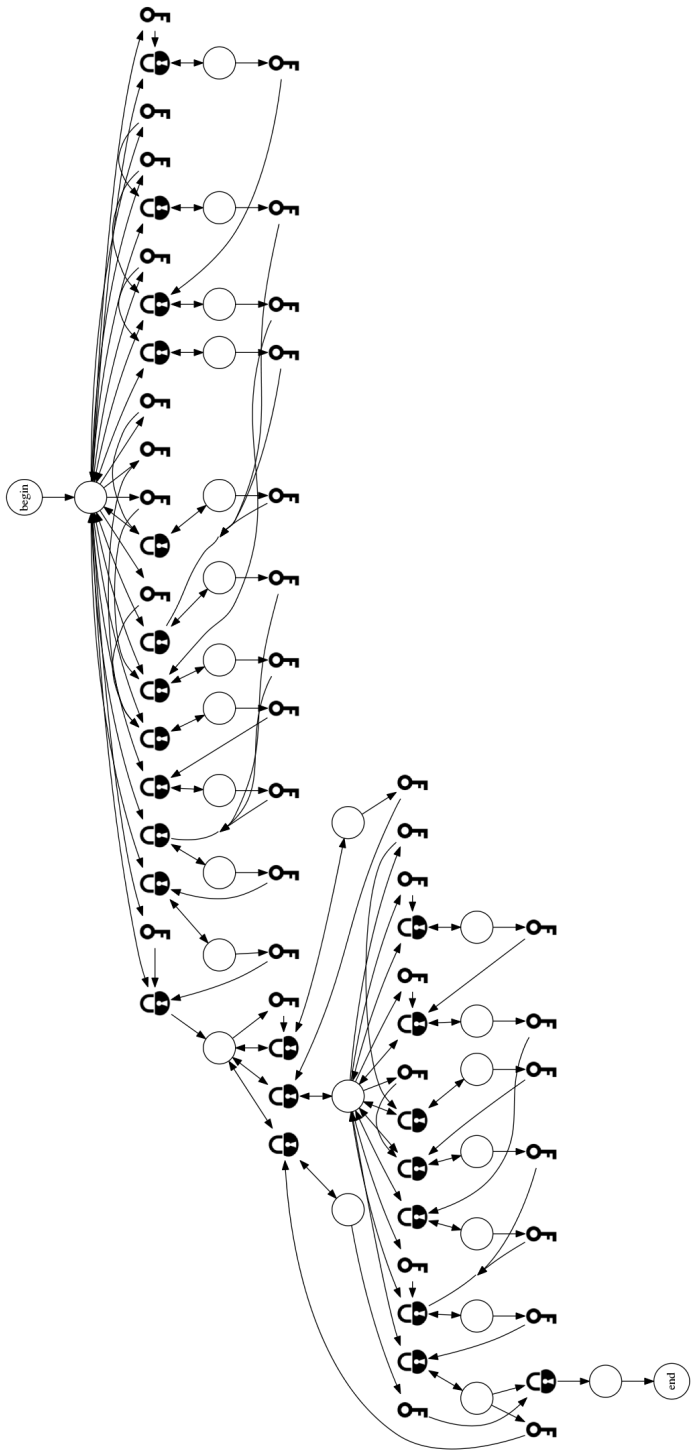


Figure 6.8: Puzzle graph created using the *Tomb Raider* grammar

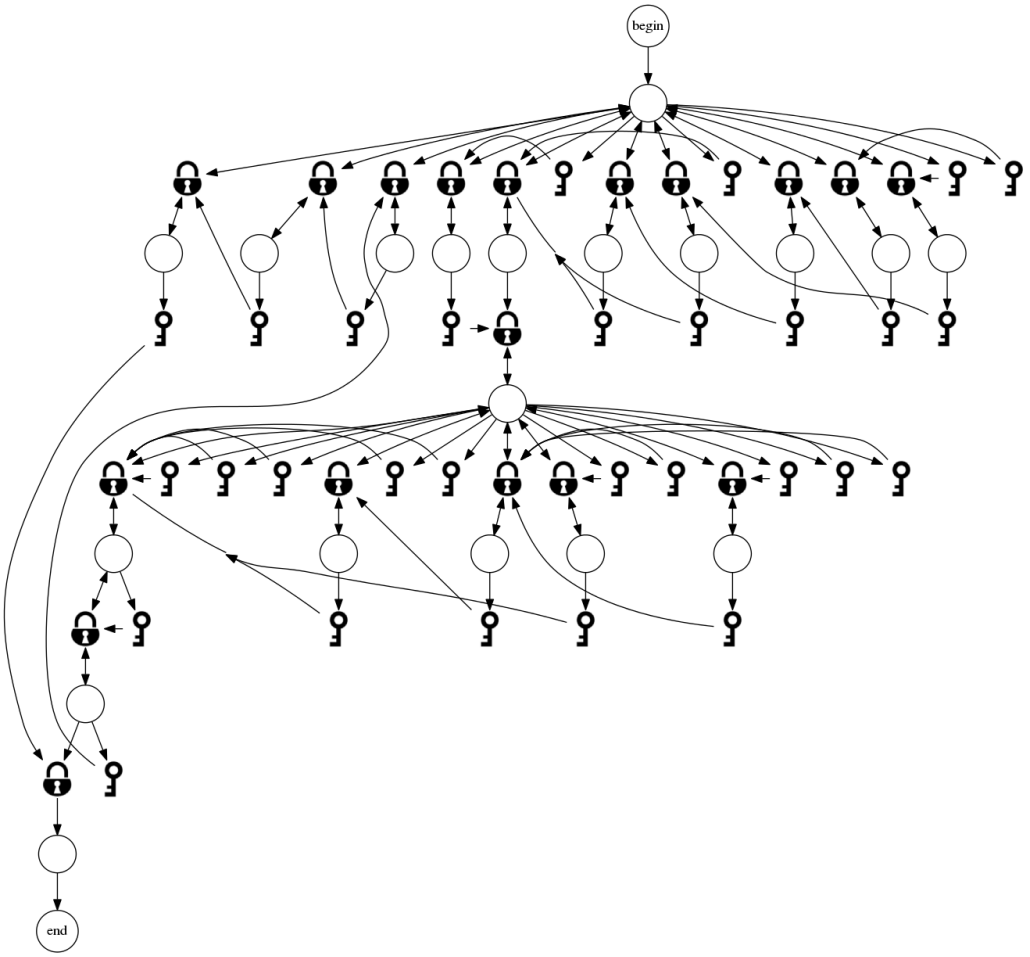


Figure 6.9: Puzzle graph created using the *Tomb Raider* grammar

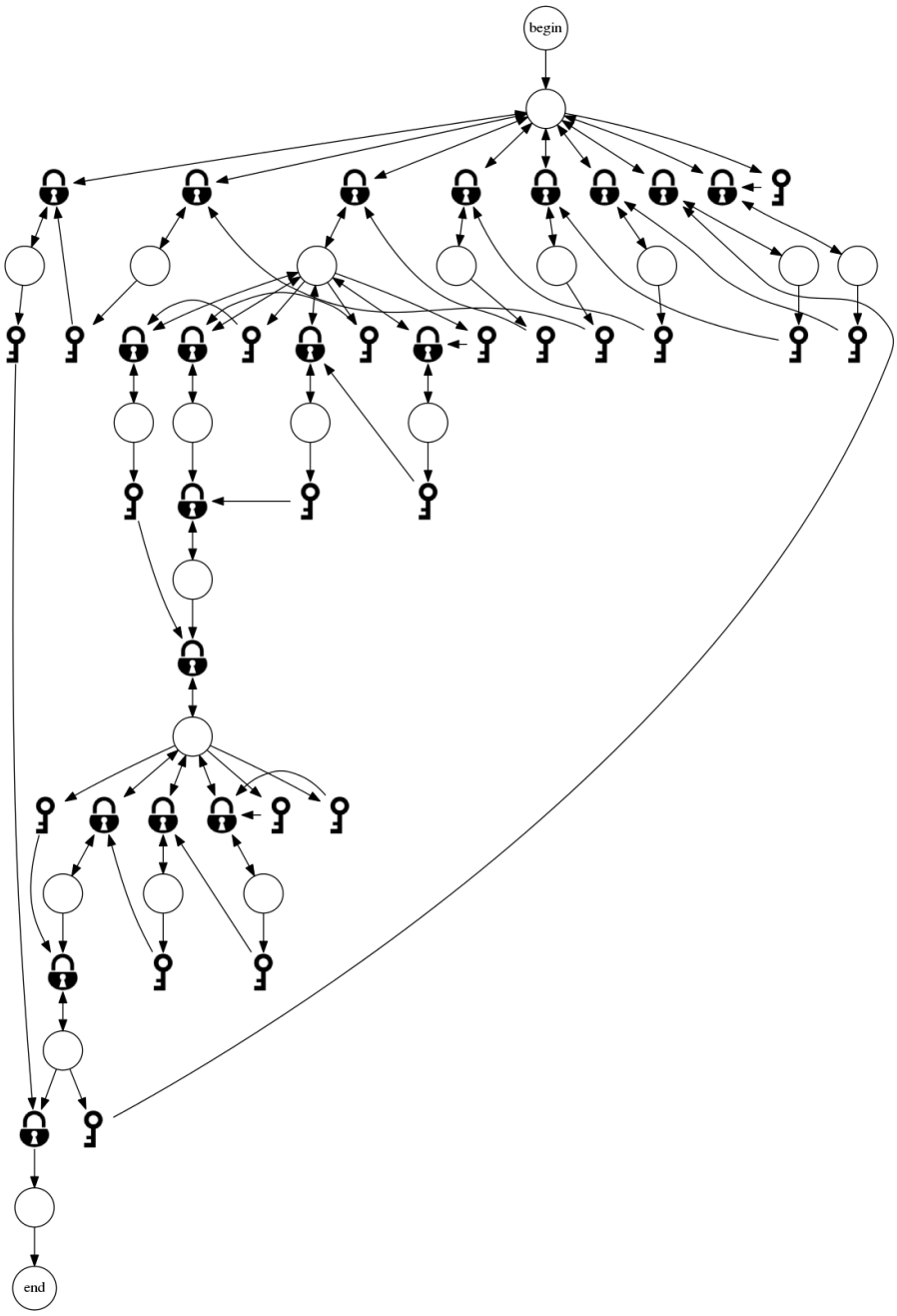


Figure 6.10: Puzzle graph created using the *Tomb Raider* grammar

6.3 Random Puzzles using the Legend of Zelda: Link's Awakening Grammar

The following figures were generated using the *Legend of Zelda: Link's Awakening* grammar as presented in Section 5.4.1. Figure 6.11 below clearly shows the key item structure that is common in the *Legend of Zelda* games.

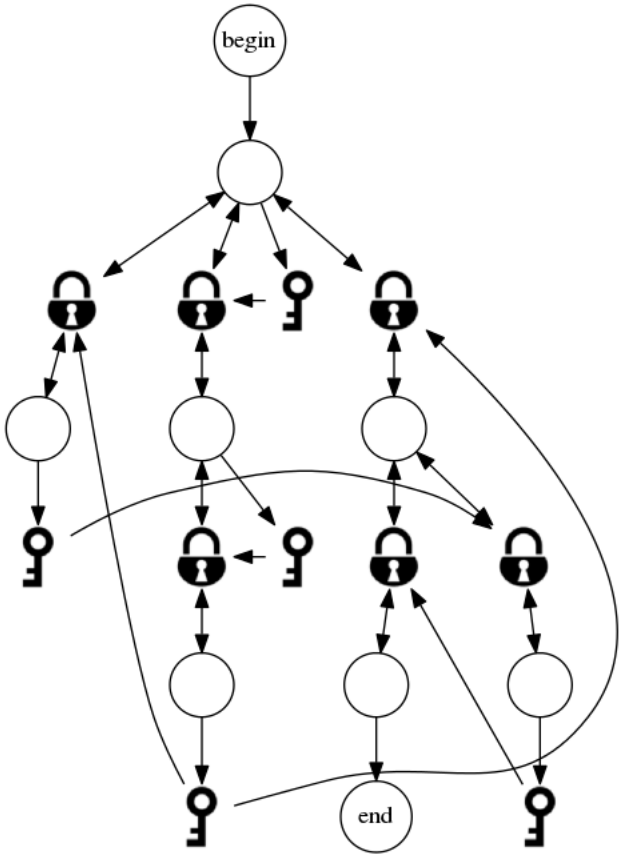


Figure 6.11: Puzzle graph created using the *Legend of Zelda: Link's Awakening* grammar

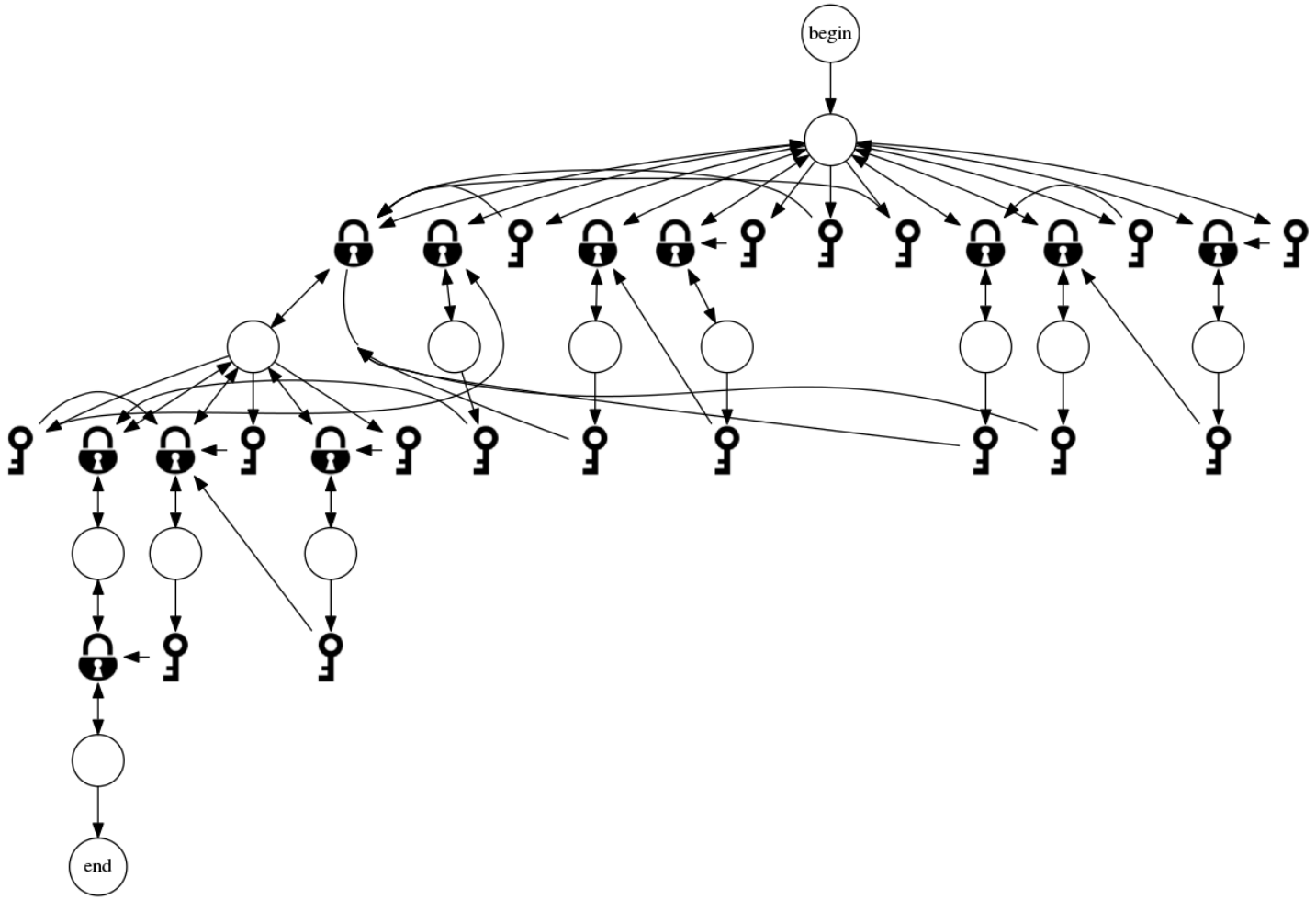


Figure 6.12: Puzzle graph created using the *Legend of Zelda: Link's Awakening* grammar

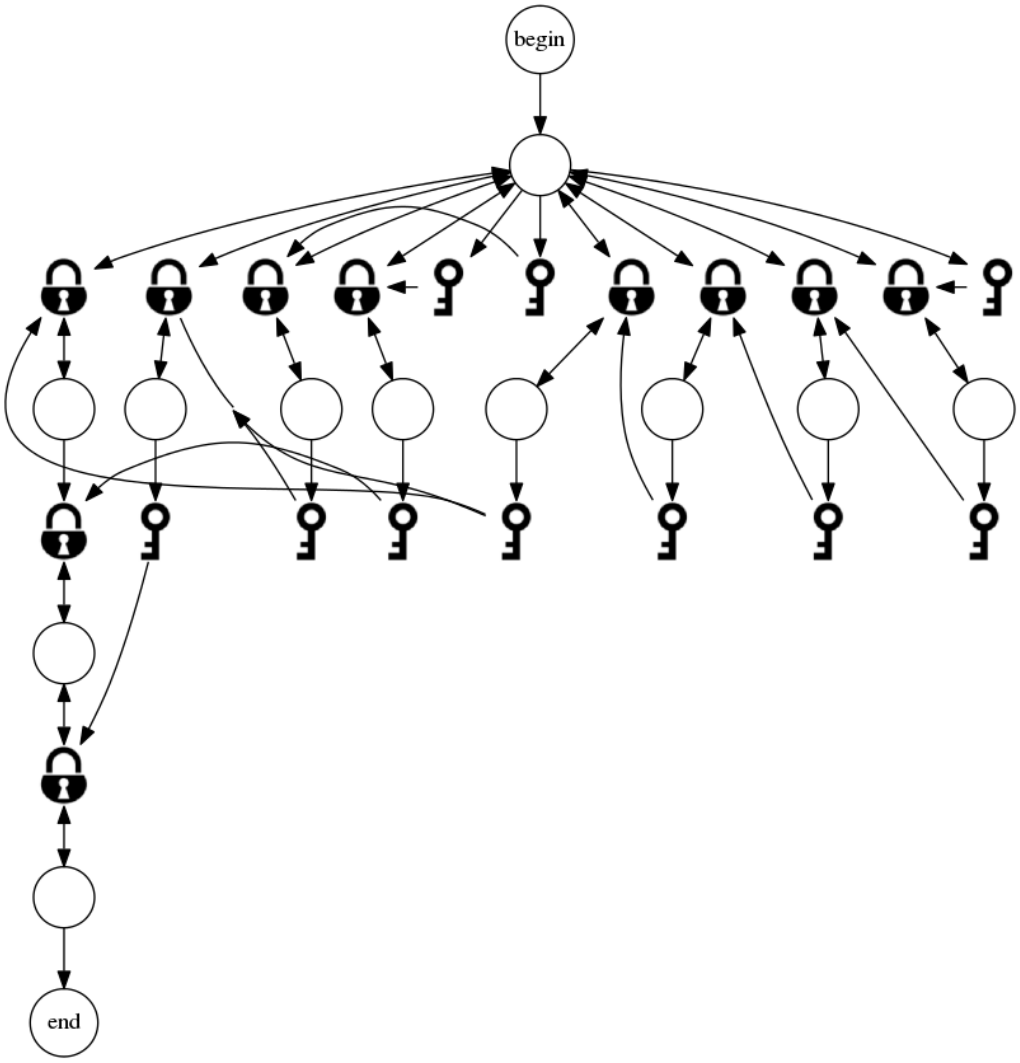


Figure 6.13: Puzzle graph created using the *Legend of Zelda: Link's Awakening* grammar

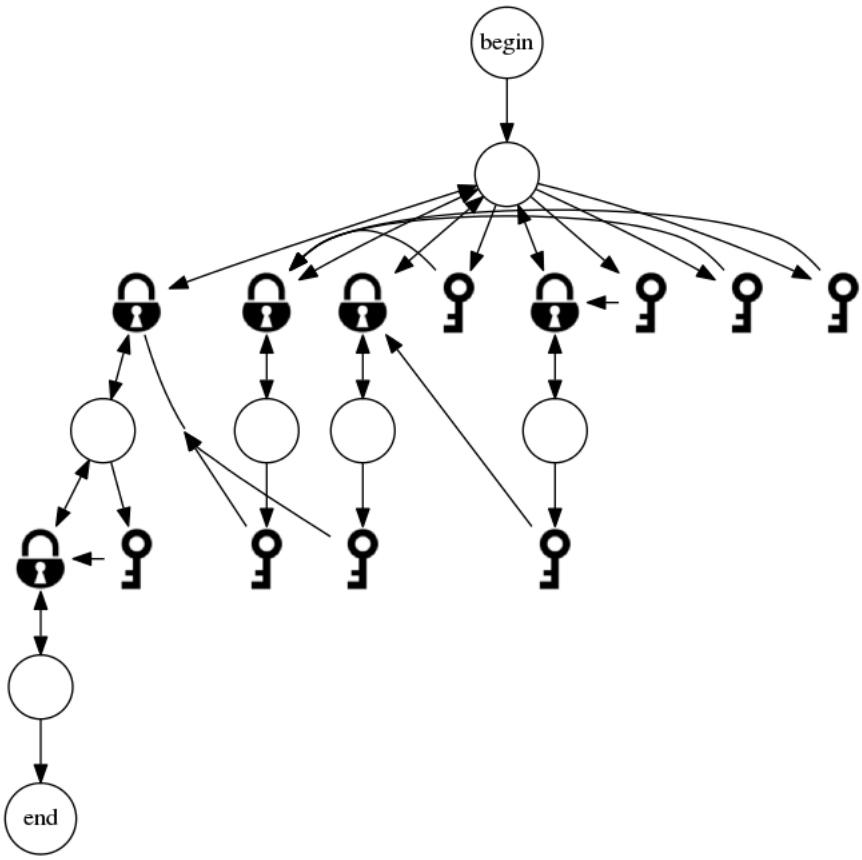


Figure 6.14: Puzzle graph created using the *Legend of Zelda: Link's Awakening* grammar

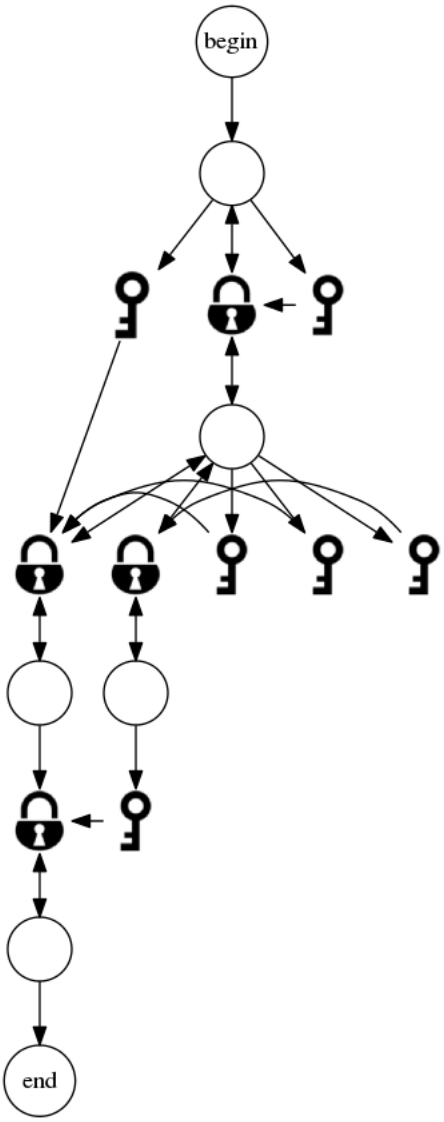


Figure 6.15: Puzzle graph created using the *Legend of Zelda: Link's Awakening* grammar

6.4 Random Puzzles using the Legend of Zelda: Ocarina of Time Grammar

The following figures were generated using the *Legend of Zelda: Ocarina of Time* grammar as presented in Section 5.5.1.

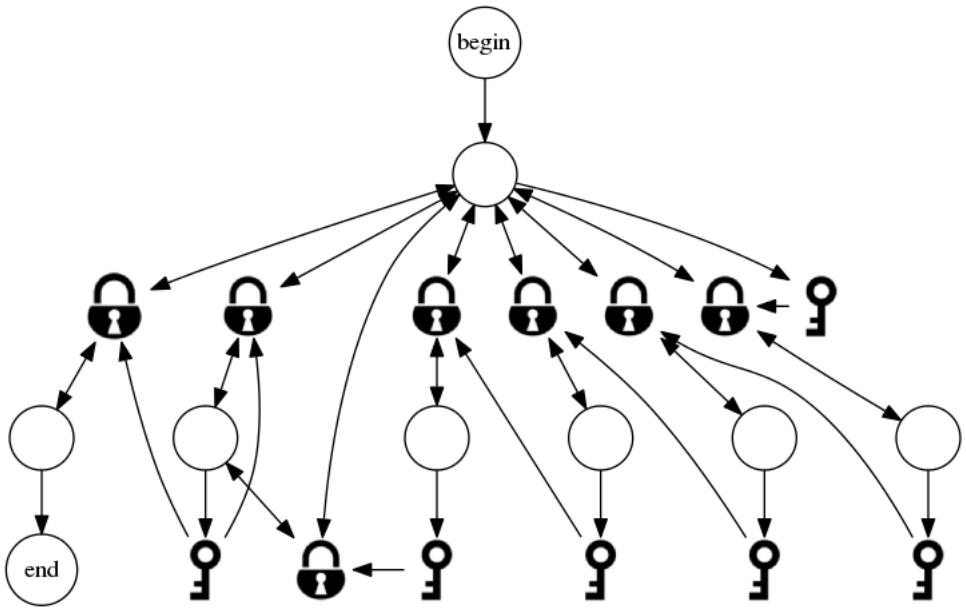


Figure 6.16: Puzzle graph created using the *Legend of Zelda: Ocarina of Time* grammar

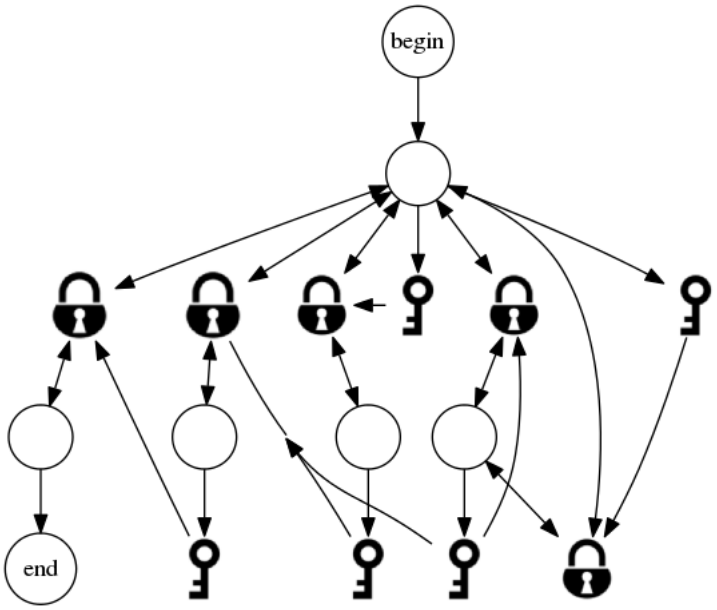


Figure 6.17: Puzzle graph created using the *Legend of Zelda: Ocarina of Time* grammar

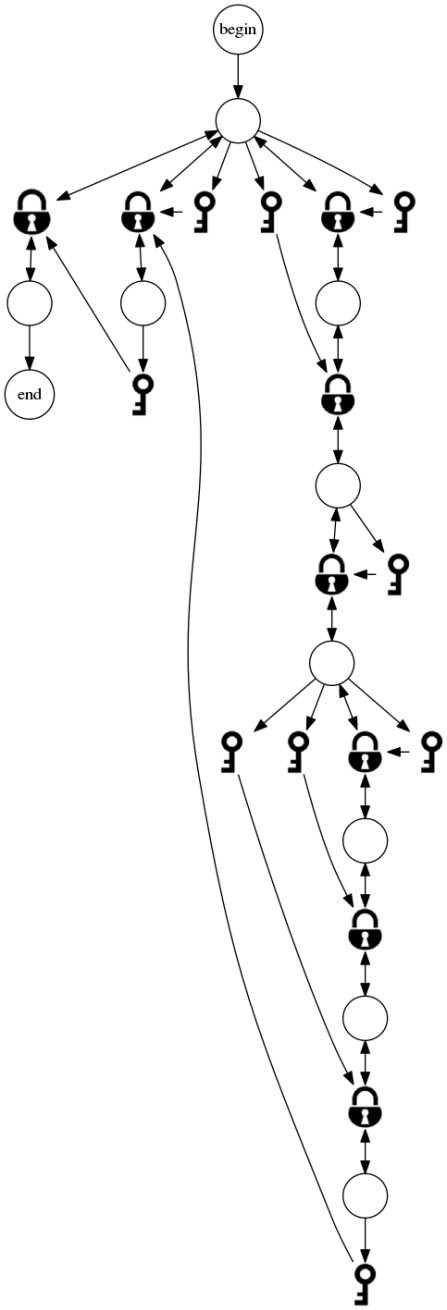


Figure 6.18: Puzzle graph created using the *Legend of Zelda: Ocarina of Time* grammar

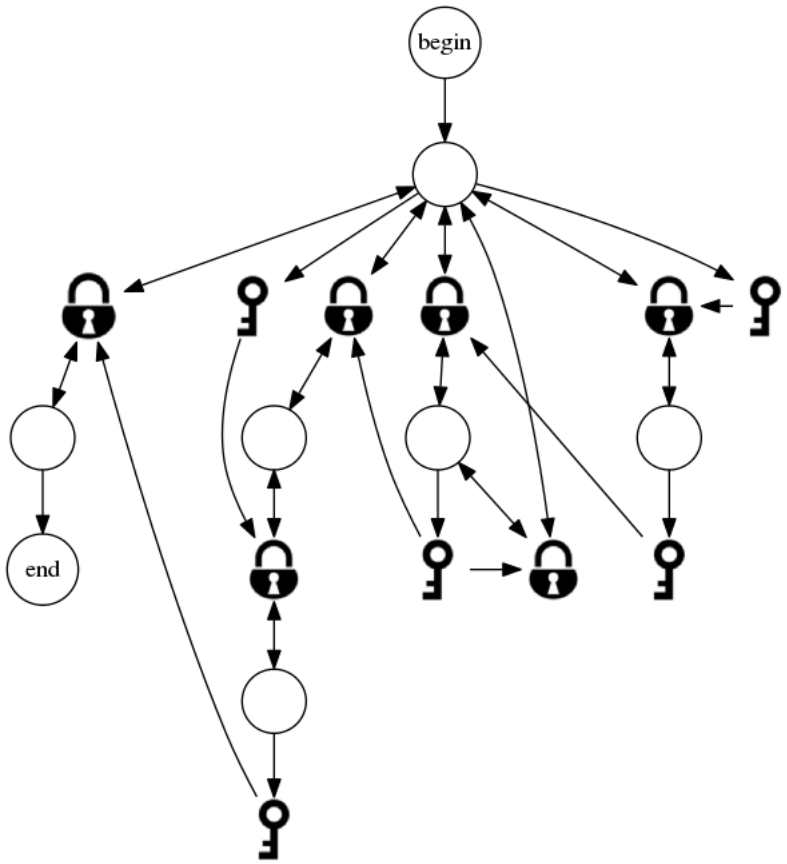


Figure 6.19: Puzzle graph created using the *Legend of Zelda: Ocarina of Time* grammar

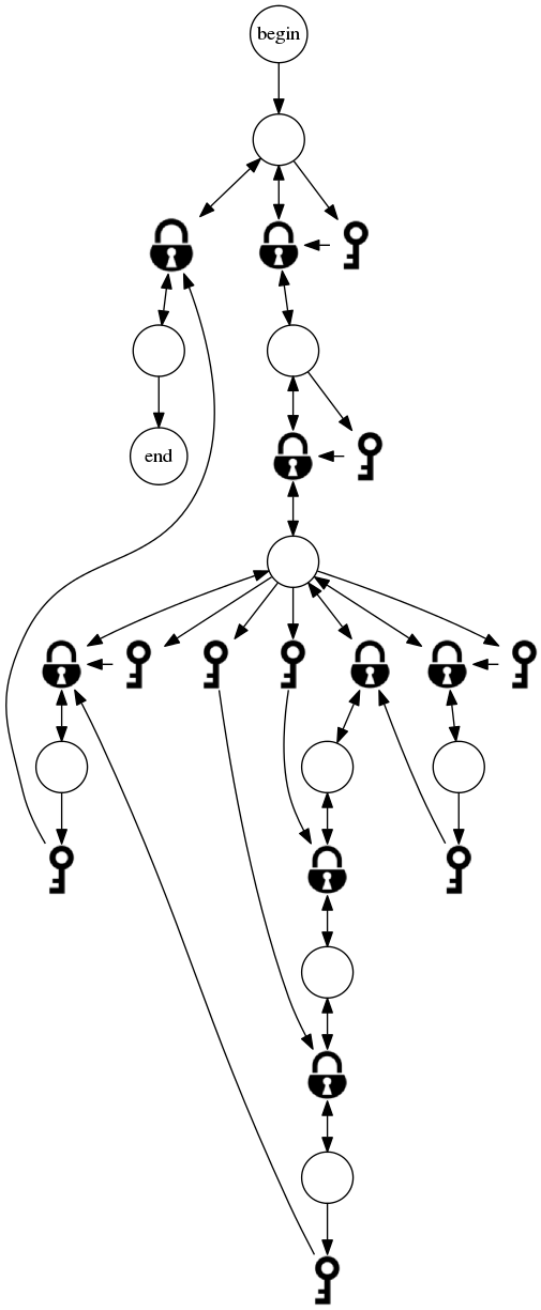


Figure 6.20: Puzzle graph created using the *Legend of Zelda: Ocarina of Time* grammar

Chapter 7

Conclusion & Future Work

The contribution of this research was to create a formal model of puzzles for action-adventure video games, and provide a framework for the procedural generation of these puzzles.

We presented a way to prove that a grammar would always produce a solvable puzzle. This is important if we want a method to reliably create levels that can be played from beginning to end.

Playing the graphs in the application serves as a proof of concept. They could also be used in combination with human designers or other Procedural Content Generation systems. A designer could use a generated graph as inspiration for a level. This would assist them with the task of coming up with novel puzzle structures. Another procedural content generator could use the graphs as a starting point for creating a level. This would allow a game to provide players with an infinite number of puzzles, greatly extending the time people can keep playing the game.

Some possible future work would be to look at escape rooms. These are puzzle games where a group of players are locked in a room and must try to escape before the time runs out. To find the key the players must solve a sequence of puzzles together. Some puzzles give the code for combination locks which open new clues, puzzles, and even sections of the room.

Bibliography

[Adams 2002] David Adams. *Automatic Generation of Dungeons for Computer Games*, 2002. BSc(Hons) Thesis. University of Sheffield, UK.

[Adams 2010] Ernest Adams. *Fundamentals of Game Design*. Pearson Education, 2010.

[Andersen *et al.* 2010] Erik Andersen, Yun-En Liu, Ethan Apter, François Boucher-Genesse, and Zoran Popović. Gameplay analysis through state projection. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 1–8. Association for Computing Machinery, 2010.

[Ashmore and Nitsche 2007] Calvin Ashmore and Michael Nitsche. The quest in a generated world. In *Proceedings of 2007 Digital Games Research Association (DiGRA) Conference: Situated Play*, pages 503–509, 2007.

[Ashmore 2006] Calvin Ashmore. *Key and Lock Puzzles in Procedural Gameplay*, 2006. MSc Thesis. Georgia Institure of Technology, USA.

[Balakrishnan and Ranganathan 2012] Rangaswami Balakrishnan and Kanna Ranganathan. *A Textbook of Graph Theory*. Springer Science & Business Media, 2012.

[Brown 2016] Mark Brown. *The Legend of Zelda: The Wind Waker's dungeon design — Boss Keys*, 2016. https://www.youtube.com/watch?v=DuU6ojvBIic Retrieved 2016-08-15.

[Brown 2018] Mark Brown. *How I make a dependency graph*, 2018. https://www.patreon.com/posts/how-i-make-graph-20631617 Retrieved 2018-08-08.

[Coxen and Baylis 2012a] Tom Coxen and Jay Baylis. *Linking up Lenna and Metazelda*, 2012. https://bytten-studio.com/devlog//2012/04/14/sss-12-linking-up-lenna-and-metazelda/ Accessed 2020-07-18.

[Coxen and Baylis 2012b] Tom Coxen and Jay Baylis. *Lock and Key Puzzle Generation*, 2012. http://bytten-studio.com/devlog//2012/11/30/lock-and-key-puzzle-generation/ Accessed 2016-10-14.

[Coxen and Baylis 2020] Tom Coxen and Jay Baylis. *Linking up Lenna and Metazelda*, 2020. https://bytten-studio.com/devlog//2020/01/17/lennas-inception-is-out-now/ Accessed 2020-07-18.

[Dale and Lewis 2011] Nell Dale and John Lewis. *Computer Science Illuminated*. Jones and Bartlett Publishers, Inc., 2011.

[Dormans and Bakkes 2011] Joris Dormans and Sander Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011.

[Dormans 2009] Joris Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, pages 33–40, 2009.

[Dormans 2010] Joris Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 1. ACM, 2010.

[Entertainment Software Association 2017] Entertainment Software Association. *2017 Essential Facts About the Computer and Video Game Industry - Sales, Demographic, and Usage Data*, 2017.

[Entertainment Software Association 2019] Entertainment Software Association. *2019 Essential Facts About the Computer and Video Game Industry - Sales, Demographic, and Usage Data*, 2019.

[Heineman *et al.* 2008] George T Heineman, Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell*, 2008.

[Hendrikx *et al.* 2011] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural content generation for games:

A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2011.

[Johansen 2016] Rune Skovbo Johansen. *Working with puzzle design through state space visualization*, 2016. http://blog.runevision.com/2016/04/puzzlegraph-puzzle-state-space.html Accessed 2016-04-07.

[Lavender 2015] Becky Lavender. *The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games*, 2015. BSc(Hons) Thesis. University of Derby, UK.

[Lorc 2009] Lorc. *Padlock icon*, 2009. https://game-icons.net/1x1/lorc/padlock.html Accessed 2017-11-06.

[Martin 1991] John C Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill NY, 1991.

[Minarik 2011] Tyler Minarik. *PS3 Review: Prince of Persia Trilogy HD*, 2011. https://www.playstationlifestyle.net/2011/05/04/ps3-review-prince-of-persia-trilogy-hd/ Accessed 2017-11-29.

[Müller *et al.* 2006] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (TOG)*, 25(3):614–623, 2006.

[Odow 2015] Charlie Odow. *Los momentos mas epicos que nos dejo Ocarina of Time*, 2015. http://www.revistalevelup.com/2015/11/especial-ocarina-of-time-momentos-epicos.html Accessed 2017-11-29.

[Pereira *et al.* 2021] Leonardo Tortoro Pereira, Paulo Victor de Souza Prado, Rafael Miranda Lopes, and Claudio Fabiano Motta Toledo. Procedural generation of dungeons maps and locked-door missions through an evolutionary algorithm validated with players. *Expert Systems with Applications*, 180:115009, 2021.

[Prusinkiewicz and Lindenmayer 1990] Przemysław Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., 1990.

[Rees 2004] Gareth Rees. *The Puzzle Structure of Ocarina of Time*, 2004. https://garethrees.org/2004/12/01/ocarina-of-time/ Accessed 2016-09-08.

[Rouse III 2005] Richard Rouse III. *Game Design: Theory and Practice*. Wordware Publishing Inc., 2005.

[Rozenberg 1997] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

[sbed 2013] sbed. *Key icon*, 2013. https://game-icons.net/1x1/sbed/key.html Accessed 2017-11-06.

[Shaker *et al.* 2016] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. Constructive generation methods for dungeons and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 31–32. Springer, 2016.

[Square Enix Limited 2012] Square Enix Limited. *Tomb Raider 1+2+3*, 2012. https://www.gog.com/game/tomb_raider_123 Accessed 2017-11-29.

[van der Linden *et al.* 2014] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[Wolf 2012] Mark JP Wolf. *Encyclopedia of Video Games*, volume 2. ABC-CLIO (Publisher), 2012.

[Zelda Dungeon Team 2001] Zelda Dungeon Team. *Zelda Dungeon*, 2001. https://www.zeldadungeon.net Accessed 2016-10-14.

[Zelda Dungeon Team 2012] Zelda Dungeon Team. *The Legend of Zelda: Links Awakening DX*, 2012. https://www.zeldadungeon.net/wiki/File:Link_LA.png Accessed 2021-12-16.

[Zena 2013] Luis Zena. *The Legend of Zelda: Links Awakening DX*, 2013. http://obsoletegamer.com/the-legend-of-zelda-links-awakening-dx/ Accessed 2017-11-29.

YouTube Videos

[badassgamez 2013a] badassgamez. *Tomb Raider 1 Walkthrough [No Meds] — Level 1 - Caves*, 2013. https://www.youtube.com/watch?v=MF2BS3OBBDM Retrieved 2016-08-15.

[badassgamez 2013b] badassgamez. *Tomb Raider 1 Walkthrough [No Meds] — Level 2 - City of Vilcabamba*, 2013. https://www.youtube.com/watch?v=MXUvJuGC-Dc Retrieved 2016-08-15.

[badassgamez 2013c] badassgamez. *Tomb Raider 1 Walkthrough [No Meds] — Level 6 - Colosseum*, 2013. https://www.youtube.com/watch?v=z8AArtjGDWo Retrieved 2017-01-15.

[badassgamez 2014a] badassgamez. *Tomb Raider 1 Walkthrough [No Meds] — Level 10 - City of Khamoon*, 2014. https://www.youtube.com/watch?v=cFPPEl0HY-w Retrieved 2016-08-15.

[badassgamez 2014b] badassgamez. *Tomb Raider 1 Walkthrough [No Meds] — Level 14 - Atlantis*, 2014. https://www.youtube.com/watch?v=paxEou-o_5g Retrieved 2016-08-15.

[Brown 2014] Mark Brown. *Sequence Breaking with Toki Tori 2 — Game Maker's Toolkit*, 2014. https://www.youtube.com/watch?v=084BUNlI7Gk Retrieved 2016-09-08.

[Brown 2016a] Mark Brown. *The Legend of Zelda: Link's Awakening's dungeon design — Boss Keys*, 2016. https://www.youtube.com/watch?v=AezAN2RcO8Y Retrieved 2016-08-15.

[Brown 2016b] Mark Brown. *The Legend of Zelda: Ocarina of Time's dungeon design — Boss Keys*, 2016. https://www.youtube.com/watch?v=6LO8Z1DkDqc Retrieved 2016-08-15.

[Brown 2017] Mark Brown. *The Legend of Zelda: Skyward Sword's dungeon design — Boss Keys*, 2017. https://www.youtube.com/watch?v=1ewa26q76fE Retrieved 2016-08-15.

[Brown 2021] Mark Brown. *Boss Keys Playlist*, 2021. https://www.youtube.com/playlist?list=PLc38fcMFcV_ul4D6OChdWhsNsYY3NA5B2 etrieved 2021-02-09.

[HardlyAstounding 2012] HardlyAstounding. *Prince of Persia [Sands of Time] FULL WALKTHROUGH*, 2012. https://www.youtube.com/watch?v=0sSkt1ZXykM Retrieved 2016-07-14.

Video games

[Acornsoft 1984] Acornsoft. *Elite*. BBC Micro, 1984.

[Bytten Studio 2020] Bytten Studio. *Lenna's Inception*. PC, 2020.

[Core Design 1996] Core Design. *Tomb Raider*. PC, 1996.

[Nintendo 1998a] Nintendo. *The Legend of Zelda: Link's Awakening DX*. Gameboy Color, 1998.

[Nintendo 1998b] Nintendo. *The Legend of Zelda: Ocarina of Time*. Nintendo 64, 1998.

[Ubisoft Montreal 2003] Ubisoft Montreal. *Prince of Persia: The Sands of Time*. PC, 2003.

Software

[Ciskey 2016] Joe Ciskey. *Pygraph*, 2016.

[Graphviz Team 2016] Graphviz Team. *Graphviz*, 2016.

[Graphviz Team 2017] Graphviz Team. *DOT Language Documentation*, 2017. https://graphviz.org/doc/info/lang.html Accessed 2021-11-28.

[Project64 Team 2005] Project64 Team. *Project64 version 1.6*, 2005.

[VBA Team 2004] VBA Team. *VisualBoyAdvance version 1.8.0*, 2004.

[yWorks 2000] yWorks. *yEd*, 2000.

Appendix A

JSON

Below we show the JSON representation for the *Prince of Persia: The Sands of Time* graph grammar as described in Section 5.2.1.

Listing A.1: JSON grammar for Prince of Persia: The Sands of Time

```
1  {
2    "game": "Prince of Persia: The Sands of Time",
3    "productions": [
4      {
5        "ruleName": "S1",
6        "ruleDescription": "Start",
7        "leftSide": "start",
8        "nodes": [
9          "begin", "traversal", "puzzleSection",
               ↪ "traversal", "end"
10       ],
11       "edges": [
12         [0,1],
13         [1,2],
14         [2,3],
15         [3,4]
16       ],
17       "connectingRules": []
18     },
19     {
20       "ruleName": "PS1",
21       "ruleDescription": "Puzzle Extension",
```

```
22        "leftSide": "puzzleSection",
23        "nodes": [
24          "puzzleSection", "traversal", "puzzleSection"
25        ],
26        "edges": [
27          [0,1], [1,2]
28        ],
29        "connectingRules": [
30          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 0},
31          {"neighborType": "traversal",
              ↪ "neighborDirection": "out",
              ↪ "connectionDirection": "out", "nodeID": 2}
32        ]
33      },
34      {
35        "ruleName": "PS2",
36        "ruleDescription": "One-way gate",
37        "leftSide": "puzzleSection",
38        "nodes": [
39          "keySearch", "lock"
40        ],
41        "edges": [
42          [0,1]
43        ],
44        "connectingRules": [
45          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 0},
46          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 1},
47          {"neighborType": "traversal",
              ↪ "neighborDirection": "out",
              ↪ "connectionDirection": "out", "nodeID": 1}
48        ]
49      },
50      {
51        "ruleName": "PS3",
```

```
52        "ruleDescription": "Two-way gate",
53        "leftSide": "puzzleSection",
54        "nodes": [
55          "keySearch", "lock"
56        ],
57        "edges": [
58          [0,1]
59        ],
60        "connectingRules": [
61          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 0},
62          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 1},
63          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "out", "nodeID": 1}
              ↪ ,
64          {"neighborType": "traversal",
              ↪ "neighborDirection": "out",
              ↪ "connectionDirection": "in", "nodeID": 1},
65          {"neighborType": "traversal",
              ↪ "neighborDirection": "out",
              ↪ "connectionDirection": "out", "nodeID": 1}
66        ]
67      },
68      {
69        "ruleName": "KS1",
70        "ruleDescription": "Basic key",
71        "leftSide": "keySearch",
72        "nodes": [
73          "key"
74        ],
75        "edges": [],
76        "connectingRules": [
77          {"neighborType": "traversal",
              ↪ "neighborDirection": "in",
              ↪ "connectionDirection": "in", "nodeID": 0},
```

```
78          {"neighborType": "lock", "neighborDirection":
               ↪ "out", "connectionDirection": "out",
               ↪ "nodeID": 0}
79        ]
80      },
81      {
82        "ruleName": "KS2",
83        "ruleDescription": "Stacking key search",
84        "leftSide": "keySearch",
85        "nodes": [
86          "keySearch", "lock", "traversal", "key"
87        ],
88        "edges": [
89          [0,1],
90          [1,2],
91          [2,1],
92          [2,3]
93        ],
94        "connectingRules": [
95          {"neighborType": "traversal",
               ↪ "neighborDirection": "in",
               ↪ "connectionDirection": "in", "nodeID": 0},
96          {"neighborType": "traversal",
               ↪ "neighborDirection": "in",
               ↪ "connectionDirection": "in", "nodeID": 1},
97          {"neighborType": "traversal",
               ↪ "neighborDirection": "in",
               ↪ "connectionDirection": "out", "nodeID": 1}
               ↪ ,
98          {"neighborType": "lock", "neighborDirection":
               ↪ "out", "connectionDirection": "out",
               ↪ "nodeID": 3}
99        ]
100     },
101     {
102       "ruleName": "KS3",
103       "ruleDescription": "Multiple keys",
104       "leftSide": "keySearch",
105       "nodes": [
106         "keySearch", "keySearch"
```

```
107         ],
108         "edges": [],
109         "connectingRules": [
110           {"neighborType": "traversal",
                ↪ "neighborDirection": "in",
                ↪ "connectionDirection": "in", "nodeID": 0},
111           {"neighborType": "traversal",
                ↪ "neighborDirection": "in",
                ↪ "connectionDirection": "in", "nodeID": 1},
112           {"neighborType": "lock", "neighborDirection":
                ↪ "out", "connectionDirection": "out",
                ↪ "nodeID": 0},
113           {"neighborType": "lock", "neighborDirection":
                ↪ "out", "connectionDirection": "out",
                ↪ "nodeID": 1}
114         ]
115       }
116     ]
117
118 }
```

Appendix B

Full Derivations

B.1 Prince of Persia: The Sands of Time

This appendix contains the full derivations of all the levels that we have analysed in this dissertation. We discuss the context of each level in Chapter 5.

B.1.1 I'll Meet You at the Baths

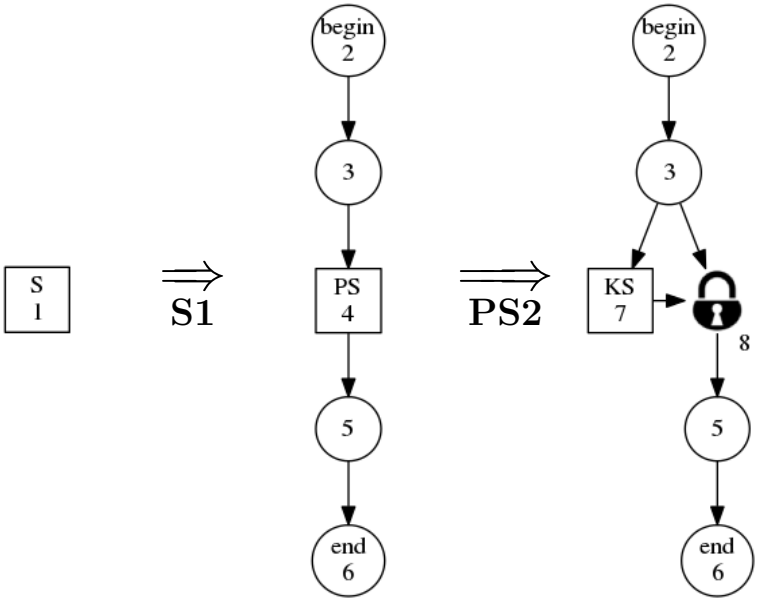


Figure B.1: Part 1 of *I'll meet you at the Baths* derivation

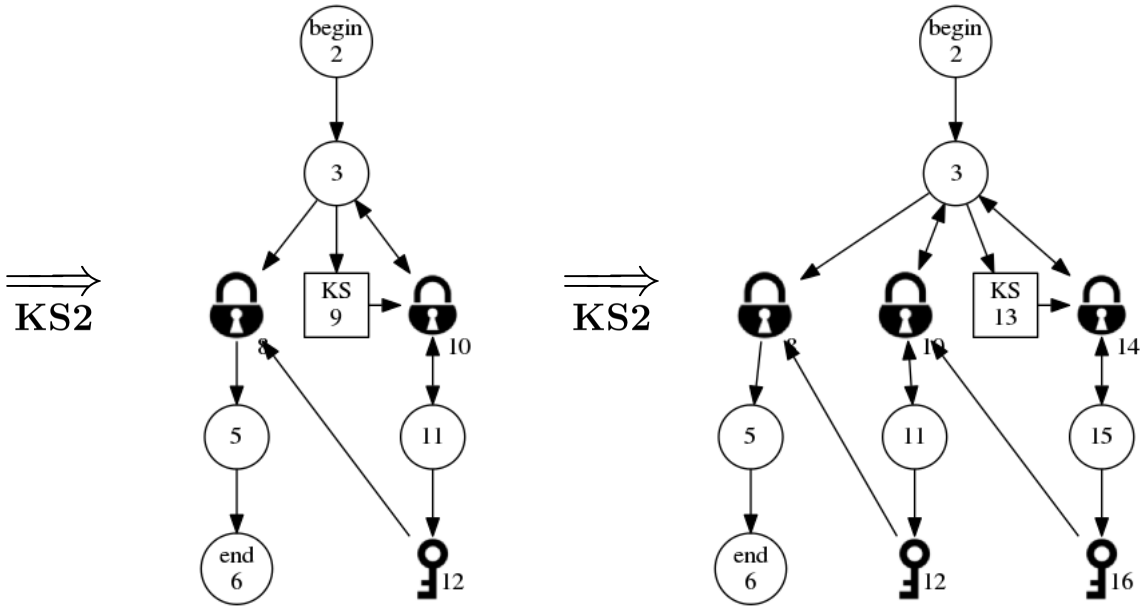


Figure B.1: Part 2 of *I'll meet you at the Baths* derivation

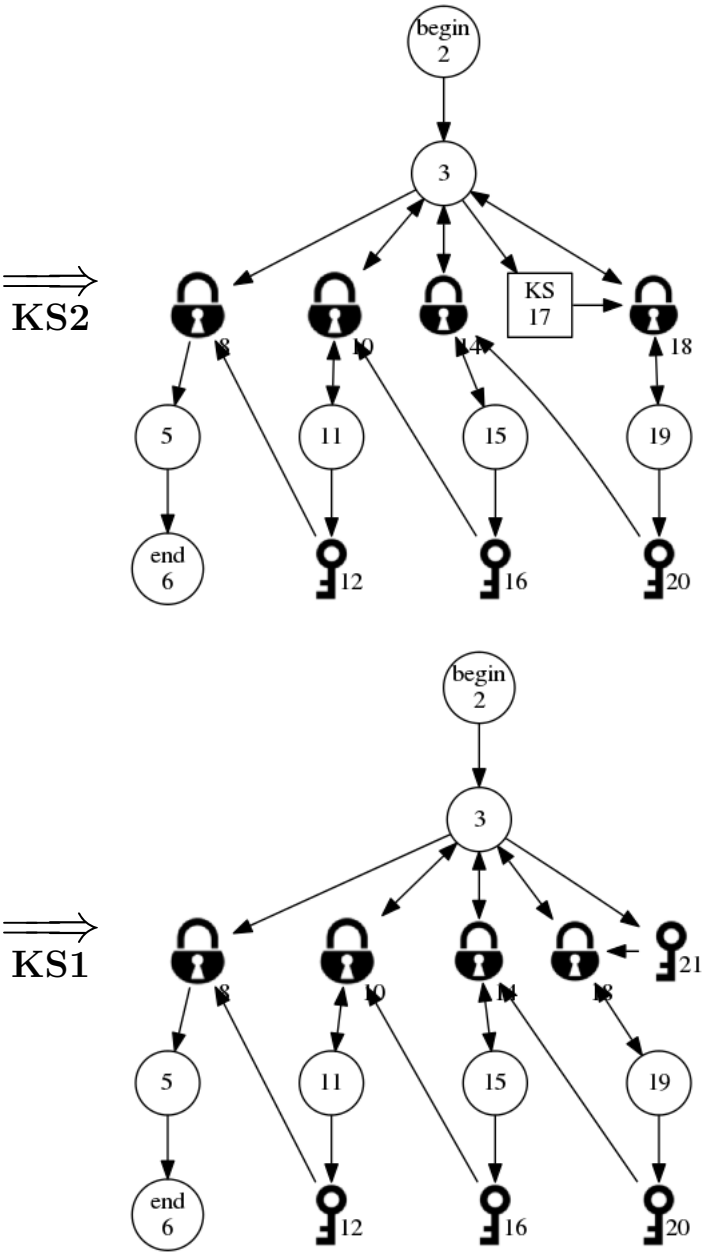


Figure B.1: Part 3 of *I'll meet you at the Baths* derivation

B.1.2 Hall of Learning Courtyards

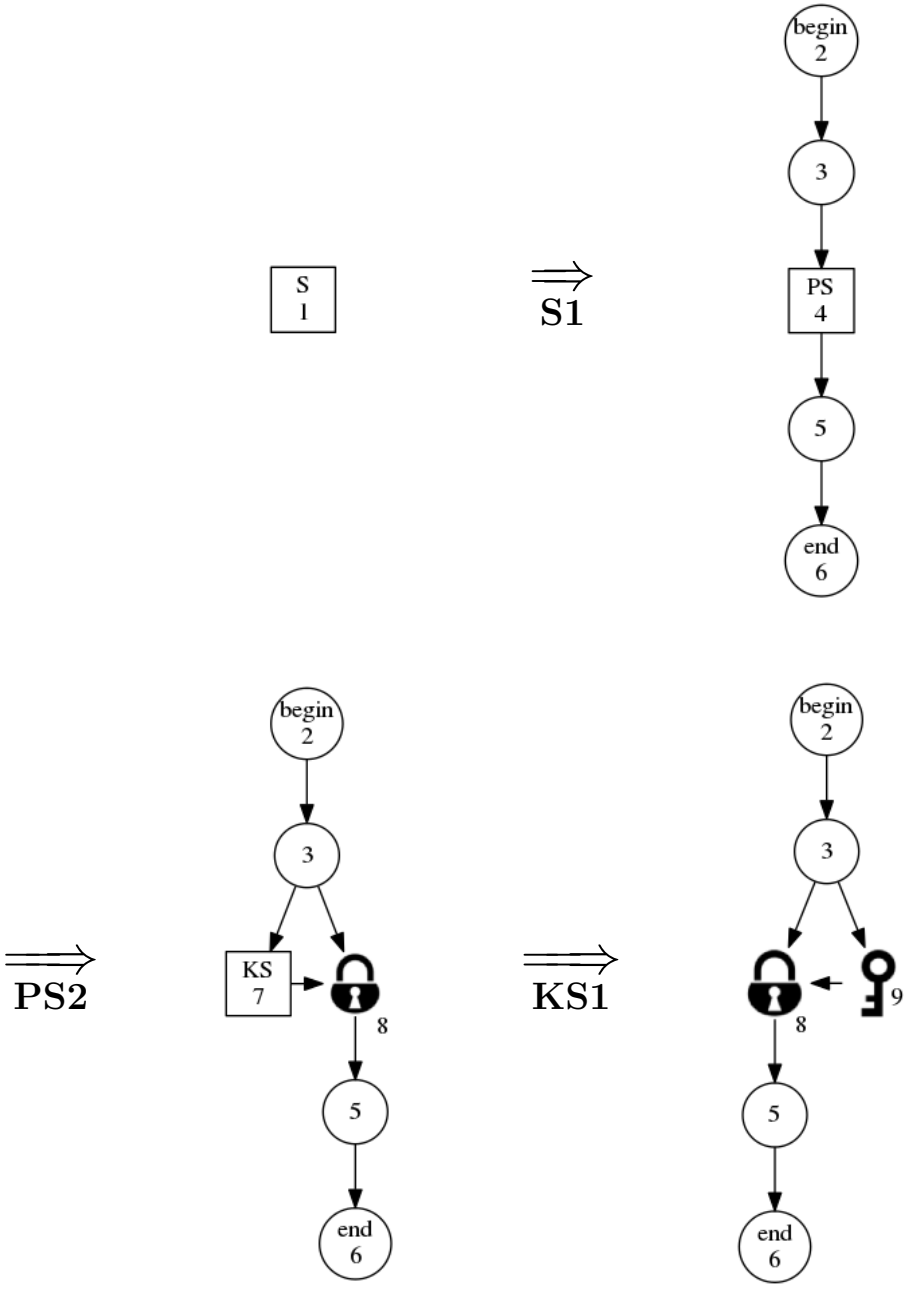


Figure B.2: *Hall of Learning Courtyards* derivation

B.1.3 Out of the Well

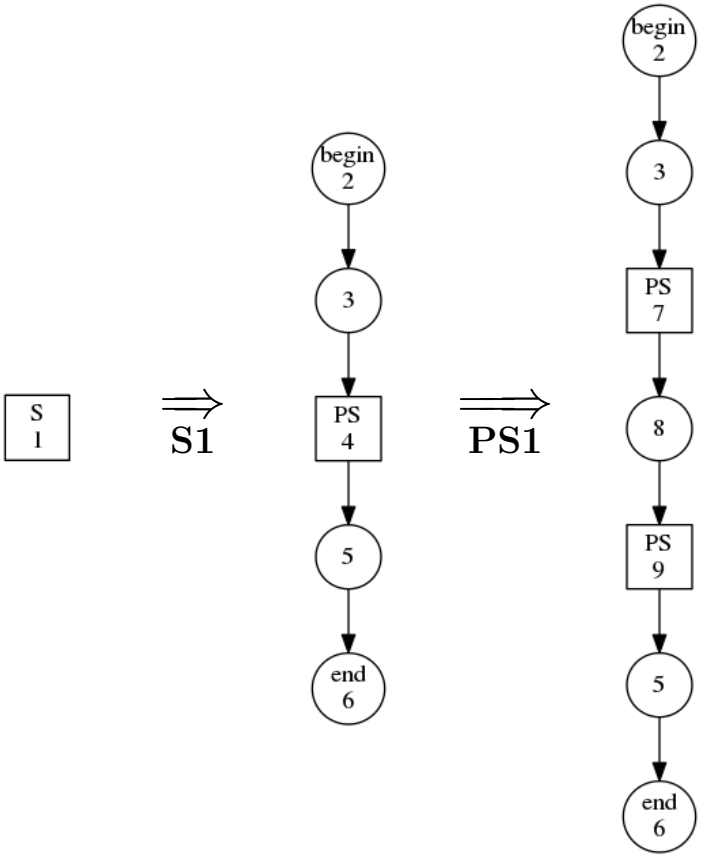


Figure B.3: Part 1 of *Out of the Well* derivation

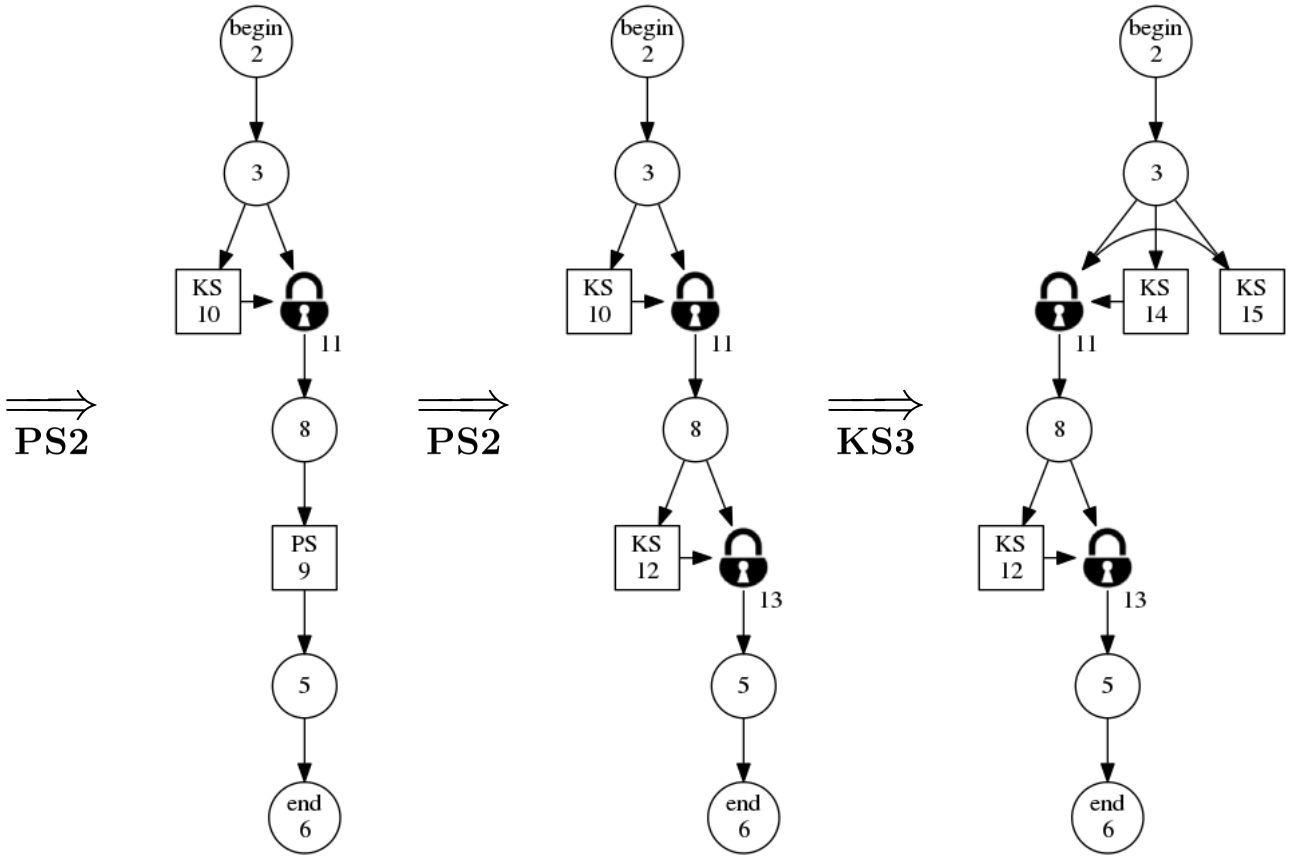


Figure B.3: Part 2 of *Out of the Well* derivation

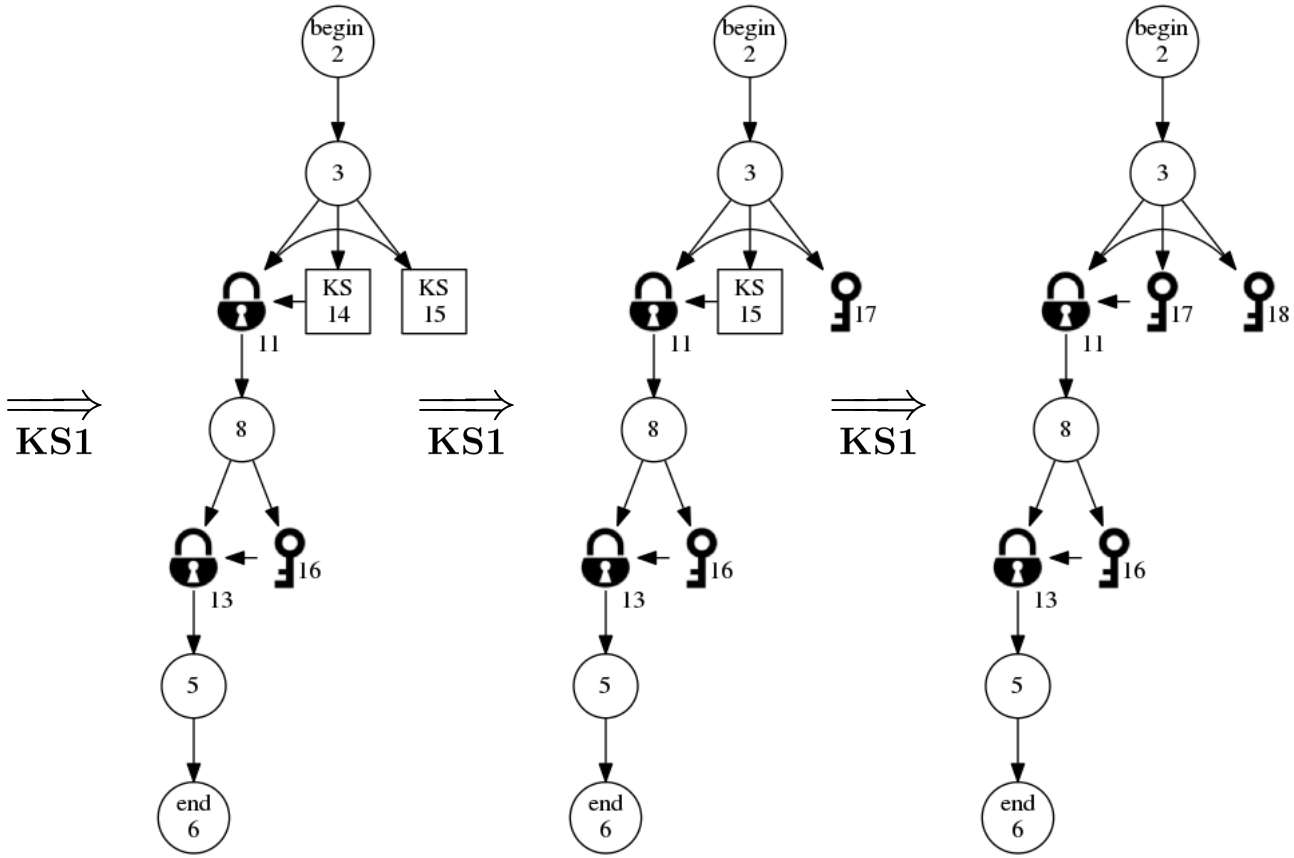


Figure B.3: Part 3 of *Out of the Well* derivation

B.1.4 An Underground Reservoir & Atop a Bird Cage

The two levels *An Underground Reservoir* and *Atop a Bird Cage* have identical puzzle graphs.

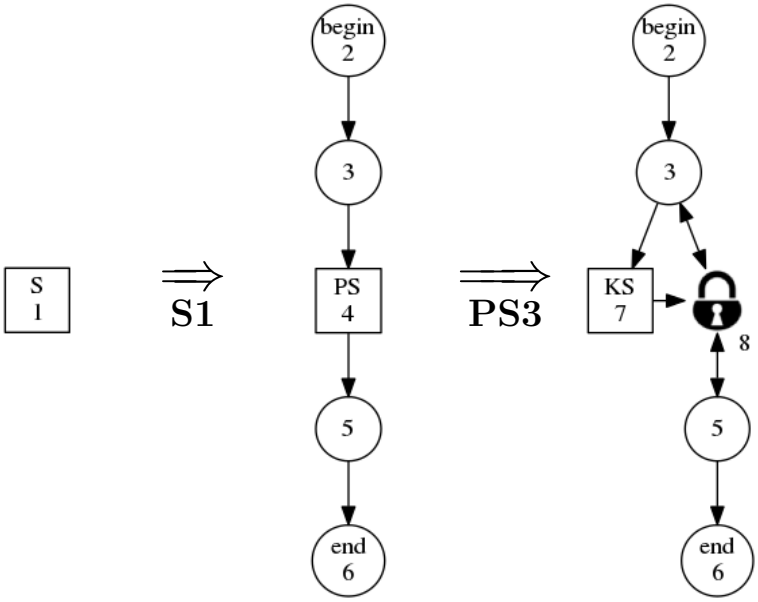


Figure B.4: Part 1 of *An Underground Reservoir* and *Atop a Bird Cage* derivation

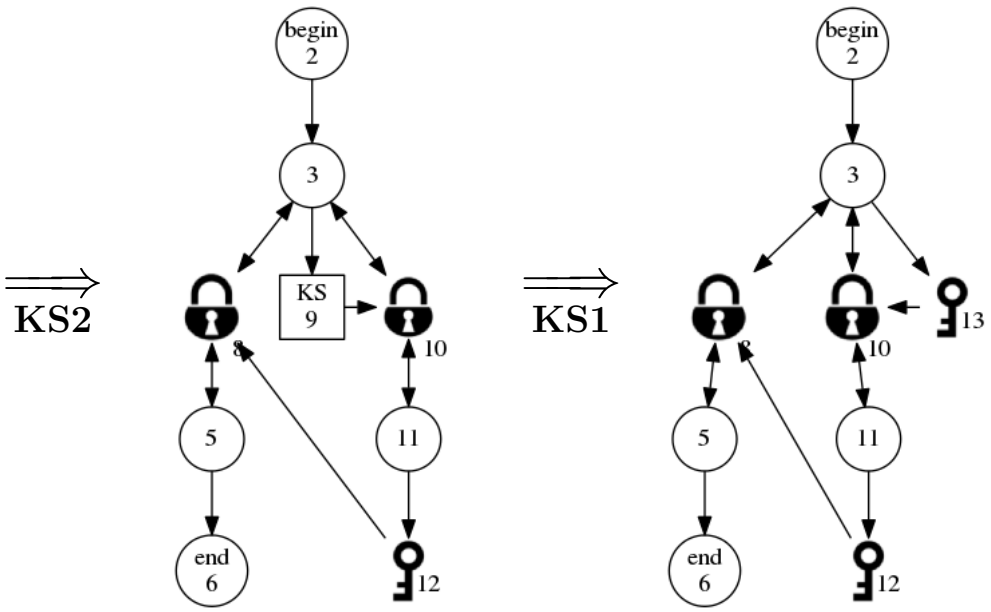


Figure B.4: Part 2 of *An Underground Reservoir* and *Atop a Bird Cage* derivation

B.2 Tomb Raider

B.2.1 Level 1: Caves

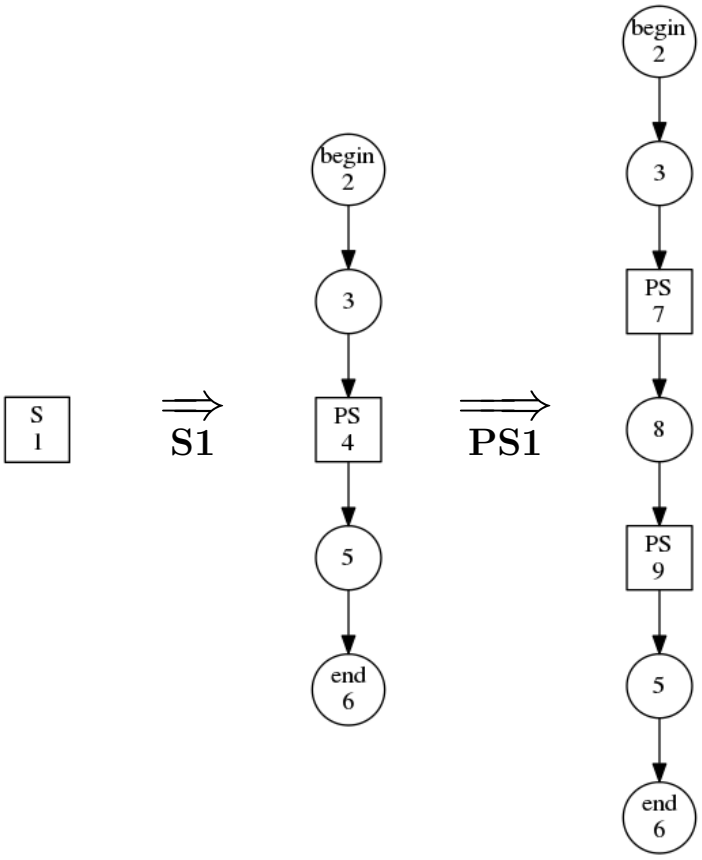


Figure B.5: Part 1 of *Caves* derivation

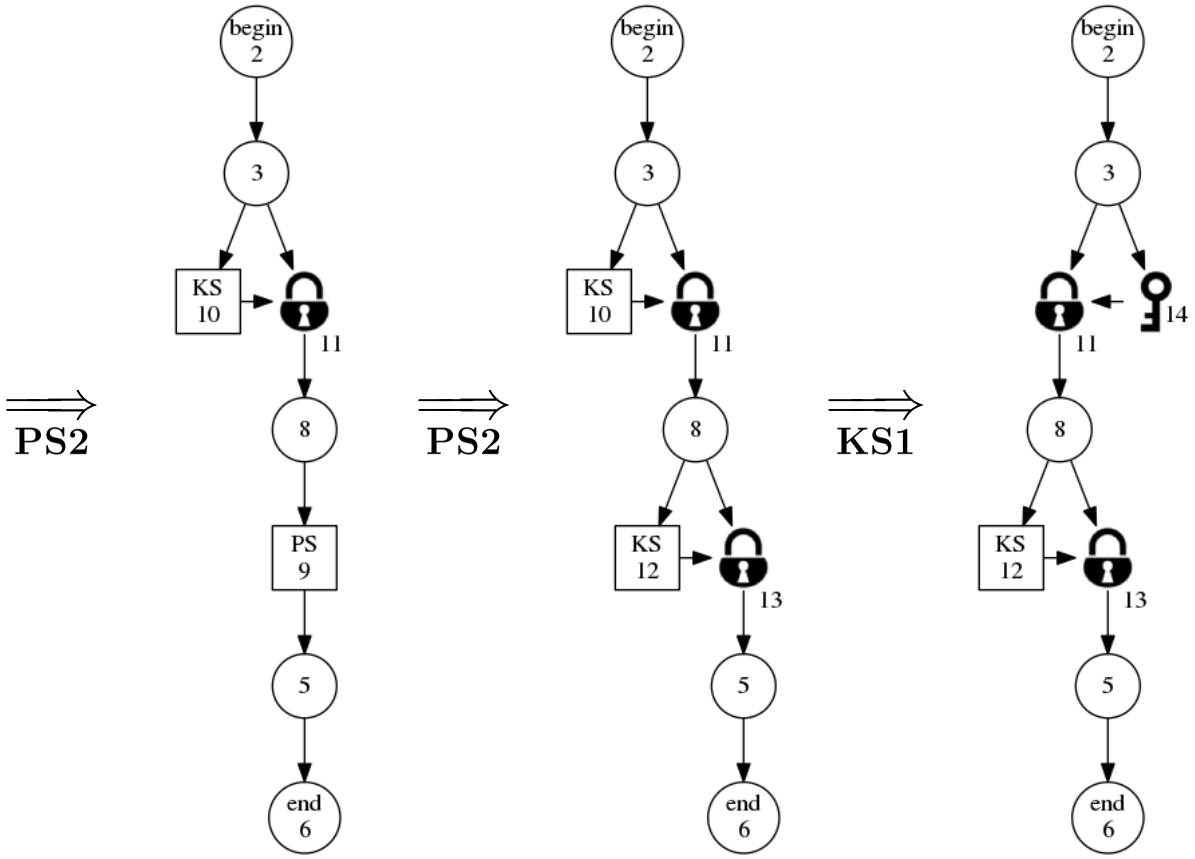


Figure B.5: Part 2 of *Caves* derivation

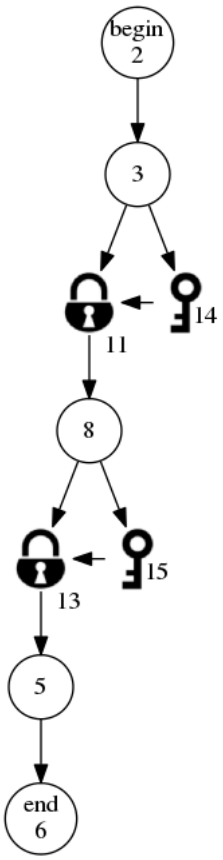


Figure B.5: Part 3 of *Caves* derivation

B.2.2 Level 2: City of Vilcabamba

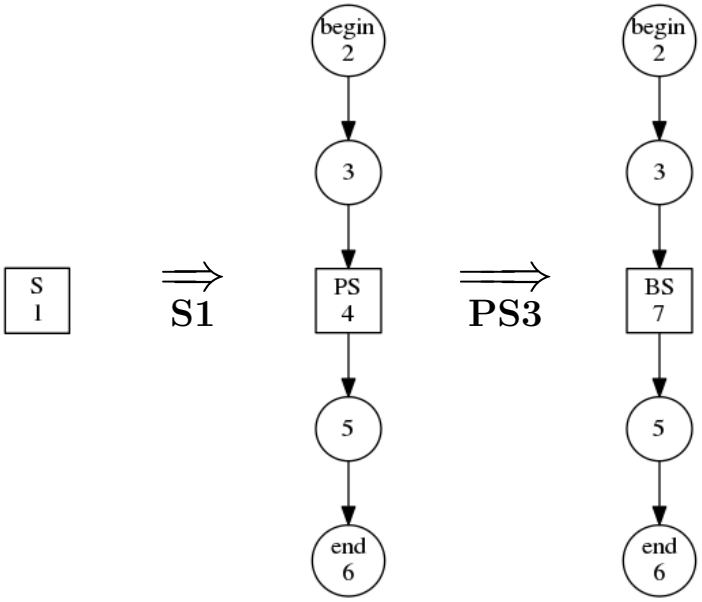


Figure B.6: Part 1 of *City of Vilcabamba* derivation

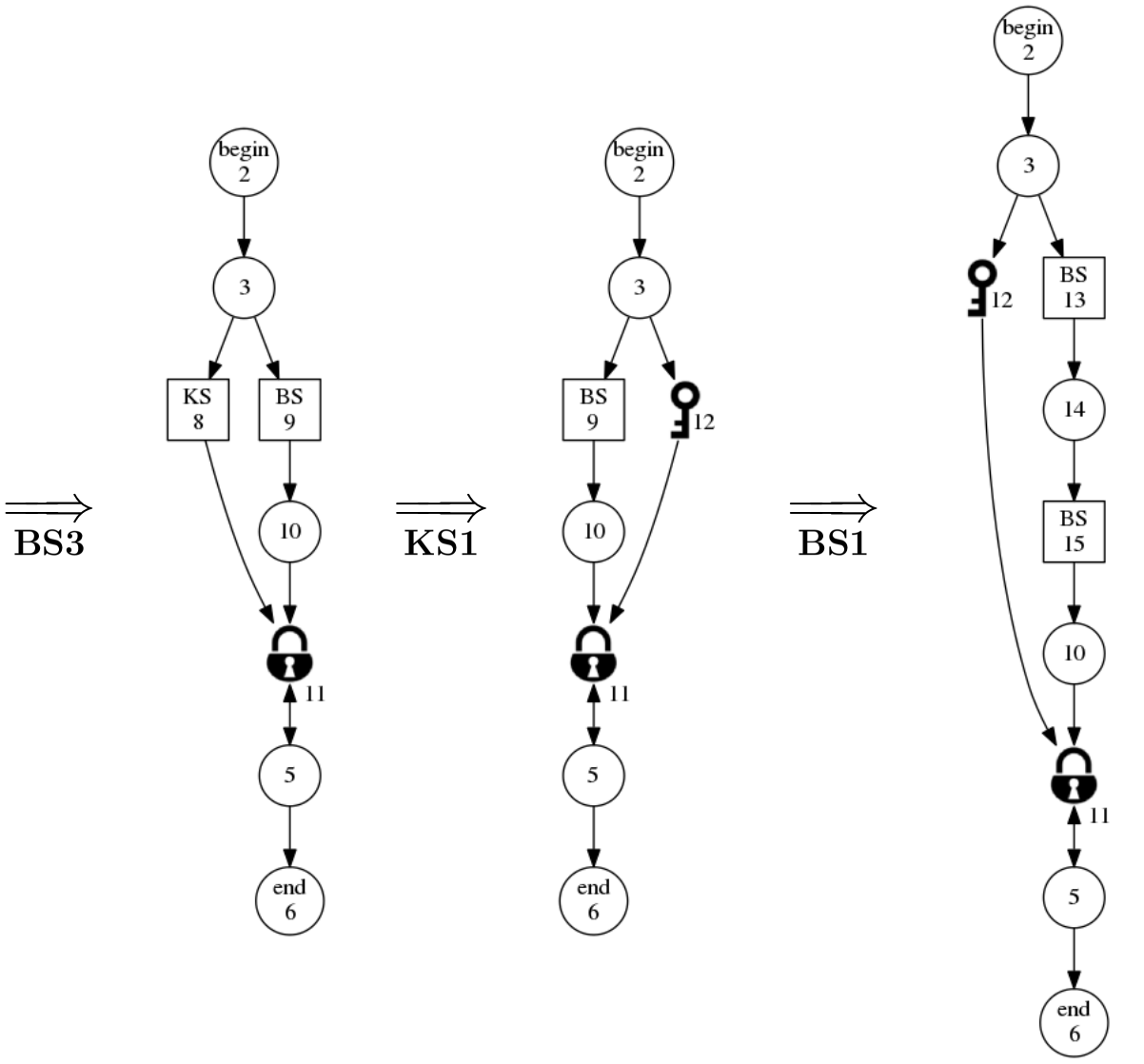


Figure B.6: Part 2 of *City of Vilcabamba* derivation

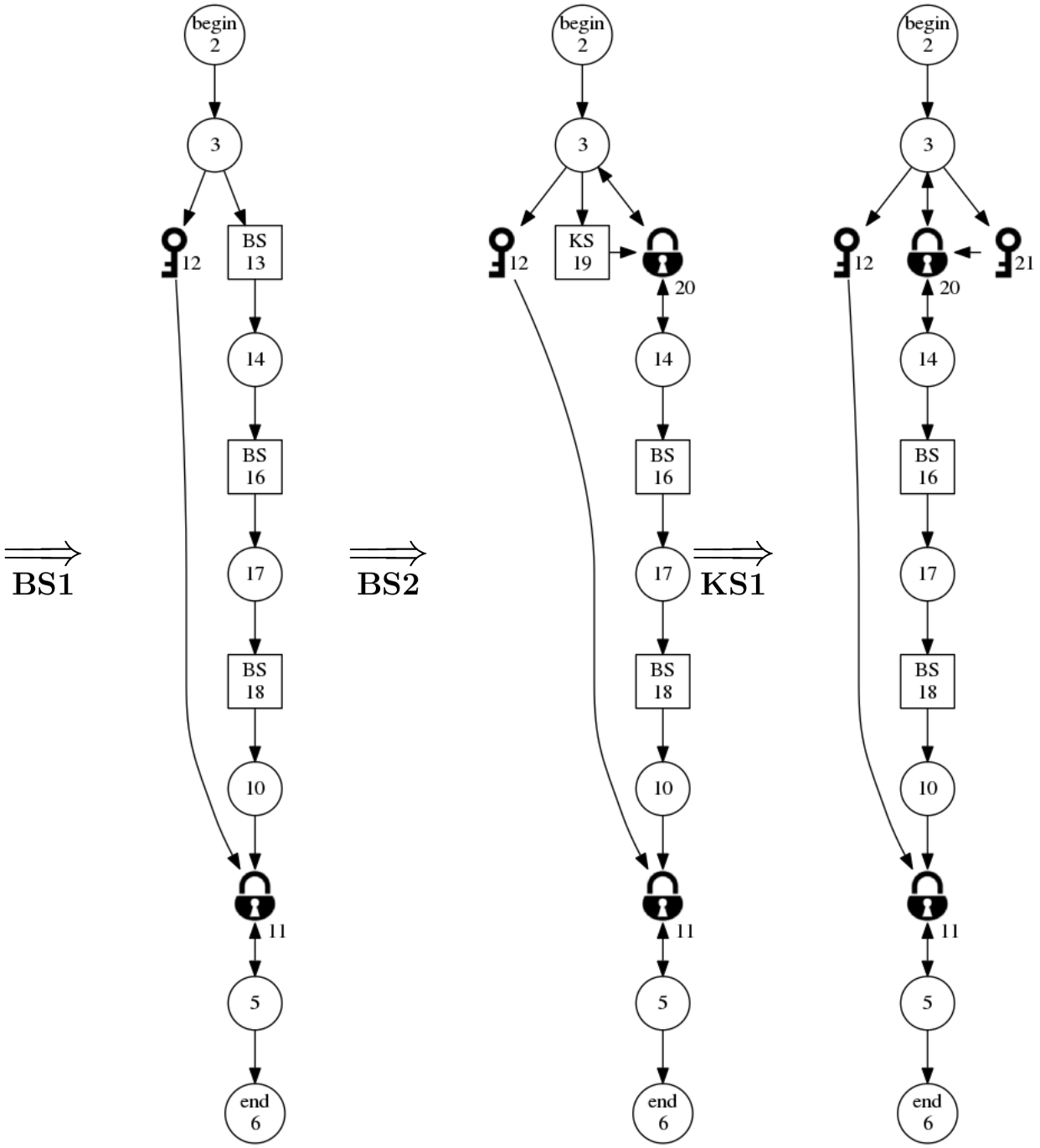


Figure B.6: Part 3 of *City of Vilcabamba* derivation

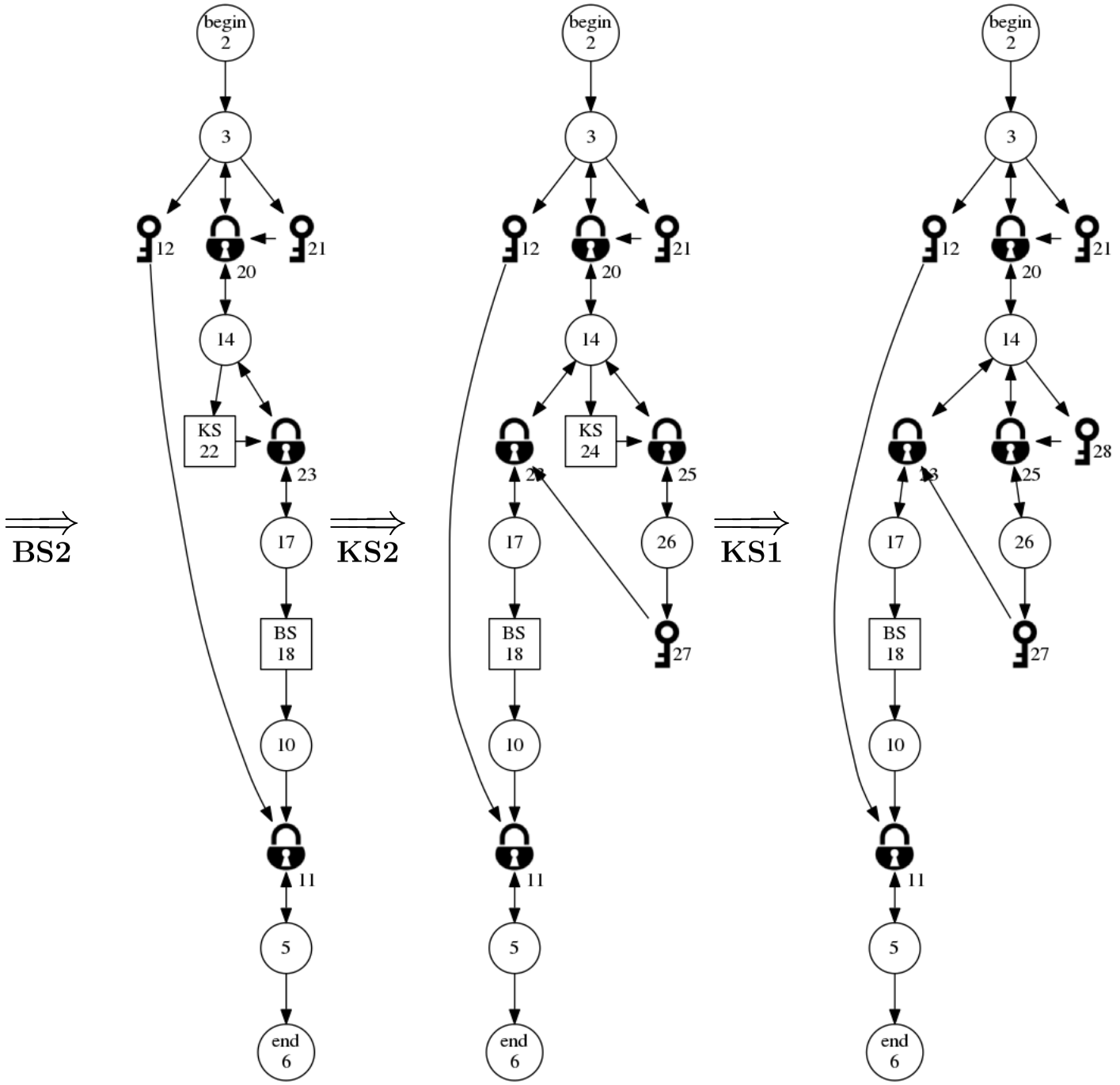


Figure B.6: Part 4 of *City of Vilcabamba* derivation

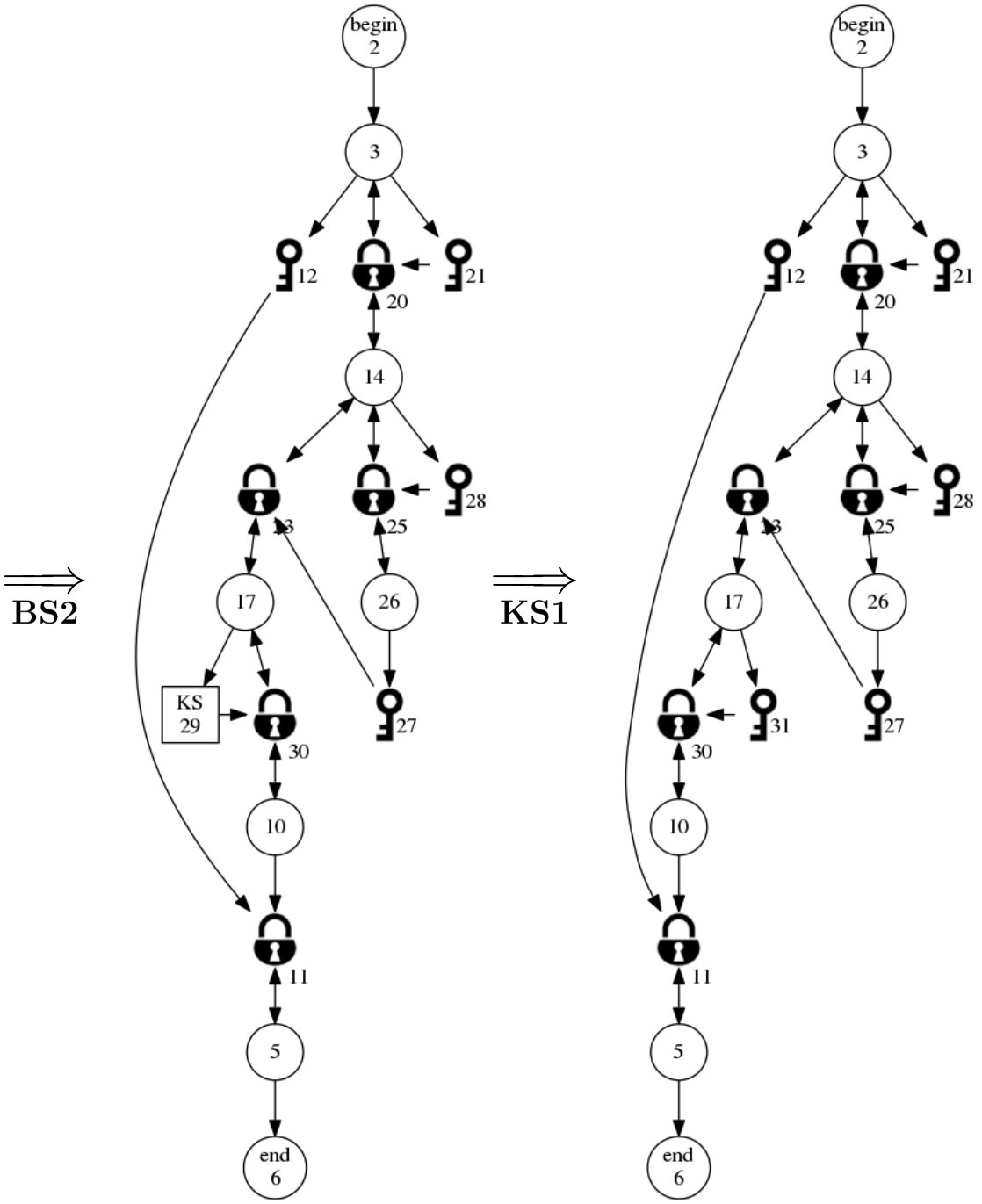


Figure B.6: Part 5 of *City of Vilcabamba* derivation

B.2.3 Level 6: Colosseum

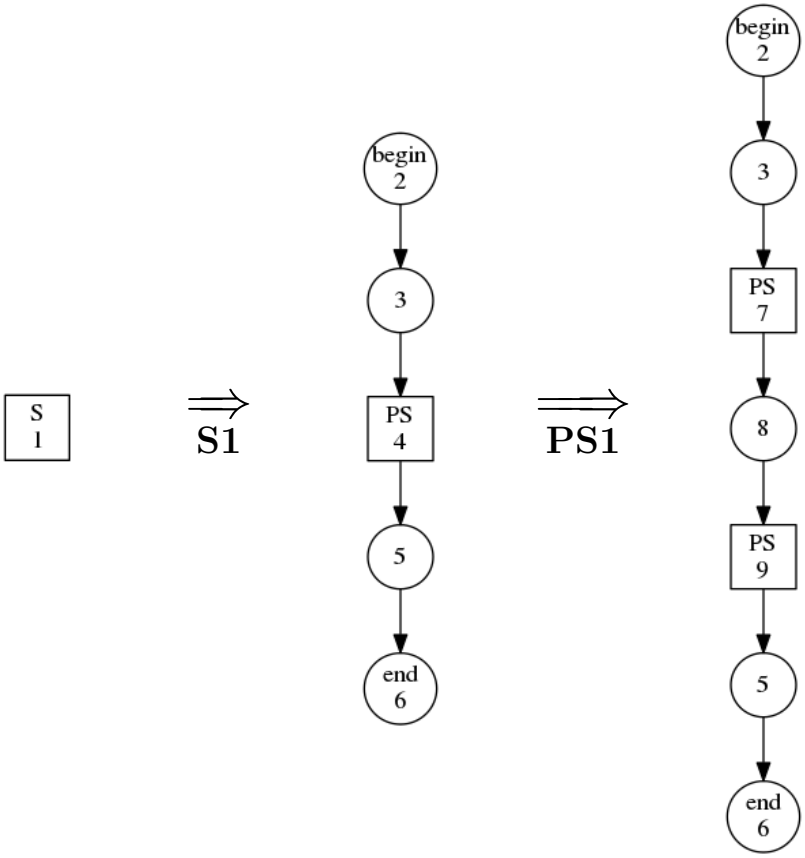


Figure B.7: Part 1 of *Colosseum* derivation

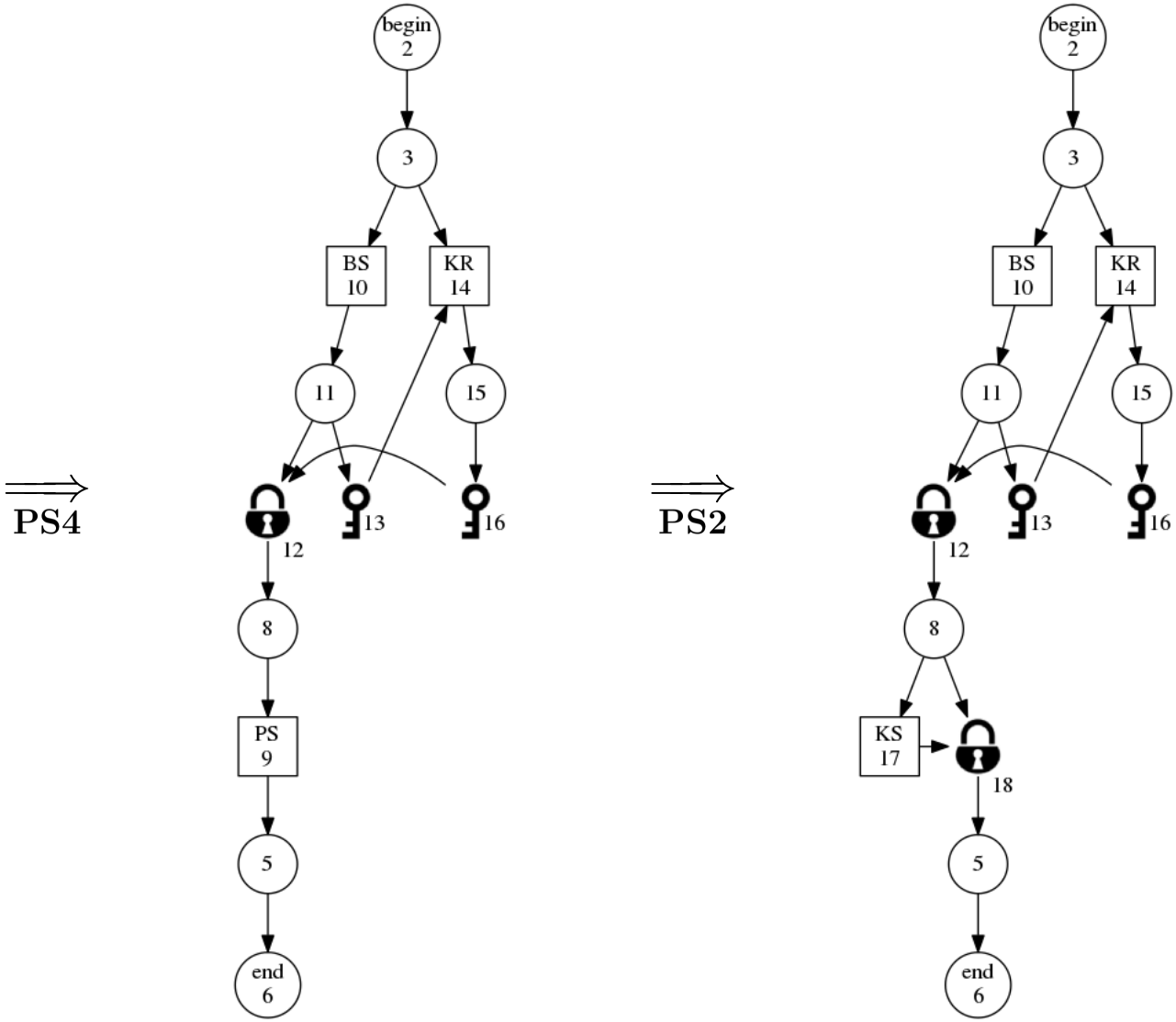


Figure B.7: Part 2 of *Colosseum* derivation

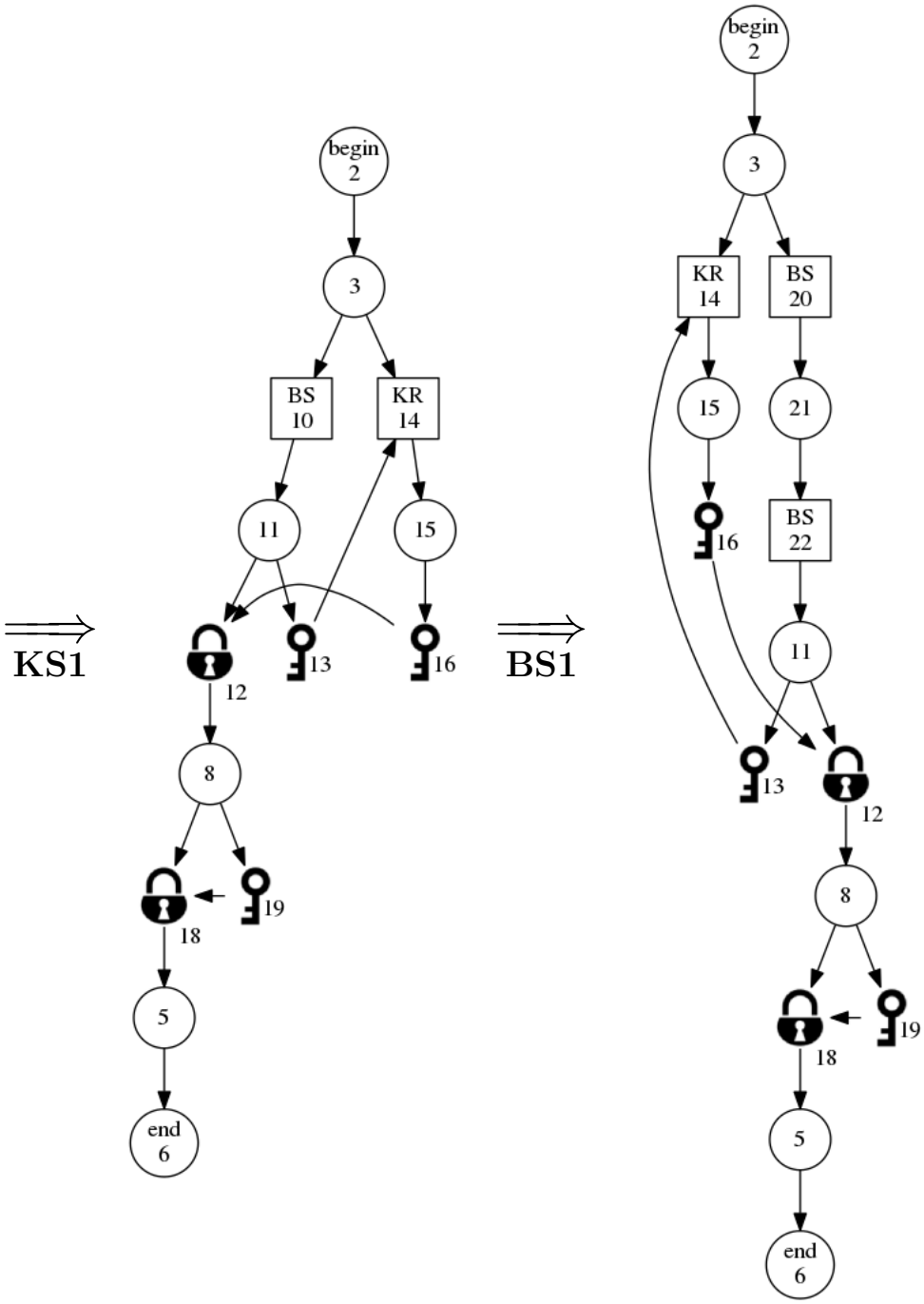


Figure B.7: Part 3 of *Colosseum* derivation

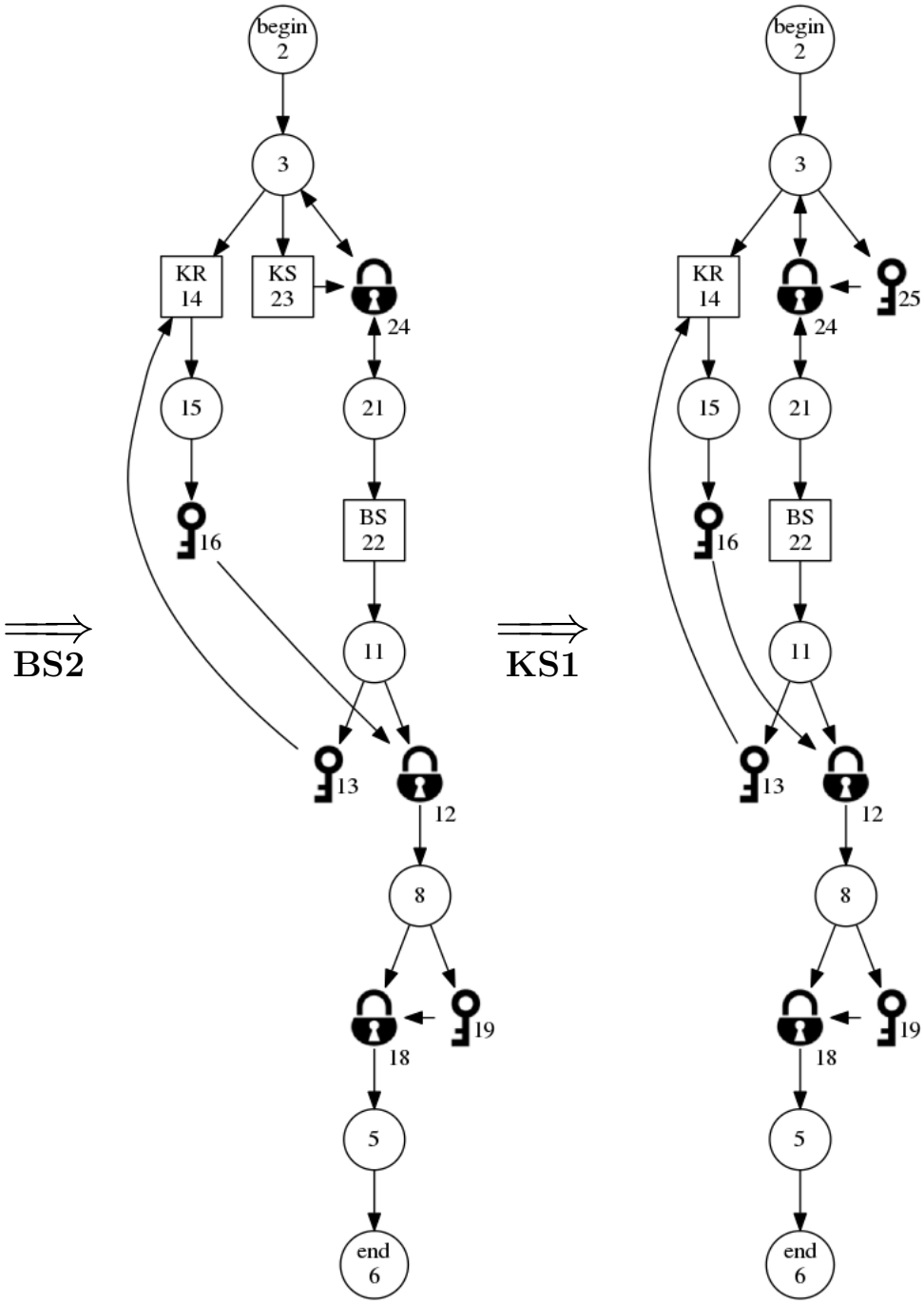


Figure B.7: Part 4 of *Colosseum* derivation

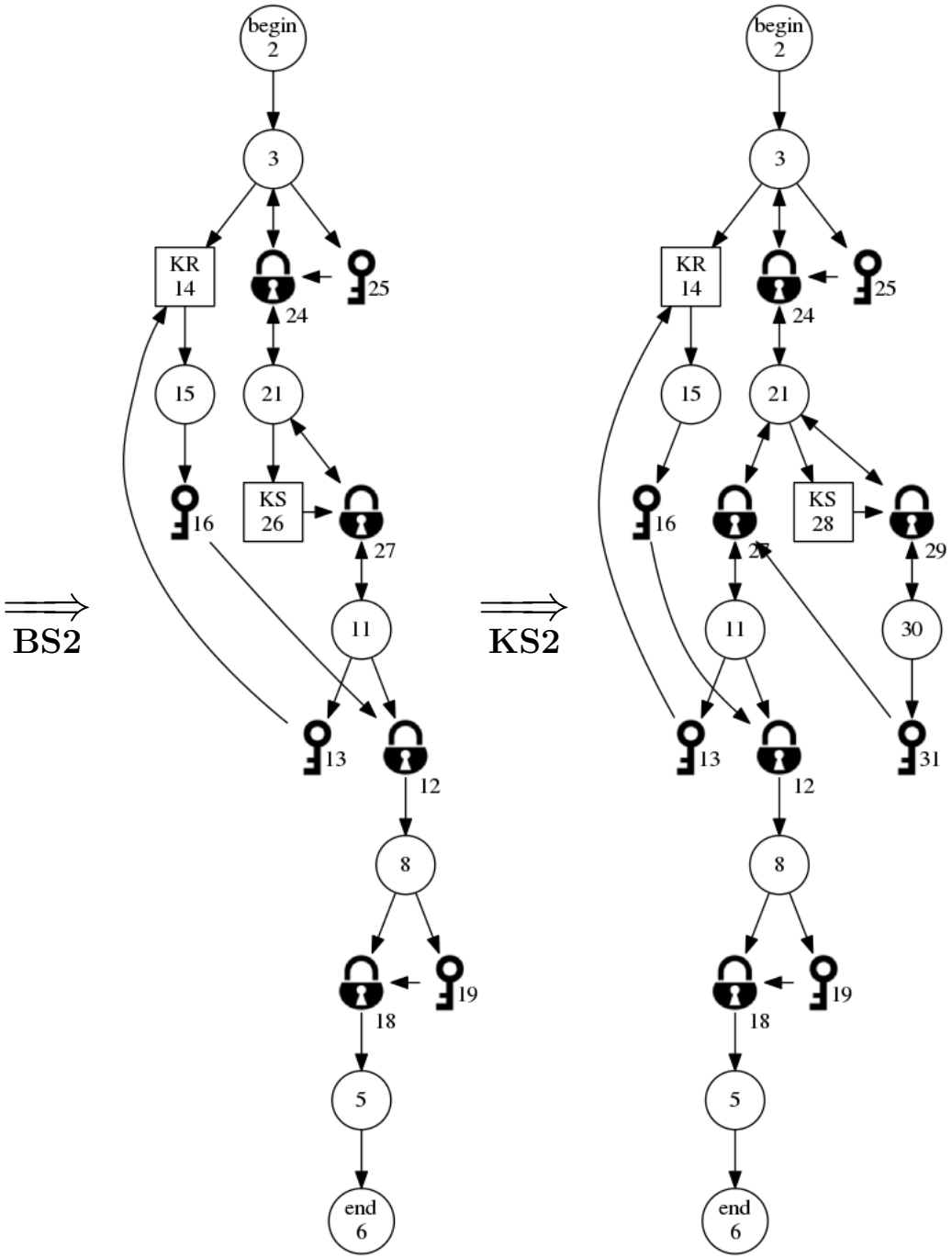


Figure B.7: Part 5 of *Colosseum* derivation

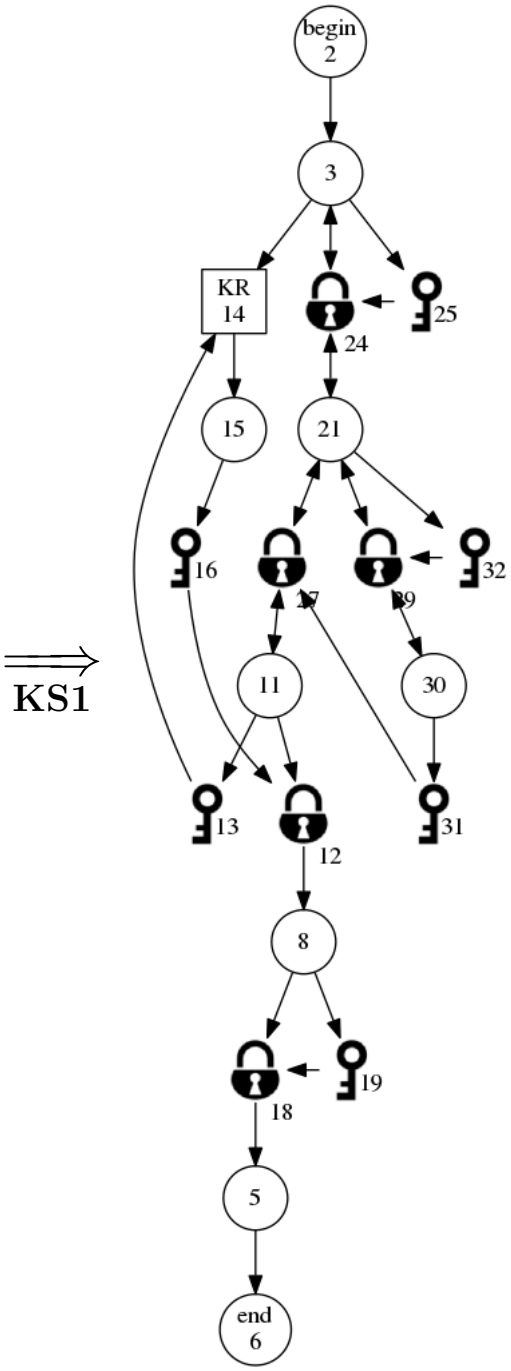


Figure B.7: Part 6 of *Colosseum* derivation

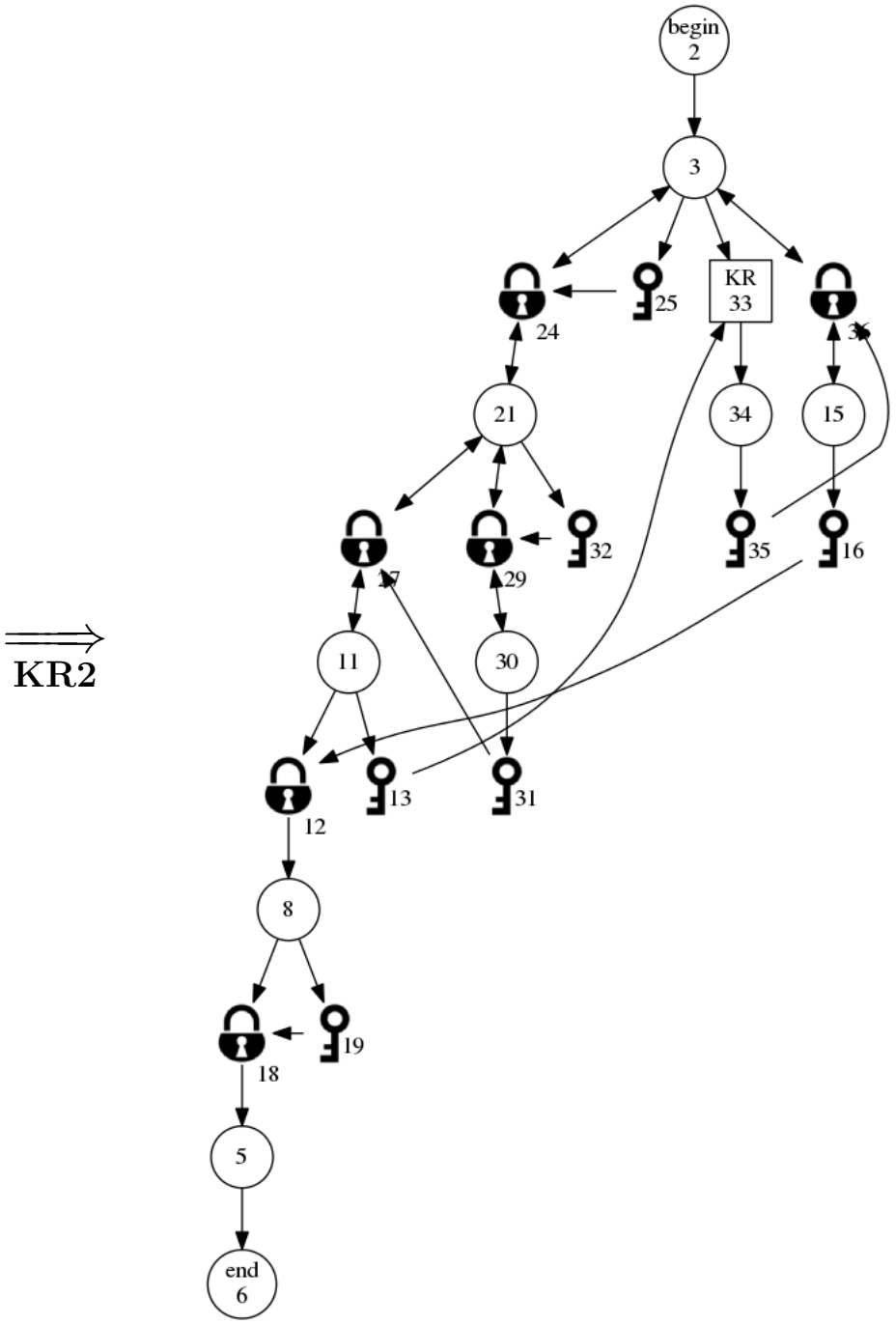


Figure B.7: Part 7 of *Colosseum* derivation

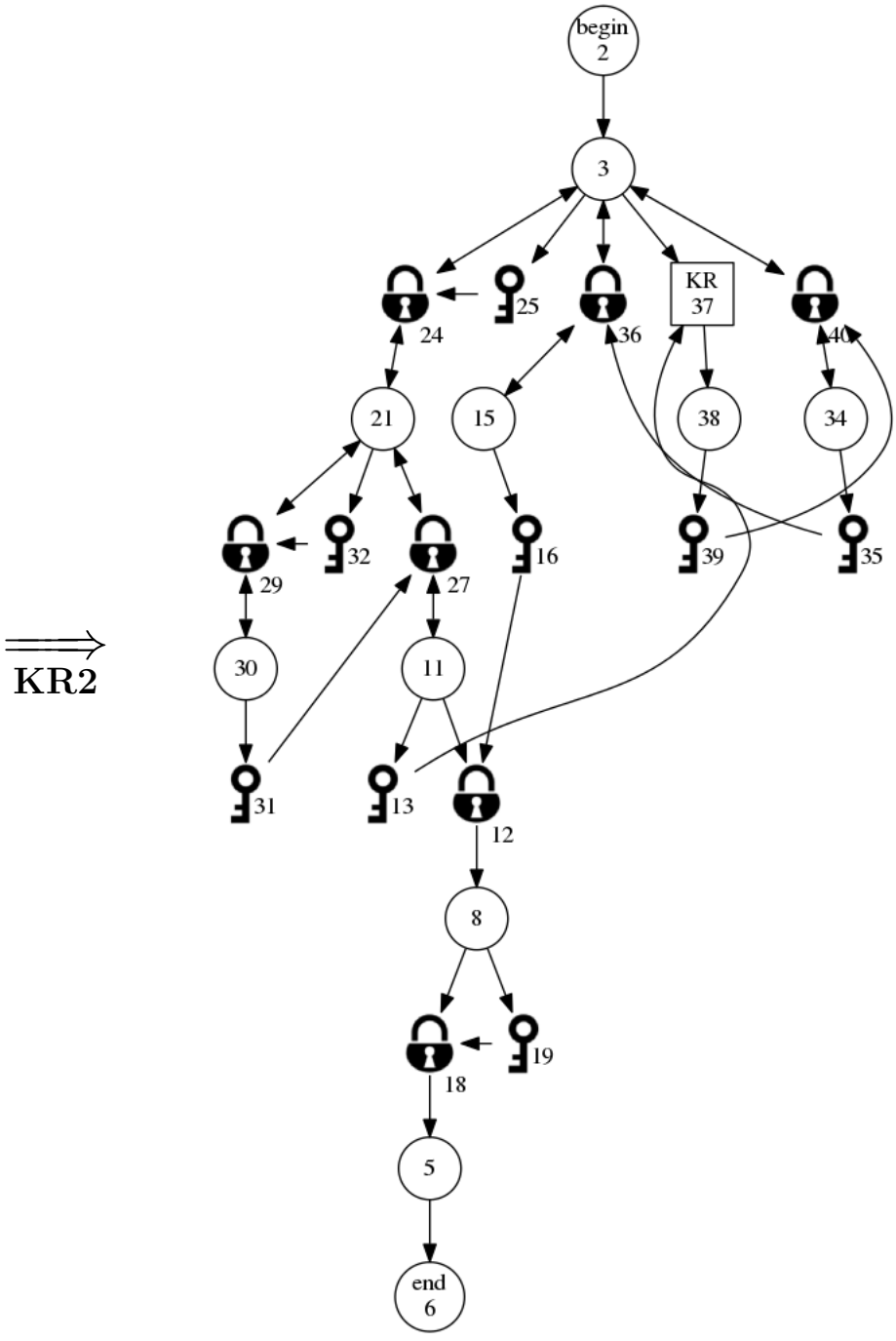


Figure B.7: Part 8 of *Colosseum* derivation

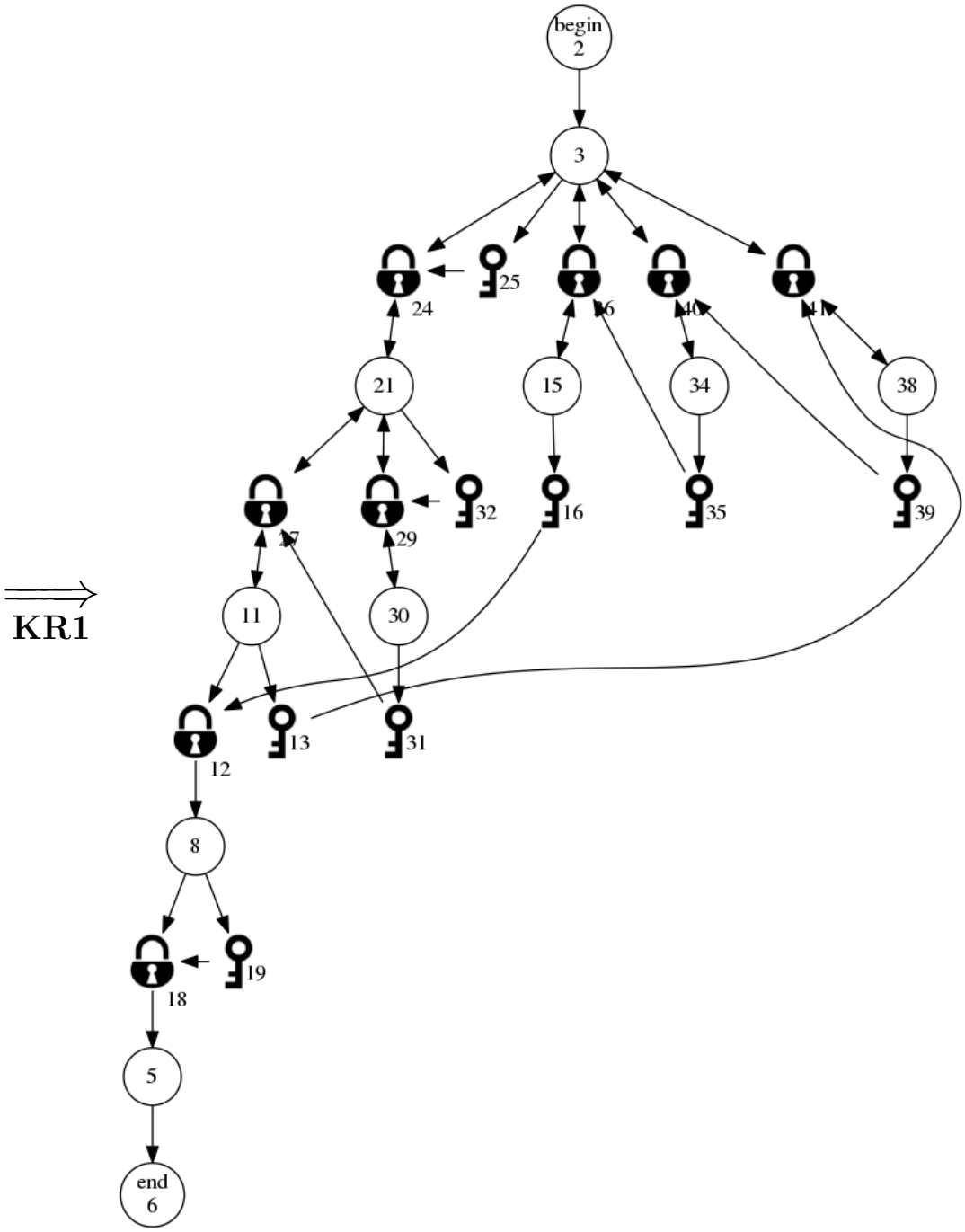


Figure B.7: Part 9 of *Colosseum* derivation

B.2.4 Level 10: City of Khamoon

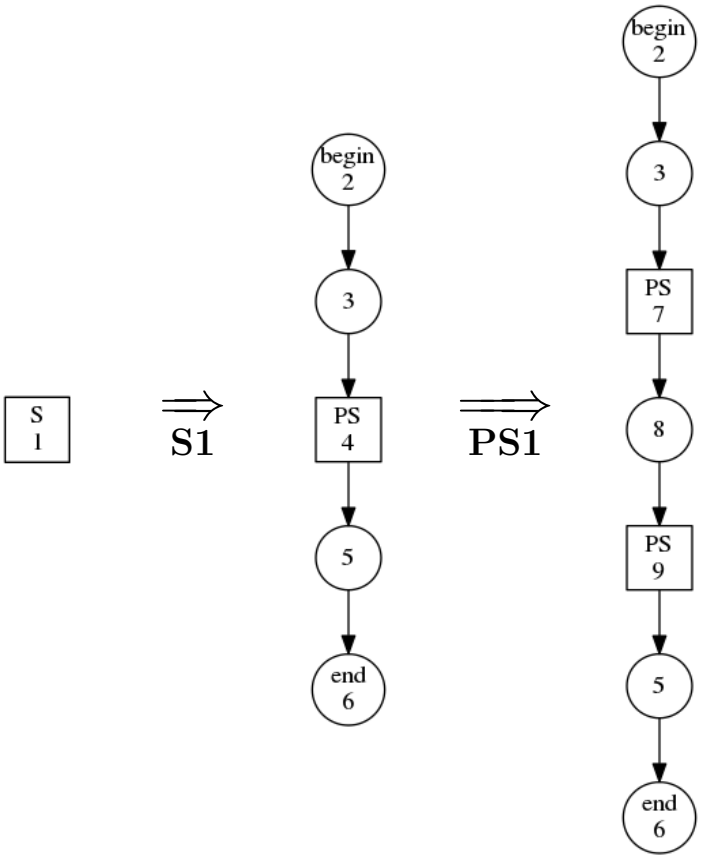


Figure B.8: Part 1 of *City of Khamoon* derivation

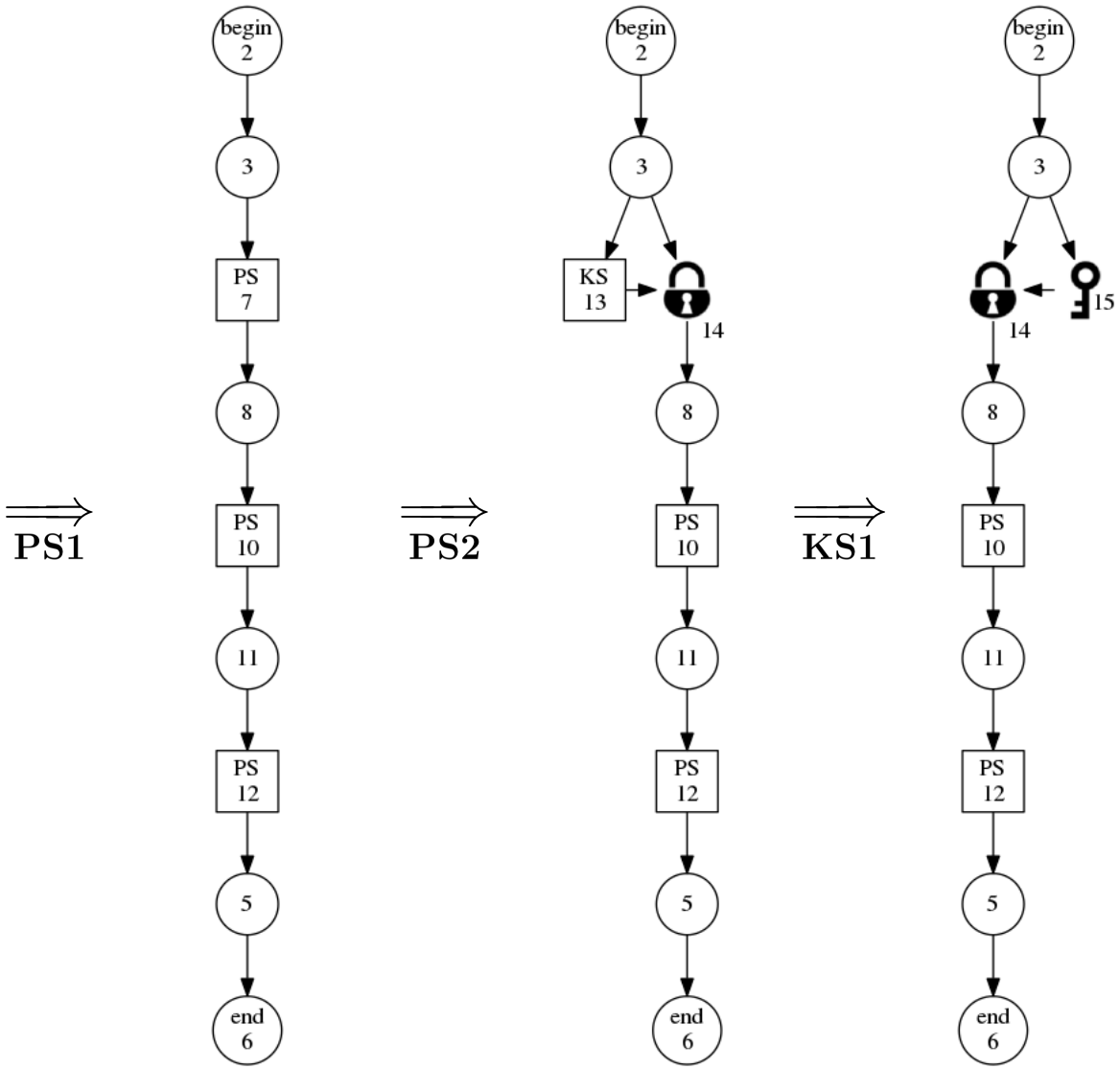


Figure B.8: Part 2 of *City of Khamoon* derivation

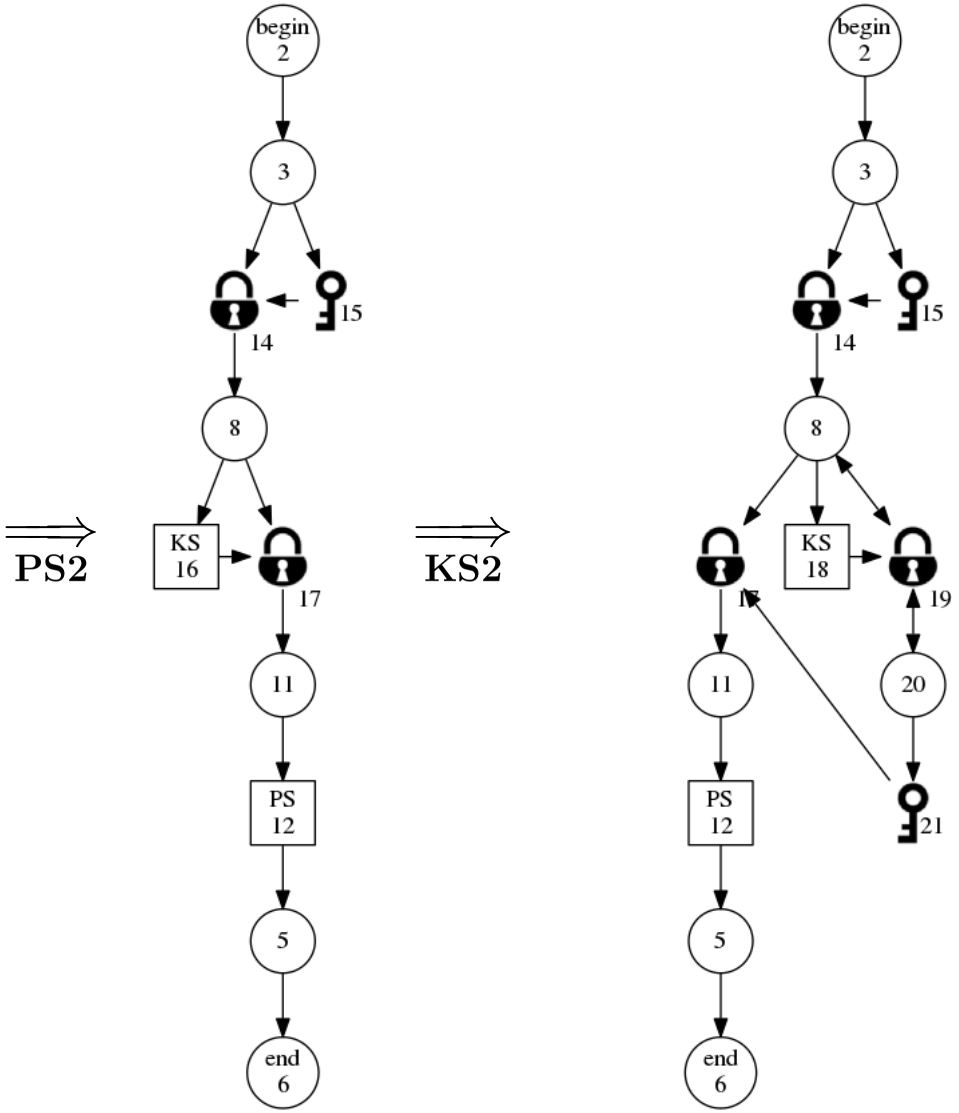


Figure B.8: Part 3 of *City of Khamoon* derivation

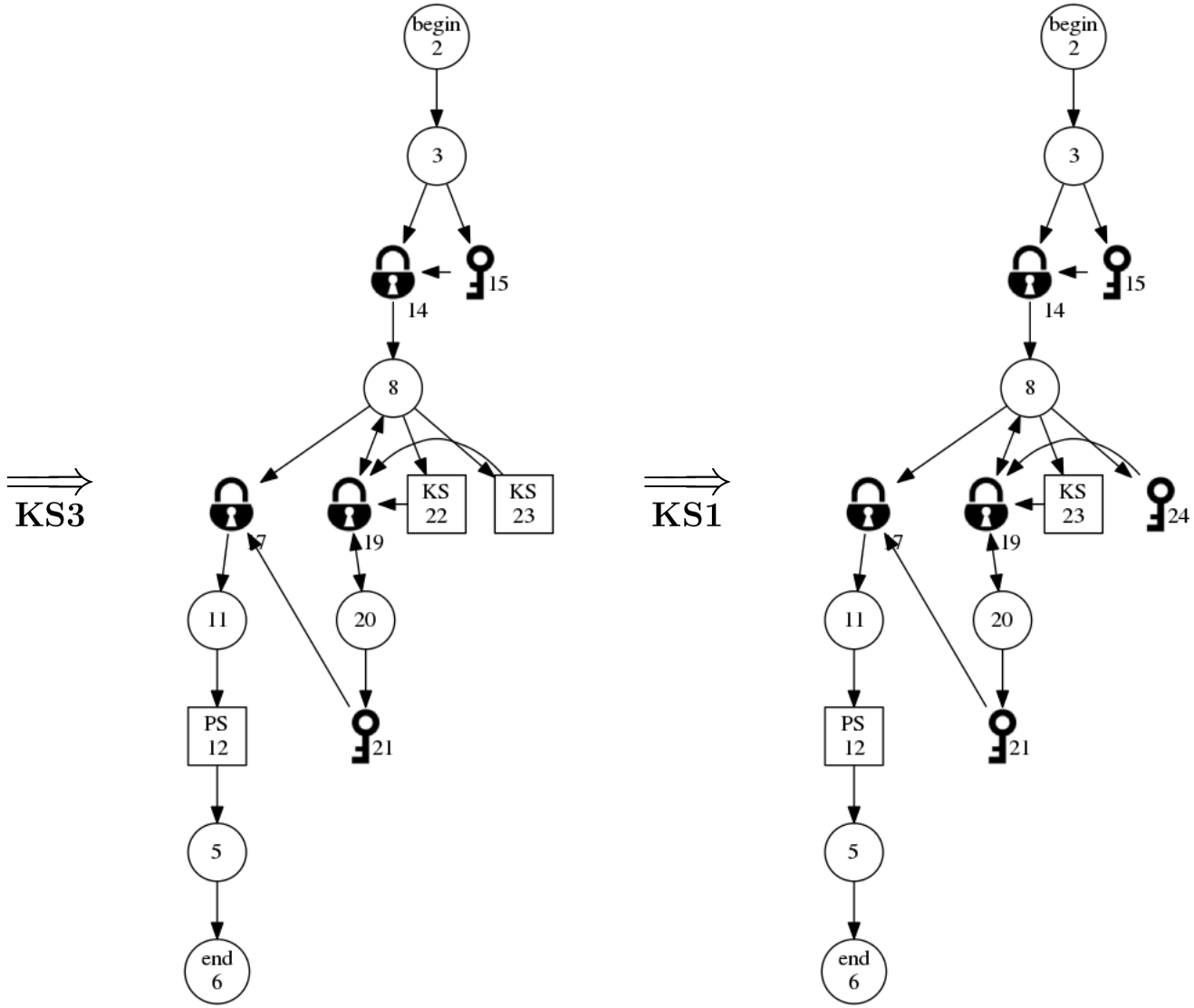


Figure B.8: Part 4 of *City of Khamoon* derivation

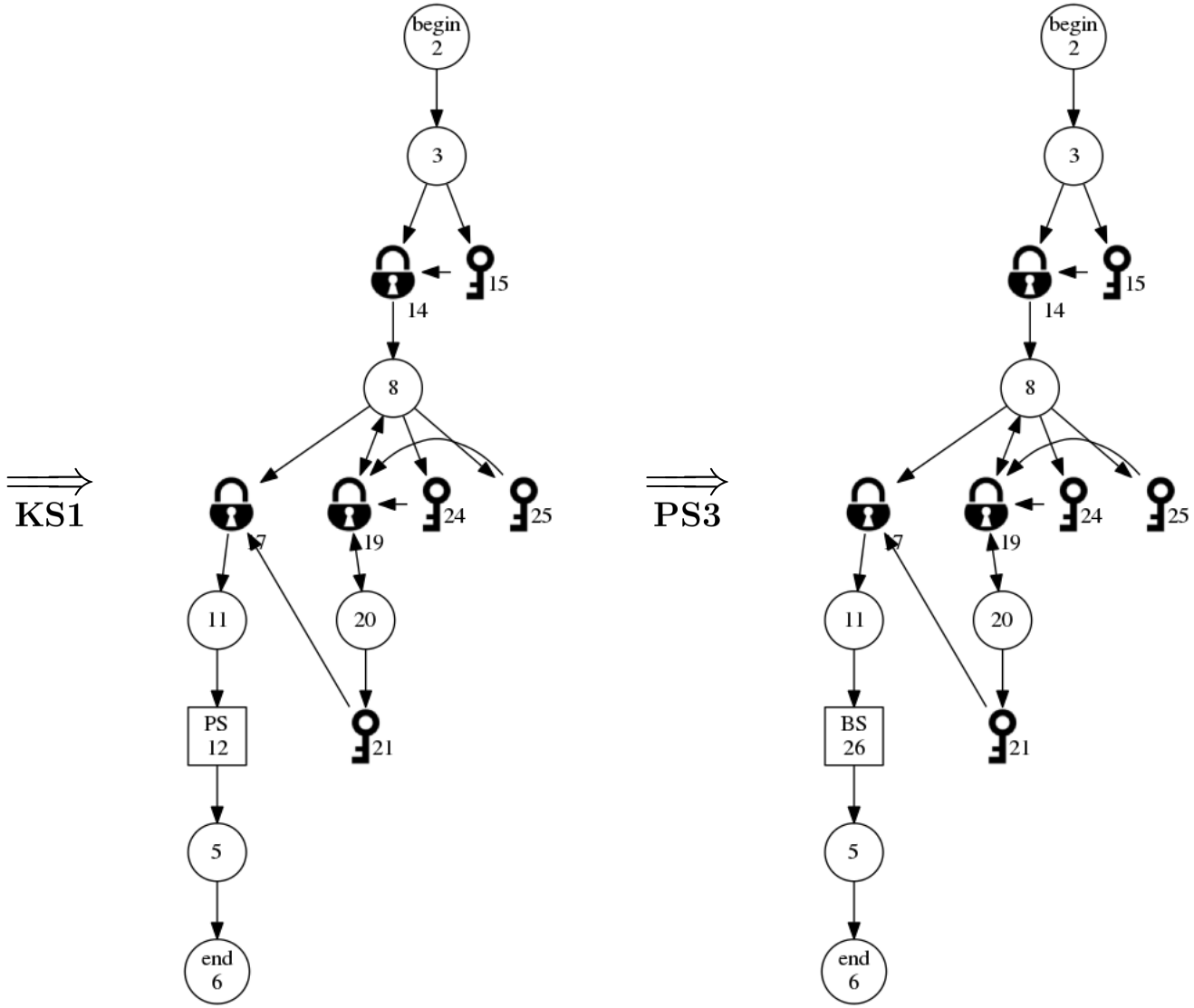


Figure B.8: Part 5 of *City of Khamoon* derivation

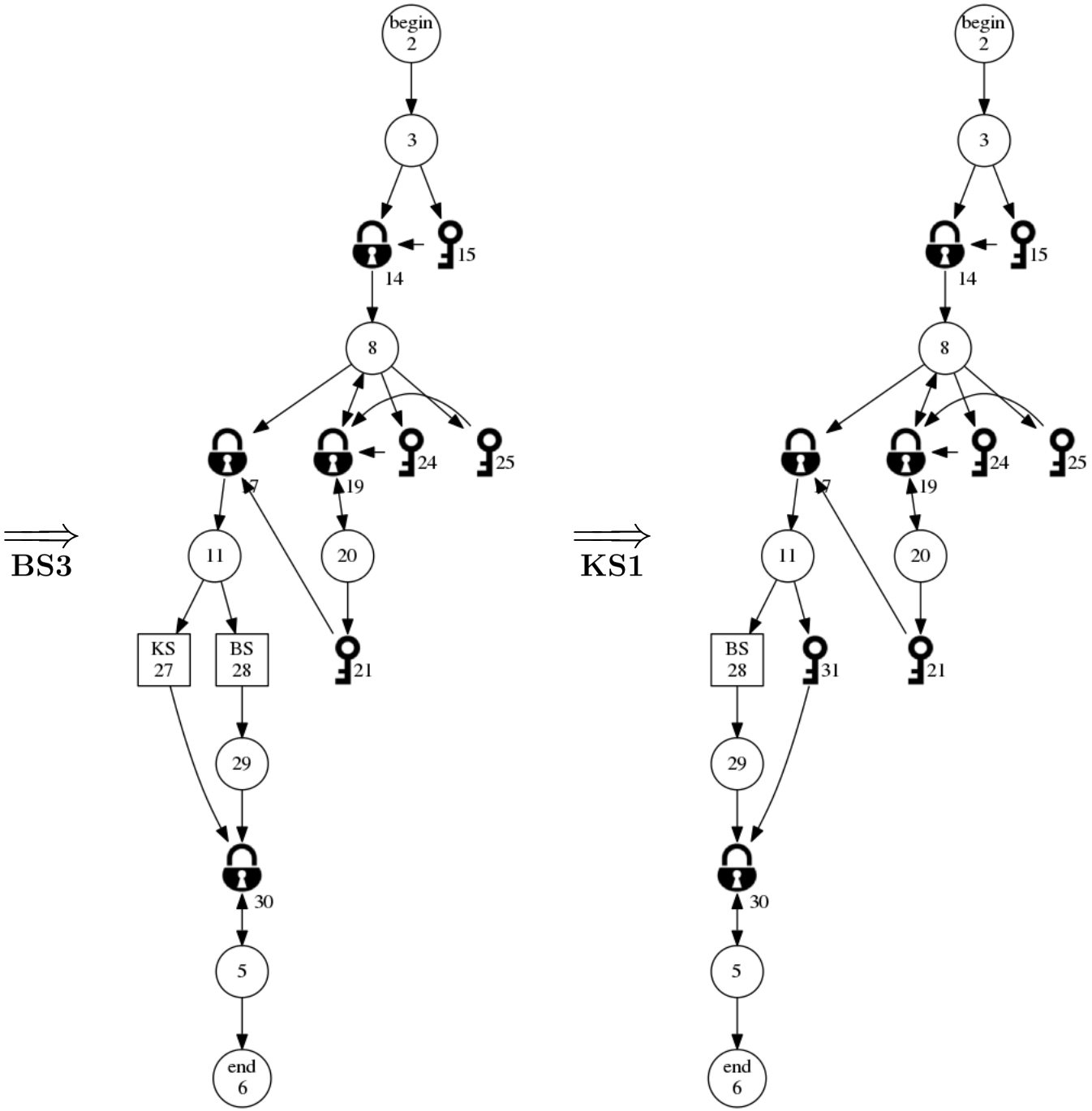


Figure B.8: Part 6 of *City of Khamoon* derivation

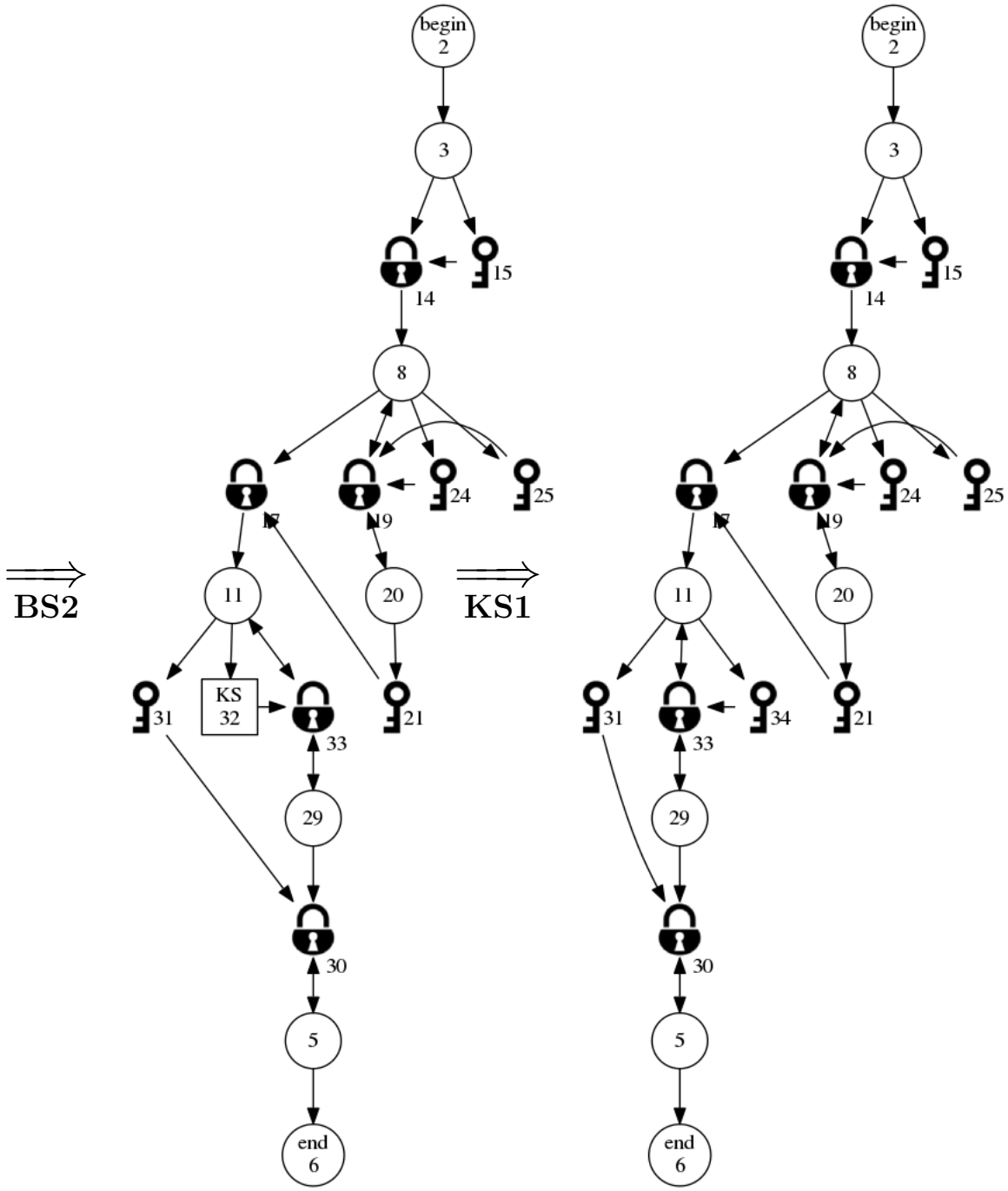


Figure B.8: Part 7 of *City of Khamoon* derivation

B.2.5 Level 14: Atlantis

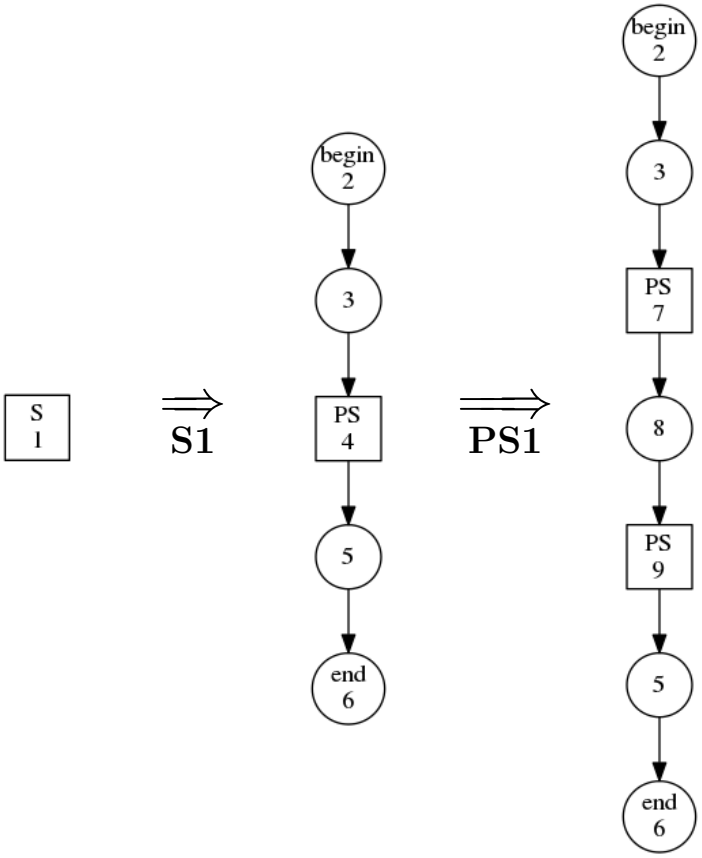


Figure B.9: Part 1 of *Atlantis* derivation

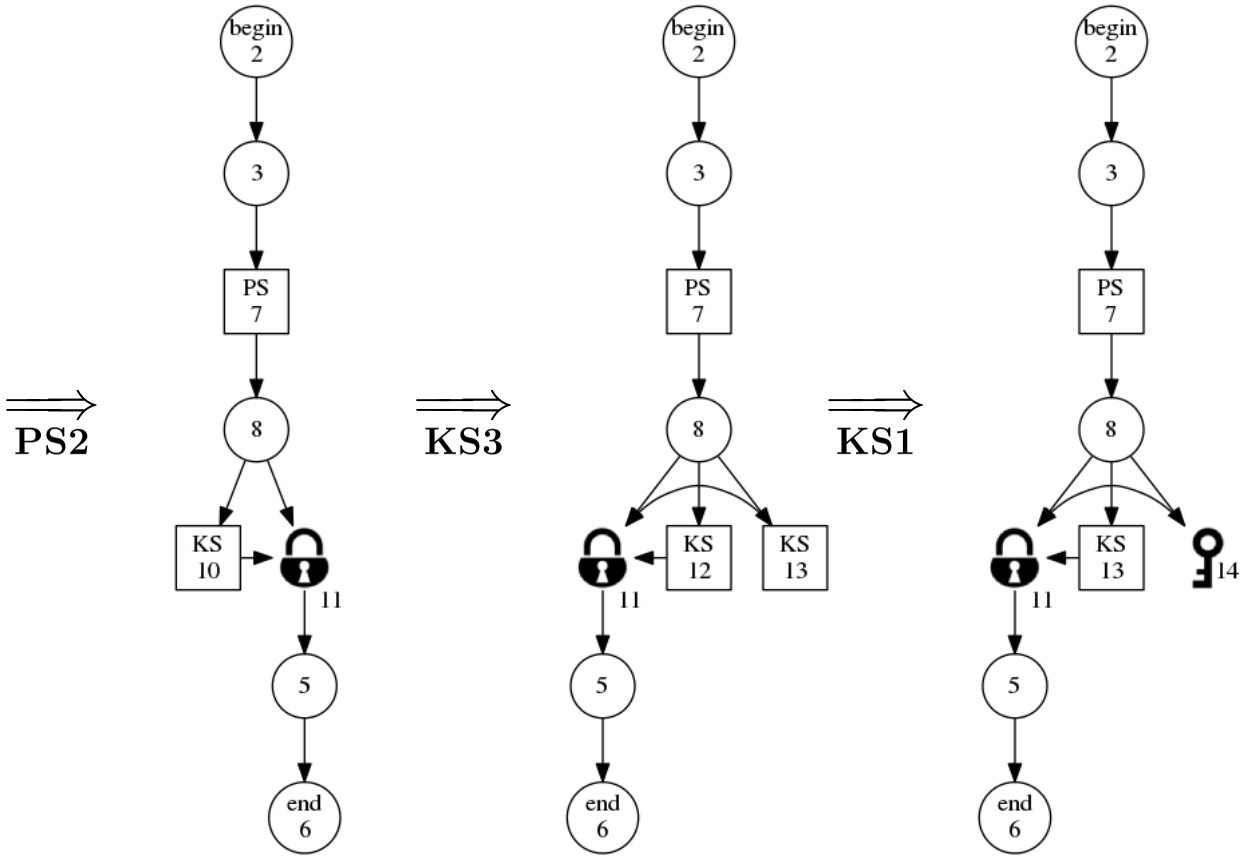


Figure B.9: Part 2 of *Atlantis* derivation

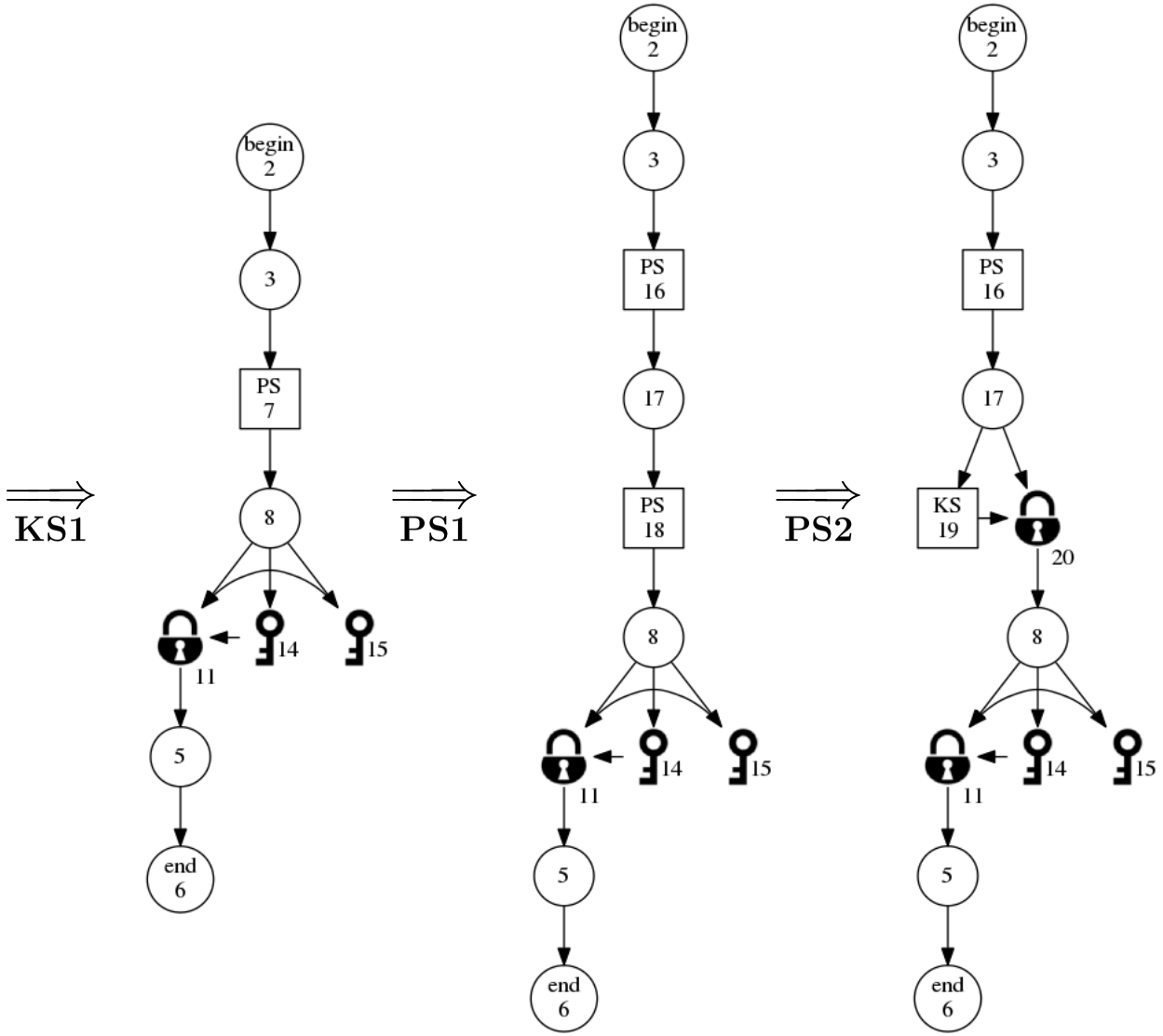


Figure B.9: Part 3 of *Atlantis* derivation

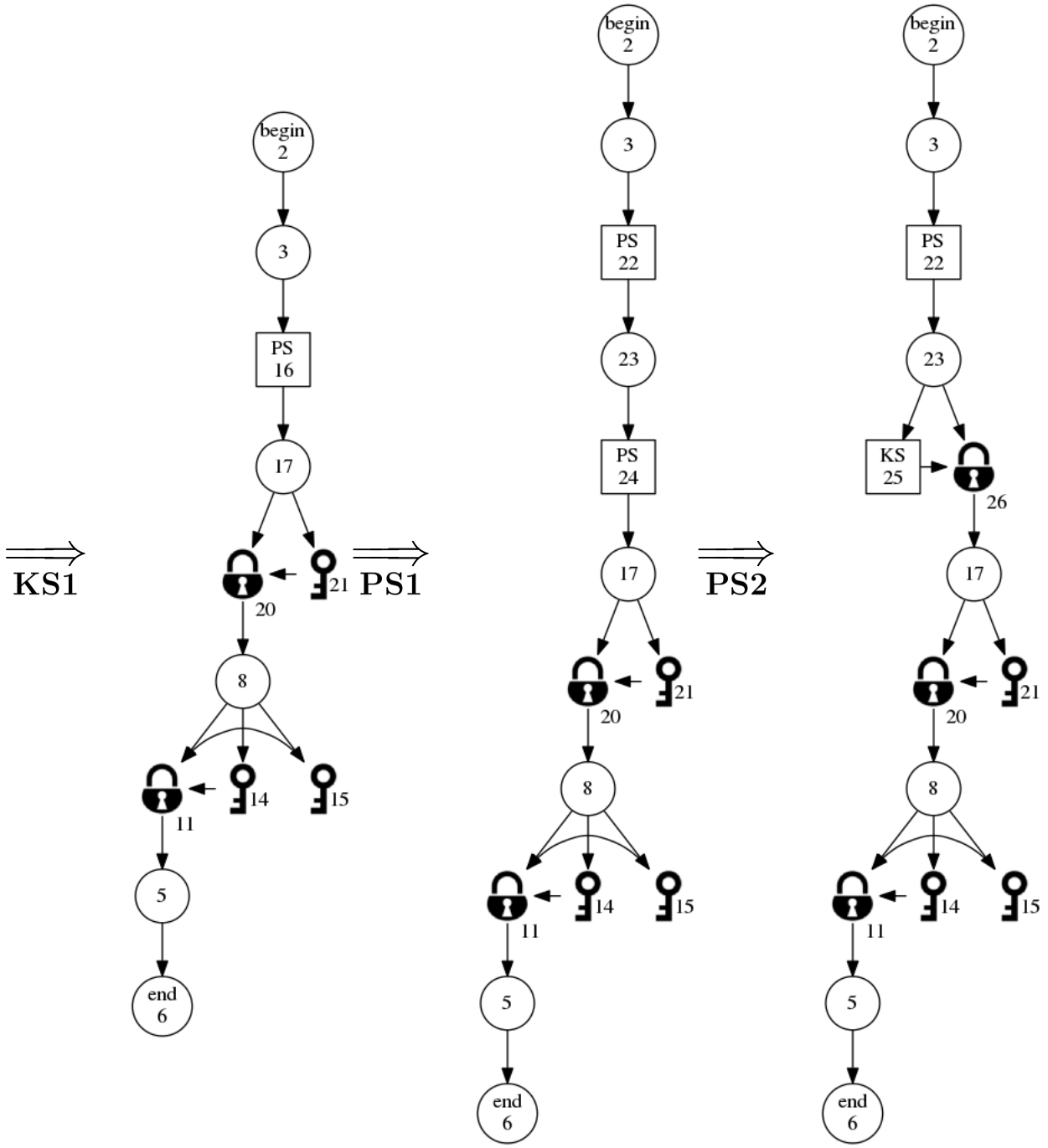


Figure B.9: Part 4 of *Atlantis* derivation

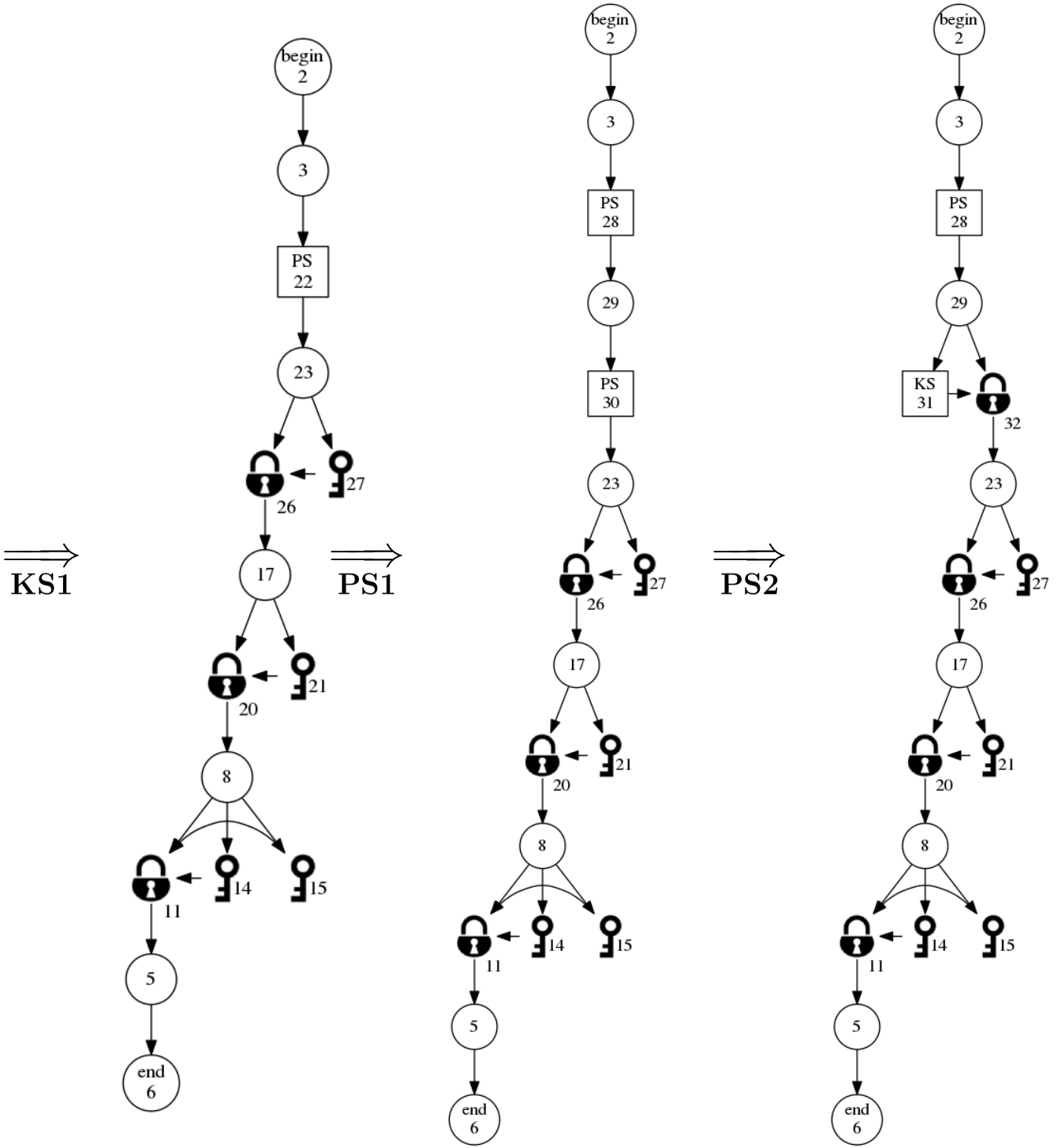


Figure B.9: Part 5 of *Atlantis* derivation

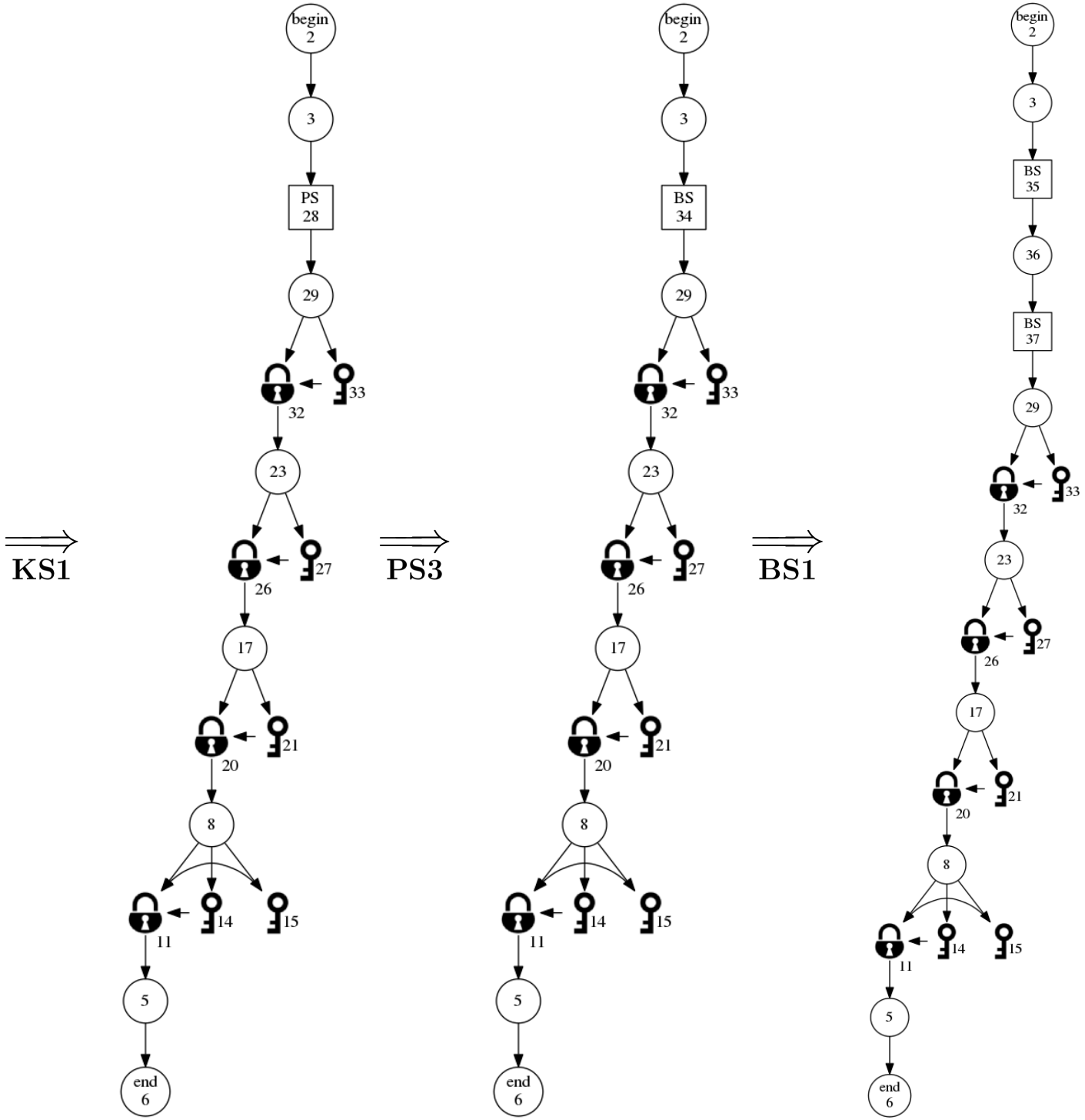


Figure B.9: Part 6 of *Atlantis* derivation

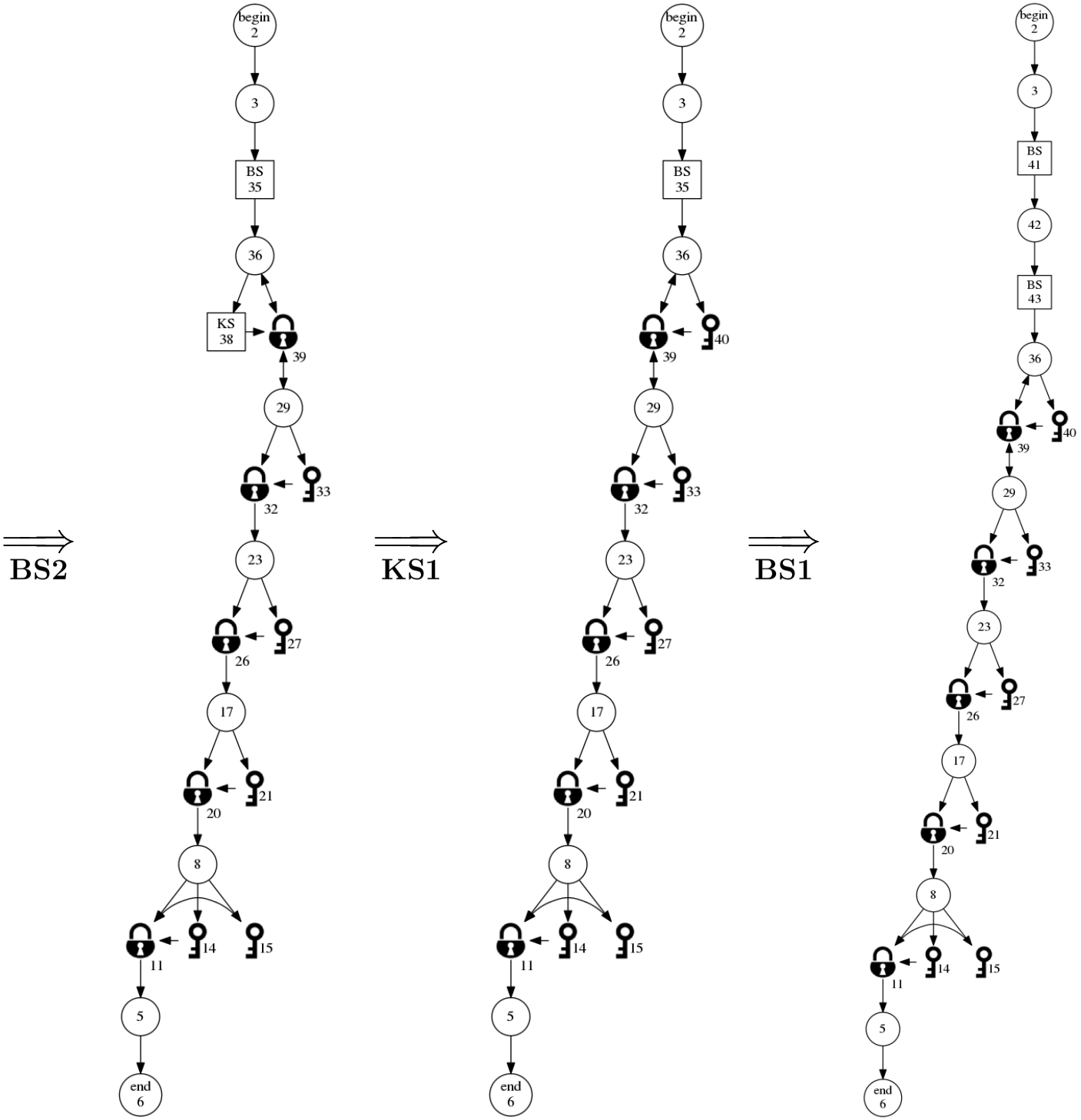


Figure B.9: Part 7 of *Atlantis* derivation

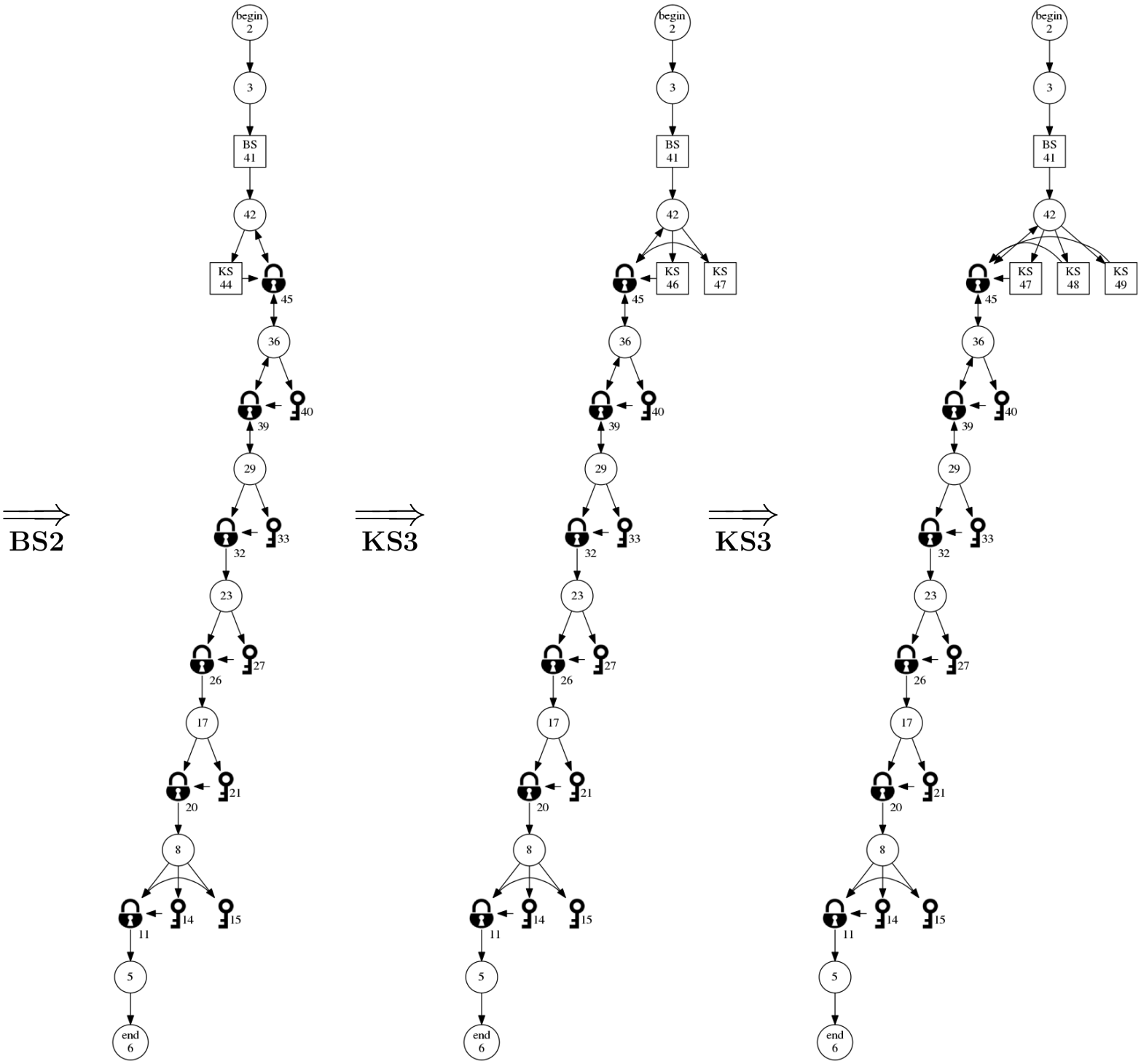


Figure B.9: Part 8 of *Atlantis* derivation

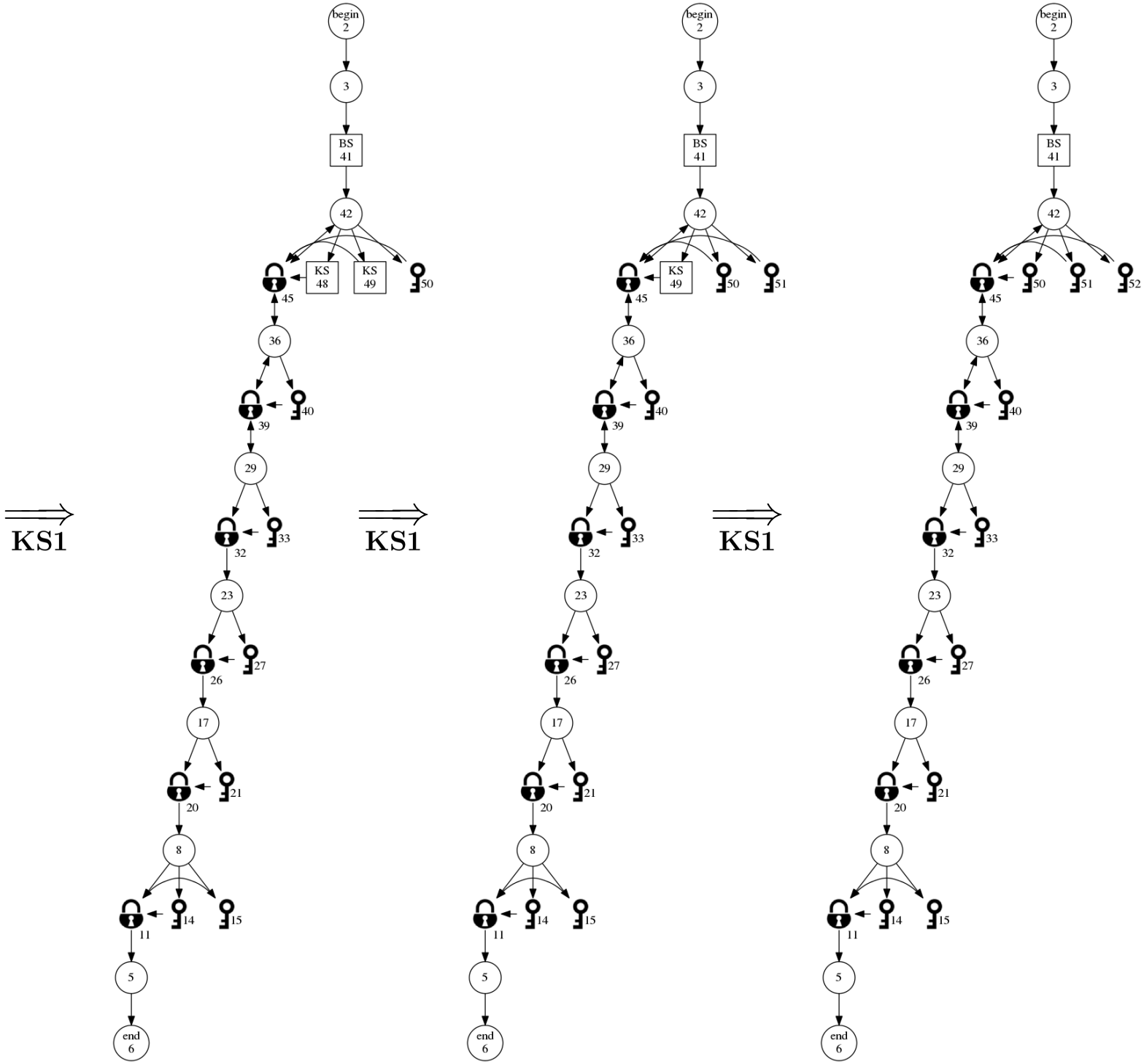


Figure B.9: Part 9 of *Atlantis* derivation

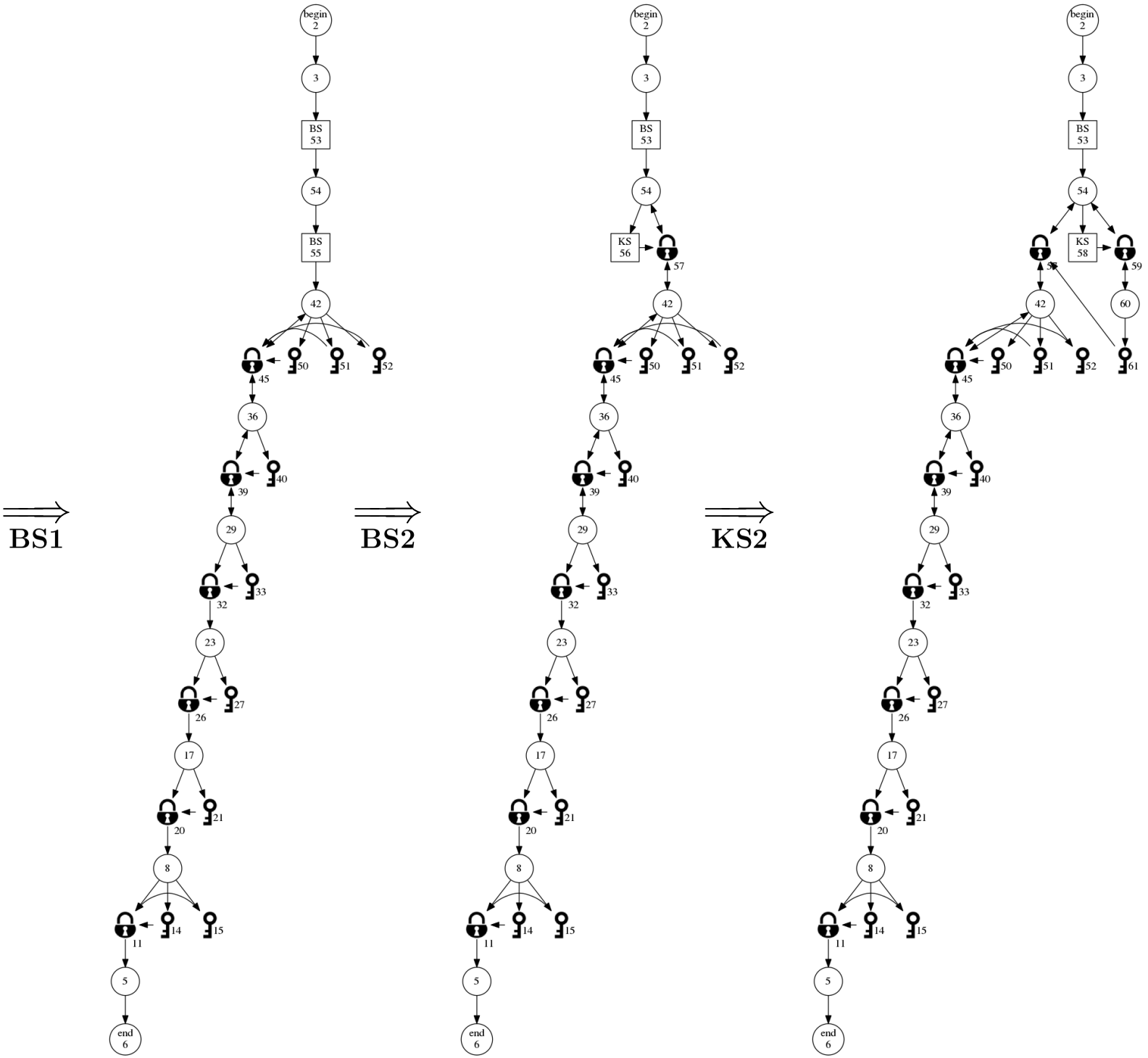


Figure B.9: Part 10 of *Atlantis* derivation

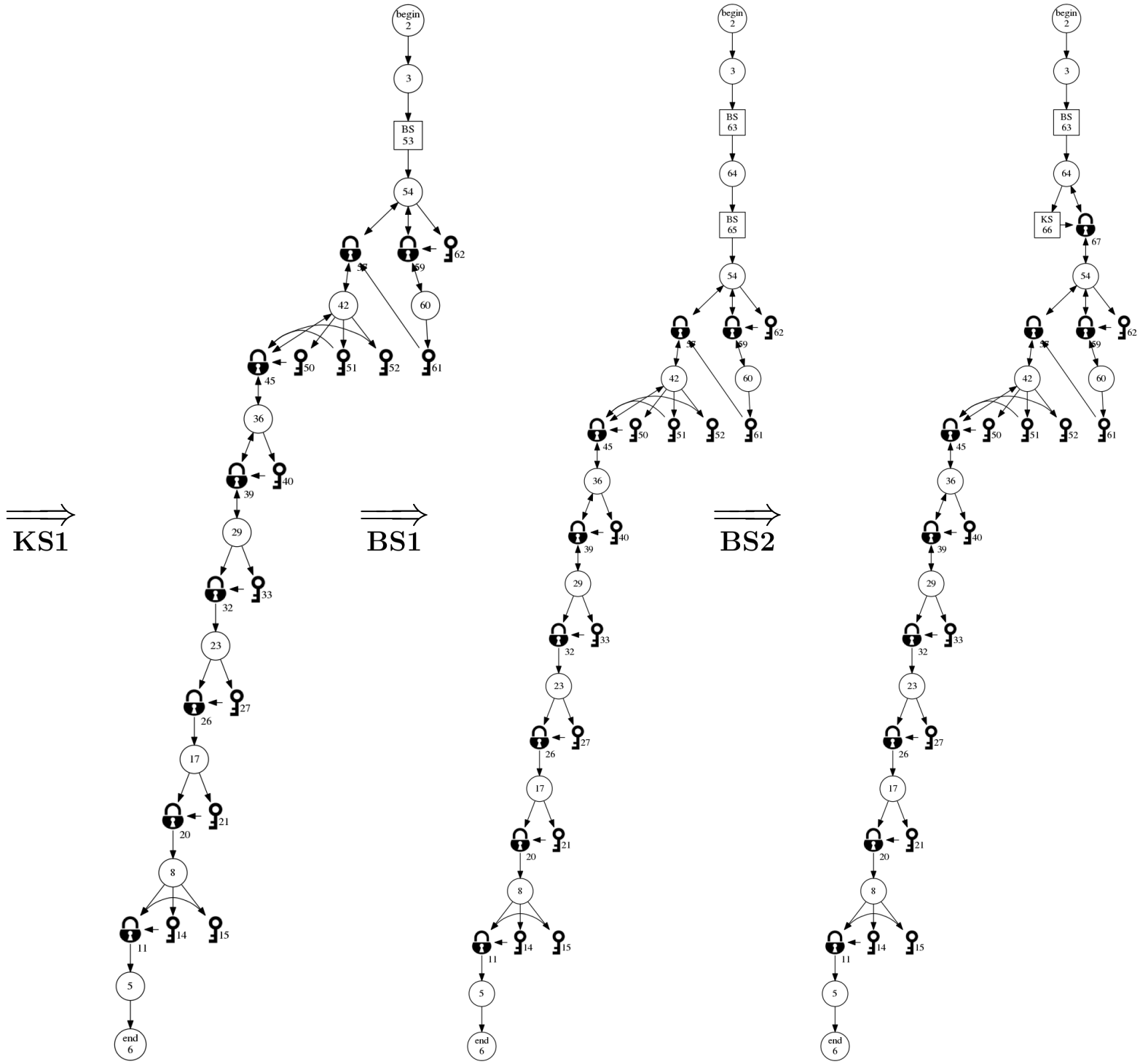


Figure B.9: Part 11 of *Atlantis* derivation

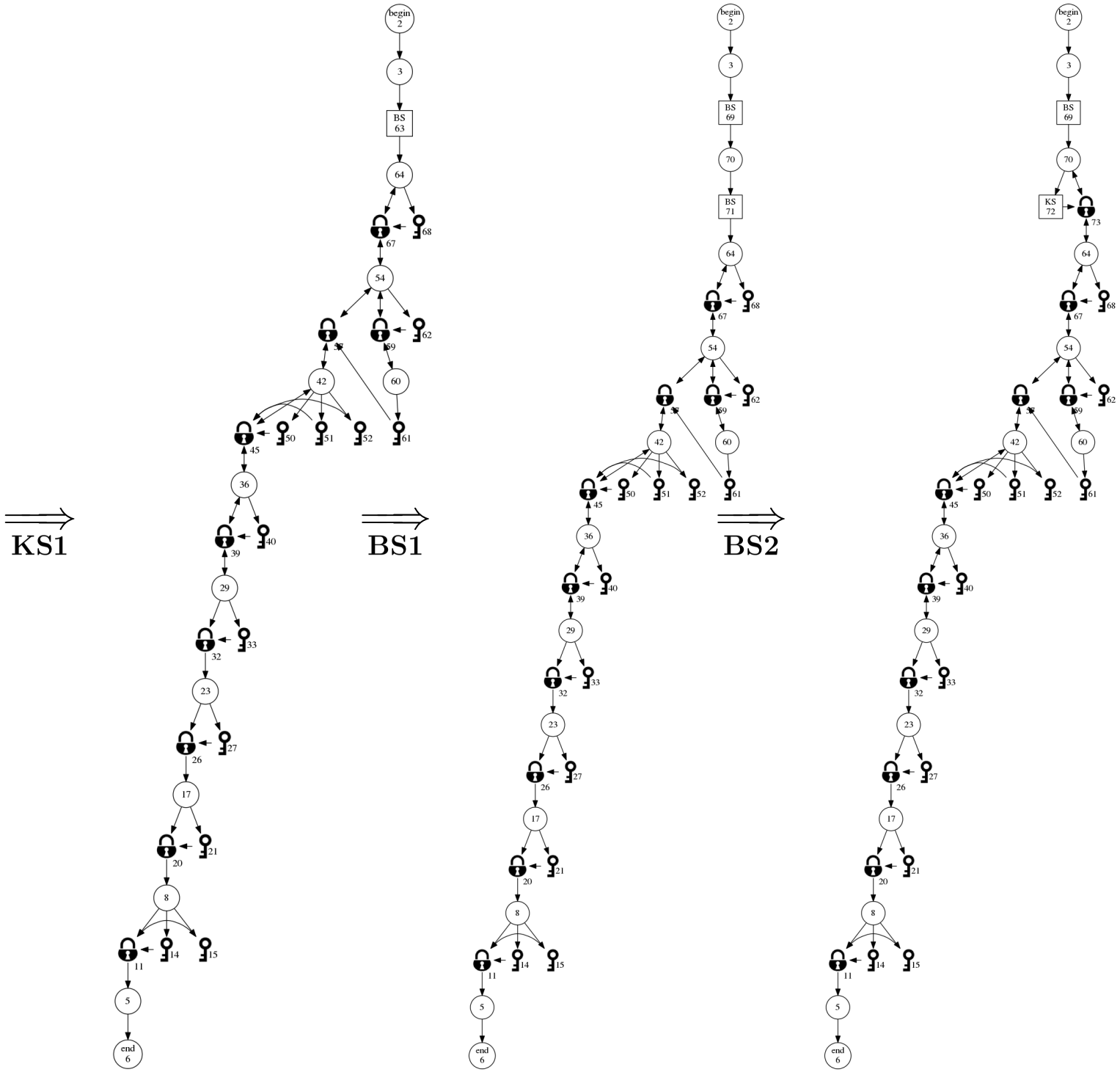


Figure B.9: Part 12 of *Atlantis* derivation

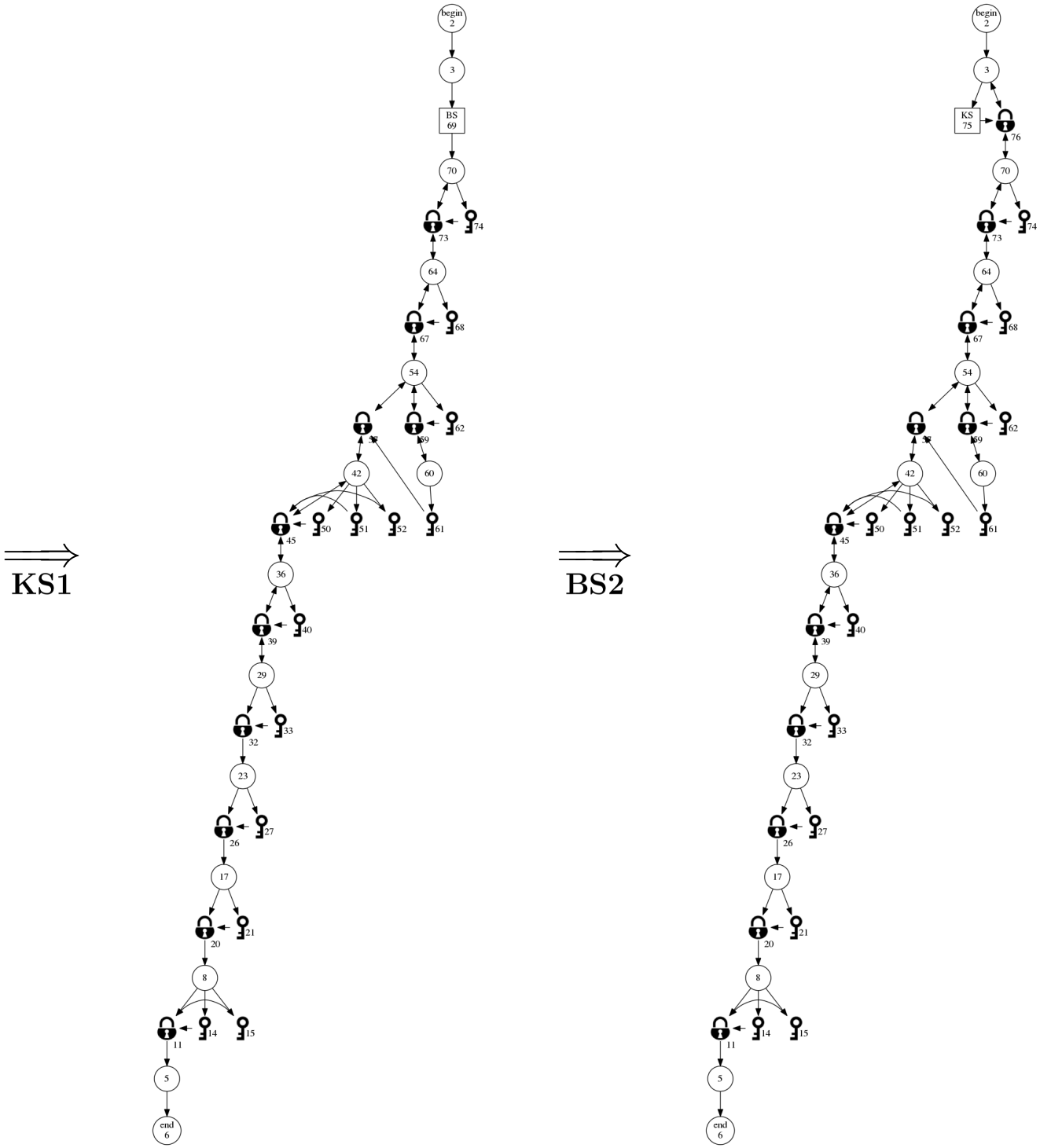


Figure B.9: Part 13 of *Atlantis* derivation

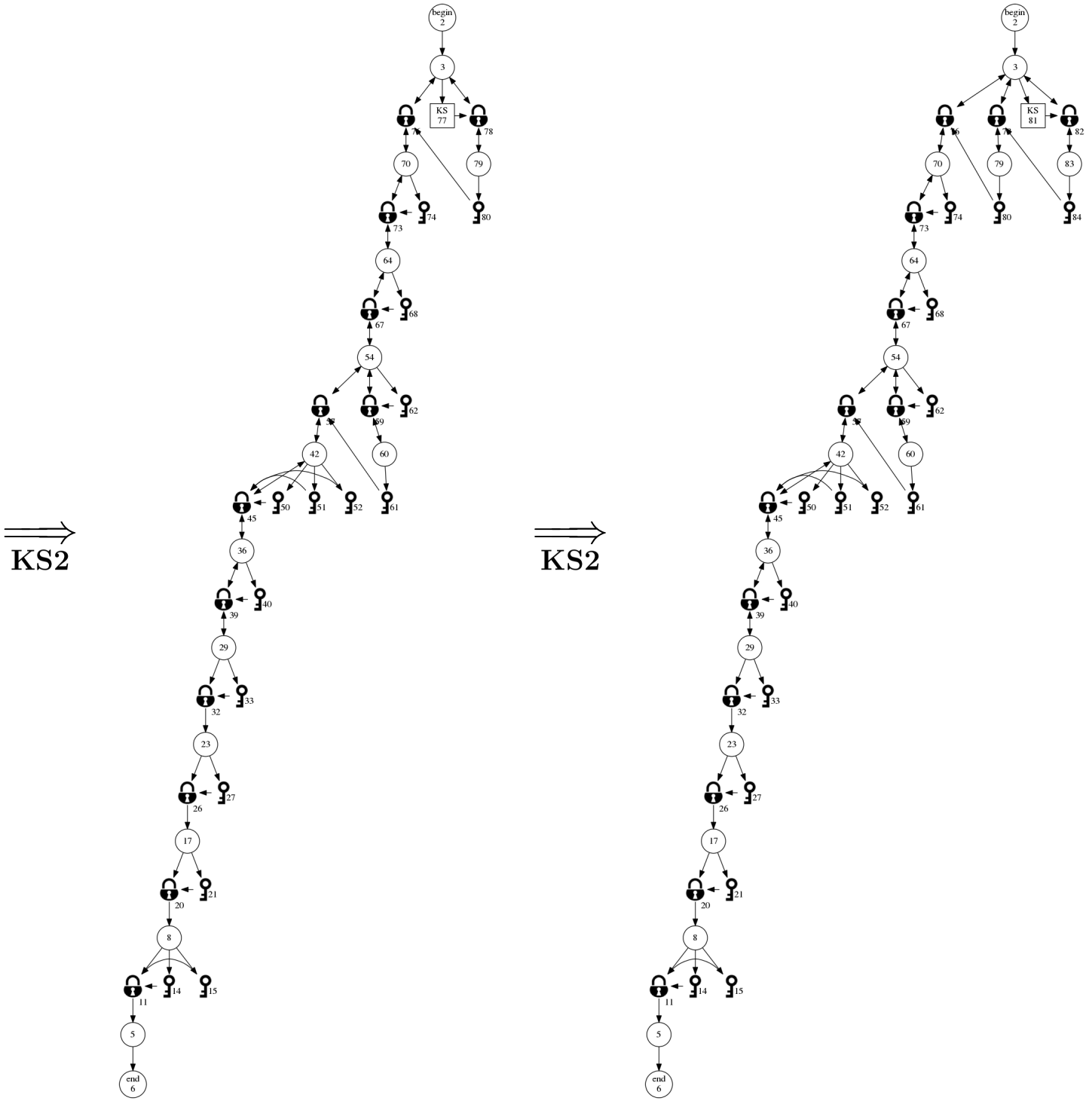


Figure B.9: Part 14 of *Atlantis* derivation

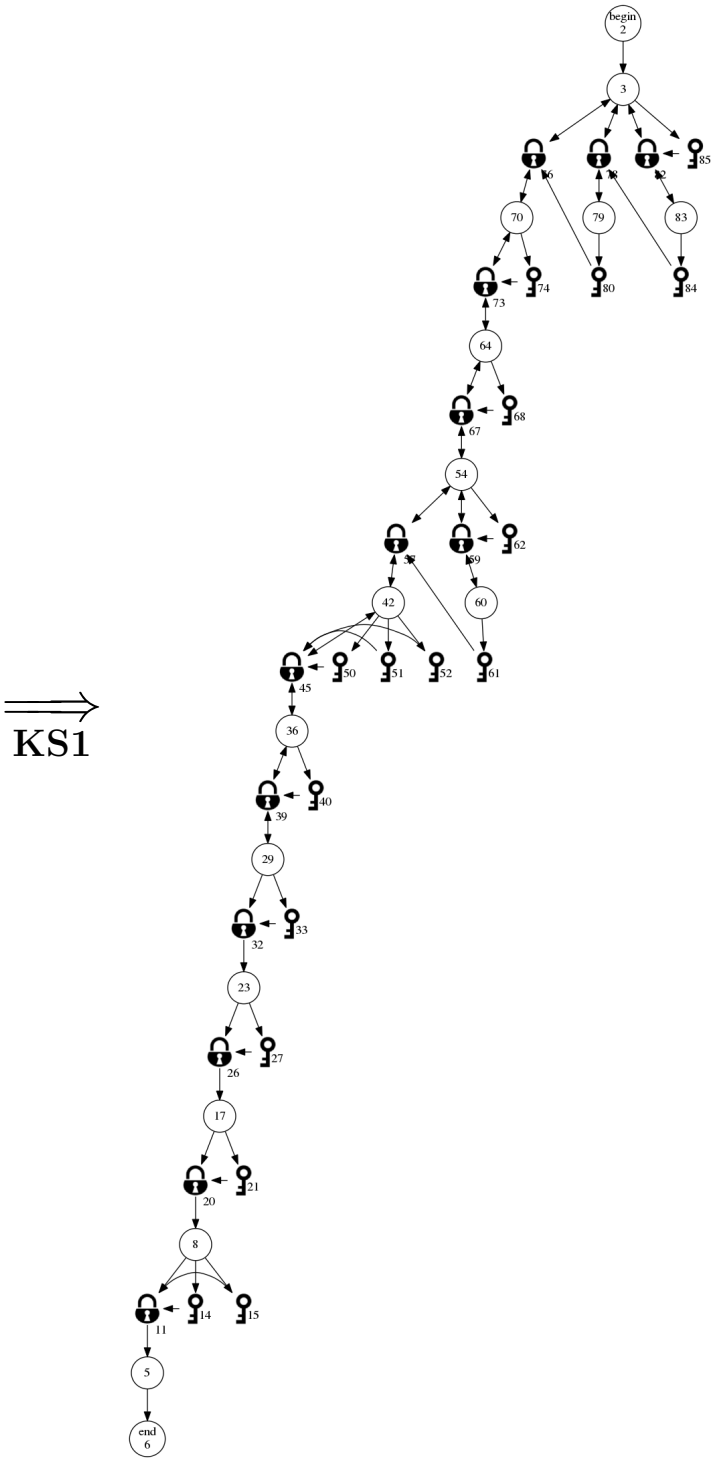


Figure B.9: Part 15 of *Atlantis* derivation

B.3 Legend of Zelda: Link's Awakening

B.3.1 Tail Cave

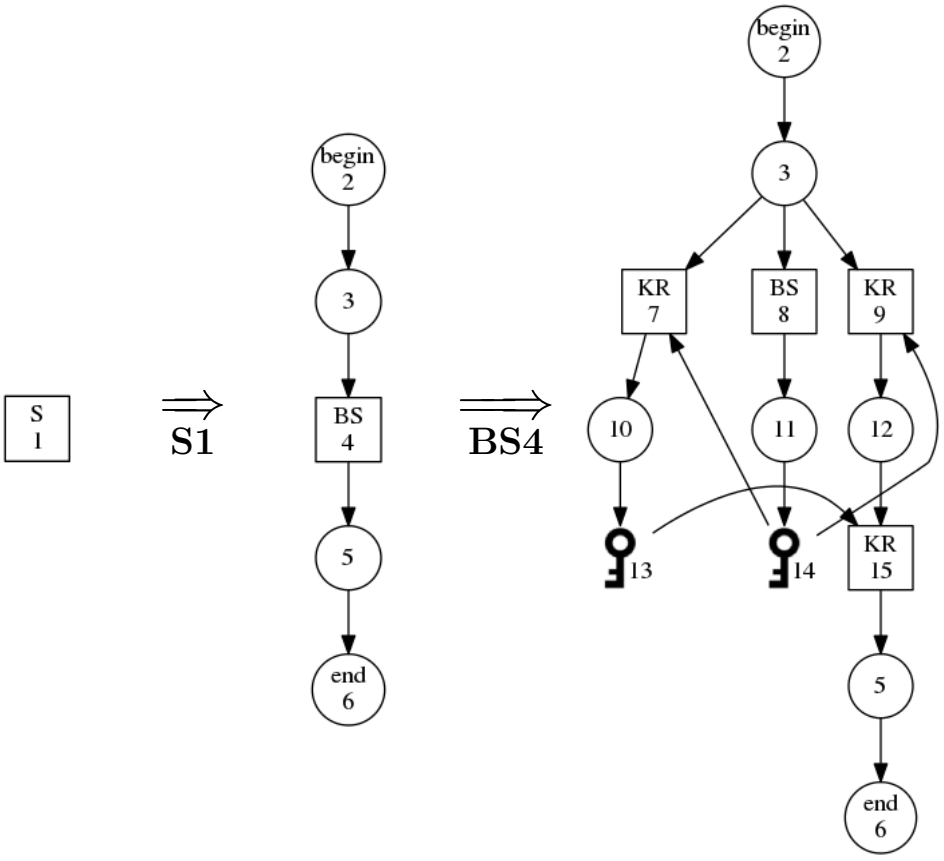


Figure B.10: Part 1 of *Tail Cave* derivation

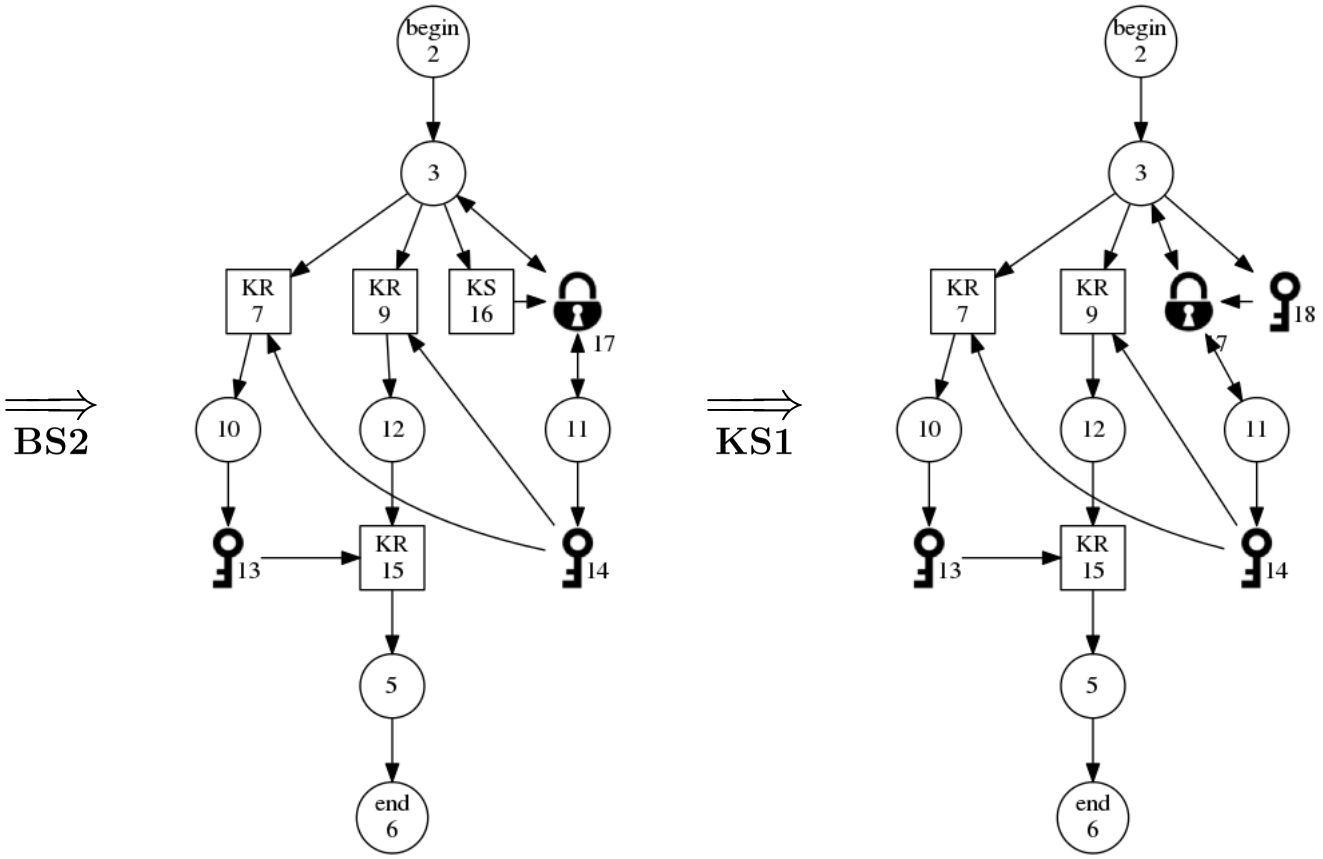


Figure B.10: Part 2 of *Tail Cave* derivation

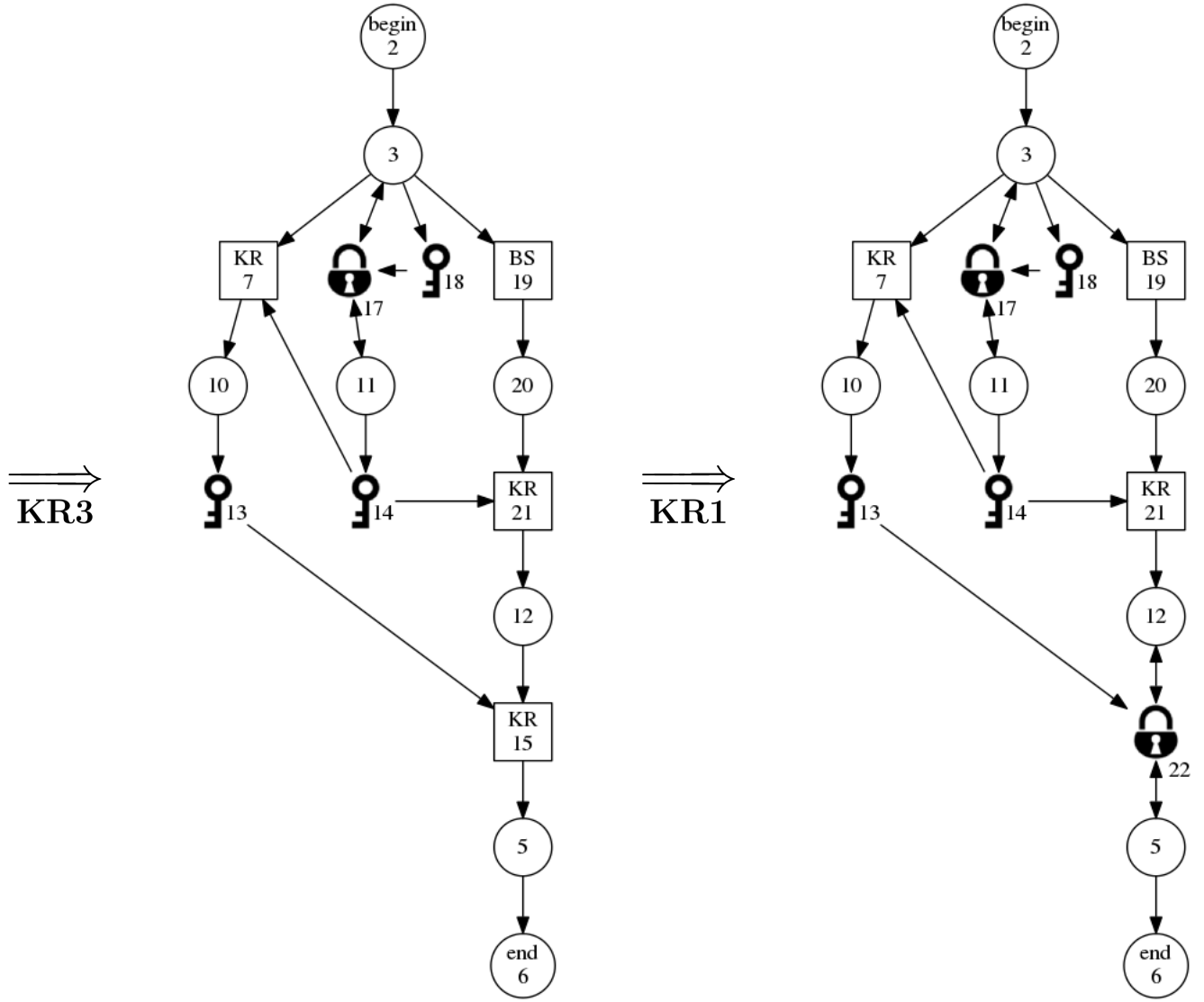


Figure B.10: Part 3 of *Tail Cave* derivation

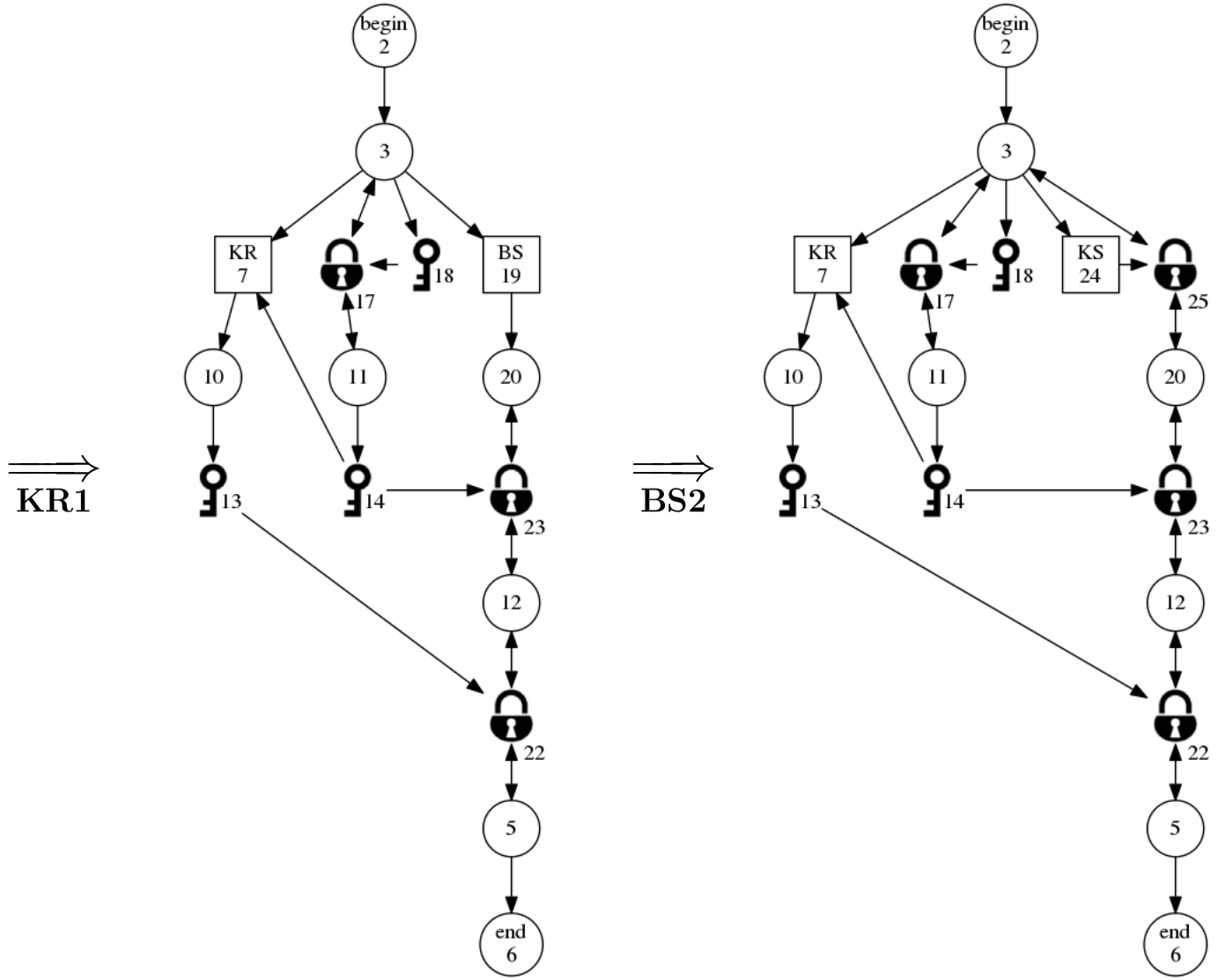


Figure B.10: Part 4 of *Tail Cave* derivation

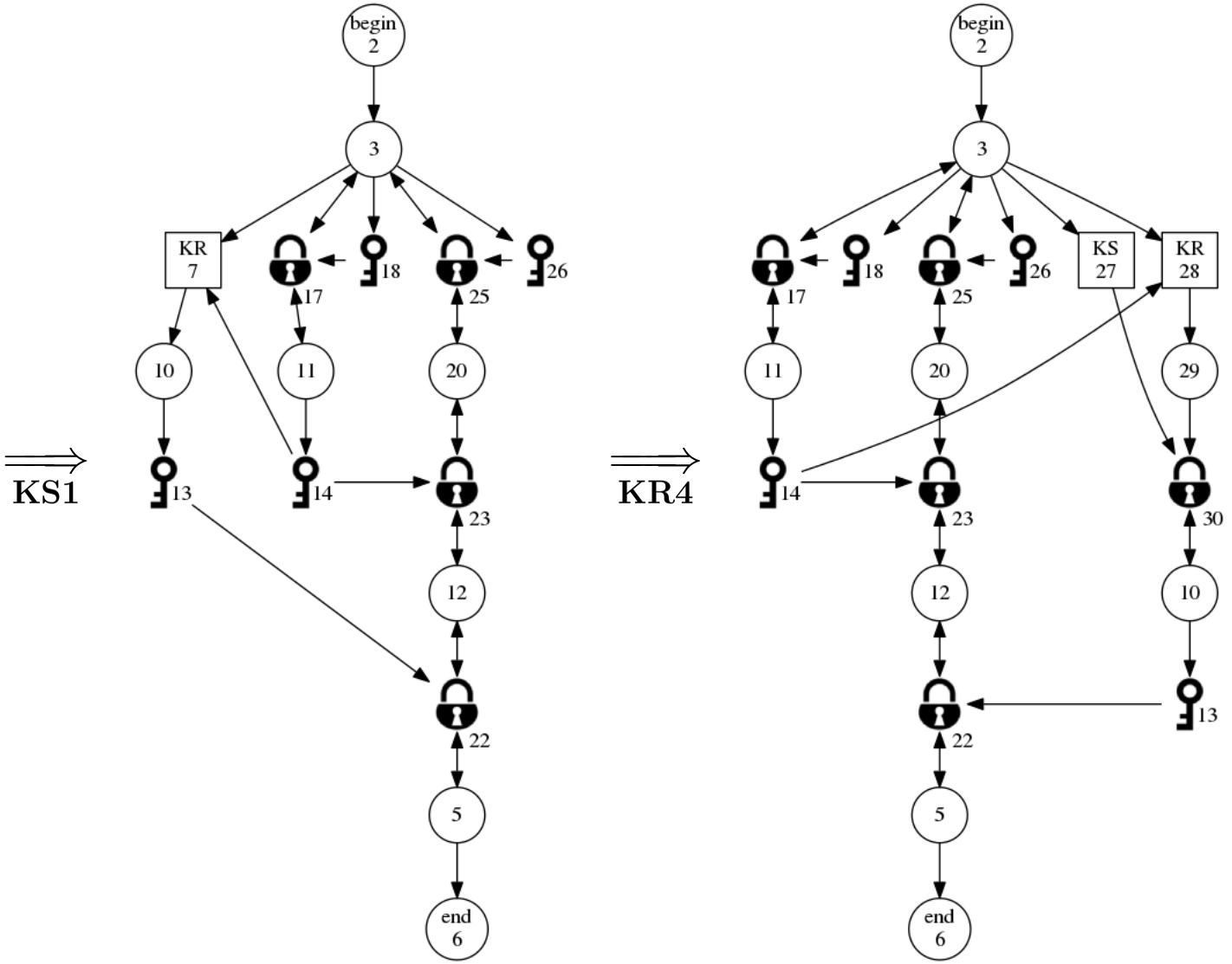


Figure B.10: Part 5 of *Tail Cave* derivation

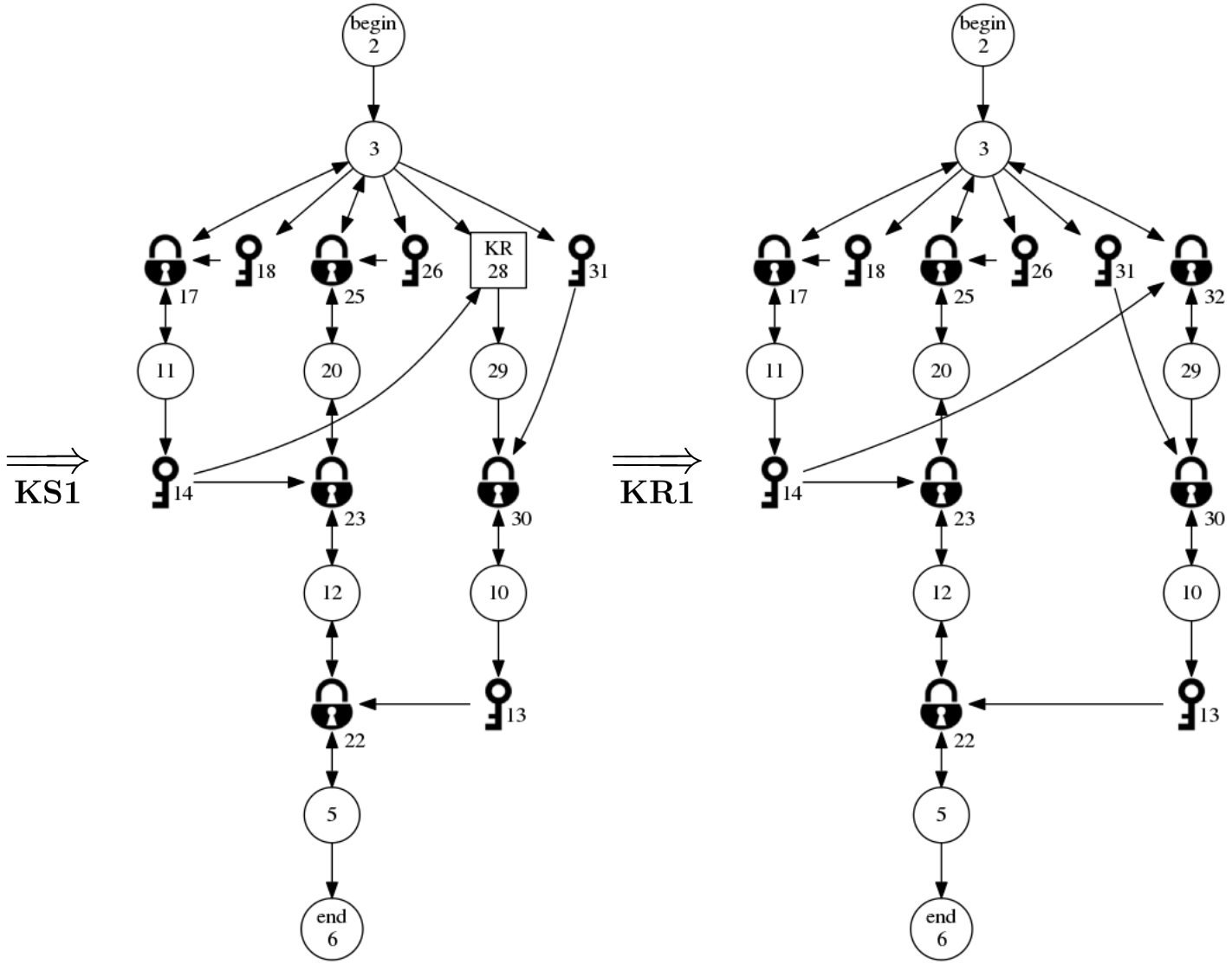


Figure B.10: Part 6 of *Tail Cave* derivation

B.3.2 Bottle Grotto

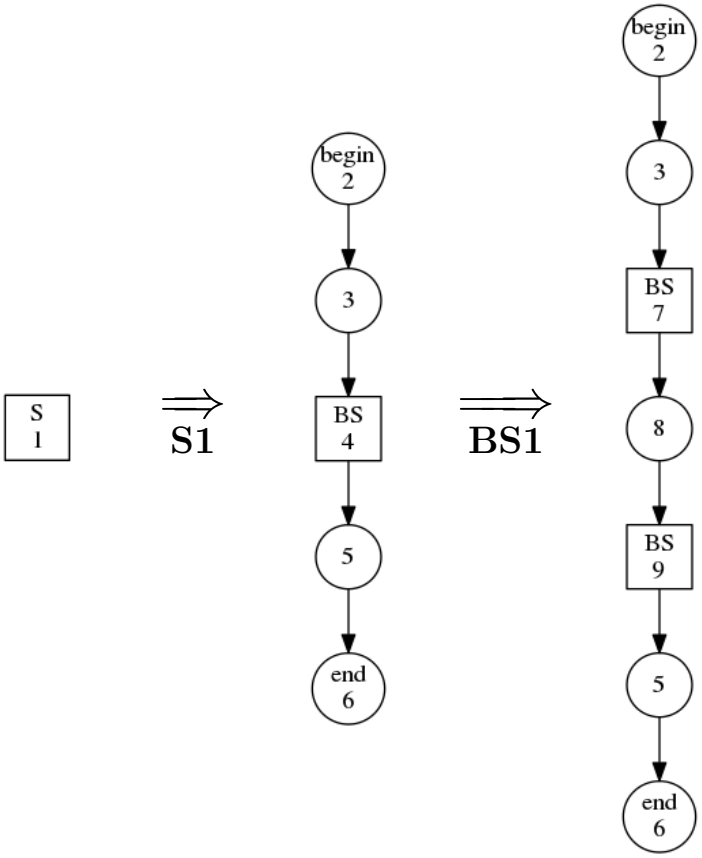


Figure B.11: Part 1 of *Bottle Grotto* derivation

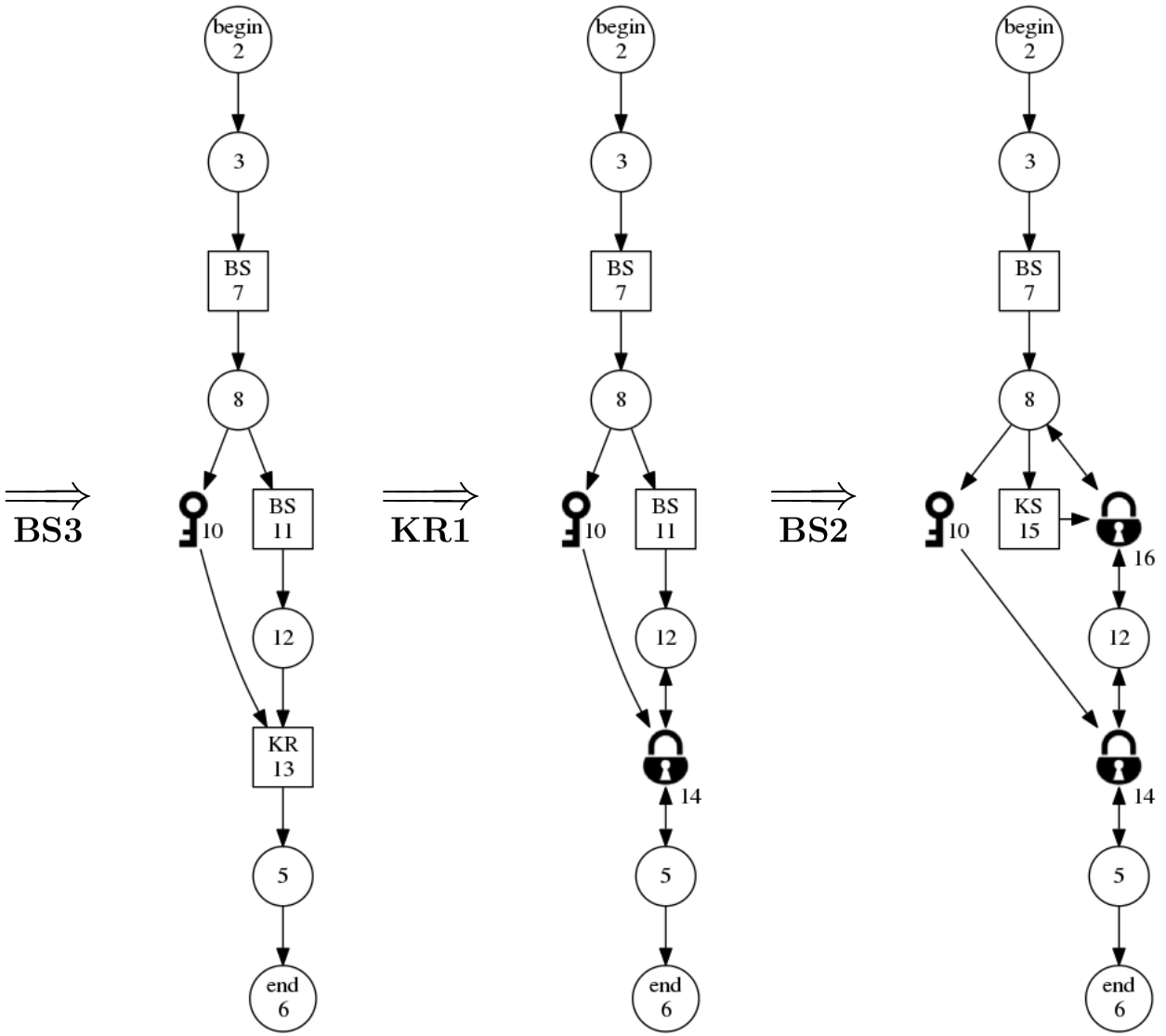


Figure B.11: Part 2 of *Bottle Grotto* derivation

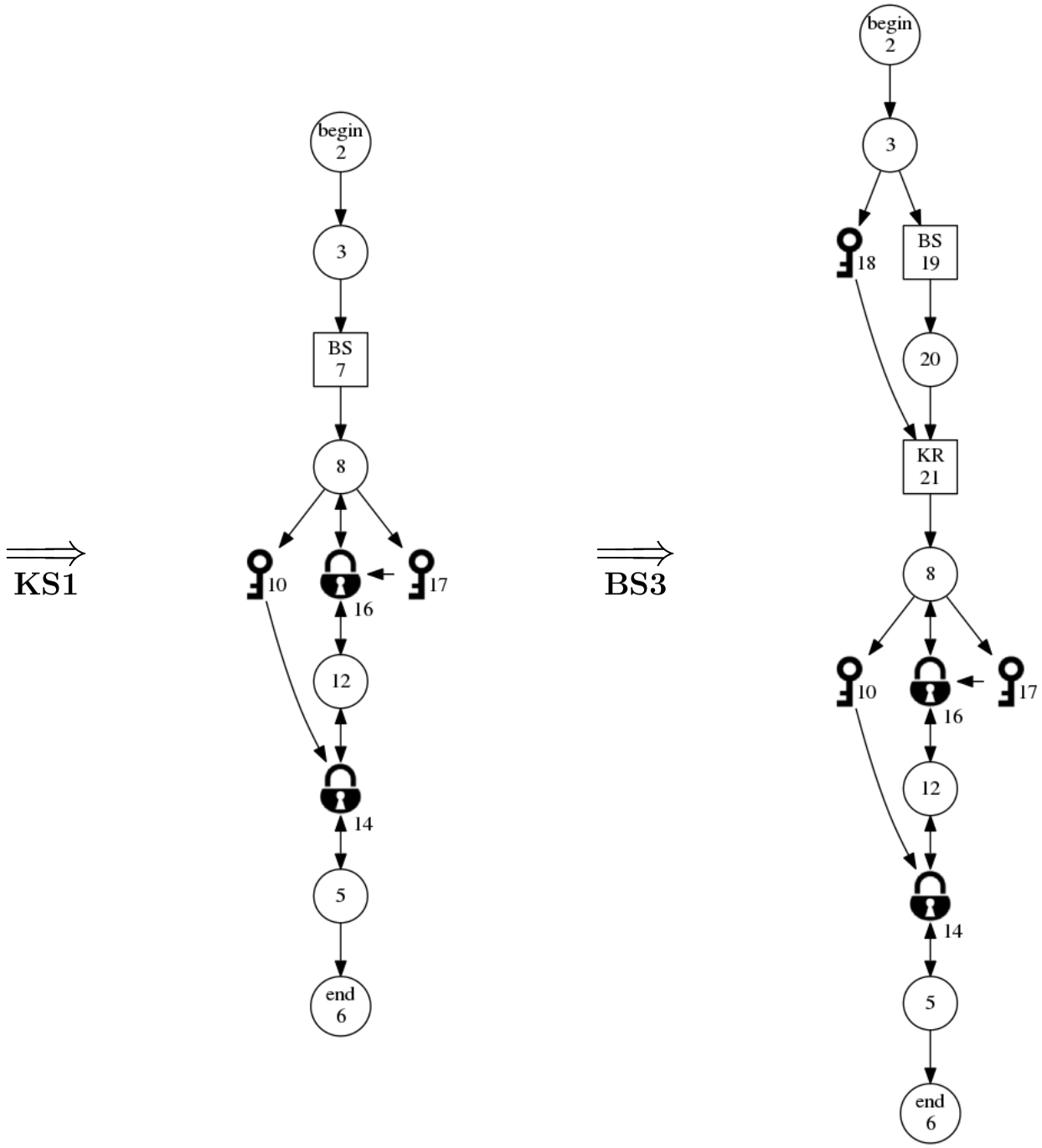


Figure B.11: Part 3 of *Bottle Grotto* derivation

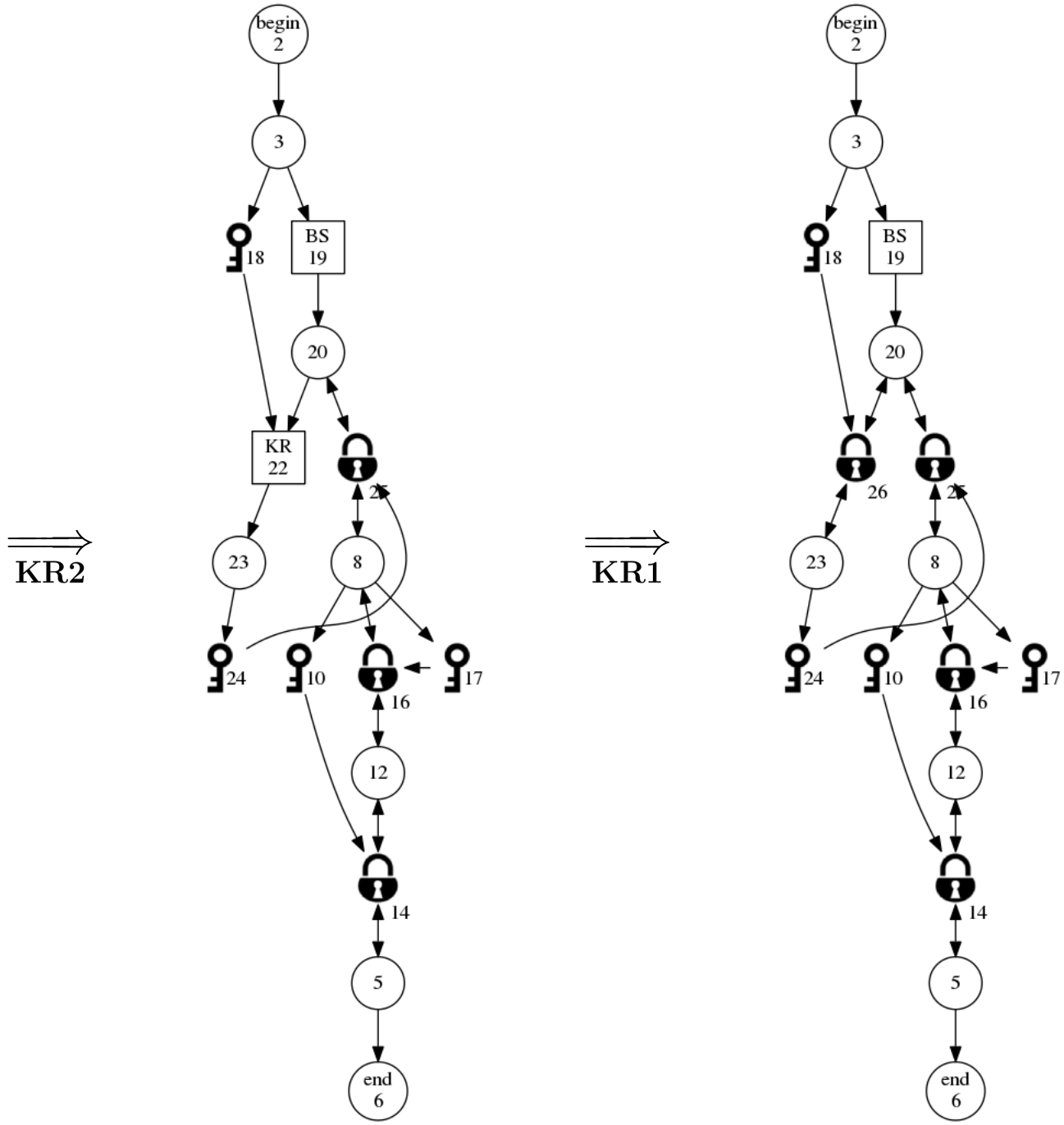


Figure B.11: Part 4 of *Bottle Grotto* derivation

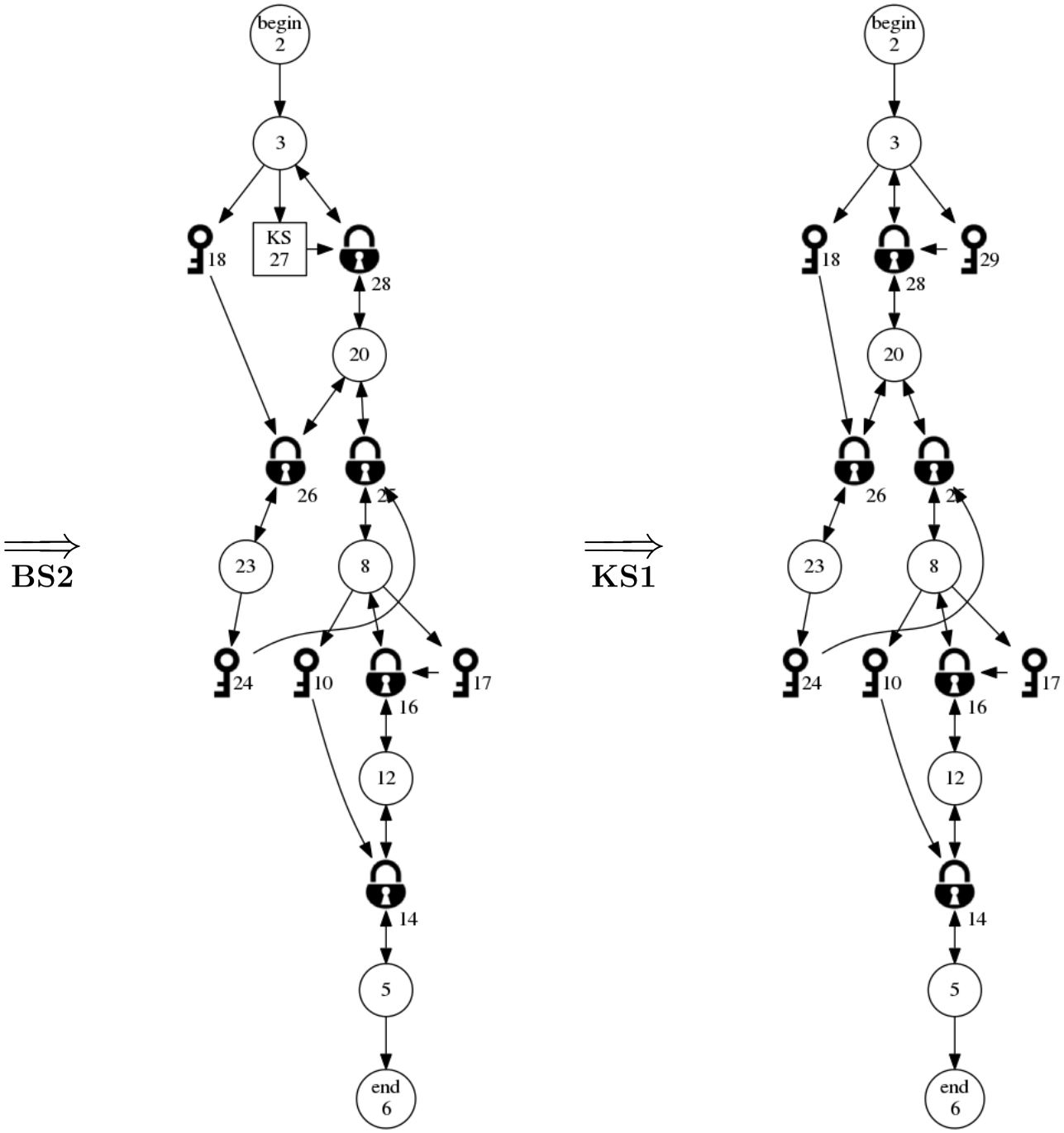


Figure B.11: Part 5 of *Bottle Grotto* derivation

B.3.3 Key Cavern

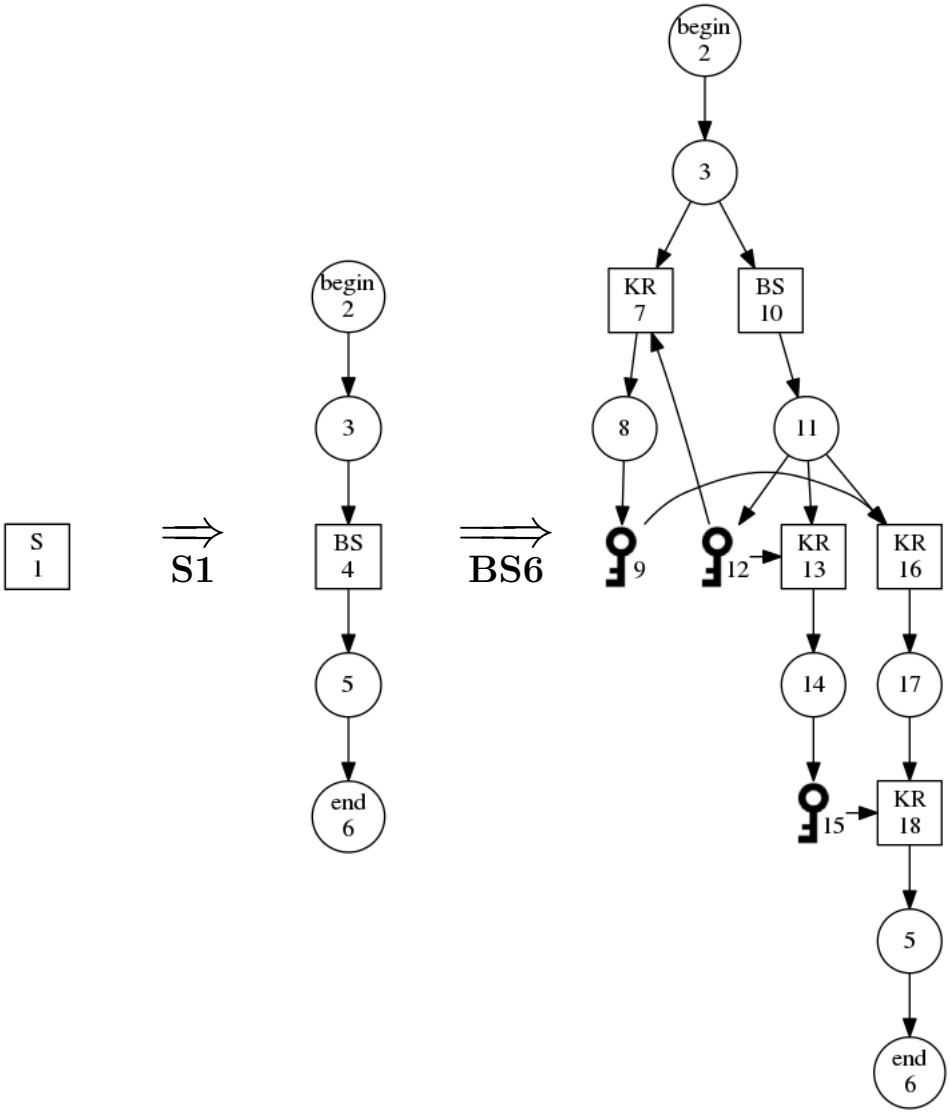


Figure B.12: Part 1 of *Key Cavern* derivation

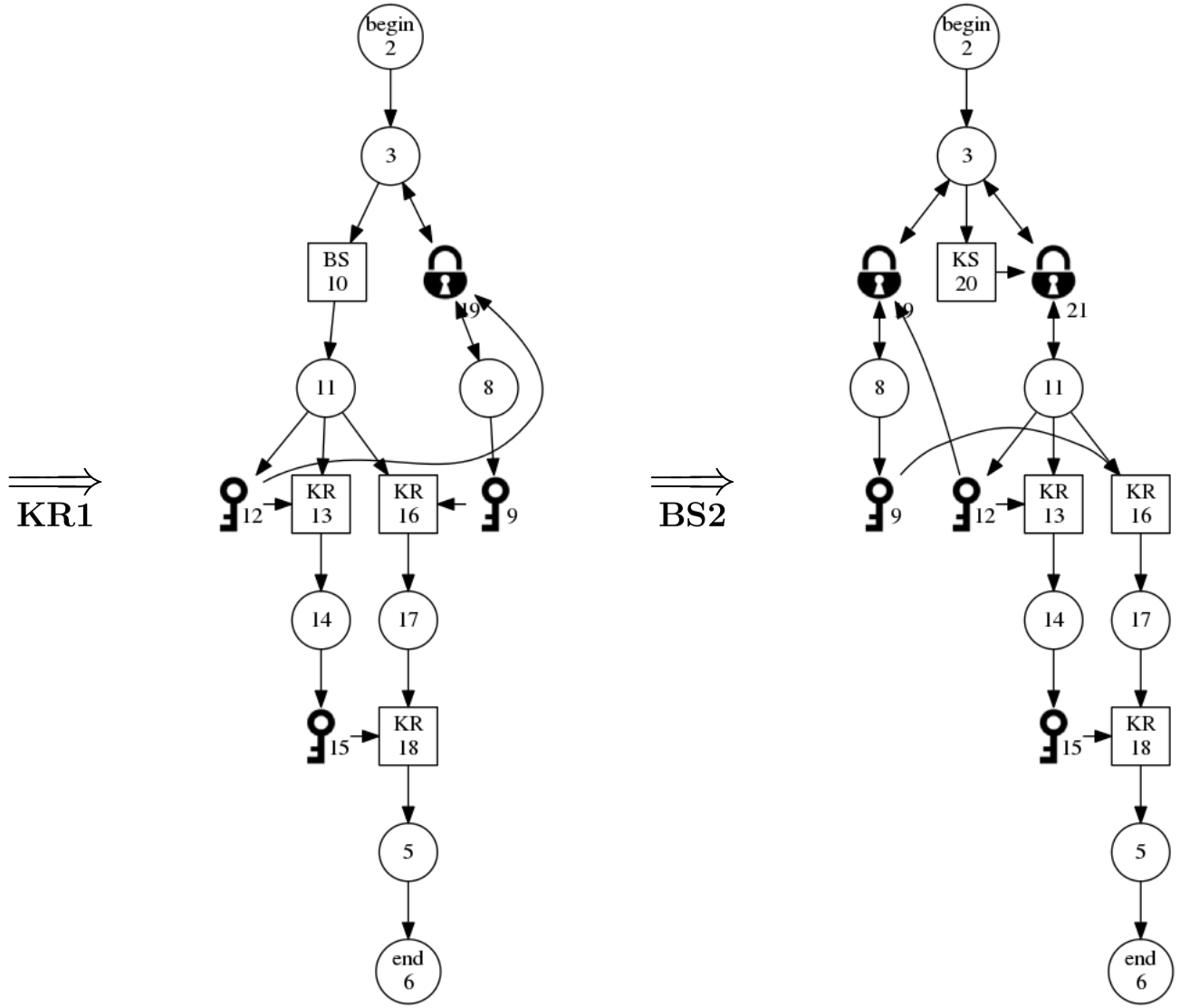


Figure B.12: Part 2 of *Key Cavern* derivation

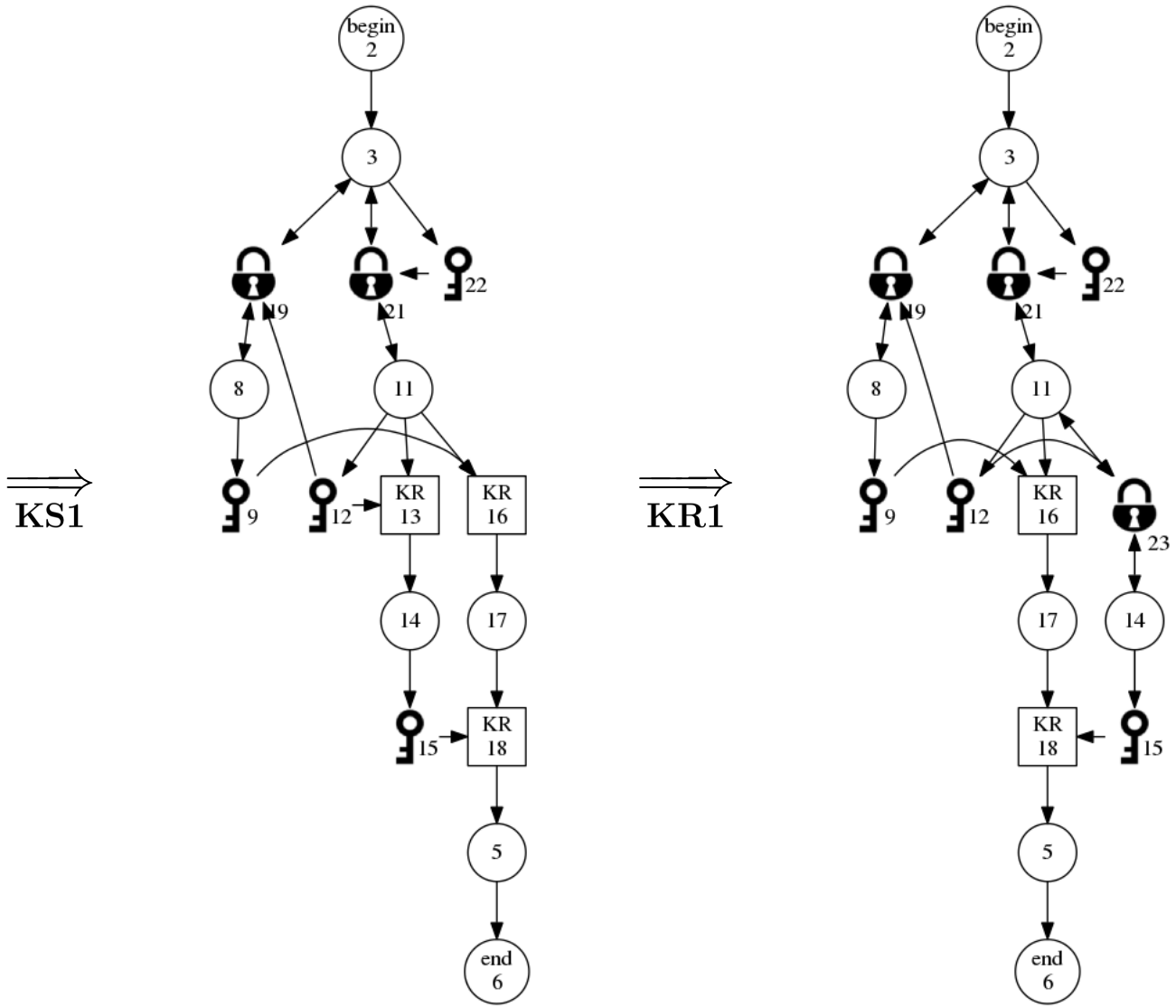


Figure B.12: Part 3 of *Key Cavern* derivation

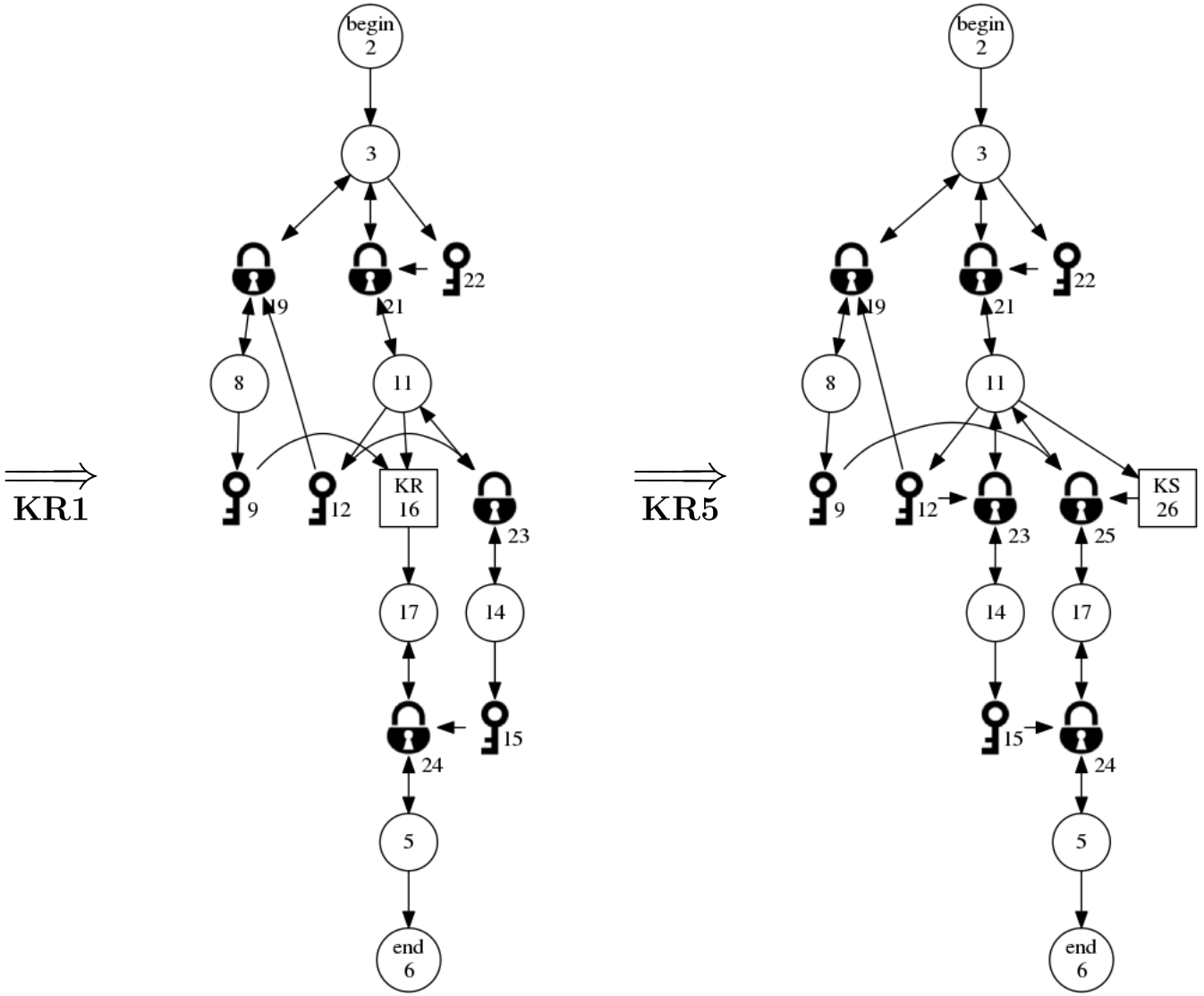


Figure B.12: Part 4 of *Key Cavern* derivation

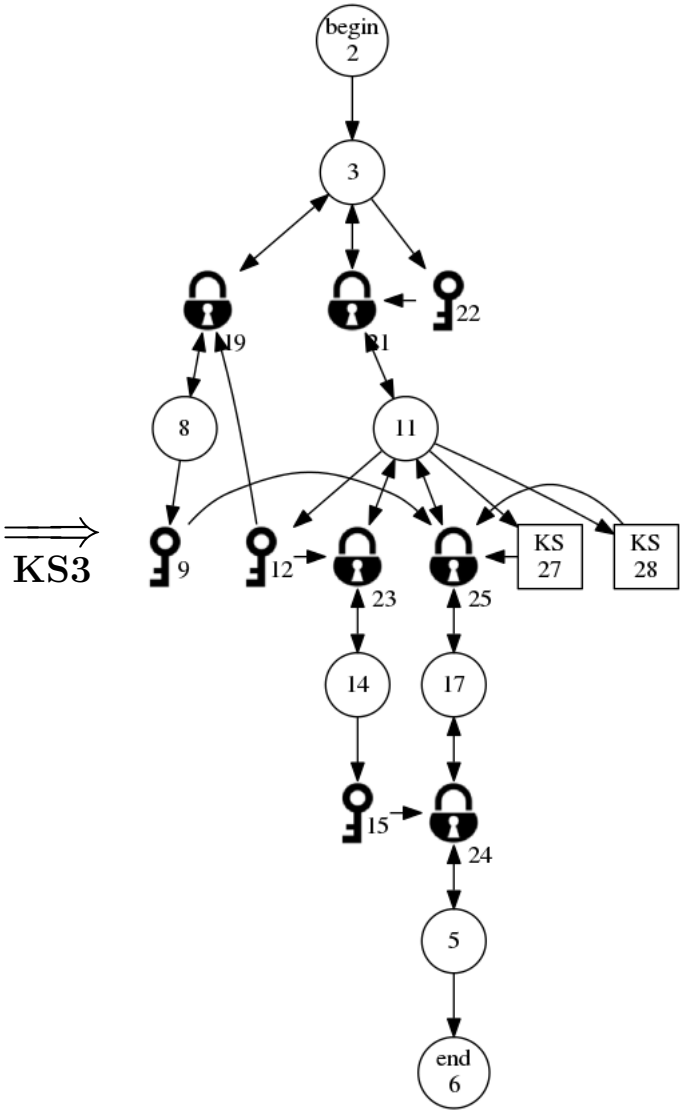


Figure B.12: Part 5 of *Key Cavern* derivation

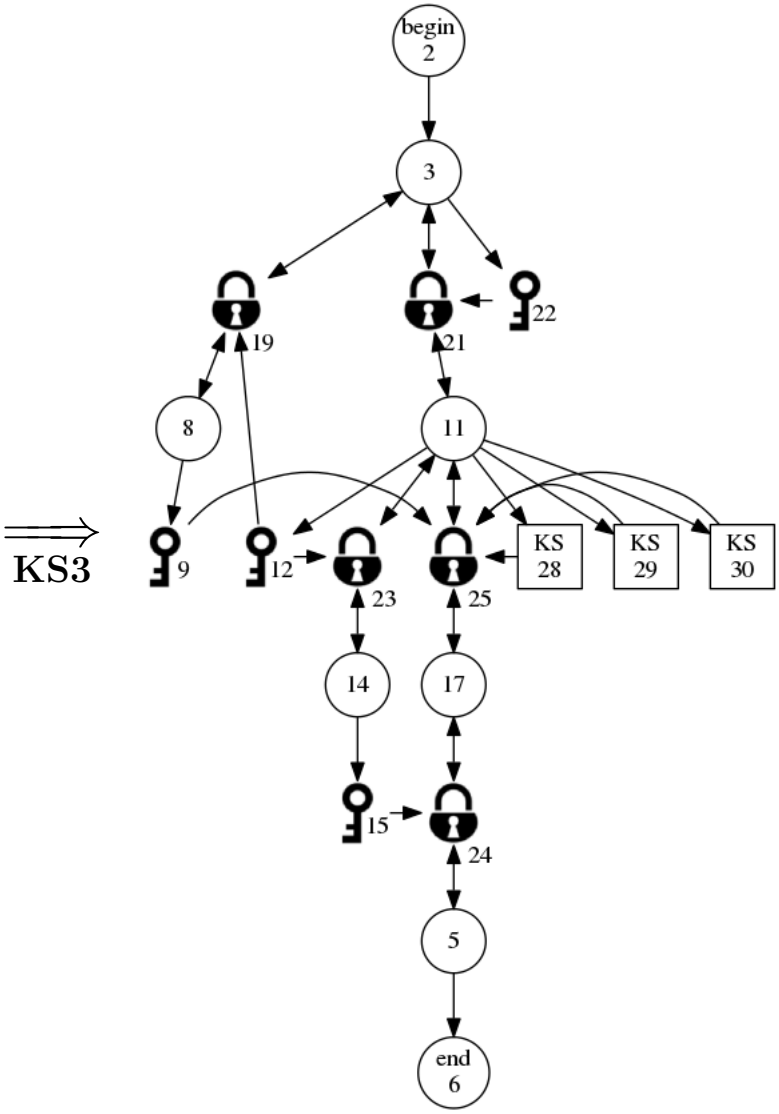


Figure B.12: Part 6 of *Key Cavern* derivation

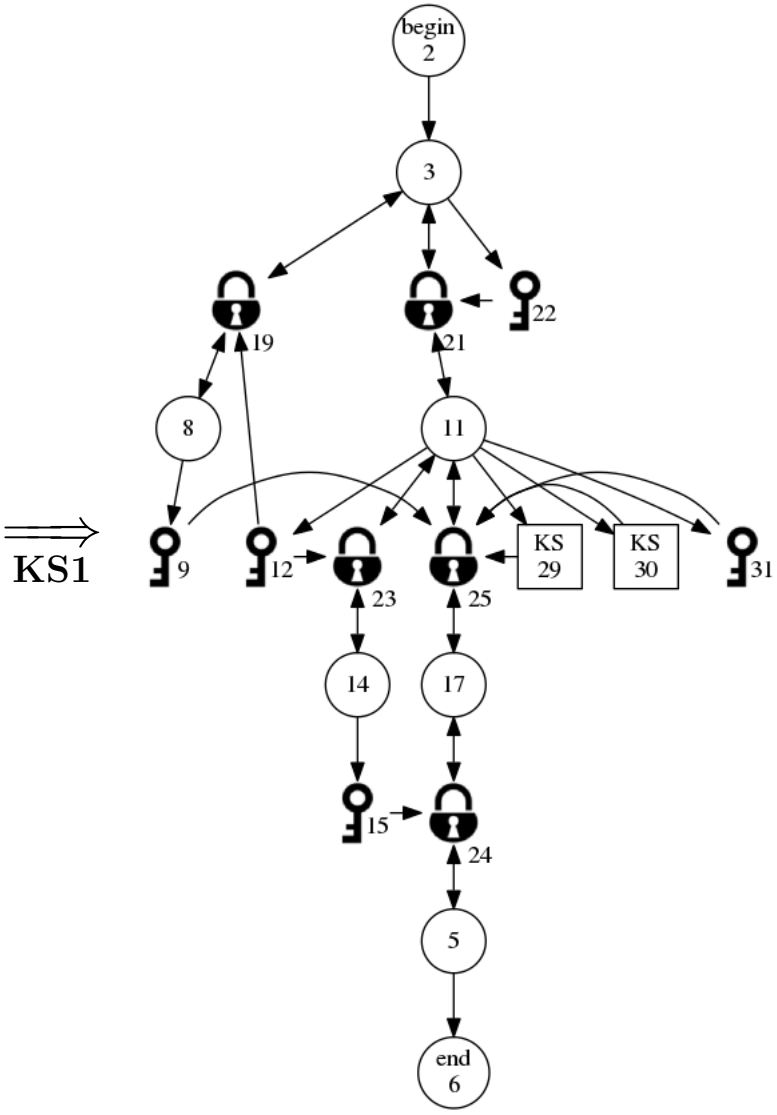


Figure B.12: Part 7 of *Key Cavern* derivation

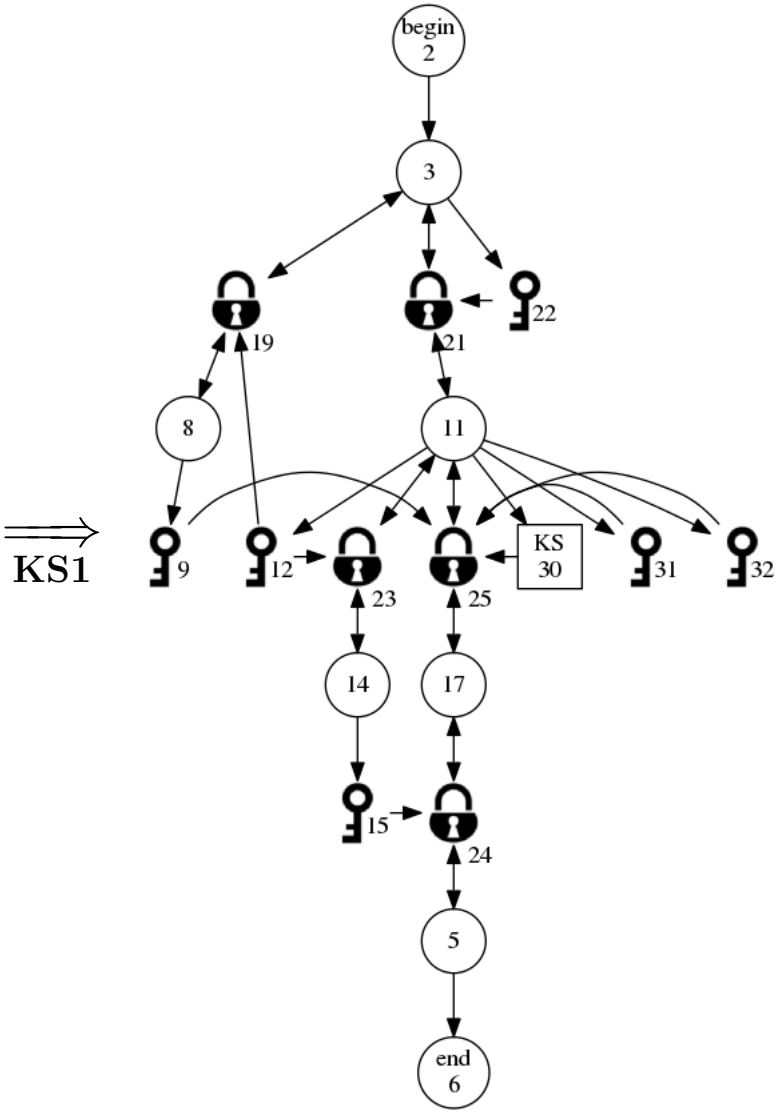


Figure B.12: Part 8 of *Key Cavern* derivation

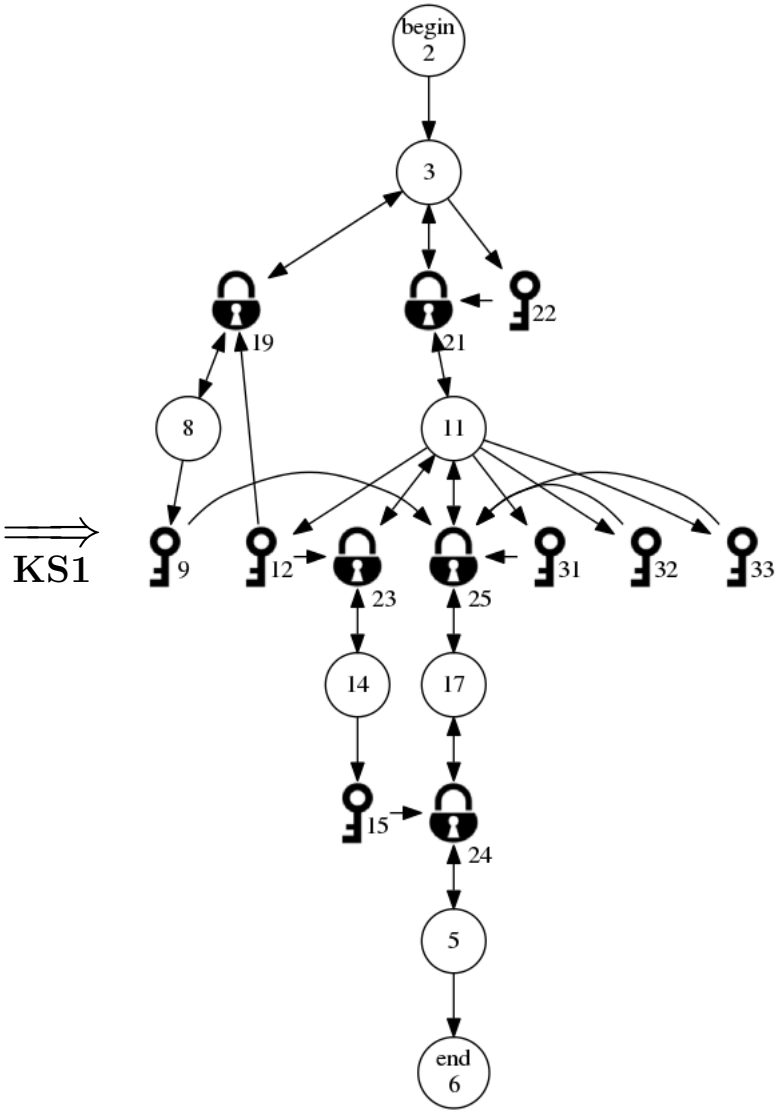


Figure B.12: Part 9 of *Key Cavern* derivation

B.3.4 Angler's Tunnel

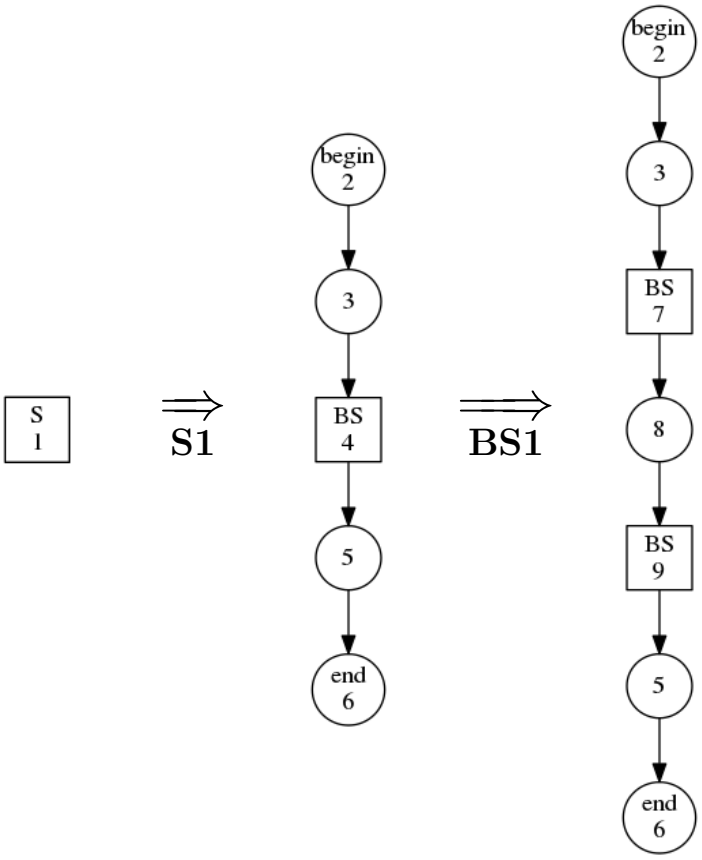


Figure B.13: Part 1 of *Angler's Tunnel* derivation

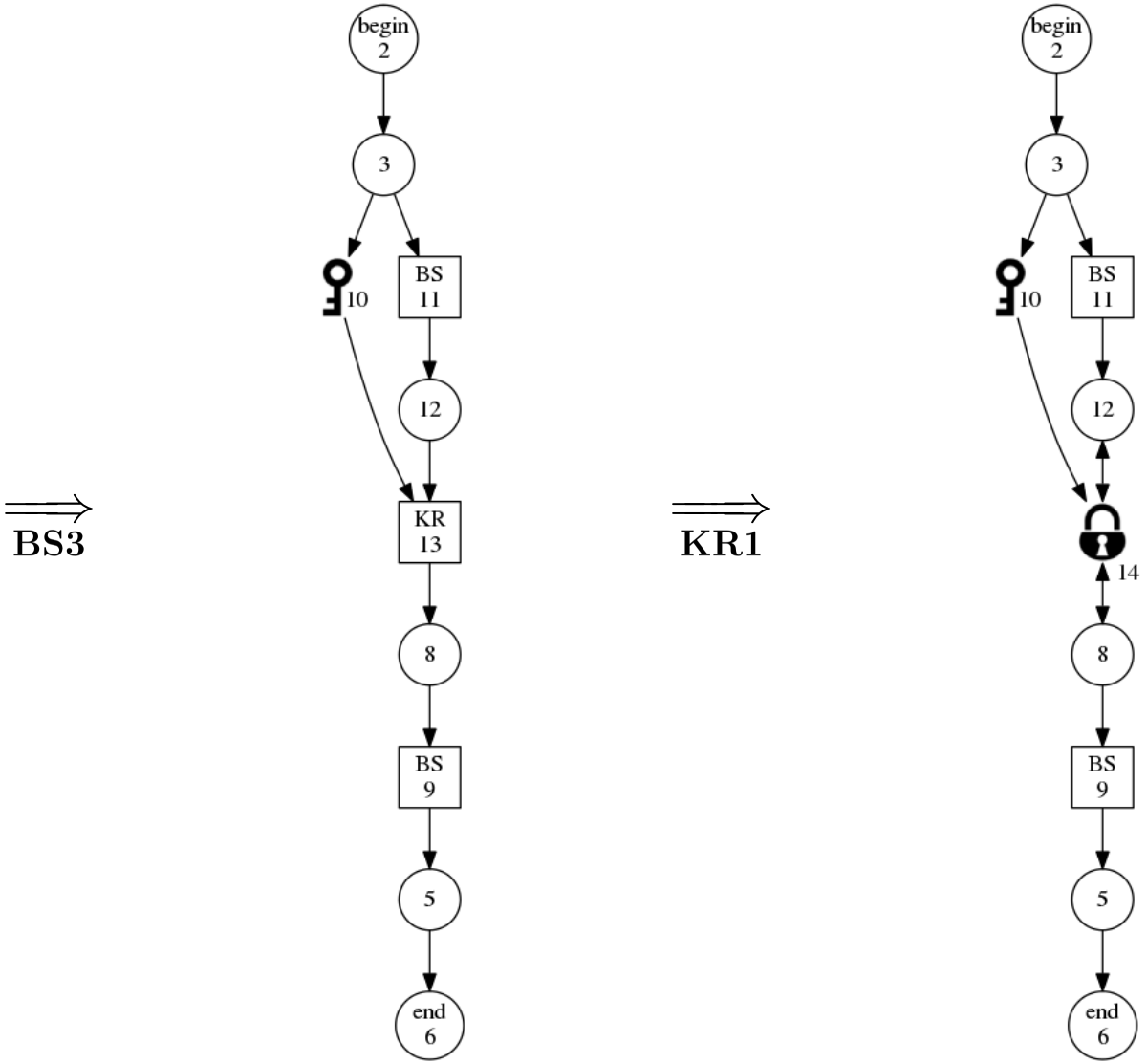


Figure B.13: Part 2 of *Angler's Tunnel* derivation

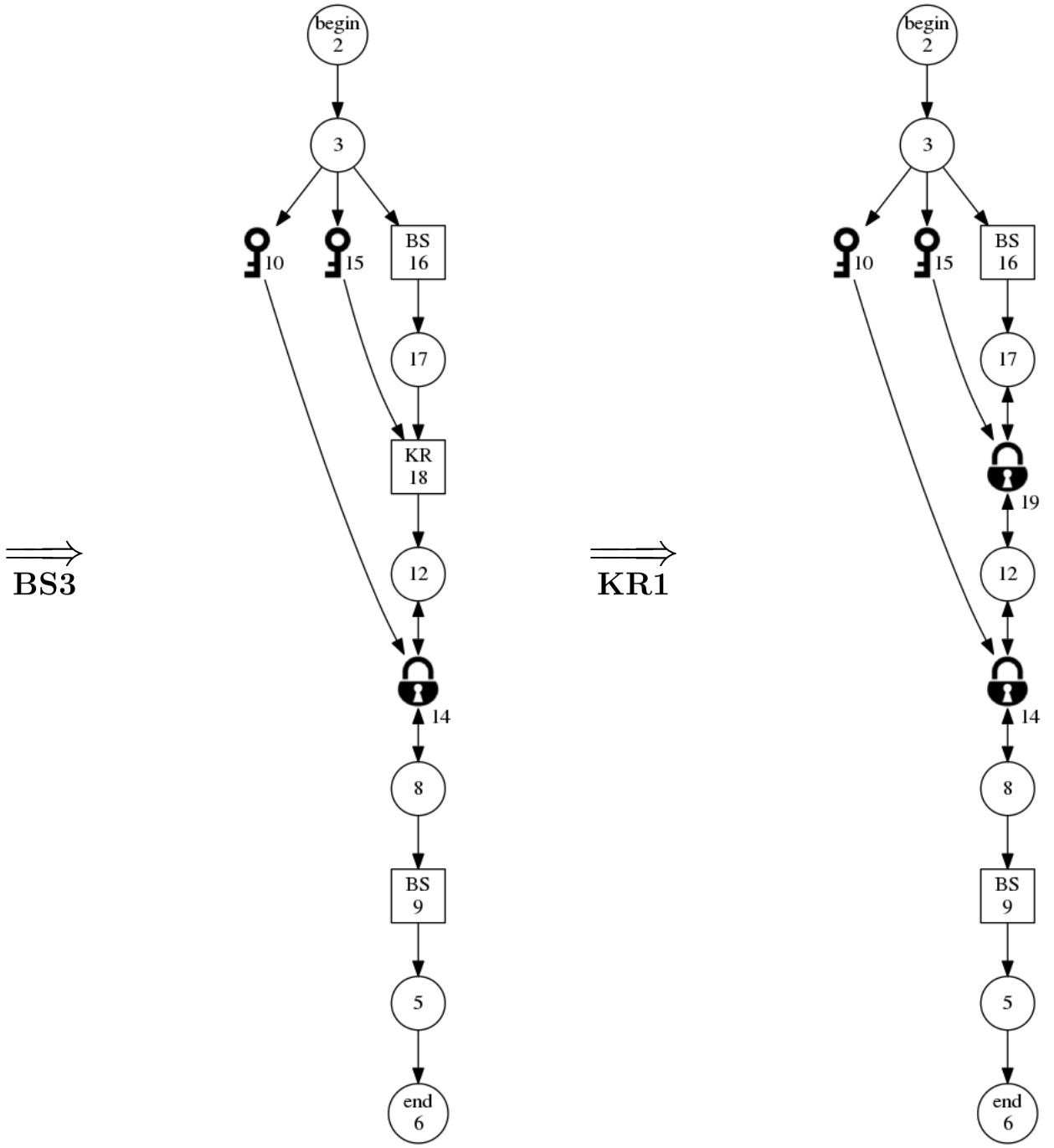


Figure B.13: Part 3 of *Angler's Tunnel* derivation

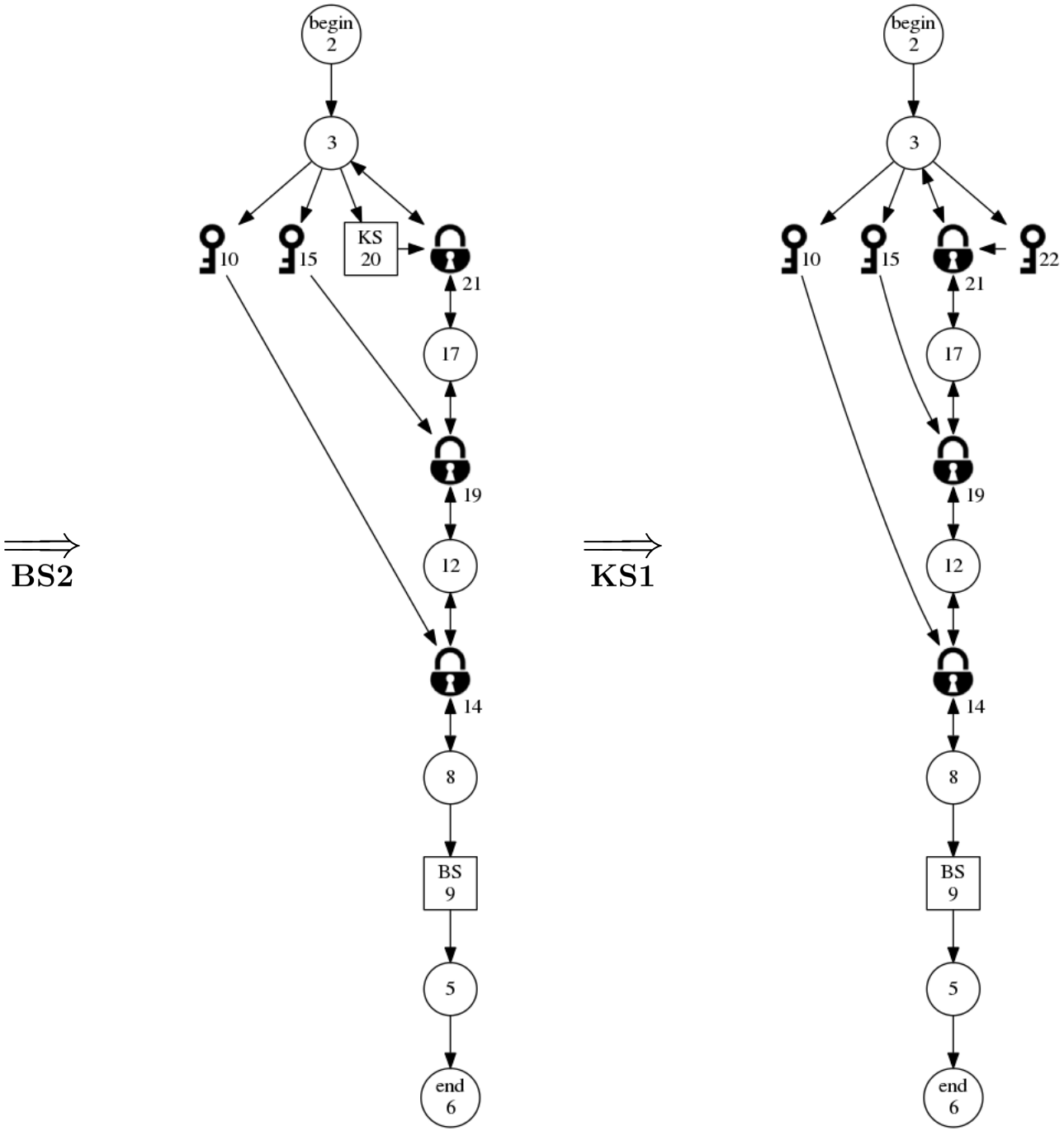


Figure B.13: Part 4 of *Angler's Tunnel* derivation

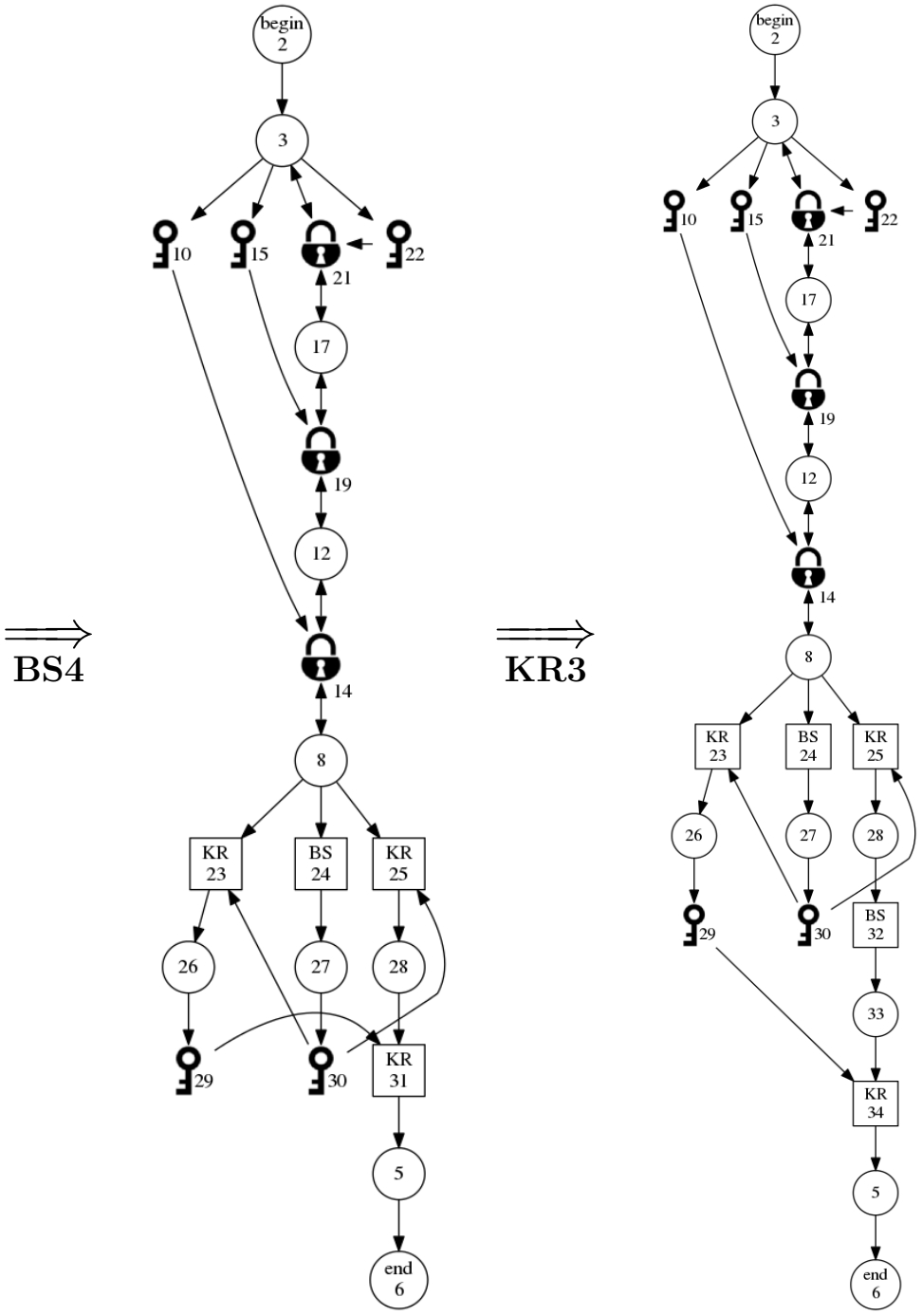


Figure B.13: Part 5 of *Angler's Tunnel* derivation

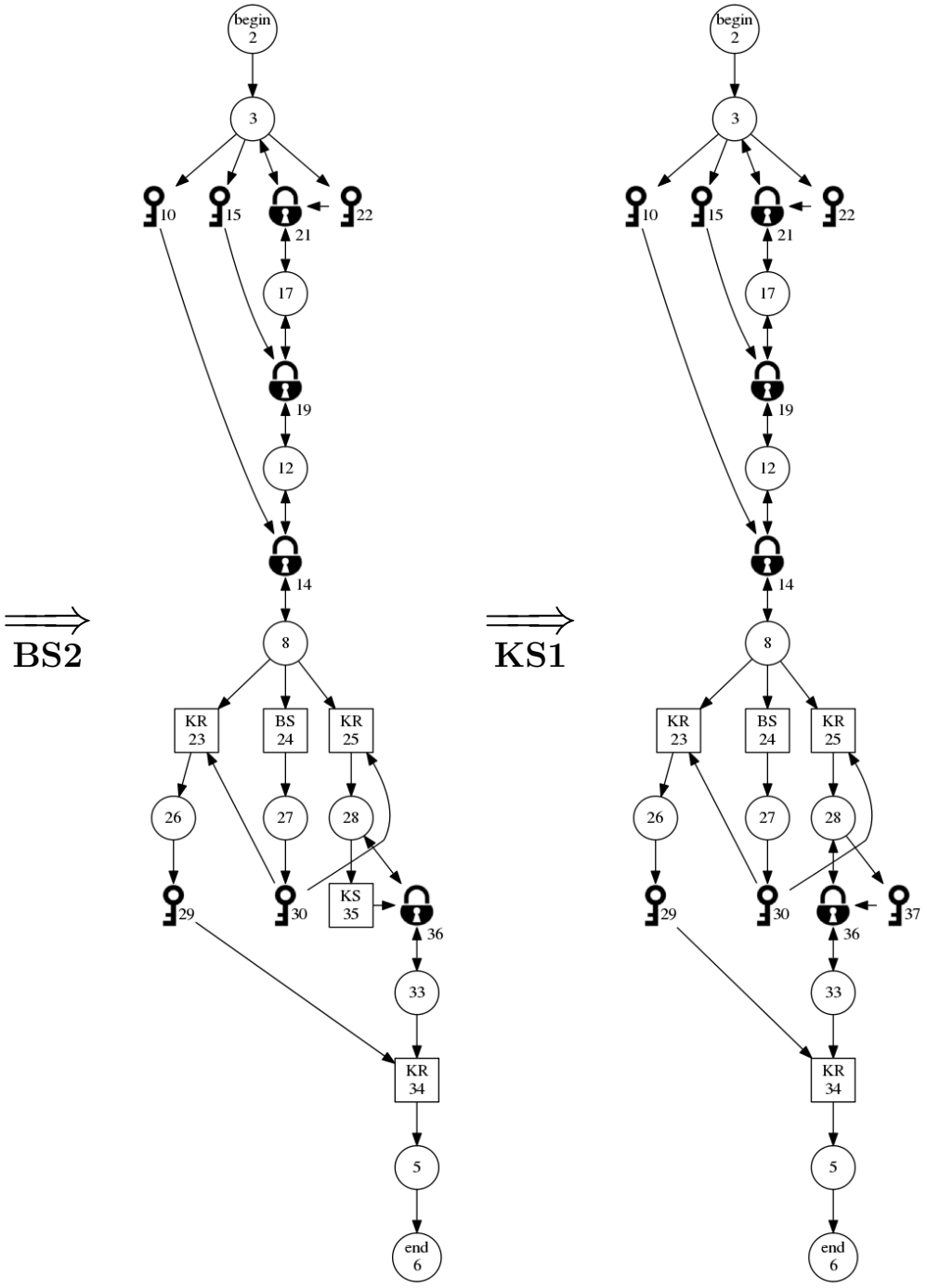


Figure B.13: Part 6 of *Angler's Tunnel* derivation

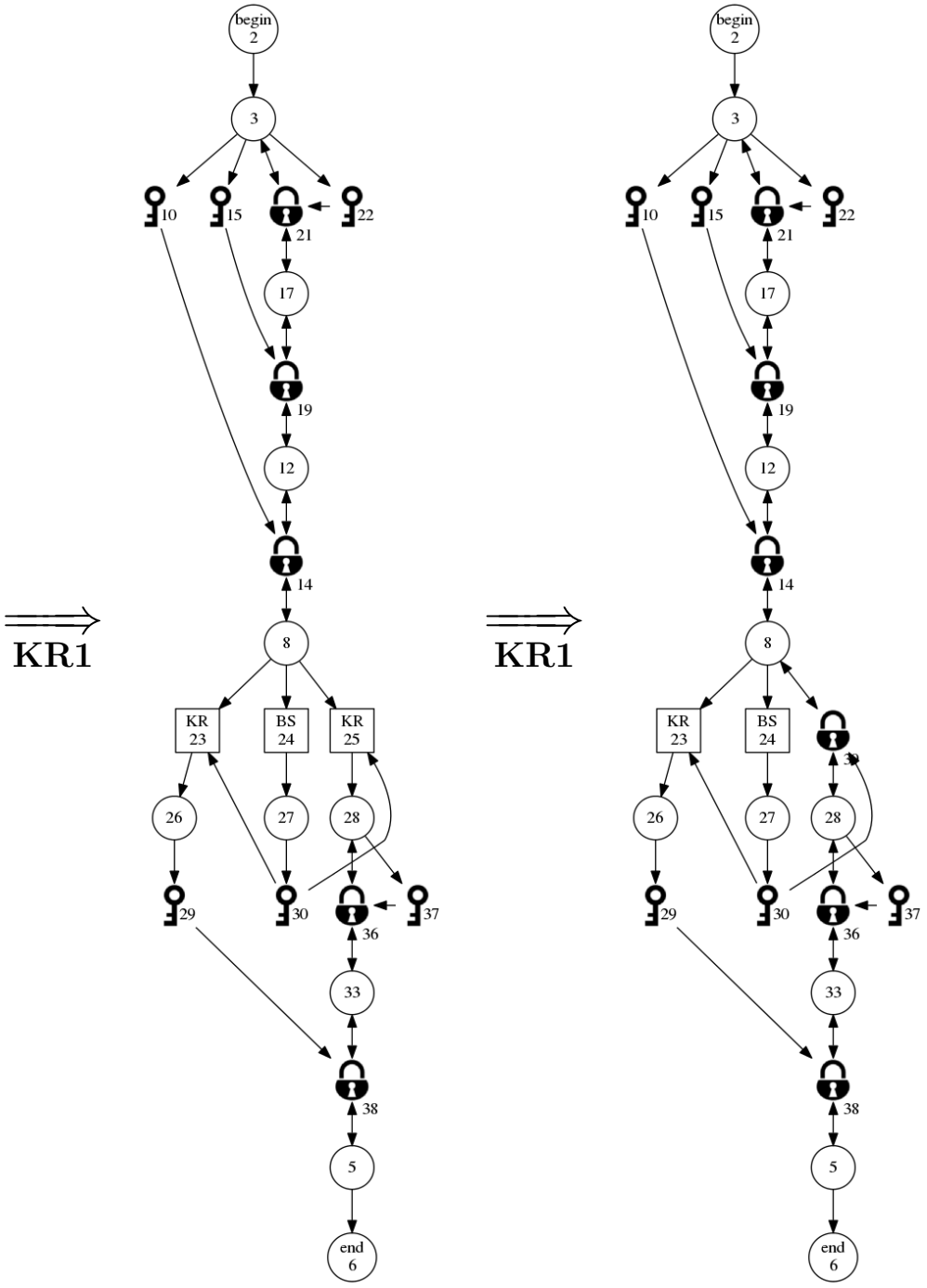


Figure B.13: Part 7 of *Angler's Tunnel* derivation

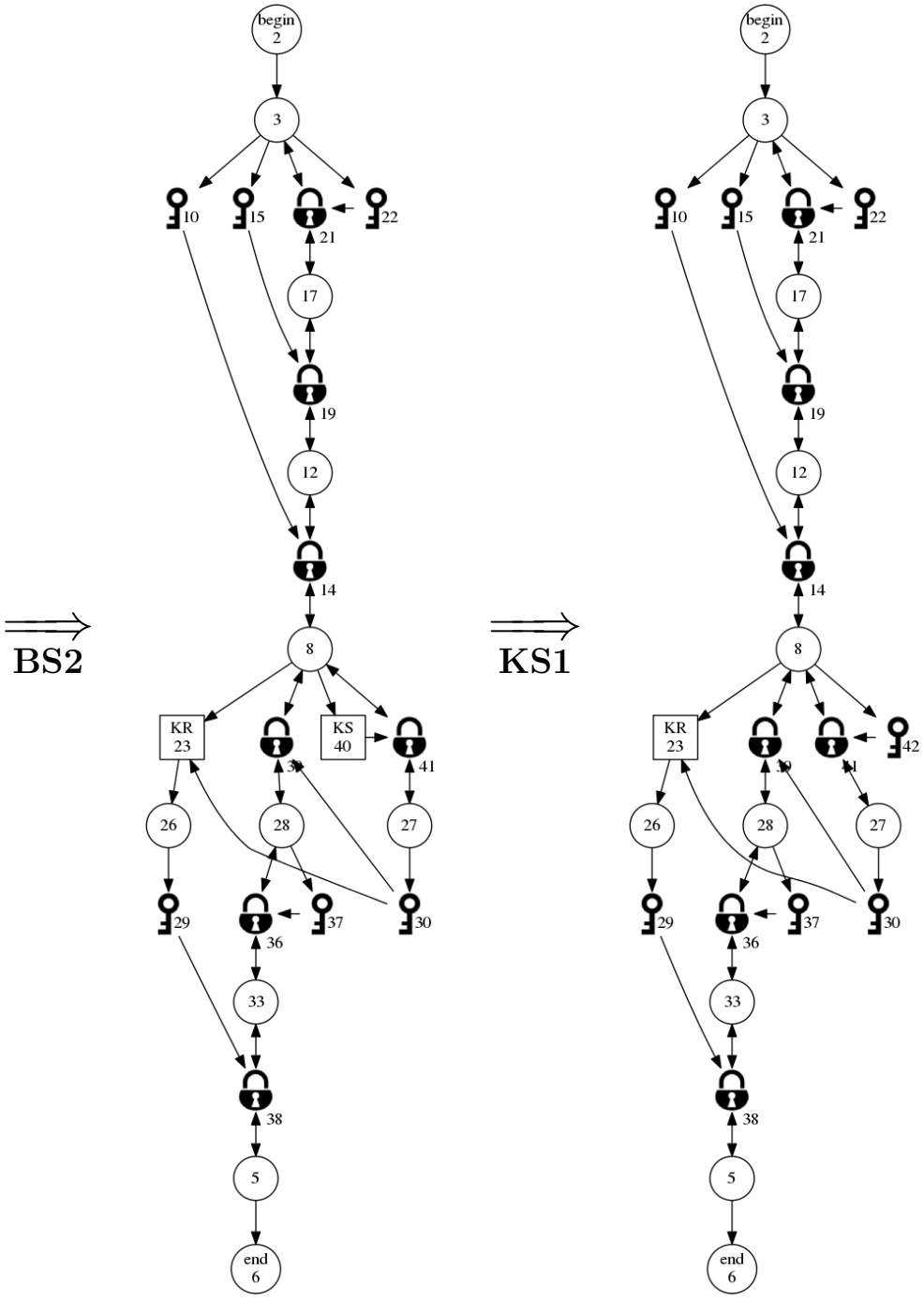


Figure B.13: Part 8 of *Angler's Tunnel* derivation

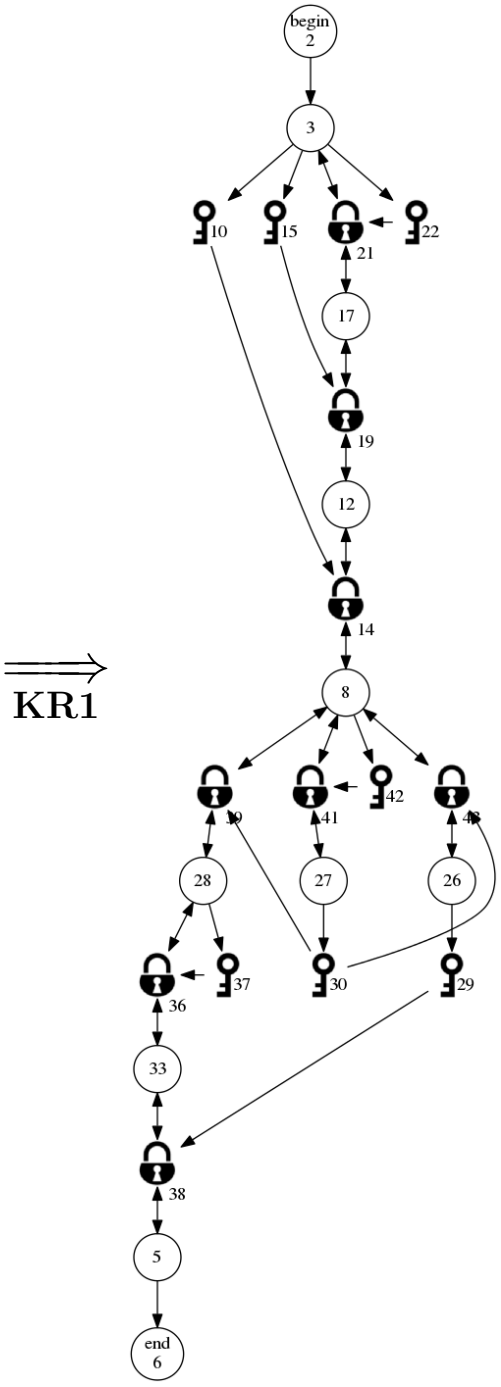


Figure B.13: Part 6 of *Angler's Tunnel* derivation

B.3.5 Catfish's Maw

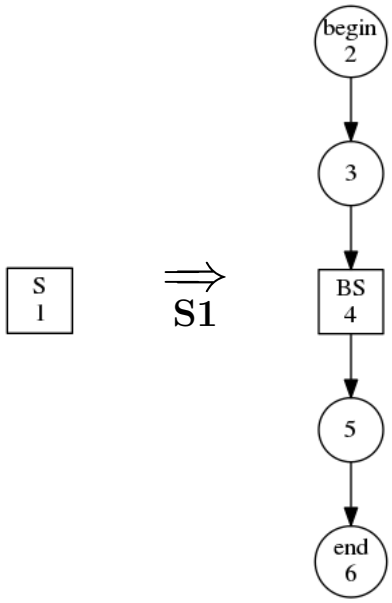


Figure B.14: Part 1 of *Catfish's Maw* derivation

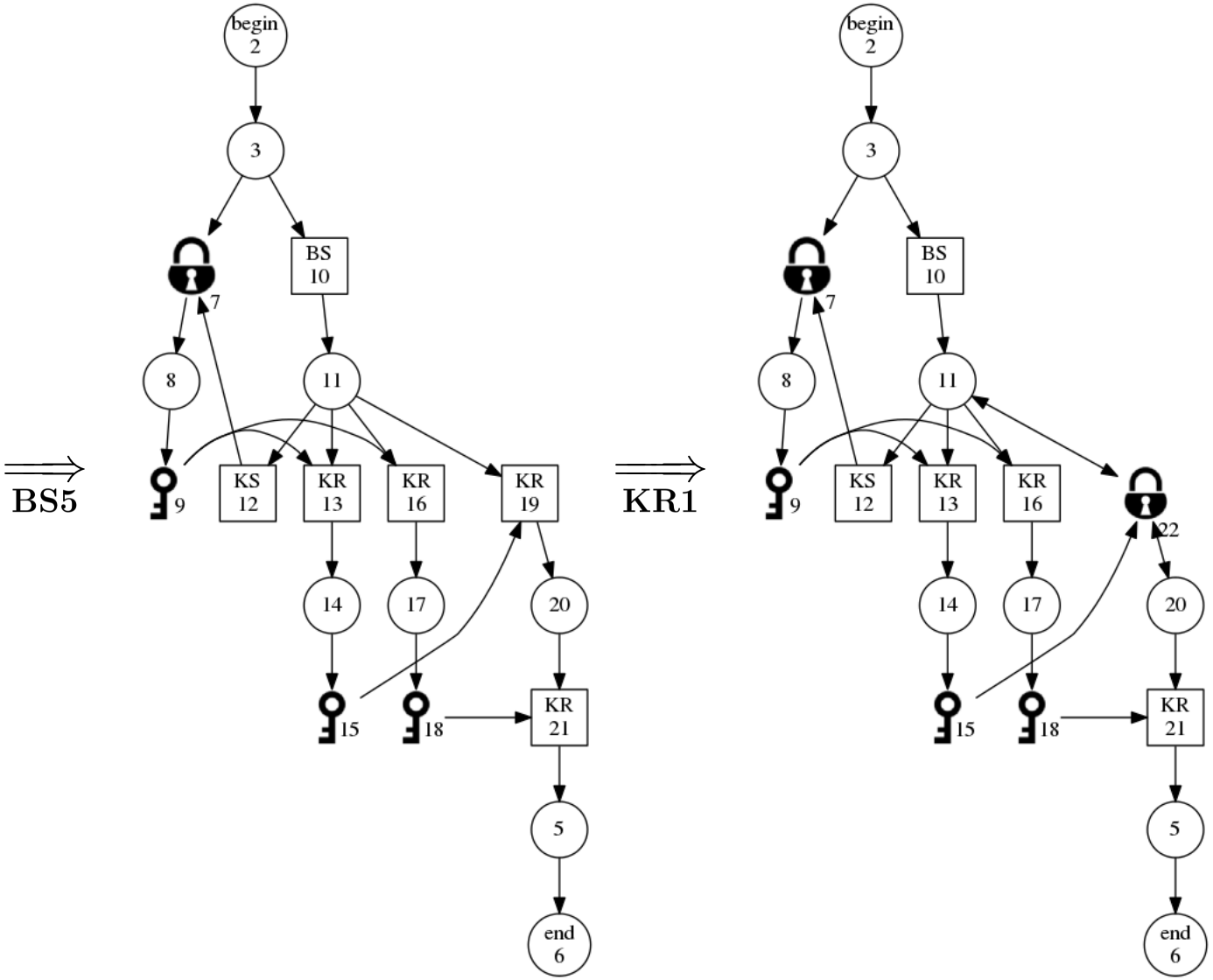


Figure B.14: Part 2 of *Catfish's Maw* derivation

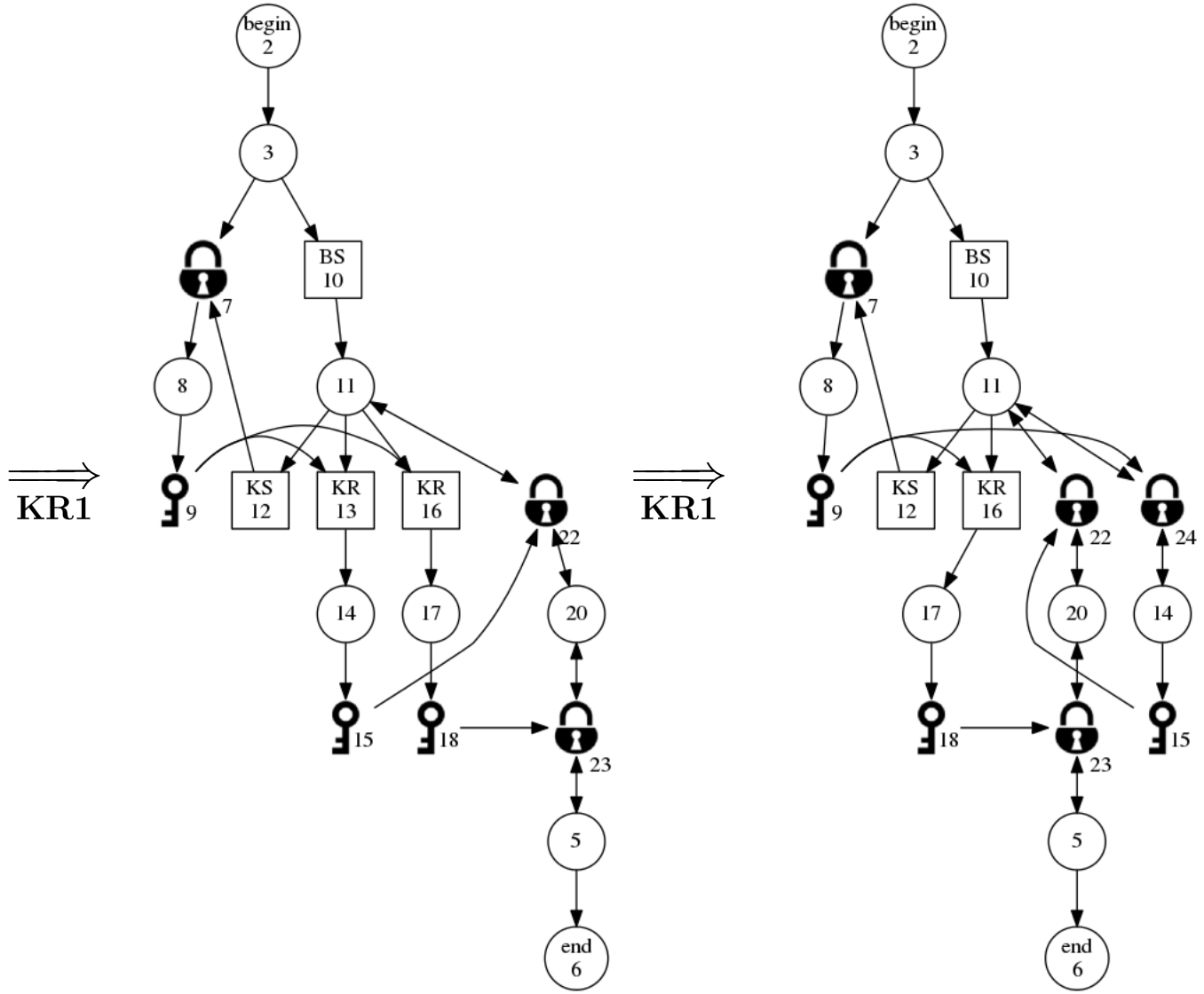


Figure B.14: Part 3 of *Catfish's Maw* derivation

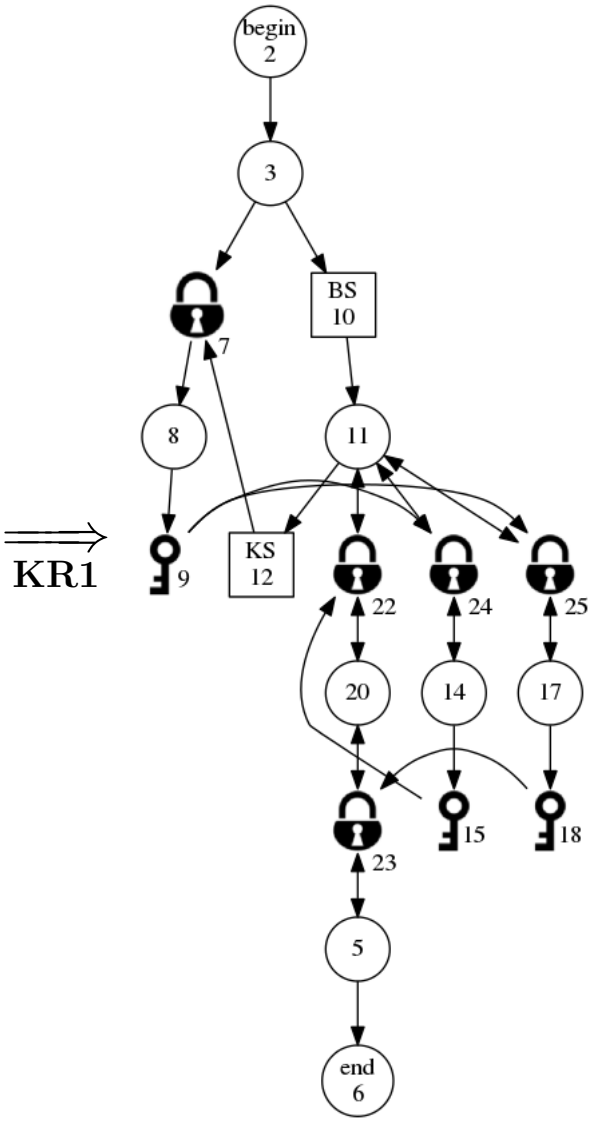


Figure B.14: Part 4 of *Catfish's Maw* derivation

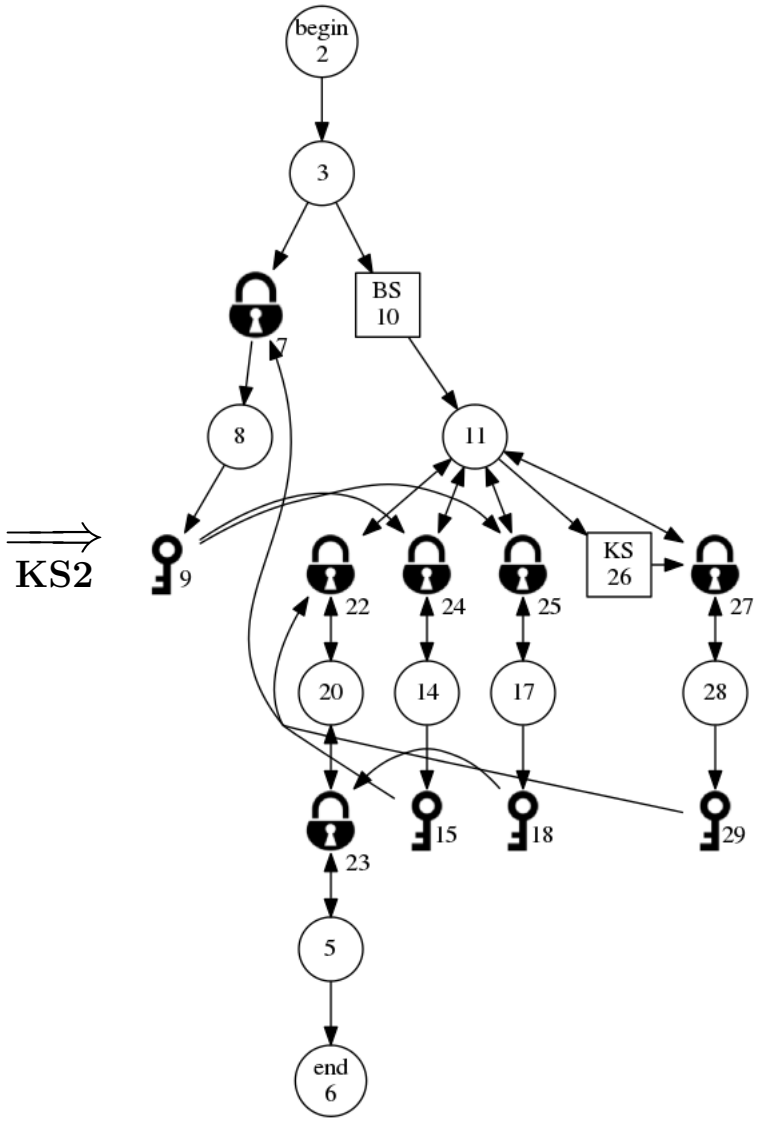


Figure B.14: Part 5 of *Catfish's Maw* derivation

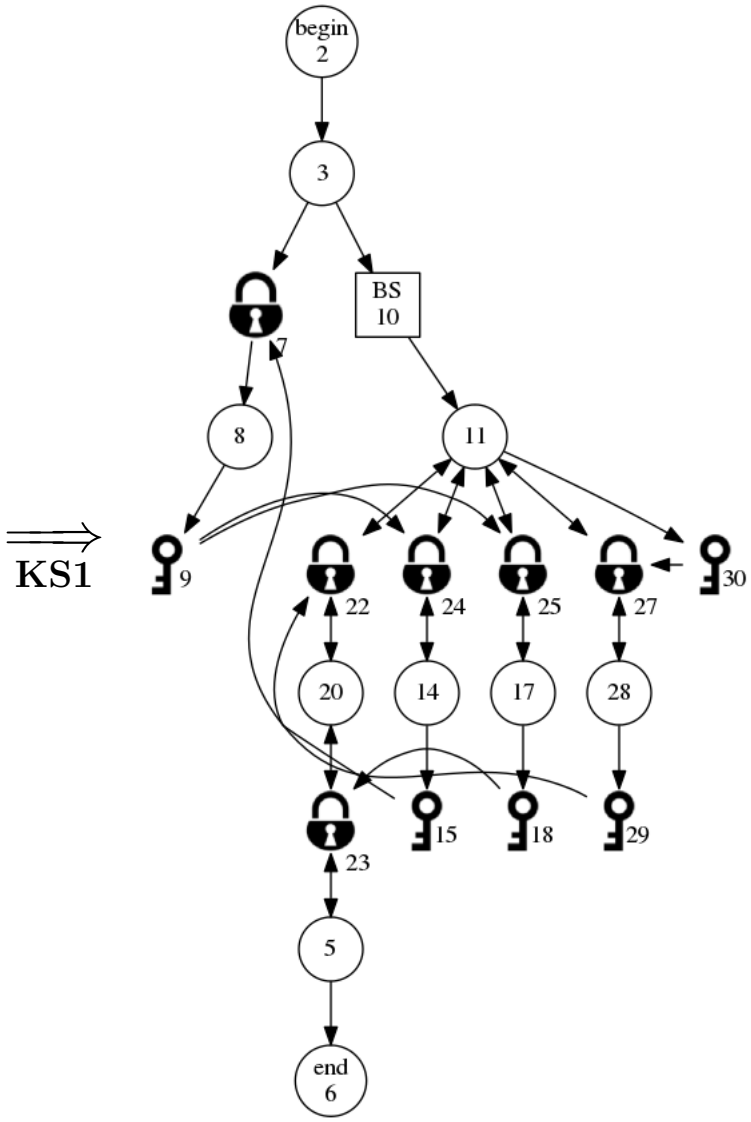


Figure B.14: Part 6 of *Catfish's Maw* derivation

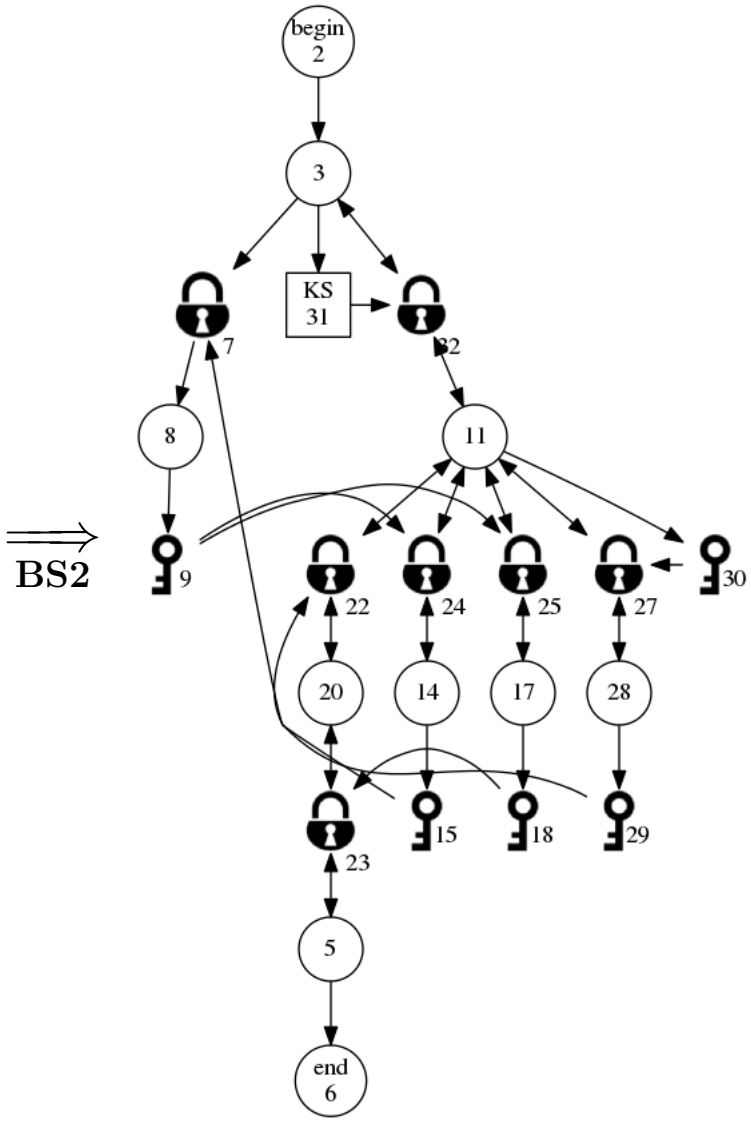


Figure B.14: Part 7 of *Catfish's Maw* derivation

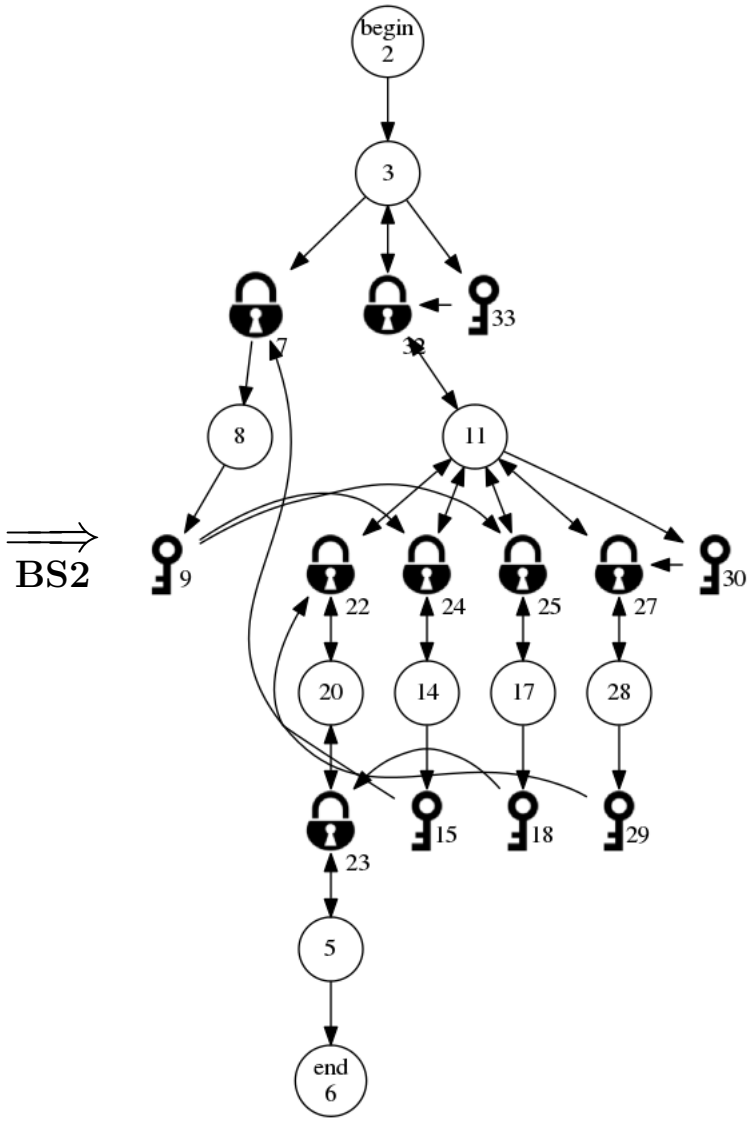


Figure B.14: Part 8 of *Catfish's Maw* derivation

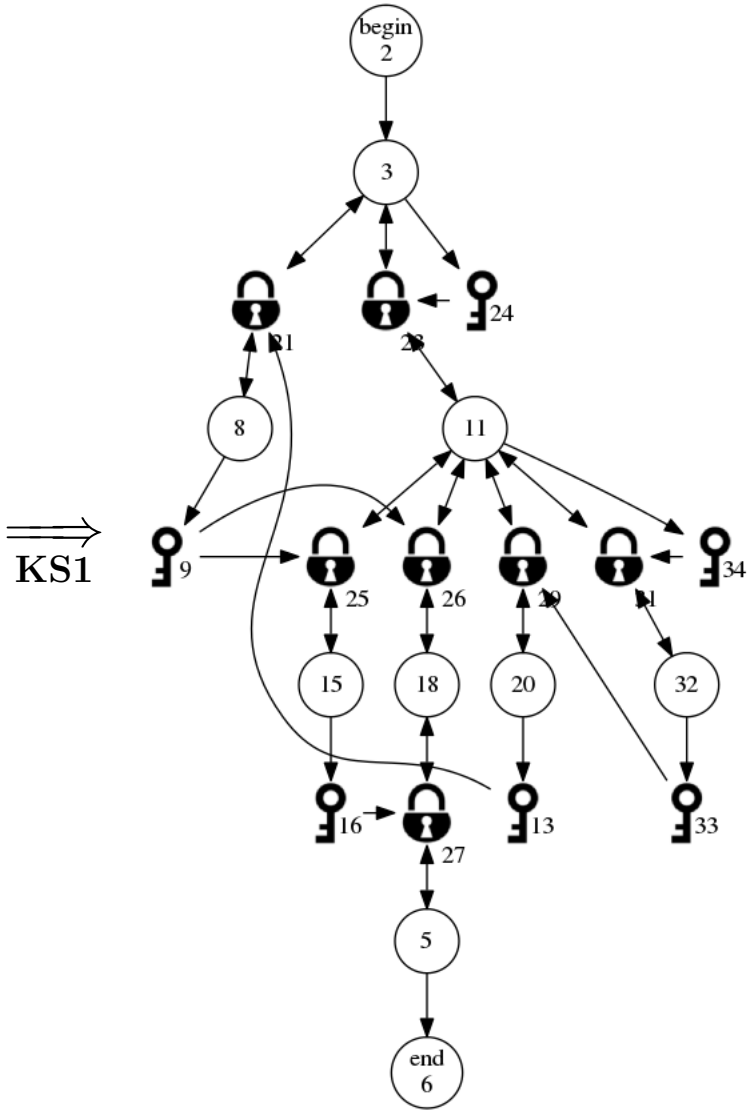


Figure B.14: Part 9 of *Catfish's Maw* derivation

B.4 Legend of Zelda: Ocarina of Time

B.4.1 Inside the Great Deku Tree

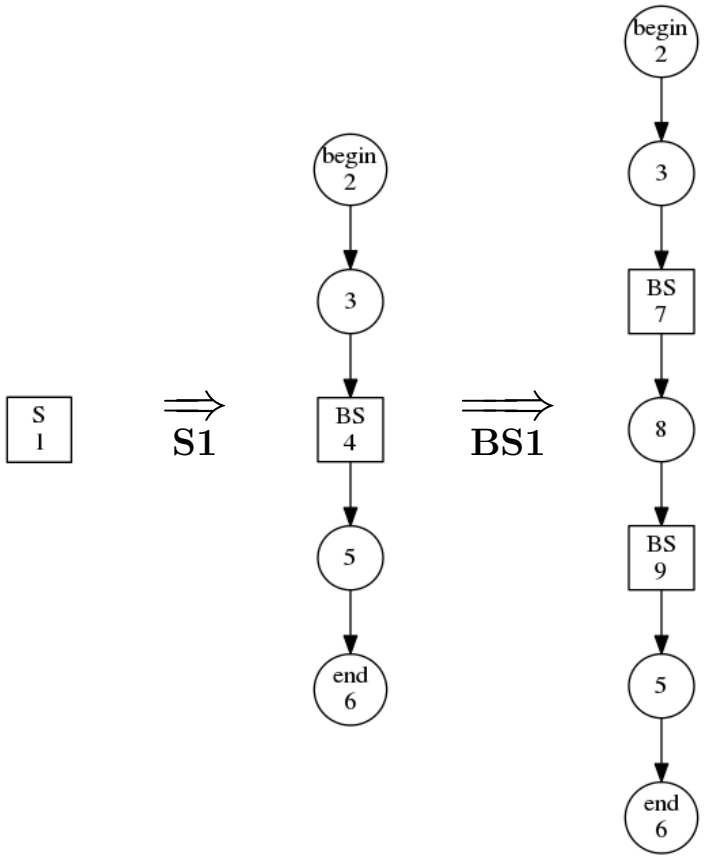


Figure B.15: Part 1 of *Inside the Deku Tree* derivation

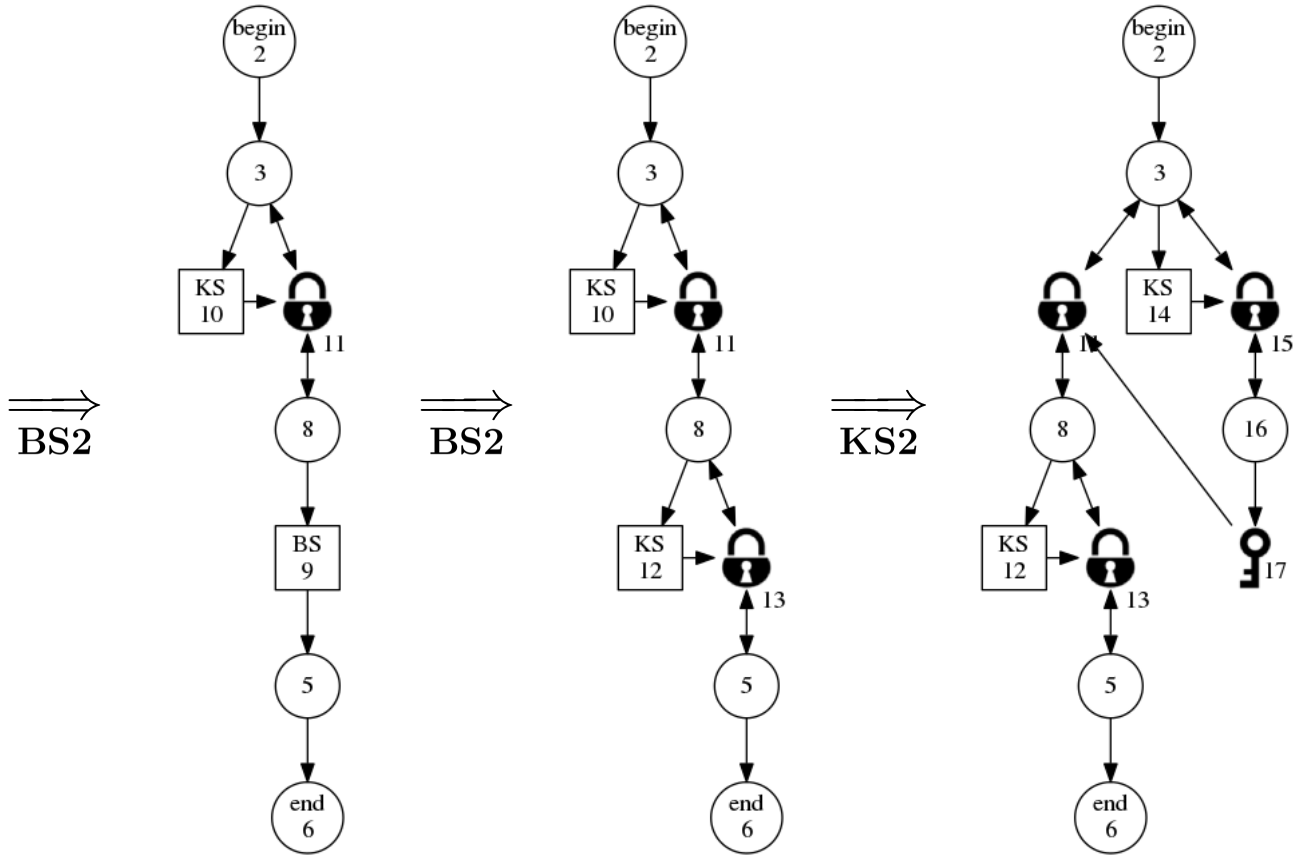


Figure B.15: Part 2 of *Inside the Great Deku Tree* derivation

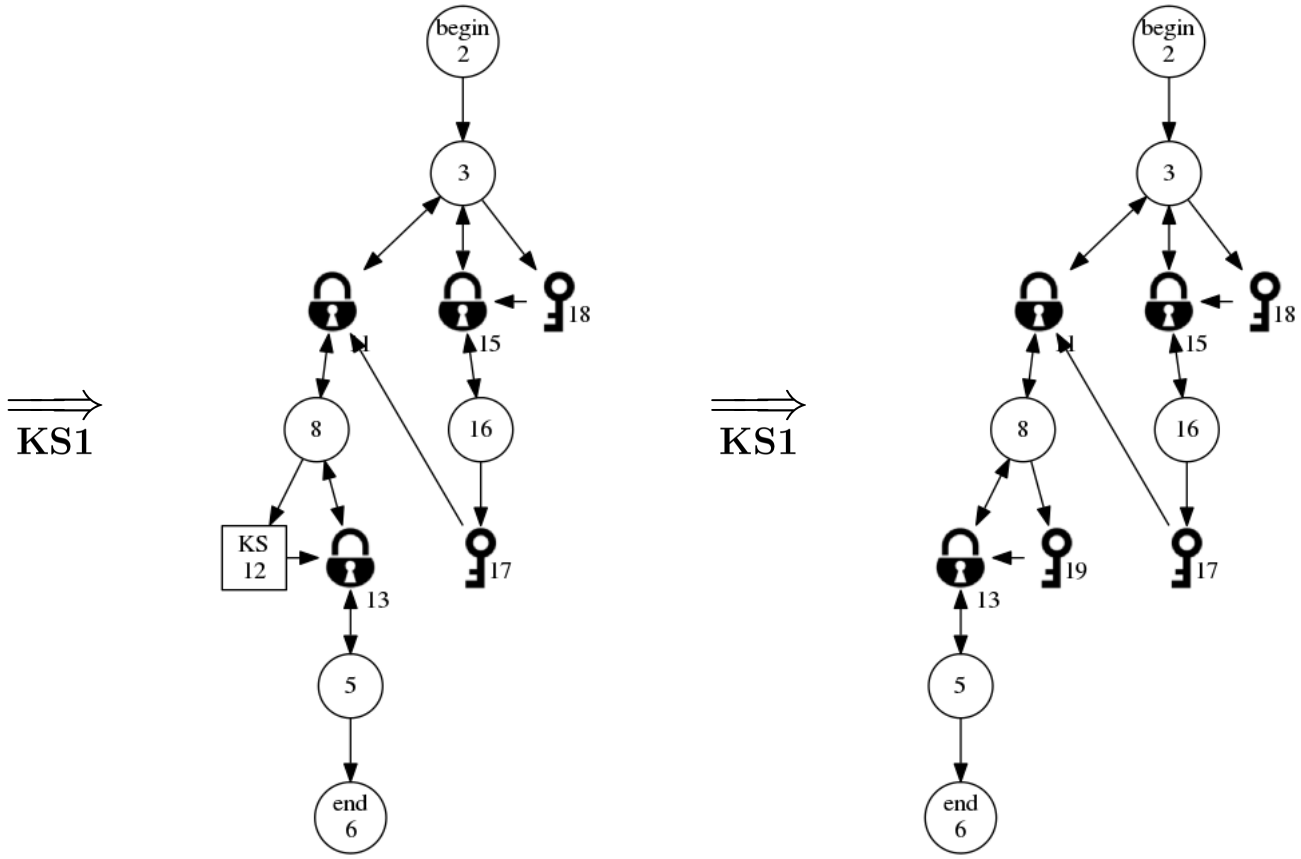


Figure B.15: Part 3 of *Inside the Great Deku Tree* derivation

B.4.2 Dodongo's Cavern

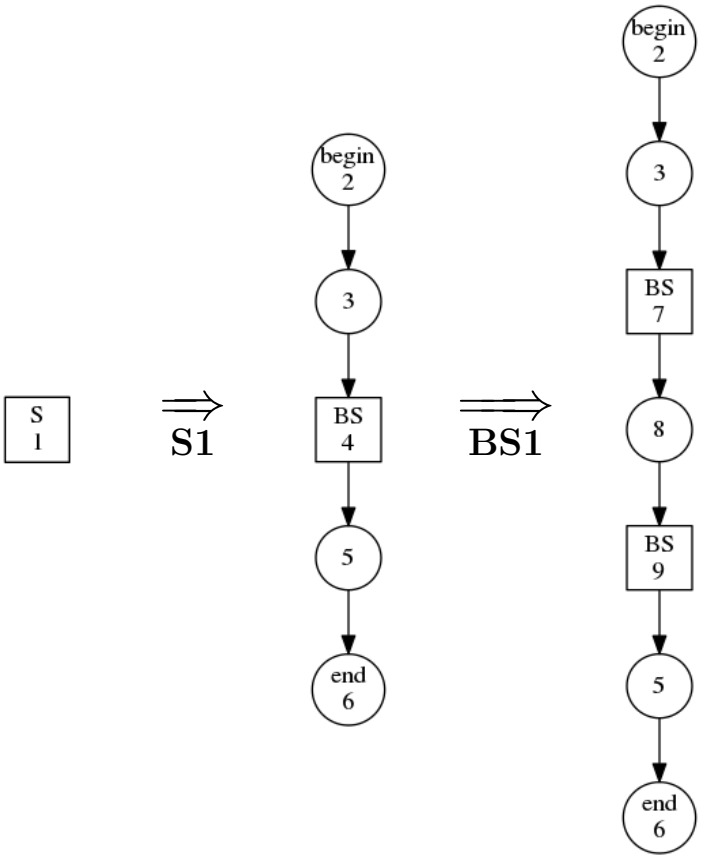


Figure B.16: Part 1 of *Dodongo's Cavern* derivation

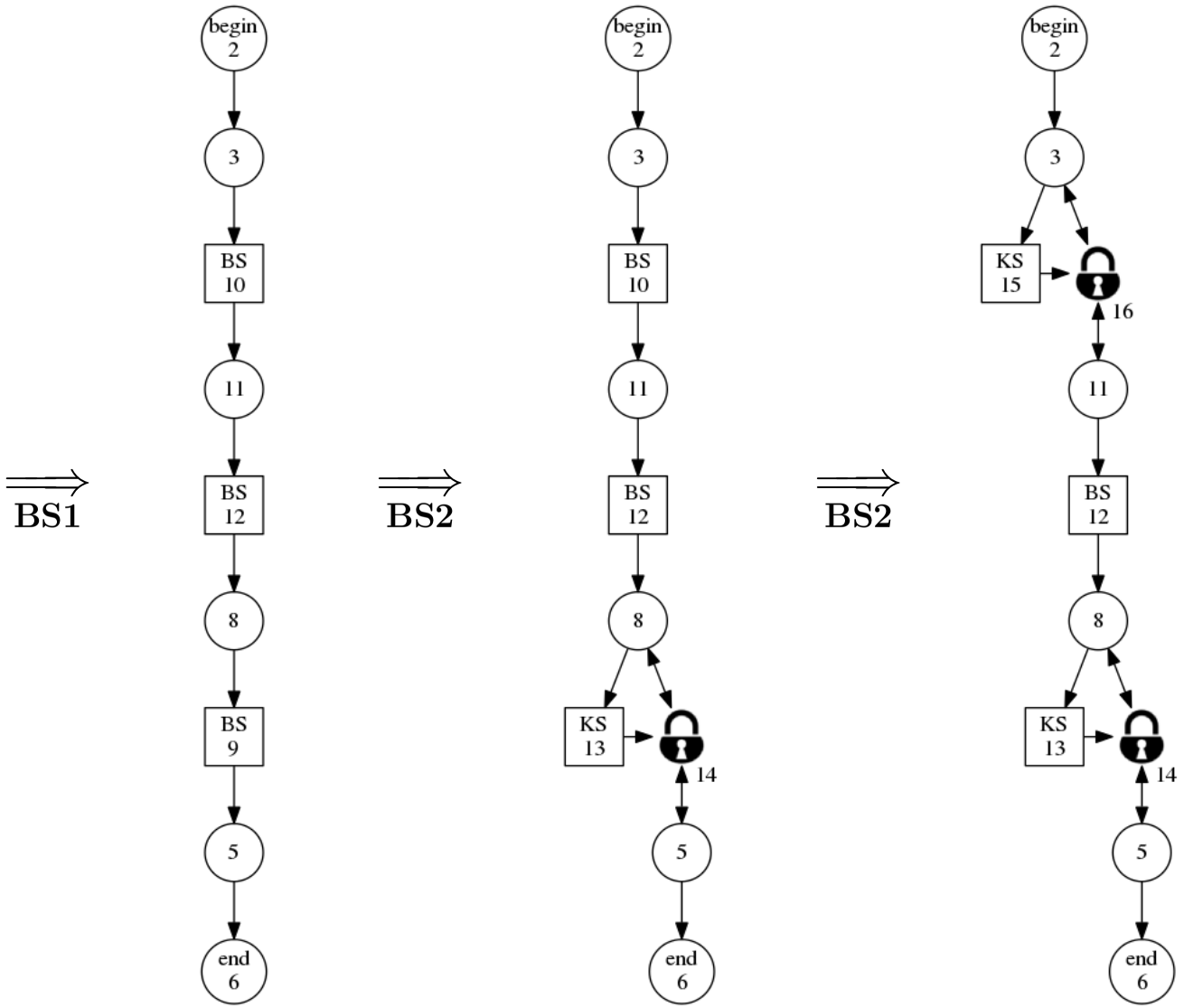


Figure B.16: Part 2 of *Dodongo's Cavern* derivation

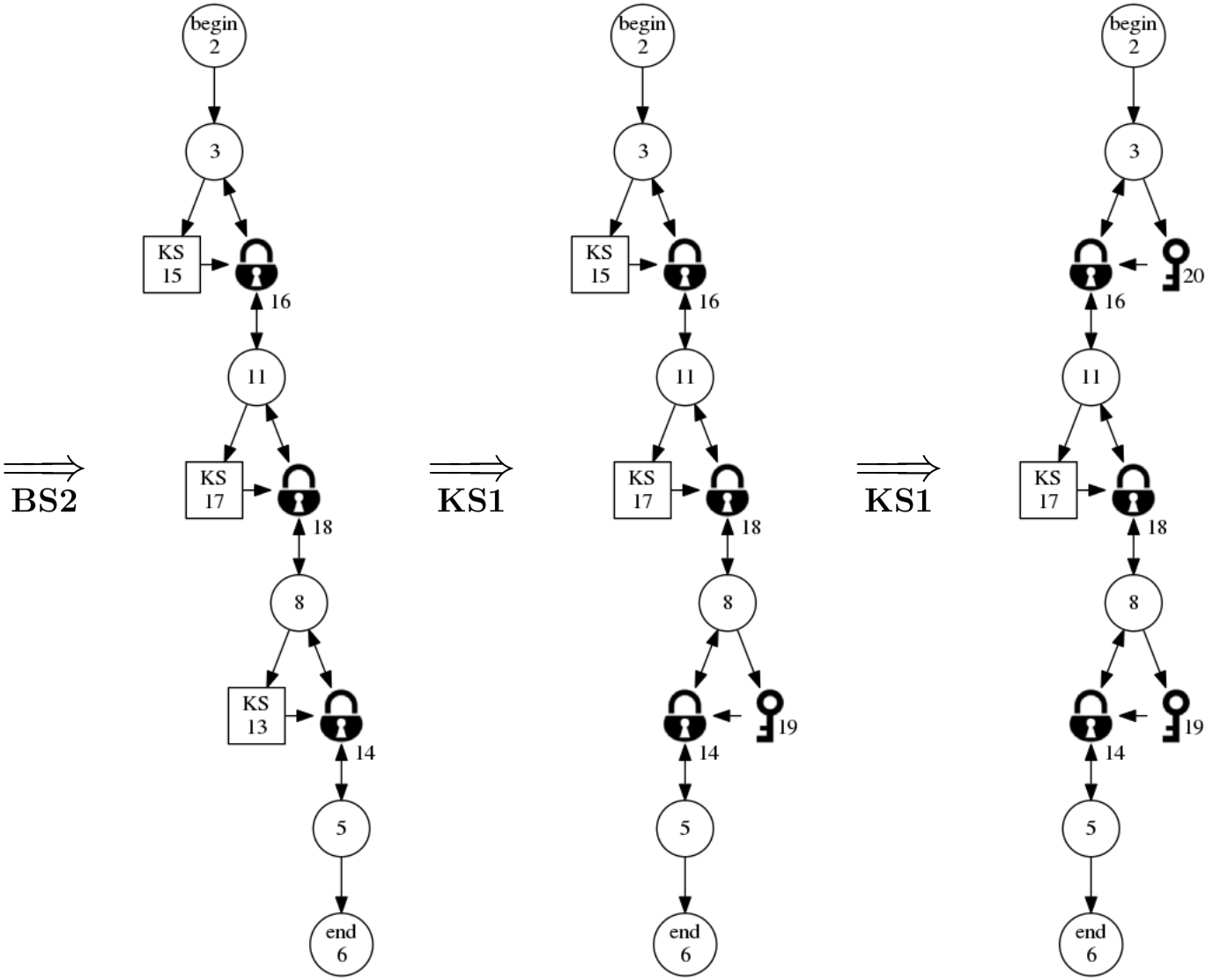


Figure B.16: Part 3 of *Dodongo's Cavern* derivation

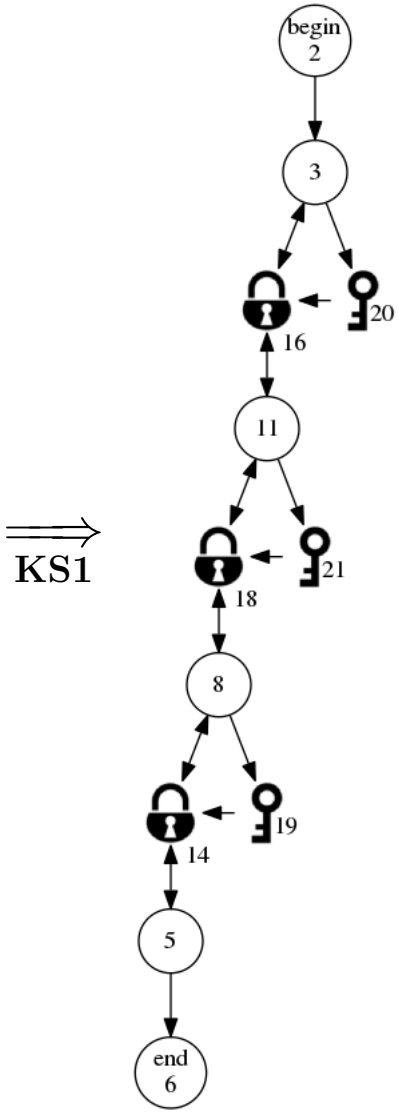


Figure B.16: Part 4 of *Dodongo's Cavern* derivation

B.4.3 Inside Jabu-Jabu's Belly

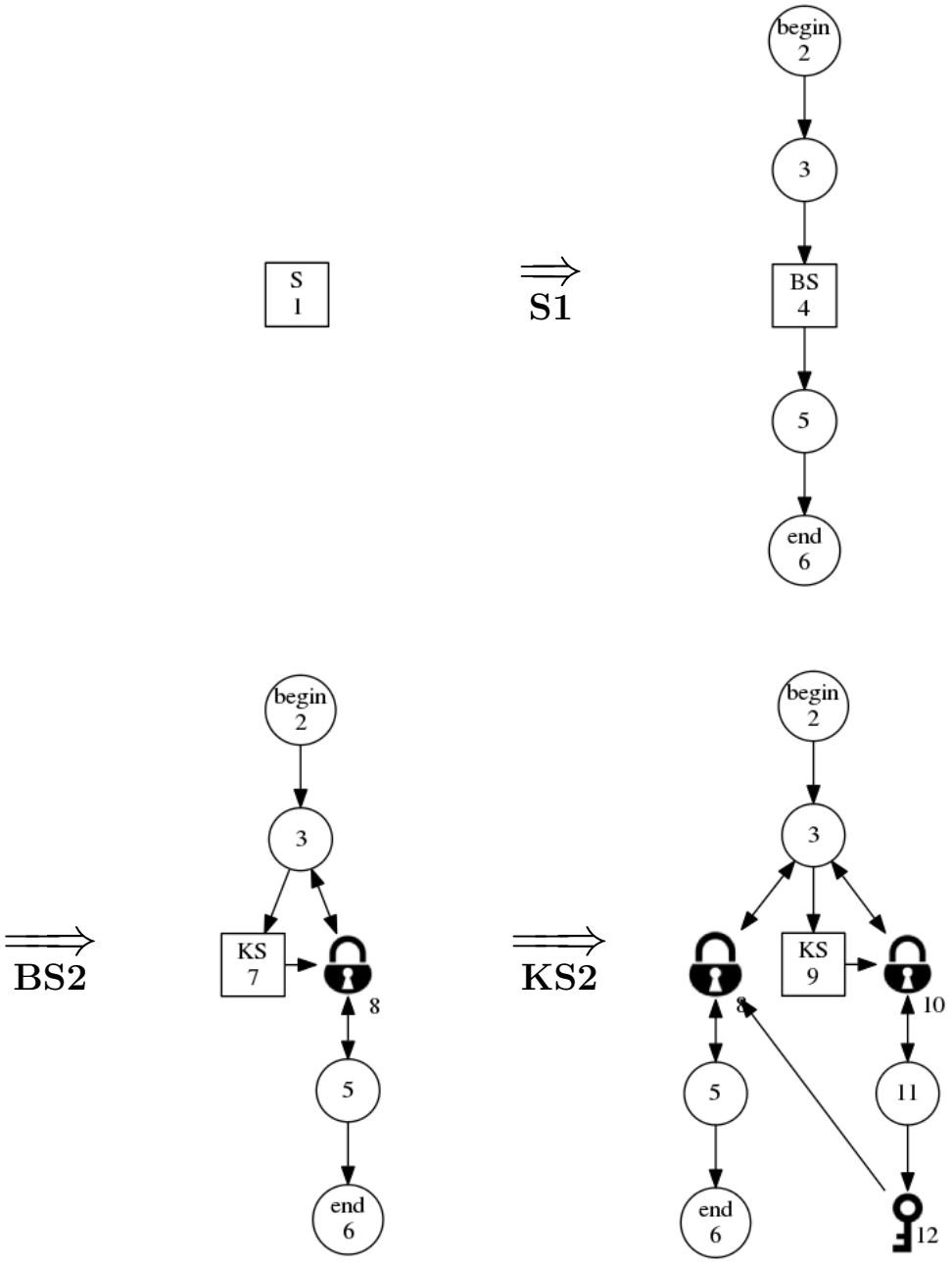


Figure B.17: Part 1 of *Inside Jabu-Jabu's Belly* derivation

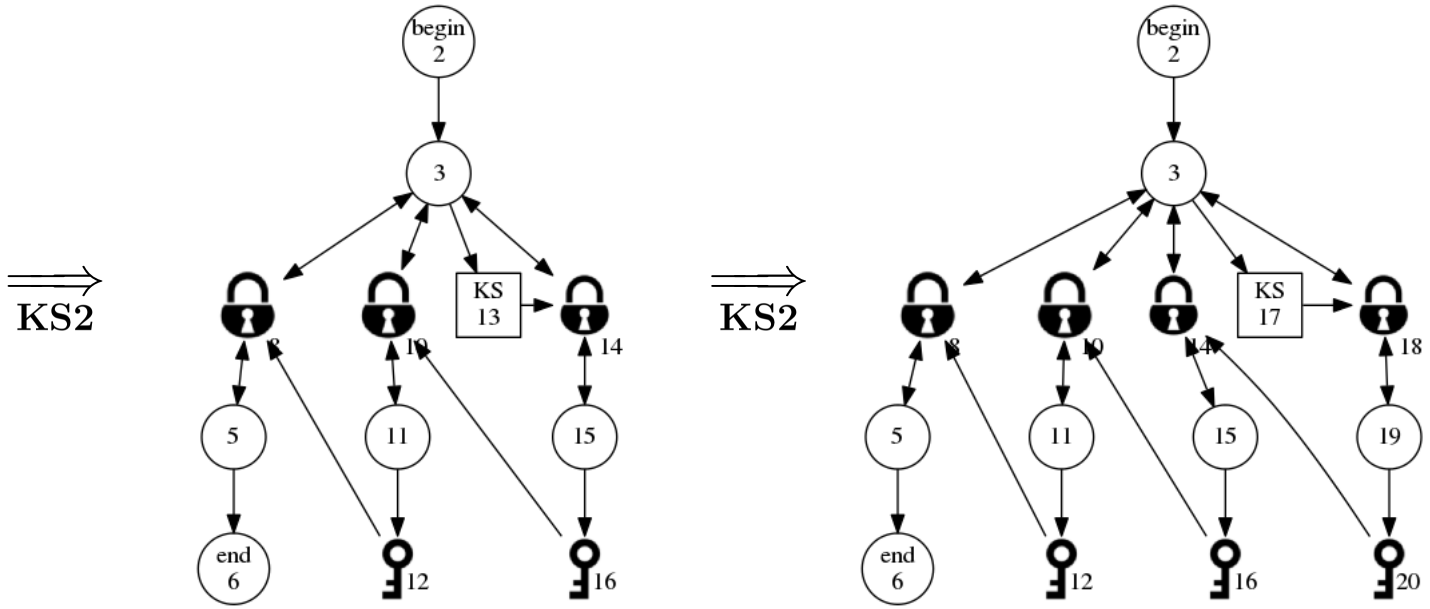


Figure B.17: Part 2 of *Inside Jabu-Jabu's Belly* derivation

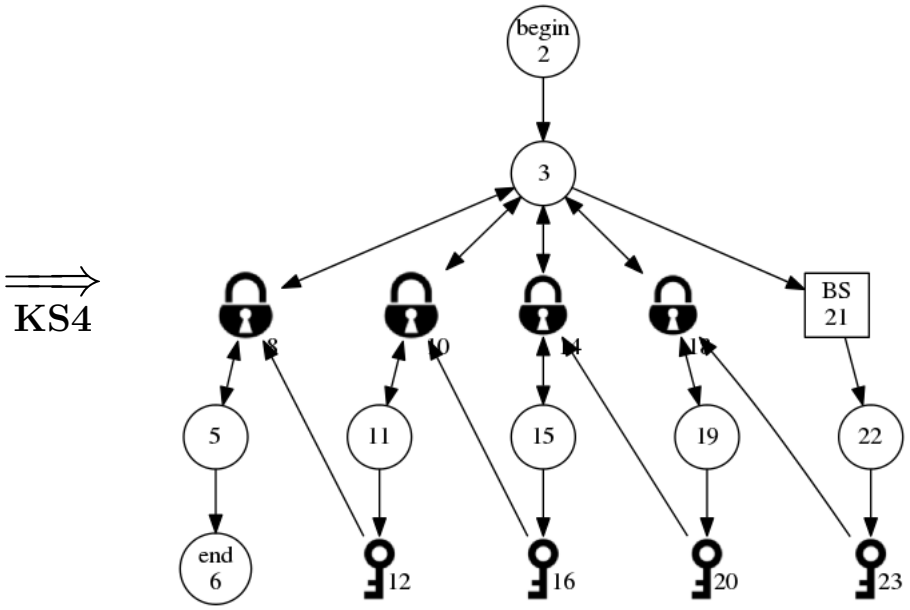


Figure B.17: Part 3 of *Inside Jabu-Jabu's Belly* derivation

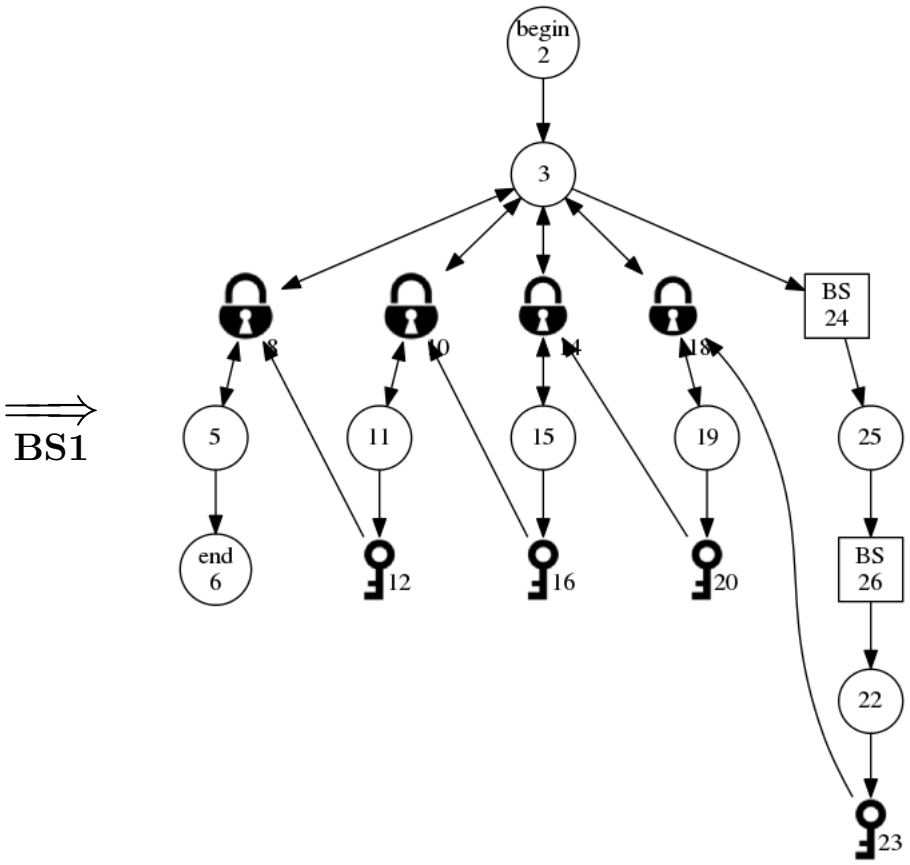


Figure B.17: Part 4 of *Inside Jabu-Jabu's Belly* derivation

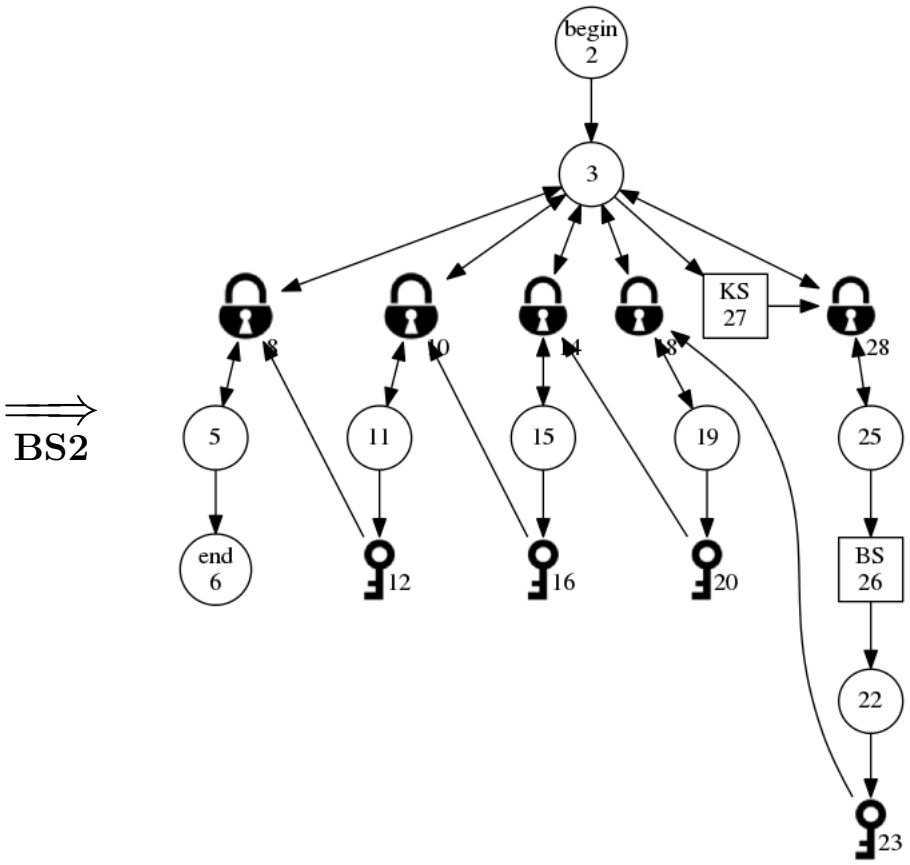


Figure B.17: Part 5 of *Inside Jabu-Jabu's Belly* derivation

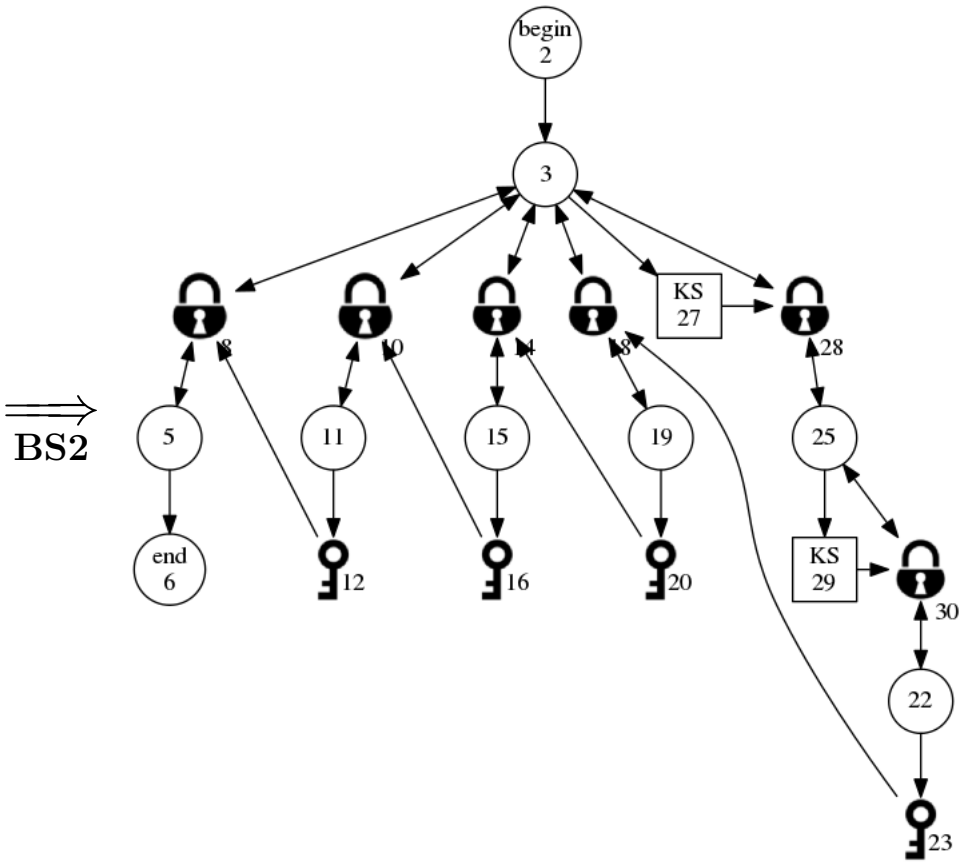


Figure B.17: Part 6 of *Inside Jabu-Jabu's Belly* derivation

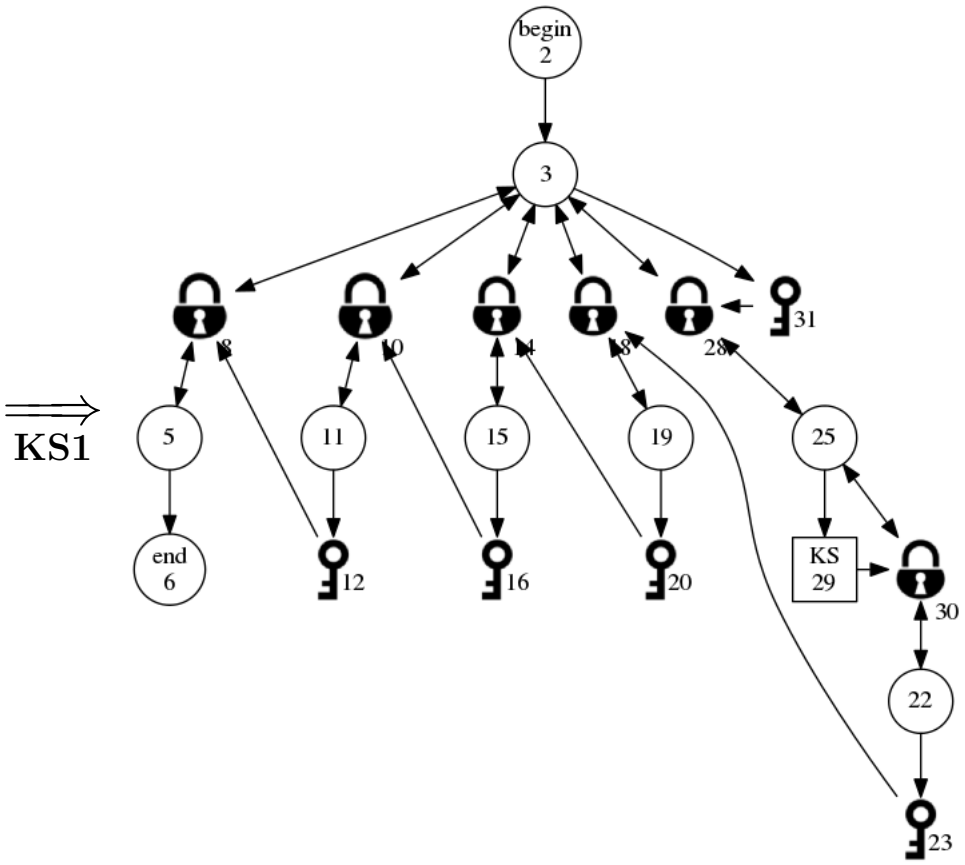


Figure B.17: Part 7 of *Inside Jabu-Jabu's Belly* derivation

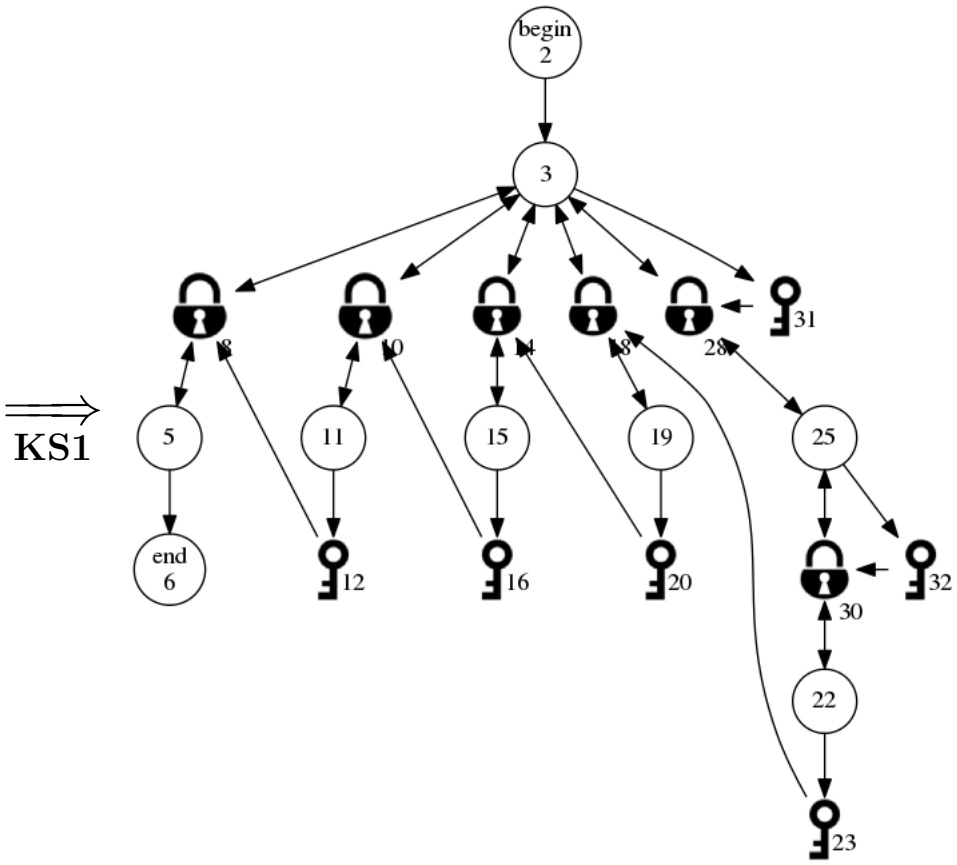


Figure B.17: Part 8 of *Inside Jabu-Jabu's Belly* derivation

B.4.4 Fire Temple

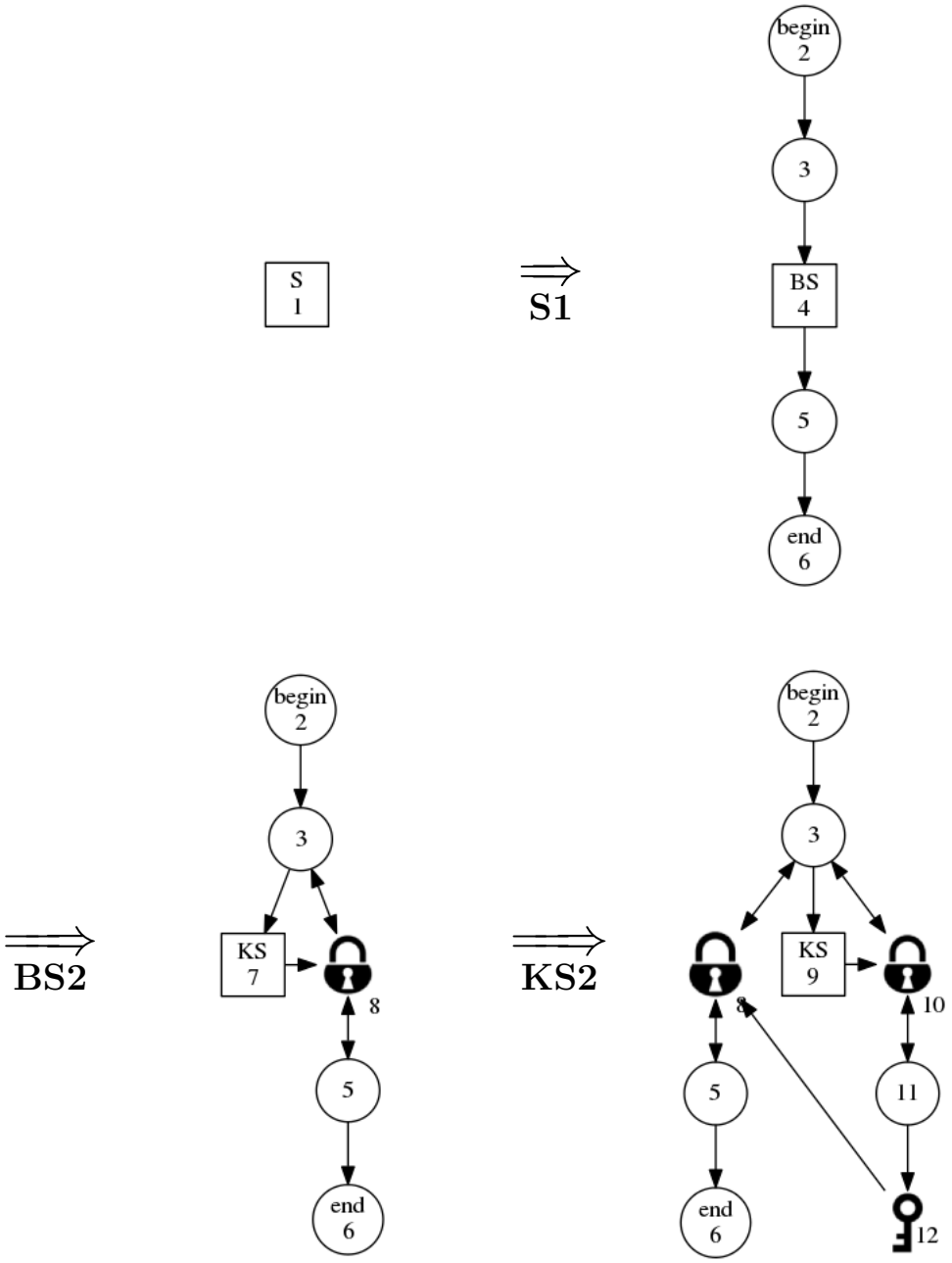


Figure B.18: Part 1 of *Fire Temple* derivation

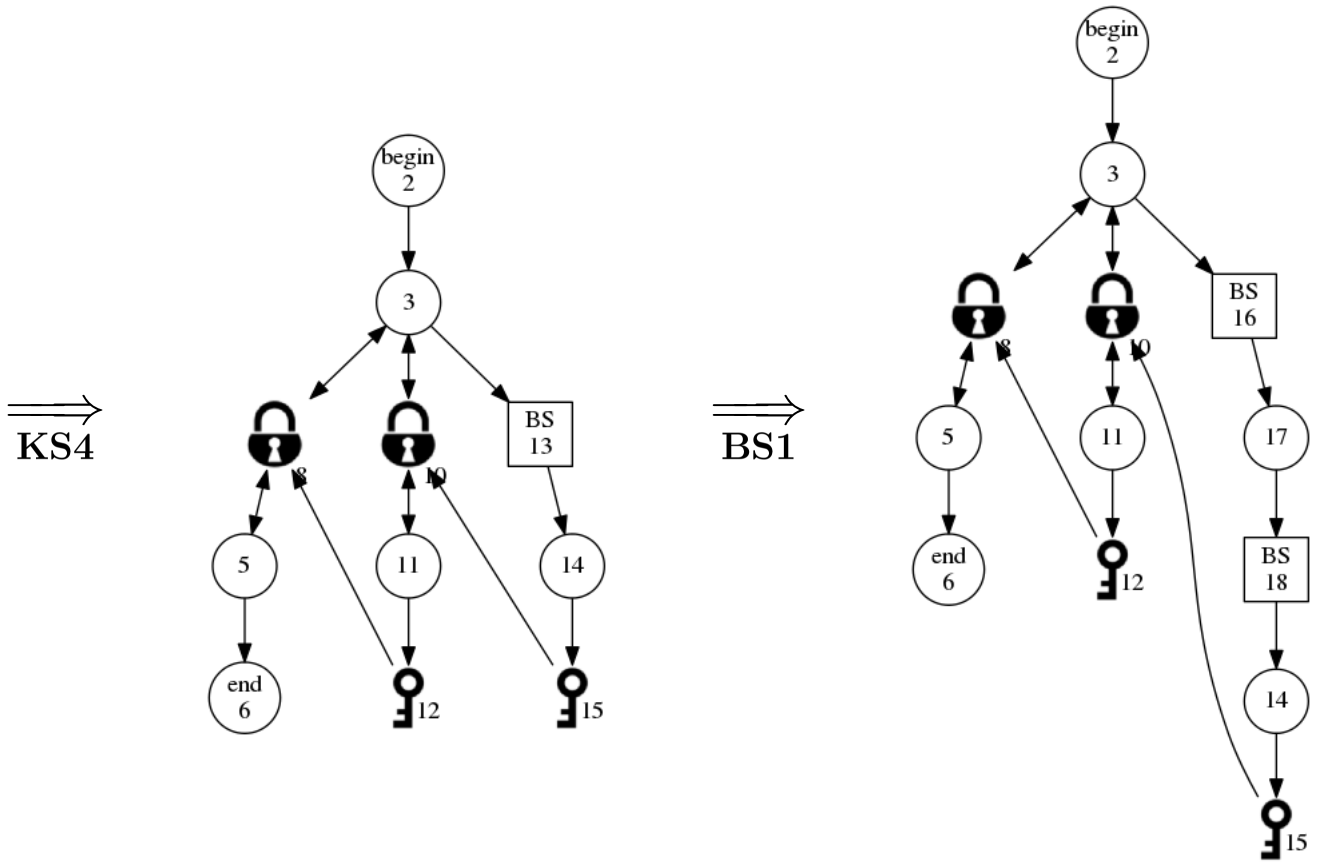


Figure B.18: Part 2 of *Fire Temple* derivation

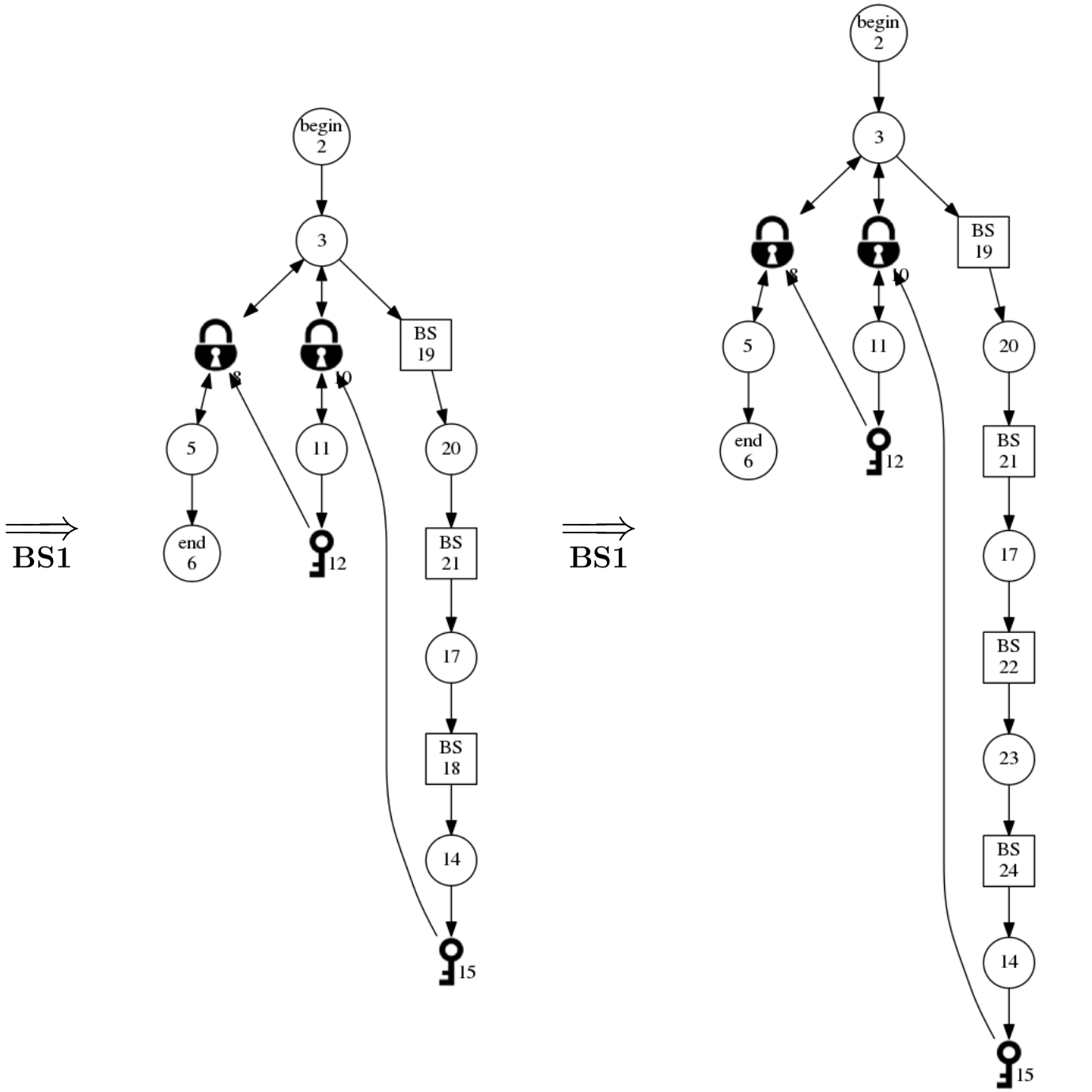


Figure B.18: Part 3 of *Fire Temple* derivation

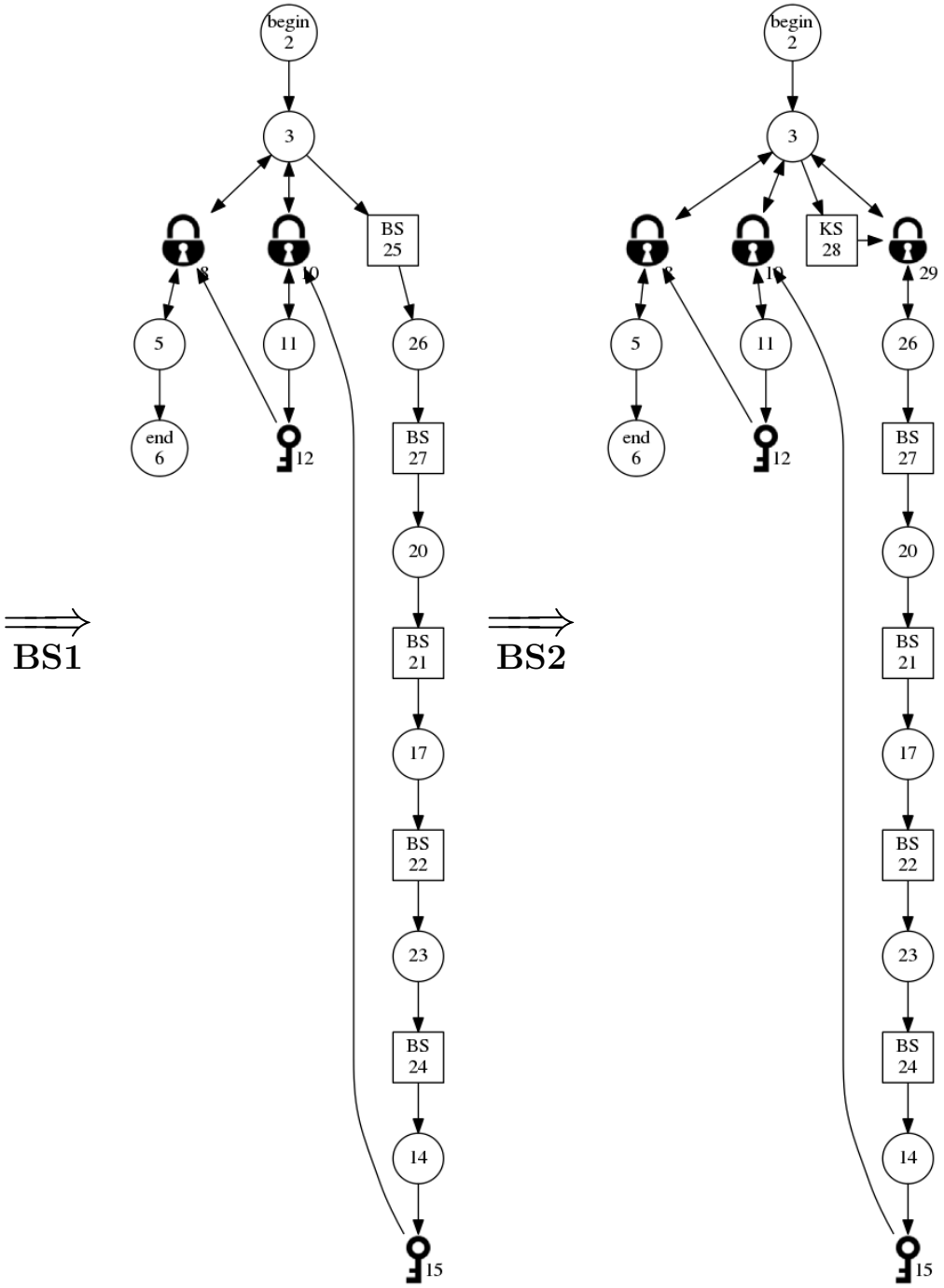


Figure B.18: Part 4 of *Fire Temple* derivation

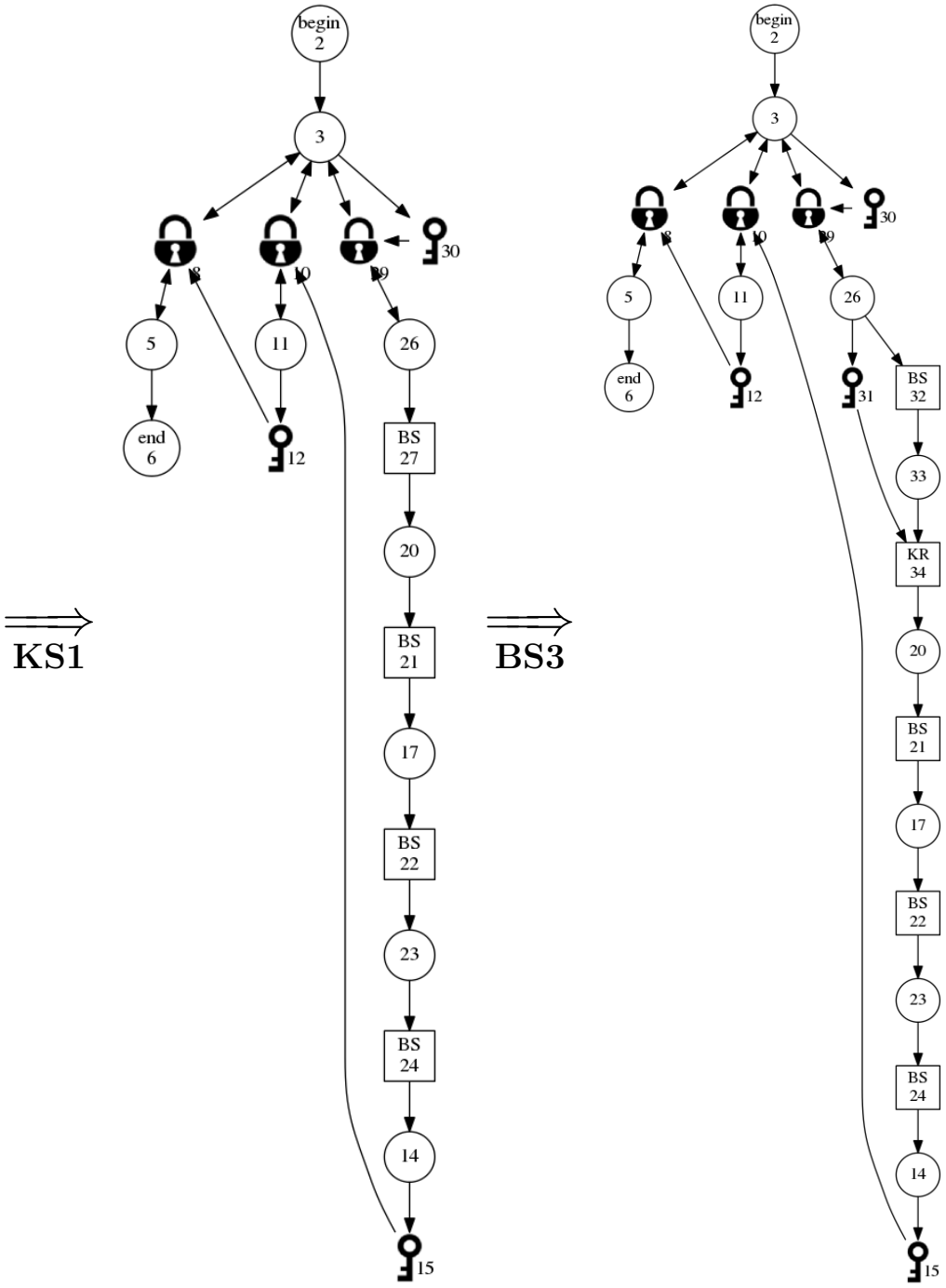


Figure B.18: Part 5 of *Fire Temple* derivation

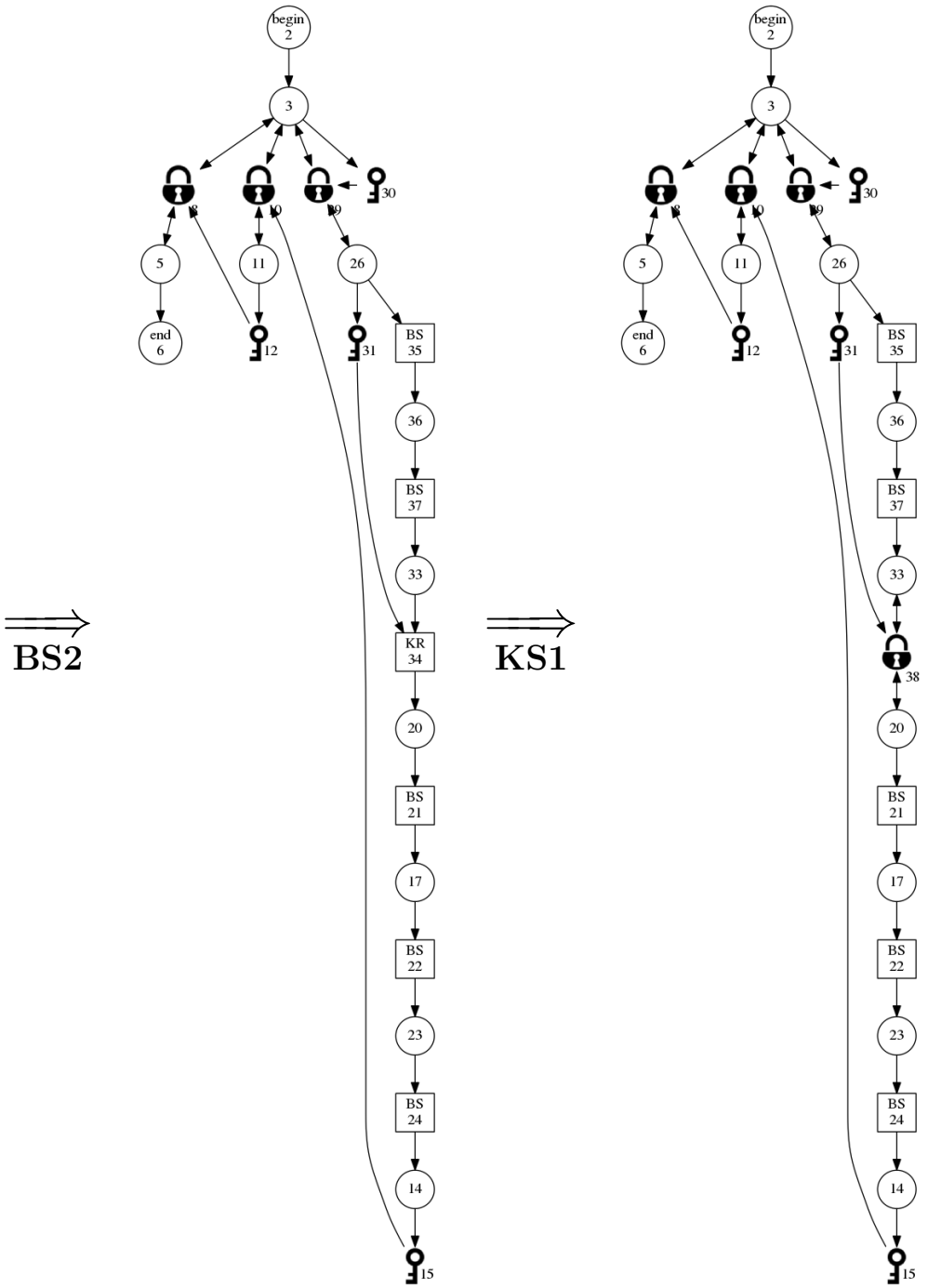


Figure B.18: Part 6 of *Fire Temple* derivation

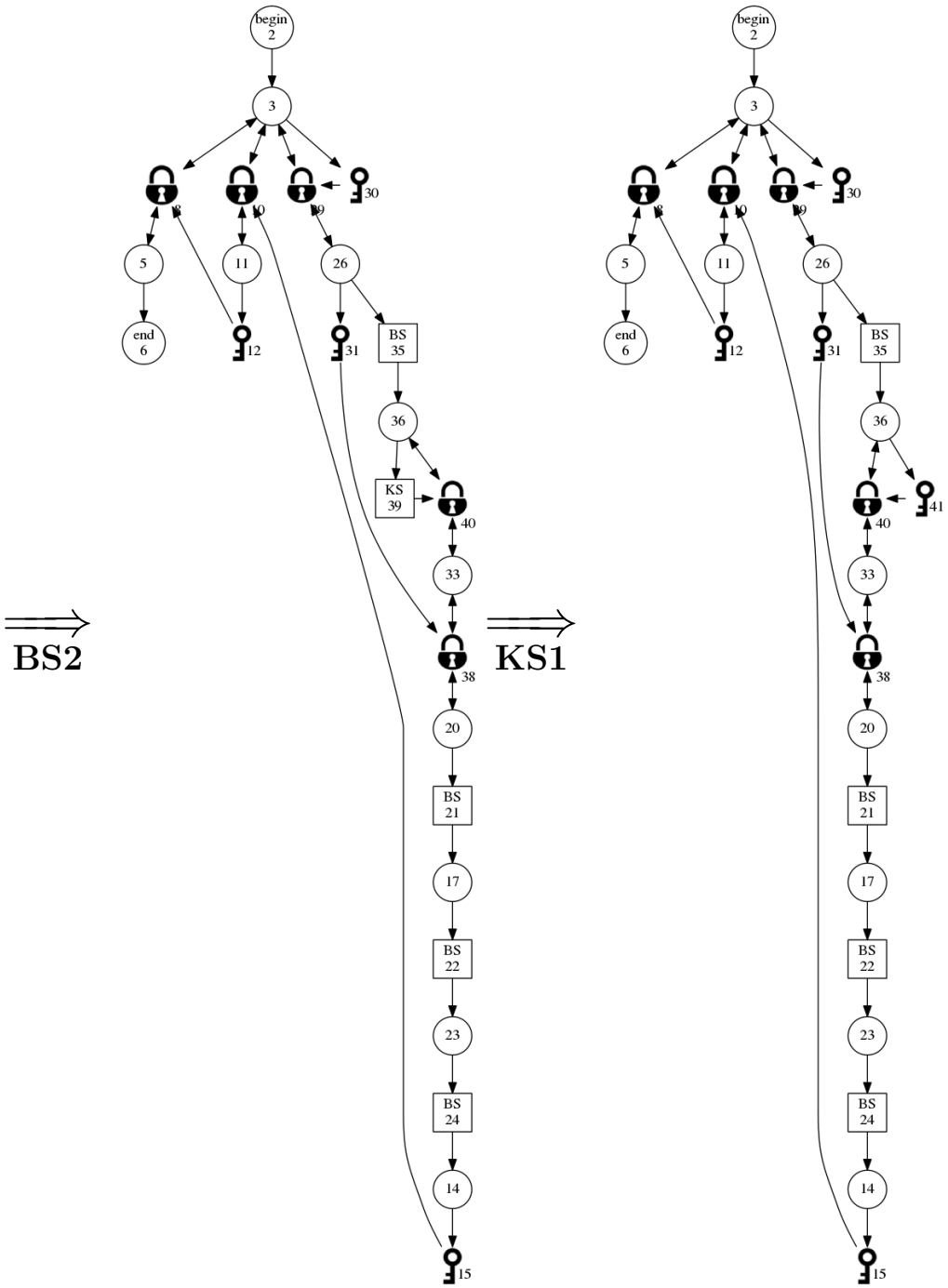


Figure B.18: Part 7 of *Fire Temple* derivation

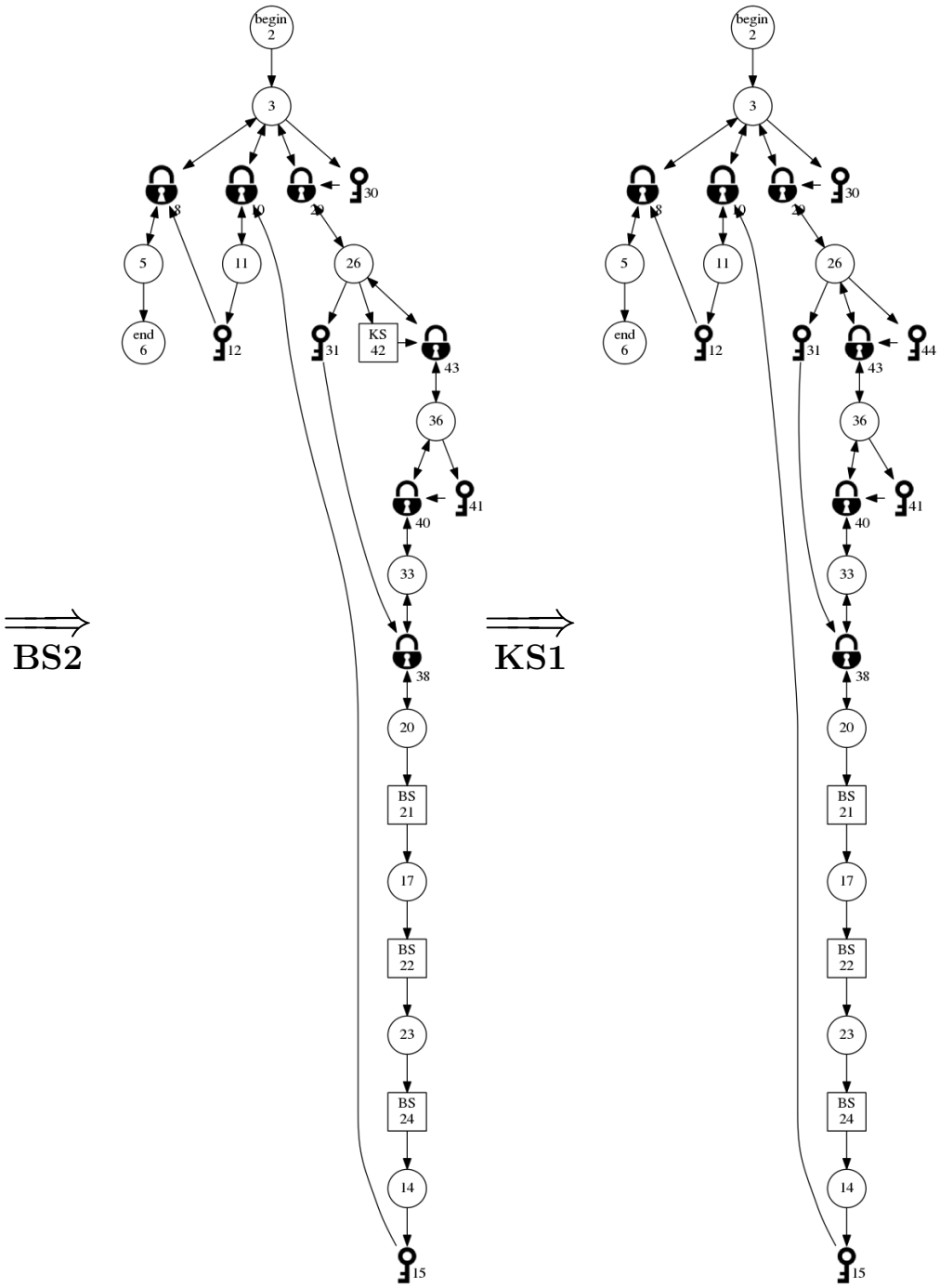


Figure B.18: Part 8 of *Fire Temple* derivation

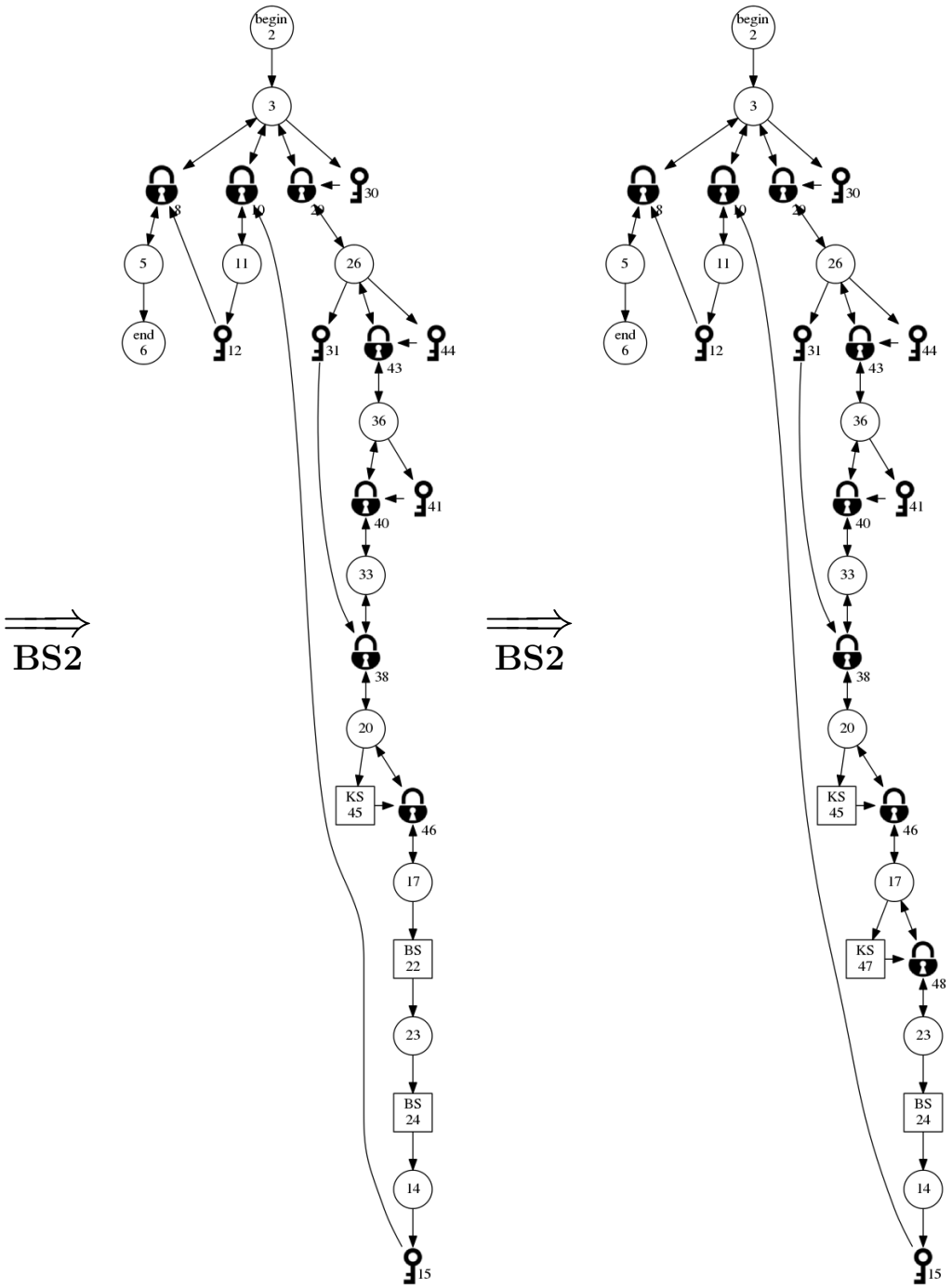


Figure B.18: Part 9 of *Fire Temple* derivation

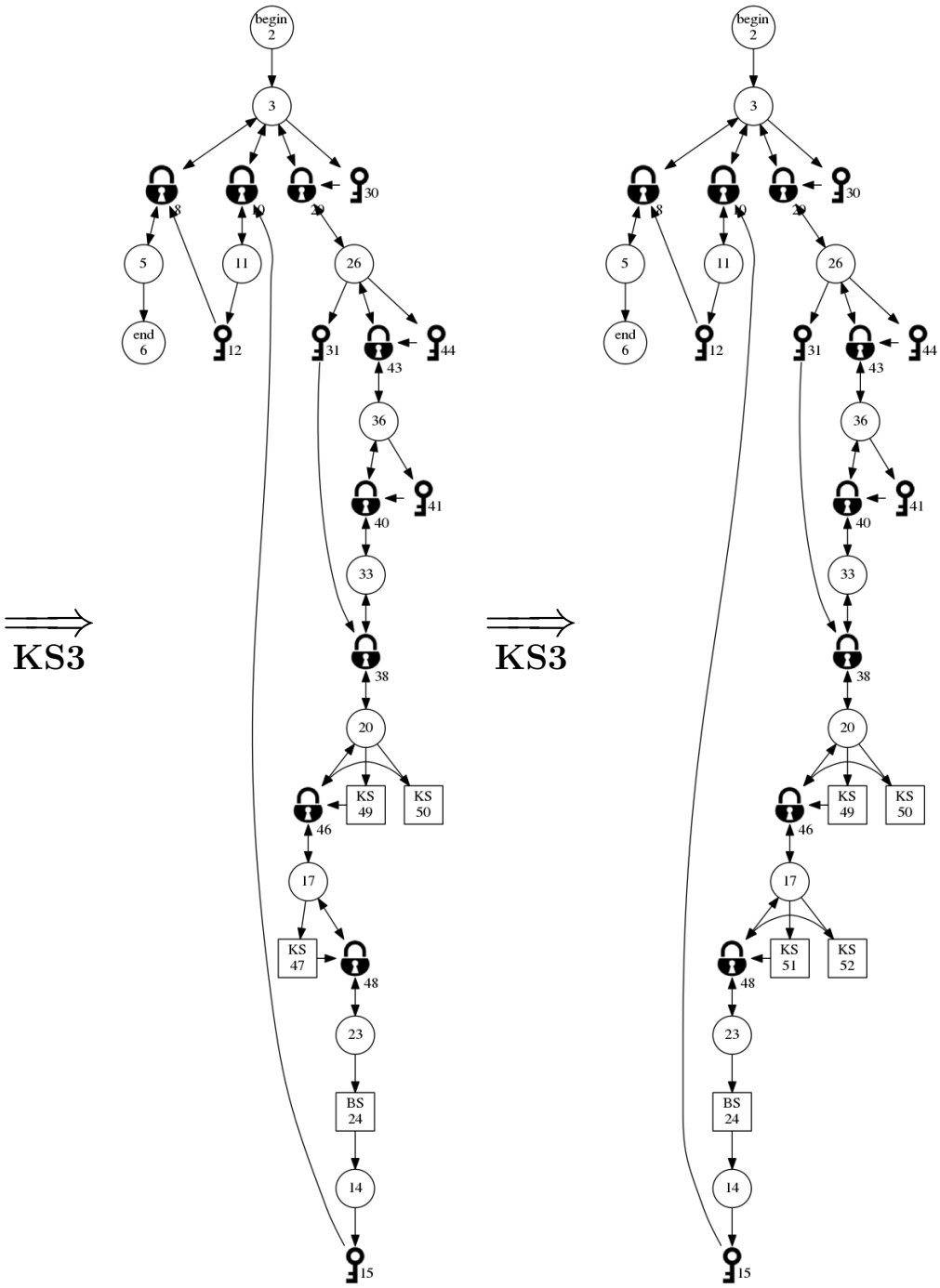


Figure B.18: Part 10 of *Fire Temple* derivation

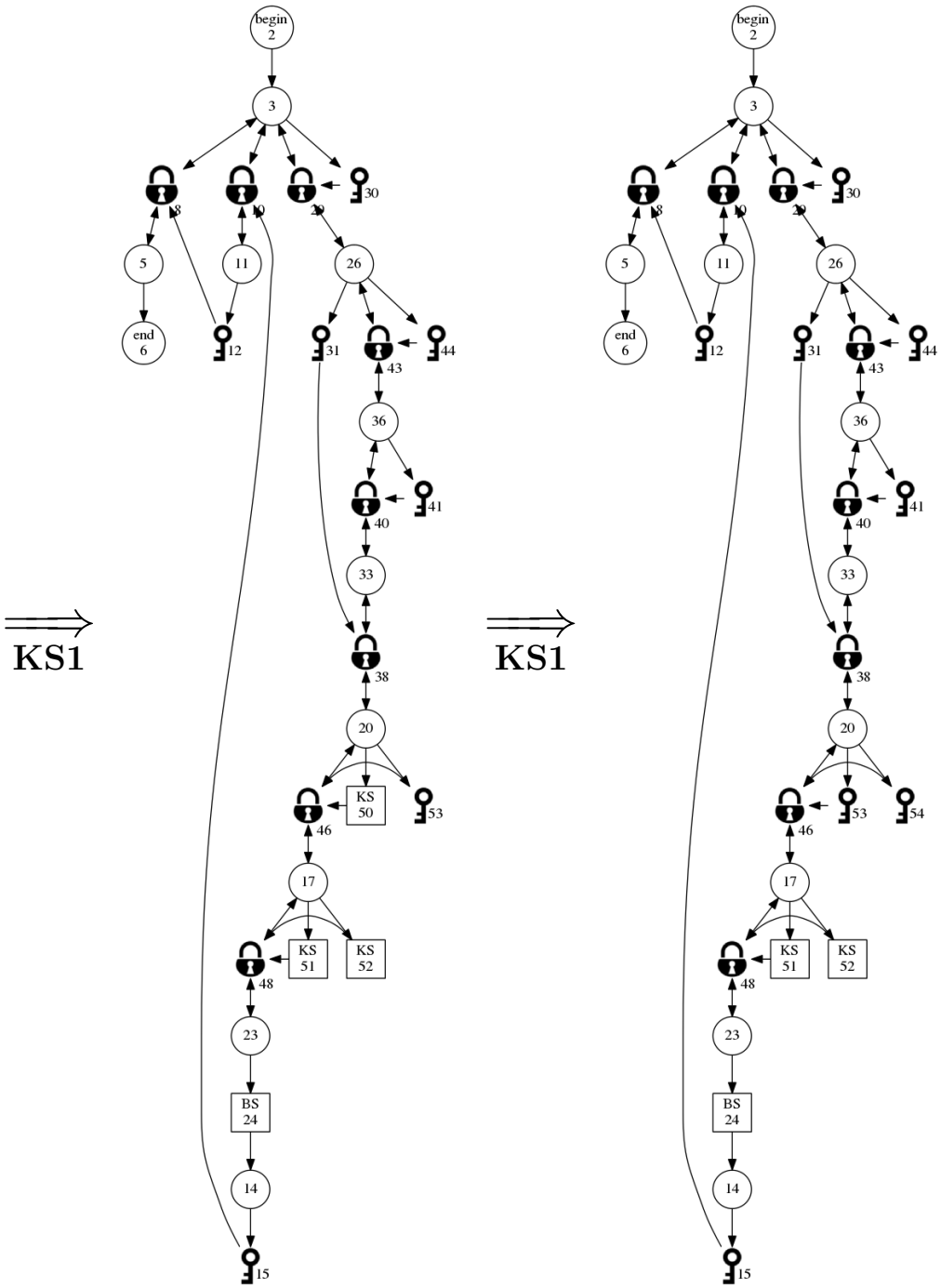


Figure B.18: Part 11 of *Fire Temple* derivation

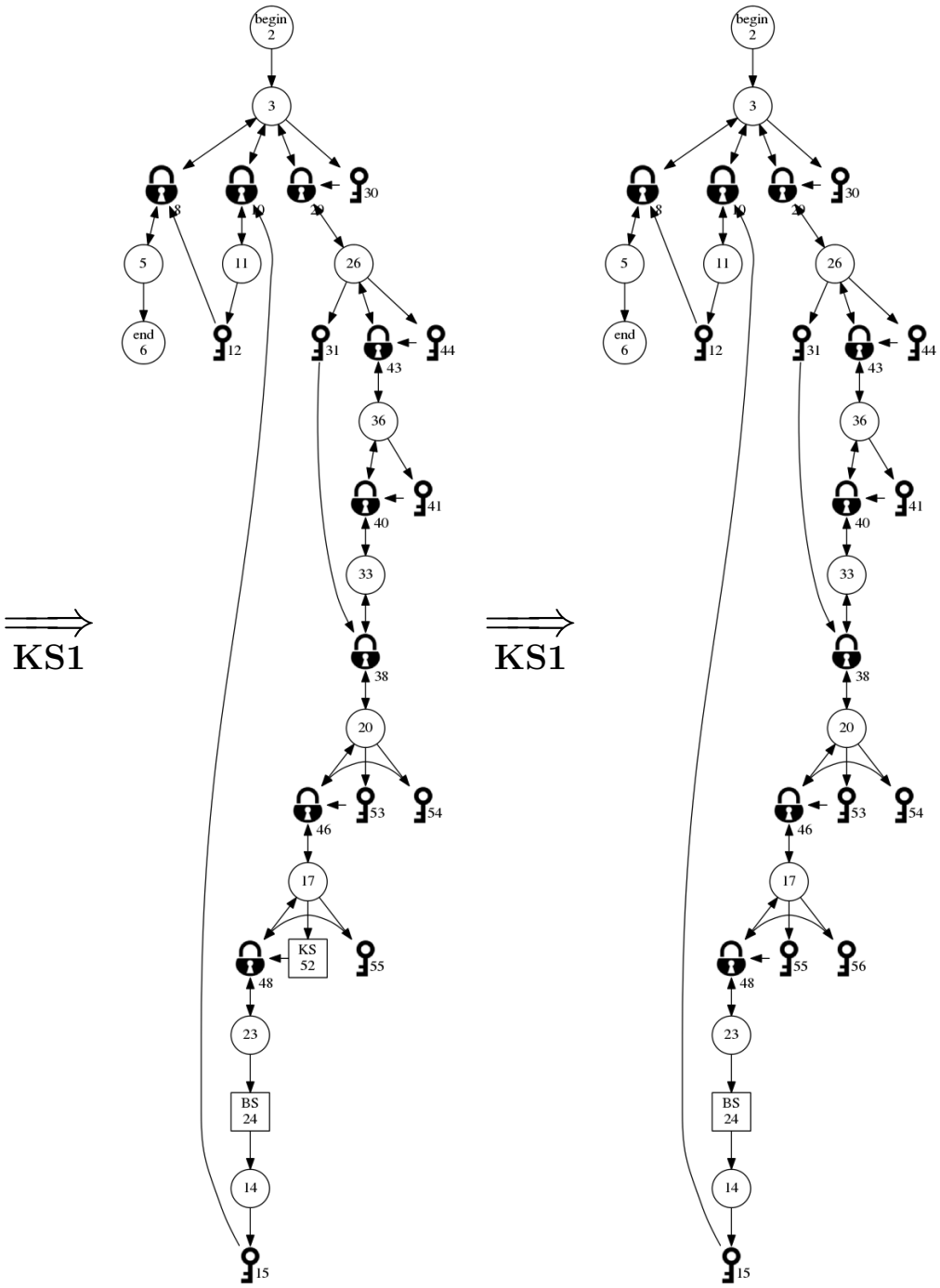


Figure B.18: Part 12 of *Fire Temple* derivation

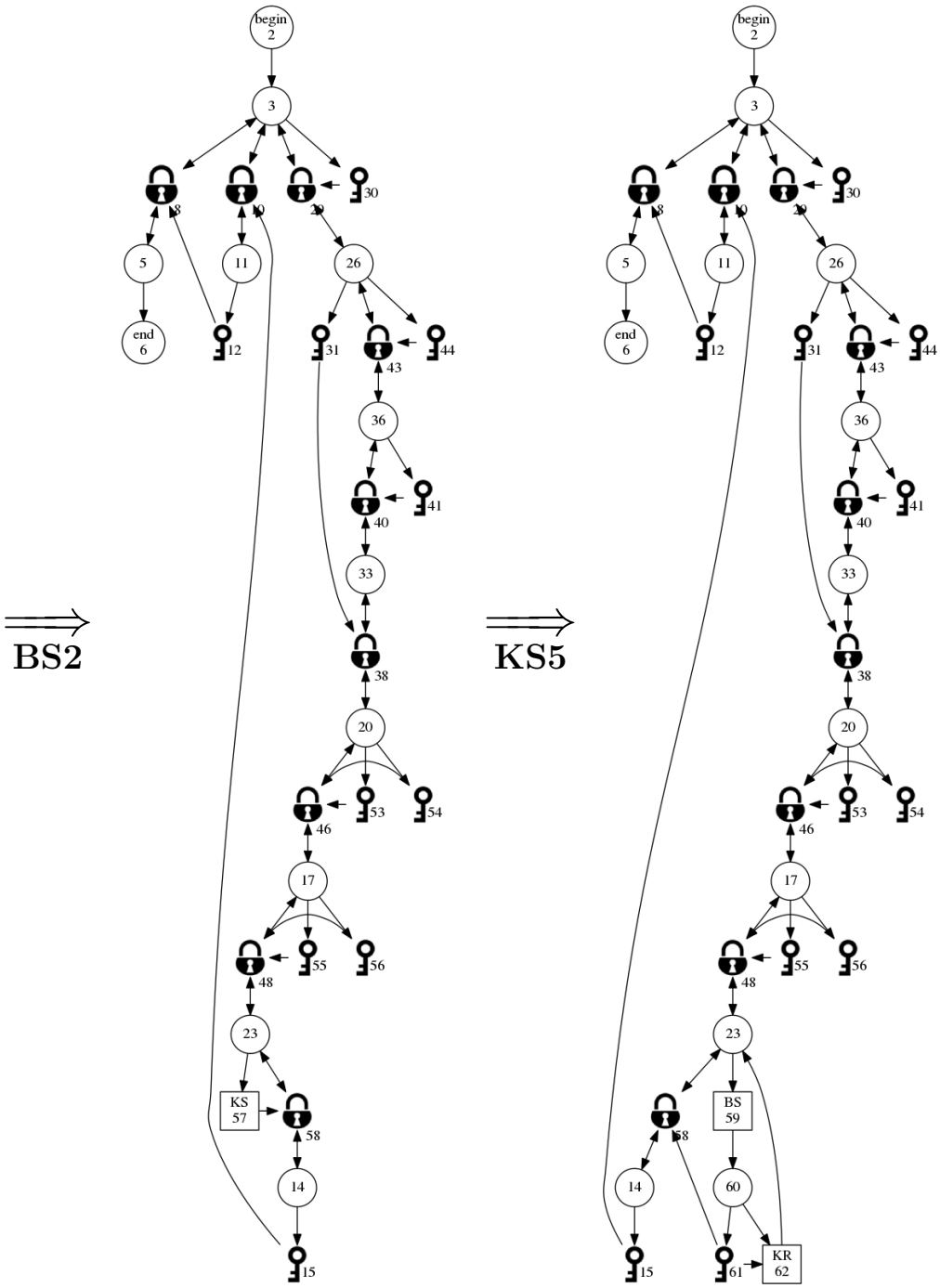


Figure B.18: Part 13 of *Fire Temple* derivation

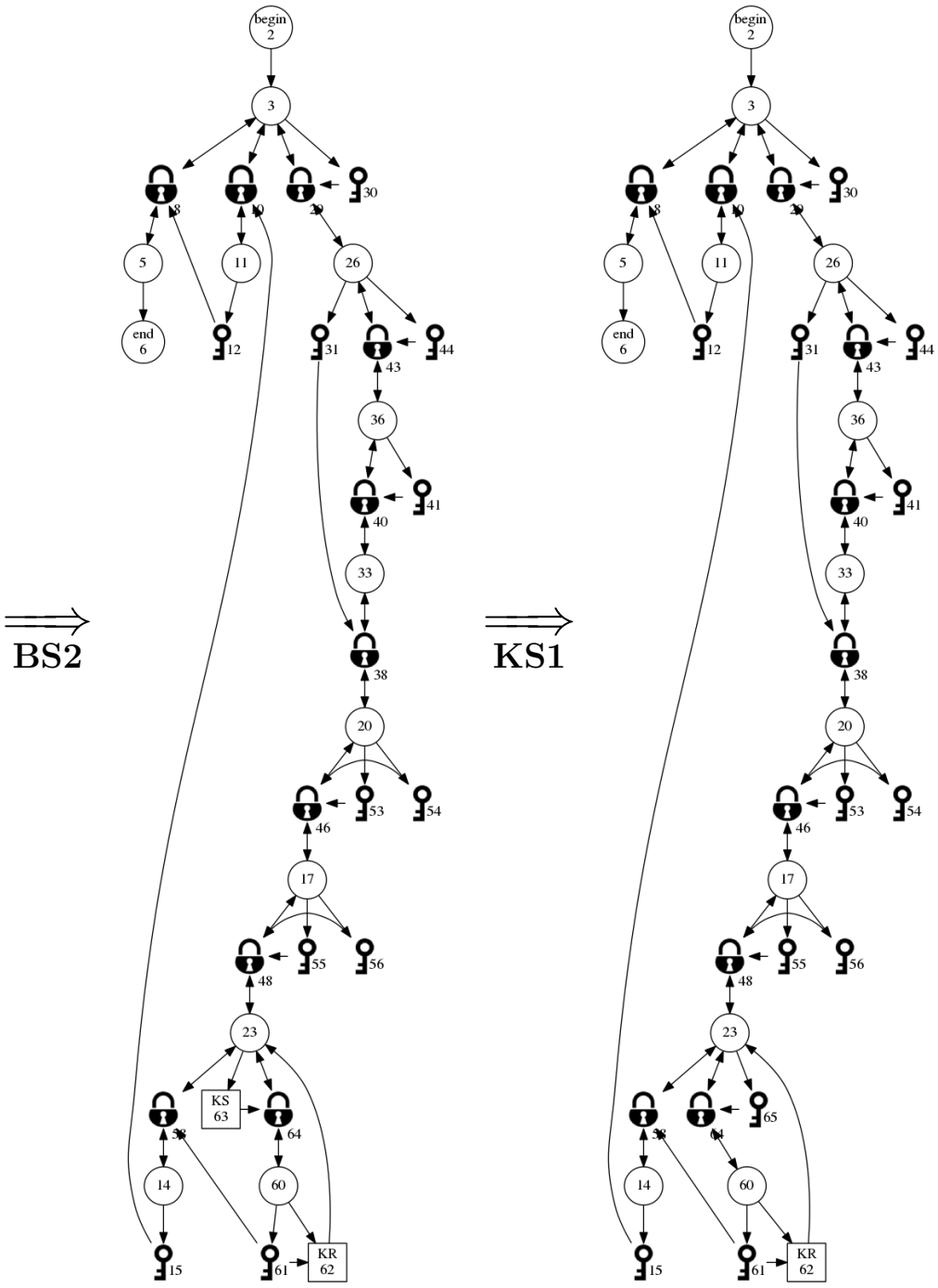


Figure B.18: Part 14 of *Fire Temple* derivation

$\Longrightarrow$
KR1

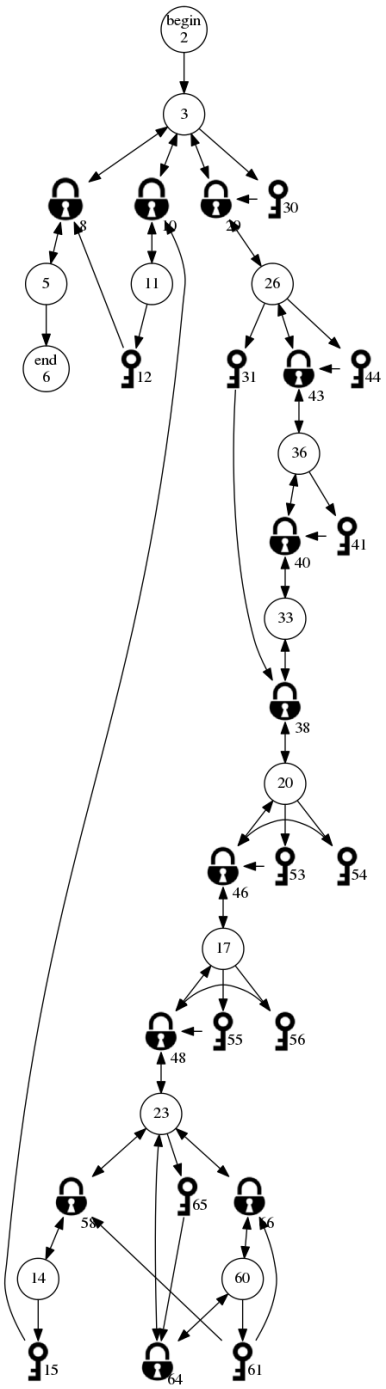


Figure B.18: Part 15 of *Fire Temple* derivation