Author
Alexander Voglsperger

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc

January 2024

LALR(1) Parser Table Generator

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis
LALR(1) Bottom-up Parser Generator

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Student: Alexander Voglsperger
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: February 2023

In the lecture "Compilerbau" (Compiler Construction) at the Johannes Kepler University, students are tasked to develop a top-down LL(1) parser throughout the semester. This parser always targets the language "MicroJava", with only slight deviations from year to year. Thus, the amount of work that has to be put in by the lecturers to prepare the material and homeworks is reasonable.

Yet, at the end of the semester, the course also teaches how bottom-up LALR(1) parsers work (using so-called *shift* and *reduce* operations stored in a *state table)*. If the input has no syntax errors, the parser repeatedly performs these steps until the whole input has been consumed and the complete parsing tree has been built. In the homework targeted at LALR(1), it is the students' task to derive the *state table*, i.e., which shift and reduce operations have to be performed in which situations. The same work has to performed by the lecturers that prepare the sample solution for the homework - most of the time even multiple times due to reworkings and restructuring of the assignment. This is error-prone and time consuming.

The goal of this thesis is to develop a generator for bottom-up LALR(1) state tables given a user-specified grammar. The grammar is given in BNF such as

```
S = A B.
A = x x.
A = x y z.
B = .
B = z.
```

Based on this, the generator should simulate which situations ("states") the bottom-up parser might encounter, as taught in the lectures. Data structures such as `Production`, `NTS` (non-terminal symbol), `TS` (terminal symbol), `Item`, `State` and `StateTable` might proof useful to keep track of any information about the grammar under test as well the generator's current state. In the end, a .csv file that lists the various shift and reduce operations in the different states should be produced that can be handed out as a sample solution.

As a last part of this thesis, a system should be developed that can "run" the parser and determine whether a certain sentence corresponds to the underlying grammar. For example, it should be able to simulate whether the sentence "xxy" will be accepted by the grammar mentioned above. If this is not the case, the system should be able to "recover" from the parsing error as taught in the lecture, i.e., it should calculate the "escape path" based on the states' guide symbols and use this information to synchronize the parser's current state with the remaining input.

Modalities:

The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.07.2023.

Abstract

In the exercise course *Compilerbau* at the Johannes Kepler University Linz, a typical task is to create a *LALR(1) parser table* and simulate an example input on it. The task of creating a homework with an appropriate level of difficulty is often time-consuming and tedious for the lecturers. This is where the idea of automation comes in. The main goal is to extract a generalized algorithm for this task and to create said tables for a given grammar. The simulation involves running inputs against the grammar and performing error recovery in the case of an invalid symbol. The result allows estimation of the workload and also automatically provides an example solution, which can then be distributed to the students. This saves the lecturers valuable time and effort, which can be redirected to developing engaging teaching materials and delivering lectures.

Kurzfassung

In der Lehrveranstaltung *Compilerbau* an der Johannes Kepler Universität Linz besteht eine typische Aufgabe darin, eine *LALR(1) Parsertabelle* zu erstellen und darauf Beispieleingaben zu simulieren. Für Lehrende ist es oft zeitaufwändig und mühsam, eine Hausaufgabe mit angemessenem Schwierigkeitsgrad zu erstellen. Hier setzt die Idee der Automatisierung an. Das Hauptziel besteht darin, einen verallgemeinerten Algorithmus für die Tabellenerstellung und Simulation zu extrahieren und zu implementieren. Die Simulation besteht darin, Eingaben auf einer Grammatik laufen zu lassen und im Falle eines ungültigen Symbols eine Fehlerbehandlung durchzuführen. Dies erlaubt es, den Arbeitsaufwand abzuschätzen und liefert automatisch eine Beispiellösung, die an die Studierenden verteilt werden kann. Dies erspart den Lehrenden wertvolle Zeit und Mühe, die für die Erstellung interessanter Lehrmaterialien und die Abhaltung von Lehrveranstaltungen verwendet werden kann.

Table of Content

Contents

1 Introduction

In the field of programming languages and compiler construction, the process of transforming source code into executable programs using compilers relies heavily on efficient parsing techniques. Parsing is a crucial phase that takes place after the source code has been lexically processed. The parser analyzes the syntactic structure of a program according to a given language's grammar. Parsing is a fundamental process that enables compilers to understand and interpret syntactic elements of a programming language.

There are several approaches to parsing a grammar within a compiler. The output of parsing a grammar are syntax trees, and typical methods for generating them include *top-down parsing* or *bottom-up parsing*. The former creates a tree by starting at a program entry and recursively descending through parts, such as methods and expressions. It ends with leaves that represent symbols that cannot be broken down further, i.e., terminal symbols. With bottom-up parsing, the exact opposite is performed. Here, the parser starts at the leaves and works its way up to the program entry.

The *Compiler Construction* course at Johannes Kepler University (JKU) teaches how such bottom-up *Look-Ahead, Left-to-Right, Rightmost derivation (LALR)* parsers work. More precisely, they teach about LALR(1) parsers, which are bottom-up parers that use a single *look-ahead token*. The homework for this topic includes the creation of the *state-transition-table*, simulation of example sentences and perform error recovery when necessary. The same work has to be done by the lecturers in order to prepare the homework and a sample solution. This task is very time-consuming and error-prone, as it requires several iterations to obtain a grammar of appropriate difficulty.

The tool developed for this thesis reads a user-specified grammar from a text file and transforms it into a representative internal structure, that reflects the user's intended representation. The loaded data is used to generate an intermediate parser table, and from this the state-transition-table. The parser table holds the unfolded built grammar, which is used to derive the state-transition-table. The state-transition-table then holds the transitions between states for a given symbol. The simulation of example sentences is based on an algorithm taught in the corresponding lecture. It makes it possible to determine whether a sentence gets accepted by the grammar or in case of rejection a recovery is performed based on guide symbols. Finally, it is possible to export the generated tables and simulation steps as *Comma-Separated Values (CSV)* files, and the simulation log as a simple text file. These files can then be handed out as sample solutions.

2 Background

Before diving into the development of the tool itself, it is important to establish foundational knowledge. This chapter introduces key concepts for comprehending LALR parsing. These concepts include the *Backus-Naur Form (BNF)* grammar in Section 2.1, the *extendend BNF (EBNF)* grammar in Section 2.2, and the operation of different parsing strategies in Section 2.3.

2.1 BNF Grammar

The BNF is a meta language for formal notation and is typically used to describe the syntax of programming languages [3]. BNF provides an easy-to-understand format and consists of four basic blocks as described:

- *Terminal Sysmbol (`TS`)* – Represents an atomic unit that cannot be broken down any further. It either consists of a *literal*, which is typically enclosed by double-quotes, or is a *terminal class*, e.g., *ident*, *number* or a combination in the form of an alpha-numeric (`an`).
- *Non-Terminal Symbol (`NTS`)* – Serve as placeholders which can be expanded or decomposed further into other `NTS` and/or `TS`.
- *Production* – Rule that specifies how a `NTS` can be decomposed into `NTS` and `TS`. For marking the end of a production, there is typically a designated character. To be consistent with the lecture, a simple dot will be used in this paper. Here is an example that specifies a production to represent the current directory: `Dir = "." "/" .`
- *Start symbol* – Symbol from which every sentence of the grammar's language can be derived from, analogous to a *main*-method in programming. It serves as the root of a grammar, from which valid derivations and expansions of the language begin.

For the purpose of simplifying the notation within this paper, `NTS` are denoted by an initial capital letter, while `TS` are represented with an initial lowercase letter.

If there are several possible derivations of an `NTS`, several productions with the same `NTS` can be created. Consider file paths as an example. The possible actions are to stay relative to the current path (`./`) or to go one step back relative to the current path (`"../"`). A description in BNF results in the grammar, as shown in Figure 1.

```
Dir = "." "/" .
Dir = "." "." "/" .
```

Figure 1: Example for a grammar that represents a directory, either the current one or the parent one.

To have constructs such as loops, recursion is used in combination with multiple productions. This means that there is a recursion termination condition and a recursive call with some extra. In the case of a name that can be several characters long, a grammar as shown in Figure 2 results.

A BNF grammar can be visualized in several ways to enhance its comprehensibility. Two commonly used techniques are *Railroad Diagrams* or *Syntax Trees*. These visualizations serve

```
Name = an .
Name = Name an .
```

Figure 2: Example for a grammar that represents a Name that can be arbitrary many alphanumeric (an) due to recursion.

as valuable tools for understanding and analyzing BNF grammars. These diagrams provide a concise representation that highlights important elements and relationships within a grammar. As a general rule of thumb, NTS are typically shown in a box with sharp corners, while TS are shown in a box with rounded corners.

Railroad diagrams provide a visual representation that makes it easy to see optional branches, merges and loops within a grammar. This enables clear identification of the hierarchical structure and facilitates understanding of how different elements relate and interact with each other. By providing a graphical depiction of the grammar's flow, railroad diagrams enhance the intuitive understanding of the structure of a grammar and aid in its analysis and interpretation. Figure 3 shows the grammars with branches and recursions clearly visible.

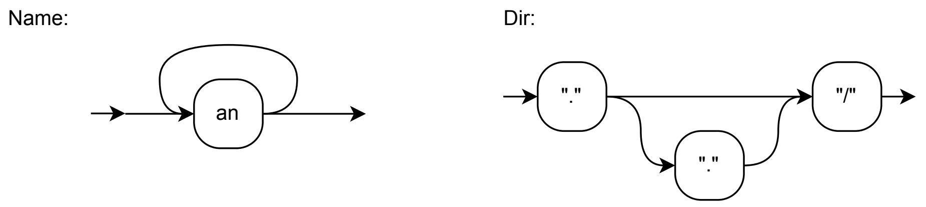


Figure 3: The left railroad diagram shows the recursive nature of Name. The right diagram shows an optional branch in the production Dir.

Syntax trees are a way to visualize the derivation of a sentence from a grammar. The nodes are comprised of the NTS and TS of the grammar, with the NTS being the inner nodes and the TS being the leaf nodes. The edges represent the relationship between the nodes. The root node is the start symbol of the grammar. Figure 5 shows the syntax tree for the sentence ./etc with the grammar shown in Figure 4. The figure shows how the Dir is created from ".", "/" and a Name. The name itself is recursively constructed from Name and an, with the three an being e, t and c.

```
Dir = "." "/" Name .
Name = an .
Name = Name an .
```

Figure 4: Example grammar for a syntax tree.

The railroad diagram provides a high-level overview and intuitive understanding for the grammar while the syntax tree allows for more detailed analysis of the constitution of a specific sentence.

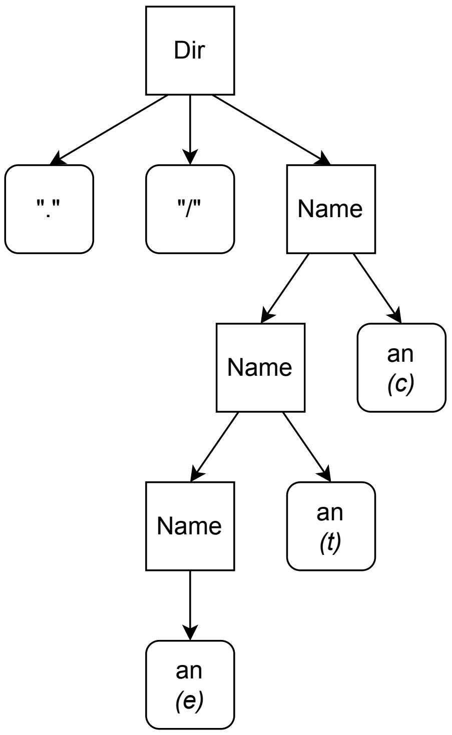


Figure 5: Syntax tree for the sentence `./etc` with the grammar shown in Figure 4.

2.2 Extended BNF

The extendend BNF (EBNF) is a superset of the BNF explained in Section 2.1. EBNF provides additional features to efficiently describe more complex grammars. For this reason, it is often used to describe programming languages or markup languages. The new features include the ones listed:

- *Repetition* – Specify a part in the grammar which is repeated zero or more times.
- *Alteration* – Specifies an alternative choice within a production.
- *Optional* – Specifies a part in the grammar which is not required, i.e., which can be present in a sentence zero or one times.
- *Grouping* – Specify groups of elements.

These EBNF features are typically used instead of certain BNF patterns to make the language definition more readable, as shown in Table 1

Feature	BNF	EBNF
Repetition	Recursive definitions	`{ ... }`
Alteration	Multiple definitions	`...|...`
Optional	Multiple definitions	`[ ... ]`
Grouping	Multiple definitions	`( ... )`

Table 1: Comparison of BNF patterns and EBNF features.

2.3 LL(1), LR(1) and LALR(1)

Having now addressed the subjects of BNF and EBNF, the next important terms are *Left-to-Right, Leftmost derivation (LL)*, *Left-to-Right, Rightmost derivation (LR)* and *Look-Ahead, Left-to-Right, Rightmost derivation (LALR)*. To be more precise, the version with one look-ahead token will be explained in more detail (i.e., LL(1), LR(1) and LALR(1)). These parsing strategies typically use BNF or EBNF grammars as input and produce some sort of state-transition-table as output. The examples described in this thesis, and many more, are described in great detail in *Parsing Techniques - A Practical Guide* by D. Grune et al. [2].

2.3.1 LL(1)

This section begins with an exploration of LL(1) parsing, a top-down parsing strategy that uses left-to-right, leftmost derivation. However, to understand the advantages of bottom-up parsing, it is helpful to mention LL(1) top-down parsing as well. The definition of LL(1) are as listed:

1. The parser must be able to decide between two alternatives only by using one look-ahead token.
2. This achieves (1), the first symbols of alternative routes in the grammar must be pairwise disjoint.

To explain the LL(1), we will use the basic path grammar shown in Figure 6. The grammar allows paths such as `./folder/file.csv` and `./folder1/../folder2/file.txt`.

```
Path  = "." "/" Step {"/" Step} "/" File .
Step  = an {an} .
Step  = "." "." .
File  = an {an} "." an an an .
```

Figure 6: Example path grammar written in EBNF.

The acronym LL already tells a lot about how the parsing strategy works. The algorithm starts with the topmost symbol, which the example is the start symbol `Path`. From there the algorithm moves from left to right, performing a derivation when it encounters an `NTS` in the grammar. To infer a leftmost `NTS`, the algorithm uses the look-ahead token to select an appropriate production. The derivation results in a recursive descent to expand all required productions. Figure 7 shows the input `./folder/file.csv` being processed by the LL(1). The final path then consists of `TS` only.

```
Path -> "."
Path -> "." "/"
Path -> "." "/" Step
  Step -> f o l d e r
Path -> "." "/" f o l d e r "/"
Path -> "." "/" f o l d e r "/" File
  File -> f i l e "." csv
Path -> "." "/" f o l d e r "/" f i l e "." csv
```

Figure 7: Application of LL(1) on `./folder/file.csv` with the grammar from Figure 6.

However, there are some drawbacks to LL(1) that are very important to consider in terms of grammar. A common cause of *LL(1)-conflicts*, i.e., situations in which the starts of alternatives are not disjunctive, are left-recursions. Left-recursions occur in situations where an `NTS` can directly or indirectly produce a sequence that calls itself on the leftmost position. The LL(1) approach causes indecisive branching, where the algorithm cannot decide whether to descend recursively or not. Looking again at the grammar in Figure 2, it is clear that there is a left recursion in the second production. The presence of such a left-recursion must be resolved either by replacing it with another pattern that does not cause an LL(1)-conflict, or by using another parsing strategy that works even when LL(1) does not, such as LR(1) or LALR(1), which are described in the following chapters. Even with these drawbacks, LL(1) can still be considered favorable for certain grammars that don't contain conflicting alternative starts, since it is relatively easy to implement.

2.3.2 LR(1)

LR(1) stands for Left-to-Right, Rightmost derivation with a single look-ahead token and is a bottom-up parsing strategy. As the name implies, the parser moves from left to right, performing the rightmost derivation. A bottom-up parser typically uses a *Pushdown-Automaton*, which internally uses four actions to mark what to do. These three more important actions are `SHIFT`, `REDUCE`, `ACCEPT` and will be explained shortly.

Before explaining the theoretical idea of the algorithm, we need to introduce some information about its structure and terminology, as shown in the following list:

- Each production has one or more *followers*. These followers consist of `NTS` and `TS` that may come after the respective production. For example, the follower of the production `Step` shown in Figure 6, is the literal `"/"`. The reason is that after a `Step` only a `"/"` can follow. If the next symbol is a `NTS`, the first symbol of its production is used. In the latter case, it may happen that the follower is a few levels into another `NTS` before a `TS` is reached.

- When the parser is working at a specific place of the production, it is called an *Item*.

- Depending on the position of the parser in an item, there are different actions possible:

  - `SHIFT` – The parser is not yet at the end of the production and encounters a symbol with which it can continue in the production. Read the symbol from the input and move forward with the item.

- `REDUCE` – The parser has reached the end of the production and encounters a suitable symbol on the input. The `NTS` of the production is put on the input.
- `ACCEPT` – Indicates a successful parsing process, i.e., a full sentence has been read (for this, EOF is represented with a special `TS` typically `'#'`, which has to be encountered at the correct location).
- `ERROR` – Indicates that the look-ahead symbol could not be processed.

These actions are also the reason why bottom-up parsers are also called *Shift-Reduce-Parser.*

- The dot in an item indicates the current position of the parser. This is different to the dot at the end of a production that was mentioned in Section 2.1.
- The parser represents the progress in so-called *States*, which represent a snapshot of the processed information.
- When the parser processes several productions at the same time, such a state can consist of multiple items. This is the case when the parser is not able to distinguish between multiple items by using the look-ahead token. An example is `an {an}`, which is the beginning of `File` and the first part of `Step`.
- Each state contains so-called *core items.* Items are called so, if the parser is not at index zero. In other words, at least one symbol of an item has already been processed. An exception is the start symbol, which is a core item even if the parser is still at the beginning. In Table 2 are examples of some core items and some that are not.
- The core items of a state are always known and are used to derive the *hull* of a state. The hull of a state are all corresponding items that the parser can use to perform actions on, based on the look-ahead token.

Core Item	No Core Item
Path = . "." "/" ...	Step = . an an
Path = "." . "/" ...	Step = . "." "."
File = an . an ...	File = . an an ...

Table 2: Examples of core items and non-core-items.

Another important thing to think about is the grammar compliance. A grammar is LR(1) compliant, when in every state the look-ahead symbols allows the parser to decide

- if it is a `SHIFT` or `REDUCE` action,
- and in the case of a `REDUCE` action to which `NTS` it gets reduced.

The following is a theoretical introduction to how the LL(1) actually works. The technical implementation details are explained in Section 4.3. For the theoretical explanation the grammar shown in Figure 6 is reused.

The first goal is to create the parser table, which consists of all possible states for the grammar. First, the initial state is created, which consists of the production of the start symbol with the parser at index zero, as shown in Figure 8. Note the `'#'` at the end of the production. It makes it easier to detect of the end of the start symbol's production.

```
Path = . "." "/" Step {"/" Step} "/" File #
```

Figure 8: Initial state of LR(1) for the grammar shown in Figure 6.

The next step is to build the hull from all the core items in the state. Since this is the first state, the start symbol's production is also a core item. If the symbol to the right of the current parser position is a `NTS`, the algorithm recursively derives through the corresponding production(s). In the case of the example, the initial state doesn't change after the hull is built. The reason is that the symbol to the right of the parser position is the `TS` with the literal `"."`.

Now assume that the parser is between the literal `"/"` and the `NTS` for the `Step`. The state before and after building the hull is shown in Figure 9. There the algorithm recursively descended into the `Step` and stopped at both versions, because there are no more `NTS` for derivation to the right of the parser position inside the `Step`.

```
Before:
  Path = "." "/" . Step {"/" Step} "/" File #

After:
  Path = "." "/" . Step {"/" Step} "/" File #
  Step = . an {an}
  Step = . "." "."
```

Figure 9: Hull building if parser is in front of the first `Step`.

The built hull is the base for the algorithm to detect the next action to be performed. The `ACCEPT` is the most basic action to check for. The action is performed when the parser is at the end of the production for the start symbol and the look-ahead token is the special `TS` for `'#'`. Similarly easy is the detection of `REDUCE`, which is only possible when the parser is at the end of an item. The action indicates which production is used to perform the reduce operation. The `SHIFT` action is the most complex, as it requires several checks to find a match or create a new state. The checks are not as trivial as the others and are explained in Section 4.3.2, but in simple terms the algorithm does one of the following in the order listed:

1. Search for reusable action in the local state.
2. Search for reusable action in all other states.
3. Create a new state.

When using the state shown in Figure 8 again, it was already clarified why nothing changes when the hull is built. However, the action still needs to be detected. In this case, the action would be a `SHIFT` with the `TS` `"."`. Since this is the first and only item in the initial state, there are no other actions yet. So the algorithm creates a new state. Into this new state, the algorithm copies the item from the current state and moves the parser one step forward, as shown in Figure 10.

```
(Initial) State 0:
  Path = . "." "/" Step {"/" Step} "/" File #     | SHIFT "." 1

(Created) State 1:
  Path = "." . "/" Step {"/" Step} "/" File #
```

Figure 10: Detection of `SHIFT` and creation of a new state.

Building the hull and finding a feasible action is performed iteratively for each state until all states have been processed. This includes new states that arise during the process.

When this is done, a state-transition-table is created, which is used to perform the actual simulation of a sentence. The table is not strictly necessary, but it is much easier to look up information from the state-transition table. The creation of the table is not too difficult, since it is just a mapping from a state and a look-ahead token to an action. The state is represented by a number, which is the index of the state in the list of states. The action is typically represented by a string naming the action and varying additional information. The beginning of the state-transition-table for the example is shown in Table 3. In addition to state 0, the action for the single item in state 1 has been added to the table.

State	#	"."	"/"	an	Path	Step	File
0		SHIFT 1					
1			SHIFT 2				
...							

Table 3: State-transition-table for state 0 and state 1 of grammar Figure 6.

2.3.3 LALR(1)

Look-Ahead, Left-to-Right, Rightmost derivation (LALR)(1) is a variation of LR(1) which allows smaller parser states and thus a smaller resulting parser table. The reduction in states is achieved by merging states with identical core items that have identical look-ahead tokens. During the combination, the followers must also be merged. The example shown in Figure 11 contains two states of a fictional, but similar grammar to the one used before. The symbols after the '/' represent followers of the given item. The example shows that state 1 and state 2 share the same core item (`Step = "." . "."`). Therefore, the two states can be combined into one state and the followers can be merged.

```
State 1:
  Step = "." . "."              / an
  File = . "." an an an         / "#"

State 2:
  Step = "." . "."              / "/"

State 1-2:
  Step = "." . "."              / an, "/"
  File = . "." an an an         / "#"
```

Figure 11: Combinations of two states with equal core items.

3 Overview

The internal structure of the program is in most parts similar to what is taught in the course. A simplified overview of the data flow in the program is shown in Figure 12.

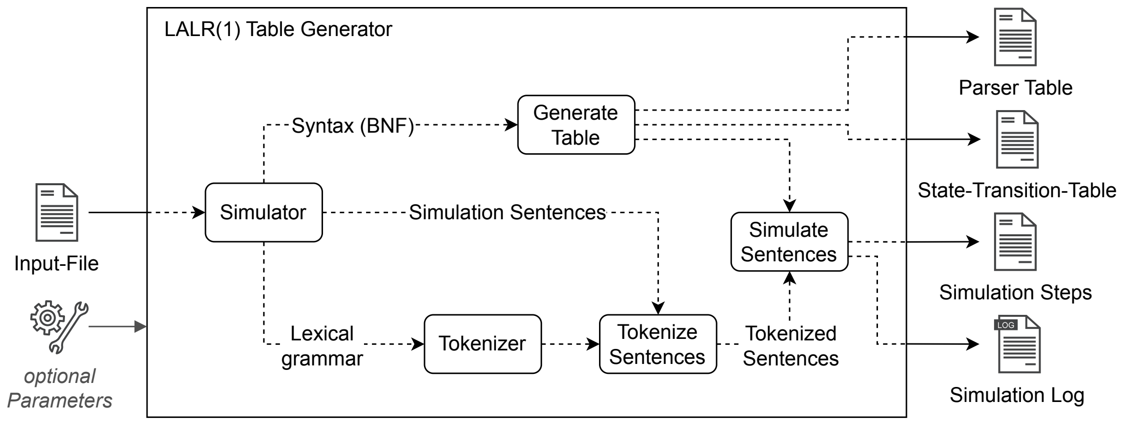


Figure 12: The program flow starts at an input file that is read and split into chunks. These chunks are processed by different sub-processes and then reassembled for simulation. At the end, the program outputs four files to the user.

The input file resembles a homework specification, which contains the necessary grammar, the definition of the lexical structure and inputs that should be checked against the grammar. An example of such a file is shown in figure Figure 13. The tool loads this input file and splits the content into the different parts for further use. As an input for the grammar and lexical structure, a basic BNF is used.

```
Path = Dirs Name .
Dirs = Dir .
Dirs = Dirs Dir .
Dir  = Name "/" .
Name = an .
Name = Name an .

an   = letter | digit .

home/user/file
etc/config
```

Figure 13: Example content of an input file.

The input sentences in the last block can't be processed directly by the program. To convert them into the internally used TS, the tool uses Coco/R as another parser. However, because this tool supports different atomic TS that are not known at compile time, the other parser must be created and loaded at runtime. Figure 14 shows a sentence converted into a list of TS for further processing. In the figure it is visible that the words used for names are converted into a series of `an`. It is also shown that the slashes are converted into the literal `"/"`.

```
Input:
  home/user/file
Converted:
  an an an an "/" an an an an "/" an an an an
```

Figure 14: Example sentence converted into a list of `TS` by additional parser.

The simulation stage takes the parsed simulation sentences and the state-transition-table to perform the simulation itself. The individual steps are stored in a human-readable file for the user. If the simulation encounters an error, e.g., an unexpected `TS`, the error handling algorithm from the lecture is applied. The error handling also produces a log with all error messages, e.g., *"X inserted at position Y"*, which makes it easier to follow the steps taken by the algorithm.

In summary, the tool takes one input file and produces four output files. The four output files consist of parser table, state-transition-table, simulation steps and simulation log. In addition to the input file, optional parameters can be passed, i.e., output directory, action format (how the actions are indicated, e.g., all lowercase, all uppercase or just a single character) and a Java binary (to execute the Coco/R dependency). These optional parameters are then taken into account at the stages where these properties are used, e.g., the action formatting in the output tables.

4 Implementation

Since the previous sections have presented the background and an overview of the tool, this section will delve into the implementation with greater detail. The implementation will cover theoretical aspects and be illustrated with an example, that extends until the end.

4.1 Internal Data Structure

Before getting into the actual implementation of the algorithm and other functionalities, this chapter provides some information about the internal data structure used. Since the internals are quite extensive, only the more relevant parts will be explained.

4.1.1 Symbol, TS and NTS

Starting with the very basic structure of a grammar, the `TS` and `NTS`. More specifically, the `Symbol`-class, which abstracts common parts of `TS` and `NTS`, as shown in Figure 15.

The abstract class holds the name of the symbol, e.g., *Expression*, *Term* or *Identifier*. The boolean flag `isExpandable` indicates whether a symbol or an inheritor is expandable, i.e., it is true for `NTS` and false for `TS`. The reason for this is that a symbol is only expandable if it can be replaced by its children, which is only possible for `NTS`. The flag is hard-coded into the constructors of `TS` and `NTS`. Next to the fields, an abstract function `first` is declared, which in its implementation should return the terminal beginners. These terminal beginners are nothing more than the symbols that an `NTS` or a `TS` can start with. Another advantage of the abstract class is that both `TS` and `NTS` can be stored in lists and other data structures at the same time.

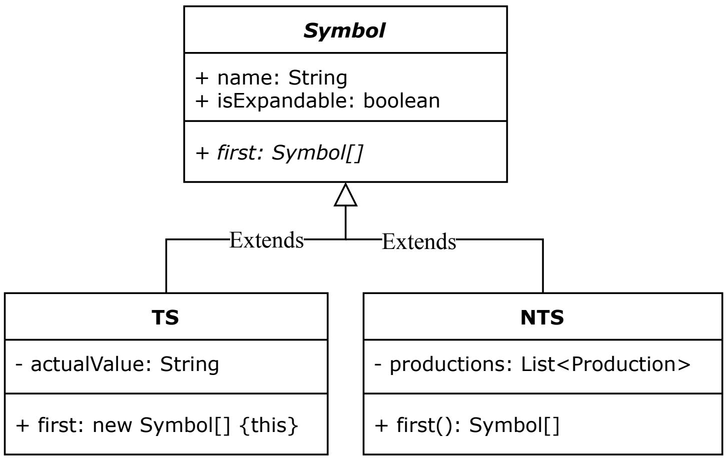


Figure 15: UML diagram for Symbol, `TS` and `NTS`.

The class for `TS` contains an extra field that holds the effective value of a symbol. The effective value is needed for terminal classes such as numbers. For example, the name of a `TS` is *number* and the effective value is *6*. Next to the field, the class overrides the `first` method, which returns the first symbol of the `Symbol`-object. In the case of a `TS`, this is itself.

The `NTS` class also provides an extra field. It is used to store the productions that can be used to create the `NTS`. The main use is to make it easier to traverse the data structure, since the `NTS` is aware which productions can be used to create it. It also overrides the `first` method. Unlike the `TS` class, the method of the `NTS` class must look in each of its productions. On each production's `NTS`, the `first` method is called, which causes a recursive descent until a `TS` is hit.

These are the very basic classes, which are used to form more complex structures, that are explained next.

4.1.2 Production and Item

The next two structures are the *Production* shown in Figure 16 and the *Item* shown in Figure 17. They can be created by the knowledge from the data structures explained in Section 4.1.1.

The Production class consists of three important fields. The first one is the number of the production, i.e., the position in the input grammar. The `Symbol[] symbols` stores all symbols of the production's right-hand side. For example, the production `X = a B c`, symbols stores three Symbol-objects, the `TS` *a*, the `NTS` *B* and finally the `TS` *c*. The stored `NTS` in `nts` is a reference back to the corresponding `NTS` instance from Section 4.1.1. The reference allows us to also traverse the data structure in the other direction.

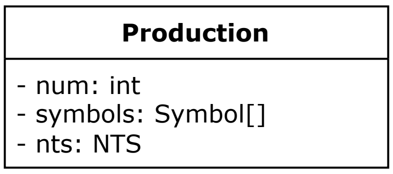


Figure 16: UML diagram for the production class.

An Item is an "evolution" to the production, but behaves different enough that the decision was made to not use inheritance. Instead, the item has a dedicated field that stores the production it represents. Other than that, the item has a few more fields. The flag `isCore` indicates if the item is a core item of its respective state. The `position` indicates the position of the parser within the item, with the initial value being 0, or in a more figurative description, right before the first symbol of the production. The action needed to process further through the item is also stored here (for example `SHIFT a`, but the actual detection is performed later on and is explained in Section 4.3.2. The last field contains the item's follower set, i.e., which look-aheads are valid for the item's final `REDUCE` action. The set is passed at construction, but may gain additional followers during the processing stage when the table is transformed from an LR(1) to an LALR(1) table. The `moveStep()` function creates a copy of the item, and moves the position one step forward (which updates the position and the action). At last, the class contains a `calculateFollow()` function, which calculates followers for the item and will be described in more detail in Section 4.3.1. However, as a brief explanation, the `calculateFollow` calculates the followers for a moved item based on the current context, i.e., position of parser and followers of the item.

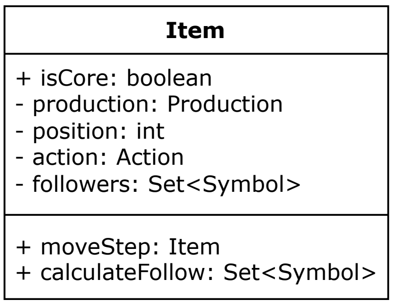


Figure 17: UML diagram for the item class.

4.1.3 State and Parser Table

Following, we explain the `State` class and the `ParserTable` class. These classes represent the top of the entire data structure.

The State class consists of two fields, as illustrated in Figure 18. The first field denotes the state number to be able to tell different states appart. The latter field, `items`, comprises a list that holds all items that make up the state. Additionally, the class offers a public method called `buildHull` along the corresponding private method for recursive descent. These two methods will be further elaborated upon in Section 4.3.1. The `setGuide` method is used and explained in Section 4.3.2. The final method is `getAnchor`, which returns a set of `TS` that is used for error handling. This process is explained in Section 4.8.1.

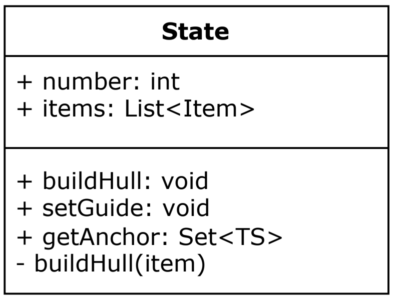


Figure 18: UML diagram for the state class.

The parser table class exists as a single instance, responsible for all direct and indirect computation regarding table generation. The class consists of two fields and three functions, as depicted in Figure 19. The first field is for the grammar (represented as an array of `Production` objects). This is demonstrated in Section 4.2. The other field, `states`, is the list of the table's states that fills up during table generation, as described in Section 4.3. The `detectAction` method detects the possible action for each item in a state, and it is placed in the parser table class, as it requires information from all other states. The `createShift` method also requires insight on all other states in order to create a correct result, and is therefore also placed in the parser table. A more detailed explanation is given in Section 4.3.2. At last, the `build` method is called by the constructor and internally utilizes a loop to build the hull and detect actions for each state.

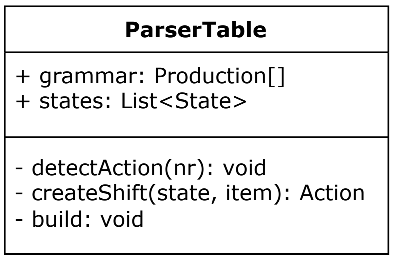


Figure 19: UML diagram for the parser table class.

4.1.4 Action and Simulation Step

Last but not least, there are two additional classes that serve as a wrapper to bundle up information.

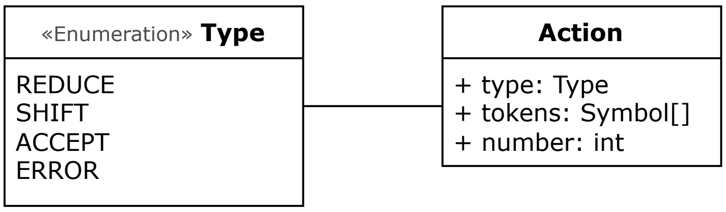


Figure 20: UML diagram for the action class.

The Action class represents an action and includes three fields, as shown in Figure 20. The `Enumeration` distinguishes between the different types of actions, i.e., `SHIFT`, `REDUCE`, `ACCEPT` or `ERROR`. The `tokens` field holds symbols depending on the specific type of action. For a `SHIFT` and `ACCEPT` the symbol to consume is stored, while for `REDUCE` all possible `TS` for the reduction, derived from an item's followers, are stored. The number has varying meaning depending on the type of the action. When performing a `SHIFT`, the number represents the number of the target state. When performing a `REDUCE`, then the number represents the number of the production based on which we reduce.

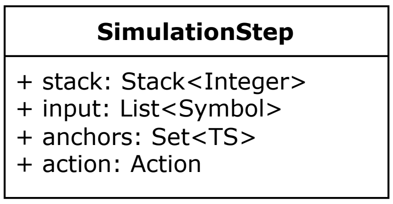


Figure 21: UML diagram for the simulation step class.

The SimulationStep class ends this chapter. The class holds all relevant information for a simulation step taken in Section 4.7 and contains four fields as shown in Figure 21. The stack holds the taken path up to the current step, simulating the input. The input list contains the remaining input during normal simulation or the guide symbols when performing error recovery.

The anchors set contains the anchor symbol during error recovery, and the last field holds the action performed in the step.

4.2 Import from File

The input is a simple text file (`.txt`) and uses a special format for input, as already briefly mentioned in Section 3. The sections are grammar, lexical structure and simulation inputs. These different blocks are shown in Figure 22 and are differentiated by the empty line separating the individual blocks.

```
Path = Dirs Name .
Dirs = Dir .
Dirs = Dirs Dir .
Dir  = Name "/" .
Name = an .
Name = Name an .

an   = letter | digit .

home/user/file
etc/config
```

Figure 22: Example content of an input file

The input is read as a regular text file and gets split by the empty lines. Afterwards the special `TS '#'`, mentioned in Section 2.3.2, is added at the end of each simulation inputs. Then the input is passed to Section 4.2.1 for conversion to the internal object structure.

4.2.1 Converting Text-based Grammar to Object-based Grammar

Since the tool itself can't perform the task of generating a parser table and state-transition-table from the text, the text must be converted into the object structure shown in Section 4.1. This means that the text must be converted into `TS` and `NTS`, which can then be used to create a production for each line of the grammar. It is important to note, that productions for the same `NTS` must be sorted by their length. The reason for this is that the whole creation of the parser table depends on this order and will therefore produce different results.

To accomplish the task of converting the input, the tool must first know which `NTS` exist in the grammar. Finding the `NTS` is easy because they must be defined in a production, and the fact that the left-hand side and the right-hand side of a production are separated by an equal sign makes it easy to extract. It is a matter of looping through all the lines of the grammar, splitting at an equal sign, taking the object at index zero, and then removing potential whitespace characters with a `.trim()` command. This process results in an array of `NTS`, which are then used to parse the productions.

To build the productions, the grammar is looped again, but already with the knowledge of the existing `NTS`. This time, the right side of the equal sign is split by whitespace characters to get individual symbols. The individual symbols are then compared to the list of `NTS`, and if found, the symbol is a NTS and is treated as such. However, if the symbol is not in the list, it is treated as a TS. The symbol (which is either a NTS or a TS) is then added to the list of symbols on the right side of the corresponding production.

At the end, the so-called *synthetic start symbol* is created. It is simply a new production of an NTS, which is the start-NTS with an apostrophe added. An example of such a synthetic start NTS is shown in Figure 23. This synthetic start symbol will be added at the top production, to be used as the new start for building the parser table. The corresponding definition is then the start NTS followed by the hash sign. The main idea behind the synthetic start symbol is, that it is easier to detect if the simulation on an input ended successfully.

```
Path' = Path # .
```

Figure 23: Example of a synthetic start NTS created from `Path = Dirs Name .`

The complete result is now the complete grammar in an object-based form, which is used in Section 4.3 and Section 4.4 to create the tables.

4.3 Generate Parser Table

In this chapter, the focus is on generating the parser table again. Instead of the theoretical part, which was covered in Section 3, the focus will be the explanation based on a concrete example. The productions are shown in Figure 24, which are imported using the technique presented in Section 4.2. To facilitate easy reference and comprehension, take note of the numbering in the parenthesis at the beginning of each production.

```
(0) Path' = Path # .
(1) Path = Dirs Name .
(2) Dirs = Dir .
(3) Dirs = Dirs Dir .
(4) Dir  = Name "/" .
(5) Name = an .
(6) Name = Name an .
```

Figure 24: Example productions for the explanation of parser table and state-transition-table generation.

From the productions, the parser table is generated. The algorithm can be divided into two blocks, one for building the hull and the other one for the detection of possible actions. The two blocks are sequentially applied for each existing and/or newly created state, as visualized in Figure 25.

4.3.1 Building the Hull

In the following, we will explain the hull building algorithm. An overview in the form of a flowchart diagram is shown in Figure 26. For ease of understanding, this section will use several examples based on the input from Figure 24.

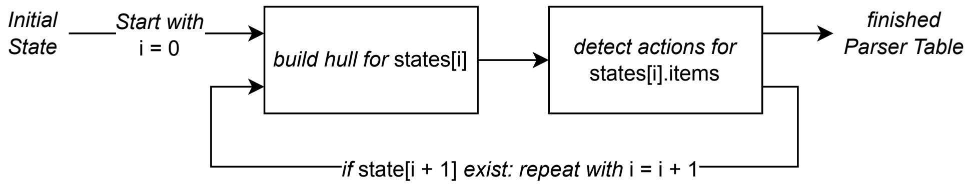

Initial State
Start with i = 0
build hull for states[i]
detect actions for states[i].items
finished Parser Table
if state[i + 1] exist: repeat with i = i + 1

Figure 25: Visualization of building the parser table.

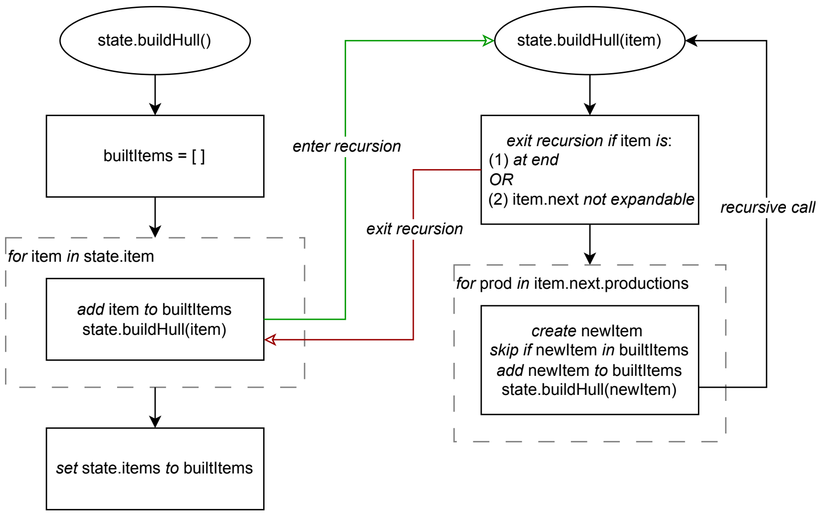


Figure 26: Visualization of the algorithm to build the hull of a `State`.

At the beginning of the hull-algorithm, an empty list called `builtItems` is created. The list will contain all the original and derived items of the state at the end. Initially, the list contains all the core items of the state from which the hull is built. The initial state zero, before the hull is built, is shown in Table 4. Only the start symbol `Path'` can be seen, because the algorithm has not performed any major actions yet.

Core	Production	Follower	Action	Guide
|	Path' = . Path # #			

Table 4: Initial state zero with only the start symbol `Path'` as a core item.

To fill said list, the algorithm iterates over all core items already in the newly created list. The first step is to mark the item as built to avoid duplicate processing. Then the algorithm enters a recursive descent by calling the `buildHull` function with the currently iterated item. The first step within the recursive descent is a series of checks. The checks include the following exit conditions:

- The parser has reached the end of the item.
- The next symbol (which is the symbol after the dot) is not a `NTS` and therefore not expandable.

Otherwise, the next symbol is a valid `NTS`. Therefore, the algorithm expands the `NTS` by iterating over the productions for the given `NTS`. The rest of this chappter provides a step-by-

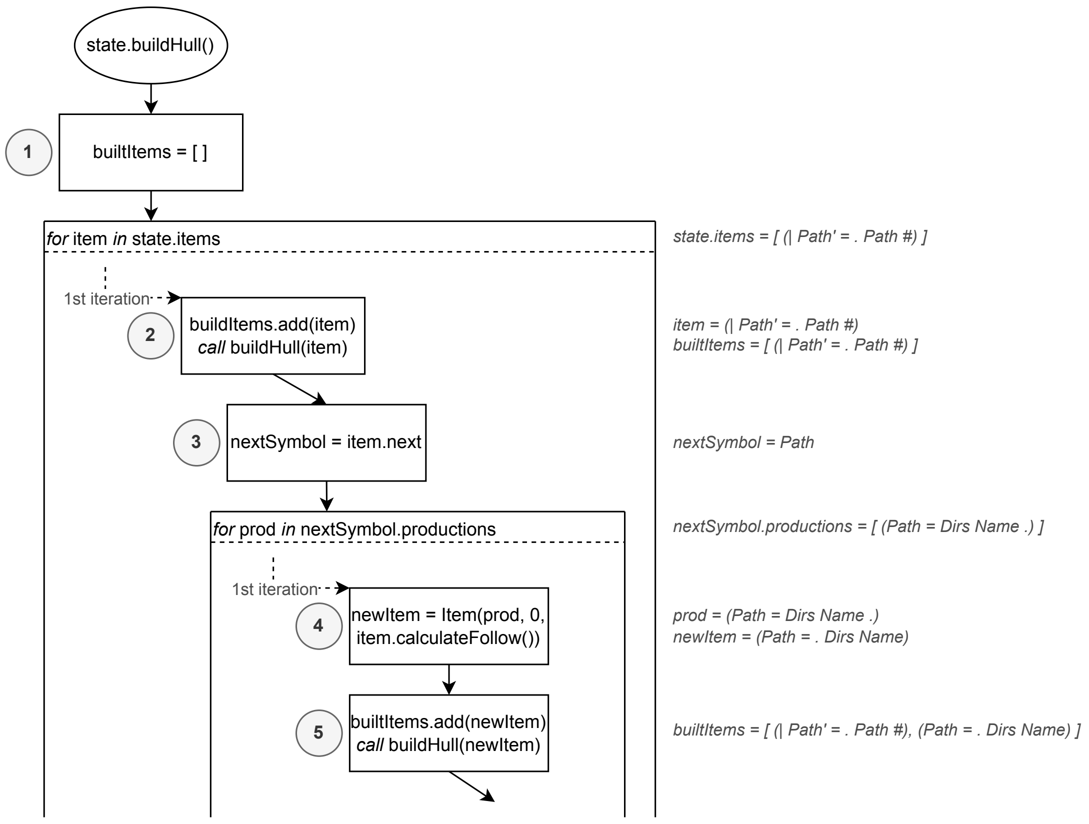

state.buildHull()
1
builtItems = [ ]
for item in state.items
state.items = [(| Path' = . Path #)]
1st iteration
2
buildItems.add(item)
call buildHull(item)
item = (| Path' = . Path #)
builtItems = [(| Path' = . Path #)]
3
nextSymbol = item.next
nextSymbol = Path
for prod in nextSymbol.productions
nextSymbol.productions = [(Path = Dirs Name .)]
1st iteration
4
newItem = Item(prod, 0, item.calculateFollow())
prod = (Path = Dirs Name .)
newItem = (Path = . Dirs Name)
5
builtItems.add(newItem)
call buildHull(newItem)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name)]

Figure 27: Flow of `buildHull` for start and first two recursive descents.

step walk-through of the hull build phase based on the state hero from Table 4. The complete diagram is shown in Section 9 as Figure 40 and Figure 41. A part of with the first two recursive steps are shown in Figure 27 and is split up into the following parts:

1. Create an empty list `builtItems` which is used to mark items as built and avoid duplicated executions. Afterwards, the algorithm starts iterating over all items in `state`. The only item of `state.items` is the core item itself, which implies that only a single iteration is performed for this loop.

2. The core item `| Path' = . Path #` gets added to the list of processed items and starts the recursive descent by calling `buildHull(item)`.

3. Store the symbol that gets processed next, which at the current step is `Path`. The algorithm checks if the symbol triggers any exit conditions. As none are triggered here, the algorithm starts iterating over the products that make up said `NTS`. Since there is only the product `Path = Dirs Name .` for `Path`, the loop also performs one iteration.

4. Create a new item for the iterated product. The item has to start at position zero, and the followers are calculated with the algorithm from Figure 28.

5. Add the newly created item into the `builtItems` list, which at this step now holds two entries. Afterwards, another level of recursive descent gets entered by calling `buildHull` with the new item from step 4.

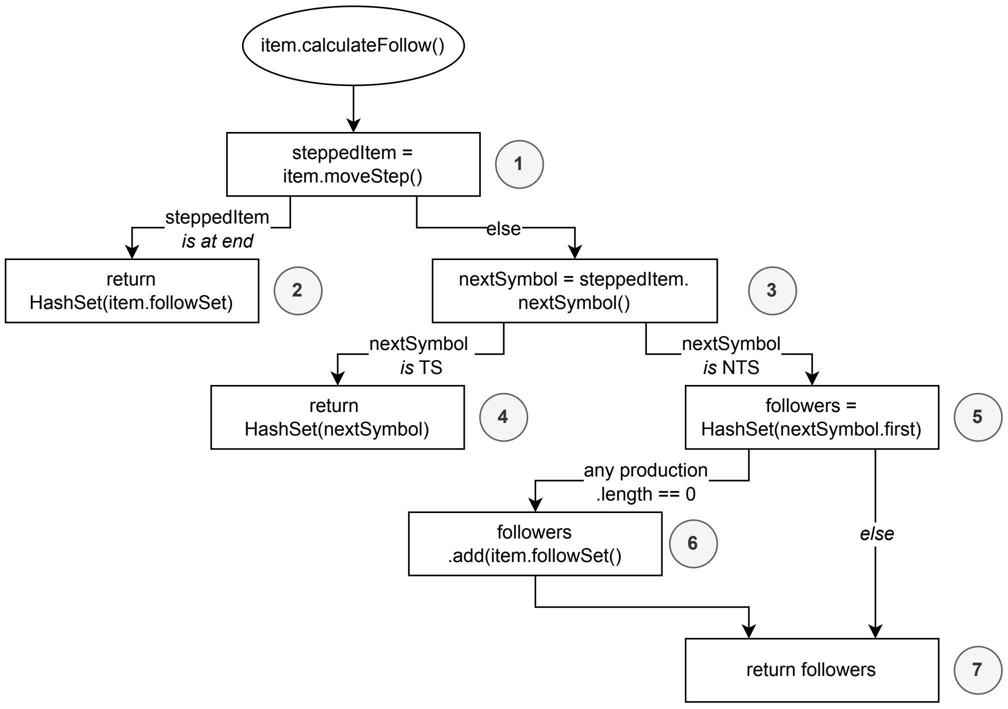

item.calculateFollow()
steppedItem = item.moveStep()
1
steppedItem is at end
else
return HashSet(item.followSet)
2
nextSymbol = steppedItem. nextSymbol()
3
nextSymbol is TS
nextSymbol is NTS
return HashSet(nextSymbol)
4
followers = HashSet(nextSymbol.first)
5
any production .length == 0
followers .add(item.followSet()
6
else
return followers
7

Figure 28: Flow of the calculation of followers for a certain item.

As already mentioned in the fourth step of Figure 27, the followers are calculated with the algorithm from Figure 28. The followers depend on the context of the item, i.e., in which state the item is in. The algorithm is shown in Figure 28 and is split up into the following steps:

1. Temporarily store the item as if the parser had moved forward by one step.
   E.g., `Path = . Dirs Name` results in `Path = Dirs . Name` being stored.

2. If the parser's position is at the end, the follower set of the production.
   E.g., `Path = Dirs Name .` returns `#` as the single entry in a set.

3. If the parser's position is at the end, then the algorithm fetches the next symbol from the temporary item and performs a type check.
   E.g., in the temporary variable from step 1, the checked symbol is `Name`.

4. If the symbol is a `TS`, then the symbol is returned as part of a set.
   E.g., the temporary item `Dir = Name . "/"` returns the literal `"/"`.

5. If the symbol is a `NTS`, then the followers of the next symbol are stored into a temporary variable. The reason is, that the follower set may get expanded in step 6.

6. If there exist any production for the `NTS`, where the length is zero, it is called a deletable production. In this case, the followers of said symbol are also added to the set stored in step 5.

7. The temporary variable from step 5 is returned.

The descent with steps 6 to 11 is very similar to step 3 to 5 performed twice in each other. At step 12, the recursion reaches the loop with the symbol `Name` as shown in Figure 29. In the first iteration processes production `Name = an .` where steps 13 and 14 are similar to the ones seen in the explanations before. The exception here is step 15, which retrieves the `TS` alpha-numeric (`an`). The symbol triggers the second exit condition, as it cannot be expanded any further.

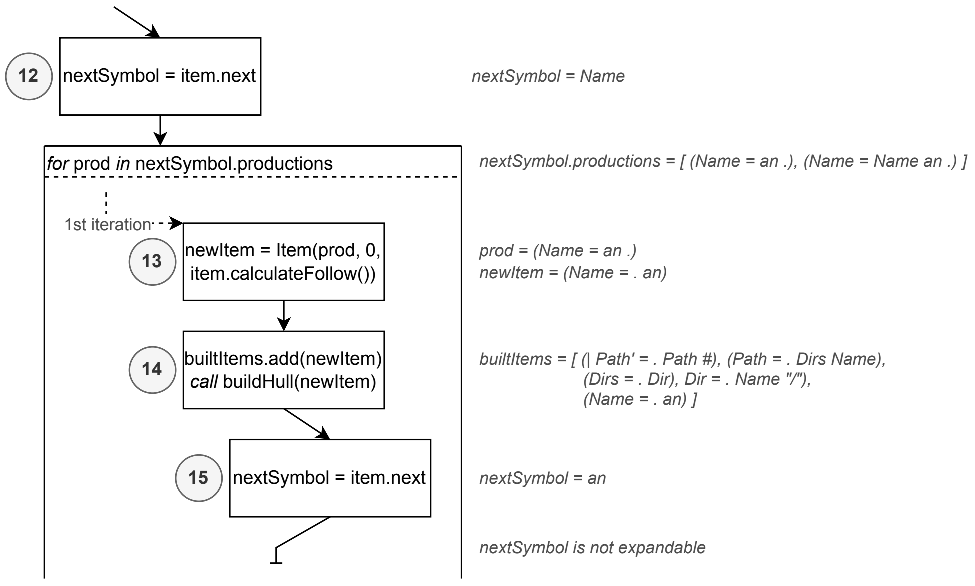

12
nextSymbol = item.next
nextSymbol = Name
for prod in nextSymbol.productions
nextSymbol.productions = [(Name = an .), (Name = Name an .)]
1st iteration
13
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = an .)
newItem = (Name = . an)
14
builtItems.add(newItem)
call buildHull(newItem)
builtItems = [(\| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir), Dir = . Name "/"), (Name = . an)]
15
nextSymbol = item.next
nextSymbol = an
nextSymbol is not expandable

Figure 29: Flow of `buildHull` for the loop over `Name`.

The second iteration shown in Figure 30 processes `Name = Name an .` with another recursive descent due to the left recursion in the production. The recursion processes the `NTS Name` and it's productions again. There both iterations are stopped prematurly, as each iteration's `newItem` is already contained in `builtItems`. If this check would not be in place, it would cause an infinite recursion and eventually cause a crash of the tool.

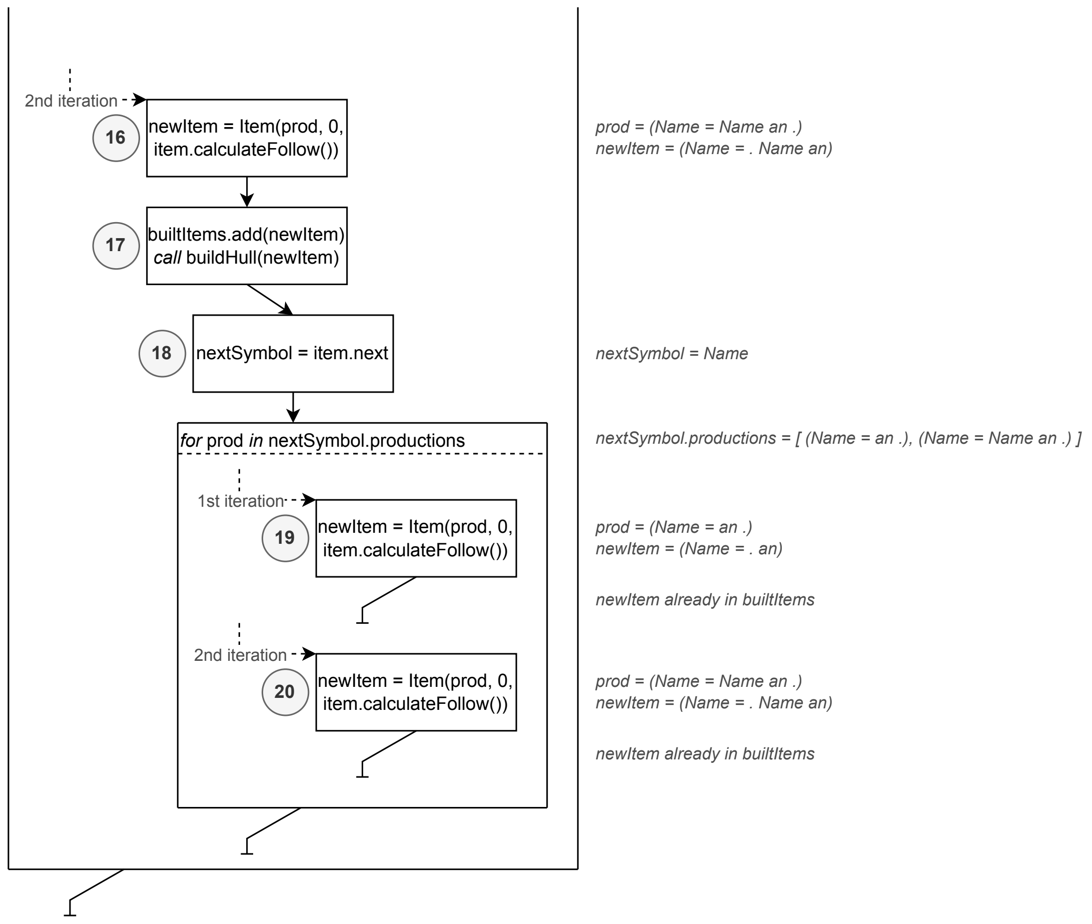

2nd iteration
16
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = Name an .)
newItem = (Name = . Name an)
17
builtItems.add(newItem)
call buildHull(newItem)
18
nextSymbol = item.next
nextSymbol = Name
for prod in nextSymbol.productions
nextSymbol.productions = [(Name = an .), (Name = Name an .)]
1st iteration
19
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = an .)
newItem = (Name = . an)
newItem already in builtItems
2nd iteration
20
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = Name an .)
newItem = (Name = . Name an)
newItem already in builtItems

Figure 30: Flow of `buildHull` for the loop over `Name` with already built items.

Similar to Figure 30, the second iteration of `Dirs` also contains a left recursion. The recursion in step 23 also performs loops over already built items. From here on, the recursive part of the algorithm fully retreats and the algorithm exits the the loop created in step 1.

After step 15 every relevant item of state zero has been processed. After the loop, the algorithm exchanges the existing items of the state, with the ones from the `builtItems` list. Table 5 shows the items of state zero from what has been performed up until now.

However, the table is not done yet. When looking at the rows of the productions for the `Name`, there seems to be missing something with the followers. The missing part is the `TS` for the alpha-numeric (`an`). The symbol is missing due to the recursion being present within the production `Name = Name an .`. To fix such issues, the algorithm has to perform an action called *merging*. This takes advantage of the properties from the LALR(1). Instead of expanding the recursive part again, the algorithm expands and then merges the item's followers with the already existing item. Despite that, the implemented algorithm performs things a bit different, due to issues with finding indirect recursions.

The implemented algorithm shown in Figure 31 runs a loop-based-check with a *didChange*-

Core	Production	Followers	Action	Guide
|	Path' = . Path #			
	Path = . Dirs Name	#		
	Dirs = . Dir	an		
	Dir = . Name "/"	an		
	Name = . an	"/"		
	Name = . Name an	"/"		
	Dirs = . Dirs Dir	an		

Table 5: State zero after running recursive part of `buildHull()`.

flag. The main loop runs as long as changes have been made. Inside the main loop, two other cascaded loops span over the state's items. The two cascaded loops form a matrix of all combinations of source and targets for merges. Within the loop for the source, the algorithm filters out any items that are core items, since these items are the base of building a hull and therefore were already a source. At the same position, it is also checked if the item's next symbol is a `NTS`. This check is performed as merging is not required if the next symbol is a `TS`. On the very inside, the actual check for the compatibility to merge are performed. The algorithm checks if the next symbol of the source matches the `NTS` of the target and if the compatibility is verified, the actual merging is performed. The merge process takes the source's next symbol and from that calculates the followers. These followers are then added to the target's follower set. After adding the followers to the target, the flag is set to allow for another round of merging. This approach is definitely not very efficient and there are better strategies to accomplish merging. But as the amount of productions used in the lecture's homework manageable, the explained method is acceptable.

When applying the merging on the output from Table 5, the algorithm detects that the followers of the item `Name = . Name an` can be merged into `Name = . an`, which in this case is the symbol for an alpha-numeric (`an`). The same is true for the target `Name = . Name an`. The result after the merge operation is shown in Table 6.

Core	Production	Followers	Action	Guide
|	Path' = . Path #			
	Path = . Dirs Name	#		
	Dirs = . Dir	an		
	Dir = . Name "/"	an		
	Name = . an	"/", an		
	Name = . Name an	"/", an		
	Dirs = . Dirs Dir	an		

Table 6: State zero after building the hull and merge operation.

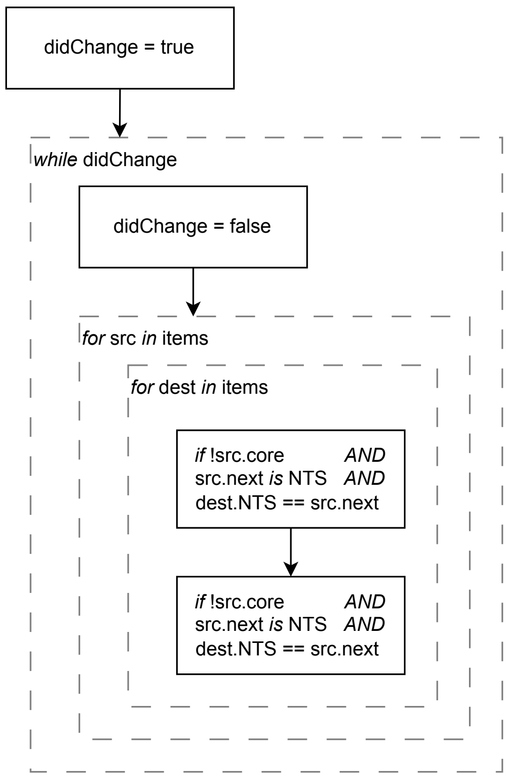


Figure 31: Flow of merge process after recursive `buildHull` calls.

4.3.2 Detect Possible Actions

With the hull built in the last chapter, in this one the focus is on detecting the possible actions. Furthermore, the so-called *guide* symbol is also detected in this part of the process. The guide symbol is used for error handling and is explained in Section 4.8.

An overview of the detection step is shown in Figure 32. The examples that the explanation uses, are based on the grammar from Figure 24 and state zero with a completed hull from Table 6. The function, which performs the detection, is placed in the `ParserTable`-class. The reason is, that the internally used function `createShift` requires access to the whole parser table in order to find and reuse existing actions in other states.

The process starts by retrieving the state that gets processed. The state (`curState`) is the same that the hull was built, as shown in Figure 25. Then the algorithm starts iterating over the items contained in the state. Inside the loop, the item's next symbol is stored into the temporary variable `nextSym`, and three checks are performed on it as listed:

1. If the `nextSym` is the indicator symbol `#`, then the end of the synthetic start symbol has been reached. The only action here is the `ACCEPT`, which does not need any parameters.

2. If the parser's position of the `curItem` is at the end of item, a `REDUCE` action is possible. To get the index of the production, the algorithm uses the `.indexOf(...)` function on the array with the loaded grammar. The passed argument is the production of `curItem`. The

followers of `curItem` are used as the guide symbols. Both the index and the guide symbols are used to create a `REDUCE` action.

3. In any other case create a `SHIFT` action. The implementation uses the helper function `createShift`. The function takes the current state and current item as arguments and is

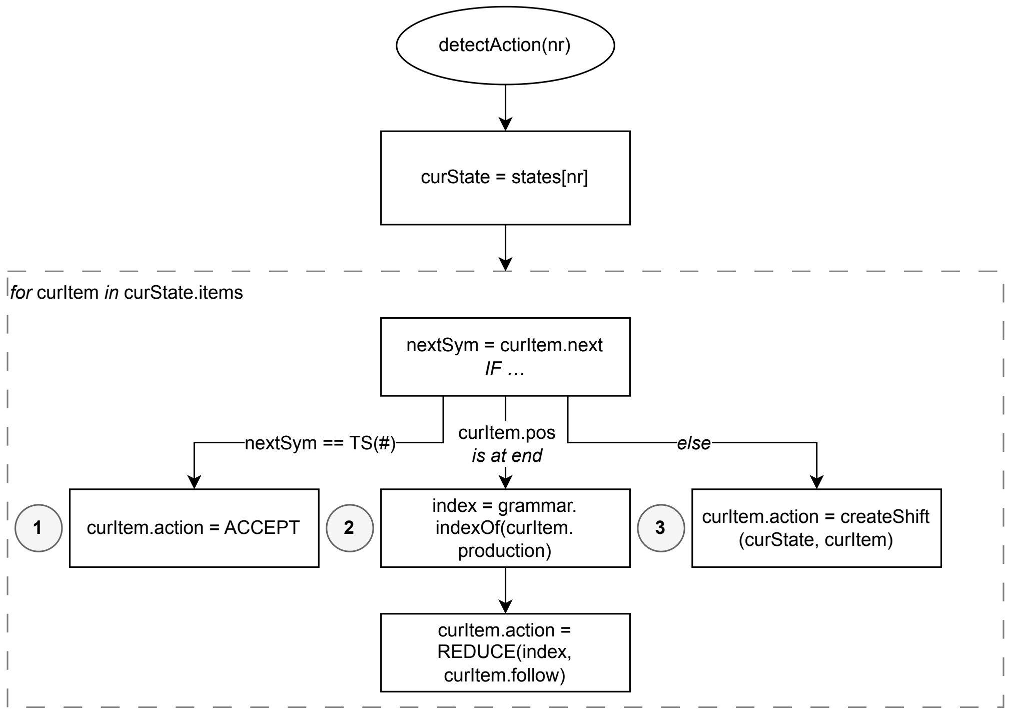


Figure 32: Overview for detecting actions.

Creating the `SHIFT` action is not as straightforward, as the other two actions, as already briefly mentioned in Section 2.3.2. For this reason, the algorithm works in three phases to find a suitable action. The first one is to check if there is already a reusable action in the local state. The second phase checks if there is a reusable action in the parser table. The last phase creates a new state and matching action.

Let's start with the first phase, which checks if there is already a reusable action in the local state. Figure 33 shows how the phase handles the process and is split up into the following steps:

1. Create a flag called `found` and set it to false by default. The flag indicates if a reusable action has been found. Afterwards, start iterating over all items of `curState`.

2. Check if `other` has an action, and if the next symbol of `other` is equal to the next symbol of `curItem`. Only continue if both checks are successful.

3. At this step, a reusable action has been found. The flag is set to `true` and set the action of `curItem` to the action of `other`. Temporary store a moved version of `curItem` as `next`.

4. Check if `next` is already contained in the successor state. If this is not the case, continue with step 5.

5. Add `next` to the successor state.

6. After the loop, the flag is checked. If the flag is set, no further action is required and the `createShift` function returns. Otherwise, the algorithm continues with phase 2.

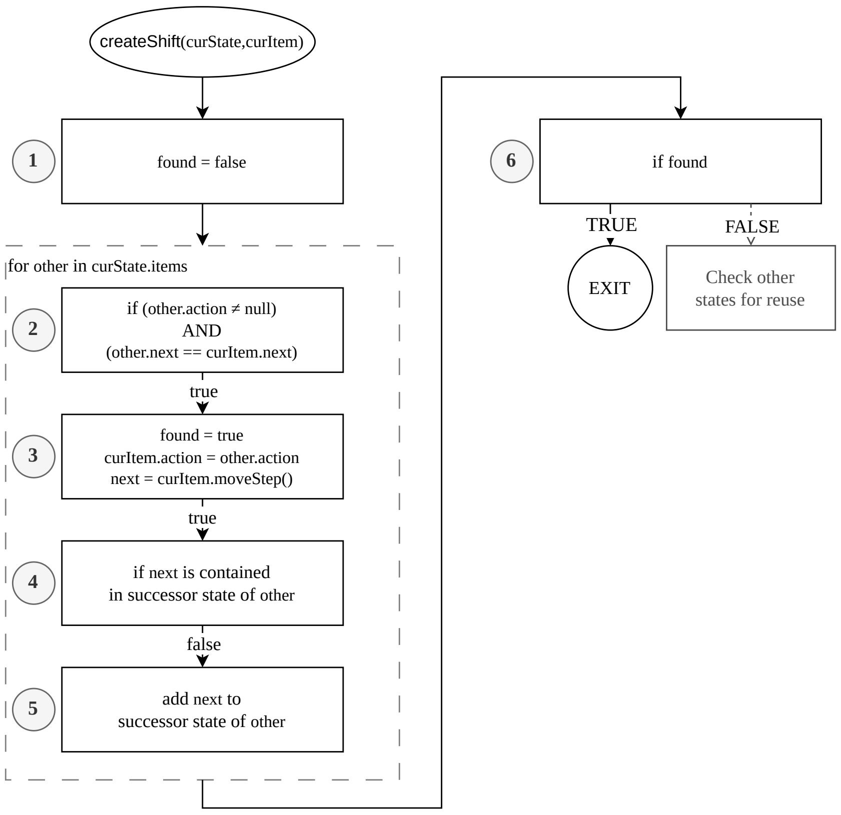


Figure 33: Phase 1 – Search for reusable action in local state.

Phase two is shown in Figure 34 and begins by creating a temporary state for the moved `curItem`. The algorithm then iterates over all states but skips the `curState` because it has already been processed by phase one. The temporary state is then used to search for similar states that already exist in the parser table. A state is similar to another state if both contain the same core. The term same core for states means, that at least one core item exists in both states. However, the items must be identical only from the beginning to the position of the parser on the item.

Table 7 shows an example of two states with an equal core. The example table is based on a different grammar, since the grammar of Figure 24 does not contain such a case. Nevertheless, the example shows that both states do indeed contain a core item that satisfies the requirement of being equal from the beginning to the parser position. In other words, the part between the

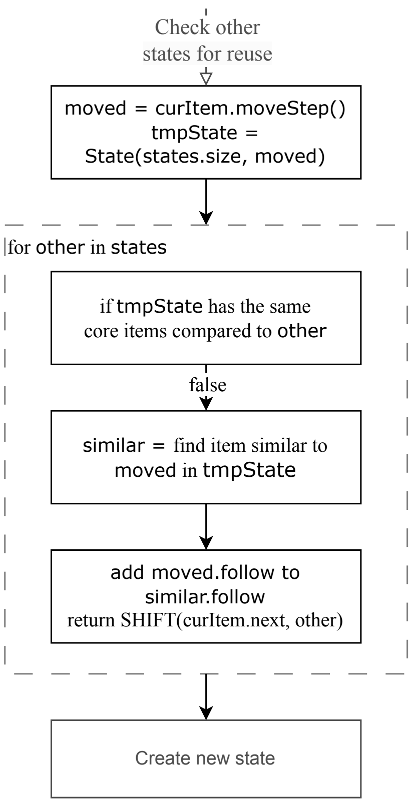

Check other states for reuse
moved = curItem.moveStep()
tmpState =
State(states.size, moved)
for other in states
if tmpState has the same core items compared to other
false
similar = find item similar to moved in tmpState
add moved.follow to similar.follow
return SHIFT(curItem.next, other)
Create new state

Figure 34: Phase 2 – Search for reusable state in parser table.

equal sign and the parser position is the same. With this terminology cleared, the algorithm reuses the state that satisfies the equal core. The temporary state is dropped because it is no longer needed. Instead, the reused state is used for further operations. The moved `curItem` from the beginning of phase two is added to the reused state at the end of the phase.

StateNr	Core	Production
1	|	Dir = "." . "/"
2	|	Dir = "." . "." "/"

Table 7: Example, for two states with the same core.

Phase three is a very quick one, as the only option left is to create a new state. The new state is the same as the temporary one from phase two. But this time, the state is stored into the list of states in the parser table. Last but not least, the `SHIFT` action that references the newly created state is created returned, to be set as the action for `curItem`.

The result of applying the detection on Table 6 is shown in Table 8. A new state has been created for each item except the last one. In phase one, the algorithm is able to detect the action for `Dir = . Name "/"` and reuse it for `Name = . Name an`. The same is true for `Path = . Dirs Name` and `Dirs = . Dirs Dir`.

Core	Production	Followers	Action	Guide
|	Path' = . Path #		SHIFT Path 1	
	Path = . Dirs Name	#	SHIFT Dirs 2	
	Dirs = . Dir	an	SHIFT Dir 3	
	Dir = . Name "/"	an	SHIFT Name 4	
	Name = . an	"/", an	SHIFT an 5	
	Name = . Name an	"/", an	SHIFT Name 4	
	Dirs = . Dirs Dir	an	SHIFT Dirs 2	

Table 8: State zero after building the hull and merge operation.

The last thing to fill in is the `Guide` column. These symbols are used to calculate the escape route while the simulation performs error handling, which is described in Section 4.8. Since these guide symbols are the fastest way to escape the state, the algorithm takes advantage of the implicit sorting of items in a state. The implicit sorting is a result of the input grammar being sorted by length when a `NTS` has more than one production.

Figure 35 provides an overview of the process. The algorithm loops over all items of the given state until a suitable guide symbol is found. Otherwise, the algorithm ran into a problem and the tool exits with an exception. A guide symbol is suitable if

- the action is an `ACCEPT`.
- the action is a `REDUCE`.
- the action is a `SHIFT`, with the symbol being a `TS`.

If the action is an `ACCEPT` or a `SHIFT` with a `TS` symbol, the stored symbol is returned as the guide symbol. If the action is a `REDUCE`, then the algorithm has to check if one of the symbols contained in the action is a `TS` with the value `#`. The reason for this is that the fastest exit is desired, and the exit of the synthetic production is most desired. So if such a `#` is found, it will be used as the guide symbol. Otherwise, similar to the other actions, the first symbol of the action is used.

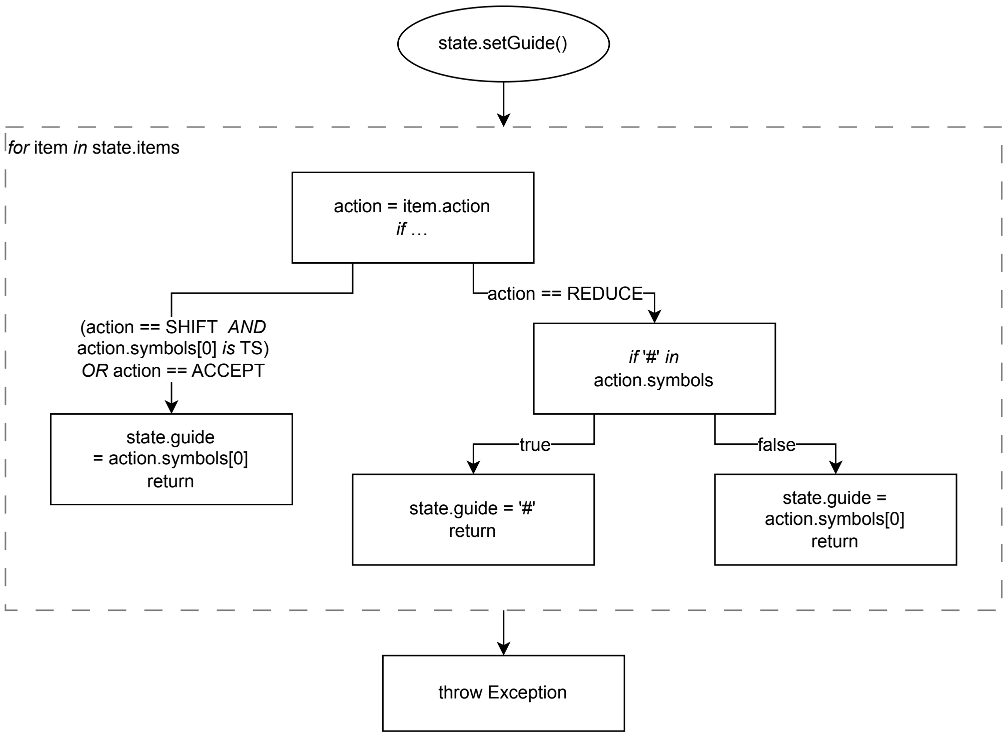


Figure 35: Detection of the guide symbol

Table 9 shows the zero state from Table 8, but with the guide symbol also detected and set. The loop iterated over the first and second item without finding a suitable guide. The reason was the `NTS` used for the `SHIFT` action. The same is the case for the second, third and fourth item. The fifth item is found suitable because the symbol of the `SHIFT` is a `TS`. Therefore, the `TS` with the value `an` is used as the guide symbol for the whole state zero.

Core	Production	Followers	Action	Guide
|	Path' = . Path #		SHIFT Path 1	an
	Path = . Dirs Name	#	SHIFT Dirs 2	an
	Dirs = . Dir	an	SHIFT Dir 3	an
	Dir = . Name "/"	an	SHIFT Name 4	an
	Name = . an	"/", an	SHIFT an 5	an
	Name = . Name an	"/", an	SHIFT Name 4	an
	Dirs = . Dirs Dir	an	SHIFT Dirs 2	an

Table 9: State zero after the complete state is built.

4.4 Generate State-Transition-Table

The generation of the state-transition-table starts off with calculating how much columns are needed. Compared to the parser table explained in Section 4.3, the state-transition-tables has a variable amount of columns. The amount depends on the amount of `TS` and `NTS`. In addition to the columns for the symbols, two columns are certain, namely the columns for the *state number* and the guide. Each row represents a state, and each column is dedicated to a potential symbol found. The combination of the rows and columns shows certain transition depending on the action in a cell.

To populate the state-transition-table, the algorithm proceeded in an iterative fashion, compared to the recursive approach of the Section 4.3. In the iteration, the already known state number and guide are set. Next, the algorithm iterates over all items of the processed state. For each item, the action is looked at, and then decided where to put which entry. The actual format depends on a program argument, but the basic schema is the same for all. In the case of `ACCEPT`, the entry is a simple "accept". Almost the same is done for `SHIFT`, which is a "shift" followed by the number of the state it references to. If the action happens to be a `REDUCE`, there is not only one entry to be made. Potentially, multiple `REDUCE` can be performed within a single state. Therefore, each symbol contained in the `REDUCE` is looped over, and for each single one an entry. Such an entry is indicated by a "reduce" followed by the number of the production in parantheses. The formatting options include all upper-case, all lower-case, or the first letter upper-case.

The algorithm applied on state zero from Table 9 is shown in Table 10. It can be clearly seen which `SHIFT` occurs depending on the look-ahead symbol.

StateNr	"/"	an	#	Dir	Dirs	Name	Path	Path'	Guide
0		shift 5		shift 3	shift 2	shift 4	shift 1		an

Table 10: State-transition-table for state zero.

4.5 Parser for Simulation Sentences

As already briefly mentioned in Section 3 the simulation sentences have to be parsed into compatible `TS`. However, these `TS` are not fixed at compile-time of the tool and are subject to change depending on the assignment the lecturer wants to create. For this reason, the tool creates a scanner and parser at run time.

4.5.1 Using Coco/R for Sentence Parsing

To make life easier, the program *Compiler Compiler generator Recursive descent (Coco/R)* by H. Mössenböck et al. [4] is used. Coco/R is able to generate a scanner and a parser from an *Attributed Grammar (ATG)* that describes the grammar to be parsed. Most of the structure stays the same, so a template for the ATG is used. This template contains a `Record` that holds a single `TS` and an `ArrayList` that holds all parsed `TS` records for later use. Since the `TS`s must be based on some characters, three atomic `TS` are declared by default. These declared values include a single `letter`, a single `digit` and a `sign`, that hold their respective set of symbols. The `TS` that are created from these atomic `TS` are placed in the `TOKENS` section of the ATG file without changing anything. The valid symbols consist of all `TS` and all literals that need to be added into the ATG and is explained in Section 4.5.2.

4.5.2 Injection of TS and Literals

The injection of TS and literals is relatively straight forward as they are easily extracted from the input. The symbols can be extracted from the second input block, that contains the user-provided TS. From every symbol on the left side of the equals-sign gets extracted. Filtering the literals from the input grammar is equally simple, since they are encapsulated by double-quotes by the convention used in the course.

The production that matches all these TS and literals requires them to be in a OR-chain, with each part having a special ATG code snipped. Each snipped creates and adds a corresponding record-instance, that holds information about the kind, if it is a literal and the actual value. An example of such an entry is shown in Figure 36 that matches all idents or literals that are an exclamation mark. The SimTS in the example is the record and the tss is the list of Record-objects mentioned in Section 4.5.1

```
X = {
  ident    (. tss.add(new SimTS("ident", false, t.val)); .)
  | "!"    (. tss.add(new SimTS("!",     true,  t.val)); .)
} .
```

Figure 36: Example for a production in Coco/R that matches an ident and an exclamation mark literal

To avoid errors when parsing a sentence that contains unexpected symbols, the catch-all type ANY is used. This special type is added at the very end of the OR-chain, and matches everything that was not recognized until the end. Matching the ANY allows parsing to continue and also stores information about which symbol was not expected. This information is then used for error handling, which is explained in Section 4.8.

The generated OR-chain production is then placed in the PRODUCTIONS section of the ATG file. The ATG file is now ready to be used by Coco/R, which is explained in Section 4.5.3.

4.5.3 Applying Coco/R to ATG file

To get the scanner and parser from Coco/R, two additional files are needed, namely Scanner.frame and Parser.frame. These files contain the basic structure from which the actual scanner and parser are generated. In order to avoid problems with duplicate files, accidental overwriting etc., the generation with Coco/R is performed in a temporary directory provided by the Operating System (OS). All necessary files are copied into such a directory. This includes the .frame files mentioned before, the Coco.jar and the generated ATG file. To execute the Coco.jar, a Process-object with arguments to the temporary directory is used.

4.5.4 Compile and Load at Run Time

After the generation of Scanner.java and Parser.java with Coco/R in Section 4.5.3, these files must be compiled and loaded into the current Java Virtual Machine (JVM) on which the tool is currently running.

To accomplish the task of compiling a `.java` file at run time in Java, the `JavaCompiler` class provides a convenient solution with its `CompilationTask` feature. This feature allows dynamic compilation of Java source code from a JVM. Such a task is created by specifying the path with the `.java` files. The compiled `.class` files are then stored in the same specified path and are ready for loading.

To load the `Scanner.class` and `Parser.class` at run time, another feature of Java is very convenient. The `URLClassLoader` allows loading classes by simply pointing to the required files. Loading a `.class` file returns a corresponding `Class<?>` instance. To use the scanner and parser, an instance of the classes must be created by using a *reflection*-like approach. The `Parse()` method can be extracted from the instance of the parser class. This method is used in Section 4.6 to parse the simulation sentences.

4.6 Parsing the Simulation Sentences

The scanner created by Coco/R offers two ways to pass a sentence. Either by a file which contains a sentence, or via a `InputStream`. For simplicity and some issues with `InputStreams` the chosen method for this tool is the file with a single sentence. A simulation sentence, which is initially loaded from the general input file, is saved into a text file in the temporary directory. With this temporary file, a scanner instance is created, which then is used to create a parser instance. The final step for parsing is to invoke the `Parse()` method of the parser instance.

The simulation sentence gets parsed and the list with the record-objects are now available and ready for further use. To get objects of the list, the records and the records' entries, another round of reflection-like access is required. These `SimTS` records are then transformed into actual `TS` objects by extracting the `kind` and the `val` fields. The actual `TS` objects are packed into a list and then used in Section 4.7 to simulate the sentence on the given grammar.

4.7 Simulating a Sentence

The simulation uses the state-transition-table generated in Section 4.4 and the parsed sentences from Section 4.6. The flow for the simulation of a single is shown in Figure 37.

The individual steps of the algorithm are as follows:

1. Create a new list `stepList`, which will hold all steps of the simulation. Furthermore, create an initial step with a stack that holds the initial state, the whole sentence as an input and the action for the initial state. Afterwards, start loop that runs as long as the action of `curStep` is not an `ACCEPT`.

2. Add the current step to the list of steps and start preparations for new step. The preparations include creating a copy of the stack and the input. The reason for creating copies is that changes later on would otherwise propagate back and cause incorrect results.

3. If the action of `curStep` is a `SHIFT`, the state of the action is pushed onto the stack and the first symbol of the input is consumed by dropping it.

4. If the action is a `REDUCE`, then the algorithm pops as many states from the stack as the production of the `REDUCE` has symbols. The `NTS` of the production is then added to the front of the input.

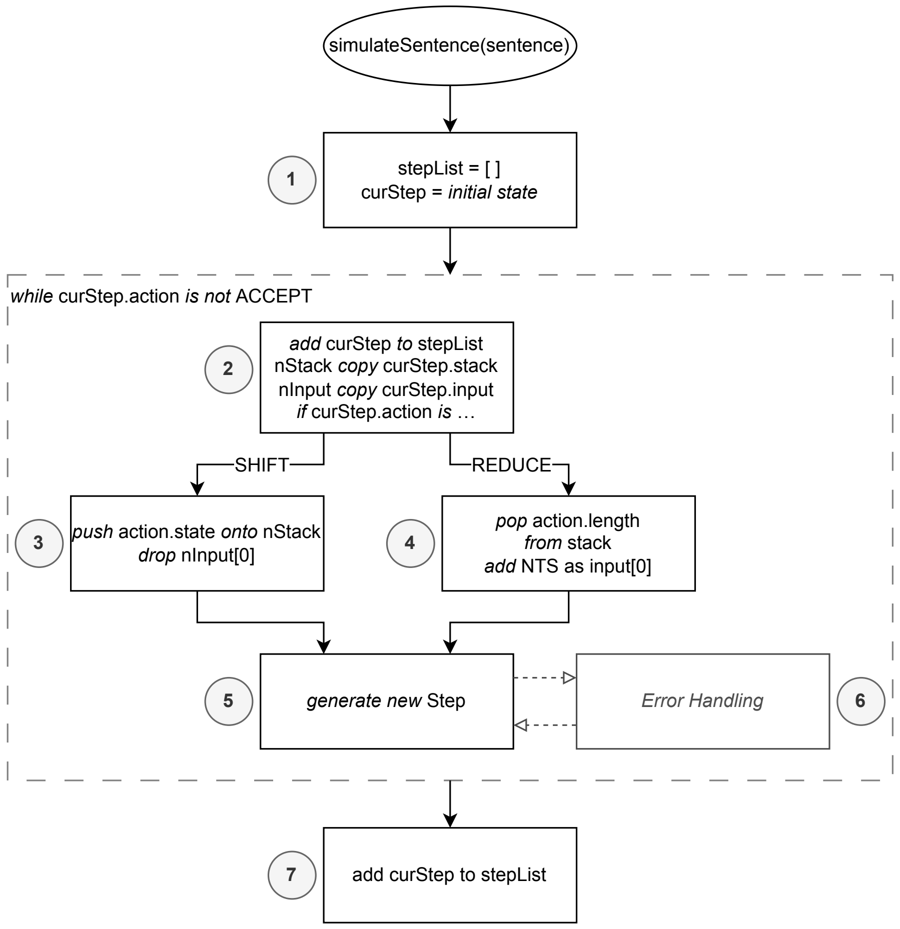


Figure 37: Program flow for the Simulation

5. With the `nStack` and `nInput` the algorithm creates a new step and adds it to the list of steps. The action of the new step is determined by the state-transition-table. The algorithm looks at the cell that is defined by the state of the stack and the first symbol of the input.

6. If the cell in step 5 is empty, the algorithm transitions to an error handling mode, which is explained in Section 4.8. The error handling then produces a new step which can be used for the next iteration.

7. After the loop, the last step is added to `stepList`, which is the `ACCEPT` step. The simulation is now finished and ready for export as explained in Section 4.9.1.

4.8 Simulation Error Handling

The lecture on Compiler Construction introduces an algorithm specifically designed to address exception handling. The taught algorithm is divided, into three parts. The algorithms structure can be briefly summarized as follows:

1. *Search for Escape Path*:
   Replace the input with the guide-symbol to navigate to the escape-state as fast as possible. Add the anchors of each step into a set for later.
2. *Delete Faulty `TS`*:
   Skip and remove all `TS` from the input until a valid symbol from the set of anchors is found.
3. *Insert missing `TS`*:
   Adds all read guide-symbols until a valid state is reached to the front of the input.

For explanation, the example input `"h!me/user"` is used. The faulty symbol in the input is the exclamation mark (`!`) at position 1, where a alpha-numeric (`an`) is expected.

The simulation steps up to the error are shown in Table 11 and has a single `SHIFT` to the fifth state. The next action already fails because of the exclamation mark. At this point, the exception handling starts with the calculation of the escape route, which is explained in Section 4.8.1.

Stack	Input	Action
0	an ! an an "/" an an an an `#`	SHIFT 5
0 5	! an an "/" an an an an `#`	ERROR

Table 11: Simulation Steps until failure due to unexpected exclamation mark.

4.8.1 Calculate Escape Route

From a programming point of view, the calculation of the escape route is very similar to the simulation itself. However, instead of sticking to the input, the escape route uses the guide symbol to find the shortest route to the end. This mainly means that the normal input is swapped with the guide symbols for the corresponding state. So the first step of processing is the step that led to the error, but with an `"#"` as input as seen in Table 12. In the case of escape route calculation, a `REDUCE` and a directly following `SHIFT` are combined into one step. In code, the merging of these `Actions` is done by a second iteration after the whole escape route computation.

Stack	Guide	Action	Anchor
*0 5	`#`	reduce (5), shift 4	"/", `an`, `#`
0 4	"/"	shift 8	"/", `an`
0 4 8	`an`	reduce (4), shift 3	`an`
0 3	`an`	reduce (2), shift 2	`an`
0 2	`an`	shift 5	`an`
0 2 5	`#`	reduce (5), shift 6	"/", `an`, `#`
0 2 6	`#`	reduce (1), shift 1	"/", `an`, `#`
0 1	`#`	accept	"/", `an`, `#`

Table 12: Escape route calculation for input `"h!me/user"`.

The set of anchor symbols can be easily derived, from the calculated escape route. This important for removing invalid symbols as described in Section 4.8.2. In the example, the anchor set contains `"/"`, `an` and `#`.

4.8.2 Remove Invalid Symbols

Removing invalid input symbols is straightforward. The input is looped over, and if the current `TS` is not in the set of anchors, the `TS` is removed from the input. Applying the information to the example in Table 11, the second step is to remove the exclamation mark from the input, resulting in the repaired input `"an an "/" an an an an #"`. The repaired input is used to continue with step 3 of the recovery process.

4.8.3 Insert Missing Symbols

To see if and which symbols to add to get a working input again, the algorithm has to check at which point it can reattach. This is done by iterating over the escape route again and checking at which step the next input symbol is included in the anchors of that step.

In the case of the example, the next input symbol is an `"an"` as calculated in Section 4.8.2. The symbol already appears in the anchors of the first step and can therefore be used as a reattachment point, as also indicated with an asterisk (`*`) in Table 12. Since there are no `SHIFT`s without a preceding `REDUCE`, no `TS` need be added. If there would be a `SHIFT` without a preceding `REDUCE`, the algorithm would have to add the symbol of the `SHIFT` to the input.

The modified input is used to create a new simulation step that has been reattached at the position of the error. The simulation is now able to continue as normal according to the algorithm in Section 4.7, which is partially shown in Table 13.

State	Input	Action
0 5	an an "/" an an an an #	reduce (5)
0	Name an an "/" an an an an #	shift 4
0 4	an an "/" an an an an #	shift 9
0 4 9	an "/" an an an an #	reduce (6)
0	Name an "/" an an an an #	shift 4
0 4	an "/" an an an an #	shift 9

Table 13: Simulation after reattachment

4.9 Export to File

Exporting is the last step performed by the tool and is explained in this section. The export is divided into two parts. First, the actual output of the tables and steps in the form of CSV and text files. The latter is the simulation log, which helps to understand the simulation process.

4.9.1 Export Parser Table, State-Transition-Table and Simulation Steps

The parser table from Section 4.3, the state-transition-table from Section 4.4, and the simulation steps from Section 4.7 are saved in a CSV format to allow a tabular view without the need for additional programs. The parser table from the example generated in Section 4.3 in form of an CSV is partially shown in Figure 38. The complete CSV is shown in Figure 42 Keep in mind that the CSV is formatted a bit to make it easier to look at.

In order to make concentrate on the implementation of the actual tool, the `OpenCSV` library by G. Smith et al. [5] is used to perform the write operations for the CSV files. The library allows

```
"Nr", "Core", "Item", "Successor", "Action", "Guide"
"0", "|", "Path' = . Path #", "", "SHIFT Path 1", "an"
"0", "", "Path = . Dirs Name", "#", "SHIFT Dirs 2", "an"
"0", "", "Dirs = . Dir", "an", "SHIFT Dir 3", "an"
"0", "", "Dir = . Name ""/""", "an", "SHIFT Name 4", "an"
"0", "", "Name = . an", """/"",  an", "SHIFT an 5", "an"
"0", "", "Name = . Name an", """/"",  an", "SHIFT Name 4", "an"
"0", "", "Dirs = . Dirs Dir", "an", "SHIFT Dirs 2", "an"
```

Figure 38: Partial parser table from Section 4.3 in CSV format.

easy implementation of CSV operations, and handles character escaping and different separators in the background as well. The usage of the library is straight forward, as the table has to consist of a list that holds a `String[]`.

The array represents the columns, and each entry in the list is a row. In case of the parser table and the simulation steps, the amount of columns is known at compilation of the tool, as for these the column are a fixed amount. For the state-transition-table the amount of columns is already calculated in Section 4.4 which can be reused here. With this knowledge, it is as simple as iterating the states and placing the desired values in the correct spot of the `Array`.

4.9.2 Export Simulation Error Log

The log files for the simulation consist of messages such as "*Removed '!' from input at position 1*" and "*No symbol was inserted into input*". The messages are generated during the simulation of each individual input. The strings for the messages are concatenated with the help a `StringBuilder` and then saved into a plain text file at the end of a simulation run with the use of a basic `FileWriter` from Java.

5 Usage

The usage of the developed tool is simple and uses a Command-line interface (CLI) for controls. The tool provides several options to customize its behavior, which can be set using the CLI arguments. The simplest way to run the program is to just pass the input file as an argument via `-i` or `--input`. However, there are additional options available to tailor the functionality of the program according to the specific needs of the user. The following list shows all available options that can be used with the tool.

- `-i, --input <file>`: Specifies the input file containing the grammar, `TS` and simulation sentences. This option is required, and expected to be correct
- `-o, --output <dir>`: Sets the output directory for the generated CSV files. By default, the tool uses the same directory as the input file is in.
- `-a, --actionformat <str>`: Determines the format of the action symbols in the generated parsing table. The supported formats include `UPPER_CASE`, `LOWER_CASE` and `SHORT`. By default, the format `UPPER_CASE` is used.
- `-j, --java <path>`: Specifies the Java runtime used for running Coco/R. By default, the tool uses the `"java"` environment variable to locate the Java runtime. However, with this argument, the user can provide the path to a specific Java runtime if necessary.

An example is the command shown in Figure 39, which takes a local `grammar.txt` as the input, sets the output directory to `/tmp`, and uses the `SHORT` format to display only the first letter of the action.

```
java -jar lalr1_table_generator.jar
  --input ./grammar.txt
  --output /tmp/
  --actionformat SHORT
```

Figure 39: Example command for running `lalr_table_generator.jar` with options.

After executing the command, the tool performs all the necessary steps to generate a LALR(1) parser table, state-transition-table. The tool also performs the simulation and produces the simulation steps and a simulation log. In addition to the output files, the program also prints some information about the current state. So, all the steps which are explained in Section 4 are performed.

6 Related Work

The field of related work is not that large, when it comes to the requirements demanded for the tool described in this paper. Nevertheless, there are some papers worth mentioning.

The paper "*Implementation of an LALR(1) Parser Generator*" by G. Svenheim [6] is more than 40 years old and accomplishes almost everything that is required for today's tool. One of the few parts limiting its use is the fact that the input grammar used, and the algorithm used, are slightly different from those used in the course. However, the main obstacle is the used language. G. Svenheim's implementation is based on Programming Language One (`PL/1`), an IBM language first published in 1964 and is therefore impractical to use today.

Another work that can still be used today is a web-based "*LALR(1) Parser Generator* by G. Apou [1]. The tool offers similar input, output and simulation as the tool described in this paper. The only disadvantage is that the formatting is a bit different and would not work as well for use as a sample solution. Also, the simulation part does not perform error handling at all and sometimes locks up the browser

In conclusion, the field of related work to the requirements described in this paper may not be extensive, but there are notable works worth mentioning. While these works contribute valuable insights and functionalities, the tool presented in this paper aims to provide the specific implementation for the requirements given by the lecturers of the Compiler Construction course.

7 Lessons Learned

During the process of developing the LALR(1) Parser Table Generator, several valuable lessons were learned that significantly contributed to the successful implementation of the tool, or will contribute in future projects. This section summarizes the key takeaways and insights gained from the project and are listed below:

- *Thorough Understanding of the Parsing Algorithm*: The development of the tool was based on existing, but fundamental, knowledge of the LALR(1) algorithm. This prior knowledge served as a crucial foundation for the project and guided the implementation process. Nevertheless, the knowledge deepened as many things came up during development that were not on the radar before. Much of this knowledge was gained during the development process itself, creating issues of understanding that led to the task of rewriting. In subsequent endeavors, our approach will involve a proper understanding of an algorithm and process prior to the development of a tool.

- *Test-Driven Development (TDD)*: The use of TDD proved to be a successful instrument in the development of the tool. TDD helps to ensure that the functionality of the tool is thoroughly validated throughout the development process, with tests implemented beforehand. It helped to keep the focus on the desired outcomes and encourages incremental development. However, we learned the importance of validating the accuracy of the test cases, as faulty or incomplete tests could mislead our understanding of the tool's behavior and lead to incorrect implementations. This realization led us to re-evaluate and refine the tests later in development to ensure proper validation of the tool.

- *Approaching a Bigger Project*: Taking on a larger project has taught us the importance of a systematic approach with some planning up front. Whether it was clearly defined goals such as the topic description, or clear milestones with set due dates. Regular meetings with our supervisor also played an important role, providing valuable feedback and guidance for the development process. This experience highlighted the significance of planning and efficient communication between individuals, to achieve a successful outcome when tackling larger projects.

Overall, the work on the tool and this paper has provided valuable insights and reflection on the challenges and successes encountered. By emphasizing the importance of a thorough understanding of the algorithms, proper planning and effective communication, this experience has highlighted the importance of a structured approach successfully to tackle larger projects. By applying the lessons learned, we are confident in our ability to address future challenges with a proactive mindset to achieve a positive outcome.

8 Limitations and Future Work

Finally, it is important to acknowledge some limitations and shortcomings of the tool developed in this paper. Although the tool is able to perform the tasks required, it falls short in some important areas. These limitations, along with other extensions and a final conclusion, are elaborated in this section.

8.1 Limitations

First, the tool lacks certain features that are considered standard. One such limitation is the lack of support for the EBNF grammar as an input. The absence of EBNF in the tool limits compatibility to normal BNF-based grammar. This means that the handed out EBNF grammar must first be manually converted into a valid BNF.

Furthermore, the tool also does not perform any input validation. The lack of such validation makes the tool fragile if a non-conforming input is passed. In particular, validation of the input BNF grammar, or the lack thereof, can cause unintended side effects such as incorrect results, freezes or crashes. Besides missing validation, the tool also expects that the input grammar is already correctly ordered. This means that the user has to make sure that the grammar is free of syntax errors and that the grammar follows an appropriate hierarchical structure. Such a limitation places the burden on the user to ensure the correctness of the grammar before using the tool.

Moreover, the tool has not undergone extensive testing to validate its correctness and robustness. Although test-driven development principles were used during the development of the tool, the test coverage remains very limited. The tests performed during the development phase focused mainly on individual components and their functionality. The lack of more comprehensive testing in the later stages of development may result in undetected problems and limitations that could affect the stability and overall reliability of the tool.

8.2 Future Work

There are several areas of improvement and future work that can be based on the tool described in this paper. Improvements include, but are not limited to, increasing functionality, reliability and overall effectiveness. This also includes addressing the limitations mentioned in Section 8.1.

One important aspect to address is the tool's heavy reliance on correct input. In addition to proper input validation, the tool could be enabled to perform proper sorting and ordering of the input grammar itself. This would improve the usability of the tool and save the user the effort of manually arranging the grammar in the desired order.

Another notable improvement would be to extend the input grammar to support EBNF over the current BNF. This would eliminate the need for manual conversion to BNF and improve usability. This change would indirectly reduce the risk of subsequent errors due to incorrect conversions. Supporting EBNF would also make the tool less cumbersome to use for more complex grammars.

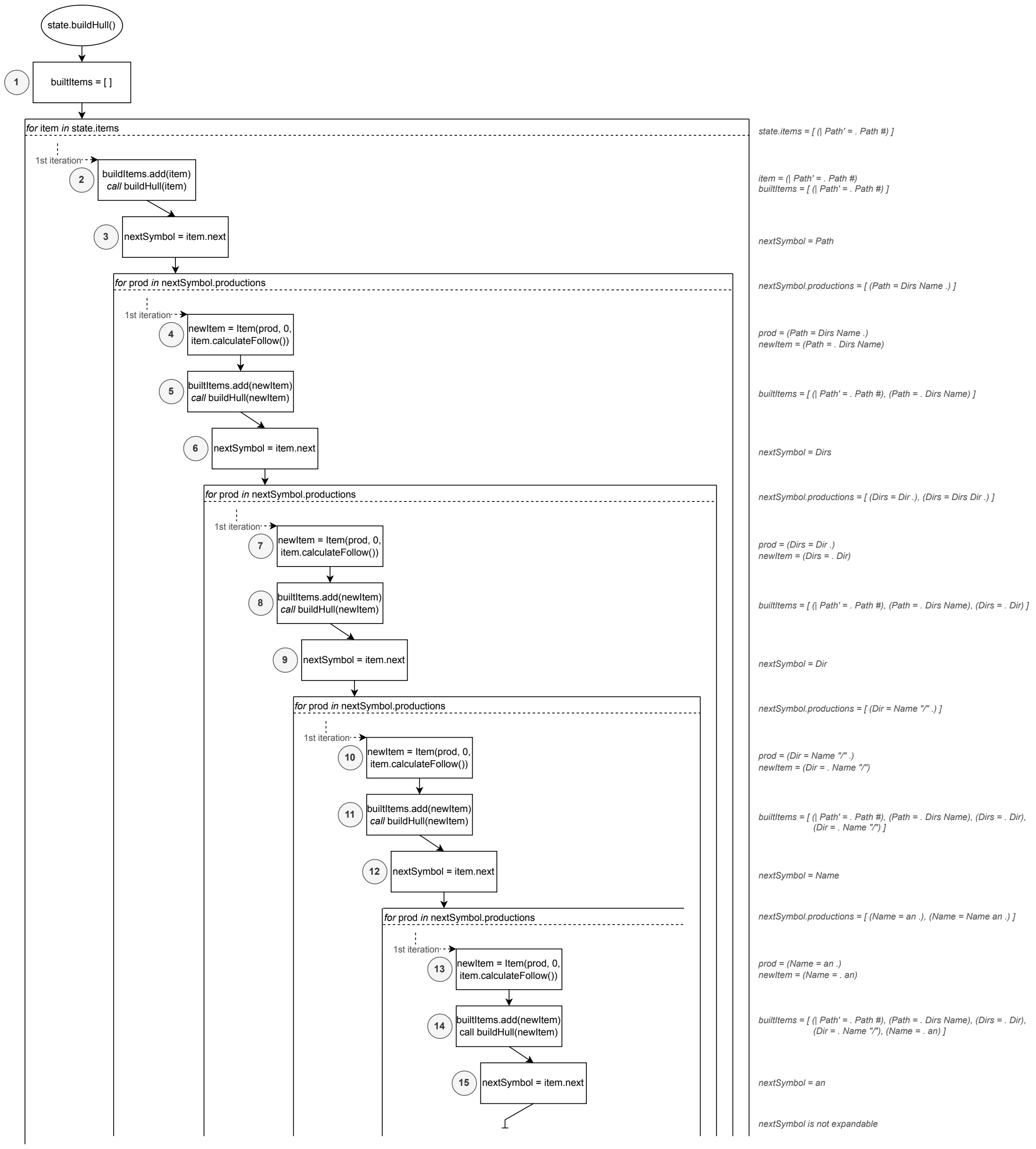
9 Conclusion

In this paper, we presented our *LALR(1) Parser Table Generator* tool, that allows the generation of a parser table, a state-transition-table and the simulation of sentences on a given grammar. We have discussed the general structure of the tool, followed by the implementation. In the implementation, for the generation of a parser table and a state-transition-table are elaborated. The use of Coco/R to parse simulation sentences is also discussed. Finally, the simulation of sentences and the error handling with the algorithm learned in the *Compilerbau* course are explained.

While our tool is not without limitations, we consider it a solid foundation upon which future improvements can be built. The tool is implemented as a command-line application in Java. We believe that this tool can save time for lecturers, allowing them to allocate their time and effort elsewhere. Furthermore, we anticipate that the sample solution provided will serve as a valuable resource for future students seeking to grasp the concepts of bottom-up parsing.

In conclusion, the developed tool provides a simple solution for generating the required tables from a given grammar. While further refinements and enhancements can be made, we are confident that the tool serves as a valuable contribution to provide a helpful resource for both the lecturers at Institute for System Software (SSW) and the students at JKU.

Appendix



state.buildHull()
1
builtItems = [ ]
for item in state.items
1st iteration
2
buildItems.add(item)
call buildHull(item)
3
nextSymbol = item.next
for prod in nextSymbol.productions
1st iteration
4
newItem = Item(prod, 0, item.calculateFollow())
5
builtItems.add(newItem)
call buildHull(newItem)
6
nextSymbol = item.next
for prod in nextSymbol.productions
1st iteration
7
newItem = Item(prod, 0, item.calculateFollow())
8
builtItems.add(newItem)
call buildHull(newItem)
9
nextSymbol = item.next
for prod in nextSymbol.productions
1st iteration
10
newItem = Item(prod, 0, item.calculateFollow())
11
builtItems.add(newItem)
call buildHull(newItem)
12
nextSymbol = item.next
for prod in nextSymbol.productions
1st iteration
13
newItem = Item(prod, 0, item.calculateFollow())
14
builtItems.add(newItem)
call buildHull(newItem)
15
nextSymbol = item.next
state.items = [(| Path' = . Path #)]
item = (| Path' = . Path #)
builtItems = [(| Path' = . Path #)]
nextSymbol = Path
nextSymbol.productions = [(Path = Dirs Name .)]
prod = (Path = Dirs Name .)
newItem = (Path = . Dirs Name)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name)]
nextSymbol = Dirs
nextSymbol.productions = [(Dirs = Dir .), (Dirs = Dirs Dir .)]
prod = (Dirs = Dir .)
newItem = (Dirs = . Dir)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir)]
nextSymbol = Dir
nextSymbol.productions = [(Dir = Name "/" .)]
prod = (Dir = Name "/" .)
newItem = (Dir = . Name "/")
builtItems = [(| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir), (Dir = . Name "/")]
nextSymbol = Name
nextSymbol.productions = [(Name = an .), (Name = Name an .)]
prod = (Name = an .)
newItem = (Name = . an)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir), (Dir = . Name "/"), (Name = . an)]
nextSymbol = an
nextSymbol is not expandable

Figure 40: Part one of omplete flow for `buildHull` for state zero.

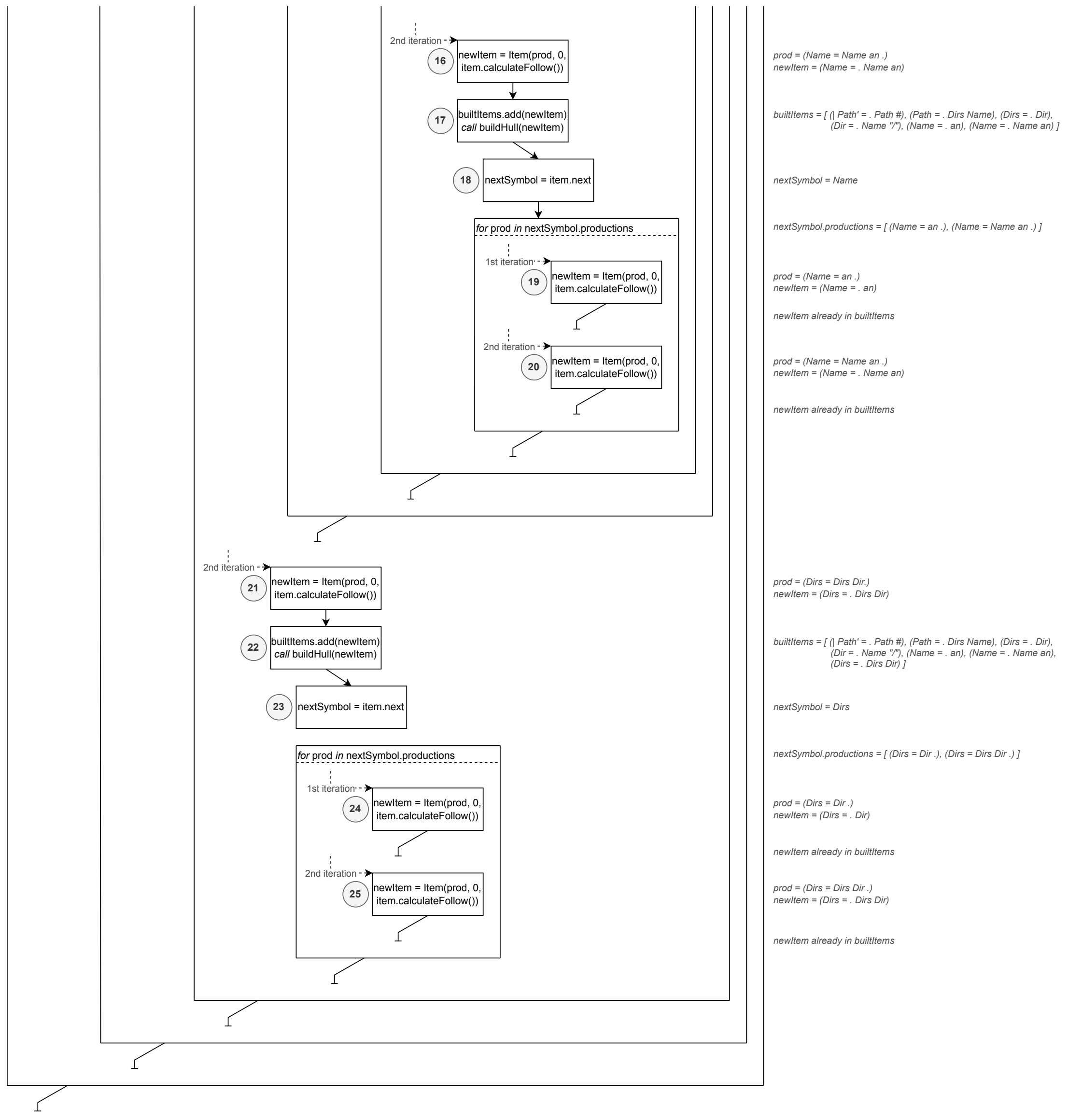

2nd iteration
16
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = Name an .)
newItem = (Name = . Name an)
17
builtItems.add(newItem) *call* buildHull(newItem)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir), (Dir = . Name "/"), (Name = . an), (Name = . Name an)]
18
nextSymbol = item.next
nextSymbol = Name
for prod *in* nextSymbol.productions
nextSymbol.productions = [(Name = an .), (Name = Name an .)]
1st iteration
19
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = an .)
newItem = (Name = . an)
newItem already in builtItems
2nd iteration
20
newItem = Item(prod, 0, item.calculateFollow())
prod = (Name = Name an .)
newItem = (Name = . Name an)
newItem already in builtItems
2nd iteration
21
newItem = Item(prod, 0, item.calculateFollow())
prod = (Dirs = Dirs Dir.)
newItem = (Dirs = . Dirs Dir)
22
builtItems.add(newItem) *call* buildHull(newItem)
builtItems = [(| Path' = . Path #), (Path = . Dirs Name), (Dirs = . Dir), (Dir = . Name "/"), (Name = . an), (Name = . Name an), (Dirs = . Dirs Dir)]
23
nextSymbol = item.next
nextSymbol = Dirs
for prod *in* nextSymbol.productions
nextSymbol.productions = [(Dirs = Dir .), (Dirs = Dirs Dir .)]
1st iteration
24
newItem = Item(prod, 0, item.calculateFollow())
prod = (Dirs = Dir .)
newItem = (Dirs = . Dir)
newItem already in builtItems
2nd iteration
25
newItem = Item(prod, 0, item.calculateFollow())
prod = (Dirs = Dirs Dir .)
newItem = (Dirs = . Dirs Dir)
newItem already in builtItems

Figure 41: Part two of complete flow for `buildHull` for state zero.

```
"Nr", "Core", "Item", "Successor", "Action", "Guide"
"0", "|", "Path' = . Path #", "", "SHIFT Path 1", "an"
"0", "", "Path = . Dirs Name", "#", "SHIFT Dirs 2", "an"
"0", "", "Dirs = . Dir", "an", "SHIFT Dir 3", "an"
"0", "", "Dir = . Name ""/""", "an", "SHIFT Name 4", "an"
"0", "", "Name = . an", """/"",  an", "SHIFT an 5", "an"
"0", "", "Name = . Name an", """/"",  an", "SHIFT Name 4", "an"
"0", "", "Dirs = . Dirs Dir", "an", "SHIFT Dirs 2", "an"
"1", "|", "Path' = Path . #", "", "ACCEPT #", "#"
"2", "|", "Path = Dirs . Name", "#", "SHIFT Name 6", "an"
"2", "", "Name = . an", "#,  an,  ""/""", "SHIFT an 5", "an"
"2", "", "Name = . Name an", "#,  an,  ""/""", "SHIFT Name 6", "an"
"2", "|", "Dirs = Dirs . Dir", "an", "SHIFT Dir 7", "an"
"2", "", "Dir = . Name ""/""", "an", "SHIFT Name 6", "an"
"3", "|", "Dirs = Dir .", "an", "REDUCE an (2)", "an"
"4", "|", "Dir = Name . ""/""", "an", "SHIFT ""/"" 8", """/"""
"4", "|", "Name = Name . an", """/"",  an", "SHIFT an 9", """/"""
"5", "|", "Name = an .", """/"",  an,  #", "REDUCE ""/"", an, # (5)", "#"
"6", "|", "Path = Dirs Name .", "#", "REDUCE # (1)", "#"
"6", "|", "Name = Name . an", "#,  an,  ""/""", "SHIFT an 9", "#"
"6", "|", "Dir = Name . ""/""", "an", "SHIFT ""/"" 8", "#"
"7", "|", "Dirs = Dirs Dir .", "an", "REDUCE an (3)", "an"
"8", "|", "Dir = Name ""/"" .", "an", "REDUCE an (4)", "an"
"9", "|", "Name = Name an .", """/"",  an,  #", "REDUCE ""/"", an, # (6)", "#"
```

Figure 42: Complete parser table from Section 4.3 in CSV format.

List of Tables

List of Figures

References

[1] Gregory Apou. JSMachines, 2011.

[2] D. Grune and C. J. H. Jacobs. *Parsing Techniques - A Practical Guide.* Monographs in Computer Science. Springer, 2008.

[3] D. D. McCracken and E. D. Reilly. *Backus-Naur Form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR, 2003.

[4] H. Mössenböck. A generator for production quality compilers. In D. K. Hammer, editor, *Compiler Compilers, Third International Workshop on Compiler Construction, CC'90, Schwerin, Germany, October 22-26, 1990, Proceedings*, volume 477 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1990. `https://ssw.jku.at/Research/Projects/Coco/`.

[5] OpenCSV Development Team. A simple library for reading and writing CSV in Java, 2005.

[6] G. Svenheim. Implementation of an LALR(1) parser generator, March 1980. Note: imported from RIT's Digital Media Library running on DSpace to RIT Scholar Works. Physical copy available through RIT's The Wallace Library at: QA76.6.S913.

Statutory Declaration