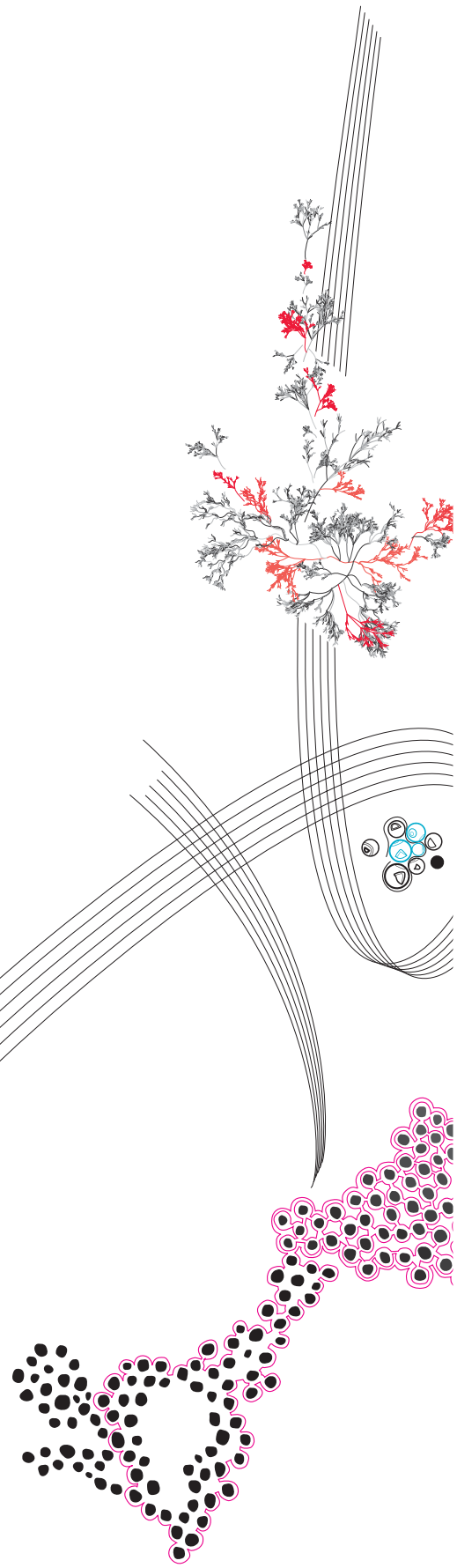


MSc Computer Science
Thesis

WAGon — A Weighted Attribute Grammar Oriented Notation

Rafael M. Dulfer

Supervisor: Vadim Zaytsev

June, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Weighted Attribute Grammars (WAGs) are an emerging field of study and have been used for a variety of different tasks. However, due to the as-of-now obscure nature of the field, researchers are left to re-invent the wheel every time, having to implement a WAG parser before they can start the interesting part of their research. This paper proposes the creation of a standardized Domain Specific Language (DSL) to define WAGs, as well as a workbench to facilitate working with this DSL and a parser generator that functions as a business case. **Keywords**: weighted attribute grammars, attribute grammars, weighted grammars, parsing, parser generator, GLL parsing, Context-Sensitive languages

Chapter 1

Introduction

> "Let's start with what we have come into the room to do."
>
> *Fela Ransome Kuti*

Weighted Attribute Grammars (WAGs) are a currently relatively young field of study which attempts to combine the powers of weighted grammars with those of attribute grammars. Primarily, they have been used for the generation of data, such as dungeons [3] or smart assistant responses [43], but they could conceivably also have applications in other fields such as context-aware parsing and machine learning [11].

Because WAGs are a relatively young area of study, researchers who would like to use them for their applications are required to write a parser and design a DSL by hand, which costs valuable time. Currently, when a computer scientist wants to use conventional grammars for any purpose, they have access to a multitude of different parser generators which purport various feature sets. Rarely will they have to design their own handwritten parsers. However, none of the current parser generators have support for Weighted Attribute Grammars.

In this thesis, we present WAGon. A dedicated DSL and ecosystem for defining and working with weighted attribute grammars. Additionally, in order to show the capabilities of WAGon and Weighted Attribute Grammars in general, we present a Generalized LL (GLL)-based parser generator which uses the DSL and ecosystem as a base. Showing how future researchers will be able to use WAGon as a starting point, preventing them from being required to reinvent the wheel.

The thesis is structured as follows: Chapter 2 gives some historical and mathematical background of weighted attribute grammars in general. Chapter 3 discusses the design approach taken towards the DSL and ecosystem. Chapter 4 explains the DSL in detail. Chapter 5 will focus on the ecosystem, while Chapter 6 will go over the creation of the GLL-based parser generator and an example case study. Finally, Chapter 7 concludes by providing some possible future improvements to WAGon as well as potential research topics that WAGon could be employed for.

Chapter 2

Weighted Attribute Grammars

"Language is a virus from outer space."

William S. Burroughs

A Weighted Attribute Grammar (WAG) combines the powers of Attribute Grammars and Weighted/Probabilistic Grammars. [43] As such, these must first be discussed to understand WAGs and to understand those, one must first understand grammars.

2.1 Grammars

Grammars, in the context of formal language theory, are a much studied field first popularized in the 50s by Noam Chomsky [8], but with roots as far back as Pāṇini in the 4th century BCE [34]. They are used everywhere in fields as diverse as computer science [20] [1], linguistics [34], biology [29] and many more [25].

A grammar G is formally defined as a tuple (N, T, P, S) where:

- N is a finite set of *nonterminal* symbols.
- T is a finite set of *terminal* symbols. $T \cap N = \emptyset$
- P is a finite set of *production rules*.
- $S \in N$ is the nonterminal start symbol.

For example, the following grammar defines a language which consists of some number $n >= 1$ occurrences of the terminal symbol 'a', followed by n occurrences of the terminal symbol 'b':

$\langle S \rangle \rightarrow \langle aSb \rangle$

$\langle S \rangle \rightarrow \langle ab \rangle$

Each nonterminal can form the left-hand side of multiple production rules. In this paper, we will refer to individual production rules of a given nonterminal X as "an *alternative* of the rule X". Each alternative is delineated with a | symbol. So the above grammar can be rewritten as:

$\langle S \rangle \rightarrow \langle aSb \rangle \mid \langle ab \rangle$

Type	Name	Example
Type-3	Regular	$L = \{a^n \mid n > 0\}$ I.E. *aaaa...*
Type-2	Context-Free	$L = \{a^n b^n \mid n > 0\}$ I.E. *aabb*
Type-1	Context-Sensitive	$L = \{a^n b^n c^n \mid n > 0\}$ I.E. *aabbcc*
Type-0	Recursively Enumerable	$L = \{w \mid w$ describes a terminating Turing machine$\}$

TABLE 2.1: Chomsky Hierarchy

Generally, grammars are usually described as "producing" text (in this case, the string of a's and b's), but are in practice often used for analyzing text instead (in this case, checking whether the given input consists out of exactly some number n a's and then some number n b's and **nothing else**). In reality, they are capable of both (and more), depending on how the grammar is interpreted and used.

2.1.1 Expressive Power

Not every grammar is equally "powerful". Computational linguists commonly categorize grammars into a type in the Chomsky hierarchy [9]. Which type the grammar belongs to tells us what class of languages can be expressed using that type of grammar. A short summary of this hierarchy can be found in Table 2.1. Each higher class of languages is able to express everything the lower classes can and more (so a higher class is a superset of the lower class).

In the domain of computer science, we usually only concern ourselves with Regular and Context-Free Grammars (CFGs). This is because they are easy and fast to analyze algorithmically. As such, most tools developed by computer scientists only support problems expressible in Type-3 and Type-2 grammars. Weighted Attribute Grammars (as we will discuss in Section 2.4.3) are actually part of the Context-Sensitive class of grammars, making them more powerful than conventional grammars. As such, WAGs (and tooling that supports them) are able to express everything that context-free grammars can and more. Allowing them to express and solve a completely new class of problems.

2.2 Attribute Grammars

Attribute grammars were first fully described by Knuth [21] in 1968. He introduces the concept of "inherited" and "synthesized" attributes; where the former are evaluated from the top-down, while the latter are evaluated from the bottom-up.

An attribute grammar is defined as a variant of a CFG $G = (N, T, P, S)$ such as described in Section 2.1. Additionally, we define the set $V = N \cup T$. For each symbol $X \in V$ we associate a finite set $A(X)$ of attributes, which is partitioned into the disjoint sets $A_0(X)$ for synthesized attributes and $A_1(X)$ for inherited attributes. We define $A_1(S)$ to be empty, as well as $A_0(X)$ if $X \in T$.

Attribute grammars are already pretty powerful (in Knuth's paper, he uses them to evaluate a binary string at parsetime) and have been researched extensively. Of the major parser generators, Happy directly supports AGs [24], while ANTLR and Bison support them in the form of "actions" [30] [12][1]. Additionally, AG-first parser generators like Silver [41] and JastAdd [16] allow language designers to make full use of the power of attribute grammars. AGs have been used for a variety of purposes, such as equation

[1] ANTLR and Bison often use "actions" only in the context of attribute grammars, but they are treated differently in their respective documentations, with attribute grammars being less important.

discovery [5], movement sequencing in robots [22] and generating tests [15] and will continue to be a popular research topic for some time to come.

Because our ecosystem will support Weighted Attribute Grammars and WAGs are a superset of attribute grammars (see the discussion in Section 2.1.1) our WAGon ecosystem is automatically also an ecosystem for attribute grammars.

2.3 Weighted/Probabilistic Grammars

As opposed to attribute grammars, weighted grammars have received less attention. They were originally described by Chomsky and Schützenberger in 1963 [9] and primarily find use in the fields of natural language processing [27] and bioinformatics [13]. A weighted grammar is rather simply a CFG G in which each rule $r \in P$ has an associated weight. This weight is then often used to handle ambiguity of G [28]. A probabilistic grammar is a special case of a weighted grammar which denotes the probability that the rule will be chosen [35]. A probabilistic grammar can easily be identified in that the weights of the alternatives of a rule all sum up to 1. Probabilistic grammars traditionally see use in both natural language processing and text generation as they can help disambiguate the often ambiguous nature of natural languages and provide a schematic for structured randomized text generation.

A major difficulty for weighted grammars is how to define the weights. Frequently, this is done through the creation of a corpus, from which probabilities are calculated which are then manually fine-tuned [33]. Modern developments use neural networks to define these weights [42]. WAGs can help alleviate this problem by having the fine-tuning happen during parsing. For example, lowering the probability of an alternative every time it is taken (and subsequently increasing the probability of the other alternatives).

As of writing, tool support for weighted grammars is not nearly as elaborate as that of attribute grammars. While they could be replicated using "actions" in ANTLR and Bison, this is only possible in specific cases and requires manually checking a value, failing the parse if required and manually redirecting to the required rule (using, for example, the `YYBackup` macro in Bison) [12] [30]. Alternatively, there are at least two tools out there that have dedicated weighted grammar support. The first being Allauzen et al's `GRM Library` [2] which was licensed by AT&T and now seems to be lost media. The second being ModelCC by Quesada et al., which supports probabilities, but is explicitly not grammar based [32] as it uses object-oriented style inheritance diagrams instead.

Because WAGs are a superset of weighted grammars, our WAGon ecosystem is likely one of, if not the first grammar-based parser workbench/generator with dedicated weighted parsing capability.

2.4 Weighted Attribute Grammars: What and Why?

A Weighted Attribute Grammar, as said previously, is then a combination of weighted grammars and attribute grammars. The way these two approaches compliment each other is quite obvious. One can take the attributes derived by the attribute grammar and use them to define probabilities or weights to be used in the further parsing of the file. For example, you could conceivably use a WAG to, for example, analyze whether a file is in `python2` or `python3` dialect and the moment you are sure which one it is, keep state and pivot to whichever dedicated grammar you need. Or you could keep track of sentiment earlier in a text and use this to change the probability that a later statement is sarcastic or not.

A WAG in which every weight is equal is just an AG, whereas a WAG in which no attribute evaluation occurs and only constants occur is just a WG.

2.4.1 Formal Definition

We define a WAG as follows (inspired by the work of Zaytsev [43]):

Definition 1. A weighted attribute grammar is a tuple $\mathscr{G} = \langle \Gamma, \Omega, \Delta, \Phi \rangle$, where

- $\Gamma = \langle N, T, P, s \rangle$ is a Chomskian grammar
- $\Omega = \langle \omega, \beta \rangle$ is a tuple of type-value pairs:
  - $\omega : P \to \mathbb{T}$ assigns types to weights.
  - β assigns weights to each production rule.
- $\Delta = \langle A, \tau, \kappa, \pi \rangle$ is a tuple of attribute components:
  - A is the set of attributes
  - $\tau : A \to \mathbb{T}$ assigns types to attributes (as with weights, these types are inferred at initialization).
  - $\kappa : N \to A^*$ specifies inherited attributes for each nonterminal.
  - $\pi : N \to A^*$ specifies synthesized attributes for each nonterminal.
- Φ are computation formulae to define the attributes.

2.4.2 Informal Description

We take a conventional CFG and add to it both the concepts of attributes (as defined in Section 2.2) and weights (as defined in Section 2.3). We will denote **inherited** attributes with a `*` and **synthesized** attributes with a `&`. Every nonterminal rule $S \in N$ has a number of attributes that it requires to use. In this paper, we define these attributes for each rule in a comma separated list surrounded by `<>` (I.E. `S<*a, &b> -> ...`).

Each alternative of a rule has a weight associated with it. Exactly what this weight means is undefined and left to the language designer. The weights are based on the attributes and could dictate many things, for example: "only take alternatives with the highest weight", "take alternatives randomly based on weight", "just annotate the tree with the weights" etc.

Inside of each alternative, the attributes can be modified. This is done through a series of computational formulae. Additionally, when new nonterminals are encountered, they are given the attributes they require (again in a comma separated list surrounded by `<>`). For a more comprehensive overview over how these features are expressed in the WAGon DSL and why these specific symbols were chosen, see Chapter 4.

2.4.3 Abilities

Because WAGs have not yet been studied much, the true extent of their abilities is still up for speculation. The most direct hint for their true power lies in the fact that they are fundamentally context sensitive. By combining attributes and weights, the history of the parse influences the current states. Therefore, WAGs are a whole Chomsky hierarchy level more expressive than conventional Context-Free Grammars. For example, while a CFG is completely incapable of expressing the language $L = \{a^n b^n c^n | n > 0\}$, it can be relatively easily described in a WAG:

```
S      -> {*n = 0} A<*n> B<*n> C<*n>;
A<&n>  -> {&n = &n + 1} "a" A<&n> | ϵ;
B<*n>  -> [*n > 0] {*n = *n - 1} "b" B<*n>
       |  [*n == 0] ϵ
       ;
C<*n>  -> [*n > 0] {*n = *n - 1} "c" C<*n>
       |  [*n == 0] ϵ
       ;
```

Many natural languages [37], as well as many programming languages and other complex systems are at least context-sensitive (if not recursively enumerable). WAGs allow us to describe at least the context-sensitive structures in a formal way, allowing us new insights into how they work, as well as a blueprint to automatically generate tooling for these structures (like, for example, parsers).

2.4.4 Purpose

In truth, there is nothing a WAG can do that can not be done through other means. A sufficiently complicated series of nested `if ...  else ...` statements could perform the analytical and generative tasks of a WAG in much the same way and modern parser generators can be convinced to do cases and attribute evaluation. As such, the power of WAGs lies not in its abilities but in its possibility to have a pure[2] and relatively concise notation. We introduce WAGs for the same reason Knuth introduced inherited attributes [21]. Not because they are more powerful, but because they are hopefully less complicated to write, understand and extend than the alternative.

[2]Meaning, we stay in the pure declarative world of grammars

Chapter 3

Design

> "Confine your attention to designs that look good because they are good."
>
> *Bruce MacLennan* [23]

As discussed in Section 2.4.4, WAGs are only useful insofar as that they are easier to use for their purpose than the alternatives. As such, it is incredibly important that the language used to write them is easy to use and read. In fact, we believe the design of the language to be the most vitally important part of this whole project. If WAGon turns out to be difficult or annoying to use, there would be no point to its existence.

A full grammar of the WAGon DSL can be found in Listing 4.2 and an explanation of the DSL can be found in Chapter 4. What follows here is a discussion on the design philosophy.

3.1 The Language

The design philosophy is so important to WAGon that we will state it again. If it is not easy to use, it is worthless. The issue, however, is that "easy to use" is an incredibly nebulous concept [39]. What makes a language easy to use? What makes a tool easy to use? How does one design a good language? While it is possible to do in-depth A/B testing and empirical research (such as was done for the Quorum programming language [38]), this requires a significant time investment. Since we are also developing a library ecosystem and example case study it was decided that a full study was not feasible. Still, lessons can be learned from the studies that have already been done. Additionally, processes exist which, while less elaborate, require significantly less time.

3.1.1 PLIERS

PLIERS (Programming Language Iterative Evaluation and Refinement System) is a process for programming language design developed by Coblenz et al. [10]. While the process was designed for iterative refinement of the language (something which, as stated above, we did not have time for), the first 3 steps did help inform our initial design.

The first step of PLIERS is "Need Finding". We cheat a little bit here by assuming the need for a standardized WAG DSL as a given. Additionally, the existing literature on WAGs was analyzed to see what "conventions" (as little as there were) existed. We've

incorporated elements from the languages used in those papers if they fit our design goals (for example, weights being inside `[]` which was inspired by a paper by Zaytsev [43]).

The second step of PLIERS; "Design Conception" iterates between two kinds of work:

1. Develop a theoretical foundation
2. Prototype

As a DSL specifically designed for WAGs, an already existing research field, a lot of the theoretical foundation was already laid for us (as described in Chapter 2). Therefore, we could focus on the prototyping aspects. During the prototyping stage a number of simple programs and ideas were sketched out in simple file editors. This was supplemented by a few informal rounds of natural programming [10] in which some computer scientists with knowledge of grammars were approached and asked to write a grammar on a notepad. Then, concepts of WAGs were introduced (i.e. weights and attribute assignments) and the subjects were asked to simply write it down in a way that seemed natural to them. The ideas they came up with were partly absorbed into the design.

Finally, the third step of PLIERS is "Risk Analysis". Part of this step was already done through the natural programming from the previous step, but the authors also suggest analyzing the language through the Cognitive Dimensions of Notations framework [4]. A framework which, while it has received criticism in recent years [39], provides us with a common language to talk about (programming language) design, as well as pointers for what to look for in a language. We determined the following dimensions to be most important for WAGon:

- High closeness of mapping. The notation should closely match the formal definitions of grammars.
- High consistency. Symbols should mean the same thing in the same context.
- Low diffuseness. The language should not be overly verbose.

3.1.2 Principles

Having gone through the first 3 steps of the PLIERS process and establishing the important cognitive dimensions for the language, we laid out the following design principles:

- Leverage the vocabulary of popular languages.
  - This is a double-edged sword, as it may happen that users believe they have access to certain "abilities" from the language that the vocabulary was lifted from even though they do not. It does however mean that users are more likely to intuitively understand how something is intended to work.
- Do not use too many arcane symbols.
  - There should be a purpose behind why a symbol was chosen that can be easily explained and intuitively assessed.
- Do not be too verbose.
  - Users should be able to read a grammar without having to skim too much uninteresting boilerplate.
  - This conflicts with the above point and the two are constantly at odds.

- Stick to grammars
  - The beauty of WAGs is that they incorporate concepts from the "dirty" world of computer programming into the "clean" world of formal grammars. As discussed before, many existing parser generators can do what WAGs do, but they require the language designer to step into the impure outside world. As such, we should aim to use as little "programmy" aspects as possible and to stick to the declarative nature of grammars.

These principles are still highly nebulous, as is the sad truth of any design work, but they provide some sort of structure above simply blindly trying things.

As an example of how this process and these principles affected the design, we will use the case of what delimiters to use when passing attributes to non-terminals:
Originally, we designed the DSL to use parentheses (`()`) for this purpose. However, this conflicted with the use of parentheses for EBNF operators (e.g. `(A B(*a))+`). Because consistency is valued, it was decided not to allow the grammar to become ambiguous and to search for a new delimiter. A number of other delimiters were considered (such as `[]`, `/\` and even `::`) and presented to a few third-party computer scientists before we finally settled on `<>`. This particular set of characters was chosen because it encloses the input, just like parentheses, is not in use in this way anywhere else in the DSL (they are used to make arrows but they are clearly differentiable) and because there is precedent for using them in a similar manner to define generic types in languages like Rust and TypeScript. In these cases, the value inside the angle bracket defines what exact type we are invoking the generic struct on. We can also see the nonterminals this way. Defining what exact attributes we are invoking the nonterminal with, or letting it be inferred (as described in Section 4.3.1).

We go more in-depth on the language itself and some of the specific design decisions in Chapter 4.

3.2 The Ecosystem

WAGon is a DSL, but for a DSL to be useful, tooling around it should also exist. Because WAGs have many different possible applications, it is very difficult to write a single tool that can do everything, but it **is** doable to create an ecosystem of libraries that facilitates creating whatever tool is needed. The ecosystem was created in a modular fashion (for example, the lexer can be used completely separately from the parser) such that a user can step in at whatever level is necessary for them to realize their project. It also contains various useful modules that handle generic tasks when working with WAGs (for example, generating code that evaluates expressions). A deeper dive into the ecosystem can be found in Chapter 5.

Chapter 4

The WAGon DSL

> "Gonna paint our wagon, gonna paint it good."
>
> *Clint Eastwood*

Let us start with an example grammar written in the WAGon DSL:

```
type: analytical;
=========================================

S -> A | B;
A -> {$did_a = true;} E<$did_a> B | ;
B -> {$did_a = false;} E<$did_a> A | ;
E<*did_a> -> [*did_a * 2 + 1] C
| [(!*did_a) * 3 - 4] D
;
C -> [0.3] F
| [0.7] G
;
D -> [0.7] F
| [0.3] G
;
F -> "1";
G -> "1";
```

We will explain the constituent parts of the grammar step-by-step.

4.1 Metadata

```
type: analytical;
=========================================
```

Every grammar file can optionally start with a metadata section. Inside of the section, `key: value` pairs may be written (the use of which lies with the tool designer). Additionally, includes can be written inside of the metadata section to specify that external files should be included in this grammar.

This section was added to allow for easy configuration options. Future users of the WAGon ecosystem may want to allow various runtime arguments to change the behavior of their tools (for example, the parser generator uses it to determine whether maximal or minimal weights should be used for disambiguation). It is impossible for us to predict all possible configuration requirements, so we simply allows developers to define them themselves. The `value` of each pair can be any basic literal (see Section 4.3.1).

The metadata section is delineated with 3 or more `=` signs. This was chosen to show a very clear separation between the section of the file which just defines meta-information for the backend tooling and the section which actually defines the grammar. Anyone who is not interested in the meta-information can then quickly scan over it (either with their eyes or with a parser) to see where the line appears and the actual grammar starts.

4.2 Rules

```
S<*a, &b> -> A<*a> | B<&b>;
S => "hello";
```

Each grammar consists of a number of "rules" which are of the form `LEFT ARROW RIGHT | ALT`. Most readers should be familiar with "analytical" rules (which use the `->` arrow), but WAGon also supports "generative" rules as inspired by Zaytsev [43]. Generative rules signify that, as opposed to parsing some data or analyzing structure, this rule is intended to specify how some (possibly structured) data should be generated. Generative rules are denoted using the `=>` arrow[1].

If the rule uses attributes, they must be declared at the start by proving a comma separated list surrounded by `<>` (i.e. `S<*a, &b -> ...`). Any time a nonterminal is encountered for a rule that requires attributes, the "input" attributes should be provided in the same way.

Each complete rule is terminated with a C-style `;`.

4.2.1 Weights

```
S -> [1] A | [2 - 1] B;
```

Every alternative of a rule optionally has a weight. The weight is a full expression that should eventually evaluate to some numerical value. The exact meaning of the weight (as well as what to do with the possible absence of weight) is left up to the language designer. A comprehensive look at attribute calculations and their types is found in Sections 4.3 and 4.3.1 respectively.

Weights may only occur at the start of the alternative and are surrounded by `[]`.

4.2.2 Attribute Assignment

```
S -> A {*done = true; *map = $(external.sh)} B;
```

Defining attributes is similar to writing code. Therefore, we leverage the common syntax of C-style languages by having attribute definitions take place inside "code blocks" delineated by `{}` and separated by `;`. Every line in a block must be an assignment. This requirement of making every line an assignment (and consequential lack of control flow statements like 'for') makes the assignment language somewhat declarative in nature, similar to how grammars are declarative.

It may however occur that the language designer wants to do more complicated operations than is allowed by the DSL. Most parser generators solve this by allowing the writer to fall back to the host language in the rare cases that this is required. For WAGon however, we predict that this will happen relatively often, given that the expression language

[1] These separate types of rules are also the reason why arrows were chosen, since they lead themselves to easy variation for different meanings, as opposed to something like `=` or `:`. The meanings of these arrows are technically suggestions. A user of the ecosystem can change their meaning if they so please.

is rather limited. As such, to prevent requiring the writer from having to learn some new programming language, WAGon instead allows the designer to fall back to the shell.

Leveraging the common language of bash, any statement inside of `$()` will be forwarded to the shell of the host machine. This allows the user to write code in any programming language and simply use that to perform the more complicated calculations (for example, getting the time or reading from a database). Output from these "scripts" should be in JSON format and will be automatically parsed into a hashmap.

However, this approach drastically increases the amount of hidden dependencies and error-proneness of the DSL as any "program" may now depend on unknown scripts which perform unknown functions (which may even have side-effects). As such, this is the one design decision which is most likely to change in subsequent versions of the language.

4.3 Expression Language

```
{&a = *done * (3 + 4 * 7 + if !*check then 3 else 4)}
```

As hinted at in the previous sections, WAGon contains a mini DSL for attribute evaluation. The expression language is a mostly conventional mathematical language, supporting all the common operations (add, subtract, modulo etc.). Additionally, it also supports boolean equations (such as `1 < 2` and `!true`) as well as a basic ternary syntax (`if ... then ... else ...`). Anything which resolves to `true` should automatically be evaluated as 1 whereas anything that resolves to `false` should be evaluated as 0 and vice-versa.

4.3.1 Typing

Typing here refers to the common meaning as used in programming language design. Generally, grammars only have two basic types; terminals and nonterminals. However, when introducing attribute grammars we automatically start involving more, if only for defining the type of data that is stored in a specific attribute. The major attribute grammar parser generators such as Happy and Silver make extensive use of typing for their systems [24] [7] (partly as a consequence of their functional programming origins).

While static typing is helpful for a variety of reasons (not least of which is making the "program" less error-prone), it also introduces a considerable amount of boilerplate metadata which makes the file harder to read. Silver, for example, becomes incredibly verbose, as seen in the small excerpt in Listing 4.1.

For this reason, WAGon does not require the language designer to specify types. They are instead inferred at assignment based on the provided constants. Once a type has been inferred, it is now statically assigned to this attribute and must be cast to become a different type. The following types are defined for the language:

- Integers
- Floats
- HashMaps
- Arrays
- Strings
- Booleans

Name	Prefix	Explanation	Reasoning
Inherited	`*`	Value is defined earlier in the grammar. Any changes stay only in the local scope.	Pass by value
Synthesized	`&`	Value is defined later in the grammar. Changes are passed "upward".	Pass by reference
Local	`$`	Is defined inside of this scope.	Perl/PHP/Bash style variable instantiation
Unknown		Must be inferred from the rest of the grammar.	

TABLE 4.1: All types of attributes, their explanation and reasoning for the prefix

Additionally, the ecosystem provides functionality to allow language designers to define their own custom types if needed.

```
syn attr c :: String;
attr c occurs on Prog, Dcl, Dcls,
Type, Stmt, Stmts, Expr;
nonterminal TRep;

syn attr typerep :: TRep;
attr typerep occurs on Expr;
```

LISTING 4.1: Example of type metadata in Silver [41].

4.4 Attributes

In Section 2.2 we discussed that there are 2 types of attributes; inherited and synthesized. In WAGon, we extend these with 2 more; local and unknown as described in Table 4.1.

Local attributes function essentially the same as inherited attributes. It just provides a signifier that this location is the initial definition of the attribute. Unknown attributes are currently reserved for non-terminals. In the future however, they could be used to automatically infer the type of attribute to reduce boilerplate [43]. If an attribute is used that has not yet been defined, it should evaluate to 0.

Each instance of a different type of attribute is seen as distinct, even if they share a name. For example, `*a` and `&a` are both distinct attributes with distinct values.

4.4.1 Scope

Note that the type of attribute on the "calling" side may be different from that of the "definition" side. For example, we may have `S<*a, *b> -> A<*a, *b>;` and `A<&b, &c> -> ...;`. There are three possible approaches to dealing with this discrepancy:

1. The calling side defines how changes are propagated.
2. The defining side defines how changes are propagated.
3. Do not allow this at all.

The third option is self-explanatory, but the first two require some elaboration. Say we have the following grammar:

```
S<&x> -> {&x = 2} A<&x>;
A<*x> -> {*x = 3} ;
```

What is the final value of `&x`? If we take the first approach, then it is a synthesized attribute and as such, it value should come from the "lower" rule, meaning the final value is 3. If we take the second approach, that means that `A` will handle it as inherited and not

propagate changes upward, making the final value 2. The first approach is the classical approach as already defined by Knuth [21], the second approach is easier to implement.

The ecosystem is mostly agnostic in which approach can be taken. It is ultimately left to the language designer. However, the parser generator takes the second approach, as it was deemed significantly easier to implement. This may require a new way of looking at how attributes are propagated for people who are already familiar with the classical approach. Ideally, in the future, tooling could be added to the ecosystem for the classical approach, allowing us to standardize it into the language.

4.5 Terminals

```
A     -> "a";
B     -> `b`;
Word  -> /[a-zA-Z]+/;
```

The leafs of a parse tree are the Terminals of the grammar. WAGon allows for two types of terminal:

1. String literals (defined using either `"` or `` ` ``)
2. Regular Expressions (delineated by `//`)

Regular Expressions sit halfway between terminals and non-terminals. They express more complicated rules (like non-terminals), but also function as leaves of the tree (no other rules have to be parsed to deal with them). At the parsing level, the regular expression should return a greedily matched string.

In the ecosystem, the Regular Expression is parsed using a native library and compiled to a DFA. This DFA can then either be used directly or compiled to bytecode and loaded later on in any generated code that requires it.

4.6 EBNF Operators

```
S -> A* B+ C?;
```

EBNF operators are shorthand for more complicated rules. WAGon allows them for both terminals and non-terminals and the ecosystem includes tooling to automatically rewrite them to their equivalent normal form. The way these basic rewrites are done can be found in Table 4.2.[2] However, while this is relatively simple to do for normal grammars, everything becomes much more complicated when attributes are involved. For example:

```
A<*a> -> B<*a, $b>+;
```

This essentially means we want to parse `B` 1 or more times, given the same attributes each time. Any changes that are propagated upward in a singular "call" of `B` should thus be propagated upward in the "recursive call" as well. If we use the classical approach as explained in Section 4.4.1 then this will be handled automatically, but if we use the other approach, we need to do some fancier rewrites.

Any attributes that are passed to a chunk with an EBNF operator, or that are used inside a grouped chunk, must be available in the helper rules. Additionally, any modifications made to these attributes must be passed upwards as if the helper rules were completely

[2]Note that these rewrites use right-recursive rules. They can trivially be implemented using left-recursive rules if so desired.

Operator	Original	Rewrite
?	`A -> B?;`	`A -> A·0·0;` `A·0·0 -> B | ;`
*	`A -> B*;`	`A -> A·0·0;` `A·0·0 -> B A·0·0 | ;`

+	`A -> B+;`	`A -> A·0·0;`
		`A·0·0 -> B A·0·0·p;`
		`A·0·0·p -> B A·0·0·p | ;`

() (groups)	`A -> (B);`	`A -> A·0·0;` `A·0·0 -> B;`

TABLE 4.2: EBNF Rewrite Rules

inlined. As such, we pass all the required attributes encountered in the chunks as they were originally written to the first helper rule. From then on, we treat each attribute as synthesized, so that any changes will be properly passed up to the original calling rule. For example:

```
A<*a> -> B<*a, $b>+;
   ⇓
A<*a> -> A·0·0<*a, $b>;
A·0·0·p<&a, &b> -> B<&a, &b> A·0·0·p<&a, &b> | ;
A·0·0<&a, &b> -> B<&a, &b> A·0·0·p<&a, &b>;
```

Regardless of whether attributes are involved, EBNF rewrites require the creation of helper rules. For each symbol $X \in V$ that contains EBNF operators, we can create helper rules X_{ij} where i is the alternative of the rule this operator occurred in and j is the index of the specific "chunk" (read, either nonterminal symbol or grouped collection of symbols) the EBNF operator occurred at. For the `+` operator, an additional rule $X_{ij}p$ is needed. If we have several layers of nested groups (for example, `(A (B)+)+`) we create new rules X_{ijk} where k is the current "depth" of the group we are rewriting.

4.7 Modular Grammars

```
A <- A::S;
B <= B::S;
C << C::S;
C </ 1 & 2 & 3;
```

In programming, we are used to being able to re-use code written by other, better, programmers to perform tasks we do not want to develop ourselves. Any major programming language is expected to have a good library ecosystem if it wants to see any chance at adoption, but in the world of grammar design, we are often found re-inventing the wheel.

Major parser generators do usually have some sort of "importing" mechanism [30] [12], but grammars are not programs and it would be silly to lift the mechanisms of importing wholesale. Instead, WAGon intends to use the concept of Modular Grammars as defined by Johnstone et al [19].

Instead of simply copying over all the rules from the other file which are not defined in the current file (as e.g. ANTLR does), modular grammars allow a much more *modular* approach to importing grammars and leverages the language of grammars to do so.

In WAGon, rules can be defined which import rules from other files in 3 ways:

- Basic - denoted by `<- <NT>`, simply copies over a line from another file directly.

- Full - denoted by `<= <NT>`, copies over the rule from file B into file A but changes every reference from `B::<NT>` into `A::<NT>`
- Recursive - denoted by `<< <NT>`, the same as full but performs the reference changes recursively downwards.

When a grammar is imported using either full or recursive arrows, any imports which are done in that file should be resolved first.

Additionally, one can use `</ <INDEX> & <INDEX> ...` to remove specific alternatives from an imported rule. Finally, one can always just include another file and refer to rules in that file using the format `<MODULE>::<NT>`.

By allowing language designers to import other grammars in such a modular manner, it becomes possible to construct a corpora of grammars for various general purposes which can be easily modified as needed. For example, one could simply import the Python 2 and Python 3 grammars and use weighted attributes to decide what gets imported from which file depending on the file that is being parsed.

This modular approach fits the design goal of sticking as close as possible to formal grammars, by having even the importing mechanism operate in this world.

4.8 Complete DSL

A full formal description of the WAGon language (written in the DSL itself) can be found in Listing 4.2.

```
Wag                 -> Metadata? Rule*;

// Metadata Section
Metadata            -> Meta* MetaDelim;
MetaDelim           -> "==" "="+;
Meta                -> Include | Config;
Include             -> Identifier? "::" Identifier Include?;
Config              -> Identifier ":" Expression ";";

// Production Rules
Rule                -> Identifier NTArgs? "->" Rhs;
Rhs                 -> Weight? Chunk* "|" Rhs
                    |  Weight? Chunk* ";"
                    ;
Weight              -> "[" Expression "]";
Chunk               -> ChunkP EbnfType?;
EbnfType            -> "+" | "*" | "?";
ChunkP              -> Symbol
                    |  "(" Chunk* ")"
                    ;
Symbol              -> NonTerminal
                    |  Terminal
                    |  Assignment
                    |  // This is an empty rule, aka ϵ aka epsilon.
                    ;

NonTerminal         -> Identifier NTArgs?;
NTArgs              -> "<" AttrIdentifierList ">";
AttrIdentifierList  -> AttrIdentifier "," AttrIdentifierList
                    |  AttrIdentifier
                    ;
```

```
Terminal            -> "/" /[^/]*/ "/" // Regex
                    |  String
                    ;
Assignment          -> "{" (AttrIdentifier "=" Expression ";")* "}";

// Attribute Evaluation
Expression          -> SubProc
                    |  If
                    |  Disjunct
                    ;
SubProc             -> "$(" /[^)]*/ ")";
If                  -> "if" Disjunct "then" Disjunct ("else" Expression)?;
Disjunct            -> Conjunct ("&&" Disjunct)?;
Conjunct            -> Inverse ("||" Conjunct)?;
Inverse             -> "!"? Comparison;
Comparison          -> Sum (CompOp Sum)?;
CompOp              -> "<" | "<=" | ">" | ">=" | "==" | "!=" | "in";
Sum                 -> Term SumP?;
SumP                -> SumOp Term SumP?;
SumOp               -> "+" | "-";
Term                -> Factor TermP?;
TermP               -> TermOp Factor TermP?;
TermOp              -> "*" | "//" | "/" | "%";
Factor              -> Atom ("**" Factor)?;
Atom                -> AttrIdentifier
                    |  Dictionary
                    |  Bool
                    |  Num
                    |  Float
                    |  String
                    |  "(" Expression ")"
                    ;

Identifier          -> /[a-zA-Z][a-zA-Z0-9_]*/;
AttrIdentifier      -> AttrSpec? Identifier;
AttrSpec            -> "$" | "*" | "&";

Dictionary          -> AttrIdentifier "[" Expression "]";
Bool                -> "true" | "false";
Num                 -> /[0-9]+/;
Float               -> /[0-9]+.[0-9]+/;
String              -> '"' /[^"]*/ '"'
                    |  "'" /[^']*/ "'"
```

Listing 4.2: Complete WAGon DSL

Chapter 5

The WAGon Ecosystem

"rogaq.brflijplqxymtwsh.xbouldrzchxawsl."

Library of Babel, page 23 of Volume 8
on Shelf 4 of Wall 2 of Hexagon 0

Let us open this section by stating that the WAGon source-code is open-source and publicly available at https://github.com/Rafaeltheraven/wagon. In this chapter, we will be discussing the various modules as well as explaining how it can be utilized to create WAG based tools. A comprehensive overview of all the modules can be found in the online documentation at https://dulfer.be/wagon/.

5.1 The Libraries

The goal of WAGon is to prevent re-inventing of the wheel. Therefore, the source-code has taken the form of an ecosystem of libraries. Language designers can step in at whatever level is best for them as well as access various helper modules which should be useful for most WAG based tools. If the language as described in Chapter 4 is the spec, then the ecosystem is the "reference implementation". This reference implementation was written in the `Rust` programming language and some `Rust` specific terminology will be used.

5.1.1 User Facing

The ecosystem consists out of two types of crates, "user-facing" and "backend". While all the crates are available for anyone to use, a few crates form the "top" of the dependency graph (depicted in Figure 5.1) and are the first place for developers to look when wanting to work with WAGon.

5.1.1.1 WAGon Parser

The parser for the WAGon DSL can be found in the `wagon-parser` crate. It provides a struct that, given a `String` with the DSL as input, either returns a full AST representing the input file, or a proper error. The error can be handled as needed, the AST can function as the basis for any code generation or analysis that one wants to do.

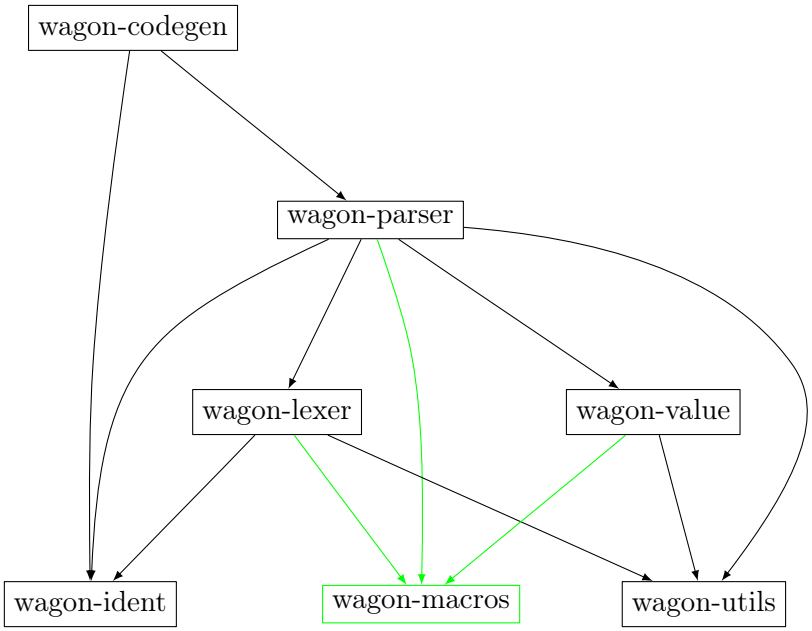


FIGURE 5.1: WAGon dependency graph. Green indicating build-time dependencies.

Additionally, this crate includes a very simple checker which does the following things:

1. Check whether any rule has duplicate attributes in their declaration (i.e. `S<*a, *a> -> ...;`).
2. Check whether alternate definitions of the same nonterminal have different attributes in their declarations (i.e. `S<*a> -> ...; S<*b, *c> -> ...;`).
3. Merge multiple rules for the same nonterminal into a single rule with alternatives.
4. Factor our EBNF operators as described in Section 4.6

This crate will be the main entrypoint for any language designer who wants to use WAGs. Listing 5.1 shows how to start using the crate.

```
1 use wagon_parser::parse_and_check;

3 let input_grammar = ...;
4 let wag = parse_and_check(input_grammar);
5 assert!(wag.is_ok());
```

LISTING 5.1: `wagon-parser` example

This crate also defines the AST for a fully parsed WAGon WAG. It's structure mirrors that of the formal DSL described in Listing 4.2.

5.1.1.2 WAGon Codegen

An obvious use of WAGs is to generate code based on what it defines. While some of this codegen is highly specific, some aspects will be the same regardless. `wagon-codegen` is a crate which includes generic functionality for code generation. It provides the following features:

1. Convert a full WAGon attribute expression into valid Rust code (either for weights or for attribute assignment).
2. A struct to represent a file structure + data in memory and write it to disk.
3. A trait for any codegen which is based on, and needs to keep track of, some state object.

Any language designer who wants to do their own code generation can use this crate to deal with expressions and assume that all value conversions and operations are dealt with. An example for how to use this crate can be found in Listing 5.2.

```
use wagon_codegen::FileStructure;
use wagon_codegen::ToTokensState;
use wagon_parser::parser::expression::Expression;

ls fs = FileStructure::new();
let expression: Expression = ... // Get expression node representing, i.e. `2 + 3`.
let state: State = ... // State object keeping track of attributes present in expression.
let label = proc_macro2::Ident::new("current_rule");
let code = expression.to_tokens(state, label, State::callback);
assert_eq!(code, quote!(
    wagon_value::Value(2) + wagon_value::Value(3);
));
fs.insert_tokenstream("file.rs", code, true); //`true` pretty prints the code.
fs.write_to_disk(); // You will now have a file called "file.rs" with the code.
```

Listing 5.2: `wagon-codegen` example

5.1.2 Backend

If `wagon-parser` and `wagon-codegen` do not provide enough functionality for the user, they can start using the "backend" crates. These crates drive the "frontend" and may give more power and extensibility as required.

5.1.2.1 WAGon Value

As described in Section 4.3.1, the WAGon DSL is pseudo-dynamically typed. `wagon-value` is a crate which handles all the dynamic typing aspects. It also allows for extension such that, if a language designer wants to introduce more types than the basic ones, they can. The code generated by `wagon-codegen` (described above), will always resolve to the basic `Value` enum defined in this crate. If the language designer wants to use their own type, all they have to do is define conversions and ensure the conversion happens at the end (an example of this is provided in Section 5.2).

In order for a type to be able to function as a `Value`, it must implement the `Valueable` trait, which is defined as in Listing 5.3.

Additionally, one must implement the traits for all basic arithmetic and comparative operations[1]. This way, we can ensure that anything that implements `Valueable` functions as we expect a dynamically typed value to function.

[1] `Add`, `Sub`, `Mul`, `Div`, `Rem`, `Pow`, `PartialEq` and `PartialOrd`

```rust
use wagon_value::ValueResult;
trait Valueable {
    /// Is this value seen as ‘true‘ or ‘false‘?
    fn is_truthy(&self) -> ValueResult<bool, Self>;
    /// Convert the value to a regular [‘i32‘].
    fn to_int(&self) -> ValueResult<i32, Self>;
    /// Convert the value to a regular [‘f32‘].
    fn to_float(&self) -> ValueResult<f32, Self>;
    /// Get a string representation of the value, as if it were a number.
    fn display_numerical(&self) -> ValueResult<String, Self>;
}
```

Listing 5.3: Code for the `Valueable` trait

5.1.2.2 WAGon Lexer

If one ever wants to introduce new tokens to the WAGon DSL, they should be defined in the `wagon-lexer` crate. This crate provides an automatically context-switching lexer (meaning that it only lexes tokens in the expression language when actually inside an expression block) which can also be used directly if one were to want to roll their own parser. An example of how to use this crate can be found in Listing 5.4.

```rust
let s = r#"
meta: "data";
============
S -> A;
"#;
use wagon_lexer::{Tokens, LexerBridge, LexResult};
use wagon_ident::Ident;

let lexer = LexerBridge::new(s);
let tokens: Vec<LexResult> = lexer.collect();
assert_eq!(tokens, vec![
    Ok(Tokens::MetadataToken(Metadata::Identifier("meta".into()))),
    Ok(Tokens::MetadataToken(Metadata::Colon)),
    Ok(Tokens::MathToken(Math::LitString("data".to_string()))),
    Ok(Tokens::MathToken(Math::Semi)),
    Ok(Tokens::MetadataToken(Metadata::Delim)),
    Ok(Tokens::ProductionToken(Productions::Identifier(Ident::Unknown("S".to_string())))),
    Ok(Tokens::ProductionToken(Productions::Produce)),
    Ok(Tokens::ProductionToken(Productions::Identifier(Ident::Unknown("A".to_string())))),
    Ok(Tokens::ProductionToken(Productions::Semi))
])
```

Listing 5.4: `wagon-lexer` example

5.1.2.3 WAGon Utils

`wagon-utils` is a crate which provides methods which are generically useful when programming in Rust using the ecosystem. Of special note are its `ErrorReport` trait and `handle_error` function. `ErrorReport` provides a consistent interface for defining `Error`

types with a message, header and associated span information. These can then be handed to `handle_error` which will print an error message to the console indicating exactly where in the file the error occurred. The output of this can be found in Figure 5.2.

```
Error: Unexpected Token
   ┌─[test.wag:1:1]
 1 │ S -> {!left = 0; $right = 0} A<$left> A<$right>;
   │       ┬ 
   │       └─ Encountered token MathToken(!) at position 6..7. Expected "identifier"
   ──────────
```

FIGURE 5.2: Example error message created by `wagon-utils`

5.1.2.4 WAGon Macros

`wagon-macros` provides build-time macros to make it slightly easier to write valid `Rust` code. Most importantly, it provides the `Base` enum which provides the general tokens used by `wagon-lexer` in all "languages", as well as easy derive macros to add support for various operators to custom defined `Value`s.

5.1.2.5 WAGon Ident

Attribute identifiers (e.g. `*a`) need to be recognized and used in various places throughout the ecosystem. `wagon-ident` very simply functions as a central location to retrieve the `Ident` struct that represents these identifiers.

5.2 Creating a Parser

In order to demonstrate the capabilities of WAGon, we have created a GLL-based parser using the ecosystem. We will discuss the specifics of how the parser works and what it was used for in Chapter 6. In order to create the parser, we have introduced 3 new crates (which can also be seen in the new dependency graph in Figure 5.3).

5.2.1 WAGon GLL

Generalized LL (GLL) is a rather complicated parsing scheme with many moving parts. The exact nature of GLL (and our extensions to it) are discussed later in Chapter 6. For the purposes of this chapter, we simply note that a dedicated library is required which handles all the common GLL tasks. This library can be found in the `wagon-gll` crate. The interesting bit of the library when it comes to the ecosystem is that that `wagon-gll` provides an extension of `Value`.

It was originally intended for the generated parser to support higher-order nonterminals, meaning that a nonterminal could be stored in an attribute and then referred to elsewhere in the grammar. While this functionality was removed, we decided to keep the extension of `Value` made to support this, as it provides a nice example for how to work with the library. A (shortened) version of the "extension" code can be found in Listing 5.5.

In this example, we create an enum with one variant that wraps `InnerValue` and another variant in case it's a `GLLBlockLabel`. We annotate the wrapped `InnerValue` with `#[value_variant]` such that the `wagon_macros::ValueOps` derive macro can automatically create implementations for us that perform all basic operations. Finally, we implement the `Valueable` trait by either forwarding the function to the `InnerValue` or by calculating

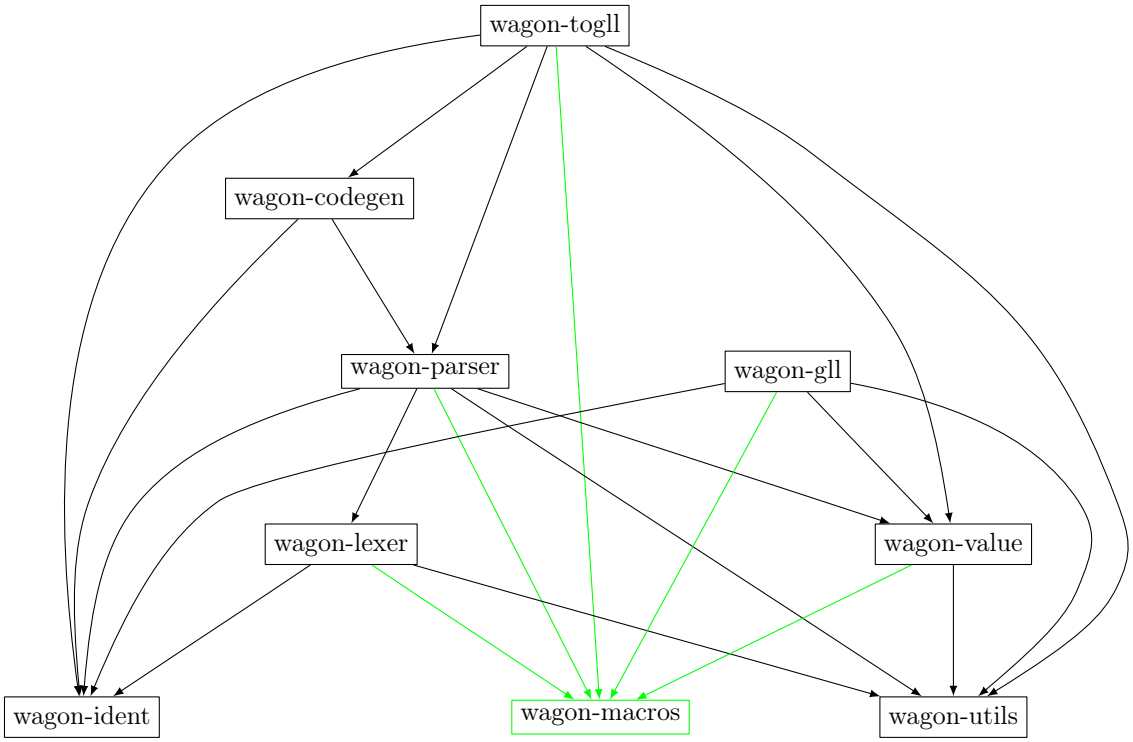


FIGURE 5.3: WAGon case study dependency graph

what makes sense for `GLLBlockLabel`. We now have an enum which can do everything `Value` can, as well as hold some additional types.[2]

```
use crate::GLLBlockLabel;
use wagon_value::Value as InnerValue;
use wagon_value::ValueError as InnerValueError;
use wagon_value::ValueResult as InnerValueResult;
use wagon_value::Valueable;
use wagon_macros::ValueOps;

#[derive(Debug, PartialEq, Eq, Hash, Clone, ValueOps)]
pub enum Value<'a> {
    #[value_variant]
    Value(InnerValue<Value<'a>>),
    Label(GLLBlockLabel<'a>),
}

#[derive(Debug)]
pub enum ValueError<'a> {
    ValueError(InnerValueError<Value<'a>>),
    ConvertToLabel(Value<'a>)
}
```

[2]Not pictured here, a number of trivial type conversions that can be implemented to make generating code a bit easier.

```rust
impl<'a> From<InnerValueError<Value<'a>>> for ValueError<'a> {
    fn from(value: InnerValueError<Value<'a>>) -> Self {
        Self::ValueError(value)
    }
}

impl<'a> From<InnerValueError<InnerValue<Value<'a>>>> for ValueError<'a> {
    fn from(value: InnerValueError<InnerValue<Value<'a>>>) -> Self {
        Self::ValueError(value.into())
    }
}

impl<'a> Valueable for Value<'a> {
    fn is_truthy(&self) -> InnerValueResult<bool, Self> {
        match self {
            Value::Value(v) => Ok(v.is_truthy()?),
            Value::Label(l) => Ok(l.is_eps()),
        }
    }

    fn to_int(&self) -> InnerValueResult<i32, Self> {
        match self {
            Value::Value(v) => Ok(v.to_int()?),
            o @ Value::Label(_) => Ok(i32::from(o.is_truthy()?))
        }
    }

    fn to_float(&self) -> InnerValueResult<f32, Self> {
        match self {
            Value::Value(v) => Ok(v.to_float()?),
            o @ Value::Label(_) => Ok(if o.is_truthy()? { 1.0 } else { 0.0 })
        }
    }

    fn display_numerical(&self) -> InnerValueResult<String, Self> {
        match self {
            Value::Value(v) => Ok(v.display_numerical()?),
            other @ Value::Label(_) => Ok(other.to_int()?.to_string())
        }
    }
}
```

Listing 5.5: `wagon-gll` `Value` extension

`wagon-gll` is technically not part of the WAGon ecosystem (as it instead exists to support the implementation of a tool *using* the ecosystem). As such, it can be found in its own repository at https://github.com/rafaeltheraven/wagon-gll.

5.2.2 WAGon TOGLL

The second half of the fully functional GLL parser is to generate the code which actually implements a specified grammar. This is done by the `wagon-togll` crate. Again, the exact specifics of what code needs to be generated will be discussed in Chapter 6. For the purposes of this chapter, let us look at how it works with the ecosystem.

The first thing the crate does is to take a fully formed WAG AST as defined by the `wagon-parser` crate and to walk it. It starts by looking at the metadata section and checking for the following `key: value` pairs:

1. `no_prune` (default `false`) - If set to `true`, the generated parser will not consider the weights when deciding which alternative to parse.
2. `min_weight` (default `false`) - If set to `true`, the generated parser will only take alternatives with the lowest weight, as opposed to the highest weight.
3. `first_set` (default `true`) - If set to `false`, the generated parser will ignore the first/follow set when choosing what alternative to parse.
4. `allow_zero` (default `false`) - If set to `true`, the generated parser will allow alternatives with a weight of 0 to be parsed.

Afterwards, it goes through all the rules + their alternatives and generates the code as needed. When it reaches an expression (either inside of a weight or attribute assignment block), `wagon_codegen::ToTokensState` is invoked on it to automatically generate working Rust code that evaluates to a `wagon_value::Value` type. This final value is then converted to `wagon_gll::value::Value`.

Once all the generated code has been collected, `wagon_codegen::FileStructure` is used to define the following file tree:

1. `main.rs` - Main executable code. Parsing input arguments to executable and setting up the `wagon_gll::GLLState` object.
2. `regexes/` - Directory holding regex recognizing DFAs.
3. `nonterminals/` - Directory holding structs representing each nonterminal.

This crate also provides an executable, which take the output `FileStructure` described above and either writes it to disk if successful (meanwhile also setting up the various dependencies using `cargo`), or prints an error message using `wagon_utils::handle_error` if an error occurred anywhere in the process.

Both the executable provided by `wagon-togll` and the executable generated by it are provided with an explanatory `help` flag. The output of this flag (explaining how to use the executables) can be found in Figure 5.4.

Like `wagon-gll`, `wagon-togll` is not part of the WAGon ecosystem itself. Instead being an executable that uses the ecosystem. As such, it too can be found in its own repository: https://github.com/rafaeltheraven/wagon-togll.

```
Example tool that uses the WAGon ecosystem to create GLL based parsers

Usage: wagon-togll [OPTIONS] <filename> <project_name>

Arguments:
  <filename>      The input WAGon grammar file
  <project_name>  The name of the project to output

Options:
      --overwrite  Delete any existing project with the same name
  -h, --help       Print help
  -V, --version    Print version
```

(A) `wagon-togll --help` output.

```
Usage: gll_test [OPTIONS] <filename>

Arguments:
  <filename>  Input file to parse

Options:
      --no-crop    Don't crop resulting sppf
      --math-mode  Print SPPF dot labels in Latex math-mode representation
      --print-gss  Also print the final GSS (works with math-mode)
  -h, --help       Print help
  -V, --version    Print version
```

(B) `--help` output for generated parser.

FIGURE 5.4: How to use the various executables.

Chapter 6

Of Parser Generators and Pokémon

"To protect the world from devastation!"

Jessie

While it is fun to discuss WAGon in the abstract world of "possible usecases", a concrete example can make things much clearer. For this purpose, we provide a fully functioning parser generator, written using WAGon and, through the use of unique WAG abilities, utilizing it to verify basic aspects of legality for various creatures from the Pokémon series of video games.

6.1 GLL Parsing

Parser generators are tools which, given a defined grammar, generate a functioning parser for this language defined by the grammar. Many parser generators, however, have two specific limitations:

1. The grammar must be deterministic.
2. The grammar may not contain left-recursive rules[1].

Generalized LL (GLL) parsing is a technique which works for *all* context-free grammars, regardless of whether they are left-recursive or deterministic [36]. The fact that it allows non-determinism is interesting for WAGs, because the attributes can be used to curb the amount of possible branches taken [26].

To fully understand how GLL works, we refer you to the vast amount of literature on the subject by Johnstone et al. [36] [17] [18]. For our purposes, we will provide the following simple explanation:

- We create a "virtual machine" (or state object) which keeps track of all parsing information.
- The virtual machine receives parse jobs, based on a position in the input and what (part of a) rule it is currently in.
- In case a rule has multiple valid alternatives, we simply add a job for each valid alternative to the queue.

[1] Or **hidden** left-recursion in the case of LR-style parser generators

- Completed parse jobs are stored in an Shared Packed Parse Forest (SPPF), a special graph which consists out of multiple tree-like structures.
- The state of the parser is stored on a Graph Structured Stack (GSS), which is used similarly to how a conventional stack is used in compiler design.
- The exact same parse job may be added to the queue multiple times, in which case the full parse is retrieved from the SPPF in a classic case of dynamic programming.

6.1.1 Trait-Oriented GLL Parsing

The original GLL algorithm makes liberal use of GOTOs, a feature not supported by most modern programming languages. In order to implement GLL in modern languages, an Object-Oriented approach (OOGLL) was developed by Bram Cappers [6]. However, WAGon was written in Rust, which specifically does not support object-oriented programming. As such, we had to slightly modify the OOGLL algorithm in order to introduce a new approach, Trait-Oriented GLL (TOGLL) parsing.

First, we define a new trait, which we will call `Label`. In order to implement the trait properly, the following methods **must** be defined:

1. `is_eps()` - Check whether this label is for an ϵ block.
2. `first_set()` - Returns the first-follow set for this rule/GLL-block.
3. `code()` - The code that should be run once it is this label's turn.

In addition, based on the previous methods, the following methods can be automatically defined:

1. `first()` - Check whether, given the current state of the parser, this label can be parsed. Based on the first-follow set.
2. `is_nullable()` - Check whether this label can ever resolve to ϵ.

For a given grammar G, we will generate a struct for every production rule in G as well as every GLL block in G. Each struct implements the `Label` trait. The struct naming scheme is the same as the function naming scheme in OOGLL. Every rule S simply gets a struct named `S`. Then, for every GLL block i in alternative j of rule S we define a struct `S_j_i`.

Secondly, we define a state object we will call `GLLState`. This state object takes the place of the generic parser in OOGLL and implements the functions `goto()`, `init()`, `create()`, `add()`, `get_node_t()`, `get_node_p()` and `test()`. As opposed to function pointers, the `GLLState` queue will consist out of the previously generated `Label` implementing structs. `goto()` then, takes one of these structs and simply runs the associated `code()` method.

6.1.1.1 Code Generation

In order to show how exactly `Label` can be implemented, let us look at the following grammar:

```
S -> A B 'b' | 'b';
A -> 'a';
B -> 'b';
```

The rule `S` has 2 alternatives. The first one consisting of 3 GLL blocks and the second of only 1. Thus, we will create the following structs; `S`, `S_0_0`, `S_0_1`, `S_0_2` and `S_1_0`. The pseudo-code for struct `S` can be found in Listing 6.1.

```
1  // For every rule S -> x_1|...|x_j ∈ G:
2  impl Label for S {
3      fn is_eps() {
4          return false
5      }
6      fn first_set() {
7          return [x_1, ..., x_j]
8      }
9      fn code() {
10         if state.test(x_1) {
11             state.add(x_1, state.gss_pointer, state.input_pointer, $)
12         }
13         ⋮
14         if state.test(x_j) {
15             state.add(x_j, state.gss_pointer, state.input_pointer, $)
16         }
17     }
18 }
```

Listing 6.1: Pseudo-code for rule structs

Now, we just need to generate structs for each GLL block. Every GLL block is responsible for 2 things: ensuring that the correct bit is parsed, and ensuring that the next block is queued. If this block is the last block, it instead needs to ensure that parser state is restored. There are a lot of little bits to keep track of when it comes to generating these structs. In Listing 6.2 you will find pseudo-code for the generation of a GLL-block struct. Code colored pink represents the meta-logic for what lines of code to generate, whereas code colored black represents the final struct definition.

Finally, `Label` should be implemented for whatever type represents a Terminal (be it an array of bytes, or a string of characters or what have you). The implementation for this type can be found in Listing 6.3. This is needed mostly for type coercing reasons.

An example implementation of the `GLLState` object and the `Label` trait can be found in the `wagon-gll` crate, whereas an example implementation of the code generation can be found in the `wagon-togll` crate.[2]

6.1.2 Weights & Attributes

GLL (and consequently, OOGLL/TOGLL) was not designed with attribute grammars in mind. However, the use of the GSS and SPPF give us very useful data structures to store the attribute information in. Inspired by the work of Josh Mengerink [26], we extend the GSS to handle inherited attributes and the SPPF to handle synthesized attributes.

[2]Note that both implementations have differences from the minimal explanation above. This was done to support various features which are not strictly necessary for TOGGL to function.

```
// For every rule S → x_1|...|x_j ∈ G, for every alternate x_i in that rule,
// for every GLL-block g_k = α_1...α_n ∈ x_i:
impl Label for S_i_k {
    fn is_eps() {
        if |g_k| = 0 {
            return true
        } else {
            return false
        }
    }

    fn first_set() {
        return [α_1]
    }

    fn code(state: GLLState) {
        for α_m ∈ g_k {
            if m ≠ 0 {
                if state.has_next(α_m) {
            }
                if α_m ∈ T { // For the purposes of this algorithm, ϵ ∈ T
                    if m = 0 ∧ |g_k| ≠ 1 { // And |ϵ| = 1
                        state.sppf_pointer = state.get_note_t(α_m, state.input_pointer)
                        state.next(α_m)
                    }
                    if m ≠ 0 ∨ (m = 0 ∧ |g_k| = 1) {
                        node = state.get_node_t(α_m, state.input_pointer)
                        state.next(α_m)
                        slot = new GrammarSlot<S, S_i, k>
                        state.sppf_pointer = state.get_node_p(slot, state.sppf_pointer,
                            node)
                    }
                } else {
                    slot = new GrammarSlot<S, S_i, k+1>
                    state.gss_pointer = state.create(slot)
                    α_m.code()
                }
            if m ≠ 0 {
                }
            }
        }
        if m = n ∧ k = j { // If this is the last symbol in the last block
            state.pop()
        }
    }
}
```

LISTING 6.2: Pseudo-code for generating GLL-block structs

```
impl Label for Terminal {
    fn is_eps() {
        return self.is_empty()
    }
    fn first_set() {
        return []
    }
    fn code() {
        return Error(``UNREACHABLE'')
    }
}
```

Listing 6.3: `Label` implementation for Terminals.

6.1.2.1 Extending the GSS

```
S<*a, *b> -> {*a = 0; *b = 0;} A<*a> A<*a>;
A<&a> -> {&a = &a + 1} ...
```

Listing 6.4: Example WAGon grammar using attributes.

The GSS, in the context of attributes, is responsible for 2 things: passing attributes "down" and storing attributes for later. Let us use the grammar in Listing 6.4 as an example.

In TOGLL, the rule `S` will be decomposed into 2 GLL blocks; `S_0_0` and `S_0_1`. First, we encounter `S_0_0` and queue it as a parse job. This is analogous to calling a function in a conventional programming language and in a similar way, we need somewhere to store the context of our variables, so that we can recover it when we continue with the rule `S`. In a conventional programming language, this info is stored on the stack. In our extension of TOGGL, it is stored on the GSS. Similarly, when we want to pass "arguments" (I.E. `*a` for `A<*a>`), we also store those on the GSS. In our example, we store `*a` on the GSS to pass it along to `A` and we store `*b` on the stack to recover it later. Once we start parsing `A`, we retrieve `&a` from the GSS node. Once we are done parsing `A`, we can recover `*b` by retrieving it from the GSS. Section 6.1.2.2 describes how we retrieve `*a`.

This can be implemented either through adding 2 separate vectors to the GSS (one to store context, one to pass them down) or, since we know exactly what attributes exist and which ones are passed down, into a singular vector which we virtually split into 2 at the codegen level. The vector(s) is added to the GSS node in the state object's `create` method.

The GLL algorithm sometimes compares GSS nodes to define whether a particular parser state has already occurred. In order to make sure this functions properly, the attribute vectors of 2 GSS nodes must also be compared (so two nodes are only equal if they have the same grammar slot, input pointer **and** attribute vectors) [26].

6.1.2.2 Extending the SPPF

Reading the SPPF Before we discuss how the SPPF is extended, it may be useful to explain how to read our extended SPPF such that the figures we will provide make more sense.

The SPPF consists out of 3 types of nodes:

1. Symbol
2. Packed
3. Intermediate

In this paper, packed and symbol nodes will be drawn in ellipses whereas intermediate nodes will be drawn in boxes.

Packed and symbol nodes are represented in their respective conventional denotations, intermediate nodes are extended with a list of all the attributes and their values at that point of the parse. These attributes are enclosed by `<>`. An intermediate node which represents a completed rule will be denoted with just the NT of that rule. All attributes in that node represent the state of the attributes **after** the rule has been completely parsed.

Additionally, any time we have a non-zero weight, we print the value of the weight on the edge from the intermediate node representing the completed rule to the packed node representing the chosen alternative.

Intermediate nodes If the GSS stores information "downward", then the SPPF provides information "upward". It, too, performs 2 tasks: passing changes to synthesized attributes "up" and telling the parser at what GSS node the attribute context is stored.

In our example in Listing 6.4, `&a` in `A` is treated as a synthesized attribute. Inside of the rule, we take whatever value it has and increment it by one. It is mostly treated like any other attribute, stored on the GSS and updated from the SPPF as needed, until the end of the rule. In GLL, every time a rule has been completely parsed, we call `pop`. This makes it the perfect place to send synthesized values upward. Once we call the state object's `pop` method, we pass along all the synthesized attributes and store them on the newly created intermediate SPPF node. When we then continue in the previous rule (`S`), we retrieve the new value of `*a` from the vector stored on the SPPF, instead of from the one stored on the GSS. When we've parsed the first `A<*a>`, the value of `*a` should thus be 1. After we've parsed the second `A<*a>`, it should be 2.

Because of the way GLL works, `state.sppf_pointer` will always point to the correct node for retrieving synthesized attributes, but this is not necessarily the case for attributes stored in the GSS, which is why we also extend the SPPF (and the state object) with a `context_pointer`. Note that this issue only exists if GSS nodes are immutable.
As described in Section 6.1.2.1, we write attributes to the GSS during the `create` method. If we have mutable GSS nodes, we can write these attributes simply to wherever the `state.gss_pointer` is currently pointing. If we have immutable GSS nodes however, we must write the attributes to the newly created GSS node. To keep track of where the attributes are stored (since it might be in some other node than wherever `state.gss_pointer` is pointing), we store a pointer to the correct node in the Intermediate nodes of the SPPF. We also extend descriptors to include this same pointer, such that we can quickly tell the state object where the current context is stored.

Additionally, when we are comparing two Intermediate SPPF nodes for equality, we want to take the context of the attributes into account. Obviously, a parse of `A` in which `&a` $= 1$ is different from one in which `&a` $= 2$, regardless of whether they represent the same grammar slot and chunk of input. For this, we can again use the added context pointer. Since we only care about attribute equivalence, we do not have to compare 2 GSS nodes, and can simply compare the associated attribute vectors directly. For this reason, we still want to store the context pointer with the intermediate nodes, even if we have mutable GSS nodes.

Note that it is possible for attribute evaluation to happen after parsing of a rule has been completed. As such, we need to store the complete state of all the attributes at the end of every parsed rule. In GLL, the end of a rule is always accompanied by a call to `pop`, making it the perfect opportunity to do this specific store operation. Inside the `pop` method, we can create a new GSS node without any edges to store the attributes on. At the moment `pop` is called, the SPPF node representing the fully parsed rule is stored at the `self.sppf_pointer`. The context pointer for this node should be set to the newly created GSS node.[3]

With all the extensions, two intermediate SPPF nodes are equal if and only if:

1. Both represent the same grammar slot.
2. They consume the same stretch of the input stream.
3. They both "return" the same synthesized attributes + values.
4. They both have the exact same attribute contexts.

Packed nodes Similarly to how equality of intermediate nodes is changed by the introduction of attributes, so too are packed nodes. In the original GLL algorithm, we check whether a given packed node exists for a given parent and do nothing if it already exists. We do not inspect the potential children of the packed node because we know that, given the same parent, slot and split point, they must always be the same. This is not the case once we introduce attributes. Let us take the example in Figure 6.1. As the parse occurs, at some point the packed node for $(S \to A\langle *x\rangle \bullet, 0)$ is created. There are two ways to reach this packed node, either because we took the alternative `A<*x> -> "a" A`, in which case `*x` $= 0$, or we took `A<*x> -> *x = *x + 1 "a" A`, in which case `*x` $= 1$. Clearly, these are different parses, but if we use the original GLL algorithm, the second instance of the packed node will be deemed the same as the first. After all, they consume the same slot, have the same parent, the same split and even the same attribute context (`*x` $= 0$).
The only way to differentiate between the two packed nodes is by looking which intermediate nodes will become it's children. In the first case, it will be $(A, 0, 1, \langle *x : 0\rangle)$ and in the second case it will be $(A, 0, 1, \langle *x : 1\rangle)$. If we do not check the children for equality, the second packed node will be mistaken as identical to the first and not be added to the SPPF, causing $(A, 0, 1, \langle *x : 1\rangle)$ to never be connected to the root node, causing that parse to become unreachable.

A packed node always has at most 2 children, so comparing child nodes is $O(2)$ and has little impact on performance. Furthermore, directly comparing pointers to the children (such as node indices in the graph) will suffice, as GLL will never allow two intermediate nodes with the exact same data to exist at different memory addresses. There is no need to do any in-depth comparisons.

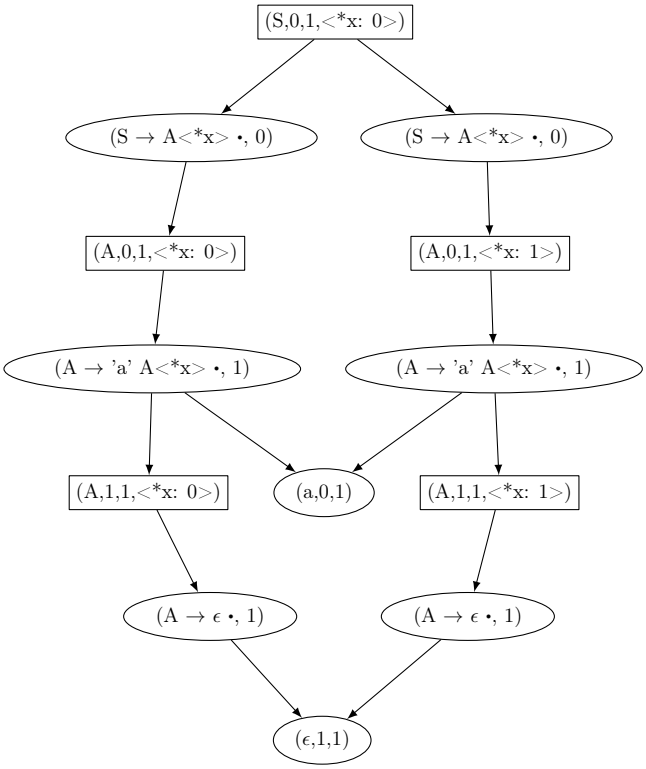
6.1.2.3 Left-Recursion

Of note is that the introduction of attribute vectors can cause the left-recursion handling mechanisms of GLL to fail. Let us take the grammar in Listing 6.5 for example. In this case, we first schedule a job for `S` with `*x` $= 0$. Then, we schedule a new job for `S` with `*x` $= 1$. In conventional GLL, the parser would stop here, as the second job for `S` is the same as the first, so it does not continue. With attributes however, every job $\texttt{S}_i$ is distinctly

[3] Alternative approaches were considered and are possible, as long as the final complete attribute context is stored somewhere retrievable.

```
S      -> {*x = 0} A<*x>;
A<*x> -> {*x = *x + 1} "a" A
       | "a" A
       |
       ;
```

(A) Grammar for language $L = a^n$



(B) AST for input "a"

FIGURE 6.1: Grammar + SPPF with subtly distinct packed nodes.

different from $\texttt{S}_{i-1}$ and as such, it will keep queuing forever. Because attributes have an infinite context space, it is impossible to completely solve this issue.

```
S<*x> -> {*x = *x + 1} S<*x>
       |
       ;
```

LISTING 6.5: Left recursive grammar.

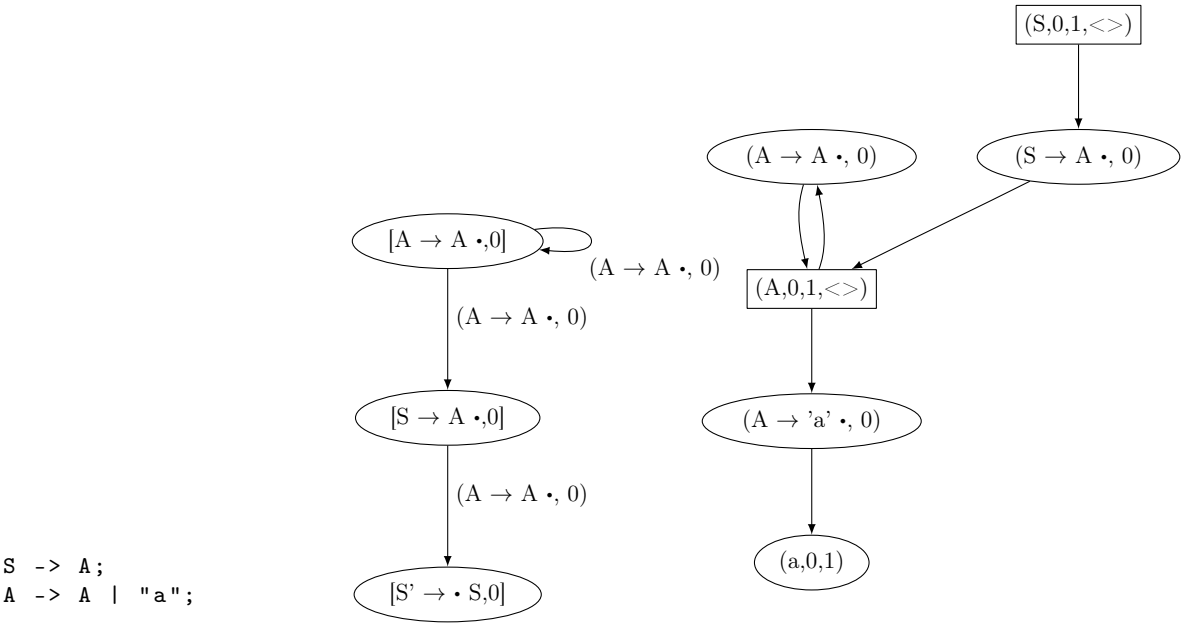
```
S<*x> -> [*x < 3] {*x = *x + 1} S<*x>
       |
       ;
```

LISTING 6.6: Left recursive grammar with guard to stop infinity.

This issue can be partly resolved by simply adding weights as guards for left-recursive rules or by not having attributes in left-recursive rules, such as in Listing 6.6. However, the introduction of attributes breaks the left-recursion mechanism in these cases as well, in slightly more subtle ways.

In classic GLL, a left-recursive grammar will cause the GSS and SPPF for the recursive parse to create a cycle, such as in Figure 6.2. Note how the intermediate node $(A, 0, 1, <>)$ creates a cycle with the packed node $(A \rightarrow A\bullet)$ indicating that one can "take" the rule `A -> A` an infinite number of times.

Let us now consider Figure 6.3 in which we add attributes (and a guard as discussed earlier). The GSS correctly removes the self-loop from $[A \rightarrow A\bullet, 0]$ because we define GSS nodes to be distinct if their attribute vectors are distinct. However, the GLL algorithm still queues up a packed node $(A \rightarrow A\bullet)$ which (now) incorrectly loops to $(A, 0, 1, <>)$. We know that a direct loop in the SPPF can occur if and only if it matches directly with a self-loop in the GSS. As such, we can easily remedy this issue by adding a check to `get_node_p` which only allows loops between an intermediate node and a packed node if we just created a GSS loop as well, resulting in the correct SPPF in Figure 6.4.



(A) Basic left recursive grammar.

(B) GSS for input "a".

(C) SPPF for input "a".

FIGURE 6.2: Left recursive grammar + resulting GSS and SPPF

6.1.3 Pseudo-Code

With all the modifications described in the previous sections, it is useful to have a single conclusive description of what the GLL algorithm looks like now. Inspired by the pseudo-code of Cappers [6], all new additions will be written in blue whereas any meta logic on the code generation level will be written in pink (like in section 6.1.1.1) and text which is both will be in yellow[4].

6.1.3.1 The State Object

The state object will need to provide the methods described in Listing 6.7.

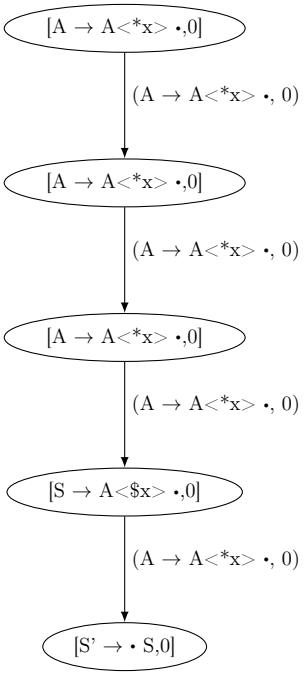
6.1.3.2 The Labels

Additionally, as discussed in Section 6.1.1.1, each GLL Block needs an associated struct that implements the `Label` trait as described in Listing 6.8. Note that we add a new method `weight` which provides the weight value for the GLL block given the current parser state. Additionally, each rule also needs an associated struct as described in Listing 6.9

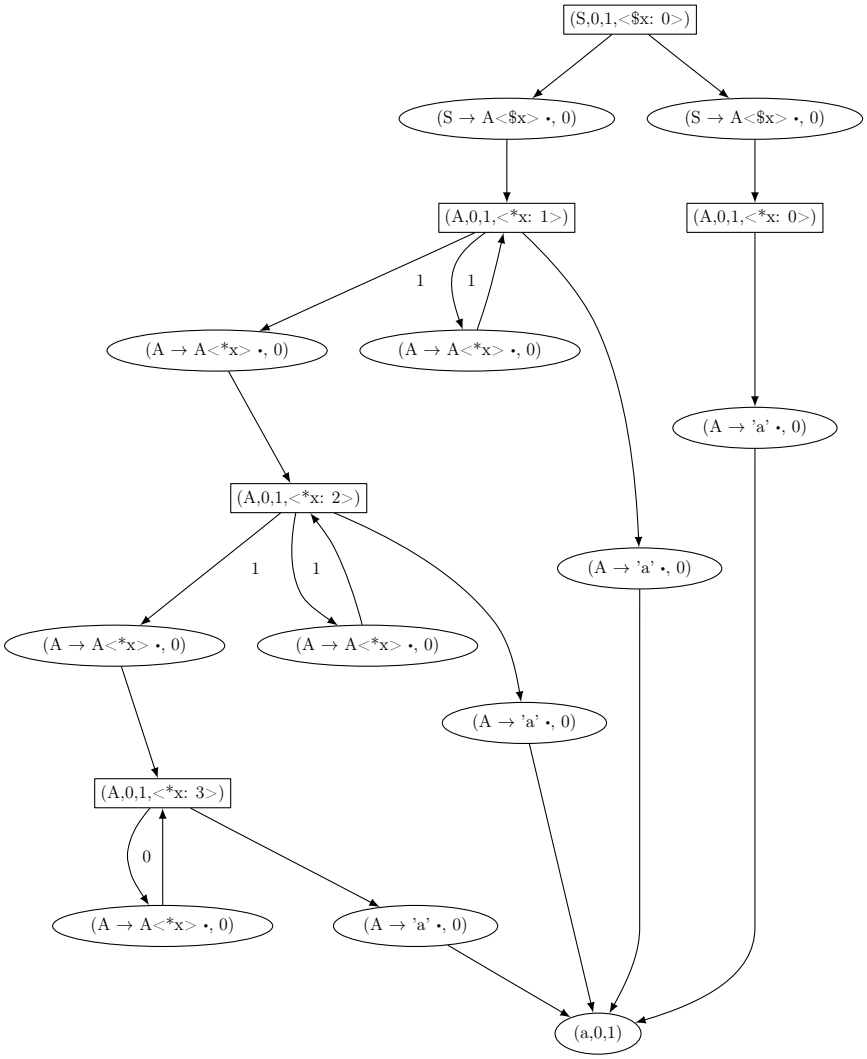
[4]These colors should be differentiable to our colorblind friends

```
S       -> {$x = 0} A<$x>;
A<*x>   -> [*x < 3] {*x = *x + 1} A<*x>
        | "a";
```

(A) left recursive grammar with attributes



(B) GSS for input "a".



(C) SPPF for input "a".

FIGURE 6.3: Left recursive grammar with attributes + resulting GSS and SPPF

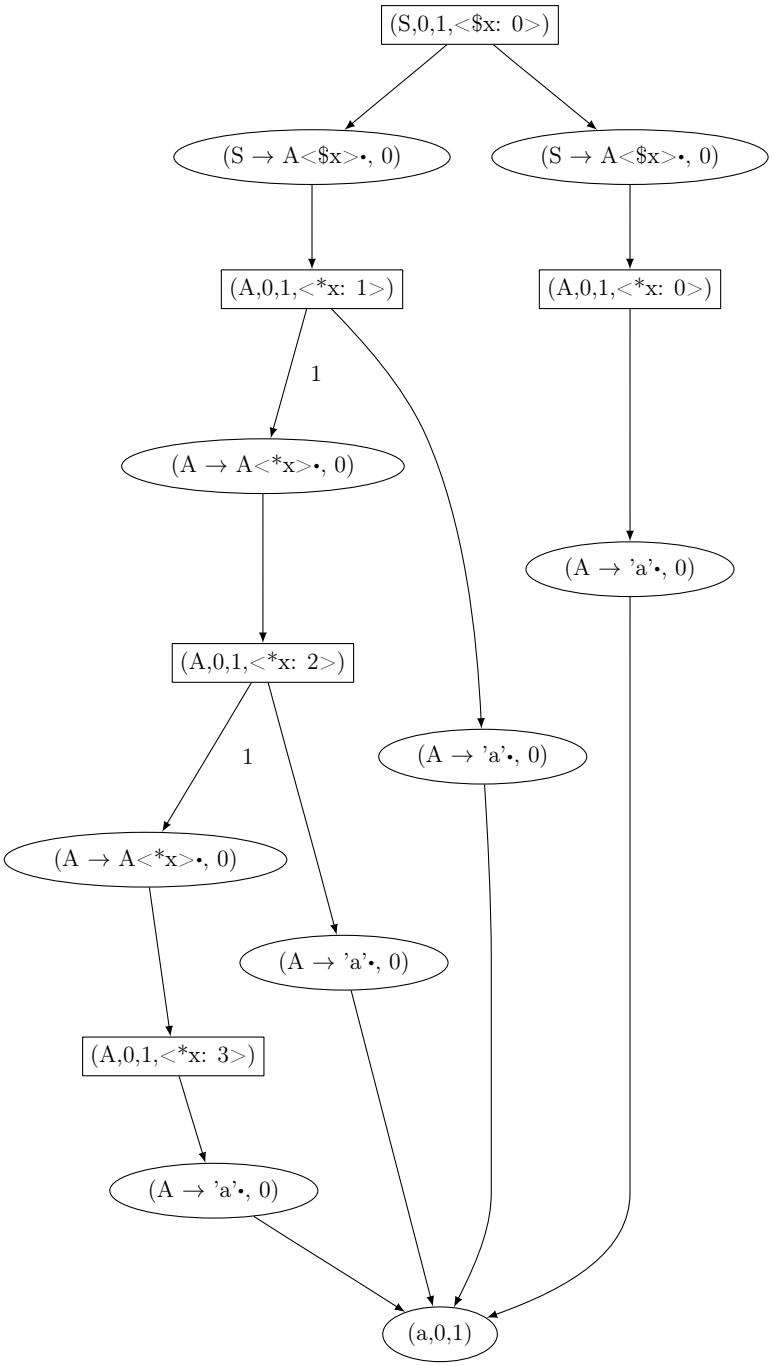


FIGURE 6.4: Fixed SPPF for input "a" from Figure 6.3

```
fn init(input_stream) -> GLLState {
    initialize  data structures
    let gss_root = ⟨⊥, 0, []⟩
    let sppf_root = $
    let state = ...
    state.add(S' → • S , gss_root, 0, $, gss_root)
    return state
}

fn add(slot: GrammarSlot, g: GSSNode, i: Integer, s: SPPFNode, context: GSSNode) {
    let d: Descriptor = (slot, g, i, s, context)
    if ¬U.contains(d) {
        U.insert(d)
        R.insert(d)
    }
}

fn get_node_p(slot: GrammarSlot, left: SPPFNode, right: SPPFNode,
                context: GSSNode, gss_cycle: bool) -> SPPFNode {
    let slot be of the form A → α • β
    if |α| = 1 ∧ (head(α) is a terminal or a non−nullable nonterminal) ∧ β ≠ [] {
        return right
    } else {
        if β ≠ [] {
            t = A
        } else {
            t = slot
        }
        if left == $ {
            let i, j = the left and right extends of the SPPF node right
            let node = find or create Intermediate SPPF node (t, i, j, context)
            if (gss_cycle || right != node) &&
                    node does not have a packed node (A → α • β, i, None, right) {
                create one with child right
            }
        } else {
            let j = the right extend of SPPFNode right
            let i, k = the left and right extends of the SPPF node left
            let node = find or create Intermediate SPPF node (t, i, j, context)
            if (gss_cycle || (right != node && left != node)) &&
                    node does not have a packed node (A → α • β, k, left, right) {
                create one with children left, right
            }
        }
        return node
    }
}

fn get_node_t(t: Terminal, left: Int, right: Int) -> SPPFNode {
```

```
    if ¬S.contains((t, left, right)) {
        create symbol node (t, left, right)
    }
    return (t, left, right)
}

fn create(slot: GrammarSlot, args: Array[Attribute]) -> GSSNode {
    if there is no GSS node v == (slot, self.input_pointer, args) {
        create one
    }
    if there is no edge from v to self.gss_pointer {
        create one with self.sppf_pointer as the weight
        for [(s, z) ∈ P|s = ⟨slot, state.input_pointer⟩] {
            let y = self.get_node_p(slot, self.sppf_pointer, z, v, v == self.gss_pointer)
            self.add(slot, self.gss_pointer, z.right_extend(), y, v)
        }
    }
    return v
}

fn pop(synth_attrs: Array[Attribute], attrs: Array[Attribute]) {
    let slot = self.gss_pointer.slot
    let ctx = find or create GSS node <slot, self.input_pointer, attrs>
    self.sppf_pointer.context = ctx
    if self.gss_pointer != self.gss_root {
        P.add((self.gss_pointer, self.input_pointer))
        for each edge from self.gss_pointer to v {
            let z = edge.weight()
            let y = self.get_node_p(slot, z, self.sppf_pointer,
                        self.gss_pointer, v == self.gss_pointer)
            y.synth_attrs = synth_attrs
            self.add(slot, v, self.input_pointer, y, self.gss_pointer)
        }
    }
}

fn get_attribute(i: Ident) -> Attribute {
    self.gss_pointer.get_attribute(i)
}

fn restore_attribute(i: Ident) -> Attribute {
    self.context_pointer.get_attribute(i)
}

fn get_synth_attribute(i: Ident) -> Attribute {
    self.sppf_pointer.get_synth(i)
}

fn goto(label: Label) {
```

```
    label.code(self)
}

fn main() {
    while (¬R.empty()) {
        (slot, gss, input, sppf, context) = R.pop()
        self.sppf_pointer = sppf
        self.gss_pointer = gss
        self.input_pointer = pointer
        self.context_pointer = context_pointer
        self.goto(slot.label)
    }
}
```

LISTING 6.7: Pseudo-code for general GLL methods

```
// For every rule S → x_1|...|x_j ∈ G, for every alternate x_i in that rule,
// for every GLL-block g_k = α_1...α_n ∈ x_i:
impl Label for S_i_k {
    fn is_eps() {
        if |g_k| = 0 {
            return true
        } else {
            return false
        }
    }

    fn first_set() {
        return [α_1]
    }

    fn code(state: GLLState) {
        for α_m ∈ g_k {
            if m ≠ 0 {
                if state.has_next(α_m) {
            }
                if α_m ∈ T { // For the purposes of this algorithm, ϵ ∈ T
                    if m = 0 ∧ |g_k| ≠ 1 { // And |ϵ| = 1
                        state.sppf_pointer = state.get_note_t(α_m, state.input_pointer)
                        state.next(α_m)
                    }
                    if m ≠ 0 ∨ (m = 0 ∧ |g_k| = 1) {
                        node = state.get_node_t(α_m, state.input_pointer)
                        state.next(α_m)
                        slot = new GrammarSlot<S, S_i, k>
                        state.sppf_pointer = state.get_node_p(slot, state.sppf_pointer,
                            node, state.gss_pointer, false)
                    }
                } else {
                    Retrieve attributes from state object
```

```
                    Evaluate attribute expressions // This can happen in multiple places
                    slot = new GrammarSlot<S, S_i, k+1>
                    state.gss_pointer = state.create(slot, [attributes in context])
                    α_m.code(state)
                }
            if m ≠ 0 {
                }
            }
        }
        if m = n ∧ k = j { // If this is the last symbol in the last block
            state.pop([Synthesized attributes])
        }
    }

    fn weight(state: GLLState) -> Value {
        if k = 0 { // If this is the first gll block
            Retrieve all attributes from the state object
            weight = Evaluate the weight expression
            return weight
        } else {
            return Error("UNREACHABLE")
        }
    }
```

LISTING 6.8: Pseudo-code for generating GLL-block structs with attributes

```
// For every rule S -> x_1|...|x_j ∈ G:
impl Label for S {
    fn is_eps() {
        return false
    }
    fn first_set() {
        return [x_1, ..., x_j]
    }
    fn code(state: GLLState) {
        if state.test(x_1) {
            state.add(x_1, state.gss_pointer, state.input_pointer, $, state.gss_pointer)
        }
        ⋮
        if state.test(x_j) {
            state.add(x_j, state.gss_pointer, state.input_pointer, $, state.gss_pointer)
        }
    }

    fn weight(state: GLLState) -> Value {
        return Error("UNREACHABLE")
    }
}
```

LISTING 6.9: Pseudo-code for rule structs

6.2 Validating Pokémon

Pokémon is a multimedia franchise created by Game Freak Inc. in 1996 [40]. In the flagship video game series, players are tasked with capturing and fighting alongside creatures called Pokémon. In game, these Pokémon each have distinct stats and movesets, based on how their owner has trained them. These stats and movesets have restrictions on them which are difficult (if not impossible) to define in a conventional grammar, but are simple to express in a WAG. Popa already used WAGs to generate Pokémon-esque creatures [31]. Now, to show the power of the WAGon ecosystem, we will validate them.

6.2.1 Dataset

Pokémon has an active competitive scene whose members meet up either in real-life or on websites like `https://play.pokemonshowdown.com/` to battle it out in tournaments or friendly matches. When players want to share their teams, they use a format called Poké-paste which is described informally at `https://pokepast.es/syntax.html`. We will be using publicly posted Pokémon in this format to test our program.

6.2.2 Restrictions

There are a lot of restrictions put on what Pokémon are valid (for example, it must be an existing Pokémon to begin with). For our purposes, we have encoded the following restrictions in our grammar:

1. Each Pokémon has at least 1 move and at most 4.
2. Each Pokémon has between 0 and 6 IVs. Each IV has a value between 0 and 31.
3. Each Pokémon has between 0 and 6 EVs. Each EV has a value between 0 and 255. The sum of all EVs must be smaller than or equal to 510.

In addition, while parsing, we can also check the following attributes:

1. Is the Pokémon Shiny?
2. Does the Pokémon have a nickname?
3. What gender is the Pokémon?
4. Does the Pokémon have an item?

While more checks could be implemented (for example, making sure Pokémon only know actually valid moves), those quickly becomes more of an exercise in simply having a large enough database and are less interesting in our quest to show what WAGon can do.

6.2.3 Validation

Given the dataset and the wanted restrictions described above, we come to the WAGon grammar described in Listing 6.10. WAGs (and thus, WAGon) allow us to employ multiple tricks to easily validate the grammar and our additional imposed restrictions.

6.2.3.1 Relaxed Order

Poké-paste is a very relaxed format. At the start, we have an informational section which is largely optional (outside of the Pokémon's legal name). After that is a statistics section which can be written out in any order and after that we have a list of up to 4 moves. While the optional parts are easy to define in most conventional grammars (as long as they support EBNF operators), allowing a section to be written in any order would be extremely cumbersome to define in a conventional grammar (as you would need to write out every possible permutation). In WAGs, however, we can add some very simple boolean logic to check whether we have already parsed a certain statistic and to disallow parsing it again from that point onward.

6.2.3.2 Calculating Guards

When we want to calculate the value of a number using a conventional parser generator, we would have to fall back to the host language in order to do so. In AGs, we can derive the value of a number at parse time, which we do in the `Decimal` rule. We can then use the value calculated by the `Decimal` rule and whether it falls inside a certain range by adding a special rule afterwards which functions as a guard (such as is done in `EVs` employing the `PerEVGuard` rule).

Other types of values can also be calculated in this way (E.G. `YesNo` calculating booleans). If we combined this with the modular grammar features described in Section 4.7 one could start using a "library" style approach to grammar writing. Allowing us to simply import utility rules like `Decimal` which parse and calculate our values for us.

6.2.3.3 Summarizing

Some of the attributes employed are "utility" attributes. We utilize them to make sure the input is valid, but after that they become unimportant. Some of the attributes however, are important until the end because they give us information about the input we are interested in. At the top, we create a special rule `S` which defines these attributes we care about. At the end, the SPPF will then have a root node which shows the final values of the attributes, leaving the others lower in the tree.

Having to define these attributes up front and constantly passing them downwards is pretty cumbersome and leads to a slightly bloated grammar. We discuss approaches to solving this issue in Section 7.1.

```
S -> {$shiny = false; $nickname = false; $gender = "U"; $item = false}
     Pokemon<$shiny, $nickname, $gender, $item>;

Pokemon<&shiny, &nickname, &gender, &item> ->
    Info<&nickname, &gender, &item>
    {$d_ability = false; $d_shiny = false; $d_level = false; $d_happy =
    false; $d_nat = false; $d_ev = false; $d_iv = false}
    Optionals<$d_ability, $d_shiny, $d_level, $d_happy, $d_nat, $d_ev,
    $d_iv, &shiny>*
    {$move_count = 0} Moves<$move_count>+
    ;

Info<&nickname, &gender, &item> ->
                                  Name
                                  ("(" Name {&nickname = true} ")")?
                                  ("(" Gender<&gender> ")")?
                                  ("@" Item {&item = true})?
                                ;
Name             -> AnyString;
Gender<&gender> -> "M" {&gender = "M"} | "F" {&gender = "F"};
Item             -> AnyString;

Optionals<&d_ability, &d_shiny, &d_level,
            &d_happy, &d_nat, &d_ev, &d_iv, &shiny> ->
      [!&d_ability] Ability                        {&d_ability = true}
    | [!&d_shiny]   Shiny<&shiny>                  {&d_shiny   = true}
    | [!&d_level]   Level                          {&d_level   = true}
    | [!&d_happy]   Happiness                      {&d_happy   = true}
    | [!&d_nat]     NatureDef                      {&d_nat     = true}
    | [!&d_iv]      {$iv_count = 0}
                    IVStart<$iv_count>             {&d_iv      = true}
    | [!&d_ev]      {$ev_count = 0; $ev_total = 0}
                    EVStart<$ev_count, $ev_total>
                    EVGuard<$ev_total>             {&d_ev      = true}
    ;

Ability -> "Ability" ":" AnyString;

Shiny<&shiny> -> "Shiny" ":" YesNo<&shiny>;
YesNo<&yes>   -> "Yes" {&yes = true} | "No" {&yes = false};

Level               ->
                      {$total = 0}
                      "Level" ":"
                      Decimal<$total>
                      LevelGuard<$total>
                    ;
LevelGuard<*total> -> [*total <= 100];

Happiness           ->
                      {$total = 0}
                      "Happiness" ":"
                      Decimal<$total>
                      HappyGuard<$total>
                    ;
HappyGuard<*total> -> [*total <= 255];

NatureDef -> Nature "Nature";

EVStart<&ev_count, &total> -> "EVs" ":" EVList<&ev_count, &total>;
```

```
EVList<&ev_count, &total>  ->
                             EVs<&ev_count, &total>
                             '/'
                             EVList<&ev_count, &total>
                           | EVs<&ev_count, &total>
                           ;
EVs<&ev_count, &total>     -> [&ev_count < 6]
                             {$total = 0}
                             Decimal<$total>
                             PerEVGuard<$total>
                             {&total = &total + $total}
                             Stat
                             {&ev_count = &ev_count + 1}
                           ;
PerEVGuard<*check>         -> [*check <= 252];
EVGuard<*check>            -> [check <= 510];

IVStart<&iv_count> -> "IVs" ":" IVList<&iv_count>;
IVList<&iv_count>  ->
                      IVs<&iv_count>
                      '/'
                      IVList<&iv_count>
                    | IVs<&iv_count>
                    ;
IVs<&iv_count>     -> [&iv_count < 6]
                      {$total = 0}
                      Decimal<$total>
                      IVGuard<$total>
                      Stat
                      {&iv_count = &iv_count + 1}
                    ;
IVGuard<*total>    -> [*total <= 31];

Stat -> "HP" | "Atk" | "Def" | "SpA" | "SpD" | "Spe";

Moves<&move_count> -> [&move_count < 4]
                      "-"
                      MoveList
                      {&move_count = &move_count + 1}
                    ;
MoveList -> AnyString "/" MoveList | AnyString;

AnyString -> /[a-zA-Z]+([ -]*[a-zA-Z]+)*/;

Decimal<&total> -> NumberList<&total>;
NumberList<&total> ->
                      {$value = 0}
                      Number<$value>
                      {&total = &total * 10 + $value}
                      NumberList<&total>
                    |
                      {$value = 0}
                      Number<$value>
                      {&total = &total * 10 + $value}
                    ;

Number<&value> ->
      '0' {&value = 0}
    | '1' {&value = 1}
    | '2' {&value = 2}
    | '3' {&value = 3}
```

```
    | '4' {&value = 4}
    | '5' {&value = 5}
    | '6' {&value = 6}
    | '7' {&value = 7}
    | '8' {&value = 8}
    | '9' {&value = 9}
    ;

Nature ->
      "Hardy"
    | "Lonely"
    | "Adamant"
    | "Naughty"
    | "Brave"
    | "Bold"
    | "Docile"
    | "Impish"
    | "Lax"
    | "Relaxed"
    | "Modest"
    | "Mild"
    | "Bashful"
    | "Rash"
    | "Quiet"
    | "Calm"
    | "Gentle"
    | "Careful"
    | "Quirky"
    | "Sassy"
    | "Timid"
    | "Hasty"
    | "Jolly"
    | "Naive"
    | "Serious"
    ;
```

LISTING 6.10: Poke-paste Grammar

6.2.4 Example Inputs

For demonstration purposes, we shall show 2 example inputs. One that is valid, and one that is invalid. They can be found in Figure 6.5. A snippet showing only the top of the SPPF for the valid input can be found in Figure 6.6 whereas the complete SPPF can be found in Appendix A. The second input will result in a parse error, returning an error message as shown in Figure 6.7. The error message is slightly confusing, which is due to the difficulty of clean error messaging in GLL parsers, but it should point the language designer in the right direction (namely that the line `Shiny: No` can not be parsed because it is unable to find a valid parse for the rule `Optional`).

```
Inteleon (F) @ Choice Specs
Ability: Torrent
Shiny: Yes
EVs: 252 SpA / 4 SpD / 252 Spe
Timid Nature
- Hydro Pump
- Ice Beam
- Air Slash
- U-turn
```

(A) Valid Input.

```
Inteleon (F) @ Choice Specs
Ability: Torrent
Shiny: Yes
Shiny: No
EVs: 252 SpA / 4 SpD / 252 Spe
Timid Nature
- Hydro Pump
- Ice Beam
- Air Slash
- U-turn
```

(B) Invalid Input.

FIGURE 6.5: Example Inputs.

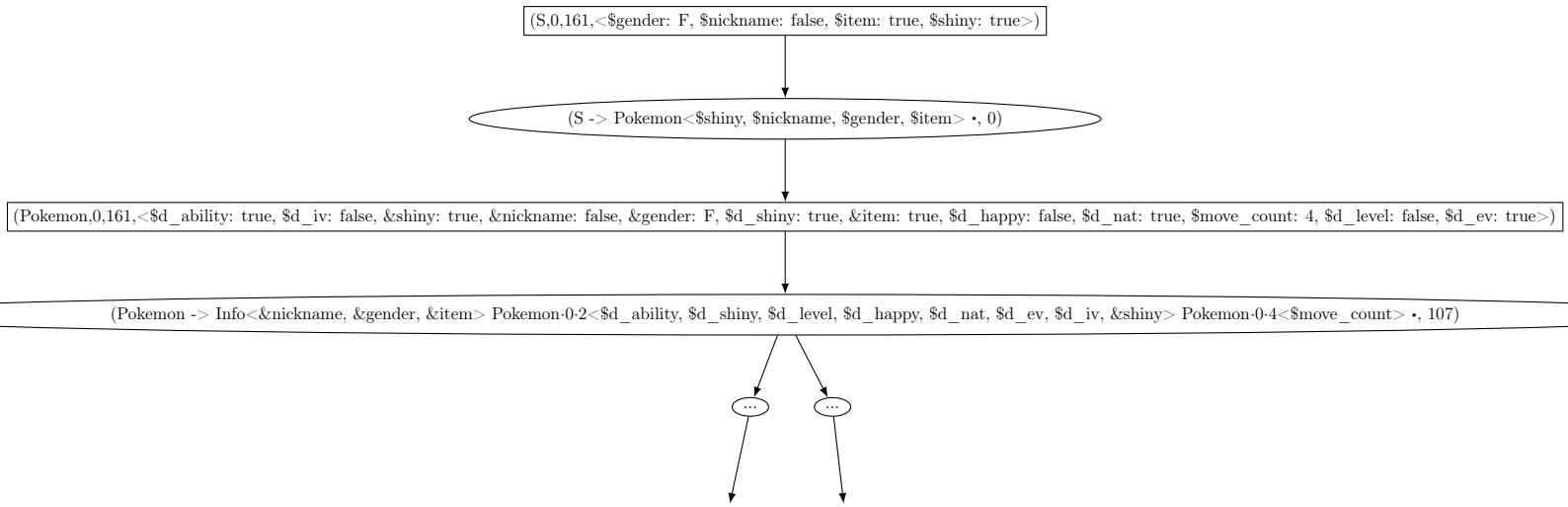


FIGURE 6.6: Top of valid (truncated) Poké-paste SPPF

```
Error: Parse Error
   ┌─[input_bad:1:3]
   │
 4 │ Shiny: No
   │   ┬
   │   └─ No parse candidates were found for rule `Optionals` in context `Pc
```

FIGURE 6.7: Example (truncated) error message for invalid input

Chapter 7

Conclusions & Future Work

> "I'm wearing dark glasses because I am seeing the future and it's looking very bright"
>
> David Lynch

7.1 Future Work

Creating a workbench means that the amount of possible projects coming from it are nearly uncountable. It is of course impossible to predict the future, but keeping an optimistic mind, we can hope that WAGon will become the basis for significant WAG related research in the future.[1] In this section we will outline some possible improvements to WAGon to make it easier to use, as well as some possible future research that WAGon could be used as a baseline for.

7.1.1 WAGon Expansions

7.1.1.1 Missing Features

Not all the features discussed in Chapter 4 made it into the final product. Specifically; modular grammars, generative arrows, bash scripts and unknown attributes, while supported by the grammar and parser, have no implemented functionality. Furthermore, type casting has no support in either the grammar or the workbench. A first step to expanding WAGon will be implementing these features. Implementing unknown attributes would be especially useful, as it can significantly lower the amount of boilerplate required to define the grammars. Additionally, in order to implement unknown attributes a call graph would be needed. This would also allow us to perform type-checking as well as implicit initializing of attributes (meaning we no longer have to define `*x = 0` all the time).

Additionally, in Section 4.4.1 we already discussed how there are two separate ways of interpreting synthesized/inherited attributes work. Properly standardizing the classical way into the DSL by adding support for it into the ecosystem is required for the language to align with our goal of having a high closeness of mapping.

[1] In fact, 2 other theses are already planning to use it as their base.

7.1.1.2 Ecosystem Improvements

Furthermore, it is of course always possible to make the ecosystem better and to have more features.

First of all, a more modular AST rewriting system is needed. Currently, all the rewrites and sanity checks of a parsed WAGon DSL AST happen at the same time and are very opinionated (for example, EBNF rewrites always use right-recursion and use the non-classical interpretation of attributes). A modular system is needed such that language designers can pick and choose the rewrites and checks that are useful for their project and can even create their own rewrites if needed. Ideally, this system would make use of the strategy pattern [14].

Secondly, it would be good for the generated GLL parser to output a more usable SPPF. The current SPPF is very large as every single node is displayed. Additionally, the only current output format is `.dot` which, while useful for research and display purposes, is less usable in the real world. Having a proper mechanism to "use" the resulting SPPF is required for more interesting practical applications of the parser.

Thirdly, allowing weights anywhere in a rule (as opposed to only at the start) would remove the need for helper "guard" non-terminals which do nothing but check a certain weight.

Finally, the ecosystem could be more generic. For example, a serialization scheme could be created to serialize the AST such that it could be used in other languages. Additionally, `wagon-parser` is tightly coupled with the tokens defined in `wagon-lexer` and `wagon-codegen` is tightly coupled with the AST defined in `wagon-parser`. Defining generic traits that handle the coupled elements (for example, having an `Expression` trait that all nodes that represent a complete WAGon Expression should implement) could allow language designers to truly take whatever bit of the ecosystem they want and easily inject their own data structures.

7.1.2 Possible WAG-based research

WAGs, being a way to define context sensitive grammars, are very powerful. We can envision the following possible use cases to be interesting research avenues[2]:

7.1.2.1 WAGs as Logic Programs

Combining WAGs with a GLL parser as described in Chapter 6 leads to interesting behavior. Consider the grammar in Listing 7.1 and resulting SPPF in Figure 7.1. It is an ambiguous grammar in which one can take either `A` rule anywhere from 0 to 2 times. By using attributes as a counter to check which point was taken, we are left with 3 roots: One in which it always takes the left `A`, one in which it always take the right `A` and one in which it takes both once.

GLL as an algorithm finds every possible interpretation of a string. We can see an intermediate node as meaning "This part of the input string can be parsed given these rules". With our extensions, this becomes "This part of the input string can be parsed, given these rules and that the attribute values at the end are as follows". If we take "this part of the input string can be parsed" to be a preposition in formal logic, the SPPF tells us "this preposition holds, given these values are as follows". In our example, the preposition holds given $(\texttt{\$left} = 0 \wedge \texttt{\$right} = 2) \vee (\texttt{\$left} = 2 \wedge \texttt{\$right} = 0) \vee (\texttt{\$left} = 1 \wedge \texttt{\$right} = 1)$.

[2] There are of course many more.

```
S -> {$left = 0; $right = 0}
     A<$left> A<$right>;
A<&count> -> "a" A<&count> {&count = &count + 1}
           | ;
```

LISTING 7.1: Highly ambiguous grammar

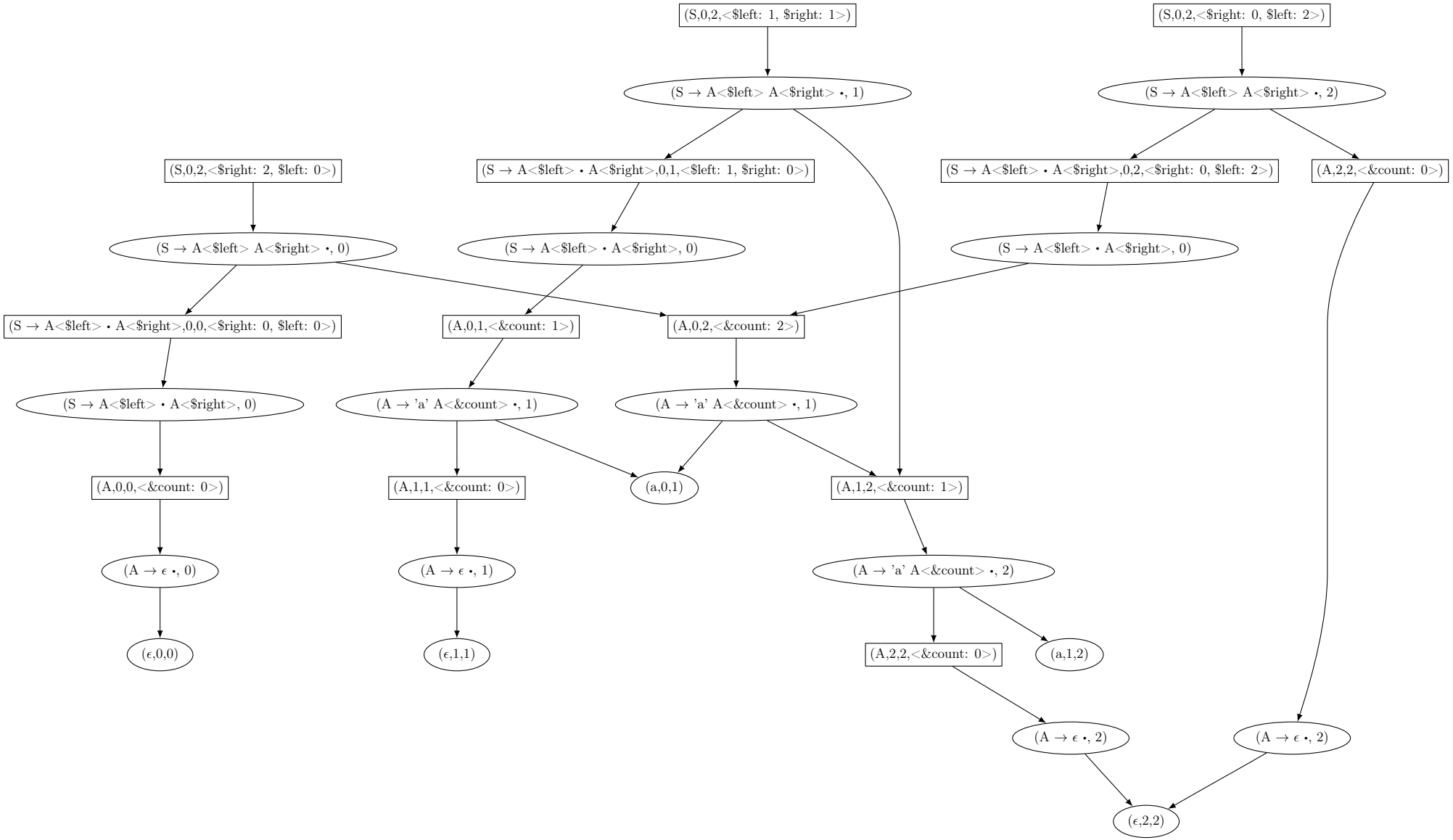


FIGURE 7.1: SPPF for Listing 7.1 with input "aa"

7.1.2.2 WAGs as Neural Networks

A WAG can be read as a concise description of a large series of if-trees. In this view, each alternative of a rule that has a weight is an alternative condition in the if-tree. While it is a reductive approach, neural networks can also be said to simply be a set of very large and complicated if-trees. If one were to define each alternative to be a neuron, and each weight of that alternative to be the weight of the neuron (we can use rules simply to group related neurons together), it seems to us potentially possible to denote neural networks (something famously difficult to write down in a human readable format) as a WAG.

7.1.2.3 WAGs for Programming Languages

Many popular programming languages, such as C++ and Rust, are context-sensitive. The solution for this "issue" by most compiler developers is to simply hand-roll a parser to deal with the resulting ambiguities. Using WAGs, one could potentially automatically generated parsers for these widely used languages.

7.2 Conclusion

In this paper, we have proposed the first standardized DSL for Weighted Attribute Grammars as well as provided a workbench such that future researches can utilize this DSL.

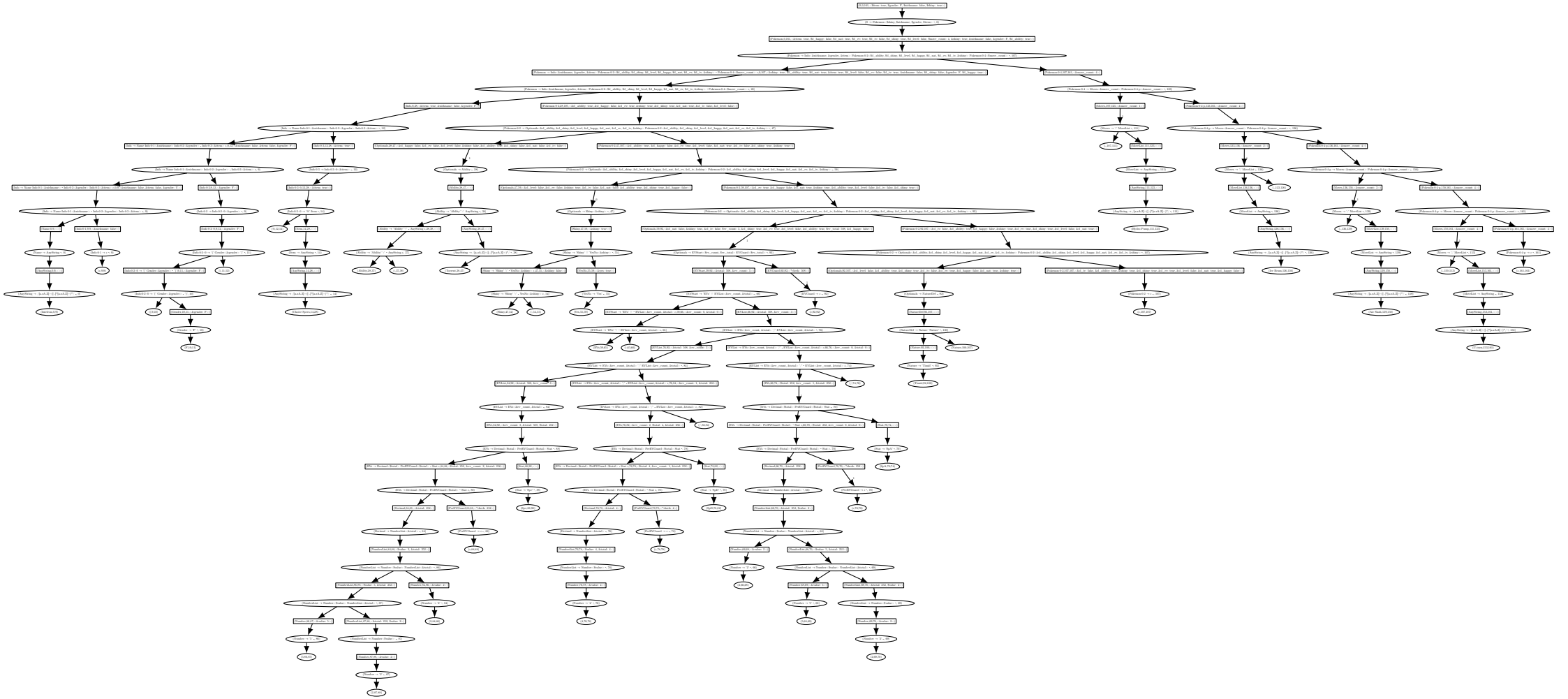
We have also investigated the potential power of WAGs through some theoretical speculation, as well as the creation of a functioning WAG-based GLL parser. Showing that it is possible to parse and analyze languages using WAGs in a way that would be impossible or highly cumbersome for conventional grammars to do. In the process of creating this parser, we have also established the alternative TOGLL parsing algorithm as well as the necessary extensions in order to implement weights and attributes.

Appendices

Appendix A

Valid Poké-paste SPPF

Note that this complete SPPF is very large. If you are reading this on a digital PDF reader, you should be able to zoom in and read it at a high resolution. If you are reading this on a physical copy, we are afraid we were unable to render this SPPF in a readable format.



Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Pub. Co., Reading, Mass., 1986. 796 pages. ISBN: 978-0-201-10088-4.

[2] C. Allauzen, M. Mohri, and B. Roark. A General Weighted Grammar Library. In M. Domaratzki, A. Okhotin, K. Salomaa, and S. Yu, editors. Redacted by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, *Implementation and Application of Automata*. Volume 3317, pages 23–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN: 978-3-540-24318-2 978-3-540-30500-2. DOI: 10.1007/978-3-540-30500-2_3.

[3] J. D. Beekman. *Procedural Location Generation with Weighted Attribute Grammars*. Bachelor's thesis, Universiteit Twente, Enschede, The Netherlands, July 2021. URL: http://purl.utwente.nl/essays/87002.

[4] A. Blackwell and T. Green. CHAPTER 5 - notational systems—the cognitive dimensions of notations framework. In J. M. Carroll, editor, *HCI Models, Theories, and Frameworks*, Interactive Technologies, pages 103–133. Morgan Kaufmann, San Francisco, 2003. ISBN: 978-1-55860-808-5. DOI: 10.1016/B978-155860808-5/50005-8.

[5] J. Brence, S. Džeroski, and L. Todorovski. Dimensionally-consistent equation discovery through probabilistic attribute grammars. *Information Sciences*, 632:742–756, June 1, 2023. ISSN: 0020-0255. DOI: 10.1016/j.ins.2023.03.073.

[6] B. Cappers. *Exploring and Visualizing GLL Parsing*. Master's thesis, Eindhoven University of Technology, Aug. 31, 2014. URL: https://research.tue.nl/en/studentTheses/exploring-and-visualizing-gll-parsing.

[7] T. Carlson and E. Van Wyk. Type qualifiers as composable language extensions. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. SPLASH '17: Conference on Systems, Programming, Languages, and Applications: Software for Humanity, pages 91–103, Vancouver BC Canada. ACM, Oct. 23, 2017. ISBN: 978-1-4503-5524-7. DOI: 10.1145/3136040.3136055.

[8] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, Sept. 1956. ISSN: 2168-2712. DOI: 10.1109/TIT.1956.1056813.

[9] N. Chomsky and M. P. Schützenberger. The Algebraic Theory of Context-Free Languages*. In P. Braffort and D. Hirschberg, editors, *Studies in Logic and the Foundations of Mathematics*. Volume 35, Computer Programming and Formal Systems, pages 118–161. Elsevier, Jan. 1, 1963. DOI: 10.1016/S0049-237X(08)72023-8.

[10] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Transactions on Computer-Human Interaction*, 28(4):28:1–28:53, July 23, 2021. ISSN: 1073-0516. DOI: 10.1145/3452379.

[11] Y. Dehbi, C. Staat, L. Mandtler, and L. Pl¨umer. Incremental Refinement of Façade Models with Attribute Grammar from 3D Point Clouds. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, III-3:311–316, June 6, 2016. ISSN: 2194-9050. DOI: 10.5194/isprsannals-III-3-311-2016.

[12] C. Donnelly and R. Stallman. *Bison Manual: The YACC-compatible Parser Generator, 3 November 1999, Bison Version 1.29*. Free Software Foundation, Boston, Mass, 1999. 94 pages. ISBN: 978-1-882114-44-3.

[13] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. In Cambridge University Press, Apr. 23, 1998. ISBN: 978-0-521-62041-3 978-0-521-62971-3 978-0-511-79049-2. DOI: 10.1017/CBO9780511790492.

[14] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994. ISBN: 0-201-63361-2.

[15] E. Habibi and S.-H. Mirian-Hosseinabadi. Generating test as a web service (TaaWS) through a method-based attribute grammar. *International Journal on Software Tools for Technology Transfer*, 24(4):511–527, Aug. 1, 2022. ISSN: 1433-2787. DOI: 10.1007/s10009-022-00649-z.

[16] G. Hedin. An Introductory Tutorial on JastAdd Attribute Grammars. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, Lecture Notes in Computer Science, pages 166–200. Springer, Berlin, Heidelberg, 2011. ISBN: 978-3-642-18023-1. DOI: 10.1007/978-3-642-18023-1_4.

[17] A. Johnstone. A Reference GLL Implementation. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*. SLE '23: 16th ACM SIGPLAN International Conference on Software Language Engineering, pages 43–55, Cascais Portugal. ACM, Oct. 23, 2023. ISBN: 9798400703966. DOI: 10.1145/3623476.3623521.

[18] A. Johnstone and E. Scott. Modelling GLL Parser Implementations. In B. Malloy, S. Staab, and M. van den Brand, editors, *Software Language Engineering*, Lecture Notes in Computer Science, pages 42–61, Berlin, Heidelberg. Springer, 2011. ISBN: 978-3-642-19440-5. DOI: 10.1007/978-3-642-19440-5_4.

[19] A. Johnstone, E. Scott, and M. van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23–43, July 2014. ISSN: 01676423. DOI: 10.1016/j.scico.2013.09.012.

[20] A. Kanev, S. Cunningham, and T. Valery. Application of formal grammar in text mining and construction of an ontology. In *2017 Internet Technologies and Applications (ITA)*. 2017 Internet Technologies and Applications (ITA), pages 53–57, Sept. 2017. DOI: 10.1109/ITECHA.2017.8101910.

[21] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, June 1, 1968. ISSN: 1433-0490. DOI: 10.1007/BF01692511.

[22] R. Lioutikov, G. Maeda, F. Veiga, K. Kersting, and J. Peters. Learning attribute grammars for movement primitive sequencing. *The International Journal of Robotics Research*, 39(1):21–38, Jan. 1, 2020. ISSN: 0278-3649. DOI: 10.1177/0278364919868279.

[23] B. J. MacLennan. "Who cares about elegance?" The role of aesthetics in programming language design. *ACM SIGPLAN Notices*, 32(3):33–37, Mar. 1, 1997. ISSN: 0362-1340. DOI: 10.1145/251634.251637.

[24] S. Marlow. Attribute Grammars — Happy documentation. Happy Documentation. 2022. URL: https://haskell-happy.readthedocs.io/en/latest/attribute-grammars.html (visited on 05/11/2023).

[25] J. McCormack. Grammar-based music composition. *Complexity International*, 3, 1996. ISSN: 1320-0682.

[26] J. Mengerink. *On-Parse Disambiguation in Generalized LL Parsing Using Attributes*. Master's thesis, Eindhoven University of Technology, Aug. 31, 2014. URL: https://research.tue.nl/nl/studentTheses/on-parse-disambiguation-in-generalized-ll-parsing-using-attribute.

[27] M. Mohri, F. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, Jan. 2002. ISSN: 08852308. DOI: 10.1006/csla.2001.0184.

[28] R. Mörbitz and H. Vogler. Weighted parsing for grammar-based language models. In *Proceedings of the 14th International Conference on Finite-State Methods and Natural Language Processing*. FSMNLP 2019, pages 46–55, Dresden, Germany. Association for Computational Linguistics, Sept. 2019. DOI: 10.18653/v1/W19-3108.

[29] S. Mukherjee and S. Mitra. Hidden Markov Models, Grammars and Biology: A Tutorial. *Journal of Bioinformatics and Computational Biology*, 03(02):491–526, Apr. 2005. ISSN: 0219-7200, 1757-6334. DOI: 10.1142/S0219720005001077.

[30] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013. 328 pages. ISBN: 978-1-934356-99-9.

[31] A. Popa. Gotta adjust them all! : dynamic difficulty adjustment of role-playing games through procedural generation of non-player characters, Master's thesis, Enschede, The Netherlands, May 2023. URL: http://essay.utwente.nl/94941/.

[32] L. Quesada, F. Berzal, and F. J. Cortijo. A Model-Driven Probabilistic Parser Generator. May 14, 2012. DOI: 10.48550/arXiv.1205.3183. arXiv: 1205.3183 [cs]. Preprint.

[33] B. Roark. Probabilistic Top-Down Parsing and Language Modeling. *Computational Linguistics*, 27(2):249–276, June 1, 2001. ISSN: 0891-2017. DOI: 10.1162/089120101750300526.

[34] R. H. Robins. *A Short History of Linguistics*, number 6 in Longman Linguistics Library. Longman, London, 3. impr edition, 1976. 248 pages. ISBN: 978-0-582-52397-5.

[35] A. Salomaa. Probabilistic and weighted grammars. *Information and Control*, 15(6):529–544, Dec. 1969. ISSN: 00199958. DOI: 10.1016/S0019-9958(69)90554-3.

[36] E. Scott and A. Johnstone. GLL Parsing. *Electronic Notes in Theoretical Computer Science*. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), 253(7):177–189, Sept. 17, 2010. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2010.08.041.

[37] S. M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, Aug. 1, 1985. ISSN: 1573-0549. DOI: 10.1007/BF00630917.

[38] A. Stefik and R. Ladner. The Quorum Programming Language (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 641, New York, NY, USA. Association for Computing Machinery, Mar. 8, 2017. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3022377.

[39] A. Stefik and S. Siebert. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*, 13(4):1–40, Nov. 2013. ISSN: 1946-6226. DOI: 10.1145/2534973.

[40] J. Tobin, redactor. *Pikachu's Global Adventure: The Rise and Fall of Pokémon*. Duke University Press, 2004. ISBN: 978-0-8223-3250-3. DOI: 10.2307/j.ctv1131ctc. JSTOR: j.ctv1131ctc.

[41] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007), 203(2):103–116, Apr. 1, 2008. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.03.047.

[42] S. Yang, Y. Zhao, and K. Tu. PCFGs Can Do Better: Inducing Probabilistic Context-Free Grammars with Many Symbols. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL-HLT 2021, pages 1487–1498, Online. Association for Computational Linguistics, June 2021. DOI: 10.18653/v1/2021.naacl-main.117.

[43] V. Zaytsev. Building Conversational AI Systems with Weighted Attribute Grammars. Preprint.