UNIVERSITEIT VAN AMSTERDAM

A Framework for SGLR Parsing in Java

C. J. Boogerd
February 2005

Master's Thesis Computer Science
Supervisors: prof. dr. P. Klint, dr. M. G. J. van den Brand

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Contents

Chapter 1

Introduction

When developing tools for language analysis and compilation, efficient and flexible parsing methods are vital building blocks. However, most widely-used deterministic techniques are limited to a very restricted class of grammars, usually LL(k) or LR(k). Since the greater part of modern programming languages do not fit these classes, one is forced to introduce workarounds when developing a language specification, thereby complicating maintenance. The Generalized LR parsing algorithm processes possible alternative derivations in parallel, and is as such able to support the full class of context-free languages. The ASF+SDF Meta Environment features a scannerless implementation of this algorithm, but due to heavy code modifications over the years this was not an appealing tool for use in further experiments. This was the reason for work on a re-implementation of the SGLR parser, taking particular care in ensuring a flexible architecture that would allow frequent use in new experiments.

1.1 The ASF+SDF Meta Environment

The SGLR parser is one of the main components of the ASF+SDF Meta Environment. How the implementation relates to its surroundings will be the topic of discussion in chapter 3. Here only a short introduction is given, more information can be found in the Meta Environment manual [5].

The Meta Environment is an interactive language development environment that can be used for a number of purposes:

- Development of syntax and semantics descriptions for formal languages
- Development of analysis and transformation techniques for formal languages

The strength of the Meta Environment can be found in the interactive support for the writing of these language specifications, and the automatic tool generation from the specifications. Moreover, both the syntax and semantics of the formal languages can be described in one formalism, namely ASF+SDF.

ASF+SDF stems from the combination of the Syntax Definition Formalism (SDF) and the Algebraic Specification Formalism (ASF). A language specification usually consists of a syntax definition in SDF and a specification of semantics in ASF. ASF uses conditional term rewriting to define semantic behaviour, the syntax of these terms is user-defined with the help of SDF. Thus, ASF+SDF is the tool that enables users to implement concise language definitions as well as language transformation and analysis rules. In addition, these definitions can be modular due to SGLR's support for the full class of context-free languages.

1.2 Problem Statement

Over the years, several of the components that make up the Meta Environment have been migrated to Java. As a central part of this environment, it is important that SGLR too takes this step. Moreover, the current C implementation has gone through too many optimization cycles to be a good candidate for further modification, thereby severely limiting the research that can be pursued in this area. It is therefore imperative that a new implementation, written in Java, is designed with further experimentation in mind. Finally, if this new implementation is to be a substitute for the current C implementation, some effort must be spent in ensuring a good performance.

1.3 Overview

We will start our journey in chapter 2 by reviewing the well-known LR algorithm, serving as a vantage point in our discussion on the GLR algorithm. The SGLR algorithm as described by Visser [8] is briefly discussed, and we will see how the implementation relates to its surroundings, i.e. the Meta Environment and its parse tree and parse table format in chapter 3. All this information is then summarised in chapter 4 in the form of implementation requirements and architecture. Some of the issues encountered are highlighted, and the results in terms of performance and flexibility are presented. In chapter 5 the dynamic combination of LR and GLR as used in Elkhound was implemented and evaluated against our default implementation as a proof experiment. This is meant to be both a test of validity for the framework and a starting point for future experimental work. Finally, conclusions and open issues are discussed in chapter 6.

1.4 Acknowledgements

Chapter 2

Language Technology

Our work is centered around the SGLR algorithm, an extension of the GLR algorithm. Therefore, in this section an in-depth explanation of these algorithms is provided. We will do this incrementally by starting from the well-known LR algorithm and working our way towards GLR and SGLR.

2.1 LR Parsing

One of the best-known efficient parsing techniques for context-free languages is the LR algorithm. This algorithm performs a Left to right scan of the input, and produces a Rightmost derivation, hence the name. The name is sometimes annotated with a number indicating the lookahead symbols used while parsing. Based on the LR algorithm, various improved techniques have been introduced, such as Simple LR, Look-ahead LR, and Canonical LR, each able to deal with more languages than the last. While we will not go into a full length discussion about those, we will describe the working of LR itself as it serves as the basis of the GLR algorithm.

The architecture of an LR parser consists of an input stream, a stack, a parse table holding an action-table and goto-table, and an output stream. These are all centered around the parser core, which contains the parsing logic. This view is perhaps better expressed with the help of figure 2.1. Usually a scanner is used to tokenize the input before delivering it to the parser. This in order to profit from the less complex parts of the language by applying simpler and more efficient parsing techniques, therefore speeding up the parsing process. This approach has some drawbacks, however. These will be discussed later in section 2.3.

The stack contains a history of states the parser has visited, we consider the state on top of the stack to be the current state. The action table contains a number of actions that change our current state, depending upon the current state and

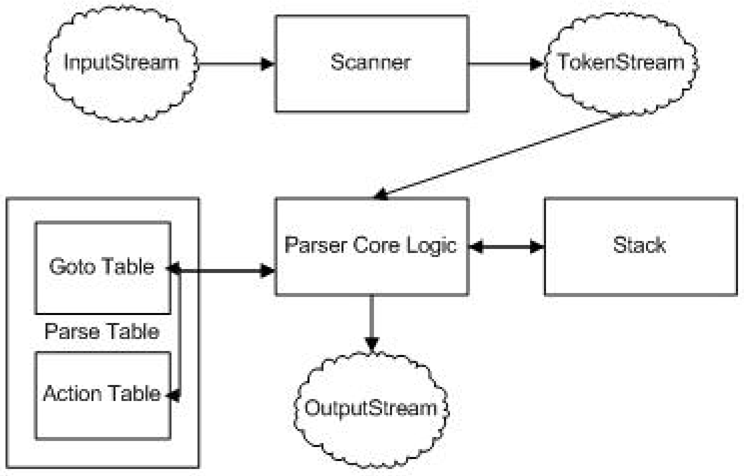

Figure 2.1: LR Parser Architecture

current token. In case of a reduce action, the next state is looked up using the goto table. The actions that can be encountered in the action table are a shift, a reduce, or an accept. If no action is defined for the current state and token, a parse error is omitted.

shift A shift *n* indicates that the next state will be *n*. That is, *n* is pushed onto the stack and the current token is discarded.

reduce A reduce *n* tells us that a reduction with grammar rule *n* should be performed. This means that the stack is popped for each symbol on the right hand side of rule *n* and the next state is looked up in the goto table. This is done by looking at the current token and the state on top of the stack after popping it.

accept An accept means that the current input is recognised and terminates the parser successfully. With the list of grammar rules as given by the reduction actions we have now obtained a rightmost derivation.

When thinking of parsing in terms of automata and their input strings, we see that the algorithm itself is merely a means to interpret the actual automaton, which is encoded in the parse table. Similarly, parse table construction is not part of the LR algorithm, so it will not be discussed here. We will only note that an LR parse table may contain shift/reduce or reduce/reduce conflicts because of ambiguities in the grammar. That is, for a given parser state and token there are more than one possible actions. As the original LR algorithm is deterministic, it cannot cope with this, therefore restricting the number of languages that can be parsed. A number of solutions have been proposed to deal with this problem, such as SLR, LALR and Canonical LR. While parsing ever increasing sets of languages, none of them works on the full set of context-free languages. One of the solutions that does is the Generalized LR approach.

2.2 GLR: The Algorithm

A plausible idea to handle conflicts in a parse table is just to try all the possible derivations in parallel. If one of the derivations is incorrect, we will end our search there and continue with the other ones. Should more than one derivation terminate successfully, we can report the ambiguity by returning the two possible derivations and possibly applying filters to choose the correct one. Running multiple complete parsers in parallel will not be cost-efficient, however. We will need an efficient way of organizing logic and data. The stack as used in the different parallel derivations can be probably be shared for a large part. This line of thought has been encoded in the Graph Structured Stack (GSS).

The GSS is the central data structure during parsing. It contains all the stacks of the parallel running parsers. The indvidual stacks are now being represented using stack nodes, therefore the resulting collection of parallel stacks should not be seen as a collection of replicated stacks, but more as a graph consisting of stack nodes. A stack node is simply an object containing a state and a number of links to other stack nodes. In this way, a stack node defines one level of the stack, and has a pointer to one level lower on the stack. Due to the possible ambiguous character of the grammar, one stack node may contain pointers to more than one stack node, or a stack node might be referred to by more than one stack node. The former case can be thought of as one derivation with multiple possible ’sub’-derivations, the latter as multiple derivations sharing a common ’sub’-derivation. The difference with a normal stack can be observed in figure 2.2. A portrayal of splitting stacks can be seen in figure 2.3.

Structuring our information this way imposes additional conditions on our GSS. For one, while in an LR parser it is obvious what the current state is, this is slightly more complex in the GSS. While conceptually, we have multiple running parallel derivations, practically, our GSS knows only stack nodes and their interconnections. There is no way of telling where the top level of the stack is. For this, we need a list of active stack-nodes. These represent in fact the derivations being tried in parallel at a given point in the parsing process. When processing a

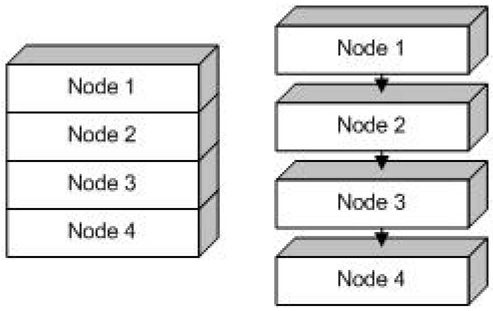

Figure 2.2: Normal stack vs. GSS

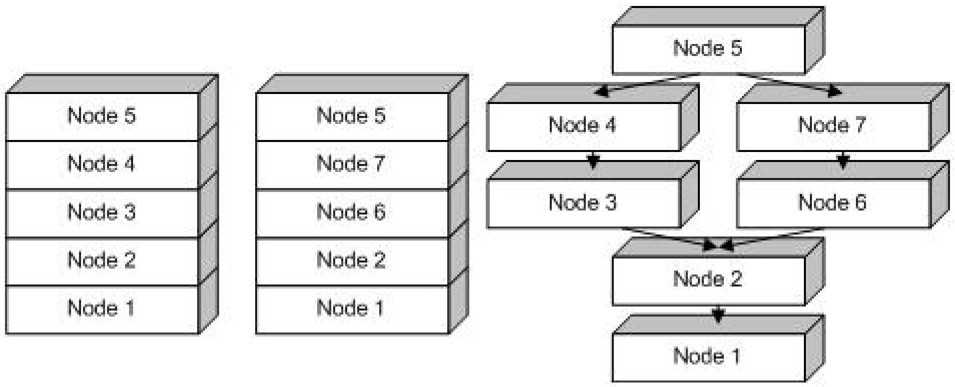

Figure 2.3: Parallel stacks vs. GSS

token, we will iterate over this list so all derivations are tried before moving on to the next token. Because only shifts modify the input stream, we will synchronize the parallel parsers on shift actions. That is, every parallel parser performs all necessary reduce action until a shift action is encountered, in which case it will wait for the other parsers to finish reducing.

Furthermore, parse tree construction is altered by the fact that we need to maintain data for all the possible derivations that are being processed. This makes it impossible to store the recognised non-terminals on the stack itself since a joining stack would need to have multiple different derivations at the same time! A suitable solution would be to attach this information to the links that tie the GSS together, as they denote the different parallel derivations.

Finally, the way in which the stacks are modified differs from the usual stack.

If a new state *s* is pushed onto a stack, a new stack node is created with state *s* and a link to the former top stack node. If however, there is already a stack node with state *s* present, a new link is added to the existing stack node. Popping a stack means we have to follow all links attached to its top stack node, adding the stack nodes found this way to the list of active stacks, while removing the old top stack node.

A more formal description can be extracted from the pseudocode by Rekers [3]. This algorithm was modified by Visser [8] to obtain the SGLR algorithm that we present in section 2.3.1. First however, we will go through a step by step example to illustrate the workings of the Graph Structured Stack.

2.2.1 GSS Example

For this example we use a modified version of the Booleans grammar as comes with the SDF distribution. The full SDF source of this grammar can be found in appendix B.1, and the states visited during parsing are available in appendix C.1 for reference purposes. For sake of simplicity, parse trees have been simplified and layout nodes have been purged from the GSS. The contents of the GSS have been displayed in figure 2.4 at every shift of a character, using ”T&F|T” as input string.

1. The character ”T” has been shifted and recognised. We can see in the SHIFTER procedure in section 2.3.1 that we create a new stack node with a link to the originating stack node (state), this link containing a parse tree equal to the current token.

2. Character ”T” has been recognised as type Boolean and character ”&” has been shifted.

3. Character ”&” has been recognised as literal ”&” and character ”F” has been shifted.

4. Character ”|” has been shifted and the sentence T&F has been recognised as a Boolean. This step serves as a good illustration of how reduction actions are performed. As can be seen in the DO-REDUCTIONS procedure in section 2.3.1, first we enumerate the reduction paths in the GSS. In this case, starting from state 59, there is only one. The trees along the path are collected and then used in the creation of a new stack node and link as is done in REDUCER. Finally, there is only a shift action for state 46 meaning that the other branch in the GSS is discontinued in the next step.

5. Character ”|” has been typed as a literal ”|” and ”T” has been shifted.

6. Finally, the whole sentence T&F|T is reduced to a Boolean which is recognised as a start symbol, thereby succesfully terminating the parse.

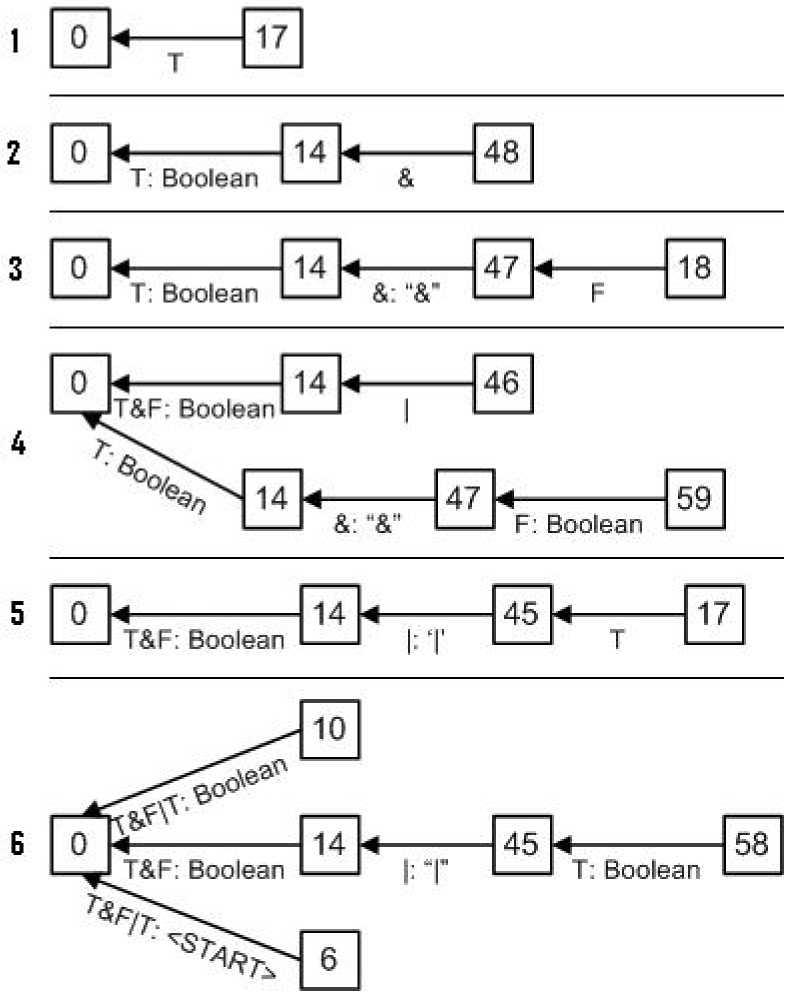

Figure 2.4: GSS Example

2.3 SGLR: GLR Extended

SGLR uses the same algorithm basis as GLR, the only difference is that each character is a token, so that we in fact discard the scanning phase. Hence the name:

Scannerless GLR. Scanners are used to speed up the parsing process by taking advantage of the less complex parts of a grammar. This lexical analysis is often based on regular expression matching using a set of default heuristics, such as prefer keyword or longest match. But most programming languages have an ambiguous lexical syntax, so that special care is needed when developing a grammar that would work with this architecture. This may be solved by a well-defined interaction between scanner and parser, whereby the scanner can use the parser to make decisions in a case of ambiguity. Another possibility is to extend the scanner itself with the functionality needed to make those more complicated decisions. Both approaches complicate scanner design and result in a slower scanning phase. Alternatively, we can integrate the lexical analysis into the context-free analysis. This has a couple of consequences: first, this means every single character becomes a token, and the need for a scanner disappears. Secondly, a number of the lexical ambiguities cease to exist, while a more expressive way of disambiguation may be provided in the language specification formalism to address the others.

2.3.1 The Algorithm

This is the algorithm as taken from Visser [8] and used as a basis for the implementation. We will discuss each of the parts briefly. PARSE takes a parse table and input term and contains the main loop during parsing. The list of *active-stacks* is initialized to contain only the initial state, the succeeding loop iterates over the characters to be processed. If there are no active stacks, the *accepting-stack* variable is checked to see if there is any, an error is omitted otherwise.

```
PARSE(table, file)
    global accepting-stack := ∅
    global active-stacks := {new stack with state init(table)}
    do
        global current-token := get-next-char(file)
        PARSE-CHARACTER()
        SHIFTER()
    while current-token ≠ EOF ∧ active-stacks ≠ ∅
    if accepting-stack contains a link to the initial stack with tree t
    then
        return t
    else
        return parse-error
```

PARSE-CHARACTER handles the list of stacks that are involved in performing reductions, *for-actor*. This list is initialized to the list of active stacks (it is re-initialized for every shift cycle later on in SHIFTER). New stack nodes created in reduction actions are added to this list, or to *for-actor-delayed* if they are rejectable. How this works exactly we will see further on in REDUCER.

```
PARSE-CHARACTER()
    global for-actor := active-stacks
    global for-actor-delayed := $\emptyset$
    global for-shifter := $\emptyset$

    while for-actor $\neq \emptyset \vee$ for-actor-delayed $\neq \emptyset$ do
        if for-actor $= \emptyset$ then
            for-actor := {pop(for-actor-delayed)}
        for each stack st $\in$ for-actor do
            if $\neg$ all links of stack st rejected then
                ACTOR(st)
```

ACTOR obtains the available actions for the state in *st* and the current token. Reduce actions are handled immediately, while shift actions are saved on the *for-shifter* list for later use in SHIFTER. In case of an accept, the current stack node is saved as *accepting-stack*. In case of an error, or there are actions available for the current state and token, we simply proceed with the next stack in PARSE-CHARACTER. This because the parse only needs to be terminated in case all branches of the GSS die.

```
ACTOR(st)
    for each action a $\in$ actions(s, current-token) do
        case a of
            shift(s) $\Rightarrow$ for-shifter := {$\langle$st, s$\rangle$} $\cup$ for-shifter
            reduce($\alpha \to A$) $\Rightarrow$ DO-REDUCTIONS(st, $\alpha \to A$)
            accept $\Rightarrow$ accepting-stack := st
```

DO-REDUCTIONS enumerates all the possible reduction paths for rule $\alpha \to A$ from node *st*, and collects the trees found on the links traveled. These are passed on to REDUCER to handle each individual path.

```
DO-REDUCTIONS(st, $\alpha \to A$)
    paths := the paths from stack st to stack st0 of length $|\alpha|$
    for each path $\in$ paths do
        kids := the trees of the links which form path
        REDUCER(st0, goto(state(st0), $\alpha \to A$), $\alpha \to A$, kids)
```

REDUCER handles a reduction given the reduction path information found in DO-REDUCTIONS. A stack node containing the new state *s* is created if it does not exist already in *active-stacks* (else branch). The new node is linked to node *st0* having as tree the application of $\alpha \rightarrow A$ to *kids*. The new stack node is added to both *active-stacks* and *for-actor*, or in case of a rejectable stack node, *for-actor-delayed*. If there exists a stack node with state *s* in *active-stacks* (if branch), the links of that stack node are checked to see if there is a direct link to *st0*. If there is such a link, an ambiguity has been found and the current tree *t* is added to the ambiguity node of this link. Otherwise the new link is created with tree *t* and added to the stack node. The newly added link might create new reduction paths from earlier processed stack nodes. This triggers a re-examination of all the active stacks in DO-LIMITED-REDUCTIONS. To see how this can happen, have a look at the illustrative example in figure 2.6.

1. First, a reduction action with length 2 is performed on stack node 3, creating a new node with state 6 and a link to stack node 1.

2. Then, on the other branch a reduction action with length 1 is performed, linking the existing stack node with state 2 to the node with state 4, this creates another path for the previous reduction action on stack node 3.

3. In DO-LIMITED-REDUCTIONS that reduction action is perfomed again, leading to the creation of a new stack node 7.

Finally, if $\alpha \rightarrow A$ is a reject production, the links involved will be marked as rejected.

```
REDUCER($st0$, $s$, $\alpha \to A$, $kids$)
    $t$ := application of $\alpha \to A$ to $kids$

    if $\exists$ $st1 \in active\text{-}stacks$ : state($st1$) = $s$
        if $\exists$ a direct link $nl$ from $st1$ to $st0$ then
            add $t$ to the possibilities of the ambiguity node at tree($nl$)
            if $\alpha \to A$ is a reject production then mark link $nl$ as
            rejected
        else
            add a link $nl$ from $st1$ to $st0$ with tree $t$
            if $\alpha \to A$ is a reject production then mark link $nl$ as
            rejected
            for each $st2 \in$ active-stacks such that $\neg$ all links of $st2$ re-
            jected $\wedge$
            $st2 \notin for\text{-}actor$ $\wedge$
            $st2 \notin for\text{-}actor\text{-}delayed$ do
                for each reduce($\alpha \to A$) $\in$ actions(state($st2$), $current$-
                $token$) do
                    DO-LIMITED-REDUCTIONS($st2$, $\alpha \to A$, $nl$)
    else
        $st1$ := new stack with state $s$
        add a link $nl$ from $st1$ to $st0$ with tree $t$
        $active\text{-}stacks$ := $\{st1\} \cup active\text{-}stacks$

        if rejectable(state($st1$)) then
            $for\text{-}actor\text{-}delayed$ := push($st1$, $for\text{-}actor\text{-}delayed$)
        else
            $for\text{-}actor$ := $\{st1\} \cup for\text{-}actor\text{-}delayed$

        if $\alpha \to A$ is a reject production then mark link $nl$ as rejected
```

DO-LIMITED-REDUCTIONS works similar to DO-REDUCTIONS, but only reduction paths going through link *nl* are considered.

```
DO-LIMITED-REDUCTIONS($st$, $\alpha \to A$, $l$)
    $paths$ := the paths from stack $st$ to stack $st0$ of length $|\alpha|$ going
    through link $l$
    for each $path \in paths$ do
        $kids$ := the trees of the links that form $path$
        REDUCER($st0$, goto(state($st0$), $\alpha \to A$), $\alpha \to A$, $kids$)
```

SHIFTER marks the transition to a new character. First, the list of active stacks is re-initialized to the empty list. Then, for every tuple ⟨*s*, *st0*⟩ a new stack node is created with state *s* and a link to node *st0* having as tree the current token. If a stack node with state *s* already exists, the new link is added to this node. This mechanism ensures that stacks that cannot perform a shift will not be alive in the next iteration.

```
SHIFTER()
    active-stacks := $\emptyset$
    t := current-token

    for each $\langle s, st0 \rangle \in$ for-shifter do
        if $\exists\, st1 \in$ active-stacks: state(st1) = s then
            add a link from st1 to st0 with tree t
        else
            st1 := new stack with state s
            add a link from st1 to st0 with tree t
            active-stacks := $\{st1\} \cup$ active-stacks
```

This algorithm is only slightly different from the one by Rekers. The abandoning of the scanner actually has no influence on the algorithm itself, as this simply creates one-character tokens. The most important change is the support for reject productions.

2.3.2 Disambiguation: reject productions

Disambiguation can be applied in one of three phases, namely during parse table construction, during parsing, or after parsing as a filter on a parse forest. The earlier it is applied, the faster the resulting parsing process will be. Post-parsing disambiguation will be implemented in a separate package and is not an issue here. Neither are the disambiguation methods as are implemented during parse table generation. For both we refer to [7]. The one method that remains and does play a role during parsing, involving reject production, is to be accounted for the additional change in algorithm with regard to GLR. Reject productions have been introduced in order to implement keyword reservation. Suppose we have a language definition describing Java, and it contains rules to define lexicals and keywords.

```
[a-zA-Z][a-zA-Z0-9_]* -> LEX
"static" -> KEYWORD
```

In this case the sequence ”static” can be both recognised as being a lexical and a keyword. Because we do not want to allow variables named ”static”, we add an extra restriction.

```
"static" -> LEX {reject}
```

This is the SDF-way of specifying that this sequence may not be recognised as a lexical. The parse table generator treats these production rules as normal productions, which means that we will have extra ”branching” in the GSS during parsing of this sequence. Namely, it will be recognised as a lexical twice, resulting in an ambiguity node. This node can then simply be ignored as being an invalid representation. As it is best that disambiguation is applied as early as possible, this filtering takes place during parsing and tree construction. To understand how that can be done, we merely need to look at how the data is represented in the GSS. An ambiguous parse results in a stack node repeatedly trying to create a link to one other stack node. The existing link will then be updated to hold the ambiguity node containing the newly added tree and the earlier present trees. The desired behaviour can be achieved by marking the link as rejected and refraining from performing actions on stack nodes whose links are all rejected.

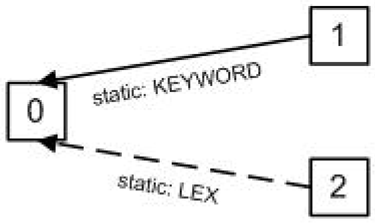

Figure 2.5: Reject Production Example

Figure 2.5 exemplifies this idea by showing the resulting stack nodes when parsing the sequence ”static”. The link from node 1 to 0 represents the literal ”static” being recognised as a KEYWORD, while the link from node 2 to 0 represents both the literal ”static” being recognised as a LEX and the sequence of characters s, t, a, t, i, c being recognised as a LEX. Since the former is the reject production, this link is rejected, and further actions are only performed on stack node 1.

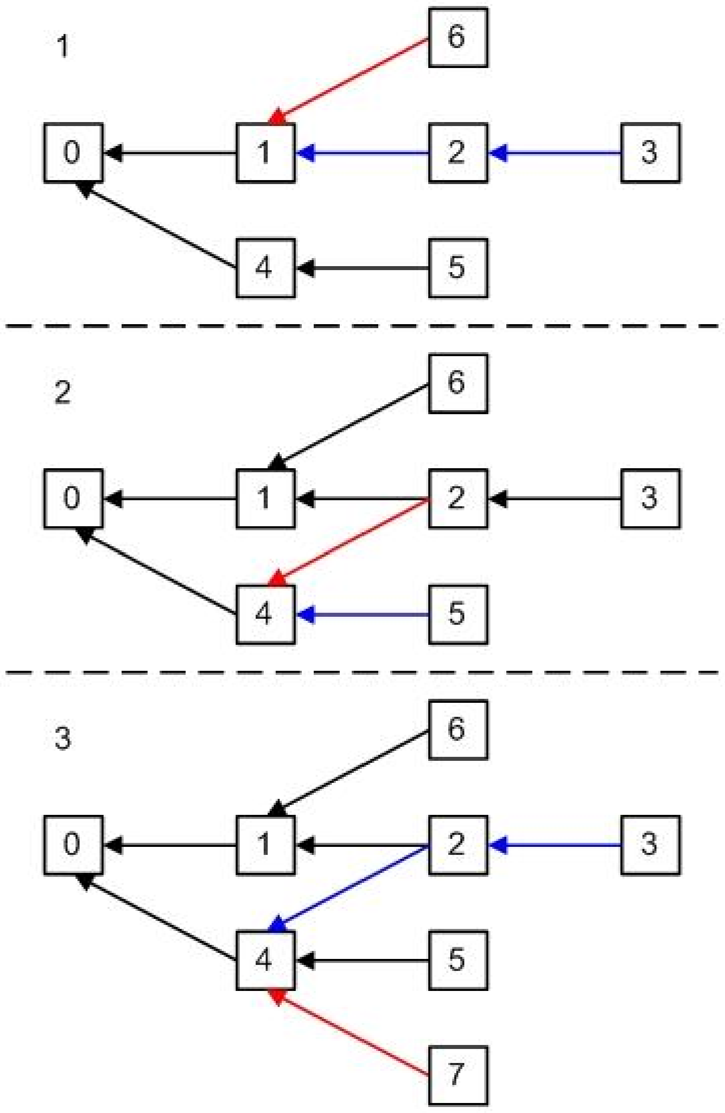

Figure 2.6: New reduction path example

Chapter 3

SGLR in the Meta Environment

SGLR plays an important role in the Meta Environment. Because it interacts with different components of that environment, this interaction is an extra restriction on the implementation. Serialization and transfer of the tree-like structures, such as parse tables and parse trees, are accomplished using the ATerm library. The specific format in which these parse table and parse tree ATerms are created is called AsFix. Additionally, the API code to access these structures was generated using a tool called ApiGen. Since these aspects together make up a large and critical part of our implementation, they have been discussed here separately. The actual implementation of algorithmics is covered in chapter 4.

3.1 Introduction

In the Meta Environment, SGLR is used to parse specifications written in SDF and produce a parse forest. This forest is in turn fed to the parse table generator to obtain a parse table for that specification (that is, the language defined by that specification). The resulting parse table can then be used in conjunction with SGLR to parse or recognise terms in that language. This process is visualised in figure 3.1.

The figure suggests that there are certain formats in which parse tables and parse trees must appear, namely the ATerm structure and AsFix format we mentioned earlier. The interface to these structures has been generated by ApiGen, and to achieve a better understanding of this part of the implementation, we will start with elaborating on the features of the ATerm library.

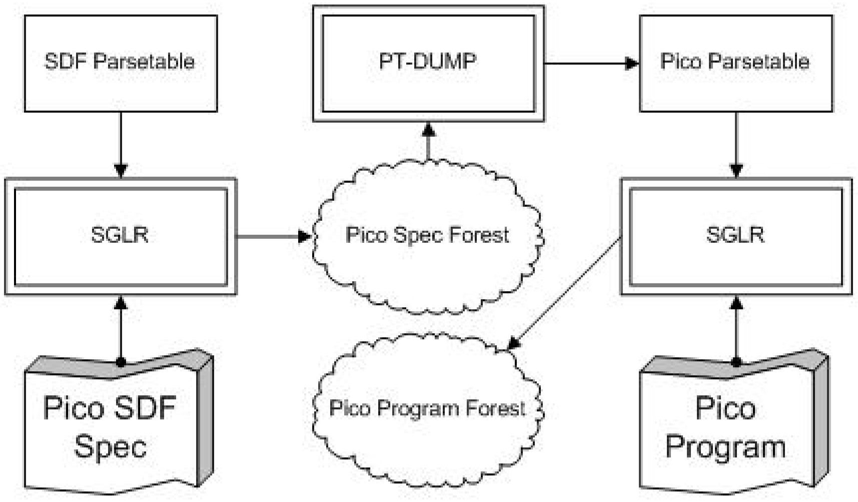

Figure 3.1: SGLR in the Meta Environment

3.2 The ATerm library

The ATerm datatype (short for Annotated Term) is designed to be used for the exchange of tree-like data structures between (distributed) applications. In the Meta Environment this typically includes abstract syntax trees, generated code, and formatted source texts. More specific to SGLR, this also encompasses parse trees and parse tables. Currently, ATerm libraries are available for C and Java. One of the more prominent features of the ATerm implementation is maximal subterm sharing. This has been the means of keeping the ATerm representation as small as possible, and it has raised some issues in our own implementation, which is the reason for highlighting this feature.

3.2.1 Maximal subterm sharing

Maximal subterm sharing ensures that only one instance of any given term is present in memory. This is enforced by only creating terms that do not exist already. Upon creation of a new term, the present set of terms must be checked to see if a similar term already exists, and if so, a shared term is returned. This process is therefore invisible to the user. In order to minimise the penalty that results from the checking, pointers to all the existing terms are stored in a hash table for a quick lookup. This approach has a couple of effects:

- By using the maximal subterm sharing principle, the in-memory term size is reduced significantly, leading to a decrease in execution time.

- The equality check on ATerms is very cheap, since it amounts to a pointer equality check.
- Because sharing is transparent to the user, modifying existing terms can lead to unpredictable behaviour. This is why ATerms are immutable after creation.

Use of the ATerm library has raised some issues in SGLR-Java. For one, the last mentioned property severely penalties ATerm list manipulation. How this has influenced the SGLR-Java implementation can be found in section 4.4.3. Moreover, the Java ATerm library uses the factory design pattern to accomplish maximal subterm sharing. This issue has been elaborated in section 4.4.1, while maximal subterm sharing in Java is described in [6]. For a more intimate discussion of ATerms and their maximal sharing in general, see [4].

3.3 The AsFix tree format

The parse trees that are emitted by SGLR abide by a certain signature, this format is called AsFix. Currently, versions ”2” and ”2ME” are supported, with the latter being the format of choice in the various Meta Environment applications. AsFix2ME is simply AsFix2 with flattened lexicals, lists and layout. The exact transformation from AsFix2 to AsFix2ME as used in our implementation is described in section 4.4.2. The format of both can be found in appendix A.1.

3.4 ApiGen explained

When working with the ATerm library in a given application, there is always a layer of code that incorporates certain structural knowledge of the format of these terms. In SGLR for example, these would be the layers responsible for parse table access and parse tree construction. This approach leaves us vulnerable to any change in parse table or parse tree format.

Since many applications in the Meta Enviroment use the ATerm library, this was the incentive for developing ApiGen, a tool that generates implementations of abstract syntax trees. The result is a strongly typed data structure featuring maximal subterm sharing, that can be used either as a separate tool or as an internal data structure. Using ApiGen involves a number of steps: first, an SDF specification that describes the signature of the tree is written. That specification is then used to collect all the necessary information into an Annotated Data Type (ADT) specification. Using one of the backends provided by ApiGen we can then generate an interface to the tree in either C or Java. An overview of this sequence can be seen in figure 3.2. In SGLR Java, the parse tree and parse table API's have been created in the described manner. The resulting ADT specifications can be found in appendix A. For more information on ApiGen we refer to [1] and [6].

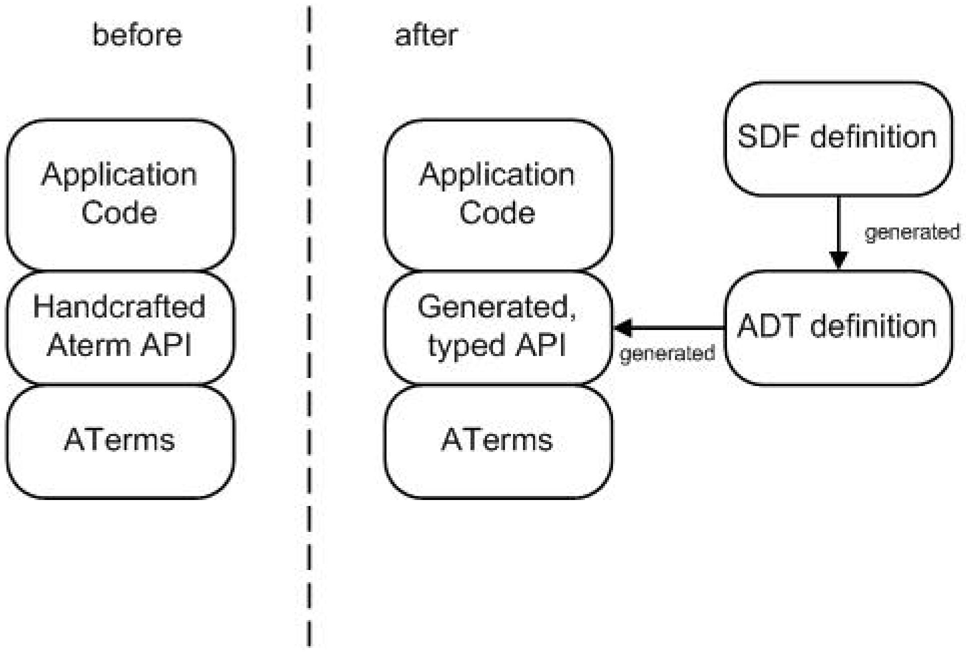

Figure 3.2: ApiGen generation sequence

Chapter 4

The SGLR-Java Implementation

The SGLR algorithm as described in section 2.3 has been implemented in Java. Our discussion on the implementation process will include its requirements in section 4.1 and the way the implementation process was shaped ,in section 4.2. The actual design is then presented in section 4.3, after which we present the issues encountered. Performance results can be found in section 4.5, and a comparison between the old an new implementation is made in section 4.6.

4.1 Requirements

The current implementation of the SGLR algorithm has been written in C, and has been modified over the years to add features and improve performance. The downside of these modifications is a reduction in applicability and maintainability. For example, Unicode is no longer supported in this implementation due to the heavily modified character handling. As maintainability is an issue, extending the current implementation for the sake of experimentation is a tedious process. Therefore, a new implementation must both be correct in the general case and easily extendable. A correct straightforward implementation would satisfy the former, while the latter involves some more concepts. Apart from the aforementioned requirements, this implementation is meant to be a part of the Meta Environment, and should therefore be tailored to interact with different parts of that environment. Summarising, we can state that the implementation must meet the following requirements:

- It should display behaviour similar to the old SGLR implementation in terms of correctness

- It should be easily maintainable and extendable

- It must allow extension from LR to GLR
- It must support reading input terms and parsetables in the ATerm format
- It must be able to omit parsetrees in the AsFix2/AsFix2ME ATerm format

Finally, the implementation process itself influences the design as well. This is the reason behind the third criterion, and it seems therefore a good idea to elaborate the different steps in the process before presenting the actual design.

4.2 Trajectory

Since we want a complete new implementation, a fresh perspective is preferred over the salvaging of old code. Therefore, we refrain from any help the old implementation might offer us and work only with the algorithm. To minimise programming errors and deepen our understanding of the workings of this algorithm, we will model our first steps on the theory inventarisation as described in chapter 2. That is, we will work incrementally and the first prototype to be built will be an LR parser. After validating correctness we can then extend the prototype to full GLR, and test it accordingly. This path culminates in a working SGLR parser, but a very straightforward implementation: the next step will therefore be a series of optimizations. The final step will be to extend the program for a new optimization taken from Elkhound. This part is discussed in chapter 5.

4.3 Architecture

The requirements enumerated in section 4.1 show us that we need to use a modular approach where input, output, the parsetable and parsetree structures and the LR/GLR core logic must be considered separate entities. Moreover, if we want to build an SGLR parser on the foundations of an LR parser their algorithmic code should be separated somehow. This should not be too difficult if we treat the SGLR parser as an LR parser working on a Graph Structured Stack instead of an ordinary stack. Consequently, a good starting point for the architecture would be the schematic view of an LR parser as has been presented earlier in figure 2.1.

Implementing access for a parsetable and create methods for parsetrees can be quite cumbersome. And maintaining that code in case of modifications in the underlying formats may prove to be even more challenging. This is were ApiGen comes in. It creates the necessary access and creation methods from a simple specification describing the format of these structures. To shield the rest of the implementation from changes in these formats, we will build an additional layer with high-level access on top of the generated code. This way we not only limit the maintainance needed in case of format changes, we also prevent the shattering of code responsible for parsetree and parsetable throughout the program.

When we define the supplying of parameters and options the responsibility of a user interface, we have gather sufficient information to define our architecture. The result can be observed in figure 4.1

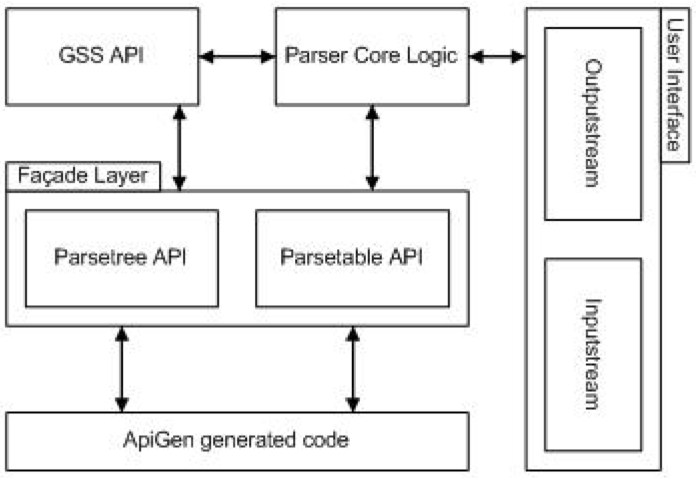

Figure 4.1: SGLR Parser Architecture

4.4 Optimization issues

By following the trajectory as described in section 4.2, we were able to build an LR prototype and extend it to a working SGLR variant. While the trajectory descriptions itself gives a good idea of the implementation history, some issues were raised in the process, and the most interesting of these have been presented in this section.

4.4.1 Heavy Factories

After completing a straightforward implementation and running some initial correctness tests, profiling revealed that the program spent an unusal amount of time initializing `Factory` instances. To understand this problem, we will briefly highlight the Factory design pattern that is used by both ApiGen and the ATerm library before discussing the issue and its solution.

In section 3.2.1 the maximal subterm sharing feature of the ATerm library was briefly explained. The Java ATerm library uses the factory design pattern to enforce this subterm sharing. There is only one class to create and manipulate ATerm instances, this way the set of existing ATerm instances can be checked upon creation of a new ATerm. This class is called `PureFactory`, and it is used by the code generated by ApiGen to create and manipulate the ATerms underlying the generated data structure. The structural information regarding these ATerms is captured in the `Factory` class, that is responsible for creation and manipulation of the objects implemented by the generated code. This `Factory` class in turn uses a `PureFactory` for the manipulation of the underlying ATerms.

Because of the maximal subterm sharing feature, a program that uses the ATerm library can have at most one instance of a `PureFactory` at a time. As ATerm equality is defined in terms of a pointer equation, comparing two ATerms created by different `PureFactory` instances leads to unexpected results. However, we can still create multiple `Factory` instances using the same `PureFactory` and have the expected behaviour with regard to ATerms.

On top of the parse tree and parse table API generated by ApiGen, a high-level API was defined. This was done in order to make the implementation more robust to changes in generated code, to provide high-level access, and to generally prevent the shattering of code responsible for parse tree manipulation and parse table access. Initially this was accomplished using one `PureFactory` instance and multiple `Factory` instances. Because the `Factory` class incorporates all the structural information needed to create the ATerms that represent the higher-level data structures, it initialises a signifant number of these upon creation. For example, creation of the tree application pattern looks like this:

```
pattern_Tree_Appl = factory.parse("appl(<term>,<term>)");
fun_Tree_Appl = factory.makeAFun("_Tree_appl", 2, false);
proto_Tree_Appl = new sglr.ptable.types.tree.Appl(this);
```

Some 100 other pattern ATerms and their respective objects are created as well in every instance of the `Factory` class. It makes sense therefore, to limit the number of instances as instantiation can be quite costly. Adapting the high-level API to have only one instance of the `Factory` class thus lead to a major improvement in performance. For example, parsing a 508 bytes Pico program took 23 seconds before, and approximately 1 second after the modification.

4.4.2 Building AsFix2ME parsetrees

After dealing with the first issue, the implementation could parse any context-free language correctly, and construct the corresponding parsetree in AsFix2 format.

The AsFix2 format embeds a lot of information that is not used frequently, and it is therefore not the format of choice in the Meta Environment. When reading the ADT parsetree specification in appendix A.1, one notices that there are multiple ways of expressing a similar tree construct. This is because the specification defines two parsetree formats; a small set of modifications defined on a parsetree in AsFix2 lead to a more compact parsetree in the AsFix2ME format. These modifications are applied traversing the original parstree top-down and have been visualised in figure 4.2, where the ”iterations” case is similar to the ”lexicals” case. They can be summarised as follows:

literals The old-style literal node is an application node with the corresponding characters as its children. This is replaced by one simple literal leaf (that is, having no children).

layout Layout in an AsFix2 parsetree is represented quite intricately encoding the precise structure of the whitespace. When encountering an ’optional layout’ node denoting a whitespace subtree, the subtree is traversed, collecting its character leafs, and the layout node anew is created anew using the collected characters as its direct children. This way the whitespace subtree is flattened to one with depth 1.

lexicals Similar to the whitespace, the subtree found under a lex - cf injection is flattened. This time a lexical list-node is created with the characters as its children. This can be seen as a specific case of the general iterations transformation defined below.

iterations Lists are defined recursively causing a high number of list nestings in the resulting AsFix2 parsetree. These are flattened by traversing a list subtree and collecting the first non-list nodes encountered. The node is then created anew using a new-style list node and the collected subtrees as its direct children.

The resulting parsetree is significantly smaller and will thus result in faster interchange between applications. However, the process of first building a parsetree in one format, then traversing it completely to build yet another, is not very efficient. It might be interesting to see whether the AsFix2ME parsetree can be built directly. The main problem is, that the current filter is defined top-down while the parser constructs the parsetree bottom-up. That means that the filter must be redefined in terms of a bottom-up traversal, and though there are some dificulties, this can be accomplished relatively straightforward if we have a good understanding of the workings of the original top-down filter. So we will reformulate the filter per section:

literals This basically remains the same operation. The moment a literal production is encountered, a new-style literal leaf is created instead of an old-style application node.

layout This is slightly more complex, as it forces us to postpone tree construct production. If a layout production is encountered, the current children are collected and stored for later use. If a `cf(opt(layout))` production is encountered (see figure 4.2) the stored children are collected to be used to create the node as usual.

lexicals Again similar to the layout production, the children are ”forwarded” until a lex `-` cf injection production is encountered, in which case the node is created as usual.

iterations There were two possible courses in this case. One, the children could be ”forwarded” until a non-list node is encountered, saving the list production as it would be needed to create the final list node. This would mean that all the children of every non-list node to be created need to be checked to see if one of them is a list node ”pending creation”. Moreover, it would mean the need for yet another piece of data to be stored explicitly. That is why the second solution was chosen: the list node is created as usual, and there is a check to see if there are any list children. If there are, their children are added to the list of children for the current node, removing the list child itself. This means the child nodes need to be checked only in case of a list node.

4.4.3 List optimizations

The last issue to be elaborated evolves around the most widely-used data structure in SGLR-Java: the list. Manipulation of ATerm lists proved to be quite expensive due to the way they are constructed.

ATerm lists are implemented as (head, tail) constructs. That is, a list `[1,2,3]` is represented as `[1,[2,[3]]]`. This has everything to do with the fact that, due to the maximal subterm sharing feature, ATerms are immutable after creation. For example, take the list *l1* defined as `[1,2,3]` and the element *e* defined as `4`. If *e* needs to be inserted at the beginning of *l1*, this can now be easily accomplished by creating a new list with *e* as its head and *l1* as its tail. If, on the other hand, *e* needs to be appended to *l1*, it needs to be created anew. Recall that the last element of *l1* is in fact a list with one element, and this list is immutable. As this is valid for all the ”sublists” in *l1*, e.g. `[1,[2,[3]]]`, `[2,[3]]` and `[3]`, these all need to be created anew. The process has been implemented by creating a new list `[4]` and repeatedly inserting the elements fetched from *l1*. Summarising, this has a number of effects on the list operations:

insert This operation is of constant complexity, and forms the basis of a number of the other important list manipulation operations.

append This operation is defined in terms of `insert` operations, and the complexity is therefore linear in the number of elements in the list the element is appended to.

concat Similar to `append`, this operation is defined in terms of `insert` and is linear in the number of elements of the first list.

It seems therefore wise to use more efficient data structures wherever possible. Especially in the methods responsible for parse tree construction, these ATerm lists were used intensively, even for temporary data. The use of these lists was eliminated wherever possible, and a more efficient use of the `insert` operation provided a good solution for the other cases. The result was a general parse time decrease of approximately 50%.

4.5 Results

In the evaluation of SGLR-Java, three important issues were considered: correctness, performance, and source code properties. The last issue is discussed in section 4.6, in this section we will deal with the former two. Correctness was enforced by using the old SGLR regression tests and comparing output and expected output. Parsetrees produced by the Java implementation are equal to the old version modulo ambiguities, for which many filters have not been implemented yet. The other issue was performance. What is the time performance behaviour of the new implementation compared to the old one? To test this, two sets of real-world programs, in C and Java, were prepared and fed to both implementations. These tests have been performed on an Intel Pentium 4 2.8GHz with 512MB memory running Linux kernel 2.6.9. The time characteristics of both runs have been plotted in figures 4.3 and 4.4. To minimize the influence of the JVM startup time on SGLR-Java's performance the average parsing time over 50 parses in one run was used to represent the parsing time for a specific input. The influence of the JVM startup time is discussed a little later one in this section. Additionally, figures 4.5 and 4.6 show the performance of both implementations on the *EFa* and *EEb* grammars These grammars were used for evaluating the hybrid technique discussed in chapter 5.

It can be observed that, apart for a large constant overhead, the time complexity for both implementations is roughly equivalent, though it appears as if some more performance improvements should be made in the parse tree construction process. How much of that overhead is due to the Java interpreter remains an interesting experiment. Attempts to compile the code to machine code using GCJ have been unsuccessful because of the lacking support for the latest Java API version.

In the Meta Environment SGLR is used as a continuously running process waiting for input terms and their associated parse tables. This means that startup time is eliminated, and some data can be cached, such as the parse tables. While startup time might not be a major factor in the old implementation, startup of the Java Virtual Machine might have a significant impact on the performance of SGLR-Java, especially on smaller inputs. Gaining a better understanding of this factor will therefore give us a better prediction of the performance when used in the Meta Environment.

By simulating the continuous run and feeding ever-increasing numbers of the same input the startup overhead was approximated. The inputs were parsed 1 time, then 5, 10, 20 and 50 times in one run. The input terms used were those of the C and Java test batch mentioned before. The results can be seen in figures 4.7 and 4.8, where the average parse time per input is displayed for a number of inputs (sizes given in bytes). These figures show that startup time indeed has a significant impact on performance, especially for small inputs, but even for larger ones.

4.6 SGLR and SGLR-Java compared

Though performance is an important criterion, simplicity in the source code is perhaps even more important. After all, the incentive was a need for experimenting. This section is therefore dedicated to evaluating this aspect of the implementation. A good starting point might be to look at the number of lines of code. Stripping both SGLR and SGLR-Java from excess whitespace and comments, the former amounts to a total of approximately 5600 lines while the latter is limited to slightly more than 1600 lines. The last figure does not include the API's generated by ApiGen, as this part is not meant to be maintained by hand anyway. However, SGLR implements a number of postparsing filters that SGLR-Java lacks. Close to 950 lines were found to be associated with that functionality. That still leaves the balance in favour of SGLR-Java. Part of this can be attributed to the fact that SGLR uses its own memory management system, while SGLR-Java leaves that up to the JRE garbage collector. Another issue is that SGLR needs to have its parse table and parse tree access defined up to the term signature level. To illustrate this, two sets of methods from old and new implementation representing the same functionality were compared. In particular, the following improvements can be noted:

- A number of small, functional methods replaces the two large functions.

This increases both readabilty and maintainability.

- The object hierarchy automatically solves the need for a large switch.
- No structural ATerm information is present in the new set of methods, this makes the code more robust to changes and also provides better typing of the different entities.

The example considers the lookahead functionality in both implementations, starting with the old one:

```
for(; permitted && !ATisEmpty(las); las = ATgetNext(las)) {
    ATermList cc;
    lookahead morelooks;
    ATerm la = ATgetFirst(las);

    if(ATmatch(la,
    "look(char-class(<term>),[<list>])",
    &cc,
    &morelooks)) {
        if(SG_CharInCharClass(c, cc)) {
            permitted = ATisEmpty(morelooks)
            ? ATfalse
            : SG_CheckLookAhead(morelooks);
        }
    }
}
```

The `SG_CharInCharClass` function is found elsewhere in file `parser.c`. Apart from being more reading material, this function is part of one very large file instead of defined as a method on a lookahead object. This leaves the developer desiring insight into the main `parser.c` file with circa 870 lines to digest.

```
ATbool SG_CharInCharClass(int c, register ATermList cc)
{
    ATerm ccitem;

    for(; !ATisEmpty(cc); cc = ATgetNext(cc)) {
        ccitem = ATgetFirst(cc);

        switch(ATgetType(ccitem)) {
            case AT_INT:
                if(c == ATgetInt((ATermInt) ccitem))
                    return ATtrue;
                break;
            case AT_APPL:
                if(c >= ATgetInt((ATermInt) ATgetArgument(
                    (ATermAppl) ccitem,
                    0))
                && c <= ATgetInt((ATermInt) ATgetArgument(
                    (ATermAppl) ccitem,
                    1)))
                    return ATtrue;
                break;
            case AT_LIST:
            {
                register ATermList l = (ATermList) ccitem;
                assert(0 &&
                "Hypothesis: This should never happen\n");
                for(; !ATisEmpty(l); l = ATgetNext(l))
                    if(c == ATgetInt((ATermInt) ATgetFirst(
                        (ATermList) l)))
                        return ATtrue;
            }
            break;
        }
    }
    return ATfalse;
}
```

The LookAhead object, part of this example, consists of a list of CharRange objects. These in turn can be either a single number, i.e. `3` or a list of two numbers denoting a range, such as `[38,76]`. ApiGen created the complete classes necessary for manipulation, and on top of that a high-level API was defined. Thanks to ApiGen, this does not encode the level of structural detail present in the old implementation.

```
    for (index = 0; index < las.size(); index++) {
        number = read();
        if (number == CharacterIterator.DONE ||
        !las.inRange(index, number)) {
            index++;
            permitted = true;
            break;
        }
    }
```

Where the `inRange` method on the LookAhead object works on a list of CharRange objects:

```
    public boolean inRange(int number) {
        for (int i = 0; i < size(); i++) {
            if (fetch(i).inRange(number)) {
                return true;
            }
        }

        return false;
    }
```

The CharRange class has two subclasses, a Character class for the ”single” variant, and a Range class. This substitutes the switch as used in the `SG_CharInCharClass` function by a clean object hierarchy. The `inRange` method of the Character object:

```
    public boolean inRange(int number) {
        return single == number;
    }
```

The `inRange` method of the Range object:

```
    public boolean inRange(int number) {
        return (from <= number && number <= to);
    }
```

Similar comparisons can be made at different locations throughout the source code. These examples, together with an indication of the number of lines of source code, serve to illustrate that SGLR-Java is indeed easier to comprehend and therefore a better maintainable solution. In chapter 5 this statement is further strengthened by giving insight into the actual implementation of an experiment using SGLR-Java.

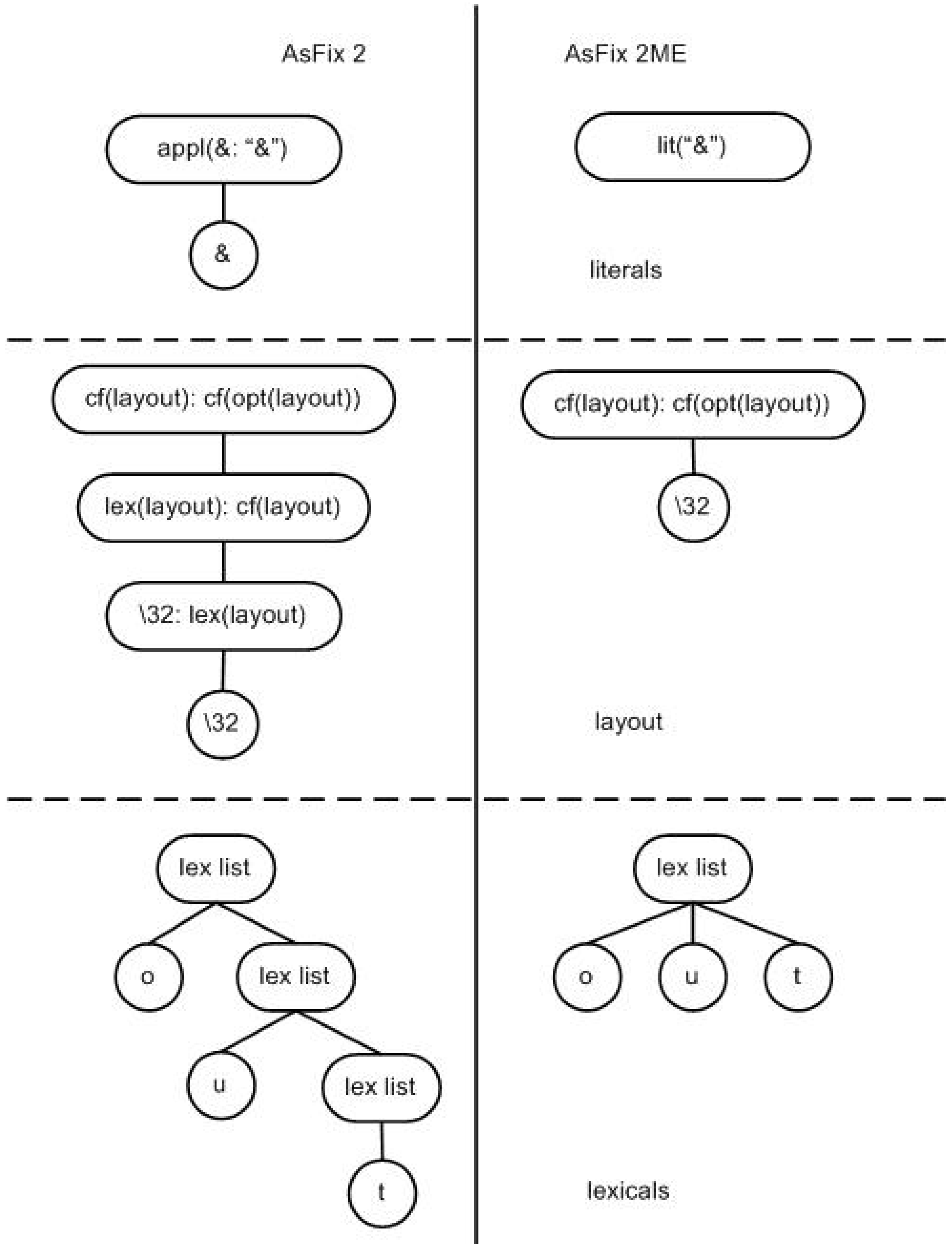

Figure 4.2: AsFix2 and AsFix2ME

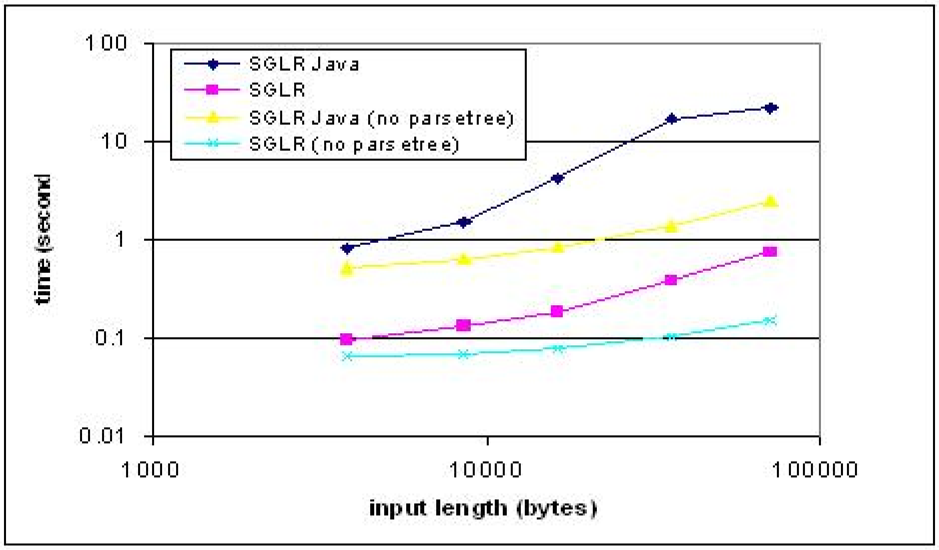

Figure 4.3: Parsing performance in C

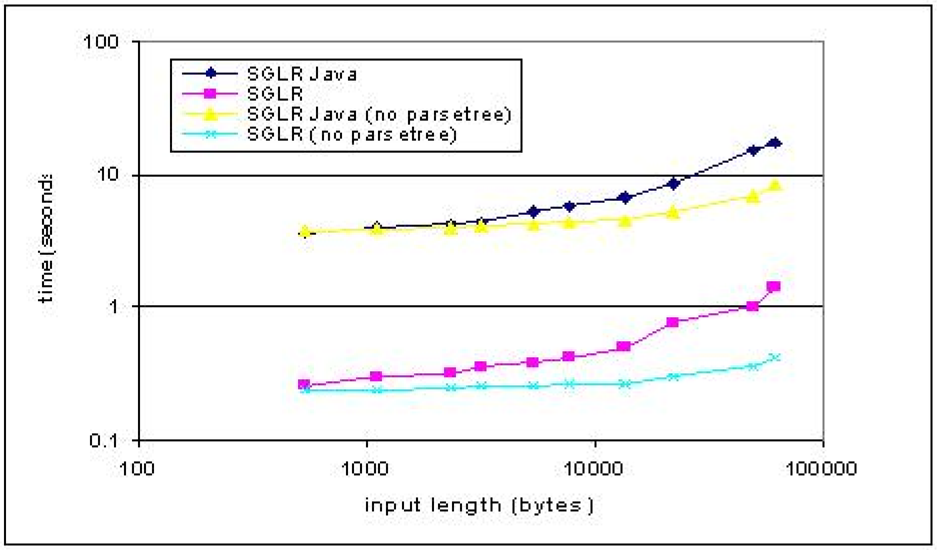

Figure 4.4: Parsing performance in Java

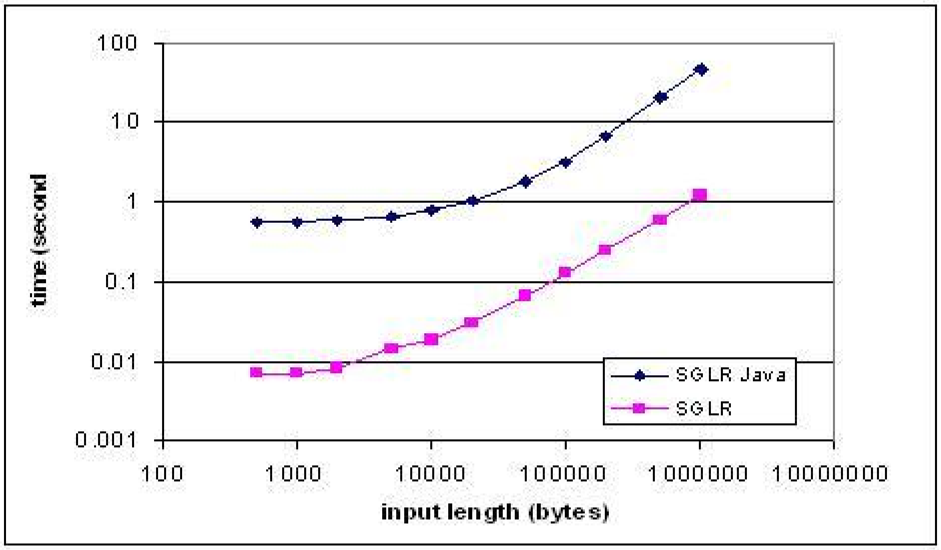

Figure 4.5: Recognise performance on EFa grammar

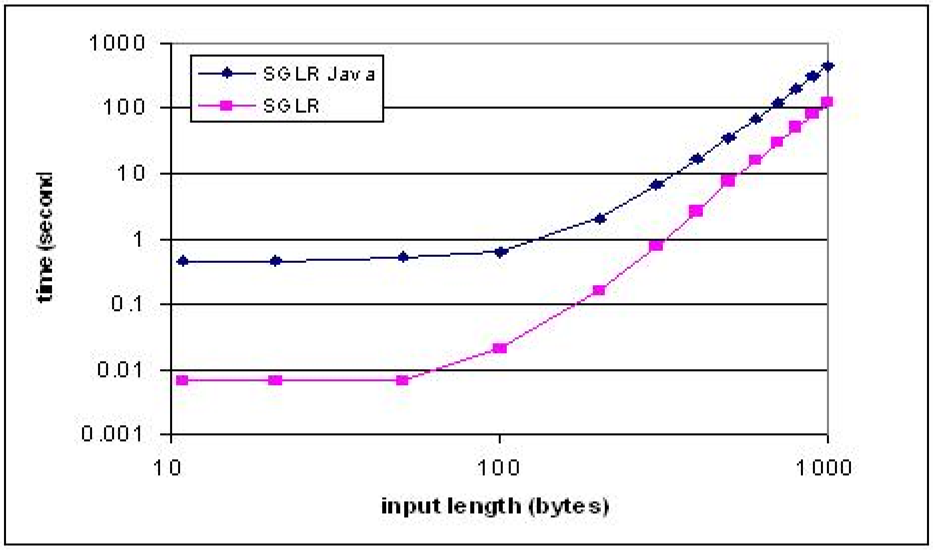

Figure 4.6: Recognise performance on EEb grammar

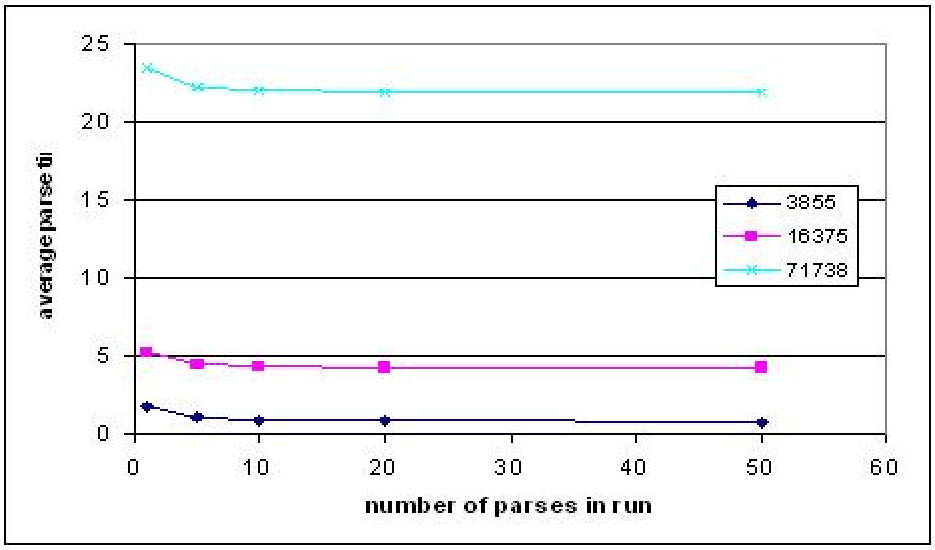

Figure 4.7: Impact of JVM startup time on performance (C testbatch)

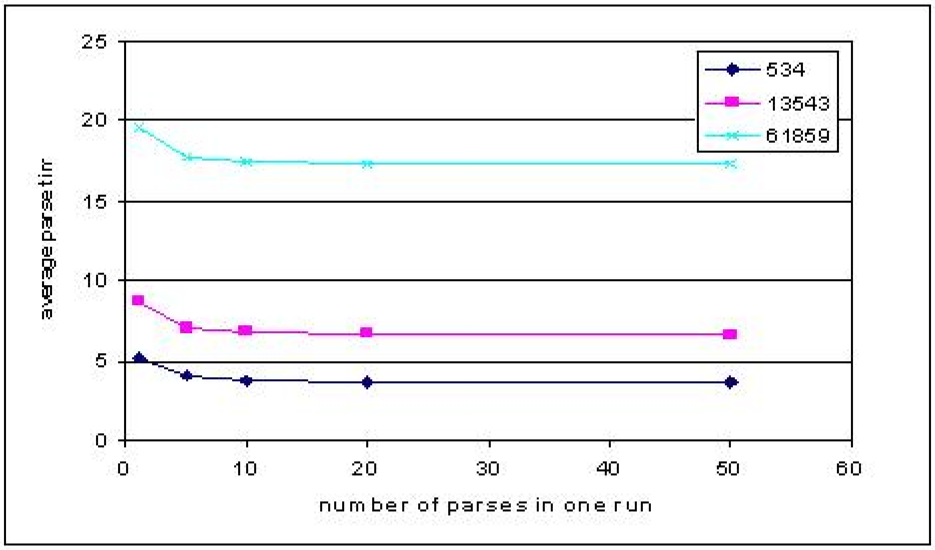

Figure 4.8: Impact of JVM startup time on performance (Java testbatch)

Chapter 5

Case study: Elkhound

Elkhound is a parser generator based on the GLR parsing algorithm. It features a technique that allows it to switch between GLR and LR, thereby allowing it to take advantage of the deterministic parts of the grammar. We studied this technique and applied it to our SGLR implementation.

5.1 The Hybrid Technique

In parsing a term using GLR, a significant part of the execution time is spent performing reduce actions. While the reduce action in an LR implementation consist merely of stack operations, in GLR we need to traverse the GSS and generally deal with a lot of administrative overhead. It is therefore, that an increase in performance may be gained by succesfully applying an LR-type reduce wherever possible. To be able to make this choice, Elkhound introduces the notion of *deterministic depth*. The deterministic depth of a stack node is defined to be the number of nodes it can visit before encountering a node with out-degree greater than one. The deterministic depth of the bottom node is defined to be 1. This notion can now be used when performing reductions in predicting the number of reduction paths. That is, if the reduction length is smaller than the deterministic depth only one reduction path is possible and therefore, an LR reduction can be used. How this is exploited in Elkhound can be found in [2], we will only discuss its use in SGLR.

5.2 Applied to SGLR

The benefit of the hybrid technique heavily depends upon the number of reductions that can actually be performed as an LR reduction. Though this seems to be working perfectly well in Elkhound's case, the question remains whether this property will remain untouched in case of a scannerless implementation. In order to find out, we extended our implementation with support for the deterministic depth administration and measured the number of reductions that could have been performed as an LR reduction.

Extending the implementation formed the first real test of our framework in terms of extensibility. Adding this extra administration turned out to be very straightforward. First, we created a `Log` class that would be responsible for containing, manipulating and outputting the various statistics we wished to obtain. It consists of a default output stream, a string to hold the messages to print to that output stream, and various fields to hold the statistics while they are gathered. Some simple access and manipulation methods were included, total lines of code (including comments and whitespace) numbering around 60. The only modifications necessary in the existing code would be some calls to the `Log` access methods. In our DO-REDUCTIONS we have all the information necessary and it was there that we added the following lines:

```
if (act.getLength() == 0) {
    (..)
    statistics.appendReduction(0, true);
}
else {
    (..)
    statistics.appendReduction(
        act.getLength(), trav.size() == 1);
}
```

where `appendReduction` is:

```
public void appendReduction(int length, boolean simple)
```

In our DO-REDUCTIONS we distinguished between 0-length reductions and other reductions because the former do not need to enumerate reduction paths. This is where the first line is concerned: 0-length reductions are always ”simple” since they never involve reduction paths over branches. The second line is the general case: here is `act.getLength()` the length of the reduce action and `trav.size()` the number of reduction paths found. If there is only one reduction path, we can label this reduction as ”simple” LR reduction.

This straightforward extension testifies that indeed the implementation is well-designed. Moreover, the resulting statistics were promising, as can be observed in figure 5.1. Using ever increasing inputs, the total number of reductions were measured and plotted together with the percentage of reductions that could be labeled

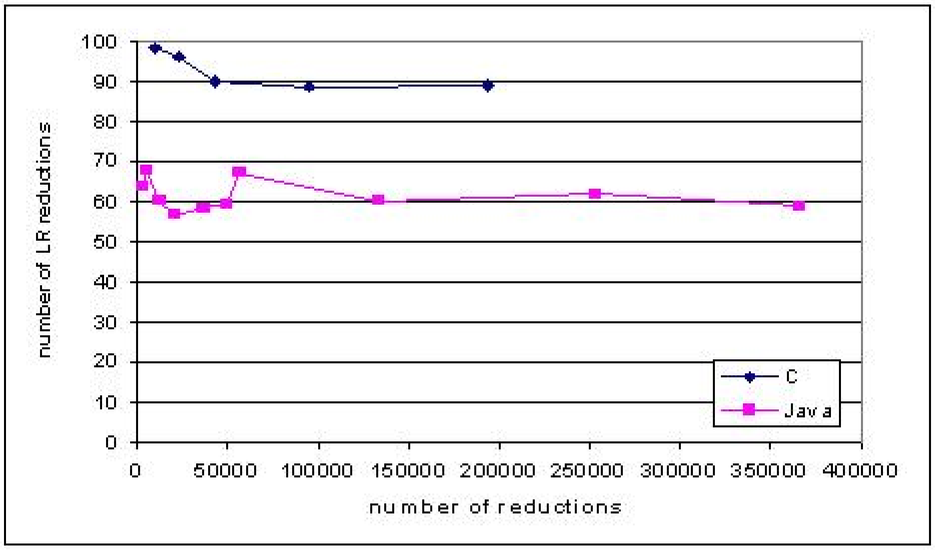

Figure 5.1: Percentage LR reductions

as ”simple” LR reductions. It turned out that in C almost 90%, and in Java approximately 60% of the reductions were suitable. Strengthened by these results, we set out to exploit the hybrid technique. Not all the optimizations as used in Elkhound were valid in SGLR-Java, so we have pinpointed only two issues:

Simplified traversal Since we know there is only one reduction path, we do not need to perform a breadth-first traversal of the GSS if we embed a reference to the next stack node into the stack node object itself. This will eliminate a significant amount of overhead associated with the enumeration of the reduction paths.

Stacknode deallocation If there is only one reduction path, the nodes encountered during traversal can be deallocated immediately, until a ’split’-node is encountered, i.e. a node with in-degree greater than 1. This would require an extra administration of reference counts for each stack node. As we use the Java garbage collector for our memory management, this feature has not been implemented yet.

These two optimizations both depend on the reduction length for their efficiency. As can be seen in figure 5.2, the average reduction length is approximately 1.4 and 1.2 for our C and Java test sets. It can be expected that the benefit of these two optimizations will therefore be limited in these cases.

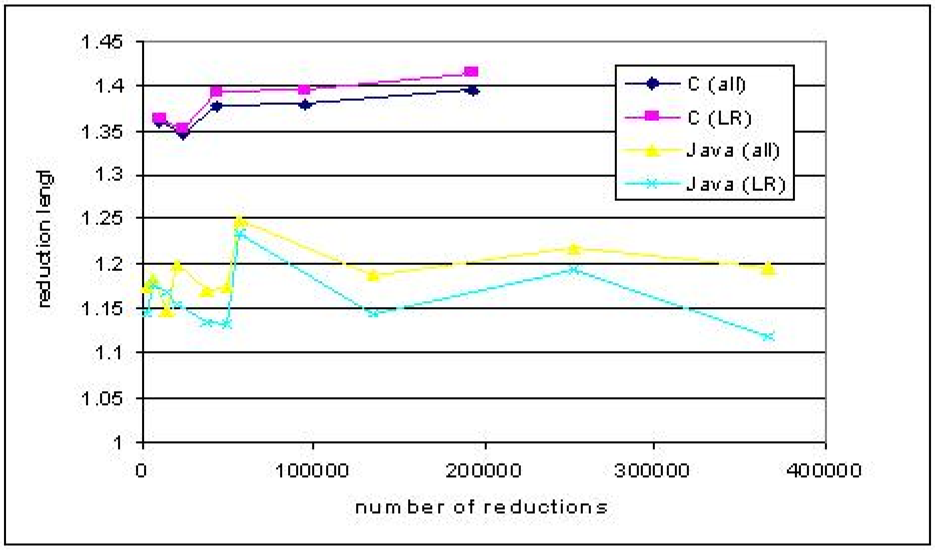

Figure 5.2: Average reduction length

To test the actual value of these optimizations, we implemented the simplified traversal. In order to know when this should be applied, we extended the stack node objects with support for the deterministic depth administration. This entailed an extra field, initialized to 1 in new objects, and an update mechanism in case a link is added to the stack node. If the stack node already contains a link to another stack node, the deterministic depth is reset to 0 (else branch). If there is no link attached, the deterministic depth is updated to the linked stack's depth + 1. Moreover, the linked stack node and tree are embedded into the stack node object itself to be able to take advantage of the simplified traversal.

```
if (links.size() == 0) {
    linkedStack = link.getLinkedStack();
    linkedTree = link.getTree();
    linkRejected = link.isRejected();
    deterministicDepth =
        linkedStack.getDeterministicDepth() + 1;
}
else {
    deterministicDepth = 0;
}
```

This administration, in combination with the embedding of the linked stack node and tree into the stack node object itself, allows us to extend the DO-REDUCTIONS method. We add an extra branch that checks if the current reduction action is an LR reduction, and if so, uses the simplified traversal in the form of one tight loop. The advantage here is not having to iterate over several lists and objects the way happens in the usual enumeration. Finally, the REDUCER routine is called in the usual manner with the obtained reduction path.

```
if (hybrid &&
st.getDeterministicDepth() >= act.getLength()) {
    ParseStack current = st;
    PTreeList trees = new PTreeList();
    for (int i = 0; i < act.getLength(); i++) {
        trees.addFirst(current.getLinkedTree());
        current = current.getLinkedStack();
    }

    reducer(
        current,
        parseTable.getNextStateNumber(
            current.peek(),
            act.getLabel()),
        parseTable.getLabelByNumber(act.getLabel()),
        trees
    );
    statistics.appendReduction(act.getLength(), true);
}
```

As can be seen in section 4.5, the performance bottleneck in our implementation is building parsetrees. And this to such an extent, that any algorithmic optimizations will have a very limited effect on overall parsing performance. Therefore, a more interesting experiment will be to see what the impact of this technique is on the performance of the algorithm itself, i.e. recognizing an input term without building a parsetree.

5.3 Results

For testing purposes, we took the two grammars as were used in testing Elkhound [2], the *EFa* and *EEB* grammars. *EFa* is defined as:

$E \rightarrow E + F \mid F \; F \rightarrow \mathrm{a} \mid (E)$

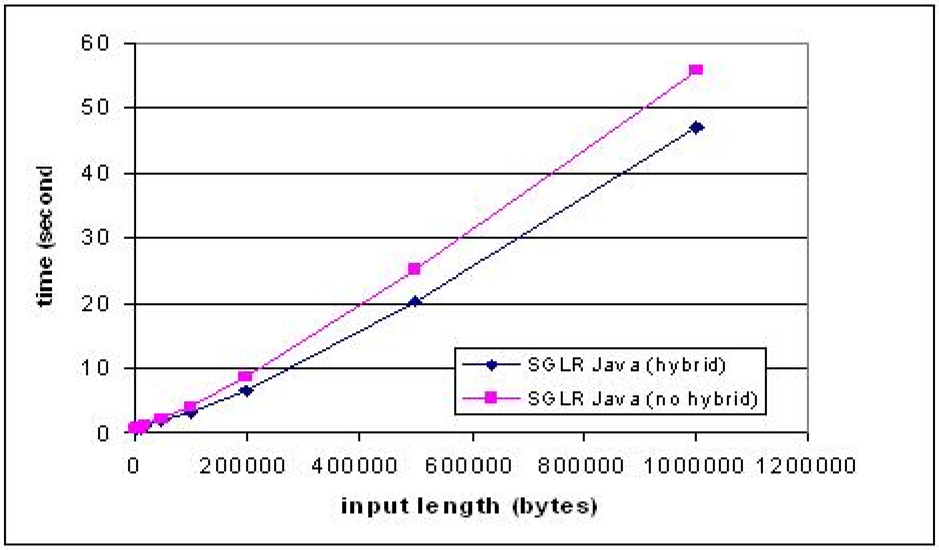

Figure 5.3: Recognize performance on EFa grammar

This grammar has been used in a comparison test with an LALR(1) implementation, and as such it is not ambiguous. This results in a high number of possible LR reductions; a small decrease in constant time overhead can therefore be observed in figure 5.3. The *EEB* grammar is defined as:

$$E \rightarrow E + E \mid \mathrm{b}$$

This grammar is highly ambiguous, and has little LR reductions as a result. It is therefore no surprise that performance with and without the hybrid technique does not differ. Though the conceptual difference between LR and GLR reductions can be exploited further, its usefulness will be limited because reduction lengths appear to be small on average.

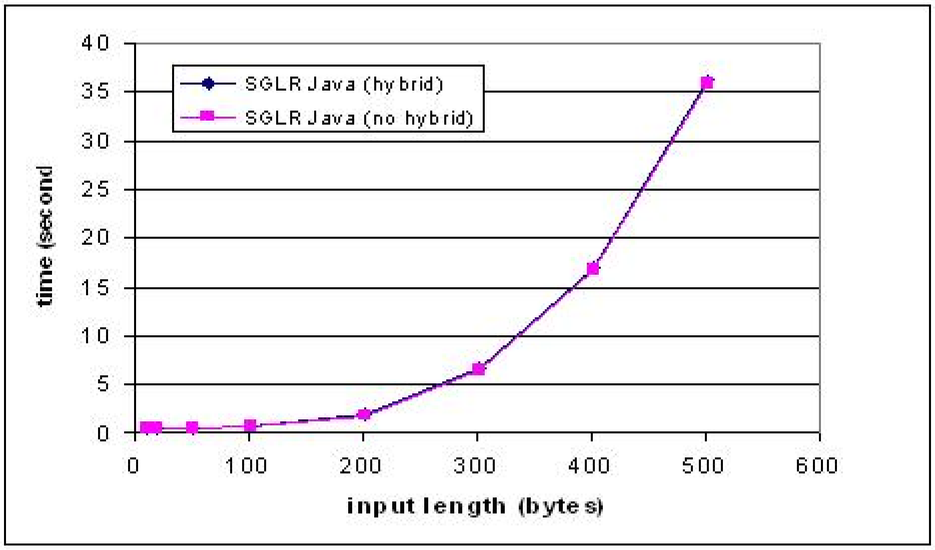

Figure 5.4: Recognize performance on EEb grammar

Chapter 6

Conclusions

This chapter represents the final step in our traject and will consist of summarizing the problems we meant to solve, the solutions found and the issues open to further exploration.

6.1 Problems and solutions

Throughout this thesis, we have addressed the problems as summed up in section 1.2. The solutions found include:

Experimentability The code produced has been kept clean by rigidly following the guidelines provided by the algorithm, and ensuring a robust implementation by a modular approach. The correctness of this approach has been established by the effortless implementation of an experiment involving the LR/GLR hybrid technique.

Performance Time and effort have been spent in a number of optimisation cycles, ensuring a time complexity behaviour similar to the old SGLR implementation. However, since there remains a constant factor 10 difference, additional action is needed to make the Java implementation a worthy substitute for the old C implementation.

Moreover, some issues were encountered in the process that have been addressed as well:

AsFix2ME While outputting both formats. the old SGLR implementation constructs an AsFix2ME tree by first constructing an intermediate structure, then an AsFix2 tree, and transforming that tree to one in AsFix2ME format. By redefining the transformation step, SGLR Java supports a direct construction of both AsFix2 and AsFix2ME parse trees, skipping the intermediate steps.

LR/GLR Hybrid Technique Apart from being a proof of concept, the experiment yielded some results regarding the applicability of this technique in SGLR. Measurements taken indicated that in our test batch there was an abundance of reduction actions that qualified for this technique, but that the potential benefits in the optimisations applied was limited due to a small average reduction length.

6.2 Future Work

As can be seen in the summation of addressed problems, some of the issues are open-ended and need additional thought.

Performance There are some possibilities to improve SGLR-Java's performance. One might be to use a Java machine code compiler, losing the overhead associated with the Java interpreter. Another might be to use explicit memory management, reducing memory usage and increasing locality. Finally, algorithmic enhancements will lead to performance improvement. One of these could be to sort the possible reduction paths in order to be able to skip the expensive DO-LIMITED-REDUCTIONS step.

LR/GLR Hybrid Technique The distinction between LR and GLR reductions can be exploited beyond the experiment conducted in chapter 5. Specifically, explicit memory management will support the improved stack node deallocation technique that has not been implemented in the experiment.

Bibliography

[1] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59:35–61, 2004.

[2] S. McPeak and G. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Compiler Construction*, pages 73–88, 2004.

[3] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, Universiteit van Amsterdam, 1992.

[4] M.G.J van den Brand, H.A de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software – Practice & Experience*, 30:259–291, 2000.

[5] M.G.J van den Brand and P. Klint. *ASF+SDF Meta-environment User Manual*. Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

[6] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, 2005. to appear.

[7] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction*, volume 2304, pages 143–158, 2002.

[8] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Universiteit van Amsterdam, 1997.

Appendix A

ADT specifications

Here we list the format specifications of the parse tree and parse table as used in our implementation of SGLR. These were fed to APIGen to produce the typesafe access api's.

A.1 Parsetree format

```
[
  constructor(ParseTree, top, parsetree(<top(Tree)>, <amb-cnt(int)>)),

  constructor(Tree, appl, appl(<prod(Production)>, <args(Args)>)),
  constructor(Tree, char, <character(int)>),
  constructor(Tree, Lit, lit(<string(str)>)),
  constructor(Tree, Amb, amb(<args(Args)>)), list(Args, Tree),

  constructor(Production, Default, prod(
        <lhs(Symbols)>,
        <rhs(Symbol)>,
        <attributes(Attributes)>)),
  constructor(Production, List, list(<rhs(Symbol)>)),
  constructor(Production, Lit, lit(<rhs(Symbol)>)),

  constructor(Attributes, no-attrs, no-attrs),
  constructor(Attributes, attrs, attrs(<attrs(Attrs)>)),

  list(Attrs, Attr),
  constructor(Attr, assoc, assoc(<assoc(Associativity)>)),
  constructor(Attr, term, term(<term-arg(term)>)),
  constructor(Attr, id, id(<module-name(str)>)),
  constructor(Attr, bracket, bracket),
  constructor(Attr, reject, reject),
  constructor(Attr, prefer, prefer),
  constructor(Attr, avoid, avoid),

  constructor(Associativity, left, left),
  constructor(Associativity, right, right),
  constructor(Associativity, assoc, assoc),
  constructor(Associativity, non-assoc, non-assoc),

  constructor(Symbol, lit, lit(<string(str)>)),
  constructor(Symbol, cf, cf(<symbol(Symbol)>)),
  constructor(Symbol, lex, lex(<symbol(Symbol)>)),
  constructor(Symbol, empty, empty),
  constructor(Symbol, seq, seq(<symbols(Symbols)>)),
  constructor(Symbol, opt, opt(<symbol(Symbol)>)),
  constructor(Symbol, alt, alt(<lhs(Symbol)>, <rhs(Symbol)>)),
  constructor(Symbol, tuple, tuple(<head(Symbol)>, <rest(Symbols)>)),
  constructor(Symbol, sort, sort(<string(str)>)),
  constructor(Symbol, iter-plus, iter(<symbol(Symbol)>)),
  constructor(Symbol, iter-star, iter-star(<symbol(Symbol)>)),
  constructor(Symbol, iter-plus-sep, iter-sep(
        <symbol(Symbol)>,
        <separator(Symbol)>)),
  constructor(Symbol, iter-star-sep, iter-star-sep(
        <symbol(Symbol)>,
        <separator(Symbol)>)),
  constructor(Symbol, iter-n, iter-n(<symbol(Symbol)>, <number(int)>)),
  constructor(Symbol, iter-n, iter-n(<symbol(Symbol)>, <number(int)>)),
  constructor(Symbol, iter-sep-n, iter-sep-n(
        <symbol(Symbol)>,
        <separator(Symbol)>,
        <number(int)>)),
  constructor(Symbol, func, func(<symbols(Symbols)>, <symbol(Symbol)>)),
  constructor(Symbol, parameterized-sort, parameterized-sort(
        <sort(str)>,
        <parameters(Symbols)>)),
    constructor(Symbol, strategy, strategy(<lhs(Symbol)>, <rhs(Symbol)>)),
    constructor(Symbol, var-sym, varsym(<symbol(Symbol)>)),
    constructor(Symbol, layout, layout),
    constructor(Symbol, char-class, char-class(<ranges(CharRanges)>)),

    list(Symbols, Symbol),

    constructor(CharRange, character, <integer(int)>),
    constructor(CharRange, range, range(<start(int)>, <end(int)>)),

    list(CharRanges, CharRange)
  ]
```

A.2 Parsetable format

```
[
  constructor(Version, Default, 4),

  constructor(ParseTable, parse-table,
              parse-table(<version(Version)>,
                          <initial-state(int)>,
                           <labels(Labels)>,
                           states(<states(States)>),
                           priorities(<priorities(Priorities)>))),

  list(Labels, Label),

  constructor(Label, Default, label(
        <production(Production)>,
        <number(int)>)),

  list(States, State),

  constructor(State, Default, state-rec(
        <number(int)>,
        <gotos(Gotos)>,
        <actions(Actions)>)),

  list(Gotos, GotoAction),
constructor(GotoAction, Default, goto(
      <ranges(CharRanges)>,
      <state-number(int)>)),

list(Actions, Action),

constructor(Action, Default, action(
      <ranges(CharRanges)>,
      <choices(Choices)>)),

list(Choices, Choice),

constructor(Choice, reduce, reduce(
      <length(int)>,
      <label(int)>,
      <special-attr(SpecialAttr)>)),
constructor(Choice, lookahead-reduce, reduce(
      <length(int)>,
      <label(int)>,
      <special-attr(SpecialAttr)>,
      <lookaheads(LookAheads)>)),
constructor(Choice, shift, shift(<state-number(int)>)),
constructor(Choice, accept, accept),

constructor(SpecialAttr, none  , 0),
constructor(SpecialAttr, reject, 1),
constructor(SpecialAttr, prefer, 2),
constructor(SpecialAttr, avoid , 4),

constructor(LookAhead, default, look(
      <char-class(CharClass)>,
      <lookaheads(LookAheads)>)),

constructor(CharClass, default,  char-class(<ranges(CharRanges)>)),

list(LookAheads, LookAhead),

list(Priorities, Priority),
  constructor(Priority, left, left-prio(<label1(int)>,<label2(int)>)),
  constructor(Priority, right, right-prio(<label1(int)>,<label2(int)>)),
  constructor(Priority, non-assoc, non-assoc(<label1(int)>,<label2(int)>)),
  constructor(Priority, greater, gtr-prio(<label1(int)>,<label2(int)>))
]
```

Appendix B

SDF grammars

B.1 GSS Example Grammar

Shown below are the SDF specifications of the grammar as used in the GSS example in section 2.2.1.

B.1.1 Booleans

```
module basic/Booleans

imports basic/BoolCon
exports
   sorts Boolean

   context-free syntax
      BoolCon                        -> Boolean {cons("constant")}
      lhs:Boolean "|" rhs:Boolean    -> Boolean {left, cons("or")}
      lhs:Boolean "&" rhs:Boolean    -> Boolean {left, cons("and")}
      "not" "(" Boolean ")"          -> Boolean {cons("not")}
      "(" Boolean ")"                -> Boolean {bracket, cons("bracket")}

   context-free priorities
      Boolean "&" Boolean            -> Boolean
      Boolean "|" Boolean            -> Boolean

hiddens
   context-free start-symbols
      Boolean
   imports
      basic/Comments

   variables
      "Bool"[0-9]*                       -> Boolean
      "Bool-con"[0-9]*                   -> BoolCon
```

B.1.2 BoolCon

```
module basic/BoolCon

exports
   sorts BoolCon

   context-free syntax
      "T"    -> BoolCon {cons("true")}
      "F"    -> BoolCon {cons("false")}

hiddens
   context-free start-symbols
      BoolCon
```

B.2 Elkhound test grammars

These are the test grammars as were used in the Elkhound case study.

B.2.1 EFa Grammar

```
module Elkhound1

exports
   context-free start-symbols
      E

   sorts
      E F

   context-free syntax
```

```
      E "+" F     -> E
      F           -> E
      "(" E ")"   -> F
      "a"         -> F
```

B.2.2 EEb Grammar

```
module Elkhound2

exports
   context-free start-symbols
      E

   sorts
      E

   context-free syntax
      E "+" E  -> E
      "b"      -> E
```

Appendix C

Parse Table Excerpts

C.1 Boolean Parse Table

```
state-rec(0,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([300],7),goto([299],6),
    goto([285],1),goto([294],2),goto([287],3),goto([257],5),goto([288],3),
    goto([259],4),goto([295],3),goto([298],2),goto([286],1)
],
[
    action([38,range(40,41),70,84,110,124,256],[reduce(0,286,0)]),
    action([37],[shift(9)]),
    action([range(9,10),13,32],[shift(8)])
]),
state-rec(1,
[
    goto([40],20),goto([110],19),goto([70],18),goto([84],17),goto([289],14),
    goto([261],16),goto([290],14),goto([262],15),goto([291],14),goto([293],14),
    goto([296],12),goto([265],13),goto([297],12),goto([266],11),goto([292],10)
],
[
    action([40],[shift(20)]),
    action([110],[shift(19)]),
    action([70],[shift(18)]),
    action([84],[shift(17)])
]),
state-rec(6,
[
    goto([256],29)
],[
    action([256],[accept])
]),
state-rec(10,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([285],31),goto([294],2),
    goto([287],3),goto([257],5),goto([288],3),goto([259],4),goto([295],3),
    goto([298],2),goto([286],31)
],
[
    action([38,range(40,41),70,84,110,124,256],[reduce(0,286,0)]),
    action([37],[shift(9)]),
    action([range(9,10),13,32],[shift(8)])
]),
state-rec(11,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(1,297,0)])
]),
state-rec(12,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(1,293,0)])
]),
state-rec(13,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(1,296,0)])
]),
state-rec(14,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([285],32),goto([294],2),
    goto([287],3),goto([257],5),goto([288],3),goto([259],4),goto([295],3),
    goto([298],2),goto([286],32)
],
[
    action([38,range(40,41),70,84,110,124,256],[reduce(0,286,0)]),
    action([37],[shift(9)]),
    action([range(9,10),13,32],[shift(8)])
]),
state-rec(17,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(1,266,0)])
]),
state-rec(18,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(1,265,0)])
]),
state-rec(31,
[
    goto([124],46),goto([264],45)
],
[
    action([256],[reduce(3,299,0)]),
    action([124],[shift(46)])
]),
state-rec(32,
[
    goto([38],48),goto([124],46),goto([263],47),goto([264],45)
],
[
    action([256],[reduce(3,299,0)]),
    action([38],[shift(48)]),
    action([124],[shift(46)])
]),
state-rec(45,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([285],53),goto([294],2),
    goto([287],3),goto([257],5),goto([288],3),goto([259],4),goto([295],3),
    goto([298],2),goto([286],53)
],
[
    action([38,range(40,41),70,84,110,124,256],[reduce(0,286,0)]),
    action([37],[shift(9)]),
    action([range(9,10),13,32],[shift(8)])
]),
state-rec(46,
[],
[
    action([range(9,10),13,32,37,40,70,84,110],[reduce(1,264,0)])
]),
state-rec(47,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([285],54),goto([294],2),
    goto([287],3),goto([257],5),goto([288],3),goto([259],4),goto([295],3),
    goto([298],2),goto([286],54)
],
[
    action([38,range(40,41),70,84,110,124,256],[reduce(0,286,0)]),
    action([37],[shift(9)]),
    action([range(9,10),13,32],[shift(8)])
]),
state-rec(48,
[],
[
    action([range(9,10),13,32,37,40,70,84,110],[reduce(1,263,0)])
]),
state-rec(53,
[
    goto([40],20),goto([110],19),goto([70],18),goto([84],17),goto([289],58),
    goto([261],16),goto([290],58),goto([262],15),goto([291],58),goto([293],58),
    goto([296],12),goto([265],13),goto([297],12),goto([266],11)
],
[
    action([40],[shift(20)]),
    action([110],[shift(19)]),
    action([70],[shift(18)]),
    action([84],[shift(17)])
]),
state-rec(54,
[
    goto([40],20),goto([110],19),goto([70],18),goto([84],17),goto([289],59),
    goto([261],16),goto([290],59),goto([262],15),goto([293],59),goto([296],12),
    goto([265],13),goto([297],12),goto([266],11)
],
[
    action([40],[shift(20)]),
    action([110],[shift(19)]),
    action([70],[shift(18)]),
    action([84],[shift(17)])
]),
state-rec(58,
[
    goto([37],9),goto([range(9,10),13,32],8),goto([285],64),goto([294],2),
    goto([287],3),goto([257],5),goto([288],3),goto([259],4),goto([295],3),
    goto([298],2),goto([286],64)
],
[
    action([38,40,70,84,110],[reduce(0,286,0)]),
    action([41,124,256],[reduce(0,286,0),reduce(5,292,0)]),
    action([37],[shift(9),reduce(5,292,0)]),
    action([range(9,10),13,32],[shift(8),reduce(5,292,0)])
]),
state-rec(59,
[],
[
    action([range(9,10),13,32,range(37,38),41,124,256],[reduce(5,291,0)])
]),
state-rec(64,
    [goto([38],48),goto([263],47)
],
[
    action([38],[shift(48)])
]),
```