SAGA
Syntax Analyzer Generator for Agents

by
Anders Andersson
SAGA

Syntax Analyzer Generator for Agents

Anders Andersson

13 October 1995

Programming Systems Group
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden
email: andand@sicse

Abstract

LALR(1) parser generators in conjunction with imperative programming languages is the standard solution for parsing applications. Contrary to popular belief, this situation is not entirely satisfactory. With imperative tools like Yacc, it is difficult to add actions to a grammar without introducing conflicts. Also, LALR(1) is not powerful enough to meet the needs of languages like C++. We argue that LALR(1) parsing in conjunction with concurrent constraint programming is an interesting option that solves the former problem. We also show how deep guards and don’t-know non-determinism can be exploited to solve the latter problem. These ideas have been incorporated in the SAGA parser generator described in this report.

SAGA is an integrated generator for lexical analyzers and parsers. It is based on AKL, a multiparadigm programming language based on a concurrent constraint framework. It is intended both as a research tool to demonstrate the power offered by concurrent constraints, and as a practical tool for the Agents programming system. As a practical tool, SAGA features many improvements over tools like Yacc. For example, SAGA offers powerful syntax, elegant reporting and resolution of conflicts and powerful error handling in the generated syntax analyzers.

Keywords: Concurrent constraints, Bottom-up parsing, LALR(1), AKL, Agents
1 Introduction

The words forming the acronym SAGA describe it reasonably well. Some points are quite moot, so let us consider the words step by step.

Syntax Analyzer

A syntax analyzer is a program that takes a text and breaks it down according to a set of rules. The term is more general than the term "parser" and also include the task of lexical analysis. Thus, the term implies that SAGA integrates the tasks of lexical analysis and parsing.

Generator

A generator is something that brings something else into existence. In this case SAGA creates syntax analyzers, thus creating a certain kind of programs.

for Agents

Agents is a programming system for AKL developed at SICS. AKL [Jan94] is a multiparadigm programming language based on a concurrent constraint framework. Having the programming system represented in the acronym suggests that it had an impact on the design. In fact, concurrent constraints had a substantial impact on the design and semantics of SAGA. The fact that the "A" stands for "Agents" rather than "AKL" suggest that SAGA is a practical tool that depends on the Agents environment as well as on the AKL language. SAGA actually creates a combination of C code and AKL code, so it relies on the foreign function interface of the Agents system.

SAGA was created with a dual purpose. One purpose was to evaluate the benefits that a concurrent constraint programming language can offer a parser generator. Thus, SAGA is a research experiment. The other purpose was to create a practical, useful tool for the Agents system. The fact that SAGA is designed for real use is a guarantee that the research results are valid in practice. As a practical tool, it features many minor improvements over tools like Yacc. For example, SAGA offers integration between the lexical analyzer and the parser, powerful syntax, elegant reporting and resolution of conflicts and powerful error handling in the generated syntax analyzers.

This report can be read from different viewpoints. It covers the view of the researcher, the view of the user, the view of the designer and the view of the implementor. Essentially, these views have been given their own chapters that are largely independent. This, by necessity, implies that some material is duplicated between chapters.

Chapter 2 describes the view of the researcher, i.e. the benefits that concurrent constraints offer with respect to parser generators. It can be read completely independently of the other chapters.
The other chapters cover the aspects of SAGA as a practical tool. Since no description of a program is complete without a user manual, chapter 3 contains the full user manual for SAGA version 1.0. The user manual does not refer to any of the other chapters. Chapter 4 discusses the design decisions made for SAGA. A certain degree of familiarity with the material covered in the user manual is expected in this chapter. Chapter 5 describes the implementation of SAGA and is rather more technical than the other chapters. It requires a fair amount of knowledge of parser technology.
2 Motivation

In this chapter we discuss the benefits that come from using a concurrent constraint programming language.

2.1 Introduction

The currently dominating solution for parsing applications is to use parsers in imperative languages generated by LALR(1) [ASU86] parser generators, such as Yacc [Joh75]. There are two drawbacks of this solution. The most tangible one is that some of the currently popular languages, most notably C++ [ElS90], are difficult to parse in this way because they require unbounded look-ahead for some constructions. A more subtle, but at the same time more fundamental, problem is that while a grammar may be LALR(1), it can be difficult to augment it with actions in a natural manner. We start by examining these problems and outline how they have been handled by a switch towards LL based techniques, despite the weaker parsing power of LL compared to LR and LALR. We then discuss how concurrent constraint programming can be used to solve these problems without sacrificing the basic power of the LR approach.

2.2 Problems with Imperative LR Parsing

For the purpose of this discussion, we may ignore the fact that LALR(1) is somewhat weaker than LR(1), and focus on LR(k) parsers. First we discuss the case when LR(1) is sufficient in theory, but not in practice. Then we turn to the case when LR(1) is not even sufficient in theory.

2.2.1 Pure LR versus LR with Actions

Many of the current programming languages fit comfortably into an LR framework when their pure grammars are considered. The situation may change quite drastically when the grammar is to be extended with actions that should be executed during the parse.

LR parsers operate in a bottom-up manner, by performing successive reductions on the input until the start symbol of the grammar is reached. Virtually without exceptions, the user supplied actions that augment the grammar are executed together with the reductions. Thus, the order in which the actions are executed in a bottom-up parser corresponds to that of a post-order traversal of the parse tree. This behaviour is highly visible in contemporary LR parser generators. The most
basic problem is that the direction of the flow of information is given by the parsing strategy. Information has a natural flow upwards in the parse tree (by means of synthesized attributes) which makes it difficult to handle some constructions in a natural manner. Flow downwards in the parse-tree (inherited attributes) is perhaps less common than flow upwards, but common enough to be important.

Theoretically, LR parsers can handle a limited form of inherited attributes. In practice, programming tricks are needed to exploit this potential [Sta94]. A definition of the class of attributed grammars that an LR parser can handle, the class of LC-attributed grammars, can be found in [Fil88]. For the purpose of our discussion, we only need two properties of this class, so we omit the definition itself. First, the LC-attributed grammars is a subclass of the L-attributed grammars. Simply stated, L-attributed grammars are those that only allow information flow from the left to the right in the parse tree, i.e. that could be evaluated by an inorder traversal of the parse tree. This includes a fairly large subclass of the inherited attributes. Second, assume that we have an L-attributed LR(k) grammar G. Further assume that we from G derive a grammar G' with actions (but without inherited attributes) in the following way: To the immediate left of any (right hand side) occurrence of a non-terminal that has inherited attributes we insert an action. Then G is LC-attributed if and only if G' is LR(k) [Sta94]. Thus, we may equally well discuss the effect of actions in a grammar as the effect of inherited attributes. The problem then becomes that actions may have to be inserted in the middle of rules.

LR grammars are highly sensitive to the placement of actions inside the rules, as opposed to the natural placement of actions last in the rule. Strictly speaking, any action must be placed at the end of a rule. Many parser generators support actions in mid-rules, but they are actually implemented with grammar transformations. This technique is known as “cracking” of the rules and transforms one rule into several that all have the actions last in the rule. This means that the insertion of mid-rule actions may change the grammar so that it ceases to be LR(1). It is possible to prove that arbitrary insertion of actions weakens LR(k) into LL(k) [Par93]. While this is a worse-case scenario, rule cracking is a very real threat to the LR-ness of a grammar and thus to the practical parsing power.

2.2.2 The Need for Unbounded Look-Ahead

Some contemporary programming languages are notoriously hard to parse with non-backtracking parsers. C++ is the most well-known example [EIS90]. C++ is not LR(k) for any k. As a matter of fact, even unbounded look-ahead does not suffice, since the C++ grammar is inherently ambiguous. For example, declarations and expression statements overlap to some extent. In some situations unbounded look-ahead is needed to distinguish between them, and in some cases they are indistinguishable.
In the latter case they should be interpreted as declarations, but this cannot be expressed with any context-free grammar.

2.3 Previous Work

In this section we outline some of the techniques that have been used to address the problems in the previous section. We concentrate the discussion to LL($k$) parsers and backtracking parsers, that address the problem with actions and the need for unbounded look-ahead, respectively.

Logic programming offers a noteworthy contribution (apart from a nice form of backtracking) namely the concept of a logical variable. The benefits of logical variables are similar to, but much more limited than, the benefits of concurrent constraints, so we leave them until the next section.

Other possibilities for the handling of inherited attributes include transformations and various forms of tree parsing. Transformations of grammars to simulate inherited attributes is possible to some extent, but hardly a viable general solution. A more interesting possibility is to build a parse tree and to compute attributes and perform actions after the parse proper [ASU86]. This idea has never quite caught on. One problem is efficiency, especially in terms of space requirements.

2.3.1 LL($k$) Parsers

A drastic approach is to abandon LR parsing altogether in favour of LL parsing. This approach has always had its proponents, but it has been more successful with the recent progress made in efficient LL($k$) parsing, for $k > 1$ [Par93,PDC92]. The drawback is of course a significant loss of parsing power. The most obvious example is that left recursion must be eliminated from the grammar, but it goes much deeper than that. While LL($k$) is strictly stronger than LL($k$-1) [FiL88], it is (clearly) strictly weaker than LR($k$). In turn, LR($k$) is no stronger than LR(1) in the sense that every LR($k$) grammar has an LR(1) equivalent [Knu65]. While the latter result only concerns pure grammars without actions, it still illustrates an important point on relative strengths of LL and LR.
2.3.2 Backtracking Parsers

The need for unbounded look-ahead has mainly been addressed with backtracking parsers. Most backtracking implementations have been based on top-down rather than bottom-up techniques. A top-down framework gives us a natural way to control the backtracking quite closely. For example, to handle the ambiguity between expressions and declarations in C++, we want to try to parse a declaration first. If that fails, we want to try to parse an expression. Especially interesting is the Definite Clause Grammar, DCG, in Prolog [PPW78,PeW80]. The semantics of Prolog is ideal for top-down backtracking parsers, and the clauses of the grammar are indeed translated very directly into Prolog predicates. The main advantage is that the effects of actions are undone on backtracking, giving a clean semantics. Also, actions can be used as semantic tests that guide the parse.

As we have discussed earlier, choosing an top-down technique represents a significant loss of parsing power. When backtracking is used, this is compensated for to some extent. The problem is that efficiency considerations force us to keep backtracking down to a minimum. It should only be used when it is really needed. Backtracking for bottom-up parsers has mainly been considered in the context of natural language applications [MTH83,Tom85]. In these applications, efficiency is measured on an entirely different scale. Also, explicit control over the order in which alternatives are tried is less relevant.

2.4 Benefits of Concurrent Constraints

Concurrent constraints solve the problem of inherited attributes in a perfectly natural way without putting any strain on the grammar writer. The effect is an entirely automatic consequence of the semantics of such a language.

To find a good solution on the problem with the need for unbounded look-ahead, we need to narrow the scope of languages further. We sketch how the features of the AKL language [Jan94] can be used as a foundation for controlled backtracking in an LR parser. To various extent these ideas can be carried over to other similar languages.

2.4.1 Concurrent Constraint Programming

Concurrent constraint programming [Sar89] is based on the idea of independent agents that are performing computation and communicating information by means of a common constraint store. Variables are not assigned values in the imperative sense, but rather constrained to a set of possible values. Such a set can become
smaller with time, as more constraints are computed, but elements are never added to it. In particular, if a variable has a known value, that value cannot be changed.

Agents are defined with clauses divided into a guard and a body. The guard ask if constraints are entailed (known to hold). If the guard is entailed the body is executed. The guard could also be disentailed (known not to hold) in which case the body is not executed. The third possibility is that the asked constraint is neither entailed nor disentailed, i.e. it is not (yet) known if it holds or not. In this situation the agent waits until the constraint becomes entailed or disentailed. In the body of an agent constraints may be told, i.e. assured to hold, thus communicating information to other agents. This scheme makes computation order rather unimportant which allows for parallel execution.

2.4.2 Constrained Attributes

This behaviour of a concurrent constraint language is almost ideal for a bottom-up parser. The attributes of the grammar rules are constrained instead of assigned. The difference between synthesized and inherited attributes becomes reduced to an issue of whether the agent that asks a constraint is above or below the agent that tells the constraint, in the parse tree. The bottom-up scheme means that inherited attributes will be asked first and told later, but this is no problem whatsoever. The agent that needs to access the inherited value will simply wait until the attribute has become sufficiently constrained.

Logic programming languages also have this behaviour in a limited form. Indeed, the example we shall show in this section would work equally well in a Prolog based implementation. Prolog can handle unification of inherited attributes before there values are available. Execution of predicates are also possible to some extent. The problem is that Prolog is sensitive to execution order. Some built-in predicates, like arithmetic, require their arguments to be ground. Worse yet, performance may deteriorate severely when the computation order changes. In many cases the performance may become infinitely bad, i.e. the program fails to terminate.

Let us look at an illustrative example. We will take a look at parsing declarations in the C programming language [KeR89]. For those unfamiliar with C, this may be a typical declaration in C:

```c
const int **foo[42];
```

This declaration declares the identifier ‘foo’ to be an array of 42 pointers to pointers to constant integers. Much more complex declarations is allowed, but this small example illustrates the basic idea. A convenient way to decode such an declaration is as follows:
int **foo[42]  
is constant.

**foo[42]  
is integer.

*foo[42]  
is a pointer.

foo[42]  
is a pointer.

foo  
is an array of 42 elements.

The following grammar fragment is adapted from a C grammar. The grammar is presented without attributes and in the syntax used by the SAGA parser generator. The text in quotes are terminal symbols, colon is used to separate the head non-terminal from the body, commas separate the grammar symbols in the body and rules are terminated by periods. The percent sign is used to indicate a comment to the end of the line.

declaration: declaration_specifiers, declarator, ";". % Simplified
declarator: declarator2.
declarator: pointer, declarator2.

declarator2: identifier.
declarator2: "(", declarator, ")".
declarator2: declarator2, "[", "]".
declarator2: declarator2, "[", constant_expr, "]".
% Some additional rules removed

pointer: "*".
pointer: "*", pointer.
% Some additional rules removed
declaration_specifiers:
    declaration_specifiers, declaration_specifier.
declaration_specifiers: declaration_specifier.
declaration_specifier: "const".
declaration_specifier: "register".
declaration_specifier: "char".
declaration_specifier: "int".
% Some additional rules removed

Some of the non-terminals referred to in the grammar fragment are left out, but their names should give sufficient indication of their purposes.

Assume that we wish to augment this grammar with attributes and actions to build a syntax tree. It is reasonable to wish that the declaration:

    const int **foo[42];
should produce the following syntax tree:

```
declarate
 / \ 
foo array
   / \ 
  42 pointer
    | 
   pointer
    | 
   const
    | 
   int
```

This structure is in sharp contrast to what you can build with synthesized attributes in a traditional bottom-up parser in a natural manner. Syntax trees built bottom-up by necessity resemble the parse tree for the grammar, modulo some compaction. Such a syntax tree would be similar to one below.

```
declarate
 / \ 
int declarator
   / \ 
const pointer array
   / \ 
  pointer 42 foo
```

The desired kind of syntax tree would be easily enough built by a top-down parser, but the given grammar is left recursive and thus not \( \text{LL}(k) \) for any \( k \).

Let us now see how easy it is to build the desired parse tree when we use constrained attributes. Using SAGA syntax an augmented grammar may look something like the following. Attributes are written inside parentheses following the non-terminals. The attributes used in this example are either variables, starting with an uppercase letter, or records named by a lowercase name and with the fields enumerated inside parentheses. These records directly correspond to a syntax tree of the desired kind.

```
declaration(decclare(Id, Type)):
    declaration_specifiers(BasicType),
    declarator(Id, BasicType, Type), ";".

declarator(Id, Type0, Type): declarator2(Id, Type0, Type).

declarator(Id, Type0, Type):
    pointer(Type0, Type1), declarator2(Id, Type1, Type).

declarator2(Id, Type, Type): identifier(Id).

declarator2(Id, Type0, Type):
    "(" , declarator(Id, Type0, Type), ")".

declarator2(Id, Type0, array(Type)):
    declarator2(Id, Type0, Type), "]", "]".
```
declarator2(Id, Type0, array(Expr, Type));
    declarator2(Id, Type0, Type),
    "[", constant_expr(Expr), "]".

pointer(Type, pointer(Type)): "*".
pointer(Type0, pointer(Type)): "*", pointer(Type0, Type).

One can observe how the information flows. In the case of declarator and declarator2 the first attribute is synthesized, passing the name of the declared identifier upwards. The second one is inherited, and used to pass information about the surrounding type. The third argument is synthesized, but contains information from the second, thus incrementally adding information. A critical point where the information flow changes from going down into going up is the following rule:

declarator2(Id, Type, Type): identifier(Id).

where the second and third arguments are constrained equal. The same "down-then-up" behaviour can be observed in the first and second argument to pointer. In fact, this pattern is common enough to have its own syntax. Using the so called accumulator syntax (that SAGA has borrowed from the Agents system) pairs of attributes that behave as above may be indicated with single variables placed after the ordinary attributes. Using this syntax the rules for declarator can be written in the following manner.

declarator(Id)-Type: declarator2(Id)-Type.
declarator(Id)-Type:
    pointer-Type, declarator2(Id)-Type.

Note in particular that the three occurrences of 'Type' in the second rule are treated differently, to obtain exactly the same effect as in the original rule, thus saving the nuisance of numbering variables by hand.

In this example the flow of information is from left to right. In contrast to imperative LL and LR parsing, this is not a necessary condition for concurrent constraint LR parsing. It is quite possible to have information flow from right to left, which in some cases can simplify the actions considerably.

### 2.4.3 Deep Guards and Backtracking

Some concurrent constraint languages, for instance AKL [Jan94] that the Agents system is based on, supports so called deep guards. Deep guards means that the guard part of the clauses (the part that asks constraints) may perform almost arbitrary computation, as opposed to flat guards that just contain constraints.

Deep guards makes it possible to perform a computation in a local constraint store. If the computation fails the global store remains unchanged. If the computation
succeeds we can explicitly tell the computed result, and thus change the global
store. In the context of parsing this means that we can perform an experimental
attempt to parse a piece of text using some rule. If the computation fails we can
try another rule and so on. Deep guards are certainly not necessary to achieve a
backtracking parser, but it allows us to make it very clean and declarative. As
for logic programming, the actions in branches of the execution that eventually fail
cannot affect the state of the successful branches. Furthermore, deep guards allows
us to prune the search in a clean way. In Prolog, a cut would probably be used to
prune the search, which would result in a less clean semantics.

Recall the ambiguity in C++ concerning expression statements and declarations. In
SAGA the problem can be handled in the following manner:

```
expression_stmt_or_declaration: declaration -> true.
expression_stmt_or_declaration: expression_stmt.
```

In SAGA terminology, this is known as a conditional rule and is interpreted as follows.
To match an expression_stmt_or_declaration, first attempt to (conditionally)
match a declaration. If that fails, match an expression_stmt (unconditionally).
In general there can be any number of conditional clauses that are tried sequen-
tially, and optionally an unconditional rule. Thanks to deep guards we can also
perform actions inside conditional branches, without affecting the global state of
the computation, unless the conditional branch succeeds.

SAGA limits the use of the above construction so that the left edge of an expression
statement must be LL(1) recognizable. In other words, we must recognize that we
have either an declaration or an expression statement when we see the start of one.
This is not a very severe limitation and it is needed to get a reasonably simple se-
manitics of the construction. It also allows a fairly straight-forward implementation.

To handle really difficult languages one can use non-determinism to resolve LALR(1)
parsing conflicts by exhaustive search. This technique is quite popular in the natural
language processing community. For programming languages the constructions that
require such extreme measures are most likely isolated parts of the language. In
this case it is important to encapsulate the non-deterministic computation to avoid
combinatorial explosion. In AKL, non-determinism is encapsulated inside deep
guards, so that only the first solution will be found. This behaviour is directly
inherited by the SAGA parser generator. Syntax (analogous to the syntax of non-
deterministic guards in AKL) is provided to indicate in which parts of the grammar
non-determinism is allowed. Non-determinism is encapsulated inside conditional
rules and thus it need not affect the performance of the entire parse but rather just
the performance on the difficult parts.

The independence of computation order does have one drawback in this context. A
non-deterministic branch of the parse may continue to execute as long as it matches
the input, even if one of its associated actions fails (or rather will fail as soon as it is executed). This is because we do not guarantee any synchronization between the execution of the actions and the progress of the parse. If the agents that perform the parse proper happens to execute before the agents associated with user actions, then we may continue a futile parse longer than necessary. In contrast, in Prolog a failing action would cause immediate termination of the current branch of parse, thereby pruning the search. If such a synchronization is desired in SAGA, it must be explicitly done in user actions.

2.5 Conclusion

Parser generators are often seen as successful examples of declarativity. Unfortunately, this is not really true for generators for imperative languages. Concurrent constraint languages are much more suitable for the job, and approximate the abstract notion of attributed grammars quite closely. They also lend themselves to powerful extensions beyond the basic LR scheme gracefully.

We also note how some of the advantages of concurrent constraint programming languages over logic programming languages seems to be emphasized in LR parsing. Logic programming languages are more sensitive to execution order, which severely limits the possible actions that could be performed with inherited attributes. The guards of concurrent constraint languages also offers a way of controlling search which is clearly superior to the cut of Prolog. This is demonstrated quite clearly by how easy we generalized and exploited the guards to control search in the parse.
Chapter 3: User Manual

3 User Manual

This chapter presents the SAGA User Manual, version 1.0. It also contains the documentation of the SAGA Standard Library, version 1.0.

3.1 Introduction

SAGA is an acronym for Syntax Analyzer Generator for Agents. With generator we mean a programming tool that produces (or generates) a program as output. With syntax analyzer\(^1\) we mean a program that reads an input text, and uses some sort of syntactic rules to check that the input is valid and to extract some information from the text.

Thus, the full name indicates that SAGA is a tool that creates Agents programs that perform syntax analysis. This description is rather vague about the nature of the texts that can be analyzed and the kind of analysis performed. This is quite intentional, since the possibilities SAGA offers are sufficiently rich to support a wide range of applications. However, SAGA was originally designed to support syntax analysis of programming languages, and many of its features are especially suited to such applications.

The description of the desired syntax analyzer that is given to SAGA is based on regular expressions, context-free grammars and some SAGA specific concepts (see Section 3.3 [Basic Concepts of SAGA], page 15). The generated program uses an LALR(1) parsing algorithm (see Section 3.10 [The SAGA Parsing Algorithm], page 64), but it also supports a large class of non-LALR(1) grammars through various extensions. See Section 3.12 [Advanced Parsing Strategies], page 73 for a description of the most powerful parsing methods available in SAGA.

We begin with introductory sections on the basic concepts needed to use SAGA. See Section 3.2 [SAGA Input and Output], page 14, and Section 3.3 [Basic Concepts of SAGA], page 15. If you feel comfortable with most of the concepts mentioned above you may skim these sections. Then we develop a fairly large tutorial example, in a step-wise fashion, to demonstrate many of the features of SAGA. See Section 3.4 [Tutorial], page 22. We strongly recommend a look at this section, for beginners as well as for experts on syntax analysis. The rest of the manual contains reference sections that cover specific areas in greater detail and a more terse style.

This manual as well as the SAGA tool were written by Anders Andersson. This manual corresponds to version 1.0 of SAGA.

\(^1\) Many authors use this term in a more narrow sense than we do in this text.
3.2 SAGA Input and Output

With a tool like SAGA it is easy to get confused by all the languages involved. SAGA takes a specification file in one language and generates a program file in another language. Then the generated program is used to analyze a text in a third language, and maybe it even compiles the third language into a fourth as part of the analysis. Let us try to get the concepts straight.

You, the user of the SAGA tool, create a file describing the language you want to analyze and the analysis to perform. This file is called the SAGA grammar file or just the SAGA file for short. This file have a `.saga` suffix. This file is processed by the SAGA tool, often called the generator, which is invoked by the saga command. As a result you get three files, with suffixes `.akl`, `.c` and `.fd`, respectively. Thus, the generator behaves like this:

```
saga my-parser.saga
  ⇒ my-parser.akl my-parser.c my-parser.fd
```

The generated files are then given to the Agents system, where they implement the SAGA parser (the parser for short). The parser is an Agents agent and thus may be called from some program or from the Agents top-level. The name of the agent to call, and the number of arguments to pass to it depends on the grammar file. In general, it looks something like:

```
my_module.my_rule(Arg1, ..., ArgN, Status,
  InitialGlobals, FinalGlobals, Settings,
  Input, RemainingInput)
```

The names `my_module` and `my_rule` are taken from the SAGA file, as is the number and nature of the arguments that are passed to the grammar rule. For a description of all the arguments, see Section 3.9.1 [Calling the Parser], page 60.

The purpose of the parser is to process some input text, and perform the desired analysis. This input need not come from a file, even if it often does. The analysis usually produces some output that may be returned to the caller, written to file or used in some other fashion.

3.3 Basic Concepts of SAGA

SAGA analyzes an input text in three main phases: Lexical analysis, filtering and parsing. The lexical analyzer divides the sequence of characters in the input into “words”, known as tokens. The filter may replace certain tokens with sequences of other tokens before passing them on to the parser. The parser then imposes a
hierarchical structure on the tokens, according to the grammar. The parser might for
instance build a syntax-tree, a data-structure describing this hierarchical structure,
or it may compute some other information that depends on this hierarchical structure
in a more implicit way. In the simple case we have:

\[
\text{characters} \Rightarrow \text{raw-tokens} \Rightarrow \text{filtered-tokens} \Rightarrow \text{syntax-tree}
\]

\[
\text{lexer} \quad | \quad \text{filter} \quad | \quad \text{parser}
\]

describing the kind of information that is processed by the lexical analyzer, the
filters and the parser, in that order.

3.3.1 Lexical Analysis

The purpose of the lexical analysis\(^3\) is to break up the input, a sequence of characters,
into short sequences ("words") and to tag them with the "class" they belong to. In
our terminology, the tag and the text taken together is called a token, while just
the raw text without the tag is called a lexeme and just the tag without the text is
called a token type\(^4\). Hence, the lexical analyzer are said to tokenize its input.

3.3.1.1 Example

As a simple example, look at an arithmetical expression:

\[
42 - (7+4711+-22)
\]

A typical tokenization of these 18 characters with SAGA would be into the following
12 tokens, with their token types (e.g. number) and lexemes (e.g. '42') indicated:

1. The number '42'
2. A sequence of one blank character
3. The minus-sign ' - '
4. A sequence of two blank characters
5. The left-parenthesis '('
6. The number '7'

\(^2\) Actually, what we call parsing here would be more accurately called "syntax-
directed translation" in conventional compiler construction terminology.

\(^3\) Also called "scanning" by some authors.

\(^4\) Again, other conventions are sometimes used by other authors.
7. The plus-sign '+'
8. The number '4711'
9. The multiplication-sign '⋆'
10. The minus-sign '-'
11. The number '22'
12. The right-parenthesis ')

There are a few interesting points to consider. First, note that we consider the blanks as tokens. This is not the way it is typically done. More often, removing unwanted blanks (white space) is considered to be part of the lexical analysis. In SAGA this is performed by the filter instead. Thus, for SAGA it is true that the input is the concatenation of the lexemes of all (unfiltered) tokens. Second, note that the minus-sign token type actually is used in two different roles, both as a binary operator and as an unary operator. The lexical analyzer normally does not, cannot and should not try to observe this difference. The difference between the two is for the parser to decide. In the case of negative number constants, some languages treat the minus-sign as part of the constant, but this is another matter entirely.

3.3.1.2 Regular Expressions

To describe the tokenization process SAGA, like most other similar tools, use regular expressions. A regular expression describes a set of strings. The regular expression is said to match any string in this set. We can associate each token type with a regular expression that describes its set of possible lexemes. In SAGA we do this by letting the token type be the regular expression itself. Of course, it is quite possible that more than one regular expression match some of the same strings or even exactly the same set of strings. For now, we ignore such problems.

Regular expressions are defined inductively from a set of simple, atomic expressions combined with the help of a set of operations. These sets can be chosen in different ways. The pure form used in mathematics contains just a few while SAGA provides quite many. (It is a question of purity versus convenience.) Here we only describe

---

5 As usual, there are always exceptions.

6 In mathematical settings any set of strings are called a language, and a language that could be described by a regular expression is a regular language. We avoid this terminology since there are so many other meanings attached to the term "language" in this field.
the essential ones. For a description of the full set used in SAGA and their precise syntax, see Section 3.6.2 [Regular Expressions], page 47.

The atomic regular expressions are:

- The regular expression that matches only the empty string.
- One regular expression for each character in the input character set, matching the string that consists of a single occurrence of this character.

These are then combined as follows:

\[ a \mid b \] Matches any string that is matched by \( a \) or \( b \).

\[ a \, b \] Matches any string that can be obtained by concatenation of some string matched by \( a \) with some string matched by \( b \).

\[ a^* \] Matches any string that can be obtained by concatenation of a set of strings matched by \( a \). This includes the empty string, as this is considered to be the concatenation of the empty set of strings. This operator is known as Kleene-star.

As a simple example, consider the set of all strings that begin with ‘c’ followed by any number, including zero, ‘a’ or ‘b’.

In SAGA syntax the corresponding regular expression may be written (the parentheses are used for grouping as usual):

\[ 'c' \, ( 'a' \mid 'b' )^* \]

Regular expressions have a rather limited expressive power. They can only remember a fixed amount of information about what they have seen previously. A popular statement of this is that “regular expressions cannot count”, meaning that it is impossible to, say, write a regular expression that matches any string containing an equal number of ‘0’ and ‘1’. This statement may be a little misleading as it stands. It is quite possible to write a regular expression that matches, say, all strings that have less than 100 ‘0’ and the same number of ‘1’. Thus regular expressions can count, but only up to a fixed limit set by the expression itself. This moot point aside, the power of regular expressions is usually enough to perform tokenization. Usually they are a quite compact and convenient description of this process.

### 3.3.2 Filters

The purpose of a filter is to replace some tokens with sequences of tokens. The simplest and most common case is that of removing white space and comments. Removal of tokens is the special case where the token is replaced by the empty sequence of tokens. Another common case is where a lexeme has two possible token types, and where the correct choice of the type depends on some symbol table
information. For an example of this use, see Section 3.4.10 [Handling the Problem with `typedef`], page 42. As a more exotic example where filters can be used, consider macro expansion, where a macro name may be replaced with the sequence of tokens in its definition. See Section 3.8 [Filters], page 59, for more information about filters.

3.3.3 Parsing

The purpose of parsing is to organize the tokens into a hierarchic structure according to a set of grammar rules, and to report errors if the sequence of tokens does not follow these rules. Actually, while we parse we usually build some representation of the hierarchic structure of the program or perform some work that depends on this structure, or both. In general this process is known as syntax-directed translation. For the purposes of this text, it is more convenient to be less exact and speak of parsing whether we perform any data processing while we parse or not.

3.3.3.1 Parse Trees and Grammar Rules

Let us consider an arithmetical expression:

\[
42 - (7 + 47111 - 22)
\]

A parse tree describing this expression is:

```
expression
 / | \  
number - expression
 42 / | \  
 number + expression
   7 / | \  
   number * expression
      4711 / \  
         - number
            22
```

The leaves in a parse tree are tokens. In this context we view them as token types, or terminal symbols, with their lexemes as associated values, or attributes. The names that tag the internal nodes in the tree are called non-terminal symbols. Grammar symbols is a collective term for terminal and non-terminal symbols. Each sub-tree that have a non-terminal symbol at its root is called a syntactic grouping, and the input matched by the sub-tree is also known as a grouping.

The legal ways to form parse trees from the input are described by grammar rules, which formally express relationships between groupings. An informal statement of such a relationship may be: "An expression may be an expression followed by a minus
sign, followed by another expression" or "an expression may be a number". This can be formalized in SAGA syntax as (assuming number to be defined elsewhere):

expression: expression, ',-', expression.
exression: number.

Note that the first rule is recursive, while the second one is not\(^7\), and therefore may serve as one of the base cases we need to get out of the recursion.

The form of rules we work with are called context-free grammar rules. These rules express a relationship between a node and its children in the parse tree, by relating the non-terminal of the node to the grammar symbols of the children nodes (or children leaves). The characteristic property of a context-free grammar is that we relate a single node to its children. In contrast, a context sensitive grammar may refer to the node's neighboring nodes (the context of the node) in the rules.

### 3.3.3.2 Grammar Classes

In practice we often want to restrict the class of grammars allowed to different subsets of the context-free grammars. First, we often want to exclude ambiguous grammars. A grammar is ambiguous if it allows several parse trees to be derived from the same input text. Consider:

expression: expression, '*', expression.
expression: expression, '+', expression.
expression: number.

For the input '2+3*4' the following two parse trees both satisfy the grammar rules:

```
expression / | \ expression * number
/     |     /
| expression number + number
2     / | \   / | \ 4
| number * number number + number
3     | 4
```

Some ambiguous grammars, like the one above, can be handled by SAGA in an elegant manner by providing some additional information to select the desired parse. See Section 3.10.3 [Resolving Shift/Reduce Conflicts], page 66. Alternatively, you could write the grammar a little differently. The grammar:

\(^7\) We assume that the definition of number does not refer to expression.
expression: expression, '+', term.
expression: term.
term: term, '*', number.
term: number.

is unambiguous and matches the same input as the last grammar. It has a parse

tree similar to the one on the left:

```
expression
/    \ 
expression + term
       /    \ 
term   term * number
       /    \ 
number number
     2    3
```

The similarity may not be apparent at the first casual glance, but the one above on
the left can be obtained from this one if all nodes that have only a single child are
removed. If you are not familiar with this construction you should study it carefully,
even though SAGA supports a more direct way to express operator precedence.

Second, we usually want our grammar to be an LALR(1) grammar. The exact def-
inition of an LALR(1) grammar is quite technical, so we describe the slightly larger
class LR(1). For some more information about the differences between LALR(1) and
LR(1), see Section 3.10.4 [Mysterious Reduce/Reduce Conflicts], page 67. Simply
put, a grammar is LR(1) if we can recognize each syntactic grouping on a left-to-
right scan as soon as we have seen all its sub-groupings plus a single token (called
the look-ahead token) past the right edge of the grouping. An example fragment of
a non-LR(1) grammar:

```
statement: name, '(' index, ')', ':=', number, ';'.
statement: name, '(' parameter, ')', ';'.
index: number.
parameter: number.
```

The problem with non-LR(1)-ness for this grammar is related to the similarity
between an index and a parameter. Compare the inputs ‘foo(42):=4711,’ and
‘foo(42),’ (we assume that name matches ‘foo’). In the first case ‘42’ is an index
and in the second it is a parameter. This is not an ambiguity, since the difference
can be detected as soon as we see the token following the right parenthesis. As a
matter of fact, this grammar is LR(2). (LR(k) means that up to k tokens of look-
ahead are examined before deciding which rule to apply.) For LR(1) however, we
need to decide between the two cases as soon as we have seen the initial segment
of the input up to the right end of the subgrouping (‘foo(42)’) and a single token
of look-ahead (‘’). But as the examples show, ‘foo(42)’ can be interpreted as the
beginning of either example, so we simply do not know which one to choose. As an example of a grammar that is not LR(k) for any k, consider a similar grammar:

```plaintext
statement: name, '(' indices, ')', ':=', number, ';'.
statement: name, '(' parameters, ')', ';'.
indices: index.
indices: indices, ';', index.
parameters: parameter.
parameters: parameters, ';', parameter.
index: number.
parameter: number.
```

The problem remains the same, but now the token after the right parenthesis can be arbitrarily far away from the first index, so we cannot fix any k that will suffice as look-ahead.

### 3.3.3.3 Syntax Trees and Semantic Values

While the parse tree describes the syntactic groupings in detail, a syntax tree is a more convenient form of describing much the same thing. The arithmetical expression:

$$42 - (7+4711*-22)$$

has the following syntax tree:

```
       /
      /
     42 +
    /   /
   7   *
  /     /
4711   -
     |
      22
```

A common task for a parser is to construct (a representation of) such a syntax tree. However, this is not the only option. An obvious alternative for these arithmetic expressions is to evaluate them during parsing (yielding the result 103677 for the above example). Other possibilities certainly exist as well.

To allow for these tasks, SAGA lets you associate semantic values with each grammar symbol, and optionally a semantic action with each rule. A semantic value is a

---

8 Think of "semantic" as opposed to "syntactic" — semantic values and actions does not affect the course of the parse, only the computed result.
logical variable and a semantic action is a piece of Agents code to be executed during parsing. Sometimes we think of grammar rules as analogous to agents, and thus we may also refer to the semantic values (of a grammar symbol) as arguments to the rule defining the symbol. To build a syntax tree, the semantic values should hold some representation of syntax trees and the actions should unify the values of a node with an appropriate data structure containing the attributes of its children, thus building a node in the syntax tree. To evaluate an expression, the semantic values may hold numbers and the actions should compute the result of a node from the results of its sub-trees (i.e. the values associated with the children of the node).

3.4 Tutorial

Now we will walk you through a fairly large tutorial example of a SAGA grammar. Starting from scratch and adding a construct or two at the time, we will construct a grammar that parses a small imaginary programming language and builds a syntax tree describing it. The example language we will use is inspired by C, but much smaller, and it is named Cilly (for obvious reasons). The language is actually designed to give us the same problems as C does. At the same time it is much smaller in the sense that it has fewer number of rules, operators and keywords.

At this point the reader might wonder why a C-like language was chosen, rather than, say, an Agents-like one. After all, SAGA is based on Agents, so all its users should be familiar with Agents, but might not be familiar with imperative languages like C, right? While this is true enough, the problem is that the parsing of Agents is a straightforward task that only needs a small subset of the SAGA features. Parsing SAGA itself would be slightly more of a challenge, but still it would not use nearly as many features as Cilly does. Besides, it might be more than a little confusing for the user to deal with SAGA at two levels before getting more of a feel for it.

3.4.1 Parsing Arithmetical Expressions

Let us start our tour with the grammar equivalent of "hello world!", namely parsing of simple arithmetical expressions:

\[
\begin{align*}
\% \text{ Recognizes simple arithmetical expressions} \\
\text{:- saga(example_1, expr).} \\
expr & : expr, "\*", term ; term. \\
term & : term, "\*", factor ; factor. \\
factor & : "-", factor ; "(" , expr , ")" ; \{0-9\}+. \\
\end{align*}
\]
This example is a bare-bones SAGA file without any bells or whistles, showing the most basic elements of the syntax.

The first line is a comment. SAGA uses the same kinds of comments as Agents, either starting with a ‘\%' and ending at the end of the line, or starting with ‘/\*' and running until the first occurrence of ‘\*/'. As other comments, they are only useful for the reader of the SAGA file, they are not present in the generated program.

The second line is a directive. All directives start with ‘:\-’ and roughly look like terms in Agents. This particular directive is known as the header and must be present at the beginning of each SAGA file. The first argument gives the name of the module to put the generated program into. The second argument is the name of the syntactic grouping that matches the entire input. This is also used as a name of the agent used to parse the input. Both the first and the second arguments must always be given. The header may also be given a third optional argument. The third argument was omitted in this simple example. See Section 3.4.9 [Global Accumulators], page 40, for an example of when it is used.

The rest of the example consists of the grammar rules, in this case one for each non-terminal, expr, term and factor. In this case, the grammar rules have no associated values and no associated actions. The syntax of the pure rules is similar of that traditionally used for grammars in mathematics, BNF rules, DCG's in Prolog and Yacc grammar rules, but not identical to any one of them. Non-terminals begin with lower-case letters and the symbol to be defined (the head of the rule) is separated from the clauses defining it (the body of the rule) with a colon. The defining clauses are separated with semi-colons, read as "or", the grammar symbols in each clause are separated by commas, and the rule is terminated with a full-stop (a period followed by white space).

The example contains six terminal symbols described with regular expressions. General regular expressions should be written inside braces ('{'} and '{'). For the simple and common special case when the regular expression is just a literal string we allow the braces to be omitted for convenience and readability. Literal strings may be written inside either single or double quotes. The only non-simple regular expression in the example, ‘[0-9]+’, stands for any sequence of one or more (indicated by the postfix ‘*’ operator) characters in the range ‘0’ to ‘9’, thus expressing all legal integers (including ‘000000’ and such).

Note that the grammar in the example is written to force ‘-’ to bind tighter than ‘*’, which in turn binds tighter than ‘+’. Also, both ‘*’ and ‘+’ are treated as left associative, meaning that an occurrence on the left binds tighter than an occurrence on the right. This is accomplished by the way the grammar is written, see also Section 3.3.3.2 [Grammar Classes], page 19.
To try out this example, we first invoke the saga command. Assuming we put the
text in the file ‘ex1.saga’, this can be done from a Unix shell as follows:\footnote{There may be other ways to do this depending on the system used. Check other
documentation for details.}

\texttt{> saga ex1.saga}

This produces the files ‘ex1.akl’, ‘ex1.c’ and ‘ex1.fd’. You then feed them to the
Agents system.\footnote{Agents may produce additional files during the load process. Consult the docu-
mentation for Agents for more information about these.} An example run may look like this (with some messages removed):

\texttt{\textgreater agents ex1.c ex1.fd}
\texttt{?- load(stdsaga).}
\texttt{yes}
\texttt{?- compilef(ex1).}
\texttt{yes}
\texttt{?- load(ex1).}
\texttt{yes}
\texttt{?- example_1.expr(Status, "42+3*-7", Rest).}
\texttt{Status = true}
\texttt{Rest = [] ?}
\texttt{yes}
\texttt{?-}
\texttt{\textgreater agents ex1.c ex1.fd}
\texttt{?- load(stdsaga).}
\texttt{yes}
\texttt{?- compilef(ex1).}
\texttt{yes}
\texttt{?- load(ex1).}
\texttt{yes}
\texttt{?- example_1.expr(Status, "42+3*-7", Rest).}
\texttt{Status = true}
\texttt{Rest = [] ?}
\texttt{yes}
\texttt{?-}

\texttt{Rest} is the remaining input to be matched, typically the empty list if all went well.
The \texttt{Status} variable returns \texttt{true} to indicate that the parse went well. If errors
occurred, we return information about them. For a description of the structures
returned, Section 3.9.1.1 [The Parser Return Status], page 61. As a quick preview,
let us try a few incorrect inputs:

\texttt{?- example_1.expr(Status, "42+3*-7", Rest).}
\texttt{Status = parse_error([42], line(1,[[52,50],[43],[42]]), [] )}
\texttt{Rest = [42,51,42,45,55] ?}
\texttt{yes}
\texttt{?- example_1.expr(Status, "42++3*-7", Rest).}
\texttt{Status = repairs([deleted([43],line(1,[[52,50],[43],[43],[51],
[42],[45],[55]]))])}
\texttt{Rest = [] ?}
\texttt{yes}
\texttt{?- example_1.expr(Status, "42+3*(-7", Rest).}
\texttt{Status = repairs([inserted([40],[],line(1,[[52,50],[43],[51],
[42],[40],[45],[55]]))])}
\texttt{Rest = [] ?}
yes

?-

The first example contains a rather bad error that SAGA cannot handle, so it just
gives up and returns information about where the error occurred.

In the second example, SAGA feels more confident that it "knows" what is wrong,
so it fixes the input, continues to parse and just reports what fixes it has performed.
In this case, it deleted one of the plus signs. The third example is similar; a missing
closing parenthesis is inserted at the end of the string.

The information returned as status can be processed by various library routines and
used to present nice error messages. The most high-level error reporting routine is
\texttt{format\_error\_messages}. In its simplest form it may be used like this:

\begin{verbatim}
?- example_1.expr(Status, "42++3*-7", Rest), stderr(Out),
    stdsaga.format_error_messages(Status)-Out.
Line 1: 42++

Deleted marked token

Status = repairs([deleted([43], line(1, [[52,50],[43],[43]]))]),
Rest = [],
Out = _0stream: stderr ?
yes
\end{verbatim}

See Section 3.14 [SAGA Standard Library], page 77, for more information about the
available routines.

\subsection*{3.4.2 Building a Syntax Tree}

The next logical step might be to extend the grammar with semantic values and
actions to build a simple syntax tree. Here we choose a representation suitable for
input to \texttt{is/2}:

\begin{verbatim}
% Build syntax trees for simple arithmetical expressions
:- saga(example_2, expr).
expr(E+T): expr(E), "+", term(T).
expr(T): term(T).
term(T+F): term(T), "*", factor(F).
term(F): factor(F).
factor(E): "(" , expr(E) , ")"
    ; {[0-9]+}{Num} & chars_integer(Num, E).
factor(-F): "-", factor(F).
\end{verbatim}

To call the parser generated from this grammar, we supply an argument for the
attribute:
?- example_2.expr(E, Status, "42+3*-7", Rest), Value is E.
   E = 42+3*(-7)
   Status = true
   Rest = []
   Value = 21 ?
yes
?- 

When we look at the grammar we see some changes. First we note that the form of
the rules has changed. The rule for expr, for example, is actually an extension of
the following rules:

expr: expr, "*", term.
expr: term.

rather than the single rule we used in the first example (see Section 3.4.1 [Parsing
Arithmetical Expressions], page 23). These two forms may be combined in arbitrary
fashions, where a simple example is the rule for factor. The main point of the
second form is to make it possible to unify semantic values in the head of clauses.
However, if you prefer the first form you can equally well use explicit unifications,
as in example 3 below (see Section 3.4.3 [Filters and Definitions], page 27). It is
largely a matter of taste.

Now consider the semantic values. Semantic values associated with a grammar
symbol are written in the same manner as arguments to agents in Agents, and the
values are just constrained variables. In this particular case we use infix and prefix
operators to combine integers into a tree. We also allow accumulator arguments, as
when defining and calling agents in Agents. As a matter of fact, it is often convenient
to read SAGA rules as a form of agents.

Note in particular that terminal symbols (regular expressions) may also have values,
for example '{[0-9]+}' above. In this case the syntactic form is more restricted, to
either one or two arguments, where the first argument represents the text matched
by the expression (as a list of character codes). The second argument, if present,
gives more information about the context of the token. This information is typically
used for reporting errors in the input file, and is described later in the text, see
Section 3.11 [Error Handling], page 69.

Finally, consider the action in the factor rule. A clause and its associated action
are separated by an ampersand, '&'. The code may refer to variables in attributes.
In this case it takes the text matched by '{[0-9]+}' (a sequence of digits) and converts
it to an integer.
3.4.3 Filters and Definitions

In the next example, we introduce definitions and filters. First, we note that the grammar used in the example cannot handle any white space between the tokens, which is a bit unrealistic. To solve this problem, we write a regular expression to match the white space and then supply a filter to throw away the matched white space before it is given to the parser.

Unlike filters, which are necessary to perform certain tasks, definitions is just a matter of convenience. Using complex regular expressions in the middle of rules may make them hard to read. Also, regular expressions may contain complex or common sub-expressions that we would like to name. For this we may define a name to stand for some regular expression. It is important to realize that we name regular expressions, rather than tokens or token types.

To emphasize the importance of using definitions to achieve clarity, we also change the syntax of the numbers to forbid the presence of leading zeroes. This change could equally well be implemented without the use of definitions, but it would be poor style. Finally, we go back to using a single rule for each non-terminal, just to show the difference. (We consider both versions to be equally good style).

```
% Build syntax trees for arithmetical expressions once more
:- saga(example_3, expr).

digit = [0-9].
number = [1-9]digit* | "0".

expr(E) : expr(E1), "+", term(T) & E = E1*T
     ; term(E).
term(T) : term(T1), "*", factor(F) & T = T1*F
     ; factor(T).
factor(F) : "-", factor(F1) & F = -F1
     ; "(" expr(F) ")"
     ; number(Num) & chars_integer(Num, F).

blanks = (" " | \n | \t)*.
[] <- blanks.
```

Definitions, filter rules and grammar rules all have heads and bodies. They use different separators between the heads and the bodies.

- Definitions use `=` as separator
- Filter rules use `<~` as separator
- Grammar rules use `:` as separator
As we shall see later, ordinary Agents agents that use `:`-` or `:=` as separators may also be present in the file.

Definitions must precede their use, so after the header we first define digit to the regular expression `'[0-9]'` that matches a single digit. This definition is only used to improve the readability of the next definition that defines number to be a regular expression matching numbers. This expression is a little more complex than what we have seen so far, so let us examine it in detail. `'[1-9]digit* | "0"'` has two parts, `'[1-9]digit*` and `"0"`, separated by an `'|'`. The vertical bar (`'|'`) means "or", indicating that a number matches either one of two things. The regular expression to the right of the bar is simple and should be familiar by now, `"0"` simply matches the single digit `'0'`. The expression on the left of the bar matches all sequences of numbers that does not begin with `'0'`, namely those that begin with a digit in the range `'[1'` to `'[9]'`, expressed by `'[1-9]'`, followed by zero or more (`'*'`) digits (`'digit'`).

The rules should be easy to read by now. We have moved some of the implicit unifications in the head of rules to explicit unifications in actions. This should also give you a clear idea of how to change the example into a simple calculator by changing unifications into calls to `is/2`.

At the end of the file we have added yet another definition and a filter rule. In this example we made the rather arbitrary choice of putting the filter rule last in the file and the definition that it uses immediately before it. The definition expresses that blanks should be the regular expressions that matches one or more spaces, tabs or newlines. Note the special sequences `'	'` and `'
'` that are used to express the tab character and the newline character, respectively. Also, note that we use parentheses for grouping, since `'*'` binds tighter than `'|'`. The filter definition last in the example says “whenever a token matching blanks is seen, replace it with nothing” where nothing is indicated by an empty list. The general syntax of a filter rule is similar to that of an ordinary rule, with the colon replaced by an arrow. In the filter rule, the lefthand non-terminal of a rule is replaced by a list of terminal symbols (or variables that are unified with terms representing terminals). It may also be a variable, which is later to be unified with a list of terminals. When variables are used, they must be unified with regular expressions by the action associated with the filter. Regular expressions may be written inside agents in the SAGA file, but must then always be inclosed in `'{` and `'}'`, even in simple cases.

### 3.4.4 Lexical Modes

In addition to stripping blanks, we might want to strip comments as well. Comments in Cilly, like in C, begin with `'/`*`' and end with `'*'/`. It is quite possible to write a regular expression matching such comments, even if such an expression gets somewhat complex. The reason for the complexity is that we must write the
expression so that it acts differently on a ‘*’ depending on what follows it. It may be a ‘/’ (in which case we are done), another ‘*’ (which postpones the decision) or some other character (in which case we just continues). One solution is:

```plaintext
comment = "/*/" ("-"*"*"*"*" *["*/"]*) *"*"*" */".
```

The tilde denotes negation and binds tighter than Kleene-star, so ‘-"*"*’ matches a sequence of characters not containing any occurrence of ‘*’. Similarly, ‘-["*/*"]’ matches any character except ‘*’ and ‘/’.

The above way of matching comments is not very common, however. Partly this is due to historical reasons that are not very relevant to SAGA, and partly because of the complex nature of the expression. Usually, you use a device called **lexical modes**\(^{11}\) to handle parts of the syntax that differ from the rest, like comments and character strings. For comments we may change the definition of blanks:

```plaintext
blanks = (', ' | \n | \t)+
       | "/*/"<comment>
       | <comment>"-"*"+
       | <comment>"*"+
       | <comment>"*"*/"<start>.
```

Here, `comment` is a lexical mode. `"/*/"<comment>` means that we should enter this mode after matching ‘/*/’. The expressions prefixed with `<comment>` only match when we are in the lexical mode `comment`. The default lexical mode is called `start`, and the last disjunct is thus used to switch back to normal operation.

**SAGA** also offers **inheritance** among lexical modes. This means that we can define one lexical mode to be an extension of another lexical mode. An example of such a use might be if we want to parse Cilly and another imaginary language Cilly++. Assume that Cilly++ is a strict superset of Cilly. For the purpose of the example, let us suppose that the only difference is that Cilly++ allows comments to begin with ‘//’ and continue to the end of a line. We want to switch between Cilly and Cilly++ in some simple way. One possibility would be to add the following:

```plaintext
:- inherits(plusplus, start).
:- inherits(ppcomment, comment).

ppcomment = <plusplus>"/" "-\n* \n
| <plusplus> "/*/"<ppcomment>
| <ppcomment> "*"*/"<plusplus>.
```

\(\square\) <- ppcomment.

\(^{11}\) Lex calls them “start conditions”.
The directive inherits tells us that all tokens that match in start mode also should match in plusplus mode. We may also have multiple inheritance by giving a list for the second argument. Note that we must prevent comments to cause return to the start mode, so we must duplicate the functionality of the comment mode. However, using yet another inherited mode, we only need to override the parts that change the lexical mode.

All we have to do to implement the dual language parser is to switch between the start mode and the plusplus mode. To switch the lexical mode from an action or a filter, you have to call the agent begin/3 with the reserved variable LexMode in accumulator position. To switch to plusplus mode, you call ‘begin(plusplus)-LexMode’.

3.4.5 From Expressions to Statements

Now we have established the most basic concepts, and we may start to extend the example in the direction of a programming language. First, we take a step up from expressions to statements. We change the header to reflect this:

```prolog
:- saga(cilly, stmt).
```

and we start by adding some rules for assignments, conditional statements and loops. First of all, an imperative programming language should have a way to name things, in particular variables. In the syntax tree, we may choose to represent them as atoms. Therefore we add a few definitions and a rule:

```
digit = [0-9].
letter = [a-z,A-Z,_''].
alphanum = letter | digit.
id = letter alphanum*.
identifier(Name): id(String) & chars_atom(String, Name).
```

The meaning of the definitions should not be hard to understand once you know that ‘[a-z,A-Z,_'']’ stand for any character that is either in the range ‘a’ to ‘z’, or in the range ‘A’ to ‘Z’ or an ‘_’. Of course, we could have settled for just a rule:

```
identifier(Name): ([a-z,A-Z,_''][a-z,A-Z,_'',0-9]*)(String) & chars_atom(String, Name).
```

but that is a bit hard to read.

We want to add the use of variables (actually identifiers, since the parser does not know if an identifier is a variable or not) to the definition of expressions. While we are at it, we throw in assignment and comparison operators as well. Cilly considers them to be just ordinary expressions, just like C does. Only a certain kind of
expressions, called \textit{l-values}, may be assigned to. The new definition of \texttt{expr} then becomes:

\begin{verbatim}
expr(E1=E2): lvalue(E1), ",=", expr(E2).
expr(E): relational(E).
relational(E1<E2): additive(E1), ",<", additive(E2).
relational(E1=E2): additive(E1), ",==", additive(E2).
relational(E): additive(E).
additive(E1+E2): additive(E1), ",+", term(E2).
additive(E): term(E).
term(E1*E2): term(E1), ",*", factor(E2).
term(E): factor(E).
factor(-E): ",-", factor(E).
factor(E): identifier(E)
    ; ",(" expr(E), ")".
    ; constant(E).
constant(C): number(Num) & chars_integer(Num, C).
lvalue(L): identifier(L).
\end{verbatim}

The next step is to give rules for some common statements:

\begin{verbatim}
stmt(E): expr(E), ",;".
stmt(while(C, S)): "while" , "(" expr(C), ")", stmt(S).
stmt(if(C, S)): "if" , "(" expr(C), ")", stmt(S).
stmt(if(C, S1, S2)):
    "if" , "(" expr(C), ")", stmt(S1), "else", stmt(S2).
stmt(Ss): "{", stmt_list(Ss), "}".
stmt_list([]): 
stmt_list([S|Ss]): stmt(S), stmt_list(Ss).
\end{verbatim}

These rules contain several noteworthy points and one problem. First of all, note that we use \texttt{"{}\texttt{"} to denote the empty production.

3.4.5.1 Conflicts

The problem with the grammar above is that it is ambiguous. If you run it through SAGA you get the following error message:

\begin{verbatim}
CONFLICT. Assuming input of the form:
"if" "(" expr ")" stmt
With "else" as the next token there is a conflict between:
[shift]  'try a longer match'
1 shift/reduce & 0 reduce/reduce conflict(s)
\end{verbatim}

\textit{Shift} means that we should read more input and find a longer match for some rule. \textit{Reduce} means that we should apply one of the grammar rules on (a part of) the input we have seen. For more information about shifts and reductions, see Section 3.10 [The SAGA Parsing Algorithm], page 64.
There seems to be a problem with ‘else’. At first this seems quite strange, unless you have heard about the dangling else. It seems like there ought not to be any problem. Since an ‘else’ must belong to an ‘if’, we should always shift: Go for the longer match. But the example input that SAGA wrote out is, while accurate, not telling the whole story. Consider the following input:

```c
if(a)
   if(b) c = 0;
else
   c = 1;
```

This example has two different parse trees, one that associates the ‘else’ with the first ‘if’, as the indentation hints, and the other associates the ‘else’ with the second ‘if’, which is actually the sensible reading. Sometimes SAGA’s conflict reporting is a little bit terse, but that is nearly always a feature, since it points directly to the problem and tries to give a suitable amount\(^\text{12}\) of context.

So what do we do about this problem, now that we are aware of its origin? One possibility is to rewrite the grammar to reflect the desired parse. A simpler and better way is to tell SAGA that you know about the conflict and would like to resolve it in favour of the shift:

```c
stmt(if(C, S1, S2)):
   "if", "(", expr(C), ")", stmt(S1), "else"$shift, stmt(S2).
```

The annotation ‘$shift’ tells SAGA that you want the ‘else’ to be shifted when it is involved in a conflict.

### 3.4.5.2 Left Recursion is the Right Recursion

Consider the following fragment of the above example:

```c
stmt(Ss): "{", stmt_list(Ss), "}"
stmt_list([]): {}
stmt_list([S|Ss]): stmt(S), stmt_list(Ss).
```

We use the rules for `stmt_list` to match a sequence of statements and build a list. Each element in the list is the result of matching a statement by the rules for `stmt`. The rules for `stmt_list` is very similar to the clauses of a list-recursive agent.

---

\(^{12}\) Many other tools dump all the information they have about the grammar, which can be several megabytes of text, and thus drown you in unwanted technical information.
Note that the recursive call is at the right end of the rule. In grammar terminology we say that stmt_list is right recursive. Left recursion is when the recursive call is at left end of the rule. Ignoring the arguments to the rule for now, a left recursive definition of stmt_list can look like this:

```plaintext
stmt_list: {}.
stmt_list: stmt_list, stmt.
```

Many other tools have a strong bias in favour of either left or right recursion. SAGA handles both reasonably well, but left recursion is slightly preferable in most cases. For instance, error recovery usually works much better with left recursive rules than with right recursive ones. However, adding list arguments to the left recursive example is a little more complicated than for the right recursive counterpart. Apart from error recovery benefits, there is another strong reason to learn the technique of adding list arguments to left recursive rules. Left recursion is the most common form in existing grammars for other generators. Porting such grammars to SAGA would be hard without mastering the left recursion.

The trick is to exploit the logical variables of Agents by using a d-list technique. We pass around two arguments instead of one, where the second one represents the tail of the list. For example:

```plaintext
stmt(Ss): "{" stmt_list(Ss, []), "}".
stmt_list(Ss, Ss): {}.
stmt_list(Ss0, Ss): stmt_list(Ss0, Ss1), stmt(S) & Ss1 = [S|Ss].
```

Using accumulator syntax, it can be expressed in a nice way:

```plaintext
stmt_list-Ss: {}.
stmt_list-Ss: stmt_list-Ss, stmt(S) & [S]-Ss.
```

The Agents goal ‘[S]-Ss’ is accumulator notation (in Agents rather than SAGA) corresponding to the ‘Ss1 = [S|Ss]’ in the example above.

In the next few sections, we will see that SAGA also offers an even more elegant way to do things of this kind.

### 3.4.6 Using Operator Precedence

It can be convenient to use an ambiguous grammar and resolve the conflicts that arise in a more direct fashion than by rewriting the grammar. This is also true for parts of the grammar that are based on operators. Let us rewrite the rules for expr in the following (ambiguous) manner:
expr(E1=E2): lvalue(E1), ";" =", expr(E2).
expr(E1<E2): expr(E1), "<", expr(E2).
expr(E1==E2): expr(E1), ";==", expr(E2).
expr(E1+E2): expr(E1), ";+", expr(E2).
expr(E1*E2): expr(E1), ";*", expr(E2).
expr(-E): ";-", expr(E).
expr(E): identifier(E)
    ; "(" expr(E) ",")
    ; constant(E).
constant(C): number(Num) & chars_integer(Num, C).
lvalue(L): identifier(L).

We can now add information about the precedence and associativity of operators to disambiguate it.

:- op(800, right, ";=").
:- op(700, nonassoc, ["==", "<"]).
:- op(500, left, ";+").
:- op(400, left, ";*").
:- op(300, right, ";-").

The `op` directive is used to express the desired precedence information. The first argument is a number describing how tight the operator should bind, where smaller numbers bind tighter than larger ones. Only the relative size of the numbers matters, so we could equally well have chosen the numbers 5, 4, 3, 2, 1 instead of the numbers in the example. The second argument is the associativity, either `left`, `right` or `nonassoc`. Associativity is used to resolve the ambiguities that result from having several operators of the same precedence in sequence. In this case, `left` means that the leftmost operator should bind tighter, `right` that the rightmost should bind tighter, and `nonassoc` means that the situation cannot occur in legal input. In the latter case an error should be raised. The third argument to the `op` directive is the operator, or a list of operators, that should receive the stated precedence.

Picking operator precedences is usually a fairly straightforward process, but there is one common complication that we have avoided so far. It is quite common to have operators with precedences which depends on the context of the operators. The most well-known case is that of binary and unary `-`. Suppose we would like to add binary `-` that should have the same precedence and associativity as binary `+`, i.e. left associative with precedence 500. To do this, you must declare the correct precedence for the binary case:

:- op(500, left, ["+", ";-"]).

and use an annotation to change the rule for unary `-` to:

expr(-E): ";-"$op(300, right), expr(E).
The latter construction allows us to express that this particular occurrence of ‘-’ represents a right-associative operator with precedence 300. All other occurrences, including yet-to-be-classified ones like the look-ahead token, should be treated according to the ‘op’ directive. Note that it would not work to keep the old ‘op’ directive and change the binary rule, since this would give the wrong behaviour for other binary operators. The reason is that if an operator is look-ahead after any of the given rules it must be a binary operator\textsuperscript{13}. Therefore the precedences of binary operators affect the behaviour of other binary operators, and thus the precedence of an operator that is out of context (look-ahead) should be taken from the binary case and not the unary. (If you found that hard to follow, don’t worry about it, just trust us on this one.)

3.4.7 Using EBNF

While we are working on simplifications, we should look at the rules for the non-terminal \texttt{stmt\_list} and how we use it. Sequence rules of this kind are very common in grammars, and we would like to avoid writing similar rules over and over again. To simplify the writing of the grammar, SAGA supports a number of constructions that can be used instead of writing new rules. Such extensions are often referred to as Extended BNF, or shorter EBNF. While this is slightly strange unless the grammar actually is in BNF notation, we follow suit. We provide EBNF for zero-or-more, one-or-more and zero-or-one occurrences of an expression, and also for embedded disjunctions, conjunctions and actions. For example, we could dispose of the rule for \texttt{stmt\_list} if we change the calling rule for \texttt{stmt} to:

\begin{verbatim}
stmt(Ss): "{" , stmt*(Ss) , "}".
\end{verbatim}

The postfix ‘*’ stands for zero-or-more repetitions of \texttt{stmt}. The argument to the right of the ‘*’ (Ss) indicates a list in which to collect the results of the calls to \texttt{stmt}. In this simple case, \texttt{stmt} should be called with one argument.

As an alternative to building lists, the EBNF rules may also pass variables around. Assume we just wanted to count the statements, without actually building any list of them. Then we might write rules that looked like this:

\begin{verbatim}
count\_statements-C: count\_stmt-C*.
count\_stmt-C: stmt(_- ) & inc-C.
\end{verbatim}

When the accumulator argument is to the left of the ‘*’, it means that it should be passed between iterations. The above roughly corresponds to the following rules:

\textsuperscript{13} Or a postfix unary operator.
count_statements-C: count_stmt_star-C.
count_stmt_star-C: count_stmt_star-C, count_stmt-C.
count_stmt_star-C: {).
count_stmt-C: stmt(.) & inc-C.

As hinted by the above examples, the EBNF notation is chosen to resemble the notation used in regular expressions. We support the 'operator for one or several occurrences and the '?' operator for zero or one occurrence. Both behave analogous to the '*' operator. In particular, 'stmt?(Ss)' builds a list (which is either the empty list or a singleton list). See Section 3.7.3 [EBNF], page 56, for details on the EBNF notation supported by SAGA.

For convenience, we show the entire grammar so far:

```
:- saga(cilly, stmt).

digit = [0-9].
letter = [a-zA-Z,'_'].
alnum = letter | digit.
id = letter alphanum*.
number = [1-9]digit* | "0".

stmt(E): expr(E), ",".
stmt(while(C, S)): "while", ",", expr(C), ")", stmt(S).
stmt(if(C, S)): "if", ",", expr(C), ")", stmt(S).
stmt(if(C, S1, S2)):
    "if", ",", expr(C), ")", stmt(S1), "else"$shift, stmt(S2).
stmt(Ss): ",", stmt*(Ss), ")".

:- op(800, right, "=").
:- op(700, nonassoc, ["==", "<"]).
:- op(500, left, ["+", "-", "*", "/"]).
:- op(400, left, "+", "-", "*", "/").

expr(E1=E2): lvalue(E1), ",=", expr(E2).
expr(E1<E2): expr(E1), ",<", expr(E2).
expr(E1==E2): expr(E1), ",==", expr(E2).
expr(E1+E2): expr(E1), ",+", expr(E2).
expr(E1-E2): expr(E1), ",-", expr(E2).
expr(E1*E2): expr(E1), ",*", expr(E2).
expr(-E): "-"$op(300, right), expr(E).
expr(E): identifier(E)
    ; ",", expr(E), ")"
    ; constant(E).

constant(C): number(Num) & chars_integer(Num, C).
lvalue(L): identifier(L).

identifier(Name): id(String) & chars_atom(String, Name).
```
blanks = (" \
| "+ | <comment> | ";/*"<comment> | <comment>"/*"+ | <comment>"/*"+ | <comment>"/*"+"<start>.
[] <- blanks.

3.4.8 More EBNF

Most imperative languages are typed, and type declarations may in some cases cause tricky situations. Let us first add some simple type declarations to Cilly.

The declarations contain the basic types char for characters and int for integers, the modifiers signed, unsigned, long and short and the storage classes auto, static and register. We also allow the optional qualifier const. Finally, we want pointers to all types. The basic type must be given, but the modifiers and storage class is optional. At most one storage class may be given, but several modifiers are allowed, like 'unsigned short int'. However, signed and unsigned are mutually exclusive, as are short and long. An omitted storage class is the same as an explicit auto.

If this does not strike you as very simple, consider that C has considerably more complex type declarations, not to mention C++.

First, we have a tradeoff to consider. We could certainly write the grammar so that signed and unsigned mutually exclude one another, and likewise for long and short. When writing the rules we must take care to avoid ambiguity, but it could be done. We might do it like this, for example:

modifiers(LS, SU)
  : ls(LS), su(SU)
  ; su(SU), ls(LS)
  ; ls(LS) & SU=signed
  ; su(SU) & LS = medium
  ; {} & LS = medium, SU = signed.
ls(long) : "long".
ls(short) : "short".
su(signed) : "signed".
su(unsigned) : "unsigned".

This is messy, and it would be even more so with a larger number of groups with mutual exclusion. Instead, we are going to use the common method to solve this problem, namely to ignore the mutual exclusion in the grammar, and then explicitly check the syntax tree for correctness of such things later on. In many cases it is difficult to strike the correct balance between complicating the grammar and complicating the processing of the result. A rule of thumb is to make as much work
as possible in the grammar, except if this causes a combinatorical explosion. Note that the mutual exclusion of \texttt{auto}, \texttt{register} and \texttt{static} is easy to handle in the grammar.

In Cilly, unlike C but like C++, we consider a type declaration to be a statement. We will add a few rules to the grammar. These rules use EBNF quite heavily, partly because it is convenient, partly to demonstrate the possibilities. We could certainly have used EBNF even more, writing the \texttt{storage\_class}, the \texttt{modifier} and the \texttt{basic\_type} rules inline, but this would be very hard to read. This is quite easy to read when you get familiar with the EBNF, but some people might prefer to have a separate rule for the first item and maybe even one for the pointers. It is largely a matter of taste.

\begin{verbatim}
stmt(def(I,T)) : type_declaration(I, T), ";".

\texttt{type\_declaration(I, T) :}
\texttt{ X"\texttt{\texttt{\texttt{\texttt{\texttt{const}}} \& X = const}}"(C),}
\texttt{ storage\_class(S),}
\texttt{ modifier*(Ms),}
\texttt{ basic\_type(B),}
\texttt{ ST/(_Simple),}
\texttt{ deref-ST*,}
\texttt{ identifier(I)}
\texttt{ & Simple = type(B, Ms, S, C, Xs),}
\texttt{ T = ST.}
\end{verbatim}

\texttt{storage\_class(register) : "register".}
\texttt{storage\_class(static) : "static".}
\texttt{storage\_class(auto) : "auto" ; {}.}

\texttt{modifier(short) : "short".}
\texttt{modifier(long) : "long".}
\texttt{modifier(signed) : "signed".}
\texttt{modifier(unsigned) : "unsigned".}

\texttt{basic\_type(int) : "int".}
\texttt{basic\_type(char) : "char".}

\texttt{deref(T, pointer\_to(T)) : ".".}

The rule for \texttt{type\_declaration} uses some interesting constructions that need some explanation. First we have:

\begin{verbatim}
X"\texttt{\texttt{\texttt{\texttt{\texttt{const}}} \& X = const}}"(C)
\end{verbatim}

This uses no less than three different EBNF concepts: \textit{Embedded actions}, \textit{abstraction} and \textit{optional rules}. First, `("const" \& X = const)` is equivalent to `\texttt{\texttt{\texttt{\texttt{\texttt{foo}}}\texttt{(X)}}}` provided we add a rule `\texttt{foo(X) : "const" \& X = const.}`. This is quite a natural
operation; embedding an action inside a rule. It is important to realize how it works, since misuse may cause conflicts in an otherwise working grammar. Second, we introduce the abstraction, in this case ‘\(\text{\texttt{\textbackslash x\{"\texttt{const}\ & \ x = \texttt{const}\}}\)’. This has the effect of creating a new rule with one argument, \(\text{x,}\) that may be used by other EBNF operators. In this case, it just changes the imaginary ‘\texttt{foo(x)}’ above into ‘\texttt{foo}’, but in general it may turn a call with many arguments (more generally, an expression with free variables) into a one argument call, much like the use of abstractions in Agents. From a practical point of view, they provide a way to choose one variable as the “mapped” variable in EBNF calls. Finally, the optional EBNF rule (indicated by ‘?’) matches zero or one occurrence and puts the matches into a list, as mentioned before. Without EBNF, the call might be written like this:

\[
\text{const\_opt(C)}
\]

if we add the following rules:

\[
\begin{align*}
\text{const\_opt([X])}: & \ "\text{const}\ & \ x = \text{const}. \\
\text{const\_opt([])}: & \ {}\{\}.
\end{align*}
\]

The next thing new is the ‘\texttt{ST/(\_,\texttt{Simple})}’ statement. To understand its use we first look at the ‘\texttt{derref\_ST*}’ call. Recall that it iterates over the accumulator \texttt{ST}, gradually changing it. To make this work you must have some way to give \texttt{ST} the correct initial value and a way to access the final value. As always with accumulator syntax, the last occurrence of \texttt{ST} before the call holds the initial value, and the first occurrence after the call holds the final value. In ordinary Agents programs you can always place your goals in the appropriate order, or insert some unifications in the right places. In a \texttt{SAGA} rule this is not so easy, since the actions occur after the grammar part. Of course, you might embed actions to simulate this, but it may cause unexpected and unnecessary conflicts to crop up where you least expect them. The reason for this is that such embeddings actually changes the grammar by introducing new rules. Instead, we offer special constructs of the following forms:

\texttt{Acc/(Before, After)}

The main form.

\texttt{Acc/Inserted}

The same as ‘\texttt{Acc/(Inserted, Inserted)}’.

These are called \textit{slashing} actions (considering both their effect on the accumulator chain and their syntax). The use of a slashing action does not change the parse proper, but only the way the accumulator variables expand. Consider the main form:

\texttt{Acc/(Before, After)}
This behaves quite similar to the call:

\[ \text{slash}(\text{Before}, \text{After}) - \text{Acc} \]

if \text{slash} is defined by the following rule:

\[ \text{slash}(X, Y, X, Y) : \{ \}. \]

\text{Before} is unified with the current value of \text{Acc} and \text{After} is used as the new value of \text{Acc}. The difference is that the \text{slash} rule above does change the behaviour of the parse, since it may introduce conflicts. Another difference is that the call actually corresponds to:

\[ \text{slash}(\text{Before}, \text{Temp}) - \text{Acc} \]

where \text{Temp} is unified with \text{After} as the last goal in the rule. This difference is noticeable only when \text{After} is an accumulator argument, causing the final value of the accumulator to be used. This behaviour makes ‘\text{Acc/(Inserted, Inserted)}’ a particularly interesting case, with the effect of linking in one accumulator chain inside another. This case is common enough to have its own syntax (‘\text{Acc/Inserted}’), especially since the main form looks so bizarre in this case.

### 3.4.9 Global Accumulators

Now that we have types in the Cilly language, we may want a symbol table to keep information about declared types. We may for example annotate all the identifiers with their types.

One way to accomplish this that we might contemplate is to augment almost every rule in the grammar with an accumulator argument to hold this table. This approach have two major drawbacks. First, it is a pain to write such accumulators everywhere. Second, no such information could gracefully survive a parse error, which makes it very unpractical for symbol tables and the like in a real application.

So, to deal with these situations, SAGA allows the user to define global accumulators. They are automatically threaded through the entire execution, in the order the reductions are performed. This order corresponds to a post-order traversal of the parse tree. Of course, you may use global accumulators for any purpose you like, but we recommend use of semantic values in most cases. Global accumulators should only be used when you need data to be persistent with respect to error recovery, when you need to pass the information to a filter (in which case it is the only possibility), when you want to trace the execution order of the parser for some reason or when you want to add temporary information for debugging of your parser.
The global accumulators are declared as a third argument to the header, either as a single accumulator or as a list of accumulators. In our case the header becomes:

```prolog
:- saga(cilly, stmt, SymbolTable).
```

The initial and final values of the global accumulators are passed as lists to the parser. In this case, assuming that we want the table to be an association list from the Agents `assoc` library, a typical call may look like:

```prolog
assoc.empty(Empty),
cilly.stmt(Program, Status, [Empty], [Final], Input, RestInput)
```

The third argument is a list of initial values for the global accumulators, in this case an empty association list. The fourth argument is a corresponding list for the final values after the execution.

To annotate the identifiers with their types we change the rule for `identifier` and one of the rules for `stmt`. Before adding the type annotations these rules look like:

```prolog
identifier(Name) : id(String) & chars_atom(String, Name).
stmt(def(I,T)) : type_declaration(I, T), ";".
```

We rewrite them in the following way:

```prolog
identifier(AnnName):
  id(String)
  & chars_atom(String, Name),
  assoc.get_def(Name, Name, AnnName)-SymbolTable.

stmt(def(I,T)) : type_declaration(I, T), ";"
  & assoc.put(I, I$T)-SymbolTable.
```

If an identifier is present in the symbol table it is annotated with its type. If it is not in the table we do not annotate it at all. We could also have given identifiers a default type if they were missing from the table, but this would be a bit strange on names for types and the like. We use the agent `get_def/5` to do the annotation in a cute way. The second argument is a default value to be used if the key is not present in the table. To make this work, we store the annotated identifiers in the table rather than types.

The global accumulator is used in the actions part of rules as any ordinary accumulator variable. Global accumulators may also be used in filter actions. They may not be used directly in Agents agents. This is not a problem, since agents are called from either filter actions or grammar rules. Thus, it is easy to pass the accumulators as arguments to them.
3.4.10 Handling the Problem with typedef

Now the time has come to extend Cilly with the hardest-to-parse construct in C, the innocent-looking typedef construction. Let us look at a first attempt to introduce the typedef, to see the problem. We settle for a simple version of the construct:

\[
\text{stmt}(\text{typedef}(I, T)): \"\text{typedef}\", \text{type}\_\text{declaration}(I, T), \";\".
\]

\[
\text{type}\_\text{declaration}(I, T):
\]
\[
\text{typedefname}(ST), \text{deref-}\text{ST}, \text{identifier}(I)
\]
\[
& T = ST.
\]

\[
\text{typedefname}(I): \text{identifier}(I).
\]

The problem is the rule for type\_declaration, in conjunction with the other rules for statements. Consider the input fragment ‘foo * bar;’. If it was preceded (arbitrarily far away) by ‘typedef int foo;’ or something similar, we expect this to be a type declaration, declaring bar to be a pointer to a foo. If no such typedef preceded the occurrence, then the fragment simply represents the code that evaluates the arithmetical product of foo and bar. While the latter interpretation may seem slightly silly, it is legal Cilly (as well as legal C).

It is impossible to write a context-free grammar for this construction, but we certainly want to be able to handle it anyway. The basic idea is to use a filter to replace all identifiers that have been typedef'ed with another token type. The first thing to realize is that we must store information about what names have been typedef'ed in some sort of symbol table. This table must be in the form of a global accumulator, because this information must be passed to a filter. In this case we choose to add another global accumulator rather than using the existing table for more than one purpose. We change the header to the following:

\[
\text{:- saga(cilly, stmt, [SymbolTable, TypeDefs]).}
\]

Let us define a suitable filter and a working version of the grammar rules above. There are of course several possibilities. One possibility would be to insert typedef'ed identifiers (associated with any value at all) into the table and then use the following filter:

\[
\text{typedefname} = \leftrightarrow\text{typedefed}.
\]
\[
[T] \leftarrow \text{id}(I) & \text{check-if-typedef}(I, T)-\text{TypeDefs}.
\]

\[
\text{check-if-typedef}(I, T)-\text{TypeDefs} :-
\]
\[
\text{assoc.get}(I, \_)-\text{TypeDefs}
\]
\[
\rightarrow T = \{\text{typedefname}\}.
\]

\[
\text{check-if-typedef}(I, T)-\text{TypeDefs} :-
\]
\[
\rightarrow T = \{\text{id}\}.
\]
There are some interesting points to note. First, we need to introduce a new token that does not actually occur on the input, but only gets inserted by the filter. SAGA allows you to create a new token by prepending `<>` to an identifier, `typedefed` in this case. The intuition behind this syntax is that of an empty lexical mode, so it essentially says "in no lexical mode, match ...".

Second, note that `check_if typedef` is an ordinary Agents agents. SAGA allows you to mix such agents with the rules in the file. Third, note that we can refer to regular expressions from such agents as well as from actions. This usage requires that we enclose the regular expression within braces. The regular expressions are represented as ordinary Agents records and therefore they can be manipulated like any other Agents data structure. In particular, they may be stored in other data structures. Thus we may choose a slightly more stream-lined filter if we store the regular expression in the symbol table. Then the example looks like this:

\[
\begin{align*}
typedefname &= <>\text{typedefed}. \\
T &= \text{id(I)} \\
& \quad & \text{& \ char_atom(I, Id),} \\
& \quad & \text{assoc.get_def(I, \{id\}, T)-TypeDefs.} \\
\text{stmt(typedef(I, T)): typedef_declaration(I, T), }; \\
& \quad & \text{";".} \\
\text{typedef_declaration(I, T):} \\
& \quad & \text{"typedef", type_declaration(I, T)} \\
& \quad & \text{& assoc.put(I, \{typedefname\})-TypeDefs.} \\
\text{type_declaration(I, T):} \\
& \quad & \text{typedefname(ST), deref-ST*, identifier(I)} \\
& \quad & \text{& T = ST.}
\end{align*}
\]

There is one crucial and very subtle twist to this grammar. If we were to move the ";" to the `typedef_declaration` rule, then the grammar would not work properly. The reason for this is that we use one token of look-ahead (see Section 3.3.3.2 [Grammar Classes], page 19) to parse the input. This look-ahead token has passed through the filter before we apply the rule, so it will not be filtered in the correct way. With the above grammar this is no problem, since the look-ahead token will be the semi-colon terminating the statement.

### 3.5 Structure of the Grammar File

A SAGA grammar file consists of directives, definitions of named regular expression, grammar rules, filter rules and Agents agents. Regarding definitions, see Section 3.6.3 [Definitions], page 50. Regarding grammar rules, see Section 3.7 [Grammar Rules], page 51. Regarding filters, see Section 3.8 [Filters], page 59.
3.5.1 Directives

The possible directives are the following:

:- saga(module-name, top-symbol).
:- saga(module-name, top-symbol, globals).

The module-name and top-symbols are identifiers (unquoted Agents atoms) and globals is either an Agents variable or an Agents list of Agents variables. This directive must be present at the beginning of all SAGA files and it is therefore called the header. The module-name gives the name of the module of the generated parser. The top-symbol should be the top (or start) non-terminal corresponding to the entire input to be matched. The globals is a list of all the global accumulators used in the file. The globals may be omitted. It defaults to the empty list (‘[]’). The global accumulators are accumulator variables that are automatically chained by SAGA throughout the execution. These globals are given initial values when you invoke the parser and they are accessible from filters and actions. They roughly behaves as global variables in imperative languages. For a discussion of when and how they should be used along with some examples, see Section 3.4.9 [Global Accumulators], page 40.

:- error(handler-abstraction).

The handler-abstraction is an Agents abstraction that should be called to handle parse errors. See Section 3.11 [Error Handling], page 69.

:- inherits(derived-mode, base-modes).

The derived-mode is an identifier and base-modes is either an identifier or a list of identifiers. This directive is used to declare that all tokens that match in the lexical modes in base-modes should also match in the lexical mode derived-mode. See Section 3.6.4 [Inheritance], page 50.

:- op(precedence, associativity, operators).

The precedence is an integer, associativity is either ‘left’, ‘right’ or ‘nonassoc’ and operators is either a token or a list of tokens. This directive is used to give precedences to operators in order to disambiguate an ambiguous grammar. A numerically smaller number indicates a tighter binding operator. The associativity comes into play when we have two conflicting operators with the same precedence. In this situation left means that the leftmost operator should bind tighter, right that the rightmost should bind tighter. The nonassoc associativity indicates that this situation should cause a parse error. See Section 3.10.3.2 [Operator Precedence], page 66 for technical details.
3.6 Lexical Analysis

The description of the lexical analysis phase (see Section 3.3.1 [Lexical Analysis], page 15, for an introduction to the basic concepts of lexical analysis) in SAGA is based on regular expressions (see Section 3.3.1.2 [Regular Expressions], page 16, for an introduction to regular expressions). The differences between the SAGA concepts and the mathematical notations are partly due to an intention to provide convenient shorthands for common expression types, and partly due to the different goals of automata theory and practical parsing.

In automata theory, the goal of an regular expression is to decide if a certain string belongs to a set of strings (a language) or not. In practical parsing, we have to divide a long input into small pieces that belong to different token types (i.e. different languages). It is often the case that several tokenizations of the same input (that all respect the languages for different token types) are feasible, so we need some extra rules to disambiguate the tokenization. Also, it is desirable to be able to use some finite amount of information of the previous tokens and the remaining input to guide the tokenization.

3.6.1 Tokenization Basics

Let us begin by outlining the way a SAGA lexical analyzer work. This may appear to be a little more subtle in SAGA than in many other tools. This is because the description of the lexical analyzer is syntactically distributed over the entire grammar file rather than being confined to a separate section. However, a SAGA lexical analyzer has simpler semantics than most other tools since we disallow general actions to be associated with the matching.

Except for definitions of named regular expressions, every regular expression present in the SAGA grammar file corresponds to a particular token type. In general the regular expressions that correspond to token types should be inclosed in braces ('{'} and '{'). In the case of literal strings without arguments and defined names (with or without arguments), the brackets may be omitted in grammar rules and directives. Two regular expressions are regarded as the same regular expression if they have the same parse tree — if they look essentially alike. Definitions that give symbolic names to regular expressions are encouraged. For a correct understanding of the semantics, it is important to realize that we are naming regular expressions rather than defining tokens in order to understand the semantics correctly.
3.6.1.1 Lexical Modes

A SAGA generated lexical analyzer may have several lexical modes. Each lexical mode acts like a separate tokenizer. The current lexical mode may be changed when certain tokens match or it may be changed explicitly from an action or a filter.

All regular expressions that correspond to token types are confined to match in one or more specific lexical modes. A regular expression may include an explicit statement saying that it corresponds to a certain set of lexical modes. If such a statement is omitted, the lexical mode defaults to start which is the lexical mode the tokenizer is in initially. In addition to such explicit statements it is possible to declare that all the regular expressions from a certain mode should be included in another mode as well. We say that the larger mode inherits from the smaller mode, see Section 3.6.4 [Inheritance], page 50.

To change the current lexical mode from an action or a filter, you apply the agent 

\texttt{begin/3} to the predefined global accumulator \texttt{LexMode}. See Section 3.9.3.3 [Standard Agents], page 64.

3.6.1.2 How the Input is Matched

Each time the lexical analyzer is asked to produce a token, all the regular expressions that are associated with token types in the current lexical mode are considered. The lexical analyzer then finds the longest non-empty prefix of the remaining input that is matched by one of these regular expressions. The choice of the longest prefix is a strategy referred to as the longest match rule. If no non-empty prefix exists then a single character is matched by a default rule. When the default rule is used a parse error results (except in case of non-determinism where it causes a parse failure instead).

When the longest prefix matches more than one token type\textsuperscript{14} we resort to two additional rules. First we use the longest context rule. This rule is only relevant for regular expressions that use trailing context. It says that the match that has the longest trailing context should be preferred. In particular, a match with a trailing context is preferred over one without. Note that the context is not included when the longest match is selected. The second rule is called the subset rule.

The subset rule states that if all legal strings in the language corresponding to the token type A also belong to the language corresponding to the token type B (along

\textsuperscript{14} Actually a single matching token type may also cause problems. This can happen when we request a change of lexical mode inside a disjunction.
with at least one other string), then $A$ should be preferred over $B$. Thus, the (strict) subset language is preferred. The rationale for this is that special cases should have higher precedence than more general ones. The opposite approach would be unreasonable since it would prohibit one of the token types from ever matching. The subset rule correctly handles the most common cause of conflicts: Reserved words as a special case of identifiers. In the presence of inheritance, the subset rule is interpreted across lexical modes which achieves the effect of overriding. See Section 3.6.4 [Inheritance], page 50.

When these rules are not sufficient to choose between the expressions (for all possible strings) the generator declares an error and forces the grammar writer to resolve the conflicts manually. For example:

```plaintext
token1 = "fo" _ .
token2 = _ "oo" .
```

Here `token1` matches any three letter string that begins with ‘fo’ and `token2` matches any three letter string that ends with ‘oo’. In this case neither `token1` nor `token2` is a subset token type of the other. Since the intersection is non-empty (both matches ‘foo’), we have a conflict. Assume that we wish ‘foo’ to be matched by `token1`. Then we may rewrite the example as:

```plaintext
token1 = "fo" _ .
token2 = token1 | _ "oo" .
```

This makes `token1` a strict subset of `token2` and thus the subset rule solves the conflict.

### 3.6.2 Regular Expressions

In the following, we let $R$, $R1$ and $R2$ denote regular expressions and $N$ and $M$ denote integers. To denote sequences of characters we use `string`. Identifiers begin with lower case letters and consist of letters, digits and underscores and they are denoted by `identifier`. Other meta-level expressions are explained as they are encountered.

To fully understand the meaning of a regular expression you must first know how it is tokenized. SAGA uses the same tokenization rules as Agents does, making it simple for the grammar writer to read, write and understand the expressions, given a working knowledge of Agents. Compared with tools like Lex, the SAGA syntax is more regular and in many cases it is a little more verbose.

Here follows a list of all the legal ways to build regular expressions and their semantics. Actually, there are some limitations on the use of some of the constructs, see Section 3.6.2.1 [Restrictions on Syntax], page 50. The patterns are listed in order of precedence, with the tightest binding constructs listed first.
"string"

\texttt{\textasciitilde string} Matches the characters in \texttt{string}. Since we use the tokenization rules of Agents, we inherit the rules for escaped characters in strings and quoted atoms. The construct is called \emph{literal string}.

\texttt{-} The underscore character matches any single character. In contrast to \texttt{.\textasciitilde} in Lex and other Unix tools, \texttt{-\textasciitilde} matches any character \emph{including newline}.

\texttt{\textbackslash N} Matches the single character with the character code \texttt{N}, where \texttt{N} is interpreted as an octal number.

\texttt{\#N} Matches the single character with the character code \texttt{N}, where \texttt{N} is any Agents integer and is interpreted as such.

\texttt{\textbackslash identifier} Matches the single character denoted by \texttt{identifier}. The most well-known instances are \texttt{\textbackslash n} that denotes the newline character and \texttt{\textbackslash t} that denotes the tabulator character, but any identifier may be used. The character codes that the names denote may be given with command line options when invoking the SAGA tool.

\texttt{identifier} Matches the regular expression that \texttt{identifier} is defined to. We require that such a definition occurs prior to the use, see Section 3.6.3 [Definitions], page 50.

\texttt{[items]} Matches a single character that match any of the items in the comma-separated list \texttt{items}. An \texttt{item} may be one of the following:

\texttt{string} Matches any of the characters in \texttt{string}. We restrict \texttt{string} to contain alphanumeric characters and underscores. Since we use the tokenization rules of Agents, we also require that if the first character is a digit, then all the other characters should be digits as well.

\texttt{\textasciitilde string} \texttt{\textquoteleft string\textquoteright} Matches any of the characters in \texttt{string}.

\texttt{\textbackslash N} \texttt{\#N} \texttt{\textbackslash identifier} Matches a single character in the same way as above.

An item may also have the form \texttt{\textquoteleft item1-item2\textquoteright}, where the two items give endpoints for a range that the character should be in. In this case the two items must each denote specific single characters. The construct is called \emph{character set}.

\texttt{(R)} Matches \texttt{R} and is used to override precedence.

\texttt{\textasciicircum R} Matches any single character that is \emph{not} matched by \texttt{R}. We restrict \texttt{R} to match only single characters given by character sets, other negations, disjunctions and strings of length one. The construct is called \emph{negation}. 
\( R^* \) Matches zero, one or more occurrences of \( R \).

\( R^+ \) Matches one or more occurrences of \( R \).

\( R? \) Matches zero or one occurrence of \( R \).

\( R\{N\} \) Matches exactly \( N \) occurrences of \( R \).

\( R\{N,\} \) Matches at least \( N \) occurrences of \( R \).

\( R\{N,M\} \) Matches at least \( N \) and at most \( M \) occurrences of \( R \).

\( R1 \ R2 \) Matches \( R1 \) followed by \( R2 \).

\( R1/R2 \) Matches \( R1 \) if \( R2 \) match a prefix of the remaining input after the match. The part matched by \( R2 \) is called the trailing context. The construct itself is also called trailing context. The trailing part does not count when the longest match is determined except when choosing between two equally long matches. This construct is syntactically restricted.

\( R$ \) Matches \( R \) if it is followed by a newline, thus equivalent to \( 'R/\n' \). This construct is syntactically restricted.

\( ^R \) Matches \( R \) if it occurs at the beginning of a line. This construct is syntactically restricted.

\(<\textit{identifier-list}>R\) Matches \( R \) in any of the lexical modes in \( \textit{identifier-list} \), which is a sequence of identifiers separated by semi-colons. (Semi-colons are used because the semantics may be looked upon as that of a disjunction, and we use semi-colons for disjunctions.) This construct is syntactically restricted.

\( R<\textit{identifier}> \) Matches \( R \) and then enters the lexical mode \( \textit{identifier} \). This construct is syntactically and semantically (it may introduce conflicts) restricted.

\( <>\textit{identifier} \) This regular expression cannot ever match. It is used to generate a placeholder token type that may be inserted by a filter. The construct is called virtual token. For an example of its use, see Section 3.4.10 [Handling the Problem with typedef], page 42.

\( R1 \mid R2 \) Matches either \( R1 \) or \( R2 \). The construct is called disjunction.
3.6.2.1 Restrictions on Syntax

The syntax described above is restricted in the following fashion:

- `<>identifier` must always be an entire token type.
- expressions built using `\~`, `\$`, `/`, and `<...>` may not occur inside parentheses.
- expressions built using `\~`, `\$`, `/`, and `<...>` may not be nested.
- `\$` may not be applied to expressions built with `/`.

In conjunction with the precedence of the operators, these rules impose strict restrictions on the involved constructs. In addition, the use of defined names are restricted to obeying these rules after expansion.

3.6.3 Definitions

To make the regular expressions (and the grammar in general) easier to read, we allow the grammar writer to give symbolic names to regular expressions. The use of such definitions is strongly encouraged. A definition has the following form:

```
identifier = R.
```

where `identifier` is an identifier and `R` is a regular expression. The definition of a name must occur before the first construct it is used in. Redefinition of names is not allowed. Many examples of the use of definitions may be found in the tutorial section, see Section 3.4.3 [Filters and Definitions], page 27.

3.6.4 Inheritance

Sometimes you wish two or more lexical modes to behave nearly identically, or maybe you wish to combine two lexical modes to a combined mode. To handle this situation, we allow one lexical mode to inherit all the matches from a set of other lexical modes. (The term "inheritance" is borrowed from object-oriented design.) This is accomplished with the `inherits` directive that have the following syntax:

```
:- inherits(derived, bases).
```

where `derived` is the name of the mode that should inherit the tokens and `bases` is the name of the mode (or a list of names of the modes) that `derived` should inherit from. For an example of inheritance, see Section 3.4.4 [Lexical Modes], page 29. We require that if a mode occurs both as a derived mode and a base mode, the directives are to be arranged so that the occurrence as derived modes precede all occurrences as base modes. We only allow one occurrence as derived mode.
The subset rule is interpreted across lexical modes, creating the effect of overriding in the object-oriented sense. This means that we may change the behaviour of a regular expression in its inherited version. With *behaviour* we mean any change of current lexical mode. For an example of overriding, see Section 3.4.4 [Lexical Modes], page 29. To understand how this works, think of the old expression as "matching in either mode" and the new expression as "matching only in the derived mode". Thus the latter is a subset of the former. The effect of the subset rule is to override the old expression with the new one, as desired.

### 3.7 Grammar Rules

The bulk of a SAGA grammar file naturally consists of grammar rules. The notation used is based on that of context-free grammars, see Section 3.3.3 [Parsing], page 18. The rules consist of non-terminal symbols that refer to other grammar rules and regular expressions (see Section 3.6.2 [Regular Expressions], page 47) as terminal symbols that refer directly to the input. The grammar symbols may be annotated with arguments that pass information around and rules may be annotated with Agents goals that compute on that information. The goals may also refer to global accumulators.

We also allow the use of some shorthand notation, EBNF, that may be used instead of some common kinds of rules. We also support accumulator notation similar to the one used in Agents. Together, EBNF and accumulator notation provide quite large expressive power.

#### 3.7.1 Basic Rules

The basic syntactic elements of a grammar rule are the terminal and the non-terminal symbols. In addition we allow a few special grammar symbols.

##### 3.7.1.1 Terminal Symbols

The basic form of a terminal symbol is the following:

\[ \{ \text{regular} \} \{ \text{lexeme-term} , \text{line-term} \} \]

where *regular* is a regular expression (see Section 3.6.2 [Regular Expressions], page 47) and both *lexeme-term* and *line-term* are Agents terms. When the rule is executed and the terminal matches a token, then *lexeme-term* will be unified with the lexeme of the token and *line-term* will be unified with a structure describing the line the token occurred on. The information in the latter structure can be accessed by agents in the SAGA standard library and is primarily intended to produce nice
error messages. For a description of the structure, see Section 3.9.1.1 [The Parser Return Status], page 61.

For convenience, we allow several shorter forms. Let string be a character string and let identifier be an unquoted Agents atom. Also let regular, lexeme-term and line-term be as above. The short forms are:

' string ' Short form for '{ string }{_,_}'.
" string " Short form for '{" string "}{_,_}'.
identifier Short form for '{identifier}{_,_}'.
{regular} Short form for '{regular}{_,_}'.
identifier(lexeme-term)
Short form for '{identifier}(lexeme-term, _)'.
{regular}(lexeme-term)
Short form for '{regular}(lexeme-term, _)'.
identifier(lexeme-term, line-term)
Short form for {identifier}(lexeme-term, line-term)

We also allow terminal symbols to be annotated with extra information. An annotated terminal has one of the following forms:

terminal$shift
terminal$op(precedence, associativity)

where terminal is an unannotated terminal as described above, precedence is an integer, indicating an operator precedence, and associativity is 'left', 'right' or 'nonassoc', indicating an operator associativity. It is interpreted analogous with the op-directive, Section 3.5.1 [Directives], page 44.

3.7.1.2 Non-Terminal Symbols

The possible forms of a non-terminal are the following:

identifier
identifier(arguments)
identifier-accumulators
identifier(arguments)-accumulators

where identifier is an unquoted Agents atom, arguments is a non-empty comma-separated sequence of Agents terms and accumulators is a non-empty dash-separated sequence of Agents variables. In the same fashion as for Agents agents, each accumulator argument will be replaced by two ordinary arguments.
3.7.1.3 Special Grammar Symbols

There are two kinds of special grammar symbols: The null operator and the slashing action. The null operator is used when you want the effect of annotating a terminal with op/2 but lack a terminal to pin it on. The null operator has the following form:

\[ \{ \} \text{op}(\text{precedence, associativity}) \]

where precedence is an integer indicating an operator precedence and associativity is ‘left’, ‘right’ or ‘nonassoc’ indicating an operator associativity. The effect on the rule is completely analogous to a normal use of the op-annotation.

The slashing action’s only effect is that it changes the way accumulator variables are expanded. In fact, the name “slashing” partly refers to the effect it has on the accumulator chain. (The name was also inspired by the use of a slash (‘/’) in the syntax.) The two possible forms are the following:

\[ \text{Acc/Inserted} \]
\[ \text{Acc/(Before, After)} \]

where Acc, Inserted, Before and After all are Agents variables. The first form is a short form for ‘Acc/(Inserted, Inserted)’ and is used when Inserted is an accumulator variable. Henceforth we only refer to the second form.

For this construct to be meaningful, Acc must be a variable that is used in accumulator goals. The effect of the slashing action amounts to renaming of all occurrences of Acc that follows the slashing action in the phrase to a new name. In the process two unifications are added. (For definitions of phrase and goal see Section 3.7.1.4 [Head and Body], page 54.) Before the goal, Acc should be unified with Before and after the goal the new name should be unified with After. For an introduction to the use of slashing actions, see Section 3.4.8 [More EBNF], page 37. For an example of how the slashing actions change the way the accumulator chains are expanded, see Section 3.7.2 [Accumulator Syntax], page 55.

3.7.1.4 Head and Body

The basic form of a grammar rule is the following:

\[ \text{head : body.} \]

where head is a non-terminal and body is a non-empty sequence of phrases, separated by semi-colons. Each phrase represents an alternative body. Thus, a rule of the form:

\[ \text{head : phrase1 ; phrase2 ; \ldots ; phraseN.} \]
is semantically equivalent to a set of rules of the form:

\[
\begin{align*}
\text{head} & : \text{phrase1}. \\
\text{head} & : \text{phrase2}. \\
\ldots & \\
\text{head} & : \text{phraseN}.
\end{align*}
\]

with identical heads.

Each phrase has one of the following two basic forms:

\[
\{\} \text{ guard-operator goal}
\]

\[
\text{conjunction guard-operator goal}
\]

where guard-operator is either \('\&'\), \('\rightarrow'\) or \('??'\). The goal is an Agents goal to be executed and conjunction is a comma-separated sequence of grammar symbols and EBNF constructs.

The guard operators \('\rightarrow'\) and \('??'\) are used for conditional rules and non-determinism, respectively. (See Section 3.12.2 [Non-deterministic Rules], page 76, and Section 3.12.1 [Conditional Rules], page 74.) With one exception, we require that all clauses belonging to the same head should use the same guard operator. The exception is that a \('\&'\) guard may be used in the last clause when all other clauses are \('\rightarrow'\) (see Section 3.12.1 [Conditional Rules], page 74). Henceforth we only discuss the case when the guard operator is \('\&'\).

The EBNF constructs are discussed separately, see Section 3.7.3 [EBNF], page 56. The EBNF constructs are merely shorthand for the introduction of new rules, so for all practical purposes their semantics corresponds to a pre-processing step.

For convenience we also allow the guard operator and the goal to be omitted:

\[
\{\} \quad \text{Short form for \('\{\} \& \text{true}'\)}
\]

\[
\text{conjunction} \quad \text{Short form for \('\text{conjunction} \& \text{true}'\)}
\]

### 3.7.2 Accumulator Syntax

Accumulator syntax is a convenient way of expressing a gradually changing value in SAGA as well as in Agents. Accumulator variables are (occurrences of) variables that are given as arguments to an agent or a non-terminal after the closing parenthesis of the ordinary argument list. They are separated from the closing parenthesis and from each other by dashes ('-'). An accumulator variable in Agents typically expands during a pre-processing step to a set of new variables, two for each occurrence, and possibly a set of unifications to connect the new variables together. This is the general pattern in SAGA as well, but the situation becomes more complex
since the accumulator variables also interact with the expansion of EBNF. The interaction is such that neither expansion of accumulators nor expansion of EBNF can be performed as a pre-processing step to the other. In this section we deal only with the pure accumulator syntax. The aspects of accumulator expansion that relates to EBNF are discussed in the context of EBNF, see Section 3.7.3 [EBNF], page 56.

The basic idea with accumulator syntax is to replace each occurrence of an accumulator variable with two new ordinary variables, the in-variable and the out-variable. For simplicity we assume that all rules have been expanded so that each rule only has a single phrase in the body (see Section 3.7.1.4 [Head and Body], page 54). Consider the occurrences of an accumulator variable, say A, in the body of a rule. The out-variable part of an occurrence of A is chosen to coincide with the in-variable of the next occurrence, hence the term accumulator chain. An occurrence of A in the head of the rule is expanded so that the in-variable part coincides with the in-variable of the first occurrence in the body and so that the out-variable part coincides with the out-variable part of the last occurrence in the body.

Actually, the above description is a little misleading. In reality, we do not deal with "occurrences of accumulator variables" but rather with "accumulator occurrences of variables". Thus, intermixed with the occurrences in accumulator positions of a variable A there may be ordinary occurrences of A. These are obviously not replaced with pairs of variables but rather with the out-variable part of the previous accumulator-position occurrence. (This occurrence is also the in-variable part of the next accumulator-position occurrence.)

A simple example would be the following rule:

\[ a(A) \rightarrow B: \, b-A, \, c(A, \, B), \, d-B \, \& \, \text{foo}(A)-B. \]

The above rule is semantically equivalent to the rule:

\[ a(A_0, \, B_0, \, B): \, b(A_0, \, A_1), \, c(A_1, \, B_0), \, d(B_0, \, B_1) \, \& \, \text{foo}(A_1, \, B_1, \, B_2), \, B = B_2. \]

For technical reasons the actual expansion looks a little different, but the above example should give you a fair idea.

When working with accumulator chains in the context of SAGA rules we suffer one handicap that is not encountered in Agents agents. Accumulator variables occur both as arguments to grammar symbols and in goals. All occurrences of the former kind precede all occurrences of the second kind. This creates a sometimes artificial ordering that makes it hard to exploit the accumulator syntax to its full potential. One solution would be to allow goals and grammar symbols to be mixed freely, but this would create subtle differences in semantics between similar rules. Such a solution goes against the SAGA philosophy that a bare grammar should be extendible.
to a full-blown parser with a minimum of complications. Thus, this solution was rejected in favour of *slashing actions*. The slashing actions are used to cut the accumulator chain in two pieces at a certain point and to link in another chain between the two pieces. See Section 3.7.1.3 [Special Grammar Symbols], page 53.

As a simple example of how slashing actions are expanded, consider:

\[
\text{a-A: b-A, A/B, c-A & foo-B.}
\]

This rule is semantically equivalent to the following rule:

\[
\text{a(A0, A): b(A0, A1), c(A2, A3) & B = A1, foo-B, A2 = B, A3 = A.}
\]

### 3.7.3 EBNF

When you are writing grammars you often write rules with the same structure over and over again. To simplify the writing of grammars SAGA supports a number of constructs that can be used instead of writing new rules. Such extensions to the basic grammar are often refered to as *Extended BNF*, or shorter *EBNF*, for historical reasons.

EBNF goals may occur in phrases in any place a non-terminal is allowed. The semantics is given by a pre-processing step. An EBNF-goal is always replaced by a new non-terminal and new rules are added for the non-terminal. We have four basic kinds of EBNF-goals: Three kinds of repetition rules and one kind for nesting.

The form of an EBNF-nesting is the following:

\[
\text{(body)}
\]

where body has the same syntax as a body of a rule. The semantics of the construct corresponds to an introduction of a new rule with body as body and a new non-terminal applied to the free variables of body. The free variables in body that occur in accumulator positions are put in accumulator positions as arguments to the new non-terminal as well.

The repetition rules uses the same operators as we know from regular expressions: ‘*’, ‘+’ and ‘?’. Recall that these stand for zero-or-more, one-or-more and zero-or-one occurrences, respectively. In contrast to regular expressions, we usually want to gather some information from the repetition. This can be done by collecting information in a list or by passing information between iterations with accumulator arguments — or both. The two basic forms of EBNF-repetition with the ‘*’-operator are:

\[
\text{call*}
\]

\[
\text{Var\textbackslash call*(Arg1, Arg2)}
\]
where \textit{call} is either a grammar symbol applied to some number of arguments, possibly zero, or a EBNF-nesting. \textit{Arg1} and \textit{Arg2} are Agents terms. The difference between the two forms is that the second creates a list (actually a d-list) with one element for each match while the first does not build any list at all. The forms for ‘*’ and ‘?’ always are completely analogous to the forms for ‘*’. Therefore we list only the forms for ‘*’ in this section.

The semantics for the repetition forms are given by expansion and addition of new non-terminals. The entire EBNF construct is translated from the inside and out. Therefore, we may restrict our attention to the case when \textit{call} is a grammar symbol applied to a number of ordinary arguments and a number of accumulator arguments.

Let us look at the expansion rule for the first form of ‘*’. Consider an EBNF goal of the following schematic form:

\[
symbol(T1, \ldots, Tn)-A_1-\ldots-A_m*
\]

where \(T1\) to \(Tn\) are Agent terms and \(A1\) to \(An\) are Agents variables. Further assume that \(X1\) to \(Xj\) are the set of free variables in \(T1\) to \(Tn\). The EBNF goal will be replaced with the following (where \textit{newsymbol} is a fresh name):

\[
\textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m
\]

The following two rules for \textit{newsymbol} are then added to the grammar:

\[
\begin{align*}
\textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m & : \emptyset. \\
\textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m & : \\
& \textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m, \\
& \symbol(T1, \ldots, Tn)-A_1-\ldots-A_m.
\end{align*}
\]

The second form for ‘*’ is only marginally more complex. The variable abstracted on, \textit{Var}, is not considered free. The EBNF goal:

\[
\text{Var}\backslash\text{symbol}(T1, \ldots, Tn, \ \text{Var})-A_1-\ldots-A_m*(\text{Arg1}, \ \text{Arg2})
\]

will be replaced with the following (\(X1\) to \(Xj\) are the free variables):

\[
\textit{newsymbol}(X1, \ldots, Xj, \ \text{Arg1}, \ \text{Arg2})-A_1-\ldots-A_m
\]

The following two rules for \textit{newsymbol} are then added to the grammar (where \textit{L} is a fresh variables):

\[
\begin{align*}
\textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m-L & : \emptyset. \\
\textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m-L & : \\
& \textit{newsymbol}(X1, \ldots, Xj)-A_1-\ldots-A_m-L, \\
& \symbol(T1, \ldots, Tn, \ \text{Var})-A_1-\ldots-A_m \\
& \& [\text{Var}]-L.
\end{align*}
\]
Note that the Agents goal in the second clause, $\text{'[Var]\cdot L']$, is accumulator syntax (in Agents) to add $\text{Var}$ to the end of the d-list $L$. The expansion of `$+` and `$*$` is similar.

Let us illustrate the semantics with a example on how the constructs may be expanded. Consider the following grammar fragment:

$$a(A, B)_C-D-E : b(A)_C*, X\text{\(}\backslash\text{c}(X, B)_D*E.$$ 

This is semantically equivalent to the following fragment:

$$a(A, B)_C-D-E : b1(A)_C, c1(B)_D-E. b1(_)_C : \{\}. b1(A)_C : b1(A)_C, b(A)_C. c1(_)_D-E : \{\}. c1(B)_D-E : c1(B)_D-E, c(X, B)_D & [X]_E.$$ 

As we can observe from the examples, the free variables are propagated to the relevant goals. Accumulator variables are linked so that the information are passed between the iterations. In the second form, one of the free variables in $\text{call}$ is bound to form an abstraction ($\text{Var}$ in 'Var\backslash call*(Arg1, Arg2)'). The bound variable corresponds to the elements that are put in the d-list.

As usual we allow some slimmed shorthands for convenience. Let $\text{grammar-symbol}$ be a grammar symbol (not applied to any arguments) and let $\text{Var}$, $\text{Arg}$, $\text{Arg1}$, $\text{Arg2}$ and $\text{Acc}$ be Agents variables. Then the following short forms are allowed:

$\text{grammar-symbol}(\text{Arg})$

Short form for 'X\text{\(}\backslash\text{grammar-symbol}(X)\star(\text{Arg}, \square)' where X is a new variable.

$\text{grammar-symbol}(\text{Arg1}, \text{Arg2})$

Short form for 'X\text{\(}\backslash\text{grammar-symbol}(X)\star(\text{Arg1}, \text{Arg2})' where X is a new variable.

$\text{grammar-symbol}\star$-Acc

Short form for 'X\text{\(}\backslash\text{grammar-symbol}(X)\star(\text{Acc1}, \text{Acc2})' where X is a new variable and Acc1 and Acc2 are the accumulator expansion of Acc.

$\text{Var}\backslash call*(\text{Arg})$

Short form for 'Var\backslash call*(\text{Arg}, \square)'.

$\text{Var}\backslash call*$-Acc

Short form for 'Var\backslash call*(\text{Acc1}, \text{Acc2})' where Acc1 and Acc2 are the accumulator expansion of Acc.

Naturally, the same forms apply to the operators `$+$ and `$*$ as well.
3.8 Filters

Filters are SAGA specific and form an intermediate step between the lexical analyzer and the parser. See Section 3.3.2 [Filters], page 18, for a brief introduction. See Section 3.4.3 [Filters and Definitions], page 27, for an example of a simple filter. See Section 3.4.10 [Handling the Problem with typedef], page 42, for an example of a more sophisticated filter.

In the following, token definition refers to a regular expression applied to zero, one or two arguments. If not enough arguments are used, default values are assumed for the missing arguments. A filter definition has the following form:

\[ \text{head} \leftarrow \text{body} \]

where head is either a variable or an Agents list, possibly empty, of token definitions or variables. The body is a semi-colon separated non-empty sequence of sub-bodies where each sub-body has one of the following four forms:

- \( \text{token-def} \)
- \( \text{token-def-sequence} \)
- \( \text{token-def} \& \text{goal} \)
- \( \text{token-def-sequence} \& \text{goal} \)

where \( \text{token-def} \) is a token definition, \( \text{token-def-sequence} \) is a semi-colon separated sequence of token definitions and \( \text{goal} \) is an Agents goal.

We require that all token definitions in a sub-body (that share the same goal) have identical arguments.

To understand the semantics of a filter it should be read analogous to a grammar rule. In both cases we replace things matched by the right-hand side with the left-hand side. The left-hand side of the rule is a list of terminals rather than the single non-terminal of grammar rules. The right-hand side is restricted to a disjunction matching a single token.

When a token that matches a right-hand side of a filter rule is encountered, the token is replaced with the sequence of tokens indicated by the left-hand side of the filter rule. At the same time the goal, if any, is executed. If the token definitions on the left-hand side lack lexemes or line descriptors, then they are copied from the matched token. The goal may refer to the arguments of the token definitions on the right-hand side, the variables on the left-hand side and the global accumulators.
3.9 The Interface to Agents

Since SAGA generates Agents programs, it needs to be able to interface to Agents on several different levels. First of all, the generated program is not stand-alone but rather a subroutine, an agent in Agents terminology, so we need to be able to call agents in the generated program from Agents. We also want to use pieces of Agents code embedded in the SAGA grammar file. This embedded code has access to some additional variables, constants and agents that you need to know about.

3.9.1 Calling the Parser

The name and arity of the parser agent depends on the contents of the grammar file. Assume that the header has the following form:

```prolog
:- saga(module_name, start_symbol, globals).
```

Let us also assume that the non-terminal start_symbol takes $N$ arguments. Then the agent has the following basic call pattern:

```prolog
module_name.start_symbol(Arg1, ..., ArgN, -Status,
        +InitialGlobals, -FinalGlobals,
        +Settings,
        +Input, -RemainingInput)
```

If globals is the empty list, then InitialGlobals and FinalGlobals are omitted. The Settings is optional and may be included or omitted at will.

Arg1 to ArgN are unified with the arguments to start_symbol. The Status variable indicates any problems with the parse, see Section 3.9.1.1 [The Parser Return Status], page 61. The InitialGlobals is a list with values that are unified with the globals to provide its initial values. The FinalGlobals is unified with the final values of globals. The Settings is a feature record with the functor settings. The feature arguments is some subset of the following:

- **block_size**: Size
  
  Gives the number of characters that should be read from a port at a time. If you want to read only part of a file this should be set to 1. The value -1 denotes the size of the input. The default value is 1000.

- **non_det_block_size**: Size
  
  Gives the number of characters that should be read when preparing to enter a non-deterministic phase. The value -1 denotes the size of the input. The default value is -1.
The Input is the list to be parsed and RemainingInput is the rest of the list that remains to be parsed. The Input may also be a port that can handle getc/1 messages. In this case RemainingInput is the same port as Input, but messages sent to RemainingInput are synchronized so they appear after the messages the SAGA parser sends during the parse. Usually, RemainingInput will be the empty list or a completely read stream at the end of a successful parse. The exception is when we are only parsing part of the input, Section 3.9.2 [Parsing Only Part of the Input], page 62.

3.9.1.1 The Parser Return Status

The status value returned by the parser depends on the error recovery routines. Here we describe the values returned by the default error recovery. If you write your own error recovery scheme (see Section 3.11.2 [User Defined Error Recovery], page 72) you have full control over the returned status.

To indicate the location of parse errors we use a data structure to describe lines. Usually there will be no need to access it directly, since we have library agents to do common tasks (see Section 3.14 [SAGA Standard Library], page 77). The representation is:

\[ \text{line(LineNumber, LexemeList)} \]

The LexemeList is a list of lexemes on the line prior to filtering. The first lexeme on the line may begin on a previous line and the last lexeme may end on a later line.

The default error recovery scheme can return the following status values:

true Parsed without any problems.

repairs(RepairList)

The error recovery scheme made some repairs and recovery actions. The possible elements in the list are detailed below.

parse_error(Lexeme, Line, RepairList)

An unrecoverable parse error occurred at the Lexeme on the line Line. The RepairList is like the one above, listing the less serious errors that occurred earlier in the file. The Lexeme is a list of characters that represents the lexeme of the offending token. The Line is the line-describing structure described above.

The list of repairs mentioned have the following possible elements:

inserted(Lexeme, Before, Line)

Inserted Lexeme before Before on line Line.
deleted(Lexeme, Line)

Deleted Lexeme from line Line.

replaced(WrongLexeme, RightLexeme, Line)

Replaced WrongLexeme with RightLexeme on line Line.

recovery(Lexeme, Line)

Used an 'error'-rule to recover from the error at Lexeme on line Line.

3.9.2 Parsing Only Part of the Input

In this section, we refer to the data read from a port object as a "file", for convenience. Sometimes it is desirable to parse smaller units than an entire file, without having to use lists. The problem is to make the parser aware that it has reached the end of the unit without reading any look-ahead token. SAGA is implemented so that it always reads a look-ahead token. This behaviour has performance advantages. More important, it avoids some subtle bugs concerning look-ahead tokens since it is consistent.

To be able to get the parse to stop before time, the user has to insert an end-of-file token on the input. This token tells SAGA to stop parsing and it also acts like the look-ahead token we need. The identifier end_of_file is predefined to a virtual token regular expression that represents end-of-file. This may be inserted on the input with a filter.

The grammar must be written so that a real end-of-file is distinguishable from the simulated one. The simple way to do this is to have a rule that matches the empty input and returns something special to indicate that the input really is exhausted. In the example below, 'eof' is returned on a real end-of-file, but not on a simulated one. Apart from this restriction and the addition of a filter we write the grammar just as if we were going to match the entire file. For example, to read a list of numbers terminated with a semi-colon we may use the following grammar:

```prolog
:- saga(partial_read, list).

number = [0-9]+.
blanks = [',','\t','\n']++.

[] <- blanks.
[';','end_of_file'] <- ';'.

list(Nums): number*(Nums), ';'.
list(eof): {}.
```

You must also call the parser with a setting to stop it from buffering when you read only part of a file. See Section 3.9.1 [Calling the Parser], page 60.
3.9.3 Embedded Agents Code

In various places in the grammar file you may embed Agents code. You may have entire Agents agents in the file (see Section 3.5 [Structure of the Grammar File], page 44) and Agents goals in filters (see Section 3.8 [Filters], page 59) and actions (see Section 3.7.1.4 [Head and Body], page 54). Common for all of these are some standard data structures and some standard agents. In addition, the goals in filters and actions may access some standard global accumulators.

3.9.3.1 Tokens as Data Structures

One of the extensions in SAGA to the normal Agents syntax is that you may refer to token type regular expressions with arguments. A term inclosed in braces ('{ and '}') is interpreted as a regular expression that corresponds to a token type. If you for some reason need to simulate the normal use of braces, then you use nested braces ('{{ and '}}'). These regular expressions may be used in filters when inserting tokens and in error recovery for various reasons. They are represented as standard Agents data structures so they may be stored in data structures, unified with each other and so on.

3.9.3.2 Standard Global Accumulators

In addition to the user defined global accumulators, there are predefined ones:

LexMode  This variable holds an internal representation of the current lexical mode. It should be accessed with begin/3. See Section 3.9.3.3 [Standard Agents], page 64.

Line      This variable holds the current line represented in the usual way.

3.9.3.3 Standard Agents

Some standard agents are put in the generated file. In addition to these, there are library agents that may be used. See Section 3.14 [SAGA Standard Library], page 77. The standard agents are:

begin(+ModeName)-LexMode

Change the lexical mode to the one denoted by the atom ModeName. Please note that a change of lexical mode from an action may be delayed for one token because of the look-ahead token. This is not the case for filters, since they operate directly on the look-ahead token.
newline_code(-Code)

Returns the character code of the newline character. This is used when calling some library agents.

tab_code(-Code)

Returns the character code of the tabulator character. This is used when calling some library agents.

3.10 The SAGA Parsing Algorithm

The parsing algorithm used in SAGA is an LALR(1) parsing algorithm. This algorithm is a bottom-up shift/reduce algorithm. Here we are going to outline how this algorithm operates without going into details. A basic understanding of the algorithm may make it easier to avoid and resolve conflicts in grammars. For a basic introduction to the LR(k) class of grammars, see Section 3.3.3.2 [Grammar Classes], page 19.

3.10.1 Shift/Reduce Parsing

SAGA uses a shift/reduce parsing algorithm that operates with a stack of grammar symbols. Initially the stack is empty. Shift means pushing the look-ahead symbol onto the stack. Reduce means replacing a number of grammar symbols on the top of the stack, corresponding to the right-hand side of a grammar rule, with the single non-terminal on the left-hand side of the rule. In this process, the arguments to the grammar symbols are unified and the associated action is executed. Parsing action is a collective term for shifts and reductions. Shifting causes the stack to grow. Reducing causes the stack to shrink if the right-hand side of the rule has at least two grammar symbols, and to grow if it has an empty right-hand side. If the right-hand side consists of a single grammar symbol, the stack will remain the same size but its top symbol will change. At the end of a successful parse only the top symbol will be on the stack and all the input will be consumed.

3.10.2 LR Parsing

The problem with the above scheme is to decide when to shift and when and with what rule to reduce. To make this choice we make use of a state and the look-ahead token. The state is a number that represents a fixed amount of information about the input already seen. The states are also maintained on the stack, paired together with the symbols. They provide information needed to compute new states. The current state and the look-ahead symbol are used to look up the next parsing action in a table. In the case of a shift, the table also gives the new state to enter. In the
case of a reduction, another table is used to compute the next state from the top state of the reduced stack.

For some grammars, the input already seen and a single token of look-ahead is not enough information to choose between different parsing actions. In this case we say that the grammar fails to be LR(1). We also say that we have one or more conflicts between parsing actions. More specifically, if we cannot choose between shift and a reduction we have a shift/reduce conflict. If we cannot choose between a set of reductions we have a reduce/reduce conflict. The former kind is usually rather unproblematic and can often be resolved with additional information. Thus, we may actually use grammars that is not LR(1), thus increasing the power of the tool, Section 3.10.3 [Resolving Shift/Reduce Conflicts], page 66.

The parsing algorithm used in SAGA is not actually full LR(1), but rather a more limited and more space-efficient form called LALR(1). Therefore it is possible that we fail to make a choice between parsing actions for some LR(1) grammars. See Section 3.10.4 [Mysterious Reduce/Reduce Conflicts], page 67. Fortunately, these problems are rather rare.

SAGA also allows us to try different alternatives by allowing conditional rules and non-determinism. In these cases we need not just choose one parsing action, but rather we may try out different ones to see if any of them gives the desired result. See Section 3.12.2 [Non-deterministic Rules], page 76, and Section 3.12.1 [Conditional Rules], page 74.

3.10.3 Resolving Shift/Reduce Conflicts

SAGA allows two ways to resolve shift/reduce conflicts: shift-annotations and operator precedences. The former is used to resolve a conflict in favour of shift and the other is used to resolve conflicts based on precedence rules.

3.10.3.1 shift Annotations

The most simple form of conflict resolution is picking the shift action. This is the correct choice for the famous dangling else conflict, see Section 3.4.5 [From Expressions to Statements], page 30, for an example of this use.

This can be resolved simply by identifying the rule that we wish to use and the terminal in it that should be shifted. Annotating this with shift like this:

```
terminal$shift
```

simply tells SAGA to resolve conflicts involving this terminal in favour of the shift action.
3.10.3.2 Operator Precedence

Often a grammar is intentionally written in an ambiguous manner with regard to operator precedences. The op-directive is used to provide the additional information needed to resolve the conflict, see Section 3.5.1 [Directives], page 44. In some cases you also need to use the op-annotation, see Section 3.7.1.1 [Terminal Symbols], page 52 and Section 3.7.1.3 [Special Grammar Symbols], page 53.

Resolving conflicts with operator precedence is a little more complicated than simply shifting. With operator directives we assign precedences to operators. Rules are also assigned precedences either automatically or with op-annotations. A rule inherits the precedence of its right-most terminal that has a precedence. The op-annotation is used to override the precedence of a specific occurrence of a terminal symbol, and thereby changing the precedence of the rule. We also allow to insert an empty goal annotated with op in case you would like to give a precedence to a rule that does not have any terminal symbols.

When resolving a conflict between reducing with a rule that has a precedence and shifting a token that has another precedence, we use the following rules:

- If the precedence number of the look-ahead is numerically smaller than the precedence of the rule: Shift.
- If the precedence number of the look-ahead is numerically larger than the precedence of the rule: Reduce.
- If the precedences are equal and left-associative: Reduce.
- If the precedences are equal and right-associative: Shift.
- If the precedences are equal and non-associative: Declare a parse error.

3.10.4 Mysterious Reduce/Reduce Conflicts

The differences between LR(1) and LALR(1) parsing is rather subtle and extremely technical. Unfortunately, the differences may cause unexpected problems with some grammars. The symptom is reduce/reduce conflicts that seem inexplicable. Simply stated, the cause is that two contexts are so similar that the LALR(1) algorithm thinks that they are the same.

Let us consider an example grammar with this problem:

```
:- saga(mystery, proto).

id = [a-z]+.

proto: "prototype", name, ";".
proto: "prototype", type, "type", ";".
proto: "prototype", name, "(" protoargs ")", ";".
```
protoargs: type.
protoargs: name, ":", type.
protoargs: protoargs, ";", type.
protoargs: protoargs, ";", name, ":", type.

type: id.
name: id.

**SAGA** produces the following error message on this grammar:

**CONFLICT. Assuming input of the form:**

"prototype" {id}

With ";" as the next token there is a conflict between:

[reduce] name : {id}.
[reduce] type : {id}.

This seems quite odd. Clearly the id should be considered a name according to the grammar. The grammar is indeed LR(1), but not LALR(1).

The LALR(1) algorithm first decides what parsing states to create, and then it checks the proper look-aheads for each state. A full LR(1) parser makes use of the look-ahead information when creating the states. This means that each LALR(1) state is a combination of a set of LR(1) states. This is the great advantage of LALR(1) parsing — fewer states. The drawback is that states that are truly different in the LR(1) case may be forced into a common LALR(1) state.

In this case, there are two states in which the only possible actions are the reduction of id to a name or a type. The LALR(1) construction gives them a single parser state. When the possible look-aheads are generated, it is discovered that this decision gives a conflict on the look-ahead ";". Inside a protoargs we should reduce to a type. Directly after a "prototype", we should instead reduce to a name.

One way to fix the problem is to add some dummy rule to one of the contexts, so the difference becomes more apparent to SAGA. This can be accomplished by using a virtual token (see Section 3.6.2 [Regular Expressions], page 47) that never will match. The point is that we know that it will never actually be used, but SAGA does not know this. Since one of the states appear to have an extra parsing action (the shift action for the virtual token) it will not be confused with the other state.

Usually, we can easily construct the dummy rule from the error message SAGA displays. In this case, the problematic input was:

"prototype" {id}
What we do is that we add another rule for the start symbol. This rule should match the offending input followed by a virtual token (that we do not expect to find on the input). In this case, add the rule:

```proto: "prototype", id, {<>bogus}.
```

Sometimes the conflict involves more than two states, and then the fix will generate a new conflict. This conflict is removed with the same technique. However, mysterious conflicts are quite rare, and the ones that involve more than two states are even more so.

### 3.11 Error Handling

One of the hardest and most important tasks for a parser is to handle incorrect input in a graceful fashion. Even if you invest enormous amounts of time into a hand-written parser where you have full control and can tailor the error recovery to a given grammar it is hard to achieve really good error recovery. When you do not want to spend too much time on creating error recovery the problem gets that much harder.

SAGA tries to strike a balance between powerful recovery that requires the a lot of work on the grammar writer's part and automatic recovery that yields poor results but requires a minimum of grammar writer interaction. You may write your own recovery routines if you wish, but you may also rely on the default routines provided.

#### 3.11.1 Default Error Recovery

The default error recovery in SAGA is performed in two steps. First we attempt to repair the input by performing a *local correction*. A local correction may for instance insert a missing semi-colon or delete a superfluous right parenthesis. If no suitable local correction is found, we attempt to throw away a suitable portion of the input around the error: *Phrase level recovery*. For this we need the grammar writer to supply special error handling rules in the grammar for reasons discussed below.
3.11.1.1 Local Correction

Most of the parse errors in typical programming languages are single token errors. They include things like misspelled keywords, mismatched parentheses and missing or incorrect punctuation characters. In these cases it is often quite simple to fix the problem by inserting a missing token, deleting a superfluous token or replacing a token with another. This can be done completely without grammar writer interaction if we put some restrictions on the tokens that are considered for insertion. We are also able to produce reasonably good error messages simply by informing the user what change of the input that seems appropriate.

The decision of what local correction to perform is obviously a critical one. The strategy used by SAGA is trial-and-error based. We try all the feasible corrections until we find one that allow us to continue parsing for at least five tokens without any new parse errors. If such a correction is found it is rather likely that it is a good one, so we log the repair (see Section 3.9.1.1 [The Parser Return Status], page 61) and resume parsing.

First SAGA attempts to insert a token before the offending look-ahead symbol. If that fails, we try to delete the look-ahead symbol. If that also fails, we attempt to replace the look-ahead symbol with another symbol. If that fails, the local correction fails.

When we insert or replace symbols, we cannot insert just any symbol and expect the insertion to be transparent to the parser and clear to the user. Assume for instance that we want to insert an identifier. There are usually infinitely many to choose from. If we just pick any syntactically legal string that would pass as an identifier it will most likely result in semantic errors from other parts of the parser because the identifier is not declared or has the wrong type. The user may also be somewhat confused about all these messages about an identifier that she has never heard of. SAGA takes the conservative approach and inserts only tokens with a fixed appearance (essentially the ones that are written as literal strings).

3.11.1.2 Phrase Level Recovery

As soon as we want to handle fixes bigger than a token (or possibly a few tokens) we face a new problem, similar to that of inventing tokens to insert. The problem is that we want to force a part of the input to match some rule when it really does not. Since groupings usually are associated with attributes (arguments to rules) we must be prepared to invent values for these attributes. This is clearly impossible to do automatically, so the grammar writer must supply the necessary information.

The approach used by SAGA is to let the grammar writer add special rules that can match incorrect input. The identifier error is predefined to a virtual token regular
expression that is used for error recovery. When a parse error occurs that the local
corrector cannot handle the **error** token is inserted. Then stack context is thrown
away until the **error** token can be shifted. This corresponds to throwing away
tokens to the left of the error. Then input tokens are discarded until the parse can
continue for five more tokens without further errors. Thus, we throw away tokens
on both sides of the error and tries to "glue the rest together". The **error**-rules are
just like other rules, where **error** may match any sequence of tokens. In particular,
**error**-rules may have associated actions that can be used for error reporting and
the like. An error message is also logged about the repair, see Section 3.9.1.1 [The
Parser Return Status], page 61.

Many parser generators, including Yacc, support the use of **error**-rules. As a general
recovery mechanism they suffer from a somewhat poor reputation. Their weak point
is that they are difficult to use to handle common and simple errors. SAGA uses local
corrections to handle these, so the **error**-rules are only used for secondary recovery
where they perform well.

For an example of how to use **error**-rules, consider the following grammar fragment
from the tutorial (see Section 3.4.7 [Using EBNF], page 35) extended with **error**-
rules:

```latex
stmt(E): expr(E), ";".
stmt(while(C, S)): "while", "(" , expr(C) , ")" , stmt(S).
stmt(if(C,S)): "if", "(" , expr(C) , ")" , stmt(S).
stmt(if(C, S1, S2)):
  "if", "(" , expr(C) , ")" , stmt(S1), "else":$shift, stmt(S2).
stmt(Ss): "{", stmt*(Ss), "}".
stmt([]): error, ";".
stmt(Ss): "{", stmt*(Ss), error, "}".
```

The first **error**-rule says that a statement may be a parse error followed by a semi-
colon. This corresponds to throwing away everything from the beginning of the
current statement to the next semi-colon. Actually it is not the next semi-colon but
rather the next semi-colon *that is followed by five tokens that could occur after the
end of the current statement*. Thus, at a parse error we throw away the offending
statement, or at least something that looks like it might be the offending statement,
in its entirety.

The second **error**-rule says that a compound statement may be a parse error be-
tween braces. The `stmt*(Ss)` part is only there to rescue as much useful informa-
tion as possible. This rule is primarily useful to stop right braces (`
}`) from being
incorrectly skipped during recovery.
What error-rules to add must be decided individually for each grammar with the use of the grammar in mind. In general each “major” non-terminal should have error-rules attached. In a typical programming language declarations and statements may be considered as “major”. Expressions are often considered “major” in order to handle missing parentheses and operators, but since these errors often are single token errors there might be better to leave them to the local corrector.

Usually matching pairs, like parentheses and braces, should have recovery rules unless both the tokens in the pair are guaranteed to be thrown away during recovery. Thus we might consider to add the following rule, and similar rules for the if-statement, to the above example:

\[
\text{stmt(while(1, S))}: \text{"while", "("}, \text{error}, \text{")}\text{, stmt(S).}
\]

Note that parentheses inside expressions will all be thrown away during recovery in the absence of error-rules for expr, so they cause no problems.

### 3.11.2 User Defined Error Recovery

The grammar writer may choose to write her own error recovery routines. The error-directive (see Section 3.5.1 [Directives], page 44) is used to indicate an abstraction to call when a parse error occurs. The abstraction takes a single accumulator argument: A port object that represents the parser state. This object supports the following messages:

**fixed_strings(-Tokens)**

Returns the token types that have fixed lexemes along with their lexemes. Tokens is a list of pairs with the ' - '/2 functor. The first component of each pair is a token type that has the fixed lexeme given by the second component.

**insert(+Token)**

Insert the token Token before the look-ahead token.

**insert(+TokenType, +Lexeme)**

**insert(+TokenType, +Lexeme, +Line)**

Insert a token with TokenType as token type and Lexeme as lexeme and Line as line. If Line is not given the one of the offending look-ahead token is used.
delete
delete(-Token)
delete(-Token, -Status)
    Delete the look-ahead token. Return the token that was deleted (Token) and a flag (Status) that is true if there was any token to delete and false otherwise.
forward(+Steps, -Status)
    Check if the parsing could continue until Steps tokens have been shifted or until an accept. If successful, Status is unified with true and otherwise with false.
discard(-Status)
    Throw away the top state of the parser stack. If successful, Status is unified with true, otherwise (the stack is empty) with false.
position(-Lexeme, -Line)
    Unifies Lexeme with the lexeme and Line with the line of the original erroneous look-ahead symbol.
at_eof(-Status)
    Unifies Status with true if there are no more input and with false otherwise.
token_to_parts(+Token, -TokenType, -Lexeme, -Line)
    Used to decompose a token Token into its components: The token type TokenType, the lexeme Lexeme, and the line Line.
parts_to_token(+TokenType, +Lexeme, -Token)
parts_to_token(+TokenType, +Lexeme, +Line, -Token)
    Used to compose a token Token from its components: The token type TokenType, the lexeme Lexeme, and the line Line.
status(-State)
    Unifies the status with State. The status is whatever the grammar writer wants to save between the different errors. It is also returned as status result of the parse. Typically, it is used to log error messages. Initially, the value is true.
globals(-OldGlobals)
globals(-OldGlobals, +NewGlobals)
    Reads the current values of the global accumulators (in list form) into OldGlobals. If NewGlobals is given it specifies the new values for the global accumulators, still in list form.
reset
    Resets the parser to the state it had when the error occurred.
resume(-State)

Resumes the parse with State as the new status.

quit(-State)

Terminates the parse with State as the final status.

The default recovery strategy is itself implemented with these messages. You may also access the default strategy whole or in parts — just the local corrector or just the phrase-level recovery — as library routines, see Section 3.14 [SAGA Standard Library], page 77.

3.12 Advanced Parsing Strategies

SAGA allows you to go well beyond LR(1)-parsing\(^\text{15}\) by trial-and-error testing of different possible parses. We may specify in the grammar exactly how this is to be done. We may also want to leave the behaviour more or less open. SAGA offers two basic constructs that may be combined in different ways to implement various parsing strategies. Non-deterministic rules gives us blind search and conditional rules gives us a completely specified search. We may also confine the search to certain parts of the grammar by encapsulating the former in the latter.

Some restrictions apply if you use these advanced parsing strategies when reading from a port (rather than from a list). See Section 3.12.4 [Advanced Strategies and Ports], page 77.

3.12.1 Conditional Rules

Sometimes you wish to try out alternatives in sequence. In some cases such an approach just simplifies the writing of a grammar without being strictly necessary. However, some languages are designed with intrinsic ambiguities and are inherently hard to parse by LR(k) techniques. One may argue that such languages are badly designed, but they may nevertheless be quite successful. The most well-known example of such an ambiguous language is C++. About one of the ambiguities Margaret Ellis and Bjarne Stroustrup (see The Annotated C++ Reference Manual, page 93) writes:

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a . In those cases the statement is a declaration.

\(^{15}\) Actually beyond context-free parsing.
Thus, if it looks like a declaration it is a declaration, otherwise it must be an expression. Note that this rule cannot be handled even with infinite look-ahead. Something different is needed to express the precedence between the constructions, in a way similar to the way we use shift annotations to disambiguate grammars. The SAGA mechanism to accomplish this is inspired by the use of conditional guards in Agents. To solve the above problem, you may use something like this:

\[
\text{expr_or_decl} \quad \begin{align*}
\text{declaration \rightarrow true} \\
\text{expression.}
\end{align*}
\]

When the parser tries to match a \text{expr_or_decl} it first tries to parse a \text{declaration}. If it succeeds, the reduction is performed and the goal ('true') is executed. If the parse fails (because it is not a legal declaration), the parser tries to match an expression instead.

When used on syntactically incorrect input the parse errors are reported roughly as if the simple rule:

\[
\text{expr_or_decl} : \text{expression}
\]

had been used. The reason for this is that no ‘\text{\rightarrow}’ guard was used in the last clause. This is the only case in which you may mix different guards in clauses that belong to the same non-terminal. If all clauses have ‘\text{\rightarrow}’ guards then any parse errors are reported at the beginning of the conditional sections. There are also some differences in behaviour between the two cases with respect to encapsulation (see Section 3.12.3 [Encapsulation of Non-determinism], page 76).

### 3.12.1.1 Restrictions

We have two restrictions that apply to the conditional rules. The first one is due to the semantics of Agents. As in Agents, there are some restrictions on the actions that may be executed in guards. In particular you may not send messages to an object that exists outside the guard. This also affect parsing of input from ports (as opposed to input from lists), see Section 3.12.4 [Advanced Strategies and Ports], page 77.

The second restriction is more directly concerned with the grammar proper. Essentially, we require that the beginning of the construction is recognizable with one token of look-ahead. Consider the example:

\[
\text{expr_or_decl} \quad \begin{align*}
\text{declaration \rightarrow true} \\
\text{expression.}
\end{align*}
\]
When we reach the first token of an `expr_or_decl` the only legal thing that can follow must be an `expr_or_decl`. Thus, we require that nothing outside the conditional could possibly match. For an example of a violation of this condition, consider the following grammar:

```
:- saga(illegal, a).

a(xy): "x", "y".
a(xzs): b(xzs).

b(xz): "x", "z" -> true.
b(xs): ("x")*, "z".
```

The problem with this grammar is that when we see the first ‘x’ as look-ahead, we cannot start to try to match ‘xz’. The reason for this is that it is quite possible that the correct choice is to match ‘xy’, which is outside the conditional. This is obviously a very contrived example. The reason for this is that it is hard to find any natural example where the restriction is a problem (or at least a reasonably small example).

### 3.12.2 Non-deterministic Rules

Sometimes you need to parse grammars that are not LR(1). Sometimes there is no $k$ large enough to make a grammar LR($k$), thus you need infinite look-ahead. Actually, the look-ahead is never infinite as long as the input is finite, so unbounded may be a better term. The really brute-force way to simulate long or infinite look-ahead in SAGA is to allow the parser to be non-deterministic. A non-deterministic parser will search for a solution by trying out different alternatives.

Non-determinism is specified with the use of the ‘??’ guard operator. Conflicts that only involve rules with ‘??’ guard operators are resolved non-deterministically.

With a non-deterministic grammar you often suffer from poor performance. You may even run the risk of non-termination (i.e. infinitely bad performance!). The latter problem is caused by reductions with an empty right-hand side.

The actions executed by the parser may be used to prune the search. Writing actions to prune the search enough to guarantee termination is a topic well beyond the scope of this manual. Termination issues are studied quite extensively in the natural language parsing community. In SAGA there is another issue we have to deal with when pruning the search: A guarantee for failure of actions in infinite branches is no guarantee for termination. The reason for this is that the execution order of an Agents program is left open unless you explicitly synchronize. Usually, SAGA does not synchronize the parsing proper with the actions (for performance reasons). Fortunately, it is easy to explicitly synchronize on the LexMode global accumulator.
The lexical analyzer synchronizes on this variable and the parser synchronizes on the token produced by the lexical analyzer.

3.12.3 Encapsulation of Non-determinism

Often we want to use non-determinism in some parts of the grammar and run the rest deterministically. In some cases, only certain constructs in our language need non-determinism and we want the error recovery to work approximately as usual in all other constructs. In other cases, we operate mostly non-deterministically but synchronize at regular intervals to reduce the combinatorical explosion. A typical example of the latter is to synchronize after each sentence in a natural language application.

The SAGA conditional guard operator ('->') will select one of the solutions of the guard even if it is non-deterministic, in much the same way as in Agents. For example, the following fragment could be used in a natural language application:

\[
\text{:- saga(nlp, text).} \\
\text{text (Ss): first_solution_sentence*(Ss).} \\
\text{first_solution_sentence(S): sentence(S) -> true.}
\]

where sentence is a non-deterministic rule.

3.12.4 Advanced Strategies and Ports

As discussed earlier (see Section 3.12.1.1 [Restrictions], page 75) you may not send messages inside a guard. This affects the parser itself as well. This means that you cannot use the advanced parsing strategies freely when dealing with input from ports. Since this is a major limitation, SAGA buffers a part of the port input when entering a conditional rule.

Everything will work as expected as long as your program does not try to read past the end of the buffer. If it does, that branch will fail. The only really safe way to avoid this is to set the size of the buffer to infinity. See Section 3.9.1 [Calling the Parser], page 60, for information of how to change the size of the buffer.

3.13 Invoking SAGA

The details for invoking SAGA may differ between different systems. Therefore we refer to separate documentation for each system. On Unix systems, this documentation is found on the manual page ‘saga.1’ using the man tool.
3.14 SAGA Standard Library

The SAGA standard library provides some assistance with error reporting and error recovery for users of SAGA. To load the package, enter the query

```
?- load(library(stdsaga)).
```

3.14.1 Error Reporting

We support routines on two different levels. `format_error_messages` is a high-level routine that can be adjusted with different settings. The other routines are lower-level and may be used when more control is needed.

```prolog
stdsaga.format_error_messages(+Messages, +Stream0, -Stream)
stdsaga.format_error_messages(+Messages, +Settings, +Stream0, -Stream)
```

displays a nicely formatted error message for the errors reported by the standard recovery. Messages may be the return status returned by SAGA, a single repair or a list of repairs. Stream0 is the stream to put the message on and Stream is the stream afterwards, for synchronization.

Settings is a feature record with functor settings. All the arguments that may be given to `format_error_position/5` are allowed as well as the following feature arguments:

`delete_string`: Format

( Default: 'Deleted marked token"i-n"' )

gives the atom that is used to print the deletion message with `io.format/4`. The format atom must handle the string representing the lexeme of the deleted token.

`insert_string`: Format

( Default: 'Inserted "s" before the marked token"i-i-n"' )

gives the atom that is used to print the insertion message with `io.format/4`. The format atom must handle three strings. The first and last strings are both the lexeme of the inserted token and the middle one is the lexeme of the token after the new one. We use the lexeme of the inserted one in two places so that either argument order is possible to emulate.

`replace_string`: Format

( Default: 'Replaced the marked token with "s""i-i-n"' )

gives the atom that is used to print the replacement message with `io.format/4`. The format atom must handle three strings. The first and last strings are both the lexeme of
the inserted token and the middle one is the lexeme of the
token it replaced. We use the lexeme of the inserted one
in two places so that either argument order is possible to
emulate.

recover_string: Format
( Default: 'Parse error"i"n')
gives the atom that is used to print the recovery message
with io.format/4. The format atom must handle the string
representing the lexeme of the first offending token.

fatal_string: Format
( Default: 'Parse error"i"2nFATAL ERROR -- parse aborted"n')
gives the atom that is used to print the fatal error message
with io.format/4. The format atom must handle the string
representing the lexeme of the first offending token.

stdsaga.format_error_position(+Lexeme, +Line, +Stream0, -Stream)
stdsaga.format_error_position(+Lexeme, +Line, +Settings, +Stream0, -Stream)
displays a nicely formatted error marker. Lexeme is the offending lex-
eme, Line is the line that the lexeme is placed on. Stream0 is the
stream to put the message on and Stream is the stream afterwards, for
synchronization. We also allow Lexeme to be the empty list to handle
unexpected events. This is marked at the end of the line. Settings is a
feature record with functor settings and some subset of the following
feature arguments:

line_name: LineName
( Default: 'Line "d: ')
gives the atom that is used to print the line number with
io.format/4. The format string must handle an integer
representing the line number. If no line number is wanted,
you must have "i" in the string to handle the argument.
You may use this argument for localization purposes.

marker_char: MarkerChar
( Default: 0'' )
is the character code for the characters printed on the next
line as underlining.

nl_char: NewLineChar
( Default: 10 )
gives the character code for the newline character. It can
be obtained by the agent newline_code/1 in the generated
module.
tab_char: TabChar
(Default: 9)
gives the character code for the tabulator character. It can
be obtained by the agent tab_code/1 in the generated mod-
ule.

tab_width: TabWidth
(Default: 8)
gives the width of the tabulator character.

width: Width
(Default: 80)
gives the maximum width the message should have. The
line number of the offending token is always displayed, even
if a small number is given as the width.

stdsaga.find_token(+Lexeme, +Line, +NewLineChar, -FirstPos)
stdsaga.find_token(+Lexeme, +Line, +NewLineChar, +TabChar, +TabWidth, -FirstPos)
is true when the lexeme Lexeme starts at position FirstPos on the line
Line. The left edge corresponds to position zero. -1 is returned if the
token cannot be found at all. For this to work properly, NewLineChar
must be the character code of the newline character in the file parsed. It
can be obtained by the agent newline_code/1 in the generated module.
TabChar is the character code for the tabulator character. It can be
obtained by the agent tab_code/1 in the generated module. TabWidth
is the distance between tabulator stops.

stdsaga.line_to_string(+Line, +NewLineChar, -String, -Spill)
stdsaga.line_to_string(+Line, +NewLineChar, +TabChar, +TabWidth, -String, -Spill)
is true when the line Line contains the characters in String. If the last
token on the line contains non-trailing newlines, then the part of
the token that “spill” is split into lines. Each extra line is put as an element
of the list of strings Spill. Each element in this list is a string that does
not contain newlines and that was surrounded by newlines in the input.
For this to work properly, NewLineChar must be the character code of
the newline character in the file parsed. It can be obtained by the agent
newline_code/1 in the generated module. If the arguments TabChar
and TabWidth are given, then a conversion is performed from tabulator
characters to a sequence of spaces. TabChar is the character code of
the tabulator character. It can be obtained by calling the agent tab_code/1
in the generated module. TabWidth is the distance between tabulator
stops.

stdsaga.line_to_line_number(+Line, -LineNumber)
is true when the line Line has the number LineNumber.
3.14.2 Error Recovery

We support both entire recovery strategies and parts of recovery strategies. The former may be used directly in an error-directive.

3.14.2.1 Complete Strategies

The following routines are ready to be used directly in an error-directive.

\[ \text{stdsaga.default_recovery(+RecoveryObject0, -RecoveryObject1)} \]

is an interface to the default error recovery mechanism.

\[ \text{stdsaga.rule_recovery(+RecoveryObject0, -RecoveryObject1)} \]

is an interface to the phrase-level recovery of the default error recovery mechanism.

\[ \text{stdsaga.halt_and_report(+RecoveryObject0, -RecoveryObject1)} \]

is an interface to the null recovery mechanism.

3.14.2.2 Parts of Strategies

The following routines provide a way to use either the standard local correction or the standard phrase-level recovery, but combined with other methods in any way.

\[ \text{stdsaga.local_correction(-Success, +RecoveryObject0, -RecoveryObject1)} \]

is an interface to the first phase of the default error recovery mechanism. Success is unified with false if we failed to correct the error and ok(NewStatus) if we succeeded, where NewStatus is the status after the correction. This agent does not send resume/1 or quit/1 to the RecoveryObject, so it is handy for multi-phase recovery.

\[ \text{stdsaga.rule_recovery(-Success, +RecoveryObject0, -RecoveryObject1)} \]

is an interface to the second phase of the default error recovery mechanism. Success is unified with false if we failed to recover from the error and ok(NewStatus) if we succeeded, where NewStatus is the status after the recovery. This agent does not send resume/1 or quit/1 to the RecoveryObject, so it is handy for multi-phase recovery.
4 Design Rationale

In this chapter we shall take a look at the various design choices made for SAGA and try to motivate them. The design choices covered are those that are visible to the user. Implementation specifics are not covered unless they affect the function of the program.

4.1 Initial Design Choices

Apart from the scientific motivation (see Section 2.5 [Motivation], page 12) the purpose of SAGA is to function as a practical tool for the Agents system. This means that common tasks should be simple to perform. It also means that rarely used features should not have too much negative impact on performance when they are not used. Also, they should not complicate the semantics unduly.

While SAGA is powerful enough to deal with natural language processing, it is mainly targeted towards parsing of programming languages. Therefore, all things needed to support languages like C or C++ should be provided. The following observations were made in the initial design stages:

- The generator should be at least LALR(1). We need to use existing grammars with a minimum of change.
- The generator should support at least the regular expressions provided by established scanner generators such as Lex.
- Lexical tie-ins\(^1\) are needed to handle many modern languages and must therefore be supported.
- EBNF saves a lot of user effort and should therefore be supported.

---

\(^1\) A *lexical tie-in* is when the behaviour of the lexical analyzer is changed from parser actions.
4.2 Basic Structure

The first important area of design is the interface between the lexical analyzer and the parser. One extreme is represented by the standard Unix tools Lex [Les75] and Yacc [Joh75]. Lex and Yacc are two completely separate tools that generate lexical analyzers and parsers, respectively. The other extreme is a scannerless parser where the lexical analyzer is totally integrated into the parser (see [SaC89] for an example of this approach).

Most programming languages are defined in a way that favours a rather clear-cut separation of the lexical analyzer from the parser. There are a number of reasons that support this tradition. We need to divide the two phases in some way, unless we have a very powerful parsing scheme. The main reason for this is that we need access to an entire token as look-ahead. Also, stripping the input of white space and comments usually requires a well-defined token level. To support lexical tie-ins we also need a way to modify tokens before they reach the parser. All these factors favour a design where the two phases are clearly separated in some way.

An argument for integration is based either on elegance or on user convenience. A uniform semantics may seem desirable from a theoretical point of view. A syntactic integration of the phases is convenient for the user since it saves a lot of redundant information that otherwise occurs at both ends of the interface. When designing SAGA, we found that completely uniform semantics is not as desirable as one might be tempted to believe. In the lexical analyzer, conflicts are common and are usually resolved completely automatically. In the parser we do not want this behaviour. Besides, we want to have a lot of special features in the lexical analyzer that does not generalize well to the parser.

To save the user some unnecessary work on writing an interface, she should be allowed to write regular expressions as terminals. If used with judgement, this should increase readability as well as reduce size. This was taken as basis for the idea that the terminals should be regular expressions. That is, we write regular expressions to denote tokens and do not name the tokens at all. Naming of tokens, when desired, is reduced to naming of regular expressions, which is desirable to support anyway. This approach makes it a little unnatural to have actions associated with regular expressions. A SAGA lexical analyzer can only handle pure and simple tasks. This view has many conceptual advantages, but some practical problems must be addressed. Usually, the actions are needed to strip white space and comments and to handle lexical tie-ins and other context dependencies. To support such constructs, SAGA supports the concept of filters that can replace sequences of tokens with other sequences of tokens.
For the reasons above, we chose a classical division of the syntax analyzer into a regular language lexical analyzer and an LALR(1) based parser, with the new notion of a filter to glue them together.

4.3 Dealing with Conflicts

Dealing with conflicts is an important task in any LR based parser. Later on we will discuss its importance for powerful extensions of the parsing power, see Section 4.4 [Advanced Parsing], page 86. Here we will discuss conflict reporting and explicit resolution of shift/reduce conflicts.

In most parser generators, the reporting of conflicts is poor. For example, Yacc outputs the set of LALR(1) items to a file. Finding the source of the conflict can require quite a lot of work. Worse yet, the user is forced to understand LALR(1) parsing quite well to navigate in the dumped file. For SAGA a more direct approach was desired. We chose to try to report conflicts by example. The idea was to display an example input that the parser would fail to handle. First, displaying a text string or a string of terminals was considered. By considering some examples, we found that it would be much more descriptive to allow non-terminals as well as terminals in the string. For example, the "dangling else" in a C grammar might be reported like this:

CONFLICT. Assuming input of the form:
    declarator "{" "if" "{" expr "}" statement
    With "else" as the next token there is an conflict between:
    [shift] 'try a longer match'

In our experience, this reporting scheme works very well for pin-pointing conflicts, and it should be fairly easy to understand with only a superficial knowledge of shift/reduce parsing.

Shift/reduce conflicts like the one above are often tolerable and the user should have some way to indicate so. Yacc only offers a way of informing the generator of how many conflicts to expect. This seems both inexact and dangerous. Also, each time a new conflict is introduced all the old conflicts must be re-examined, at least to the point where they are recognized as old conflicts. This seems like a waste of time. Instead, the tolerable conflicts should be individually indicated. A very natural way to do this seems to be to annotate the terminal that should be shifted. Annotation is done directly in the rule containing the token we want shifted.

Another common type of conflict resolution is that for operator precedence. The method used by Yacc seems to work rather well, so it was adopted without any principal changes.
4.4 Advanced Parsing

For some purposes, plain LALR(1) parsing is not powerful enough. Natural language parsing is the most obvious example of a difficult parsing problem. While natural languages are slightly outside the range of languages we require SAGA to handle, there are modern programming languages that are very complex to parse. The most prominent example is C++ [EiS90]. The parsing of C++ requires unbounded look-ahead, and actually not even that is quite sufficient. For example, there is no bound on how far ahead we have to look to distinguish an expression from a declaration. Thus, unbounded look-ahead is required. It is also possible that the entire expression may be either an expression or a declaration. In the latter case, the two possibilities are distinguished by the rule:

If it looks like a declaration, then it is one.

Thus, we need some way to express a preference for a particular interpretation. See also Section 3.12.1 [Conditional Rules], page 74.

Let us look at some of the methods used for these hard parsing problems and consider if and how they might be adapted to the SAGA setting.

4.4.1 Known Solutions

One way to deal with conflicts is to rewrite the grammar so that the different possibilities are merged. For example, the C++ grammar could be rewritten so that expressions and declarations are parsed by a common rule. This rule might build a parse tree for the new mixed construct. The parse tree is then examined as a post-processing step, to determine if it belongs to an expression or to a declaration. This approach has numerous drawbacks. It is difficult to generalize a grammar in this way. As a side effect, many illegal parses (e.g. constructions that are part declaration and part expression) will be made legal by the generalization. These must then be filtered out by post-processing checks. This is cumbersome, inefficient and hampers error recovery. We are also limited to actions that build parse trees. In short, we regard this approach as unsatisfactory.

The simplest way to handle the particular problems of C++ is to use a backtracking top-down parser. The backtracking gives us unbounded look-ahead, but it also allows us to choose between conflicting parses. The latter is, for example, necessary to handle some of the ambiguities in C++. We simply try the conflicting possibilities in order of preference. If we are to use this technique with SAGA we must adapt it to an LR setting in some way.

Backtracking top-down parsers have been used quite a lot in the natural language parsing community. Another approach to natural language parsing that has become
popular in recent years is non-deterministic LR parsing [Tom85]. Non-deterministic LR parsing amounts to trying all the possible alternatives when conflicts are encountered. For natural language parsing, blind don’t-know determinism pruned by unification of parsing attributes is reasonably efficient, compared with other techniques.

If we wish to adapt this technique for parsing of programming languages we probably want much more explicit control over the parse. It is likely that most of a language can be parsed with plain LALR parsing, so that non-determinism is needed only for a few language constructs. Unless we can explicitly commit to non-deterministic choices during the parse, we run a risk of severe performance loss due to combinatorial explosion of possible parses. Also, we need some way to decide what conflicts to report to the user and what conflicts to resolve with non-determinism. If most of the grammar is deterministic, the simple technique of resolving all conflicts with non-determinism will make it hard to detect mistakes in the design of the grammar.

4.4.2 Inspiration from AKL

Since SAGA’s host language AKL [Jan94] has powerful constructs to handle non-determinism, including the ability to encapsulate it, it was interesting to try to provide similar constructs in SAGA. AKL offers deeply guarded clauses of the form:

\[ \text{guard guardop body} \]

where guard and body are arbitrary goals and guardop is either wait ('?'), conditional choice ('->') or commit ('!'). If the guard is disentailed (known to be false), the clause cannot be run. If the guard is entailed (known to be true) the clause may be run, depending on the guard operator and the other clauses matching the current goal.

A conditional choice waits until the guard of the first clause is either entailed or disentailed. If it is entailed, the current goal is replaced with the body of the first clause. If it is disentailed, the first clause is disregarded and the process is continued with the second clause.

A committed choice may replace the current goal with the body of any matching clause that has an entailed guard (this amounts to don't-care non-determinism).

A wait-construction waits until only one of the clauses have a guard that is not disentailed or until a stable situation is reached. In the former case, the whole construction is replaced by the remaining clause (including the guard if it is not yet entailed). In the latter case, we split the computation into several possibilities. Simply put, a stable situation implies that the guards no longer can be affected by outside constraints. In a stable state, the computation cannot continue unless
we try the different alternatives by don't-know non-determinism in order to search for a solution. Both conditional and committed choice can be used to encapsulate non-determinism inside a deep guard, so that only one solution is picked.

Consider how a top-down, recursive descent parser would look in AKL. As usual with recursive descent, each non-terminal corresponds to an agent. Here we have the ability to control the non-determinism, that replaces the backtracking, by using the proper guard operators in the proper places. Assume that the rules have the same format as they have in SAGA, i.e. all the calls corresponding to grammar symbols precede the actions. Then it will be quite sufficient, although not optimal, to place the guard operators between the grammar part and the action part. It is sufficient, although not optimal, to use wait guards and conditional choice only, since committed choice does not accomplish much with respect to parsing.

For example, consider the following toy grammar for a tiny subset of natural language, written in SAGA syntax:

\[
\begin{align*}
  s & : np, vp. \\
  vp & : v, np. \\
  np & : v, np, p, n. \\
  np & : n. \\
  np & : n, p, n. \\
  n & : "adam" ; "eve" ; "a telescope". \\
  v & : "sees". \\
  p & : "with".
\end{align*}
\]

Assume that we only want to know if a sentence is possible to parse with this grammar. If we present the input in form of a list of atoms, an AKL parser may look like this\(^2\):

\[
\begin{align*}
  s-S & :- np-S, vp-S \rightarrow true. \\
  vp-S & :- v-S, np-S ? true. \\
  vp-S & :- v-S, np-S, p-S, n-S ? true. \\
  np-S & :- n-S ? true. \\
  np-S & :- n-S, p-S, n-S ? true. \\
  n-S & :- [adam]-S ? true. \\
  n-S & :- [eve]-S ? true.
\end{align*}
\]

\(^2\) Note that this is somewhat poor AKL style.
n-S :- [a_telescope]-S \ ? \ true.
v-S :- [sees]-S \ ? \ true.
p-S :- [with]-S \ ? \ true.

We used AKL accumulator syntax quite heavily in the above example. Each ‘-S’ above actually denotes a pair of variables. For example, the clause for p is equivalent to:

\[ p(S1, S2) :- S1= [\text{with}\mid S2] \ ? \ true. \]

The conditional choice guard operator is used in the first clause to choose the first solution found. The other clauses in this example are all non-deterministic. There are also situations where you want to use conditional choice more directly. For instance, we might see the following in a C++ grammar:

\begin{align*}
\text{statement-S} & :- \text{declaration-S} \rightarrow \text{true.} \\
\text{statement-S} & :- \text{expression-S} \rightarrow \text{true.}
\end{align*}

This amounts to testing for a declaration first. If that fails, then we test for an expression.

### 4.4.3 Adaptation to LR Parsing

The usual way to view LR parsing is as a bottom-up technique that is guided by a viable prefix automaton. Another view of LR parsing, that is somewhat less correct but quite pleasing in a constraint oriented setting, is as a highly non-deterministic top-down parsing technique. We might imagine that the top-down scheme is augmented with some very powerful constraints that are capable of eliminating some non-productive branches in the search. In particular, infinite branches from left recursion would be eliminated.

This view is in part an illusion, but in SAGA it is reasonably solid as a mental model. In other systems it is less convincing, due to the differences between inherited and synthesized attributes. In SAGA most of these differences are hidden by the use of logical variables. This view gives us hope of constructing a semantics for non-deterministic LR parsing with guard operators that will behave in a comprehensible way in most situations.

There are some problems to solve if we are to adapt the behaviour of the non-deterministic top-down AKL parser to an LR framework. Apart from the obvious problem of finding some way to implement the scheme (see Section 5.5 [Non-Determinism], page 114), we must find some way of separating the deterministic parts from the non-deterministic ones, i.e. to determine which conflicts to report
and which to resolve with non-deterministic choices. Also, we must decide the precise semantics for the conditional choice.

Separating the deterministic and non-deterministic parts of the grammar is easy. A grammar rule that does not have a guard operator should not be involved in any conflict. An alternative way of looking at it is to consider the separator between the grammar symbols and the action (‘*’) to be a kind of guard. This “guard” should behave as the wait-operator, but force the conceptual “non-determinism” to be resolved by the conceptual “LALR(1)-constraint solver”. Use of this guard operator will cause conflicts to be reported.

The precise choice of semantics for the conditional choice guard requires some consideration. At some point we should temporarily abandon all the clauses except the first in the conditional choice. We call this point a choice point. At the choice point the computation branches in different directions. The branch that belongs to the first clause is tried first. If its execution fails, we return to the choice point and try the next, and so on. The problem is: When is this choice point to be created? The simple and straightforward answer is to create the choice point at the leftmost terminal that might belong to one of the branches. This choice means that when we reach the left edge of something that matches a conditional rule, we must know (with one token look-ahead) that the input actually should be matched by one of the clauses in the rule. If the rule is not a conditional rule, we do not decide this until the right edge of the matched input. This means that the creation of a choice point restricts this part of the grammar to be LL(1) rather than LR(1).

One might consider other possibilities that would allow a more LR like creation of choice points. It seems like such attempts result in very complex semantics, and they also seem hard to implement. Besides, the LL(1) constraint is not as strange a beast as it may sound like. Creating the choice point is analogous to executing an action. The parser must know that an action is needed in the current derivation before it is executed. It seems reasonable to expect the same to hold for trying out a conditional choice. Also, it seems like the simple solution gives us all the power that is desirable in practice.

4.5 Error Handling

The ability to handle incorrect input in a graceful manner is important for many parsing applications, especially for those concerned with programming languages. It is also one of the hardest tasks to accomplish, both in parsers written by hand and in machine generated parsers. In this section we will give some background on the topic. We will also describe the techniques chosen for SAGA, and try to explain why they were chosen.
4.5.1 Error Reporting, Repair and Recovery

Dealing with erroneous input has several aspects. We want to determine the location and nature of the error for error reporting, i.e. to display some informative message to the user of the parser. Furthermore, we want to repair the error if possible, i.e. to make a change of the input to obtain an input that is similar, parsable and hopefully what the user intended.

A successful repair will allow us to continue parsing with a reasonable hope that the modified input is close enough to what was intended, in order to avoid an avalanche of spurious errors; errors that are caused by the error handling process itself.

If we fail to repair the error, we still want to continue to parse. We want to discover as many errors as possible on a single pass through the input, without introducing too many spurious errors. We want to recover from the error, i.e. to find a safe state for the parser and a location in the input where we are reasonably sure we can resume parsing.

4.5.1.1 Error Reporting

Locating the place of the error is simple in the sense that we can easily identify the longest prefix of the input that can be complemented to a syntactically correct string. In fact, an LR parser does this automatically. When the parser enters a state in which no reductions can be performed and a position in the input where the look-ahead token cannot be shifted, we know that no suffix starting with the current look-ahead can make the input legal. More important, we know that there is a legal suffix starting with some token other than the current look-ahead token. This means that we detect errors as soon as theoretically possible. In parser terminology, an LR parser is said to have the viable prefix property.

However, this does neither mean that the position in which we detect an error is the actual source of the error from the users perspective, nor that it is the best place to attempt to repair or correct the error. It is quite possible that we have to discard the rest of the input entirely and replace it with something else in order to turn it into a suitable suffix. At the same time, changing a single token earlier in the input might very well suffice to make the rest of the input perfectly valid. Of course, it is in no way certain that such a single token change is what the parser user would do to correct the input, even though it might be quite probable that it is.

Assuming we actually found the “correct” location of the error, we must still determine the kind of error we have found in order to generate a constructive error message. The classic error message ‘parse error’ is easy to generate, and it has the distinct advantage that it seldom misleads the user, but it is not very helpful.
4.5.1.2 Error Repairs

When we consider what repairs to attempt, we should know what errors the users typically make. Ripley and Drusekis [RiD78] found that on student Pascal programs, 90% of all errors were single token errors and that 80% of all erroneous statements only had a single error. These results may or may not be representative for other languages or other categories of users, but they certainly gives us some ideas regarding repairs. It suggests that we can come a long way with extremely local repairs, involving only a single token or maybe a few tokens that belong together.

The problem with repairs is that they, if incorrect, can confuse the user and do more harm than good. This phenomenon will of course be more apparent for larger repairs, and the high frequency of the single token errors is a strong indication that the risk of misleading the user is not justifiable for larger repairs. Thus we should settle for small repairs. This view seems to be common, see [GHJ79], [Dai85]. In this text, we call repairs involving just a few tokens of input local repairs.

4.5.1.3 Error Recovery

Given the limitations of the repairs we want to attempt, it is very important that we are able to continue the parse in the event of an error that we fail to repair. We need to avoid spurious errors, but at the same time we want to be able to resume parsing as soon as possible. The extreme solution of stopping at the first error (skipping the entire input!) is undesirable in many, but not all, applications. The worst possible scenario at the other extreme is when so many spurious errors occur that the user ignores all but the first anyway.

Local repairs are often implemented on a generate and test basis, where different repair actions (such as deleting the current input token, replacing it with another or inserting a new one) are tried and evaluated before possibly selecting one of them. A common technique for evaluating a repair candidate is to do a forward move. The parser is run a few steps forward on a copy of the parser state to check that no new error is detected too close to the current one. If a few, say five, input tokens could be successfully shifted, then we may judge the repair to be a good one [Dai85].

4.5.2 Pros and Cons of Automatic Error Handling

Error handling is usually a rather ungrateful task that is usually dealt with late in the development cycle. It would be a great advantage if the parser generator allowed good error handling to be implemented with a minimal amount of work.

Ideally, we would like the generator to make all the error reporting, repair and recovery decisions from the grammar alone, without troubling the user of the generator.
with these complicated issues. For pure grammars this would be a hard enough problem, but for a practical parser for a programming language it is an impossible one.

In practice, we actually create some data structures from the parse, and call user code that manipulates them in arbitrary ways. So when we do error repair or recovery, we need to construct reasonable data for the user code to work on, even from erroneous input. Unless we demand that the user code should be prepared to handle strange things from an automatic synthesis, at all times, or require the user of the generator to help with said synthesis, the kinds of corrections we can perform becomes severely limited.

Recall that SAGA uses constrained attributes, see Section 2.5 [Motivation], page 12. This means that the approach of letting the user deal with strange synthesized data is unworkable. It is undecidable if an attribute to a grammar rule is inherited or synthesized, or possibly some subtle combination of both. The reason for this is that all attributes are logical, constrained variables so they lack a direction of assignment. (An exception to this is local repairs, since the semantic values of tokens are known to be synthesized attributes.) Because of this, a synthesizing approach would limit the power of the parser, and this is clearly intolerable.

Even in generators that do not have the special problems caused by constrained attributes, fully automatic error handling is hardly considered. Even authors that claim that their schemes are automatic usually manage to sneak in some user supplied information somewhere. For instance, Julia Dains well-written paper [Dai85] discusses automatic recovery, but her scheme does require information on how to synthesize semantic attributes for different rules.

### 4.5.2.1 Automatization of Error Reporting

Even error reporting becomes hard to do well without any assistance from the grammar writer. We can point out the position at which we detected the error, but it is difficult to give an informative message about the nature of the problem. We may list the kind of construction we expected to see, and the kind of construct we failed to recognize, but it is hard to know what names to use in these references. The names used in the grammar might be used; PCCTS [PDC92] does this kind of error reporting. However, PCCTS is LL(k) based, so this information tends to be less complex than for an LR based parser. The messages can be very confusing, especially if we use names of non-terminals of a complex grammar. Not even under the best of circumstances, when we have a nice, pure grammar with descriptive names, can we be certain that it is a good idea to use the names from the grammar in the output.
When we perform local repairs, a message describing the changes made can be of much help to the user. Even if the repair is incorrect, it supplies valuable hints. Displaying token names suffers from the same problem as displaying names of non-terminals, but keywords, operators and so on can always be displayed in the proper way.

4.5.2.2 Automatization of Local Repairs

Local repairs are reasonably suited for automatic use. There is never any problem to delete tokens. To insert or replace tokens, however, requires some creative effort. We can safely insert tokens with a fixed lexeme, such as operators and keywords, but there is a problem when we want to insert things like constants and variables; namely to generate a suitable lexeme. With access to the tables of the lexical analyzer, as we have in SAGA, we might actually synthesize a lexeme for a variable or constant, but it might be undesirable. Insertion of an arbitrary identifier could cause spurious semantic errors, complaining about types, lack of declaration or other mismatches. It is quite possible that the user will be confused about messages like:

Line 2: Replaced "union" with "A".
Line 2: Identifier "A" not declared.

that might be the result of accidentally naming an identifier with a reserved word.

Note that the second error message cannot be avoided in a fully automatic setting, since it is generated by user code and thereby independent of the first message that is created by the automatic repair. This type of problem gets much worse if we perform some syntactic checks in the semantic phase. Such checks are not uncommon, either to simplify the grammar or to improve error reporting, or both [GHJ79].

Generating spurious semantic errors may not seem all that bad, but it may be argued that they are bad as a result of local repairs. Local repairs can be reported in an exact, informative manner and the user should usually be able to trust them to be reliable. This trust is severely compromised as soon as they start to generate spurious semantic errors.

4.5.3 Phrase Level Recovery

Given that fully automatic phrase level recovery seems unpractical, we need to help the generator along. Since error recovery involves construction of syntactic groupings, it seems natural to somehow handle it with grammar rules. This is the idea used as the only error handling scheme in Yacc. This is done by supporting error productions, where you essentially specify errors as parts of a rule.
In Yacc, 'error' is a reserved word that matches any incorrect input. This is an elegant and powerful idea, but it is hard to exploit all the potential power in practice. The consequences of adding new rules to a complex grammar can be hard to understand and it may introduce subtle ambiguities into the grammar.

Interestingly enough, the problems with ambiguities seem to be linked to repairs rather than to recovery, so combining error productions with a local repair strategy can be very successful [GHJ79]. Used purely for recovery, error productions makes it easy to give meaningful error messages and to select the "major" non-terminals on which to base the recovery.

### 4.5.4 SAGA Error Handling

An error handling scheme that forces the writer of the grammar to do a lot of work runs the risk of remaining unused. On the other hand, a simple scheme that lacks flexibility may be useless for some applications. This can happen either because the scheme makes assumptions that are untrue for the application at hand, or simply because a better error handling is needed.

The solution is to offer both flexibility and power. For maximum flexibility, SAGA allows the grammar writer to specify her own error handling scheme. For maximum ease of use, some ready-made schemes are available, and one is used by default. The default strategy is to use a simple, but automatic, local corrector and to use error-rules for recovery.

A user defined corrector can make different changes to the input, and may also delete items from the parse stack. Different changes may be tested with forward moves. In order to interface the agent with the parser state, an AKL port object [Jan94] is used. This roughly corresponds to the object concept in object-oriented languages. The kind of messages the object can handle include "try an n step forward move" and "delete the look-ahead token".

#### 4.5.4.1 The Default Error Handling

The default local repairs are done by generating candidate changes to the input and then testing them with a forward move, where five shifted tokens counts as a successful repair. First, insertion of a new look-ahead token is tried, then replacement of the look-ahead token, then deletion. All insertions, and replacements, are limited to tokens that have unique lexemes.

Phrase level recovery is invoked if no local repair succeeds. The error token is predefined as a virtual token. A virtual token is a token that may not actually appear on the input, see Section 3.6.2 [Regular Expressions], page 47. It is pushed
on the input, and items are discarded from the parse stack until a shift move can be made. This corresponds to finding an applicable error production. After the shift, input is discarded until a five step forward move is possible.

Repairs are returned as part of the result of the parse. The writer of the grammar has full control over how they are reported. Some library functions are offered to produce nice messages easily. One of these displays the offending line and an indication of the location of the error along with the line number and a message indicating the correction made, like this:

```
Line 42:    a[i] = b[i+1];

Replaced the marked token with ""]"
```

4.5.4.2 Custom Local Repairs

The grammar writer can make three kinds of contributions to the local repairs. First, it becomes possible to insert arbitrary tokens. It is possible to generate sensible names. It is also possible to stop the generated names from being mentioned in semantic error messages. Second, the grammar writer may use information on the language to decide which one of several repairs is the “best”. Third, the grammar writer may design a few special repairs that possibly involve many tokens, but where you have strong reasons to believe them to be correct. An example of this may be an unclosed comment, where you may start at the beginning of the comment and discard tokens until a forward move succeeds.3

3 Starting at the beginning of an unclosed comment is not possible with most lexical analyzers, mostly due to buffer overflow. In SAGA it can be done, since we can describe a comment as a (possibly huge) token. Then all we need to do is to supply a rule for unclosed comments that match just the beginning of the comment.
5 Implementation

In this chapter we describe some key points of the implementation of SAGA. Much of the implementation is based on well-known techniques described in [ASU86]. The reader is expected to be familiar with the material on LALR parsing covered in Chapter 4 of that book. The reader is also expected to know basic automata theory, especially covering regular expressions and deterministic finite automata. This material may be found in Chapter 3 of [ASU86], but any good text on automata theory might suffice.

Many parts of the implementation are very straightforward. For example, the parsing of the SAGA input file is made by a SAGA generated parser and is straightforward, albeit rather extensive. The various checking and preprocessing steps are also straightforward. The user manual's description of accumulator expansion and extended BNF should be enough to understand how these things are implemented.

To describe the implementation of SAGA, we first outline how the generated parser (except for error handling and non-determinism) is implemented, see Section 5.1 [Runtime System], page 97. Then the different parts of the generator are considered. See Section 5.2 [Building of the Lexical Analyzer], page 105, for the construction of the DFA used for lexical analysis. See Section 5.3 [Building of the Parser], page 107, for the construction of the LALR parser. See Section 5.4 [Table Packing Algorithm], page 111, for the packing of the tables used in the runtime system. Finally, non-determinism is discussed, see Section 5.5 [Non-Determinism Implementation], page 114.

5.1 Runtime System

SAGA is based on a table driven lexical analyzer running as a subroutine to a table driven LALR(1) parser. The decoding of the packed tables are performed in C rather than AKL (for performance reasons). With the current Agents system, the code for low-level tasks such as this becomes painfully slow in AKL. Due to the small piece of C code used, an acceptable level of performance has been reached. A generated SAGA parser for AKL was found to be about as efficient as an AKL parser hand-coded in Agents, despite the fact that the generated one offers tracking of line numbers and line contents while the hand-written does not.

We start by outlining how the lexical analyzer operates. Then the format of the lexical analyzer table is described. The parser is very straightforward, so it does not warrant any special discussion. Therefore, only its table formats are described. Finally, we describe how the filters between the lexical analyzer and the parser works.
5.1.1 Lexical Analyzer

A SAGA lexical analyzer is based on a deterministic finite automaton (DFA) represented as a packed table. The basic mechanism is simple. The table is indexed by the current state of the automaton and the current look-ahead character. The table contains the state the automaton should enter before the next character is read.

5.1.1.1 Needed Features

Several factors complicate the simple scheme sketched above. A straight-forward implementation of the table as a two dimensional array would be very space consuming. Therefore, a packed — but fast — representation must be used. See Section 5.1.2 [Packed Tables in the Lexical Analyzer], page 101. We also want to avoid reading a look-ahead character if it is known that this character cannot be a part of the matched lexeme. The reason for this is that we might be reading from an interactive input, such as a terminal. For example, if the input is line-by-line we do not want to read past the newline character that terminates the line.

Another problem is that we wish to maintain some information about the structure of the lines of input we have read. We want the lexical analyzer to count the lines and to keep track of the contents of each line. This information is useful for error reporting purposes. This requires some book-keeping that may be entirely unsynchronized with the tokens read, since newline characters are quite feasible in the middle of lexemes.

Like other lexical analyzer generators, SAGA allows a wide range of expressions that goes beyond the regular expressions used in traditional automata theory. The need for them is basically due to the fact that a lexical analyzer classifies sequences of tokens rather than a single token. We must support several lexical modes, and switching between them. We must support context information about whether a lexeme starts at the beginning of a line or not, and whether the characters after the lexeme proper matches a certain regular expression (trailing context) or not.

5.1.1.2 Structure of the Lexical Analyzer

A SAGA lexical analyzer can be viewed as consisting of two parts. We have the scanner proper, which fetches characters and changes DFA states. We have a wrapper part to handle all the extra features, such as lexical modes, context information and counting of lines. This is done by modifying the DFA so that each special feature occurs at the end of a “token”. Here “token” refers to the viewpoint of the DFA. Several such “tokens” may together form a token at the user level.
To avoid reading unnecessary characters, transitions to states that have no outgoing transitions are marked to indicate this. When such a transition is made, the inner loop surrenders control to the wrapper. The lexeme is taken to be the read characters plus the look-ahead character. In the normal case, the inner loop continues to run until no transition can be found. This means that the look-ahead character does not belong to the lexeme, so an extra look-ahead have been read.

Lexical Modes

The handling of lexical modes is quite straight-forward. Each lexical mode corresponds to its own starting state in the DFA. Switching of lexical modes within the lexer only requires that the starting state of the new mode is stored in the table along with a flag that indicates that switching should be performed. To support changing of the mode from an outside action, we use a table to convert the name of the lexical mode to the corresponding starting state at runtime.

Line Structure Dependency

To handle counting of lines and beginning-of-line contexts, the transitions on newline characters are treated in a special way. The line counting is done by cutting the match short after seeing a newline and then, after the proper actions have been executed, restarting it in the correct state. When the match actually ends with the newline character, we prepare to handle beginning-of-line contexts for the next token. This is done simply by pushing a special character (outside the normal character set) on the input so that it becomes the first character of the next token. All regular expressions that do not require beginning-of-line context are augmented with an optional beginning-of-line context. This way the context is handled almost transparently. The only catch is that the special character must be deleted when found, so it does not become part of the matched lexeme.

Dummy Transitions

To handle the interaction between the wrapper and the scanner proper, we insert dummy transitions, special transitions on non-existing characters. In final table, these are represented by certain mark bits in the actions. They are used to allow references from an action to a state. These dummy transitions allow us to restart at the correct point after stripping a beginning-of-line special character, or after handling a newline.

Dummy transitions are also used when matching trailing contexts, where they refer to the starting state of a part of the automaton that matches the trailing context.
The actions that correspond to tokens with trailing contexts are marked and contain references to states that correspond to the trailing context parts so that they can be checked. At runtime, when we find an accepting state with a trailing context bit, we try to match at the indicated state. Afterwards, we put the matched characters back on the input. If the trailing context failed to match, we run the ordinary action for the state.

5.1.1.3 Lexical Backtracking

Whenever we reach a non-accepting state that has no transitions on the current look-ahead character we are forced to backtrack. This amounts to finding the longest proper prefix of the characters read that can be taken as a token. We add a default matching rule (any single character) to make sure that such a prefix always exists.

In SAGA, lexical backtracking is handled in a rather uncommon fashion. First, we make an optimistic attempt to match. In this phase no lexical backtrack information is maintained at all. The characters read are collected in a list. This list becomes the lexeme, provided that the match succeeds. If the match fails, so that the need for lexical backtracking arises, the list is used as input for the second phase. In this phase, the matched characters are used to build a list corresponding to the reverse of the lexeme, and the states visited are collected on a stack. This allows for easy backtracking. When the final match is found, part of the reversed form of the lexeme is re-reversed into the final token.

This scheme was chosen for efficiency. First note that since AKL is basically declarative, there is no way of obtaining a prefix of a list (short of copying it). Hence the use of a reversed token. The chosen scheme usually avoids this overhead. Since most matches do not need lexical backtracking, it is sensible to make the non-backtracking matches as fast as possible.

The drawback is that matches that do need backtrack must traverse the input twice. However, the second phase is much faster than a single phase would be, because a lot of tests can be omitted. For instance, there is no need to check for the end of the buffer, the end of the input or successful matches. This means that non-backtracking matches becomes much faster and backtracking matches only marginally slower with this scheme.
5.1.2 Packed Tables in the Lexical Analyzer

If we consider an unpacked table for a lexical analyzer, we are likely to find quite a lot of redundancy. Consider a two dimensional array, where each row corresponds to a state and each column corresponds to an input character. The entries in the table are numbers of the states that are results of transitions from the state corresponding to the row, on the input corresponding to the columns. If no such transition exists, the cell is empty.

5.1.2.1 Sparseness

The first type of wasted space is due to the empty cells. They are likely to be frequently occurring. The idea is to allocate the rows freely, so that the non-empty cells in one row may be allocated in the empty cells of another row. The problem with this approach is that we must be able to distinguish between empty cells belonging to one state and the proper entries belonging to another. This problem is solved by storing both the number of the state we transfer from and the number of the state we transfer to. Note that the distances between the filled cells in one row is preserved in the packed format. We use a table of base addresses to find the offset of column zero of the desired row. Then we add the input character to this offset and check the indicated cell. If the state number indicating the state transferred from matches the current state, then we have found a transition. If it does not match, then we have found an empty cell.

We follow the traditional approach [ASU86] and use two arrays: the check array for the state we transfer from, and the next array for the state we transfer to. We might as well have used an array of records, but two arrays are slightly more space efficient, since all trailing empty cells can be truncated from the next-array but not from the check-array.

5.1.2.2 Other Redundancy

Another kind of redundancy occurs when two rows are similar. Quite often only a few columns differ between a set of rows. It may also happen that the most common entry in some row is some particular transition, rather than the empty transition. To handle these cases, we attach an extra column for defaults. This column may either contain a constant value, the most common target for transitions in the row, or it may contain the number of (the row of) a similar state. If it contains a constant value, all the entries with this value may be replaced with empty cells (all the originally empty cells are filled with a special code for empty, called blank). This may permit tighter packing by increasing sparseness. When we look up a value
and find an empty (but not explicitly blank) cell, the value is taken to be the default value.

When we find an empty cell in a row that has a default state, we do not return a constant value. Rather we return the corresponding entry in the row for the default state. This means that the entry in column \( i \) of the row may be replaced by an empty cell if (and only if) column \( i \) of its default row contains the same entry. Since empty cells are looked up anew in the default state, we must take special care if column \( i \) in the row is empty, but column \( i \) in the default is not. In this case, explicit blanks must be inserted to stop the look-up in the default state.

For example, consider this fragmentary state table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>( \cdots )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>State 41</td>
<td>43</td>
<td>44</td>
<td>40</td>
<td>40</td>
<td>47</td>
<td>40</td>
<td>( \cdots )</td>
<td>40</td>
</tr>
<tr>
<td>State 42</td>
<td>43</td>
<td>44</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>( \cdots )</td>
<td>40</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
</tbody>
</table>

Clearly, state 41 and 42 have a lot in common. Also, they are both dominated by transitions to state 40. Assume that state 42 is chosen as default state for state 41. Further assume that state 42 is assigned 40 as default value. Then the table would be much more sparse:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>( \cdots )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>State 41</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td></td>
</tr>
<tr>
<td>State 42</td>
<td>43</td>
<td>44</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td></td>
</tr>
</tbody>
</table>

This sparse table might then be packed quite tightly in the next and check arrays. We need to store the defaults somewhere, in this case at offset -1. Part of the arrays
may look like this:

To make the look-up process efficient, a state that is used as a default state may not itself use another default state. This means that we limit the number of array references to a maximum of five:

1. Access the base-array for the offset.
2. Access the check-array to see if the cell is empty or not.
3. In the worst case, the cell is empty. Thus, we access the default entry.
4. The worst case is a default state. This means yet another access of the check-array.
5. Access the next-array if the cell was non-empty. If it was empty, access the default entry instead. This cannot hold a default state, so this is the final reference.

Of course, we do not always need to make five array references. It is also possible to get by with three:

1. Access the base-array for the offset.
2. Access the check-array to see if the cell is empty or not.
3. Access the next-array if the cell was non-empty. If it was empty, we access the default entry instead. If this holds a constant value, then this is the final reference.
5.1.2.3 Improvements

The scheme may be improved upon. In SAGA we do not use the state numbers (row numbers) at all, but rather we use the actual base addresses. This means that we can omit the base array. This way we get either two or four array references. In the scheme described in [ASU86], the default entries are put in a separate array. Our suggested optimization is therefore unusable in that setting. In SAGA the scheme without the base array works well, but both the default value and the code for the accepted token must be stored as columns. Since they occupy a fix column, the check-array need not contain the state address for this particular entry as long as the check-entry does not contain the address of any other state. This allows us to store the token code in the check-array (coded as a negative number, so no conflict can arise). We use column -2 for these special purposes\(^1\). In some cases we need to use column -3 as well, to store the addresses of trailing contexts and new lexical modes.

5.1.3 Packed Tables in the Parser

The parser tables are packed in much the same fashion as the table of the lexical analyzer, only simpler. We have two different parser tables, the action table and the goto table. The former indicates the parsing action to perform, and the second indicates the new state after a reduction has been performed.

The action table is packed with the check-next scheme described for the lexical analyzer, but no default states are used. We do use the base array, partly since we need the consecutive numbering for accessing the goto table, and partly because identical rows are common. The default mechanism indicates a default reduction for each row. In parser tables, there is no need to distinguish between blank entries and the default reduction, so this entry is simply the most common reduction in the row, and empty only if there are no reductions in the row.

In the goto table, each row corresponds to some reduction. The columns are indexed by states, hence the need for consecutive state numbers. We use a base array and default values but no default rows. In the goto table, no naturally empty entry is ever accessed, so they are only relevant for elimination of default values.

\(^1\) Column -1 is used for end-of-file, therefore the choice of -2.
5.1.4 Filters

From an implementation standpoint, there are three kinds of filters: No filter at all, a simple filter that just discards the token, and the general purpose filters. The first two cases are handled in the obvious fashion in the lexical analyzer, but the third is part of the parser. A general purpose filter just calls a piece of Agents code generated for that particular filter. The implementation is quite simple, due to the fact that regular expressions inside Agents code in the SAGA file are compiled to a data structure that identifies the desired token, optionally augmented with the lexeme and context arguments. The filter code just creates a list of such data structures. At runtime the structures are translated to the corresponding internal representation. If the lexeme is provided by the user, it is used. If it is not provided, the lexeme of the input token is used. The context argument is treated in a similar way.

5.2 Building of the Lexical Analyzer

The heart of the lexical analyzer generator is the part that transforms regular expressions into a DFA. Some extensions must then be made, to handle the matching of different kinds of tokens and so on.

5.2.1 Regular Expressions to DFA

The algorithm that SAGA uses to generate a DFA from a regular expression is not based on the algorithm in [ASU86], but rather on a simpler method based on transformation of regular expressions. The idea is outlined in [Hop93]. This algorithm is easy to understand, implement and to prove correct. Also, we feel that it is more fundamental than the standard. Essentially it transforms a regular expression to a regular grammar, that is then made into a DFA.

The basic idea is to transform the regular expression in certain ways until a very simple form is reached. This form corresponds to a regular grammar. The goal of the transformation is to transform the expression \( R \) into the form:

\[ a_1 R_1 \mid a_2 R_2 \mid \ldots \mid a_n R_n \]

or the form:

\[ \epsilon \mid a_1 R_1 \mid a_2 R_2 \mid \ldots \mid a_n R_n \]

where \( \epsilon \) is the empty regular expression, and \( a_1 \) to \( a_n \) are different single characters. Henceforth this form is called normal form, and \( a_i \) is called the head of \( a_i R_i \), for any \( i \). Note that the normal form corresponds to the definition of one non-terminal \( R \) in a regular grammar. Since we have a direct correspondance between non-terminals
in such a grammar and states in a DFA, the normal form also corresponds to a state in the DFA. If the empty expression is present, the state is accepting. Each of the non-empty disjuncts corresponds to a particular transition.

The regular expression is transformed to normal form. When the normal form is reached, the first non-terminal (or state of the DFA) has been generated. Then the process is continued on $R_1$ to $R_n$ and so on. In order for the generation to terminate, it is sufficient to keep track of regular expressions that already have been transformed to normal form. A regular expression that has been translated before is not translated again. This corresponds to finding loops in the regular grammar, or in the DFA.

The transformations used are the following:

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A?$</td>
<td>$\epsilon \mid A$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>$\epsilon \mid A^*$</td>
</tr>
<tr>
<td>$A+$</td>
<td>$AA^*$</td>
</tr>
<tr>
<td>$A?B$</td>
<td>$B \mid AB$</td>
</tr>
<tr>
<td>$A^*B$</td>
<td>$B \mid A^*B$</td>
</tr>
<tr>
<td>$A+B$</td>
<td>$A(A^*B)$</td>
</tr>
<tr>
<td>$(AB)C$</td>
<td>$A(BC)$</td>
</tr>
<tr>
<td>$(A\mid B)C$</td>
<td>$AC \mid BC$</td>
</tr>
</tbody>
</table>

The transformations are obviously sound and always take us closer to the normal form. The only important transformation not in the table concerns the merging of disjunctions. When no transformation can be applied, we always have a disjunction where each disjunct is of the desired form. However, we may have the same character in the heads of several disjuncts. We group the disjuncts with the same head together, merge any identical disjuncts and apply the following merge transformation repeatedly:

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$aA \mid aB$</td>
<td>$a(A \mid B)$</td>
</tr>
</tbody>
</table>

So far we have only given transformations, without any indication of what order they are to be performed in. Transformations are applied to the entire expression repeatedly, until it becomes a disjunction. Then the algorithm is called recursively on all the disjuncts and turns them into normal form. The last transformation step is to merge the normal form versions of the disjuncts.

Finally, the generated DFA is minimized with the standard partitioning algorithm described in [ASU86] (Section 3.9, Algorithm 3.6).
5.2.2 Turning a DFA into a Lexical Analyzer

It is not enough to be able to build a DFA. We also need to keep track of the actions associated with different tokens. To be slightly more general, we allow regular expressions and states in the DFA to be associated with a set of marks. A mark can be any term, but we only use marks to indicate starting states for various lexical modes, to indicate accepting states, to indicate which token to accept and to indicate contexts and switches of lexical modes.

The use of marks makes it simple to transform the DFA in various ways without losing information. In particular, it is easy to minimize the DFA. With some other approach, the actions and various lexical modes may cause an additional amount of book-keeping. The use of marks avoids complicating minimization in any noticeable way.

The big issue is to find the correct states to mark with the different accepting actions. To do this, we allow marks to be attached to regular expressions. The transformations do not affect the marks, except when disjunctions are created. When a transformation results in a disjunction, the marks are copied to each of the disjuncts. When disjunctions are merged, the union of the sets of marks becomes the new set of marks.

A state inherits the marks from the normal form regular expression it was created from, if and only if it can match the empty string. The top level regular expression is taken to be the disjunction of the regular expressions for each token type. The top level expression does not itself have any attached marks. Each of the disjuncts in the top level expression is assigned the marks for its corresponding token type.

There is a possibility of conflicting marks. If a state is marked with more than one token type we have a conflict. The subset rule (see Section 3.6.1.2 [How the Input is Matched], page 46) is used to resolve some of the conflicts. Any pair of conflicting token types is checked. The sets of states with the respective token types are collected. If one set is a subset of the other, the token type with the smaller number of states is given precedence. If neither set is a subset, then a conflict is reported to the user of the generator.

5.3 Building of the Parser

The generation of the parse tables follow the algorithms from Chapter 4 of [ASU86] quite well, but the computations have been restructured to take advantage of common computations. Due to this restructuring, some of the suggested optimizations could be omitted without performance loss. We also use a propagation algorithm that is different from the one used in [ASU86].
5.3.1 Overview

In [ASU86] it is suggested that parsing actions and goto transitions should be determined from the kernel items without the use of an explicit closure operation. This eliminates the need to ever generate any non-kernel items, and is therefore considered an optimization.

SAGA uses the full and explicit closure during the generation of new sets of kernel items. In all other contexts, the sets of items are represented by just the kernel items. The idea is to do an LR(1) closure operation and generate all the goto information and all the look-ahead propagation information at once. Each item in the closure corresponds to either a reduction or a kernel item in one of the states in the range of the goto function. This means that this simple approach is provably at least as efficient as the one in [ASU86] in the Ordo sense.

SAGA then uses a non-iterative algorithm to propagate the look-ahead sets, instead of the iterative one in [ASU86]. The first thing that is done in the generation is to compute two tables that represents the functions Nullable and First, both with the non-terminals as domain. Nullable decides if a non-terminal can derive an empty string of terminals. First returns the set of terminals that can occur at the beginning of any non-empty string of terminals that can be derived from the non-terminal.

5.3.2 Computing the Nullable Predicate

Nullable is computed by propagation of boolean values through a directed graph constructed from the grammar. The graph has two different kinds of nodes, or-nodes and and-nodes, corresponding to the alternative and concatenate operators in the grammar, respectively. The direction of the edges of the graph is from the leaves and up towards the start symbol, i.e. all edges go from grammar symbols to non-terminals.

In the following, upper-case letters denote non-terminals, lower-case letters denote terminals, $\epsilon$ denotes the empty string of terminals and $\alpha$ and $\beta$ denotes strings of terminals. In precise terms, the graph is constructed like this:

- There exists exactly two nodes that have no arriving edges, one with the value true and one with the value false. These are called the true-leaf and the false-leaf, respectively.
- Each non-terminal $A$ corresponds to an or-node.
- Each production of the form $A \rightarrow \alpha$ corresponds to an and-node that has a departing edge to the or-node corresponding to $A$. 
• Each and-node that corresponds to a production of the form $A \rightarrow \epsilon$ has an arriving edge from the true-leaf.

• Each and-node that corresponds to a production of the form $A \rightarrow a\alpha\beta$ has an arriving edge from the false-leaf.

• Each and-node that corresponds to a production of the form $A \rightarrow \alpha B\beta$ has an arriving edge from the or-node corresponding to $B$.

Each node is assigned an initial value. The true-leaf is given initial value $\text{true}$ and all other nodes are given initial value $\text{false}$. Then we propagate the values through the graph in the direction of the edges. At an and-node, boolean and is performed on the old value of the node and the values that arrive on the edges from the neighboring nodes. At the or-nodes, boolean or is performed instead. The propagation continues until a fixed point is reached. This is guaranteed to happen, since a node never moves from value $\text{true}$ to value $\text{false}$. A non-terminal belongs to $\text{NULLABLE}$ if the or-node that corresponds to the non-terminal has value $\text{true}$ in the fixed point. In this case iterative propagation is pretty efficient, since the domain is binary. Also, the efficient method described below is unable to handle the and-nodes.

5.3.3 Computing the First Function

First is also computed from a graph, but the graph only contains a single type of node. The graph is given by:

• Each terminal symbol corresponds to a node with no arriving edges. It has the singleton set containing the terminal as its value.

• Each non-terminal symbol corresponds to a node with the empty set as initial value.

• Each production of the form $A \rightarrow \alpha X\beta$, where $\text{NULLABLE}$ is true for $\alpha$, corresponds to an edge from the node corresponding to $X$ to the node corresponding to $A$.

Again we want to propagate the sets through the graph until a fixed point is found. At each node we take the unions of the old value for the node and the values that arrive on the edges. Again, a fixed point is guaranteed, because the sets in the nodes grow monotonically (and are all bounded by a finite set). The propagation can be done in a non-iterative fashion, with the aid of some graph algorithms from one of the standard Agents libraries. First the graph is reduced to its strongly-connected components (SCCs) with Tarjans algorithm (see [Sed88]). Each SCC takes on the value of the union of the values of the nodes in it. This is because each member of a set in one of the original nodes can be propagated to every other node in the SCC
(by definition of SCC). Then the reduced graph is sorted topologically (see [Sed88]). Since a reduced graph always is a DAG, the topological sort always succeeds. The values can then be propagated through the DAG in a single pass. Both the algorithm that reduces the graph to SCCs and the one that makes the topological sort run in nearly linear time, so this method is guaranteed to be efficient.

5.3.4 Building LALR(1) Sets of Items

The sets of items are represented with just the kernel items, and the LR(1) closure is taken on each item set with a dummy look-ahead symbol as in Algorithm 4.12 of [ASU86]. Then the entire image of the goto function from the state is computed at once. First we move the dot to the right of all items in the closure, and then we partition it according to the symbol that was skipped (and thus is located to the left of the dot). Thus we have removed the separate step of generating the sets of items by merging it into the look-ahead generation phase.

The dummy look-ahead is then used to establish links in a graph for look-ahead propagation as in the original algorithm. The look-ahead propagation is then performed with the same propagation algorithm we use to compute the FIRST function.

5.3.5 Resolving and Reporting Conflicts

The generation process may produce conflicting entries in the parse table. Some conflicts should be resolved and some should be reported to the user, see Section 4.3 [Dealing with Conflicts], page 85. The conflicts related to non-determinism and conditional rules are treated elsewhere, see Section 5.5 [Non-Determinism], page 114.

Some shift/reduce conflicts should be resolved with operator precedence and some are explicitly resolved by the user by a shift annotation. Resolution with operator precedence is used whenever the look-ahead symbol and the reduction both have precedences. See Section 3.10.3.2 [Operator Precedence], page 66, for the rules used to resolve the conflict.

To handle explicit resolution with shift annotations we inspect the set of items the shift transfers to. If any of the items in that set corresponds to a rule with a shift annotation of the terminal to the immediate left of the dot, the conflict should be resolved silently in favour of the shift action.

A conflict that cannot be resolved must be reported to the user. SAGA does this in form of schematic input that would result in the conflict. We search the set-of-items graph to find the shortest path from the starting set of items to the set containing the conflict. The grammar symbols labeling the edges are collected in order and form the schematic input.
5.4 Table Packing Algorithm

First we study the algorithm used to pack the check-array (and the next-array) tightly. Then we discuss how the default states are chosen in the lexical analyzer. The defaults in the parser tables are straight-forward and does not warrant special discussion.

5.4.1 Next-Check Allocation Algorithm

The packed table format has been described earlier, see Section 5.1.2 [Packed Tables in the Lexical Analyzer], page 101. The problem with this method is to find a good placement of the rows in the check-array (and next-array). The basic idea is simple enough. We take a row at a time, and place it at the lowest position where no filled cell in the row conflicts with any other filled cell. Unfortunately, this is rather demanding. The algorithm is quadratic, with a large constant (proportional to the size of the character set).

The first obvious improvement is to treat consecutive entries as blocks and to ignore the empty cells in the row. Therefore, we need only check if the blocks for the row fit. We use a tree of free blocks to allow fast checking for enough free space. Checking the largest block first increases the chance of early failure.

The next step is to avoid checking each position for possible allocation. When a certain block fails to fit, we may search for the lowest available position where it does fit. This allows us to take large jumps each time.

A further improvement is due to the fact that if the free block extends further than the right end of a block, then it is feasible to slide the block some number of steps. If we keep track of this, then we do not always need to find a new free block to allocate the block in.

For the discussion of the algorithm, we assume the existence of a function that returns locations of available free blocks (without allocating any). The function needed takes two arguments, a block size and an offset. It also returns a block size
and an offset, indicating a free block of the returned size at the returned offset. The arguments are minimum values of the size and offset, respectively. This means that we ask for a large enough free block at a sufficiently large offset. The offset returned is taken as small as possible, and the size as large as possible, i.e. we use first fit to find the position and we do not cut any free block short at the right end. The free blocks are maintained in a balanced tree, so the function runs in time logarithmic to the number of free blocks.

We use a variable, the current position index to keep track of the lowest possible position we may try to allocate at. Initially this is zero. As the algorithm proceeds, positions are found to be impossible for allocation and the index is increased. We aim to find the lowest possible value of this index that allows allocation.

One way to view the allocation process is the following: For each block separately, consider the set of positions it may be allocated at. From these sets, extract positions that respect the distances between the blocks.

Let us now formalize and implement this idea. Let us consider the blocks as separate entities that each has a position and a size. The blocks are not independent, however, so each block also has a (leftmost) column; the first column in the row that belongs to the block.

Define the point of a block to be its position minus its column. This corresponds to the position the row would have if the block is fixed at its position. For each block, we define its reallocation point to be the highest value that its point may have before it needs to be reallocated in another free block. This is the point plus the slide margin the block currently has. Stated another way, if the current position index is larger than the reallocation point for a certain block, then the block cannot slide far enough in its present free block.

The algorithm exploits a priority queue of the blocks belonging to the row currently being allocated. The priority is given by the reallocation points and the sizes of the blocks. The lower the value of the reallocation point, the higher the priority. If the reallocation points are equal, then larger blocks are given priority. Initially, all blocks in a row are given priority corresponding to negative reallocation points, i.e. all blocks are scheduled for reallocation.

The block with the highest priority is chosen. If the position is larger than or equal to the current position index, we have found a position at which we can allocate the row. If the position is less than the current position index, it indicates that the block cannot slide far enough in its current free block, so we reallocate it at the next higher position where it fits. Then the current position index is increased accordingly. The new value of the index is the new point of the block. The block is then reinserted into the priority queue, and the process is repeated.
The figure below demonstrates the allocation process by example. The first step, to find a free block for block B, has been omitted.

The above algorithm reallocates the block that have the least potential to slide. This usually means that large blocks get reallocated often, thereby moving the current position index quickly upwards. It also avoids all attempts to reallocate a block that can slide. In all, it is reasonably efficient. It is still a quadratic algorithm, at least in worst case, but it works well in practice.

5.4.2 Choosing Default States

The choice of default states in the lexical analyzer may be made in different ways. An optimal algorithm would be unpractically slow. The algorithm chosen in SAGA is simple and fast, but does a reasonable job nevertheless.

We partition the states so that each part has a common candidate to become prototype state. We simply check the most common to-state for each state and use this to form the partition. Then we choose a state from this group by giving priority to the one that has the most transitions to this common to-state. If more than one
state have equally many transitions to the common to-state, we give priority to the one that has the fewest other transitions. This is sufficient to give a good packing for many programming languages, since it is ideal for finding the catch-all cases in reserved word languages. It also does a good job in many other common cases. For instance, automata for the numerical formats used in C becomes tightly packed.

5.5 Non-Determinism

The non-determinism in SAGA is quite straight-forward to implement. Most of the work is done by pre-processing of grammars and resolving of conflicts.

Proper non-deterministic rules (that use the ‘??’ guard operator in SAGA) are implemented by non-deterministic resolution of conflicts. Instead of choosing one parsing action, a non-deterministic agent is generated and a reference to it is placed in the action table. The agent generates the conflicting parsing actions so that they can be tried non-deterministically.

The only complication is that we need to determine when conflicts should be reported to the user and when they should be resolved non-deterministically. Reductions causes no problems, since they are identified as (possibly) non-deterministic by the ‘??’ guard operator. To handle the shift actions, each terminal in a non-deterministic rule behaves as if it had a \texttt{shift} annotation (from the point of conflict reporting).

To implement the conditional rules (that use the ‘??’ guard operator in SAGA) we add new marker non-terminals to the grammar. A marker \texttt{non-terminal} is a non-terminal that has a single production that produces the empty string of terminals. A new marker non-terminal is prepended to each clause in the conditional rule.

This (almost) always leads to reduce/reduce conflicts between some or all of the marker non-terminals belonging to the same conditional rule. Since we require the start of a conditional to be LL(1) recognizable (see Section 4.4.3 [Adaptation to LR Parsing], page 89), only markers from the same rules should result in conflicts, provided that the grammar is correct. No secondary conflicts result from this scheme, since all marker non-terminals are different. The different clauses corresponds to different branches in the DFA that is given by the GOTO function.

The conflicts are resolved silently. Agents code is generated to try the different reductions one by one. This is done by calling the parser recursively from within a conditional guard in Agents. One conditional clause is generated for each marker reduction in the conflict. When the reduction belonging to the conditional clause is executed, the recursive call of the parser and the guard expression succeed.

Because of the way in which conditional guards work in Agents, the non-deterministic choices in the recursive call are committed when the guard succeeds.
This gives us the desired encapsulation of the non-determinism. One limitation is due to the behaviour of guards in Agents. Messages cannot be sent to ports inside of guards or non-deterministic computations. In particular, this means that input cannot be read from files while non-determinism is active. This is a serious drawback, and the only work-around is buffering. Apart from the normal buffering used in the generated parser, we have an extra buffering scheme for the non-determinism.

When a conditional rule is entered we buffer a large chunk of the input. The size of this buffer is subject to user control, and may be set to the size of the rest of the input. Of course, feeding SAGA list input works well, too.
References


[Par93] Terence J. Parr. Obtaining Practical Variants of LL(k) and LR(k) for k>1 by Splitting the Atomic k-Tuple, Ph.D. diss., Purdue University 1993.


Index of Concepts

(Index is nonexistent)
# Table of Contents

1 Introduction .................................. 1

2 Motivation .................................. 3
   2.1 Introduction .................................. 3
   2.2 Problems with Imperative LR Parsing ........ 3
       2.2.1 Pure LR versus LR with Actions ........ 3
       2.2.2 The Need for Unbounded Look-Ahead .... 4
   2.3 Previous Work .................................. 5
       2.3.1 LL(k) Parsers .............................. 5
       2.3.2 Backtracking Parsers ...................... 6
   2.4 Benefits of Concurrent Constraints ........ 6
       2.4.1 Concurrent Constraint Programming ...... 6
       2.4.2 Constrained Attributes .................... 7
       2.4.3 Deep Guards and Backtracking .......... 10
   2.5 Conclusion .................................... 12

3 User Manual ................................. 13
   3.1 Introduction .................................. 13
   3.2 SAGA Input and Output ..................... 14
   3.3 Basic Concepts of SAGA ..................... 14
       3.3.1 Lexical Analysis ......................... 15
           3.3.1.1 Example .......................... 15
           3.3.1.2 Regular Expressions ............... 16
       3.3.2 Filters ................................... 17
       3.3.3 Parsing ................................. 18
           3.3.3.1 Parse Trees and Grammar Rules .... 18
       3.3.3.2 Grammar Classes ..................... 19
       3.3.3.3 Syntax Trees and Semantic Values .... 21
   3.4 Tutorial ..................................... 22
       3.4.1 Parsing Arithmetical Expressions ..... 22
       3.4.2 Building a Syntax Tree ................ 25
       3.4.3 Filters and Definitions ................. 27
       3.4.4 Lexical Modes ............................ 28
       3.4.5 From Expressions to Statements ....... 30
           3.4.5.1 Conflicts .......................... 31
3.4.5.2 Left Recursion is the Right
Recursion .................................. 32
3.4.6 Using Operator Precedence .......... 33
3.4.7 Using EBNF .......................... 35
3.4.8 More EBNF .......................... 37
3.4.9 Global Accumulators ................. 40
3.4.10 Handling the Problem with "typedef" 42
3.5 Structure of the Grammar File ......... 43
3.5.1 Directives ........................... 44
3.6 Lexical Analysis ........................ 45
3.6.1 Tokenization Basics .................. 45
3.6.1.1 Lexical Modes .................... 46
3.6.1.2 How the Input is Matched ........ 46
3.6.2 Regular Expressions ................ 47
3.6.2.1 Restrictions on Syntax .......... 50
3.6.3 Definitions ........................ 50
3.6.4 Inheritance ........................ 50
3.7 Grammar Rules ........................ 51
3.7.1 Basic Rules ........................ 51
3.7.1.1 Terminal Symbols ................ 51
3.7.1.2 Non-Terminal Symbols .......... 52
3.7.1.3 Special Grammar Symbols ....... 53
3.7.1.4 Head and Body ................... 53
3.7.2 Accumulator Syntax ................ 54
3.7.3 EBNF ............................... 56
3.8 Filters ............................... 59
3.9 The Interface to Agents ............... 60
3.9.1 Calling the Parser .................. 60
3.9.1.1 The Parser Return Status ....... 61
3.9.2 Parsing Only Part of the Input ...... 62
3.9.3 Embedded Agents Code .............. 63
3.9.3.1 Tokens as Data Structures ....... 63
3.9.3.2 Standard Global Accumulators .... 63
3.9.3.3 Standard Agents ................ 63
3.10 The SAGA Parsing Algorithm .......... 64
3.10.1 Shift/Reduce Parsing ............... 64
3.10.2 LR Parsing ......................... 64
3.10.3 Resolving Shift/Reduce Conflicts .... 65
3.10.3.1 shift Annotations ............... 65
3.10.3.2 Operator Precedence .......... 66
3.10.4 Mysterious Reduce/Reduce Conflicts .. 66
3.11 Error Handling ....................... 68
3.11.1 Default Error Recovery ............ 68
3.11.1.1 Local Correction .................. 69
3.11.1.2 Phrase Level Recovery .......... 69
3.11.2 User Defined Error Recovery ....... 71
3.12 Advanced Parsing Strategies .......... 73
   3.12.1 Conditional Rules ................. 73
      3.12.1.1 Restrictions .................. 74
   3.12.2 Non-deterministic Rules .......... 75
   3.12.3 Encapsulation of Non-determinism 76
   3.12.4 Advanced Strategies and Ports .... 76
3.13 Invoking SAGA .......................... 76
3.14 SAGA Standard Library ................. 77
   3.14.1 Error Reporting .................... 77
   3.14.2 Error Recovery ..................... 80
      3.14.2.1 Complete Strategies .......... 80
      3.14.2.2 Parts of Strategies .......... 80

4 Design Rationale ....................... 81
   4.1 Initial Design Choices ............... 81
   4.2 Basic Structure ....................... 82
   4.3 Dealing with Conflicts ............... 83
   4.4 Advanced Parsing ...................... 84
      4.4.1 Known Solutions .................. 84
      4.4.2 Inspiration from AKL ............. 85
      4.4.3 Adaptation to LR Parsing .......... 87
   4.5 Error Handling ....................... 88
      4.5.1 Error Reporting, Repair and Recovery .... 89
         4.5.1.1 Error Reporting ............... 89
         4.5.1.2 Error Repairs ................. 90
         4.5.1.3 Error Recovery ............... 90
      4.5.2 Pros and Cons of Automatic Error Handling 90
         4.5.2.1 Automatization of Error Reporting 91
         4.5.2.2 Automatization of Local Repairs 92
   4.5.3 Phrase Level Recovery .............. 92
   4.5.4 SAGA Error Handling ............... 93
      4.5.4.1 The Default Error Handling .... 93
      4.5.4.2 Custom Local Repairs .......... 94
5 Implementation ........................................ 95

5.1 Runtime System ........................................ 95
  5.1.1 Lexical Analyzer ................................. 96
    5.1.1.1 Needed Features ......................... 96
    5.1.1.2 Structure of the Lexical Analyzer .... 96
    5.1.1.3 Lexical Backtracking ................. 98
  5.1.2 Packed Tables in the Lexical Analyzer .... 99
    5.1.2.1 Sparseness ............................... 99
    5.1.2.2 Other Redundancy .................... 99
    5.1.2.3 Improvements ......................... 102
  5.1.3 Packed Tables in the Parser ............... 102
  5.1.4 Filters ........................................ 103

5.2 Building of the Lexical Analyzer .............. 103
  5.2.1 Regular Expressions to DFA .................. 103
  5.2.2 Turning a DFA into a Lexical Analyzer .... 105

5.3 Building of the Parser .............................. 105
  5.3.1 Overview ..................................... 106
  5.3.2 Computing the NULLABLE Predicate ....... 106
  5.3.3 Computing the FIRST Function ............. 107
  5.3.4 Building LALR(1) Sets of Items .......... 108
  5.3.5 Resolving and Reporting Conflicts .......... 108

5.4 Table Packing Algorithm ......................... 109
  5.4.1 Next-Check Allocation Algorithm .......... 109
  5.4.2 Choosing Default States ................... 111

5.5 Non-Determinism .................................. 112

References .............................................. 115

Index of Concepts ................................. 117