Faculty of Mathematics and Physics

Charles University in Prague

Master Thesis

Rudolf Thomas

A Parser of the C++ Programming Language

Department of Software Engineering

Supervisor: RNDr. David Bednárek

Study Program: Computer Science

Acknowledgments

I would like to thank my supervisor, RNDr. David Bednárek, as well as everyone else, who made finishing this thesis possible. They know who they are.

Prehlasujem, že som svoju diplomovú prácu vypracoval samostatne a výhradne s použitím citovaných zdrojov. Súhlasím s požičiavaním práce.

V Prahe, 12. augusta 2005.

Rudolf Thomas

Contents

Title: A Parser of the C++ Programming Language

Author: Rudolf Thomas

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek

Supervisor's e-mail address: `david.bednarek@mff.cuni.cz`

Abstract: The goal of the work is to design and implement a parser for the C++ programming language, as defined by international standard ISO/IEC 14882:2003. An analysis of the difficulties of parsing C++ and a comparison with other existing parsers are presented. The work focuses on solving ambiguities of the language, which do not allow traditional parsing methods to be used for its analysis. Nevertheless, GNU Bison is used to generate LALR(1) automatons that are used for both analysis of unambiguous input, and ambiguity resolution. Providing a working implementation is the main objective.

Keywords: syntactic analysis, C++, disambiguation

Názov práce: Syntaktický analyzátor programovacieho jazyka C++

Autor: Rudolf Thomas

Katedra: Katedra softwarového inženýrství

Vedúci diplomovej práce: RNDr. David Bednárek

e-mail vedúceho: `david.bednarek@mff.cuni.cz`

Abstrakt: Cieľom práce je návrh a implementácia syntaktického analyzátora pre programovací jazyk C++ definovaný medzinárodným štandardom ISO/IEC 14882:2003. Súčasťou práce je analýza problému, ako aj porovnanie s inými existujúcimi analyzátormi. Ťažiskom je nutnosť riešiť nejednoznačnosti v syntaxe jazyka, ktoré znemožnujú použitie tradičných metód syntaktickej analýzy. Napriek tomu implementácia využíva GNU Bison na generovanie LALR(1) automatov, s pomocou ktorých sa nielen analyzuje jednoznačný vstup, ale aj riešia prípadné nejednoznačnosti. Výsledkom práce je fungujúci analyzátor.

Kľúčové slová: syntaktická analýza, C++, riešenie nejednoznačností

Chapter 1

Introduction

The goal of this work was to design and implement a parser of the C++ programming language, as defined by the 2003 international standard of the language [5]. We put emphasis on meeting the requirements that the Standard imposes on syntactic analysis. The resulting parser was used in the Lestes project, an ongoing effort to create a C++ compiler.

1.1 C++ Is Ambiguous

Unlike other programming languages, parsing C++ is not straightforward. Usually, it is sufficient to write a grammar for the language, modify the grammar to meet some requirements and let a generator produce the actual parser. However, traditional parser generators are not powerful enough to handle C++, as the language is significantly ambiguous.

By *ambiguity*, we mean that the same sequence of input tokens can be interpreted as two or more distinct language constructs that are allowed to appear in the same context. Mostly, these constructs have completely different structure.

Many of the ambiguities of C++ result from its extensively overloaded use of symbols. For example, at a specific point in a declaration, left parenthesis (`(`) can prologue a parameter--declaration-clause, or an initializer. In cases when syntaxes of both these constructs intersect, an ambiguity occurs. In other cases, the whole sequence (which ends on a closing parenthesis) is not ambiguous as defined above; however, beginning of the sequence may look like beginning of any of the two constructs, and the token, that – in effect – decides which of the possibilities is no longer vital, can be arbitrarily far in the input stream.

In some cases, *type information* (i.e. whether an identifier denotes a type or a non-type) is enough to resolve ambiguous input. In other cases, when type information does not suffice, the Standard [5] states which of the possible interpretations is to be considered. Note that nested ambiguities may appear inside an already ambiguous part of input. These should be resolved exactly the same way as the top-level ones: either type information, or the Standard will help.

Nevertheless, serious problems arise from unambiguous code too. In the example above, the parameter declaration clause semantically belongs to the declarator that precedes it. On the other hand, the initializer does not. Thus, just before the parenthesis, both LR and LL parsers have to decide whether to continue parsing the declarator, or finish the declarator and start with the initializer (this is referred to as a shift-reduce conflict in LR parsers). This, combined with the fact that the decisive token may be arbitrarily far, outcasts using ordinary LR(k) and

LL(k) automatons (for any finite k) to parse C++ (provided that they resolve all the language ambiguities).

1.2 Goals

Our goal was to implement a parser that would correctly analyze the whole C++ language. This includes having to resolve all the ambiguities as mandated by the Standard. In addition, we wanted to perform the resolution exclusively in the syntactic analyzer so that semantic analysis is presented with clear, unambiguous data. We also intended to use the GNU Bison generator [2] to produce as much of the resulting code as possible.

The parser has been designed to be a part of the Lestes project [7]. The primary function of the parser was providing syntax trees to the semantic analysis, for the purpose of consequent code generation. Other uses were not planned.

Speed, or powerful error recovery were not targeted either. We chose to prioritize correctness over performance. Recovery from syntax errors was not considered.

It is important to mention that our goals greatly contrast with current practice of some other C++ parsers. Some simply refuse to analyze corner cases in the input; there is rather good reason to abandon absolute adherence to the Standard: the corner case code is usually difficult to comprehend thus hardly ever used, and usually easy to simplify to be both more human-readable and analyzable by the program in question. A different technique merges the conflicting – ambiguous – parts of the C++ grammar and leaves the resolution of the ambiguities up to the semantic analysis[1]. There is also the option to write the analyzer by hand; some projects successfully utilize this approach.

1.3 Key Aspects of Our Solution

Type information about identifiers is provided by *hinter*. It consists of a filter between lexical analyzer and the parser. The hinter tags identifiers with type information that is looked up in symbol tables.

Our parser builds on the premise that all ambiguity resolution rules are formulated as *precedences* – e.g. if a statement looks like both expression-statement and declaration-statement, it is a declaration statement. The actual process of resolving ambiguities, *disambiguation*, lies in pausing the parser at states that precede possibly ambiguous input. At this point, all language constructs that are allowed to follow are tentatively parsed by specialized parsers, in the order deduced from the precedence rules. The first construct that matches the input is chosen as the final interpretation.

When we reach a state where nested ambiguity is possible to appear, the same principle is employed. The parser trying to find the interpretation of construct is paused and the possibly ambiguous subpart of the input is tried to be parsed. In effect, the parser uses *back-tracking* to search for the meaning of the whole language construct. It is crucial to understand that the Standard demands that the decisions should be based purely on syntactic and type information. It means that the parser does not have to perform any semantic checks during disambiguation.

[1]This resolution may be implemented as another pass between syntactic and the actual semantic analysis.

As a result, given the precedence rules, it may choose semantically erroneous interpretation instead of one that would be correct (if it were chosen)[2].

The main parser, as well as the specialized disambiguation parsers, are generated from one grammar file by Bison. The grammar is derived from the one in the C++ standard. Most of the changes were done to allow us to pause the generated automaton at specific times and run another instance for the purpose of disambiguation.

The disambiguation process is also used to handle states of the automaton that need to look more than one token ahead to decide whether to shift or reduce. The actual number of tokens that need to be looked at is not limited and depends solely on the length and complexity of input. The same back-tracking approach is exercised: we launch specialized parsers to try out the possibilities while the upper-level parser is paused, waiting for the decision. Note that we do not solve all the shift-reduce conflicts of the C++ grammar this way – minor modifications of the grammar were sufficient for most of the easy ones.

The analyzer is fully reentrant, because some parts of the source program (e.g. method bodies defined in class definition) must not be analyzed immediately. While this is not related to ambiguity resolution in any way, we are prepared to asynchronously process any part of input that has been previously stored.

1.4 Structure of This Document

Chapter 2 comprises of an overview of other approaches to parsing C++. Basics of GLR, language superset based, and recursive descend parsers are explained.

Requirements of our solution, and details of its implementation can be found in Chapter 3. The last chapter summarizes our achievement.

All disambiguation rules are enumerated in Appendix A. For every conflicting part of the language we provide an example of a conflict, the order of resolution we use, and references to corresponding sections of the Standard.

Throughout the text, we often refer to specific parts of the C++ Standard [5]. These references are written using the section number symbol. For example, §1.9 refers to section 1.9 of the Standard (entitled “Program execution”). Numbered paragraphs of a section are referred to using colon prefix (e.g. §1.9:3).

Grammar examples are written in the Bison syntax, see its documentation [2].

The latest version of this document in electronic form can be found on the Internet at
`http://rudo.matfyz.cz/thesis/`.

[2]Assuming both follow the syntax rules.

Chapter 2

Related Work

In this chapter, we will give a brief overview of other parsers of the C++ language that aim to be correct with regard to the requirements of the Standard. This rules out simple Bison--generated parsers. While they are able to analyze substantial part of the language, resolutions of the ambiguities (which result in conflicts) are hard-coded in the automaton. Usually, chosen resolution corresponds with the alternative that is in wider use. Yet, this method cannot be compared to parsers which strictly address the syntactic challenges of C++.

For a more broad coverage of the various strategies of parsing C++, see [1]. Algorithms referred to throughout this chapter are extensively described in [4].

2.1 GLR

The Algorithm

Parsers based on Generalized LR (GLR) algorithm are able to work with any context-free grammar [12]. Whenever the parser reaches ambiguous input, it continues in nondeterministic fashion. In effect, a GLR parser is able to consider all of the interpretations at the same time.

This is achieved by modifying the LR algorithm to *clone* the stack of the automaton whenever a conflict would otherwise occur. Each resulting copy is then operated on its own, in parallel with other copies: they all reduce and/or shift at the same step of the algorithm. Some of the paths prove to be wrong, making the particular automaton fail. This failure is ignored, as long as there is at least one other clone that can continue. This technique handles all the cases where longer lookahead is sufficient to resolve a conflict.

In cases when a pair of automatons finds different parse trees for the same portion of input (i.e. an ambiguity was found), a resolution (that is said to *merge* the different trees) must take place. There are three outcomes of the merge: either one of the parse trees is picked, discarding the other; or the ambiguity is retained by storing the trees; or the ambiguity was not expected to happen and an error is reported. Most of the time, a helper function must be provided by the programmer, but sometimes (as a shortcut) the selection rules can be described as priorities.

For efficiency, the states (stacks) of the automatons are not stored as separate copies, but rather a graph structure – that represents all states ordinary LR parser could have reached – is used. It is possible to implement the structures so that the GLR algorithm runs in cubic worst-case time (in the size of the input) [6, 2].

Generators

There are several GLR parser generators available. We will talk about two of them concisely: Elkhound and GNU Bison.

Elkhound [8] generates parsers that use a hybrid GLR/LR algorithm. Ordinary LR is used for the deterministic parts of the input, whereas GLR is only used when the input is ambiguous and/or needs deeper lookahead to be resolved. This hybrid parsing method yields good performance, compared to other GLR implementations – the LR part is as fast as with ordinary LALR(1) parsers, while overhead of the mechanism that switches between LR and GLR modes of operation is almost negligible. An almost complete C++ parser is provided as a case study of the generator.

Elsa C++ parser (based on Elkhound) employs a type-checking pass to resolve ambiguities. It does not categorize identifiers by type information. Instead, it stores parse trees of all the possible interpretations and type-checks them later, during semantic analysis, when all the necessary data is available. This way, it is possible to completely separate the semantic and syntactic analysis, unlike in our parser.

Recent versions of GNU Bison [2] have the ability to generate GLR parsers. Compared to LALR(1) automatons generated by Bison, their GLR counterparts defer execution of the rule actions until there is only one viable parser, or execute them just before the parse trees are handed over to user-specified merge function (see above). This delay renders any technique that tags identifiers with type information (e.g. our hinter) inapplicable, because the programmer does not have control over the time when the actions are actually performed. With regard to behavior of the generated automaton, the documentation admits that the used data structures impact the worst-case time and space efficiency, possibly resulting in exponential requirements[1].

From this overview, it would seem that GLR is ideal to base a C++ parser on. Actually, one of our first attempts at designing syntactic analyzer for the Lestes project was a GLR parser. However, after evaluating the possibilities, we found out that Bison's support for GLR was not satisfactory and Elkhound was poorly documented.

2.2 Superset Approach

Another strategy of doing syntactic analysis of C++ is to invent LALR(1) grammar that covers the C++ language, parse input with generated automaton, and analyze the synthesized syntax tree to resolve the ambiguities.

Complete parser (called FOG) that utilizes this approach is described in [13], where the author introduces an extended form of regular expressions that is able to express recursion. This novel notation simplifies analyzing a context-free grammar and helps to identify its problematic parts.

The C++ grammar is rewritten using the extended regular expressions syntax and is generalized into a new one – one that accepts language that is a superset of C++. This grammar is then converted to format that is suitable for Bison (Backus-Naur Form).

The resulting grammar, however, has its glitches. It does not try to resolve the ambiguities, but to cover the ambiguous alternatives by a more general set of rules. The same principle is used

[1]As of GNU Bison versions up to and including 2.0.

to deal with constructs that need a longer lookahead to be parsed correctly: the conflicting rules are merged.

One of the goals of the FOG project was to develop a parser that strictly does *not* use any kind of semantic information. This implies that the ambiguities (and similarities) have to be preserved in the syntax tree. Compared to the Elsa GLR parser (above), only one syntax tree is produced by the Bison-generated automaton, which makes it more difficult to inspect.

Not having information about identifiers being template names or not exposes the problem of the less-than symbol (<) when it follows an identifier. It can be interpreted either as relational operator ("less than"), or as opening angle bracket of template argument list. The implementation performs a binary tree search to find a syntactically consistent interpretation (i.e. every opening bracket must have its closing counterpart), which leads to exponential complexity (claimed to be insignificant in practice).

Searching the binary tree (to find an interpretation that follows the syntax) is quite similar to the disambiguation process of our parser. Indeed, FOG does have support for putting marks in the input and rewinding it, just as we do (see page 10). However, there is no trial pass to find out which of the alternatives is valid. Instead, both tries are handled in the same automaton, employing a trick that exploits Bison's error recovery features[2]. Nevertheless, the absence of the trial pass is adequate, because the parser – being completely separated from semantic analysis – does not run any actions that would have to be undone when the first alternative proves to be wrong.

2.3 Recursive-Descend Parsers

Recursive-descend (LL) parsers work differently, compared to LR/GLR/LALR. As the name suggests, they analyze the input from top to bottom, whereas the LR automatons proceed from bottom up. When making decisions, algorithms of both kinds depend on the lookahead tokens.

The biggest advantage of top-down parsers is that the code of the automaton is usually written as *functions*, one function for every nonterminal of the grammar. Typically, it is obvious what steps and decisions is the analyzer making. Therefore it is much easier to debug (when confronted to the huge network of states that define LR automatons). What is more, LL parsers can be written manually, while hand-coding a larger LR parser is practically impossible.

GCC C++ Parser

As of version 3.4, GNU Compiler Collection (GCC, [3]) contains a hand-written parser for C++. It is implemented in C and the source is about 500KB in size. Being a top-down parser, its code consists of functions that mainly call other parsing function. The result barely resembles the grammar found in the Standard. Being specifically tailored for the current language, it might be extremely difficult to adapt the code to new versions of C++, even if new standards introduce miner changes only.

To resolve ambiguities and conflicting constructs of the language, the parser uses lookahead both explicitly in the parsing functions (sometimes more than one token onwards), and implicitly when running in so-called tentative mode. When two (or more) alternatives are indistinguishable from each other using fixed-size lookahead, the parser switches to this mode, in

[2]In our parser, the trial passes are carried out by out-of-order parsers.

Listing 2.1: Example input that makes GCC Parser commit early.

```
int j, k, l;
class C { /* ... */ };
void g() {
  C (j) = 5, (k) = 8, l++; // (1) expression-statement (ill-formed)
  C (j) = 5, (k) = 8, l;   // (2) declaration-statement
}
```

which no error messages are output and all shifted tokens are saved. The alternative with higher precedence is tentatively parsed first. When subsequent analysis indicates that the correct alternative was chosen, the parser *commits* – switches off from the tentative mode. When still in the mode and a syntax error is found, saved tokens and parser state are rolled back[3] and the other alternatives are tried.

While investigating the implementation, we observed that the parser can commit too early and print inaccurate error messages, when it is presented with a specially fabricated (but ill-formed) input. It does not accept syntactically correct code from the example in Listing 2.1. Line (1) contains syntactically correct expression-statement. It codes two applications of the comma operator with assignment to explicitly converted `j` (§5.2.3); assignment; and post-incrementation being their arguments, respectively. Let's assume that class `C` supports construction from `int` and operator `++`. The parser incorrectly commits to a declaration-statement and reports syntax error at the `++` token. The real error is semantic, however. Although the statement is syntactically correct expression-statement, the result of explicit type conversion is an rvalue (§5.2.3:1, §5.4:1) and cannot be assigned to. To summarize, the ill-formed code is not accepted, but a bogus error is reported. Line (2) is well-formed and parsed correctly[4]. It declares `j`, `k`, and `l`, all of type `C`.

ANTLR

ANTLR (ANother Tool for Language Recognition) [9] is a parser generator that is able to work with predicated LL(k) grammars. The selection of production rules (function calls) of the created automaton can depend on any combination of the following: k symbols of lookahead (computed automatically), syntactic predicate (user-specified prefix that the input must match), semantic predicate (provided by user; e.g. the following rule is only vital when the previous identifier denotes a function). What is more, the generator accepts grammars written in EBNF (Extended Backus-Naur Form) which is similar to regular expressions and makes the grammar easier to assemble and understand.

There is a C++ grammar (with a semantic analysis helper) for ANTLR that claims to fully support the language, yet – at the time of this writing – we do not know of any production compiler that is using it.

[3]Restoring parser state is not complicated – execution only needs to return from a stack of invoked functions.
[4]Both tests were conducted using GCC version 3.4.3.

Chapter 3

The Parser

3.1 Requirements

Before we unveil the details of our solution, we outline what aspects needed to be taken into consideration and why. Many implementation decisions are explained as well in this section.

3.1.1 Type Information

We wanted to resolve all the syntactic ambiguities and similarities strictly inside the parser, and not relay the issue to semantic analysis (or another pass before it). To achieve this, it is necessary to use type information.

Using type information is in conformance with the Standard, as it distinguishes type and non--type names at the grammar level. To be able to capitalize on this, the type information needs to be tagged to the tokens before they enter the parser. We chose the traditional solution here: create a filter that *categorizes* identifiers and place it between lexical and syntactic analysis (sometimes this technique is called "lexer hack").

We call this filter *hinter*. It is stateful, as it needs to take the context of the identifier into account. This includes qualified identifiers, elaborated specifiers (names prefixed by `class` keyword). The hinter does not process those constructs itself, but is controlled from the parsing automaton, which has all the necessary notion of the contexts.

Apart from distinguishing type names from non-type ones, the categories also tell whether a name denoted a template or not. This is invaluable when an identifier is immediately followed by a less-than (<) symbol. If the name is a template name, the symbol is to be interpreted as the opening brace of template argument list, not as the relational operator (§14.2:3).

Logically, hinter operates on tokens after they leave the lexical analyzer and before they enter the parser. Technically, because the analyzer itself is rather complex, it is called from within the manager (see below, page 26), just before a token is passed to the Bison-generated automaton.

For a detailed description of the hinter, see section 3.4 on page 17.

3.1.2 Disambiguation

As the C++ language is ambiguous, it was clear that the single lookahead token – that is relied upon in LALR automatons – will not be sufficient for ambiguity resolution (this is the term used throughout the Standard).

One of our early proposals was using regular expressions for recognizing the correct alternative. We assumed that a declaration could be told apart from an expression by a prefix that can be notated using a regular expression. Needless to say, recursion would have to be taken into account. Combined with support for stateful scanning (referred to as "start conditions") in Flex [10], we thought we could implement an automaton that would give us information about validity of each of the alternatives.

However, regular expressions (even with primitive recursion) are not very well suited for recursive language recognition. It turned out that the automaton must have profound knowledge of the grammar to be capable of recognizing constructs from each other. What is more, in some cases, it is necessary to analyze the whole statement (which can be arbitrarily long), not just a regular prefix, to tell which alternative is vital. For an example of such statement, see page 7.

Therefore, specialized parsers (generated by Bison) are used when the main parser needs to look more than one token ahead[1]. At states that precede conflicts, these automatons are launched from the main one (while it is paused) to examine the upcoming input. Depending on which of the special automatons succeeds, it is known which of the alternatives is syntactically valid. To inform upper level parser of the resolution, a decisive token is inserted into the input stream. Throughout the source of our analyzer, these are referred to as *pad tokens*.

We call this scheme – of launching helper parsers and inserting tokens into input – *disambiguation*. The idea is that the top-level automaton runs "smoothly", i.e. it has all the ambiguities and would-be-conflicts already pre-solved by the time it reaches them. In fact, they are solved just before they are reached, but the process of resolution is fully transparent. Unfortunately, complete transparency cannot be ensured at all times, as in some cases the disambiguation takes place after the lookahead token is read and processed. How we handle these cases, as well as a more detailed illustration of the whole process, can be found in section 3.5 on page 21.

Figure 3.1 on the following page shows the pad tokens being inserted into the input token stream. An ideal case is pictured: No lookahead token has been read before the disambiguation takes place. This case – when disambiguating declaration from an expression – was particularly peculiar to idealize, see page 38.

Nested disambiguation, the need to resolve a would-be-conflict while already resolving one at higher level, utilizes the same concept. Specialized parsers are run, appropriate pad token is inserted, and the upper-level specialized parser continues.

3.1.3 Delayed Analysis

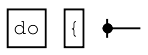
Parsing of some parts of input needs to be delayed in our analyzer, because type information needed to correctly analyze them might depend on declarations that have not been processed yet. The best example is in-class method definition. Lookup inside the method body must consider declarations from the whole class (§3.4.1:8), including ones that have not been read from the

[1]Note that we generally use the word *analyzer* when talking about the syntactic analyzer as a whole; and words *parser*/*automaton* when talking about one running instance of Bison generated code.

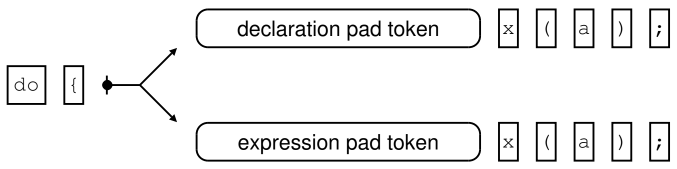
Figure 3.1: Inserting pad tokens in disambiguation.

Consider the following input at function scope:

```
do {
  x( a );
```



At this point, specialized parsers analyze upcoming input. Depending on their results, one of the pad tokens is inserted into the stream. The main parser then continues, reading the resolution from input, as if it were an ordinary token.



input yet. Thus, the contents of the method body must be preserved, not filtered through the hinter, and analyzed later, for example when the end of the class declaration is found.

Our solution to this issue is called *packing*. We find the token that terminates the sequence that needs to be processed later, cut out the tokens, and replace them with a single *pack token*, which holds the sequence of tokens that have been removed from the input. The pack token is simply shifted by the parser then. Later, when the semantic analysis has collected all the necessary information, a completely new instance of the syntactic analyzer is spawned, getting only the packed contents as its input.

The packing process is sketched in Figure 3.2 on the next page. The example input is presented as a sequence of tokens. When the body of the `clear` method is reached, we pack its contents and replace it with a pack token in the stream. If we were to analyze it immediately, the hinter would decide that `c` on line `(1)` is a type-name (and a syntax error would occur), while it actually refers to the data member declared below.

Detailed explanation of packing can be found on page 30.

3.1.4 Rewindable Input

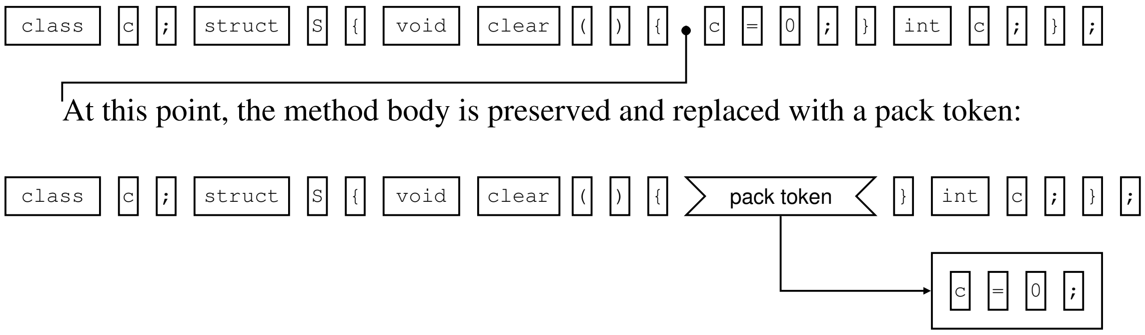
Disambiguation and packing place additional requirements on the input token stream. It needs to support insertion/removal of tokens; creating/deleting marks and rewinding back to them; and packing.

All manipulation of the token stream occurs in disambiguation *manager*. It buffers all tokens received from the preprocessor before filtering them through hinter and passing to the automatons. Before a disambiguation is started, the manager marks the current position in the buffer. When it ends, regardless of its result, the buffer position is rewound to the point where it was before the disambiguation started. After successful disambiguation, a pad token is inserted at the position.

Handling the pad tokens when there are nested disambiguations is more complicated. Semantics of nested transactions are employed: Whenever disambiguation *D* fails, any and all pad tokens

Figure 3.2: Example of packing a method body.

```
class c;
struct S {
  void clear() {
    c = 0;          // (1) refers to c below
  }
  int c;
};
```


class
c
;
struct
S
{
void
clear
(
)
{
c
=
0
;
}
int
c
;
}
;
At this point, the method body is preserved and replaced with a pack token:
pack token

that resulted from running disambiguations nested in *D* have to be removed from the token buffer.

The manager also performs actual packing, as it has exclusive access to the token buffer. Complete description of all purposes the manager is used for is available in section 3.5.2 on page 26.

3.1.5 Grammar

The grammar of the language, taken directly from the Standard, had to be heavily modified, since it contains many unnecessary conflicts. We were able to solve many of these conflicts just by exercising traditional techniques of making grammars LALR(1). The others are left to be resolved by disambiguation at run-time. Nevertheless, in some cases, adding disambiguation actions was not straightforward.

The following section covers the major changes that had to be applied to make the grammar acceptable for Bison and the disambiguation.

3.2 Grammar Modifications

The official grammar of the C++ language – found in the Standard – is not intended nor suitable for direct usage. The grammar alone covers a language that is a superset of C++, while many rules that tightly relate to syntax are set out throughout the text of the Standard, in English[2].

Thus, the grammar must be variously modified before it can be used in any parser (that is grammar-based). In this section, we mention techniques that we used to accommodate the

[2]It is very unfortunate that the Standard uses English language to formulate some rules that are clearly syntactic, instead of specifying grammar rules. This leads to confusion, as which errors to consider syntactic, especially in disambiguation that is supposed to be "purely syntactic" (§6.8:3).

grammar for use with both the Bison parser generator, and our concept of disambiguation. We also detail in which parts of the language we had to deviate from the standard grammar.

3.2.1 Techniques

Apart from countless "natural" modifications – like replacing a lone occurrence of a nonterminal with its only right hand side – we had to make many others, primarily to solve as much conflicts as possible without adding disambiguation nonterminals, which was the next logical step.

Techniques utilized in this process are well-known and we will not go into details when explaining them. The last one, based on specifying precedence, is not actually a grammar modification method, but a feature of Bison that allows solving conflicts by indirectly specifying order of precedence for the conflicting rules and/or tokens.

Left-factoring

Left-factoring is a method that is traditionally used when adapting grammars for recursive--descent parsers [4]. The idea is similar to factoring an arithmetic expression: Right hand sides of a production rule that have a common prefix are substituted for one rule with that prefix, followed by a new nonterminal that covers the alternative suffixes from the original right hand sides. Listing 3.1 gives an example.

This technique is usually used to solve conflicts that arise when generating an LL automaton. The algorithm has to descend to one of the right hand sides of the rule, but cannot make a decision, because the alternatives have a common prefix. By left-factoring the alternatives, the decision is postponed and carried out after the common prefix is processed.

For an LALR parser, common prefix of right hand sides does not introduce conflicts per se, as the decision is when reducing. However, we often need to perform an action after processing the prefix. The trivial solution of adding a nonterminal that derives to an empty word (ε) might help, but when the action needs access to the data from the prefix, we utilize a method that is similar to left-factoring.

Instead of creating a new nonterminal for the alternative suffixes, we create one that handles the prefix and includes the action. Occurrences of the prefix in the original rules are then replaced with this new nonterminal. Schematic example can be found in Listing 3.1, where `$1` refers to semantic information attached to the prefix – nonterminal `Y` that precedes the action code.

Note that although Bison supports middle-rule actions, it does not examine the action code and treats all middle-rule actions as different ones. Processing grammars that resemble the lower left one from the listing – without the suggested change – would yield reduce/reduce conflicts.

Apart from middle-rule actions (that are no longer middle-rule, after the change), we use the outlined technique to implement disambiguation based on semantic information. Its details are covered in section 3.5.3 on page 30.

Almost Identical Subgrammars in Different Contexts

We were not able to avoid copying subgrammars for use in different contexts. Some constructs have slightly different syntax when occurring at specific contexts. As one subgrammar for the

Listing 3.1: Left-factoring and middle-rule actions.

Left-factoring. Common prefix is preserved, the suffixes are moved into a new nonterminal:

```
A : X B              A : X B_or_C ;
  | X C         B_or_C : B
  ;       ⇝            | C
                       ;
```

The common prefix is moved to a new nonterminal when specifying middle-rule actions:

```
A : Y { action($1) } B                A : Y_action B
  | Y { action($1) } C                  | Y_action C
  ;                         ⇝           ;
                             Y_action : Y { action($1) } ;
```

construct is usually not able to handle the special case, we make a "deep" copy, modify it for the singular case, and use it in the context in question.

For example: Grammar for expressions is present in two almost identical copies, the only difference is that one prohibits the use of greater-than symbol (>) at the top level of the expression. This grammar is used for non-type template arguments, as the Standard states that the first non--nested > is to be interpreted as the end of the argument list (> being the closing angle bracket) rather than a greater-than operator of the expression (§14.2:3). Although the conflict – that would arise if only one grammar for expressions was used – is solvable with precedence (see below), we opted for the copy approach, as it is easier to maintain.

Another scenario, that requires a copy of a subgrammar, involves disambiguation. A particular construct is ambiguous with other constructs, but the set of these conflicting constructs depends on current context. As one disambiguation resolves ambiguities between a fixed set of constructs[3], the grammars have to be copied and modified for each of the contexts, so that the right kind of disambiguation is run it the right context.

The most outstanding example of a construct, that needs to be disambiguated differently in different contexts, is declarator. In namespace and function scope, its rule with parameter--declaration-clause is ambiguous with initializer, while in class scope this ambiguity does not exist. In the context of parameter-declaration, there is a conflict with abstract-declarator. See appendix A for a list of all the ambiguities.

Precedence

As mentioned earlier, Bison supports assigning precedence to tokens and rules. Whenever a shift/reduce, or a reduce/reduce conflict is encountered and the conflicting items have their precedence set up, a fixed resolution is selected in favor of the item with higher precedence. By items, we mean tokens and rules; in a shift/reduce conflict, the selection is between a token (to shift) and a rule (to reduce); in a reduce/reduce conflict, the selection is between two rules.

We mainly use precedence to solve shift/reduce conflicts that involve the less-than symbol (<),

[3]This limitation could be worked around, but then it would be necessary to explicitly inform the disambiguation about the current context. In our parser, this information is supplied implicitly, by having different subgrammar of a language construct for different contexts it can appear in.

Listing 3.2: Conflict involving scope resolution operator `::`.

```
typedef int type;
extern int x;     // (1)
::type ::x;       // parsed as "::[type::]x", ill-formed
                  // parentheses can be added to separate the name from ::
::type (::x);     // definition of x declared on line (1)
```

or the scope resolution operator (`::`).

When a template name is immediately followed by <, it must always be interpreted as opening angle bracket of template-argument-list, and never as a less-than operator (§14.2:3). Note that the conflict only arises after function template names, as class template names must always be followed by <.

In a simple-declaration, there is a conflict between `::` being interpreted as application of the scope resolution on the preceding name, or as global scope resolution for the following declarator-id. An example is shown on Listing 3.2. The resolution is always a shift, an application of `::` on the preceding name (§3.4.3:1). This conflict is present at other contexts as well, the resolution is the same, however.

In some C++ parsers (e.g. the parser found in older versions of GCC), the precedence technique is used to solve the dangling `else` problem[4]. We preferred to solve it by a grammar change, as it is straightforward to implement. We think of the precedence approach as a last resort, or when accommodating the grammar to solving a conflict would be too intrusive.

3.2.2 Major Changes

The Standard presents an oversimplified syntax for some constructs and provides additional constraints in the text. In particular cases, it was unavoidable to rework parts of the grammar to be able to adhere to these additional requirements. The major changes of this kind are explained further.

At Most One Type Specifier

In general, exactly one type-specifier is allowed in the declaration specifier sequence of a declaration (at most one: §7.1.5:1; at least one: §7.1.5:2), whereas the grammar alone allows any number, which consequently results in a great number of conflicts (for example with declarator--id). Nevertheless, there are exceptions to this rule that allow more than one type specifier to be combined in a declaration.

To fulfill most of these requirements, we categorize the type- and other declaration-specifiers into three groups and define possible combinations of specifiers from these groups.

- *Exclusive* type specifiers. All named specifiers belong to this group (class-name, enum--name, typedef-name, elaborated type specifier), as do specifiers that define classes and enums (class- and enum-specifier).

[4]The problem of coupling `if`s with the correct `else`s in code like `if(a) f(); if (b) g(); else h();`.

- *Non-exclusive* type specifiers. Simple type specifiers of built-in types are contained in this group. Some of them may be combined with each other in a well-formed program. All of them are keywords: `char`, `wchar_t`, `bool`, `short`, `int`, `long`, `signed`, `unsigned`, `float`, `double`, and `void`.
- *Other* declaration specifiers. This group covers the declaration specifiers that are not type specifiers (storage class specifiers, function specifiers, the `typedef` and `friend` specifiers) and cv-qualifiers. All of them are keywords.

Both following rules must be fulfilled for a declaration-specifier-sequence *S*:

- *S* can contain any number of other declaration specifiers.
- *S* must contain exactly one of the following:
  - Exactly one exclusive type specifier.
  - At least one non-exclusive type specifier.

S that follows these rules has either one exclusive specifier, or at least one non-exclusive specifier. Therefore at least one non-other specifier is present. Exclusive and non-exclusive specifier(s) are not mixed, while *other* declaration specifiers may be used without restriction.

S that violates the second rule is certainly ill-formed. It either has no non-other specifiers at all (thus has no type); or has exclusive specifier(s) mixed with non-exclusive one(s); or more than one exclusive (i.e. named) specifier is present.

We produced a conflict-free grammar that accepts a declaration-specifier-sequence if, and only if it satisfies the rules outlined above. Its purpose is not to detect ill-formed combinations of the specifiers, but to greedily consume the specifiers while the sequence complies with the rules[5]. Note that as soon as it does not, it cannot be well-formed (not vice-versa). When a specifier, that would make the sequence violate the rules, or a token that is unrelated to declaration specifiers, is read into the lookahead, the grammar ensures that it is not added to the sequence and the sequence gets closed.

This subgrammar of declaration specifier sequences no longer conflicts with declarator-id in a declaration. On the other hand, additional checks by semantic analysis are required to recognize illegal sequences (for example, ill-formed combination of “`signed unsigned const const`” is accepted by the grammar).

In contexts where a type specifier sequence is expected (rather than a declaration specifier sequence), the same principle is used, but declaration specifiers are left out from the *other* group – it only contains cv-qualifiers (`const` and `volatile`).

Constructors

Syntactically, constructors do not fit into the rest of the C++ very well. They resemble functions that return the class type they construct, but with an empty name. Conversely, they might be thought of as functions that have no return type, having the same name as the class type they construct. Variation of the latter definition is used in the Standard (§12.1:1), while the former suits the inherent disambiguation better.

[5]The Standard mandates this behavior (§7.1:2).

Listing 3.3: Examples of rare constructors syntax.

```
class c {
   (c()),                 // redundant parentheses around constructor declarator...
   (c)(&c),               // ... around the class name of copy-constructor...
   ((c)(c,c)),            // ... around both the name and the whole constructor declarator
   ((operator int)()),    // parenthesized name and declarator of a conversion operator
   (~c)();                // destructor name in parentheses
};
```

Declarations and definitions of constructors that conflict with ordinary member declarations at class scope, or with class member definitions at namespace scope, are handled in using semantic information (see 3.5.3 on page 30).

However, cases when the constructor declaration does not conflict, have to be handled separately. These include the class name and/or the whole constructor declarator being enclosed by a pair of redundant parentheses; constructor declarations mixed in a member declaration list with declarations of other special member functions that – syntactically – lack return type (their declaration specifier sequence does not denote any type; §9.2:7); and any combination thereof. Example of well-formed code that combines all of this syntax is presented in Listing 3.3.

We had to develop a completely new subgrammar to cover this rarely used syntax for declaring special member functions. It does not interfere with the rules that use disambiguation (see previous paragraphs) and is unambiguous and conflict-free.

3.3 Schematic Overview

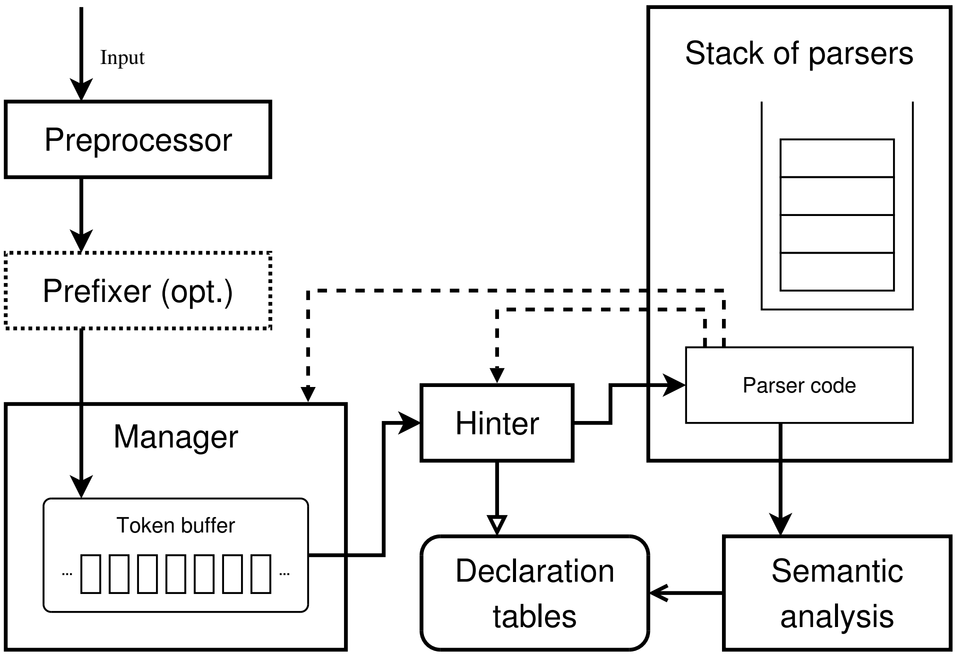
A scheme representing interaction between individual components of the analyzer can be found in Figure 3.3 on the following page.

The input is lexically analyzed and preprocessed by preprocessor. Its output consists of tokens, that enter prefixer, if it is enabled (see 3.4.4). The tokens are then buffered in disambiguation manager, which handles rewinding of the input stream, as well as inserting special tokens into it (3.5.2). Before reaching parsers (3.5.2), the tokens are filtered through hinter (3.4), which categorizes identifiers, looking up information in declaration tables. Parsed input is passed to semantic analysis.

Disambiguation and, consequently, the disambiguation manager are controlled exclusively by the parsers (3.5.1). Additionally, the disambiguation manager performs packing (3.6), which is initiated by the parsers, again. They are also responsible for switching the hinter into different modes of operation, based on the parsing context (3.4.2).

In the diagram, input data flow is represented by sharp arrows with solid lines; arrows with dashed lines represent actions in the parser code. Different operations on the declaration tables – lookup and filling – are outlined with different arrows.

Figure 3.3: Overview of the analyzer.



3.4 Hinter

Hinter provides the automatons with type information. It acts as a token filter that catches identifiers only, letting other kinds of tokens through unchanged. Lookup is performed for every identifier and depending on the lookup result, a different kind of token is passed further. The hinter must be a stateful machine, as it the kind of lookup it has to conduct depends greatly on the context of the identifier.

3.4.1 Shallow Pipeline

The "pipeline" between the hinter, the parsers, and the semantic analysis is kept shallow: at most one token is kept cached – in the lookahead of the LALR(1) automaton. This allows the hinter to provide accurate type information; as soon as the semantic analysis introduces a name to the symbol tables, the hinter can find it immediately (in the case when the very next token is an identifier and refers to the just-introduced name). It is important that the tables are always updated *after* some non-identifier token has been read (e.g. `,`, `;`, =), when no further token (possibly an identifier) needs to be read (cached). Should the pipeline be any deeper, it would be difficult to update type information in the cached token and/or roll back execution of the automaton.

Line (3) on Listing 3.4 shows an example. While the code is admittedly not very useful, it demonstrates the need for a shallow pipeline. See "Point of declaration" rules in the Standard (§3.3.1).

Listing 3.4: The pipeline between hinter, parsers, and semantic analysis must be shallow.

```
class c;        // (1)
int f() {
  char c        // (2)
         = c;   // (3) refers to c on line (2), not on line (1)
}
```

3.4.2 Modes of Operation

The hinter operates in several different modes. These are dictated by the kinds of name lookup that need to be performed (§3.4). These modes are not selected by the hinter itself, because it generally does not have enough context information to reliably decide which to use. Rather, the parsers control the hinter in this respect. Whenever a context change – that relates to the hinter operation – takes place, the parser sets up the hinter accordingly.

The state of the hinter (its mode of operation) must be restored when the input is rewound. This happens when a disambiguation finishes (commits or rolls back). Therefore, whenever a disambiguation is started, the state of the hinter is cloned and pushed on a stack. When performing lookup, only the state at the top of the stack is used; analogously the parsers control this state only. Restoring the state, when a disambiguation finishes, is then as simple as popping an item from the state stack.

Generally, the modes are left as soon as the next identifier is processed by the hinter. The parser needs to set the mode again, if it is needed. We found this technique to be easier to implement, compared to an alternative where the modes are permanent and have to be explicitly switched off by the parser. The alternative approach would require adding many switch-off actions to the grammar[6]. Automatically leaving a mode and setting it again – as appropriate – by the parser, was deemed a better solution.

Description of the various modes of the hinter follows. Note that they are not mutually exclusive in general. Some of them have not been implemented yet.

Qualified Name Mode

This mode is entered whenever the parser finds an application of the scope resolution operator (`::`). The hinter is passed a reference to the scope that it is supposed to use in the next lookup (§3.4.3). The mode is left as soon as the next identifier is processed by the hinter.

When parsing specific constructs, the parser forces the hinter to leave this mode. For example, when analyzing a pointer to member operator (§8.3.3), the `::` operator is followed by a star (`*`). The parser informs the hinter to quit the qualified mode, because any subsequent identifiers must be looked up in the enclosing scope, not in the one that the `*` was qualified with.

Note that the hinter never enters this mode, even after it performs a lookup of a name that is followed by the scope resolution operator. Entering modes is controlled exclusively from the parser.

[6]In some cases, these switch-off rules are unavoidable. See Qualified Name Mode on the current page.

Listing 3.5: Elaborated type specifier mode of the hinter.

The mode is entered after shifting the `class` token, left after the hinter processes `c`:

```
class c x;
class ::a::b::c y;
```

Elaborated Type Specifier Mode

To denote a class- or enum-name that has been hidden by a non-type declaration, `class`, `struct`, `union`, or `enum` keywords can be used to prefix such name. Lookup then ignores non-type names when searching for the name in question (§3.4.4).

The hinter holds the prefix keyword that was used. The parser turns this mode on as soon as it shifts over the keyword. This mode is left only when the last parts of a possibly qualified name is processed, as the prefix does not have an effect on the qualification part of the name.

For an example, see Listing 3.5. In both lines of code therein, the parser switches the hinter into the elaborated type specifier mode after shifting the `class` keyword. The mode is automatically left by the hinter after processing `c`.

This mode also covers the `typename` keyword prefix (§14.6:3).

Namespace Mode

When looking up names in specific contexts, only namespace names are considered (§3.4.6). The hinter state has a boolean flag that turns this lookup mode on. The mode is left as soon as the next identifier is processed by the hinter.

Member Access Mode

In expressions that contain one of the member access operators (`.`, `->`), the lookup needs to be performed in the scope of the class type on the left side of the operator (and in some cases in other scopes too, §3.4.5). In this mode, the hinter keeps a reference to the scope that corresponds to the type of the expression that the member access operators is applied to.

The mode is left as soon as first identifier is processed by the hinter.

Template Mode

This mode handles situation when the `template` keyword is used in the input to signal the parser that the following identifier is a template-id. This can be done in many constructs.

Dependent Name Mode

When the parser processes types and expressions that depend on values or types of template parameters (§14.6.2), it switches the hinter into this mode. The hinter does not perform any lookup then. The category of the resulting token depends solely on previously used `typename` and `template` keywords (see elaborated type specifier and template modes above).

3.4.3 Identifier Categories

Depending on the operational mode, and on the result of the lookup, the hinter categorizes the identifiers token. Each category has its own kind of token that is created and passed to the parser. The parser and the grammar distinguish these, and only allow the correct kind of identifier category in the individual constructs.

The following table summarizes identifier categories, as well as lookup results that lead to their usage.

Identifier category	When this category is used; notes.
Unknown identifier	Lookup did not find a declaration for the name. In the parser, this category is handled the same way as a non-type (below).
Non-type name	A non-type name is found. For example, a name of an object.
Namespace name	A namespace name is found.
Typedef name	A typedefs that is not a class typedef, nor a class template typedef is found. For example, an `int` typedef.
Class name	A class name, a class typedef, or a class template typedef is found.
Enum name	An enumeration type is found.
Non-type template name	A function template is found.
Type template name	A class template is found.

It is very important to note that the lookup might also depend on the token that follows the identifier. If it is the scope resolution operator (`::`), a different set of rules must be utilized when during the lookup (§3.4.3:1). Therefore, the hinter always looks at two tokens: the identifier, and the token that follows it. This is different from the qualified name mode described above, as that deals with names after the `::` operator.

When the hinter does find an identifier followed by the `::` operator, it does not automatically switch to the qualified name mode for the following identifier. This is solely the parser's responsibility. An alternative implementation could execute the switch automatically by the hinter, with the parser only having to control leaving the mode in special constructs (see documentation of the mode on page 18).

In Lestes, the set of declarations found by the hinter is stored into the newly created token. This is an optimization, as the lookup does not have to be performed again when the semantic analysis tries to find out what entities (declarations) the identifier actually denoted.

3.4.4 User-Provided Type Information

During development of the analyzer, it was necessary to invent a hinter that would not look up the type information in declaration tables. As support for declarations that would trigger all the identifier categories was being implemented in the semantic analysis gradually, we needed means to force a category on an identifier.

The solution is called *prefixer*, as it allows the category of an identifier to be forced by a special prefix. The prefix – lexically an identifier – immediately precedes the identifier it affects. One prefix exists for each of the categories (e.g. `_hint_nontype` to force a non-type name). The prefixer processes the prefixes, deletes them from the input stream, and embeds the category information into the identifier tokens. This information is then used by the hinter instead of data provided by lookup.

The prefixer is only used in development – primarily to test the parsers. It is disabled in the final parser.

3.5 Disambiguation

The C++ language is *ambiguous*. This means that the same subsequence of tokens from input may be interpreted as two (or more) distinct language constructs, if only syntax rules are used in the analysis. In these cases, the Standard states which of the possible alternatives is an analyzer supposed to pick.

When an ambiguous grammar is processed by a LALR parser generator, like Bison, it usually reports that it contains *conflicts*. This means that the resulting automaton (finite-state-machine with stack) contains states in which there is more than one possibility of proceeding, based on the grammar rules. For a more detailed description of the parsing algorithm, see [4].

With Bison, which generates LALR(1) automatons, the conflicts do not arise only from truly ambiguous constructs. When generating tables that represent the automaton, Bison uses at most one token of lookahead to pre-compute decisions of the resulting machine. Whenever more than that one token is needed to distinguish constructs that are allowed to follow in a specific context, a conflict occurs. Increasing the length of lookahead to k – yielding an LALR(k) automaton – would not help with C++, as the decisive token can be arbitrarily far. Listing 2.1 on page 7 shows an example of code, where the decisive token (++ in this case) can be put arbitrarily far simply by adding items to the init-declarator-list (§7:4).

In our analyzer, *disambiguation* is used to overcome both kinds of conflicts: ones that result from ambiguities, and ones that would be solved if the automaton had means to look arbitrarily far ahead[7].

It is important to point out that it is generally not known whether the parser should look for a decisive token or expect an ambiguity. We will show that this kind of information is not needed for solving conflicts.

[7]Our concept of running specialized parsers does just that – looks arbitrarily far ahead to make a well-founded decision. It is explained in the following parts of this section.

3.5.1 Fundamentals

The top-level parser, that carries out the main analysis, is said to *parse for real*, and is an LALR(1) automaton generated by GNU Bison from a modified version of C++ grammar. The disambiguation-related conflict-solving modifications follow a simple pattern: a *disambiguation nonterminal* that derives to an empty word (ε) is added so that its rule is reduced before the conflicting state can be reached, and each of the conflicting alternatives is prefixed with a different, so-called *pad token*. Recall Figure 3.1 on page 10.

As a result, the modified grammar is conflict-free thanks to the pad tokens. Being different for each of the conflicting alternatives, the conflict does not arise. However, as the pad tokens are already expected by the production rules of the grammar, they need to be inserted into the token stream by the time the parser reaches them. The essential principle of the disambiguation concept is that the pad tokens are inserted at the very latest moment.

The disambiguation itself is launched in semantic actions[8] of the empty rules of disambiguation nonterminals, and has to insert a pad token. The pad token will be read by the automaton when the control returns from the rule action. Special precautions must be taken when the automaton reads the lookahead token before reducing by the rule with disambiguation code (see below).

The code of a disambiguation has mainly two options of deciding which of the expected pad tokens to insert. The simpler – yet rarely used – one is examining semantic information of the already analyzed part of input and base the decision purely on found data[9].

The other is to launch out-of-order *trial parsers* on the upcoming input to see if it can be parsed as one of the constructs that are allowed to follow. If it can, the corresponding pad token is inserted. The combined approach, examining both semantic data and upcoming input, is used only in one case, when disambiguating constructors (see A.3 on page 40).

A trial parser can reach a state where another disambiguation is needed to solve a would-be conflict. These cases are handled exactly as their top-level counterparts: the disambiguation can either examine semantic data, or launch another set of trial parsers to commit a pad token. A disambiguation that has to be performed while another one already takes place, is referred to as *nested disambiguation*; the one that already takes place is said to be on an upper level.

Note that it is not important for parsers – both top-level and trial (in cases of nested disambiguation) – which of the strategies was used to make the decision, as long as a pad token gets inserted[10].

3.5.2 Syntactic Disambiguation

When launching trial parsers to resolve a conflict, semantic analysis is turned off. Declaration tables are not modified, and semantic checks are not preformed. This complies with the Standard, as it requires that disambiguation is purely syntactic and may commit to a semantically ill-formed alternative, even when the other would be well-formed (§6.8:3).

In the same same paragraph, it is said that only type information can be used in a trial parse. As the hinter is on, the trial parsers use the type information implicitly: categories of identifiers at

[8]Semantic action is the term used in GNU Bison. It comprises of a code snippet that gets executed when the rule in question is being reduced by the automaton. We prefer to call this code a rule action.

[9]This concept is known as semantic predicates in the ANTLR parser generator [9].

[10]There are cases where the distinction is done by inserting a pad token in one of the alternatives, and not changing the token stream in the other. This can be considered a special case, with ε being one of the pad tokens.

specific contexts must match the grammar. In other words, fulfilling this requirement needs no extra attention, because the necessary constraints are already embedded in the grammar rules.

Order of Alternatives

In syntactic disambiguation, a trial parser is run to find out whether the upcoming input *can* be interpreted as specific construct. Each trial parser recognizes just one construct – the one it is specialized for. When we want to emphasize this property of trial parsers, we call them *specialized*.

A disambiguation in turn runs a specialized parser for each of the constructs that can possibly follow, one after another. After the first one that successfully finishes, no more alternatives are tried, and a pad token that corresponds to the successful try is inserted as the disambiguation result.

When all the trial parses fail, the disambiguation is said to fail. When such failure happens, the current parser found an error. On the top level, this translates to a syntax error in the input, which is reported to the user; failure of nested disambiguation makes the upper level disambiguation try the next alternative.

The order, in which the alternatives are tried to be parsed, is dictated by the Standard. In the declaration-statement versus expression-statement case, it explicitly calls for a trial pass and specifies the order therein (§6.8:1). In general, however, only the choice that is to be made when an ambiguity actually occurs is specified.

Nonetheless, the strategy of executing the disambiguation at all times and always picking the first alternative that successfully parses, deals with all of the possibilities. When the input – tentatively parsed as a set of constructs – is not ambiguous, only the construct that was successfully parsed is picked as the resolution. When the input is ambiguous, the preferred (sooner tried) interpretation is picked. Please notice that the analyzer does not "realize" which of these two possibilities occurred; doing so would entail more processing that is needed to resolve the ambiguity.

To summarize, syntactic disambiguation in our analyzer is built on the premise that all ambiguity resolution rules are formulated as precedence rules that specify linear ordering of the alternatives. In other words, the ambiguity resolution rules must provide us with an order of the alternatives. At specific points, upcoming input is tried to be parsed as each of these alternative interpretations, in the specified order. The first alternative that successfully parses is picked as the resolution.

Specialized Parsers

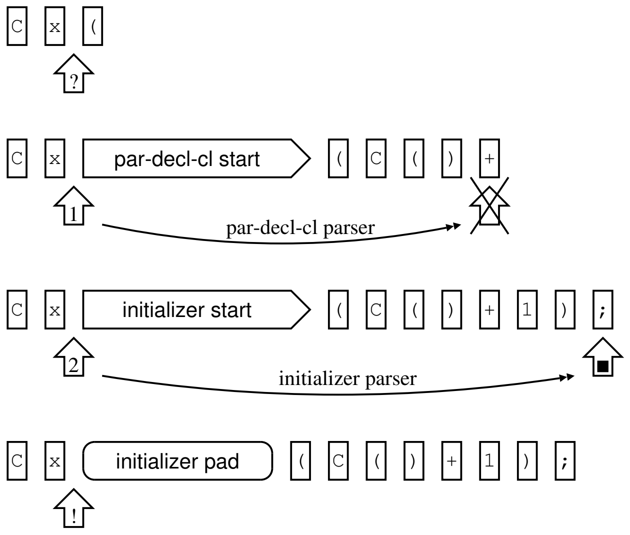
Specialized parsers are run from disambiguation to tell whether the upcoming input can be parsed as a particular language construct. For each construct there is a special parser that only accepts this construct as its input.

These parsers are generated from the same grammar file as the top-level parser. To effectively create separate automatons, each of the constructs we need a specialized parser for is prefixed with a different token and the resulting rule is added as an alternative starting rule of the grammar. In this respect, the top-level parser is no different, its construct is "a C++ program". We call these prefixes *start tokens*. Start token is read as the very first token of input for every parser and determines construct the parser is supposed to analyze.

Figure 3.4: Running specialized parsers in disambiguation.

Analysis of the following code on namespace scope:

```
class C;
C x( C()+1 );
```


C x (
?
C x par-decl-cl start (C ( ) +
1
par-decl-cl parser
C x initializer start (C ( ) + 1) ;
2
initializer parser
C x initializer pad (C ( ) + 1) ;
!

Despite generating one automaton that merges all the specialized parsers, the size of this grand automaton is not substantially larger than the one for the top-level parser only[11]. This is a property of the LALR(1) parser generation algorithm [4].

Generating all the parsers from one file has an important implication: nearly all of the code in the rule actions is the same for both the top-level parser that parses for real, and for the specialized trial parsers. This is not an issue, though. When needed, the code can query the disambiguation manager (see below) whether it is parsing for real or not. This way, syntax errors are only output when parsing for real, for example.

Before launching a specialized parser from disambiguation, a start token needs to be inserted into the stream. The insertion is done by the manager at the same time it marks the current position in the input stream. Running the specialized parser then consists only of recursively calling the parsing routine. If the alternative proves to be correct, the start token is simply replaced by a pad token.

An example of basic disambiguation is depicted in Figure 3.4. After processing `x` and seeing opening parenthesis in the lookahead, there is possible ambiguity between a parameter--declaration-clause and an initializer. The need for disambiguation is marked by a question mark in the arrow that points to the position in input. As per §8.2:1, parameter-declaration--clause is tried first. Its specialized parser fails at the `+` token[12] (the scratched arrow stands for a syntax error). Input is rewound and the second alternative is tried. Initializer parser succeeds after processing the whole initializer (successful parse is pictured as arrow with a filled square). An interpretation is found. A pad token is inserted, and parsing for real resumes (exclamation mark in arrow).

[11]Adding 38 rules with start tokens increased the number of states by a mere 6 %, from 1338 to 1418.

[12]There is a nested disambiguation within the parameter-declaration-clause trial parse, but is not relevant to the outlined example.

Nested Disambiguation

After inserting the pad token and resuming parser at the upper level, the input previously analyzed during disambiguation is analyzed again. However, all the nested conflicts that have been resolved as a part of the just finished disambiguation, have their resolutions (pad tokens) inserted into the stream. Still, the rules of the disambiguation nonterminals are reduced, and the disambiguation code is run. A plain check is used to skip launching the actual disambiguation again: see if the following token is a pad, and when it is, do nothing. See Listing 3.6 on page 29.

Conflicts and, thus, disambiguations can nest, and disambiguation at every level has to try all the possible constructs (until one that can be successfully parsed is found; or make the upper level disambiguation fail, if none is found). This means that when looking for a possible interpretation of the upcoming input, back-tracking is performed, in essence.

Disambiguation after Reading Lookahead

When a disambiguation inserts a token into the stream (and that is the purpose of running the disambiguation), special precautions must be taken, because the automaton might have already read the next token into its lookahead. Two problems originate from this.

First, the disambiguation code has to ensure that the start/pad token is inserted at the right position, i.e. before the token that has been just read. This is accomplished by telling the manager to *back up* – move its current position pointer one token backwards, so that when another token is requested, it returns the same token it returned last time[13]. The lookahead of the automaton has to be cleared as well. It we be re-read as necessary.

The other issue is that the automaton might have already used the lookahead to make a decision. The automaton might expect to see one of the pad tokens in lookahead to actually reduce the empty rule of the disambiguation nonterminal. In this case, the disambiguation is never run, because the pad tokens are never found in the input and only inserted by disambiguation, and the automaton usually fails immediately. Fortunately, it is usually known which tokens should trigger the disambiguation. By creating bogus rules that have the disambiguation nonterminal followed by these tokens, Bison generates an automaton that also reduces the rule (and runs the disambiguation) when is finds one of the trigger tokens in lookahead. Listing on page 29 showcases this bogus rule. What the listing does not show, is that the lookahead must be cleared by the disambiguation code. After the disambiguation code finishes, the automaton has to re-read and re-evaluate the lookahead before making another step. This re-evaluation allows us to change the next incoming token in the disambiguation.

In some cases, the problem – that the automatons expect a pad token on input to launch the disambiguation – remains hidden, as the reduction of the disambiguation nonterminal rule might be merged with other reductions in a default reduction action. It can be thought of as an else branch: if the lookahead is not among a few particular tokens (for which there are special actions), the default action takes place. If the pad tokens get merged into the else branch, the problem does not emerge. Nevertheless, apparently unrelated changes to the grammar may move the pad tokens out of the default action, because the grammar user does not have control over the process. The bogus rule method works at all times.

Obviously, it would be ideal to launch disambiguation only in states when the lookahead of the parser is empty. Indeed, in some cases this is possible and the grammar does not have to be

[13]As the token is filtered through the hinter again, it is possible that its category changes if it is a identifier.

modified before the disambiguation nonterminals can be added to the appropriate places. In many other cases, this would be far from straightforward. Our implementation uses assertions that guard entry to every disambiguation. The developer is notified whenever the lookahead is empty and the disambiguation expects it not to be or vice versa. To set the assertions correctly, the description of the automaton – output by Bison at generation time – has to be examined.

Disambiguation Manager

Disambiguation manager is responsible for buffering tokens from the input stream before they are passed through the hinter to the parsers. It also manages disambiguation – inserts and removes pad and start tokens, and rewinds the input stream.

The manager also performs packing, as it has exclusive access to the token buffer. Packing is described on page 30.

The buffer is implemented as a list, with the current position pointer being an iterator [11]. Marking and rewinding the stream is straightforward – the current pointer is stored, or replaced with previously stored value, respectively. When parsing for real, only one token is left in the buffer and is used when disambiguation needs to back up a token read as lookahead (see above). When parsing for disambiguation purposes, no tokens are removed from the buffer, as they will have to be provided again, after the stream is rewound. The tokens are stored before they are filtered through the hinter.

The disambiguation manager treats every alternative that is probed during one disambiguation as a separate transaction. Particular disambiguation can then be viewed as a sequence of transactions that are tried to be completed in a fixed order, and when one transaction succeeds, the sequence finishes. In fact, the manager is only aware of the transaction abstraction, as it does not need the notion of the whole disambiguation.

Consequently, transactions terminology is used throughout the manager. Before launching a trial parser, a start token needs to be inserted into the token stream buffer before the current position, and the position has to be stored for later rewinding. This is what the manager understands under *starting* a transaction. Transaction *commit* is interpreted as a request to rewind the token stream to the position it was before starting the transaction, and replacing the start token – inserted when starting the disambiguation – with a pad token; the pad token will be the next token passed to the parser. When an alternative proves to be wrong, as a result of the specialized parser failing to parse upcoming input, a *rollback* of the transaction is requested. The manager has to revert all changes made to the token stream since the transaction has been started.

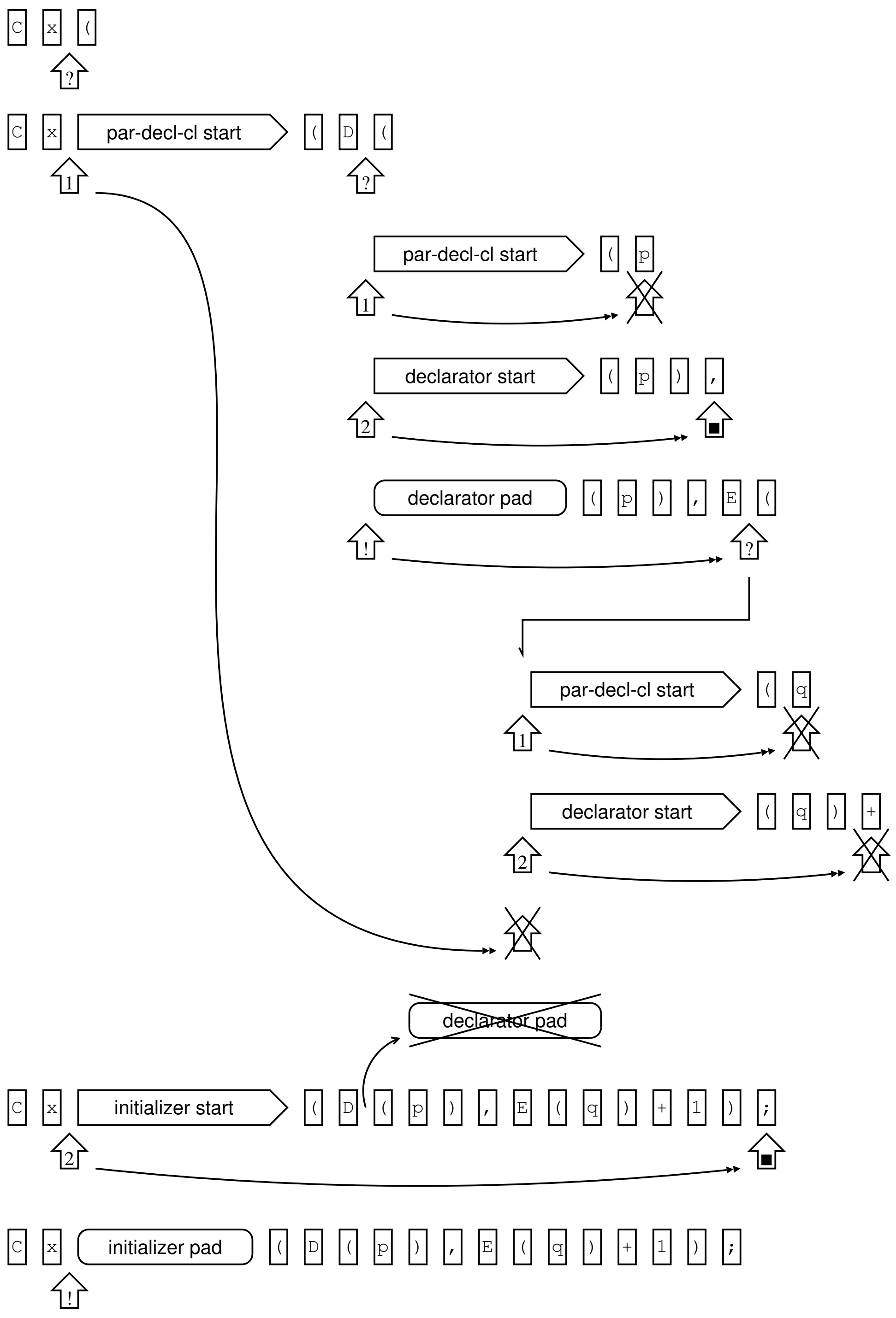
The disambiguations take place in stack-like fashion: when a nested conflict resolution is needed, it must be finished before we return to an upper level conflict. Thus, the current transaction must be committed or rolled back before we can commit or roll back the parent one.

Disambiguation manager is the only component of the whole analyzer that has notion of disambiguations that are currently being conducted. Actually, it only stores information about the transactions involved. This means that it does not know which conflict the analyzer is currently solving, yet it can tell which alternatives are being tentatively parsed. The rest of the analyzer does not need to store information about the current progress of disambiguation – it always uses the manager to commit or roll back the newest transaction, or start a new one. The only exception is a boolean flag stating whether we are parsing for real or not (i.e. whether, respectively, any or no transactions take place at the moment). This flag is used in the parsers (see page 29), and to skip calling the semantic analysis, as it is turned off during trial parsing.

Figure 3.5: Nested disambiguation failure.

Nested disambiguation failure occurs during analysis of the following code.
In the process, an already committed pad token has to be removed from the token buffer.

```
class C; class D; class E;
C x( D(p), E(q)+1 );
```



C x (
?
C x par-decl-cl start (D (
1
?
par-decl-cl start (p
1
declarator start (p) ,
2
declarator pad (p) , E (
!
?
par-decl-cl start (q
1
declarator start (q) +
2
declarator pad
C x initializer start (D (p) , E (q) + 1) ;
2
C x initializer pad (D (p) , E (q) + 1) ;
!

To implement transactions that nest, which happens when a nested disambiguation is needed, we use a simplified model of nested transactions known from the database world. Whenever transaction *T* fails (and the manager rolls it back), all of the pad tokens that were inserted by transactions nested – not necessarily directly – in *T* need to be removed from the token buffer. In other words, a commit of a transaction is not final until all of the upper level transactions commit as well. Thanks to the fact that the disambiguations follow a stack-like pattern, the implementation is not complicated. Every transaction holds a list of tokens it has to remove should it fail. When it commits, it hands this list and a its resulting pad token over to the parent transaction. At the top level, where there are no parent transactions, the list is simply forgotten – all the tokens in it are inserted permanently and cannot be removed anymore.

Figure 3.5 contains an example of a nested disambiguation failure. For a description of the symbols used, see the previous figure on page 24.

There several possible ambiguities in the code. The first needs to be resolved after reading `x`, when an opening parenthesis is ahead. Parameter-declaration-clause (referred to as p-d-clause throughout this paragraph for brevity) and initializer will be tried, in this order. In the p-d--clause alternative, tried first, `D` is parsed as the declaration-specifier-sequence (type) of the first parameter. At the opening parenthesis, a conflict between p-d-clause (that would define the parameter type as pointer to function type) and declarator in redundant parentheses arises. The latter proves to be correct after the p-d-clause alternative for "`(p)`" is dismissed (`p` is not a type name, thus cannot declare type of a parameter) – a declarator pad token is inserted.

After `E`, type of the second argument, the same disambiguation takes place. This time, however, both trial parsers fail: the first one for the same reason as in the first parameter case, `q` is not a type name; the second because + cannot appear in a declarator. A syntax error is found and the trial parser fails looking for a p-d-clause following `x`. The corresponding transaction must be undone. This entails removing the declarator pad token committed as resolution of the conflict within the first parameter[14]. The input is rewound and the input following `x` is tried to be parsed as an initializer. The trial parser succeeds and a pad token is inserted as a resolution for the top-level parser.

Disambiguation Code Example

Listing 3.6 presents a typical disambiguation. Parameter-declaration-clause (pdc) versus initializer (inizer) conflict was chosen, as it displays many of the problems and strategies outlined in the syntactic disambiguation section. For exposition purposes, the code as well as the grammar substantially differ from their counterparts used in our analyzer. Further information about this ambiguity can be found in the Appendix on page 39.

The conflict this disambiguation solves arises on namespace and function scope and occurs when the declarator-id is followed by a left parenthesis (`(`). The first thing to notice is the bogus rule that ensures that the disambiguation is launched when the lookahead contains the parenthesis. However, it is not launched when the lookahead contains opening square bracket (`[`).

The disambiguation nonterminal (`disambiguate_pdc_vs_inizer`) derives to ε and action of its rule contains the actual disambiguation code. We will discuss the code.

[14]Note that finally `D(p)` is not interpreted as a parameter declaration, but rather as a function-style cast expression. However, at the time the pad token in question was committed, the analyzer worked with the assumption that `x` is followed by a parameter-declaration-clause.

Listing 3.6: Complete disambiguation example, including pseudo-code.

```
declarator_suffix : disambiguate_pdc_vs_inizer PAD_PDC param_decl_clause
                  | disambiguate_pdc_vs_inizer PAD_INIZER initializer

                    // execute the disambiguation when ( follows, too:
                  | disambiguate_pdc_vs_inizer '('
                    { internal_error(); } // we should never get here

                    // no disambiguation needed for the array suffix:
                  | '[' const_expression ']'
                  ;

direct_declarator : declarator_id declarator_suffix ;  // simplified for brevity

param_decl_clause : '(' param_decl_list ')' ;
initializer       : '(' expression_list ')' ;

disambiguate_pdc_vs_inizer:  /* empty */
{
  // when the conflict was already solved as part of a nested disambiguation, do nothing
  if (lookahead_token_is_pad())
    goto done;

  // try parsing upcoming input as parameter-declaration-clause
  manager::start( START_PDC );
  if (parser::parse()) {
    manager::commit( PAD_PDC );
    goto done;
  }
  manager::rollback();

  // if the input could not be parsed as parameter-declaration-clause, try initializer
  manager::start( START_INIZER );
  if (parser::parse()) {
    manager::commit( PAD_INIZER );
    goto done;
  }
  manager::rollback();

  // all tries failed. if on the top level, report syntax error; otherwise force the parser to fail
  if (manager::in_disambiguation())
    YYABRT;  // this makes the parser function return immediately
  syntax_error();

done:
  ;
}
```

It has to detect whether the conflict is already solved by looking at the lookahead. If it is a pad token, the disambiguation was already run and the resolution is inserted. Nothing more needs to be done.

If a resolution has to be computed, the manager is told to start a transaction and insert a start token. A trial parser – specialized for pdc – is launched. If it succeeds (the return value of the function tells this), the manager is signaled to overwrite the start token with a pad token and we are finished.

If the trial parser fails, the manager rolls back the transaction, rewinds the input and removes the start token. The same trial parse is started for an initializer.

If it does not succeed either, we check the manager whether we are parsing for real or not. If not (we are in a disambiguation), the current trial parser is forces to exit with a failure. The disambiguation at upper level disambiguation (e.g. a statement versus expression disambiguation) will try the next alternative. If we are at the top level, it is an syntax error when we are not able to find a valid interpretation. The error is reported to the user.

3.5.3 Using Semantic Information to Solve Conflicts

In a few cases, it is possible to solve a conflict by examining semantic information, usually information about constructs that immediately precede the conflicting state.

However, a disambiguation nonterminal that derives to ε cannot be used if we want to access data that is physically stored on the stack of the automaton (semantic information attached to preceding nonterminals and terminals, that are not yet reduced). A technique similar to left-factoring is used to place the decision-making code so that it has access to the required data (see 3.2.1 on page 12 for full details on this technique).

The desired pad token is inserted by signaling the manager to start and immediately commit, without running any specialized parsers in between.

3.6 Delayed Analysis – Packing

As mentioned earlier, specific constructs cannot be analyzed immediately after being read from the input, because the analysis needs type information that is not available yet. Parts of the input that need to be parsed later are stored sideways and analyzed when the information is present. We call this process *packing*.

The constructs that must be packed are usually designated by the Name lookup section in the Standard (§3.4).

For illustration in this section, we will be using the most common construct that must be packed – method body defined in class. It needs to be analyzed in context of the whole enclosing class, thus needs to be packed (§3.4.1:8). See Figure 3.2 on page 11.

3.6.1 Packing Parts of Input

Packing is always initiated from rule actions within the parser. The nonterminal with empty (ε) right hand side is inserted after the token that prologues the construct that needs packing. In our example, this is the left opening brace of the method body ({). Then, the terminating token is found and the sequence between these two tokens is replaced with a pack token that itself contains the cut-out part of input.

The grammar rule in question follows the pattern on line (1) on Listing 3.7, except that for the method body, packing is performed at all times, as it is never reached during disambiguation.

The terminating token (balanced closing brace, }) is found using simple hand-coded finite state automaton with a stack. We have made some experiments trying to generate the automaton Bison, using the same trick as with the specialized parsers: Generate it from the same grammar file as the rest of the parsers. However, as Bison does not provide wild-cards, we found it too complicated for finding the matching brace. Either way, the implementation is general enough to add Bison-based packing method to be added at a later time[15].

The actual packing takes place within the disambiguation manager, as it has exclusive access to the input token buffer. When it has information about positions of the first and the last token of the pack, it simply cuts the sequence in between them and stores the subsequence into a newly created *pack token*, that is inserted into the token buffer instead.

The tokens in the pack must be stored without being filtered by the hinter. In general, the hinter is off when searching for the terminating token, but might be turned on when packing specific constructs. Either way, the packed values are retrieved from the manager's token buffer, where they reside as read from the preprocessor.

3.6.2 Analyzing Packed Input

Pack tokens are the only handles to the packed token sequences. They are saved for future processing by the semantic analysis. It also decides when to parse the packed data. The general rule is simple to formulate – after collecting all the needed information. In the in-class method body case, this translates to the moment when the whole class body is parsed and the symbol tables are filled.

To analyze contents of a pack token, a new instance of the whole syntactic analyzer is spawned. The pack token itself is used as the *start token* (see page 23), the packed sequence of tokens being the actual input to the newly spawned analyzer. It is fully reentrant, which means that is does not interfere with other analyzers that are currently running[16]: it has its own disambiguation process, its own copy of the hinter, and it can even pack parts of its input. In other words, the whole mechanism is completely transparent and it does not matter whether the actual input is read from the preprocessor, or from pack token contents.

Obviously, before the analysis of the packed data can begin, the semantic analysis must appropriately set up its contexts, as well as the context for the lookup performed by the hinter.

3.6.3 Packed Constructs

The following in-class constructs require packing:

- method bodies, this includes the function-try-block form (§8.4:1, §15:3)
- default arguments of method declarations

[15]This might actually be required for the default argument packing, to be discussed later in this section.
[16]In fact, due to the nature of the analysis, there is a stack of analyzers and only the one at the top can be running.

- constructor initializers (§8.4:1, §12.6.2)

To generalize the code as much as possible, we decided to pack all function bodies. However, those not at a class scope are processed immediately, unlike their in-class counterparts.

Packing is also used for simplistic error recovery. Although the parser was not designed with error recovery in mind, at points where all disambiguation tries (at top level) fail, it is possible to skip the erroneous input and continue further. The skipping is implemented by packing the erroneous part of the input. At the moment, this technique is utilized when skipping conditions (in `if` and loop statements), or when the declaration- versus expression-statement disambiguation at function scope does not find any possible interpretation. In the latter case, the terminating token is the next semi-colon (`;`) or a keyword like `if`.

3.6.4 Packing in Disambiguation

Packing can take place when doing a disambiguation. Bodies of classes or enumerations defined locally in functions cannot be analyzed during the statement versus expression disambiguation (see A.2 on page 38), as that would almost certainly fail – the tables are not filled when disambiguating. And when the trial parsing fails, the analyzer might not recognize a well-formed declaration, reporting a syntax error.

To solve the problem, whenever the opening brace of a class or enumeration body (member--specification or enumerator-list, to be exact) is read during disambiguation, the tokens up to the closing brace are *packed* and the pack token is shifted.

When the trial parser succeeds and the pack token is later encountered when parsing for real, it is unpacked and the parser normally continues[17]. When the trial parser fails, the manager unpacks the tokens automatically during rollback. It is important to note the packing and unpacking happens *immediately* after shifting the opening brace, when the automaton does not need to read the lookahead token. This timing of the action allows us to modify the token stream in a way that is transparent to the automaton.

The grammar excerpt with pseudo-code in Listing 3.7 illustrates the solution. After the opening brace of the class body has been shifted by the automaton, `pack_in_disambiguation` is reduced (it derives to ε) and its action is executed. If we are in a disambiguation, the manager is requested to pack the input up to the closing brace. The automaton then shifts over the `BODY_PACK` token that is inserted instead of the contents of the class body. When disambiguation finishes, the input is rewound to a position before the packed contents. When the opening brace is shifted again, we check whether the following token is a pack token (containing previously packed class contents) and if it is, unpack it, so that the parser can continue parsing the real contents of the class. When `class_contents` is only parsed for real, the code does not alter the input in any way.

One might object that classes cannot be defined in expressions, so when we reach a body of a class, we can be sure it is a declaration. This is true. What is more, the `class` keyword cannot appear in expressions either. One idea that exploits this fact for an optimization is outlined in possible performance enhancements in the conclusion on page 36.

[17] Alternatively, we could spawn another analyzer, just as with method bodies.

Listing 3.7: Packing in disambiguation.

```
class_contents : '{' pack_in_disambiguation BODY_PACK '}'        // (1)
               | '{' pack_in_disambiguation member_declarator_seq '}'
               ;
pack_in_disambiguation: /* empty */
{
  if (in_disambiguation())
    manager::pack_body();
  else if (lookahead_token_is_pack())
    manager::unpack_next();
};
```

3.6.5 Efficiency Considerations

Search for the terminating token is conducted by a deterministic automaton that simply scans the input. The effective time is linear in the amount of tokens packed. This claim holds true even when future packing methods are implemented by different kind of automaton that does operates in linear time (e.g. a Bison-generated parser).

Cutting out a portion of the token buffer (the final part of packing), as well as unpacking (in disambiguation), is implemented using `splice` operation of the `list` template from the standard Containers library (§23). While the Standard allows linear complexity for the operation (§23.2.2.4:14), the current implementation in GCC, which we primarily used in development, takes only constant time to complete. The same efficiency is declared by at least one other implementation, namely the one from SGI [11].

Preparing the pack token contents for analysis is a constant-time operation, as the token sequence does not have to be processed in any way.

3.7 Checks Performed by Semantic Analysis

Many errors are not being detected by our parser itself. We generally tried to deviate from the grammar in the Standard as little as possible. While it is conceivable to detect some non--syntax[18] errors in the parser (by modifying the grammar accordingly), we opted for implementing the checking in semantic analysis[19]. Detecting errors in the semantic analysis is often easier to implement, and – what is more – the semantic analysis can usually output better error messages for the user.

In a few specific cases, it is crucial that the parser does not try to detect ill-formed constructs and leaves this task up to the semantic analysis. At other places, where the grammar accepts a superset of a construct, we rely on semantic checks to immediately report an error. Both of these problems are related to disambiguation, and are explained in the following paragraphs.

[18]By syntactically erroneous input, we mean input that does not match the grammar rules, or violates a "syntactic" rule found in the text of the Standard. Footnote on page 11 outlines why this definition is inexact.

[19]Despite the fact that the simplest semantic checks reside directly in the parser source code (in rule actions), they are semantic, because they examine the syntax tree (produced by the parser) when looking for errors.

3.7.1 Semantic Errors Detectable by Syntax

To imagine an error that could be checked for by both syntactic and semantic analysis, consider function definition. Function body is only allowed to appear after a declarator that really declares a function. If it declares a pointer – for example – the program is ill-formed. While it is possible to detect such errors in the parser, a grammar that would accept function bodies only after the correct declarator, provided that the declarator can be hidden deeply in parentheses. On the other hand, it is really trivial to carry out a semantic check by examining the first "operator" of the declarator that is applied to the declarator-id. If it corresponds to a parameter-declaration--clause (§8.3.5), the declaration can contain a function body. The check could be implemented with a boolean flag to indicate whether a function body is allowed to follow the declarator in question; in Lestes, the first of the sequence of declarator operators is looked at [7].

Similar approach could be used for distinguishing pure-specifier ("`= 0`", §9.2) from constant--initializer ("= *constant-expression*") in classes. It would suffice to use one grammar rule with constant-expression, as it covers the pure-specifier case. The check would consist of inspecting the structure of the expression and the value of the numeric literal. However, our analyzer currently stops after reading the equals symbol (=), and queries semantic analysis whether the preceding declaration declares a function or not. A pad token is then inserted to guide the automaton to the two different rules[20]. The reason behind this decision was pragmatic: the underlying structures used for representing the parsing tree in Lestes cannot express parenthesized expression, so "`= (0)`" could be incorrectly accepted as a pure specifier.

Semantic checks are also utilized when handling declaration specifier sequences (§7.1). Combinations like "`long char`" or "`virtual explicit`" are recognized as ill-formed by semantic analysis, not by the parsers.

The main reason to move these simple checks away from the parser is that the grammar of the language can be kept reasonably complex then. Alas, there are cases where major modifications are required. Explanation of the most important ones starts on page 14.

3.7.2 Parsers Must Ignore Semantic Rules

When parsing tentatively during disambiguation, it is essential that the automaton does accept constructs that are semantically incorrect. As a matter of fact, during development of the grammar, we tried to detect invalid declarators like "functions returning functions" (§8.3.5:6) syntactically. This proved to yield unexpected results when declarator was tried to be parsed in disambiguation: Instead of accepting the semantically ill-formed declarator, the specialized parser failed and the next alternative was tried. If it succeeded, we could commit to a wrong parsing tree and possibly accept an ill-formed input, for that matter. It is important to understand that once we disambiguate the input, there is generally no way to tell whether the decision was correct.

The example in Listing 3.8 illustrates the problem. Our grammar only allowed one application of function declarator operator ("`(` *parameter-declaration-clause* `)`")[21]. Any subsequent ones were simply not considered a part of the declarator in question. On namespace and function scopes, this resulted in the second application of the operator to be interpreted as an initializer (1). In type-id versus expression disambiguation, the type-id alternative would be dropped

[20]See 3.5.3 on page 30 for a description of this technique.

[21]Still, not all ill-formed declarators declaring functions that return functions could be recognized by this approach.

Listing 3.8: Semantic errors in declarators must not be detected by the parser.

```
void f()( C() );    // (1) (C()) is another function declarator operator, not an initializer
void g() {
  sizeof( C()() ); // (2) C()() is a type-id (function returning function returning C),
                   // not an expression (function call on a temporary of type C)
}
```

Listing 3.9: Checking for a superset construct must be carried out immediately.

```
int i;
int f( int (::i) ); // equivalent to "int f = i;", unlike g below
int g( int (i) );   // equivalent to "int g( int i );"
```

in favor of the expression (2). Both proceedings are wrong (§8.2:1,2). Please note that the example code is not semantically correct; it merely displays syntactic difficulties. Nevertheless, it is possible to craft a source where the incorrect interpretation would pass the remaining semantic rules – the compiler would accept ill-formed input.

Therefore, declarator part of the grammar in the final parser does not differ from the one in the Standard. Illegal declarators are detected by semantic analysis, rather than by syntactic rules.

3.7.3 Checking the Parse Tree for Superset Constructs

Conversely, additional semantic checks have to be performed because of the changes we have made to the grammar. To solve a shift-reduce conflict, the grammar was modified to accept parameter-declarations (§8.3.5:1) with their declarator-id globally qualified (using operator `::`), whereas it can only be a plain identifier according to the Standard (§8.3:1). The check is simple, again: See if it is qualified and if it is, the declarator is ill-formed.

However, the actual check needs to be performed immediately, because if we are in a disambiguation, the current alternative must be dismissed. Leaving out the check, and relying on the fact that the semantic analysis will carry it out later, is unacceptable, because the disambiguation might commit to an incorrect alternative.

Listing 3.9 shows such possibility. After reading `f`, we have to disambiguate the parenthesized construct, as it can be a parameter-declaration-clause, or an initializer. When trying the former alternative, the semantic check must force the specialized parser to fail when it finds out that the declarator-id of the parameter-declaration is qualified. Postponing the check – and accepting the ill-formed parameter-declaration – would result in committing to the parameter-declaration--clause alternative (`f` would be of the same type as `g`). The correct alternative is the latter one, `f` has a parenthesized initializer, which comprises of a function-style cast expression (`i` cast to `int`).

Chapter 4

Conclusion

We have successfully implemented a working analyzer that correctly parses the whole C++ language, as defined by the 2003 standard [5]. It uses a technique that runs out-of-order parsers to resolve ambiguities of the language, as well as constructs that need unlimited lookahead to be interpreted properly. All the individual ambiguities have their resolutions implemented. Type information is provided to the parsers by hinter, a filter that looks up identifiers in symbol tables, just before they enter the automatons. Compared to other C++ parsers, we have kept the grammar very close to the one in the Standard. An unmodified version of GNU Bison (1.875d) is used to generate our parser.

The hinter must be tightly knit to both the parsers, and semantic analysis. The "pipeline" between the hinter, the parsers, and the semantic analysis is kept shallow: at most one token is kept cached – in the lookahead of the LALR(1) automaton. This allows the hinter to provide accurate type information. For debugging and testing purposes, there is means to embed this information (and other data) into a source file. This way, we were able to test the whole analyzer, even on constructs that are were not yet supported by the semantic analysis.

The fact that the parsers are machine-generated allows for much better maintainability of the resulting code, compared to hand-written parsers. For example, the hand-written recursive--descent parser in GNU Compiler Collection (GCC, [3]) is nearly 500KB big, while our Bison grammar file reaches just over a quarter of that size, approximately 125KB. It is also very useful that our grammar is in Backus-Naur Form and only slightly differs from the original one in the Standard, which makes it rather easy to read and/or modify.

There are a few areas that are open to future enhancements, though. One such area is error recovery, a typical challenge for parsers in general. Given that it was not taken into consideration when designing our parser, we currently provide very minimal alternative. When errors are detected in all the specialized parsers that are launched to resolve an ambiguity (or a shift-reduce conflict), it is often simple to skip the erroneous input and continue the analysis thereafter. This usually involves finding the closing parenthesis, or beginning of the next statement. Nonetheless, a concept of coping with errors – that do not occur at places where disambiguation is started – has to be invented.

Performance of the analyzer can be improved as well. In some cases, the overhead of running specialized parsers can be completely eliminated just by looking at the next token. For example, when a statement starts with the `class` keyword, it cannot be an expression and must be a declaration-statement. There is no need to launch the subparser, because it tries to analyze the whole statement. Instead, we can make the decision immediately and parse for real right away. The analyzer can be optimized by finding these places and adding appropriate actions by hand.

Appendix A

Ambiguities and their Resolutions

In this appendix, we provide a list of conflicts that our analyzer solves using disambiguation. Most of them are results of ambiguities of the languages.

For each case we provide a short text explaining the conflict, the strategy used by our analyzer, reference(s) to the Standard, and an example with a list of possible interpretations.

In the examples, `C`, `D`, and `E` are class names; `T`, `U`, and `V` are class template names; `a`, `b`, `c`, and `x`, `y`, `z` are non-type names. Triangle that points to the right (▷) designates the position in the input, where the disambiguation is launched. In cases where an ambiguity occurs, the ambiguous input starts at the same position and ends before the position marked by a triangle that points to left (◁).

A.1 Semantic Resolution

To resolve conflicts from this group, semantic information is examined. See page 30.

Unnamed Bit-Field vs. Inner Class

This conflict arises from the fact that unnamed bit-fields are syntactically allowed to have any declaration-specifier-sequence as their type. Including elaborated type specifiers, though those prefixed by the `class` or `struct` keyword are prohibited by semantic constraints, as a bit-field can only be of integral or enumeration type. This conflicts with definitions of inner classes that have a base specifier, as shown in Listing A.1.

The resolution is simple – the type denoted by the declaration-specifier-sequence is examined and if it is suitable for a bit-field, a pad token is inserted.

References: §9.6:1, §9.6:3

Listing A.1: Unnamed bit-field versus inner class declaration.

```
class K {
  unsigned:3;    // well-formed unnamed bit-field
  class C :3;    // ill-formed; bit-field can only be of integral or enumeration type
  class C : public A { /* ... */
```

Operator Template

Similarly to name of an ordinary function template, operator-function-id can contain a template argument list enclosed after its name. However, operator names are not categorized by the hinter, so a shift-reduce conflict arises, similar to the one described on page 14.

To resolve this conflict, a lookup must be performed to find out whether the operator function is a template and when it is, a pad token is inserted.

Example: `operator == ▷ <`

References: §13.5:1, §14.2:3.

A.2 Syntactic Resolution

To find resolutions for these conflicts, trial parsers are run in the specified order. See page 22.

Expression-Statement vs. Declaration-Statement vs. Labeled-Statement

Occurs at function scope. To recognize labels that contain an identifier, lookahead of two tokens would be needed; they are not ambiguous with the other constructs. When parsing a labeled--statement, only the label part is parsed, as the remaining part needs to be disambiguated.

Order: labeled-statement, declaration-statement, expression-statement

References: §6.8:1

Example: `▷T(a) = 4;◁`

Syntactically possible interpretations:

- declaration of object `a` of type `T` initialized to `4`
- assignment-expression; functional-style cast of `a` to `T` on the left side, `4` on the right side

Note: This disambiguation is launched with empty lookahead, before and after any statement. It checks for `break`, `catch`, `continue`, `do`, `else`, `for`, `goto`, `if`, `return`, `switch`, `try`, `while`, `;`, `{`, and `}` in the lookahead token (that is read by the disambiguation, not by the automaton). If one of these is found, nothing happens, as no ambiguity is possible. Otherwise, the alternatives enumerated above are tried. Allowing the parser to read lookahead and decide whether to launch the disambiguation or not would be possible. However, the bogus rule trick (described on page 25) would have to be utilized. The problem is that there are more tokens that trigger the disambiguation, than those which do not. We decided to run the disambiguation at all times and check for the latter explicitly.

For-Init-Statement

Similar to previous ambiguity.

Order: simple-declaration, expression-statement

Example: `for (▷T(a) = 4;◁`

Condition

Similar to previous ambiguity.

Order: declaration, expression

Example: `if (▷T(a) = 4◁)`

Type-Id vs. Expression

At various contexts a type-id or an expression is expected. These contexts include explicit type conversion (cast notation), operands of the `sizeof` and `typeid` operators, template argument, and new-expression.

Order: type-id, expression

References: §8.2:2

Example: `sizeof( ▷C()◁ )`

Syntactically possible interpretations:

- "pointer to function taking no arguments and returning `C`" type
- functional-style cast expression, `C` is constructed

Type vs. Non-Type Template Parameter

The ambiguity arises from the fact that the `class` keyword is used to denote a type template parameter. It could also be a part of an elaborated-type-specifier of a non-type template parameter.

Order: type template parameter, non-type template parameter

References: §14.1:3

Example: `template< ▷class C◁ > class X;`

Syntactically possible interpretations:

- type parameter with name `C`
- unnamed non-type parameter of type `class C`

Parameter Declaration Clause vs. Initializer

In namespace and function scope, it is possible to initialize an object using expression list in parentheses syntax. This conflicts with the function declarator operator. Also see section 3.5.2 on page 22, it uses this disambiguation in the examples.

Order: parameter-declaration-clause, initializer

References: §8.2:1

Example: `int a▷( int (b) )◁;`

Syntactically possible interpretations:

- parameter-declaration-clause with one parameter-declaration, declaring an `int` parameter named `b`
- initializer with one expression, functional style cast from `b` to `int`

Parameter Declaration Clause vs. Parameter Declarator

Whenever there is a left parenthesis in parameter-declaration, but a declarator-id has not been read yet, the upcoming input can either be a parameter-declaration-clause, or a declarator. If it is a non-abstract declarator, an ambiguity arises.

Order: parameter-declaration-clause, parameter-declarator (where parameter-declarator covers both declarator, and abstract-declarator)

References: §8.2.7

Example: `int a( ▷int (C)◁ );`

Syntactically possible interpretations (with the assumption that `a` is followed by a parameter--declaration-clause):

- declaration of unnamed parameter of type “function taking `C` and returning `int`
- declaration of parameter of type `int`, with redundant parentheses around its name `C`

A.3 Combined Resolution

To resolve this conflict, semantic information is first examined and a trial parser is possibly run.

Constructor vs. Other Member Declaration

In class scope, the decl-specifier-seq has to be semantically examined to see whether it names the currently defined class. If it does, the upcoming input is tried to be parsed as parameter--declaration-clause. If it parses, the disambiguation commits a pad token that tags the declaration as constructor declaration. In all the other cases, nothing is done. The declaration will declare a different member (data member, member function, . . .).

Listing A.2: Constructors disambiguation example.

```
class K {
  K▷(A);      // (1) constructor that takes A
  K▷(x(A));   // (2) member function x taking A, returning K
  B▷(C);      // (3) data member C of type B
};
```

On line (1) `K` refers to the currently defined class and `(A)` can be interpreted as parameter--declaration-clause. Disambiguation commits a pad token which results in the line being parsed as constructor. While `K` on line (2) refers to the current class as well, `(x(A))` cannot be interpreted as parameter-declaration-clause, so the disambiguation does not commit any pad tokens. The line declares function `x` that takes `A` as its only argument and returns `X`. On line (3) `B` does not refer the currently defined `K` class, so this disambiguation exits without running a trial parser. Later, `(C)` is interpreted as a declarator in redundant parentheses. The line declares class member `C` of type `B`.

List of Figures

List of Listings

References

[1] Birkett A.: Parsing C++. `http://www.nobugs.org/developer/parsingcpp/`

[2] Free Software Foundation: GNU Bison.
`http://www.gnu.org/software/bison/bison.html`

[3] Free Software Foundation: GNU Compiler Collection. `http://gcc.gnu.org/`

[4] Grune D., Jacobs C. J. H. (1990): Parsing Techniques – A Practical Guide. Ellis Horwood, Chichester. `http://www.cs.vu.nl/~dick/PTAPG.html`

[5] International Organization for Standardization, International Electrotechnical Commission (2003): Programming languages — C++, ISO/IEC 14882:2003(E).

[6] Johnstone A., Scott E., Economopoulos G. (2004): The Grammar Tool Box: a case study comparing GLR parsing algorithms (LDTA '04 Preliminary Version). University of London.
`http://www.cs.lth.se/Research/LDTA2004/d07_Johnstone.pdf`

[7] Lestes Team (2005): Lestes C++ Compiler. `http://lestes.jikos.cz/`

[8] McPeak S. (2002): Elkhound: A Fast, Practical GLR Parser Generator.
Report No. UCB/CSD-2-1214, University of California.
`http://www.cs.berkeley.edu/~smcpeak/elkhound/`

[9] Parr T. et al.: ANTLR, ANother Tool for Language Recognition. `http://antlr.org/`

[10] Paxson V.: Flex – fast lexical analyzer generator. `http://lex.sourceforge.net/`

[11] Silicon Graphics, Inc.: Standard Template Library Programmer's Guide.
`http://www.sgi.com/tech/stl/`

[12] Scott E., Johnstone A., Hussain S. S. (2000): Tomita-Style Generalised LR Parsers. University of London.
`http://www.cs.rhul.ac.uk/research/languages/publications/tomita_style_1.ps`

[13] Willink E. D. (2001), Meta-Compilation for C++, University of Surrey.
`http://www.computing.surrey.ac.uk/research/dsrg/fog/FogThesis.html`