HSR – University of Applied Sciences Rapperswil

Institute for Software

Master Thesis

C3P0
C-Plus-Plus-Parser-for-C++0x

Thomas Corbat
tcorbat@hsr.ch

http://c3p0.ifs.hsr.ch

Supervised by Prof. Peter Sommerlad

July 16, 2010

Abstract

For some time the C++ Standards Committee is working at a new standard for C++. Originally the release was planned to be in 2009, therefore the standard received the unofficial name C++0x. To fully profit from the new possibilities and features, a programmer needs his tools to support them. We at the Institute for Software, as participant of the Eclipse C++ Development Tooling (CDT) project, are eager to improve this integrated development environment (IDE) continuously, especially with our focus on automated refactoring support. Such tools rely on source code representation in an abstract form, usually represented through an abstract syntax tree (AST). In this master thesis we continue to build this cornerstone to support C++0x, a parser for constructing that AST. The results are outlined in this report.

Contents

Developing software without unit tests is like solving a Rubik's Cube in the dark.

1. Introduction

The C++ Standards Committee is going to release a new standard for C++ [WG2], providing new features and extending the current syntax. It is unofficially called C++0x, for a release in 2009 (C++09). Actually the official ISO release has been delayed, as the standardization takes time – more than originally expected. Nevertheless, there will be a new standard with features completely new to C++.
It is expected that the set of features will not be changed after March 2010. The final draft should be finished by the end of 2010. Until the ISO C++ standard can be released it will take additional time. It is expected to be eventually published by the end of 2011.

1.1. Vision

As a participant in the Eclipse C++ Development Tooling (CDT) project we at the IFS, Institute for Software of the University of Applied Sciences Rapperswil, are endeavoring to keep up with the evolution of C++ [GZS07] and want CDT to be able to cope with the new standard as soon as it is released. We want to allow C++ programmers using CDT to have an integrated development environment (IDE) fully supporting the new C++ features. The C3P0 term project had been the first step to achieve this endeavor.

1.2. Focus

Our contribution to CDT is focused on automated refactorings. While this is a challenging task by itself, especially for C++, this work highly depends on core components of an IDE. All of our refactoring plug-ins are based on an abstract syntax tree (AST) for structure analysis and representation. Therefore, we need to have an AST for the new C++ standard as well, if we want continue to provide useful refactoring tools.

1.3. Project Goal

In the preceding term project we have started to develop C3P0, a parser with an understandable grammar, from scratch. Its eventual purpose is to recognize C++0x code, which it constructs a CDT AST for. As the results of the term project are considered a success, this master thesis continues the work on C3P0.
Eventually, having an AST for representing C++0x programs enables us to port our automated refactorings or even implement new ones for the new C++ standard. It also allows others to adapt or create new AST-based IDE features. Our intention is to reuse as much as possible of the existing structure to alleviate adaption of current features to

the new AST. Recognizing a superset of C++0x is our primary target, as we can narrow the recognition capabilities later.
C3P0 is implemented in Java and and an ANTLR v3 grammar, to have an understandable and maintainable set of rules.

1.4. About This Document

In this report we describe the goals, tasks, implementation details, solutions to encountered problems and the results of the C3P0 master thesis. We have focused our explanations on descriptions supporting students doing follow-up projects.
Chapter 2 summarizes the milestones of the C3P0 project, names the goals of this master thesis and recapitulates the results of the term project. Chapter 3 explains the first milestone, the implementation of the symbol table, its integration into the parser and shows the feasibility of our symbol handling approach. Chapter 4 summarizes the features of a C++ preprocessor and documents the implementation of our preprocessor, C2P0 - C++-Preprocessor-for-C++0x. In Chapter 5 we have described the results of the last milestone, the construction of a CDT-AST in C3P0. Chapter 6 describes further improvements to C3P0, not directly belonging to one of the milestones. And Chapter 7 concludes, summarizing the master thesis result.

1.5. Acknowledgments

Many thanks to my supervisor Peter Sommerlad, who came up with the idea of this project and, as a member of the Standards Committee, was a very reliable source for information about changes and recent activities. Furthermore, he always had ingenious ideas how to challenge C3P0's recognition capabilities providing extraordinary, but correct, C++ code samples.
Thanks also to my co-supervisor Emanuel Graf, who was a good help in CDT related questions, especially in concerns of legal aspects. Furthermore, thanks to Michael Klenk for interesting discussions about ANTLR features and his help when figuring out how they are used.
Thanks to my fellow students Lukas Felber and Mirko Stocker who both always had an open ear when a problem had to be discussed for letting the solution appear. Special thanks to Lukas for proofreading this document.
Most of all, thanks to my partner, Regina. She supported me, not only during the master thesis, but all my studies. And she accepted insightfully the efforts for achieving my goals, which took very much time.

2. Thesis Scope

In [Cor10a] we have outlined and described the major milestones of the C3P0 project:

Project Setup Setup of the whole environment and investigation on existing C++(0x) grammars, possibly in ANTLR (v3).

Lexer Implementation Implementing the rules for lexical analysis and creating a token stream.

Recognizer Implementation Derive the parser rule set from the C++ standard draft [Dra09] for syntactic analysis. Augment these rules with semantic checks.

AST Generation Extend the parser with actions for building an AST, which serves as a structural basis for automated refactorings.

CDT Integration As a final step we want to integrate the parser, probably as a plug-in, into CDT. There it should be used for parsing C++0x code.

During the term project we have successfully achieved the first two milestones. In this master thesis we can use the existing infrastructure of the term project and do not need to setup the whole environment again. Therefore, detailed explanations of the build process are documented in [Cor10a]. Lexical analysis is implemented in the lexer and works fine; no major deficiencies are known. In the preceding term project we have implemented the basic rule set for a parser syntactically recognizing C++0x. The results are described in [Cor10a] as well. Nevertheless, the recognizer had not been completely implemented by the end of the term project. We had to halt at the end of the semester due to time restrictions.
During this master thesis we continue the work at the parser of the term project. There are three main goals planned to achieve:

- Implementation of a symbol table
- Handling of preprocessor statements
- Extend the parser to generate CDT-like ASTs

A parser, and its rule set, heavily depend on the symbol table. Some circumstances could not have been foreseen during the term project, thus, the existing grammar will necessarily be changed and adapted too. Furthermore, we want to improve the existing grammar regarding structure, readability and performance, if possible. If there is time, we will start with the integration into CDT.

2.1. Term Project Summary

With this master thesis we continue our work from the point where we stopped at the end of the term project.
As mentioned above, the build environment can be taken as is and the lexer has also been implemented successfully. Unfortunately, the recognizer has not been completed yet. We have implemented a complex rule set for C++0x, which syntactically is quite reliable. Nevertheless, it is not sufficient to have a grammar that is able to recognize C++ on a syntactical level only. There exist lots of ambiguities, especially in the construction rules from the C++ standard [Dra09], which describes a superset for C++. These ambiguities are resolved through restrictions an specifications in the textual description of the corresponding program elements. Some of them are specifications how a certain construct has to look like. Others are restrictions depending on the context.
Our term project parser implementation has already implemented some semantic checks. But these are rather simple and cannot satisfy the requirements of the C++ standard. We have coped with the issue of symbol resolution in a quite simple way, by maintaining a monolithic symbol table to track classes, templates and other symbols. This was necessary to create tests for the context sensitive rules and worked fine, until we came across examples where symbols had been redefined in the monolithic environment. Symbols that actually should have been hidden in the corresponding context got visible and let the recognition of certain constructs fail. Therefore, we primarily need to implement a better way of tracking symbols, or types respectively. This is the first part of this master thesis.
Additionally, in the term project we skipped handling of all preprocessing directives. For implementing the basic rule set we did not need to handle them, as we could use CPP of GCC [Pro] to do all preprocessing and subsequently had to deal with one single file containing C++0x code, free from all preprocessing directives. With regard to the AST, this approach will not be sufficient for C3P0, as we will need to have exact file and position information to perform automated refactorings. Subsequently, we will find a way to handle preprocessing directives.
Thanks to a large test suite (about 725 test cases), which ensures the recognition power of C3P0, modifications can easily be verified. If they break the existing capabilities we notice this immediately and have good indications about the reason of the problem.

3. Symbol Tracking

Knowledge about declarations of symbols is essential for several decisions in the parser. For example, consider the following piece of C++ source code:

```
...
X(a);
...
```

Listing 3.1: Ambiguous Code

Without further information about its context, the statement above is ambiguous. For example, it could be an expression – a call of function `X` – or a declaration – variable `a` of type `X`. The resolution depends on `X`. If it is a type, we have a declaration, or if it is a function, we have an expression.
In the term project we have already implemented the checks for types in the parser. But, we did not have a sophisticated tracking mechanism to handle scope nesting and type declarations. We used a workaround by maintaining a monolithic symbol table only, which is not sufficient and has to be replaced with an appropriate structure. Tracking type symbols should be sufficient for making decisions in our parser.
In this chapter we will describe the intention of the symbol table, our implementation and decisions. Furthermore, we will look at name resolution in the C3P0 parser, explain the integration of the symbol table, elaborate on the requirements and outline the deficiencies.

3.1. General Approach

Terence Parr, in his book [Par09b], describes how to perform symbol tracking in programming languages containing classes. He proposes a two pass approach on an existing AST, representing the structure of the program. After generating the AST, in the first pass, all declarations are matched and they are added as known symbols. In the second pass all references to these symbols are resolved. These walks through the AST are implemented using a tree grammar in ANTLR.
While this way to handle symbol tracking looks promising at a glance, we have decided not to follow this approach for the following reasons: First, our implementation in the term project already relied on a different approach. We are expecting to be able to resolve at least the type symbols, which we necessarily need to know, while parsing. Therefore, we cannot first generate an AST for a late resolution. Changing the general

approach would require a complete overhaul of the grammar, which would let us end up starting from almost zero again. Furthermore, we are not sure whether this approach would even be feasible for a language of C++'s complexity. The example language of the book, *Cymbol*, is rather simple compared to C++. Subsequently, we will stick with our approach of tracking symbols at parse-time and try to parse the C++0x code in one pass.

3.2. Requirements

Since we intend to eventually provide a CDT-like AST for C++0x, we are not required to perform static analysis as thorough as a compiler would. Our main goal is to generate an AST for automated refactorings. Therefore, we need our lookup primarily for disambiguation of certain statements, like the example in Listing 3.1. We can accept a parser recognizing a superset of C++0x, enabling to create an AST even for not completely correct C++ code. For our purpose, creating an AST as a basis for refactoring, this is much more forgivable than not recognizing valid code. Subsequently, our checks do not need to be that pedantic. Nevertheless, we will keep the possibility for more parse-time checks in mind, as the infrastructure provided by C3P0 might be reused for other tools relying on an abstract representation of C++ code.

3.2.1. Terminology

The C++ Standard [Dra10a] is quite particular about the definition of the terms used. We have listed the most important, regarding the symbol table, below (from the parts [basic] and [basic.def]):

Entity An entity is one of the following: value, object, reference, function, enumerator, type, class member, template, template specialization, namespace, parameter pack or `this` ([basic], paragraph 3).

Name A name denotes the use of an identifier, operator-function-id, literal-operator-id, conversion-function-id or template-id that refers to an entity or label ([basic], paragraph 4). Every name that denotes an entity is introduced by a declaration ([basic], paragraph 5).

Variable Declarations of references and objects, except non-static data member references, introduce variables ([basic], paragraph 6).

Declaration New names are introduced into the translation unit by declarations, which specify their interpretation ([basic.def], paragraph 1). The same name can be declared more than once, as long as the meaning of the name is not changed.

Definition Every declaration is a definition except if, the declaration... ([basic.def], paragraph 2)

- is a function without a function body.

- contains the `extern` specifier or a linkage-specification and neither an initializer nor a function-body.
- is a static data member in a class definition.
- is a class name declaration.
- is an opaque-enum-declaration.
- is a `typedef` declaration.
- is a using-declaration or using-directive

Furthermore, while there can be several declarations, declaring the same name, there can only be one definition for variables, functions, class types, enumeration types and templates. This is also known as the *one definition rule* ([basic.def.odr], paragraph 1).

Remark: Even though we, in C3P0, currently only identify functions and templates by their name, the declaration actually also contained their parameters. Therefore, overloaded functions and templates do not break the one definition rule.

In C3P0 we focus on the resolution of declarations and do not enforce the one definition rule. We primarily need to track whether a name is known or not. Consequently, we might consider statements that contain references to symbols which have been declared, but lack a definition, correct. More pedantic checks could still be implemented in following projects.

3.2.2. Scoping

Scope handing in C++ is quite complex. There are several kinds of program elements which can represent a scope or contain one. Basically, we encounter the following, described with their remarkable features [Dra10a], in Section [basic.scope]:

Namespaces There is a global namespace, which is ubiquitous in the whole translation unit. The global namespace exists without a special declaration. Furthermore, there can be other named or anonymous namespaces, which each has a local scope, that can contain any kind of declaration. Namespaces can be reopened and extended. A namespace can also contain other namespaces. It is possible to make a whole namespace visible in the scope of another namespace with the `using` directive.

Classes Class definitions have a local scope, which can be visible outside the class, in member-definitions. Classes can consist of an inheritance structure, that allows access to members of the base classes. Furthermore, classes provide one lookup feature particularly difficult to handle, as bodies of member function definitions and braced initializer lists can reference symbols declared anywhere in the class body, without forward declaration. This requires forward-lookup, which will be a topic of its own. Classes can contain declarations of other classes, members and templates.

Enumerations An enumeration contains declarations of enumerators. There are two distinct types of enumerations: Scoped and unscoped. Scoped enumerations keep the enumerators in their local scope. Enumerators of unscoped enumerations are introduced into the scope surrounding the enumeration.

Functions A function definition is a rather simple element containing a scope. All declarations inside, including the function parameters, are local and referenced symbols have to be declared before their use – except for referencing in class member functions. Functions can contain declarations of local classes and local scopes.

Local Scopes Local scopes are usually nested in other scopes, like a function body, and can contain and hide symbol declarations in outer scopes. They are rather simple to handle and more or less straightforward in concerns of resolution.

Templates Templates introduce template parameters, which are visible to the templated element but not beyond. The template declaration itself acts as the corresponding entity, a class or function, regarding scoping and resolution.

Except for enumerations all scopes above can contain variable declarations. We did not list them separately as they are not scopes themselves. Most challenging will be the distinction of the individual lookup rules, which are location and context dependent. For example, resolution in a class scope checks the surrounding scope and the scopes of the base classes, but does not check the scopes surrounding the base classes.

3.2.3. Hiding

Generally, it is not allowed to define two symbols with the same name in one scope. There are exceptions, though, which allow defining the same name several times. For example, function or template declarations with distinct parameter lists can define the same name more than once. This is known as overloading. Names inherited from surrounding scopes can be redefined locally. Such declarations hide the symbols from outer scopes. It is allowed to hide a class or enumeration name in the same scope with the declaration of a variable, an enumerator, a function or a function template. Consider the following:

```
void foo(){
  struct B {
    int b;
  };
  int B = 1;
  // Now B refers to the integer variable
}
```

Listing 3.2: Hiding a Locally Defined Class

In the example above, the declaration of the variable `B` hides the declaration of the struct `B`. It is still possible to access the struct `B` using the scope operator (`::B`). But, it is not allowed to declare a further struct named `B`.
According to the standard it shall not be possible to hide template parameters, declarations in control parts of iteration statements, like `for` loops and function parameters. We will avoid some of these obligations and keep our implementation closer to real implementations like GCC [Pro], which itself does not restrict all of them.

3.2.4. Unqualified Lookup

In this section we look at unqualified name lookup, illustrated by some examples. The source code and the lookup scenarios are taken from the C++ standard draft [Dra10a]. Basically, names shall be declared before they are used the first time – the only exception occurs in member function bodies of classes and braced initializer lists.

Names in Function Bodies

Names in function bodies shall be defined in the same or an enclosing block or namespace before their use. The following example is from the C++ standard draft [Dra10a], [basic.lookup.unqual] paragraph 6.

```
namespace A {
  namespace N {
    void f();
  }
}

void A::N::f(){
  i = 5;
}
```

Listing 3.3: Unqualified Lookup in Function Scope

The name `i`, in the function body of `f` has four possible locations, where it could be defined:

- In the function body of `f`
- In the namespace `N`
- In the namespace `A`
- In the global namespace

In all four cases the declaration of `i` must be before the statement `i = 5;`. The actual type of the symbol does not matter, the variable is only one example here. This would also hold for resolution of classes, templates, etc. Figure 3.1 illustrates how an example scope tree could look like.

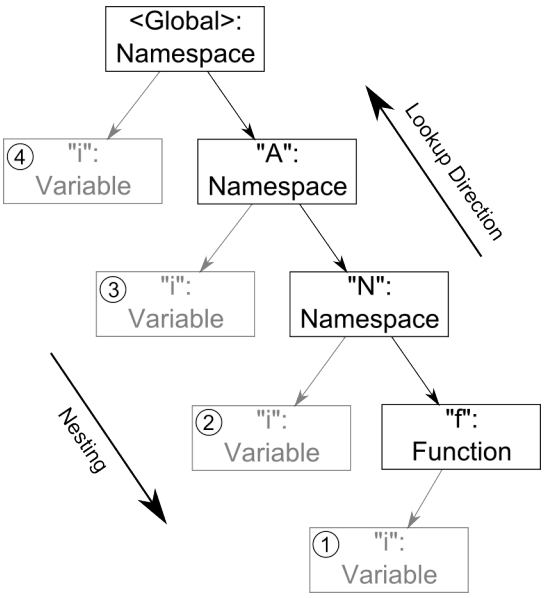

Figure 3.1.: Symbol Lookup in a Function Body

Nesting of namespaces, or scopes respectively, hide previous declarations. Therefore, `i` could have been defined in all locations mentioned above at the same time. Only the closest declaration would have been considered for the resolution.

```
// (4) int i;
namespace A {
  // (3) int i; - hides declaration (4)
  namespace N {
    // (2) int i; - hides declarations (3) and (4)
    void f();
  }
}

void A::N::f(){
  // (1) int i; - hides declarations (2), (3) and (4)
  i = 5;
}
```

Listing 3.4: Possible Declarations for Access in Function Scope

Names in Class Definitions

Names in class definitions, except names in member function bodies and braced initializer lists, resolve as follows:

- Declarations in the same class

- Member declarations of base classes
- Member declarations of enclosing classes
- Declarations in surrounding blocks of the class
- Declarations in surrounding namespaces

The declaration has to be before the use of the symbol. Below, we have another example illustrating a case where a symbol is referenced, for an array specification, in the scope of a class body. While the following example (from [basic.lookup.unqual] paragraph 7) references a variable or constant integer value, the resolution would be the same for other entities.

```
namespace M {
  class B {};
}

namespace N {
  class Y : public M::B {
    class X {
      int a[i];
    };
  };
}
```

Listing 3.5: Lookup in Class Bodies

The name `i` in the class body of `X` has five locations where it could be defined:

- In the class body of `X`
- In the class body of the enclosing class `Y`
- In the class body of the base class of `Y`, class `M::B`
- In the enclosing namespace `N`
- In the global namespace

Except for the lookup in `M::B`, which would be defined before the location anyway, the definition must be before the use of `i`. Figure 3.2 shows the corresponding scope tree and the possible locations of the declaration.
We emphasize that the scope provided through namespace `M`, which is surrounding class `B`, does not serve as a possible scope for the declaration of `i`. While the resolution process follows base classes and surrounding namespaces it does not follow namespaces surrounding the base classes.
We would obviously need an association beside the parent structure in our scope tree implementation, from the deriving class `Y` to its base class `B`.

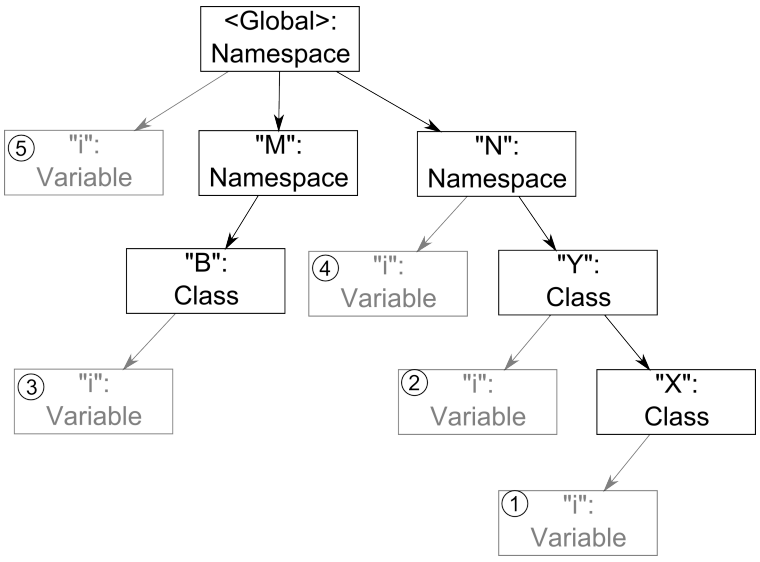

Figure 3.2.: Symbol Lookup in a Class Body

Member Function Definitions

Let us come to an example implying more difficulties, especially regarding the implementation. Member functions allow so called forward referencing, references to declarations that have not been passed in a class so far. Therefore, we have a slightly different lookup compared to names in a class definition body. The resolution of symbols in a member function body, as well as names in braced initializer lists of non-static data members, must be:

- In the same or an enclosing block, before it is referenced
- A member of the corresponding class or one of its base classes – this can imply forward lookup
- If the corresponding class is a nested class: It must be a member of the enclosing class or one of their base classes (this lookup is recursive considering nesting) – this can imply forward lookup as well
- If the corresponding class is a local class: Before the definition of the class in the surrounding block or in the surrounding class
- In a surrounding namespace, before it is referenced
- In the global namespace, before it is referenced

The following example is from the C++ standard draft [Dra10a], [basic.lookup.unqual] paragraph 8.

```
class B {};
namespace M {
  namespace N {
    class X : public B {
      void f();
    };
  }
}

void M::N::X::f() {
  i = 16;
}
```

Listing 3.6: Lookup in Member Definitions

The variable `i` could be declared at the following locations:

- In the function body of `f`, before it is referenced
- In the **whole** class `X`, either before or after the declaration of `f`
- In the class `B`
- In the namespace `N`, before it is referenced
- In the namespace `M`, before it is referenced
- In the global scope, before it is referenced

The emphasis at the second bullet might sound redundant in this case, as before and after the declaration of `f` is before the use of `i` in any case. But, the behavior does not change when defining the member function `f` inside the class definition. Referencing the field `i` before its declaration is still valid. Figure 3.3 shows an example scope tree for an inline definition of `f`. The possible variable declarations for `i` marked in green are only possible if the use of `i` is after the declaration of `f`; so to speak if the definition of `f` is outside the class, as in the code above. Nevertheless, the declaration of `i` can be after the declaration of `f`, in certain cases.

Lookup of Friends

Friends have a refined lookup behavior:

- Lookup for inline friend function definitions in the class behaves like lookup in member function definitions.
- Lookup for friend function definitions outside the class granting friendship behaves like lookup in a function defined in a namespace.

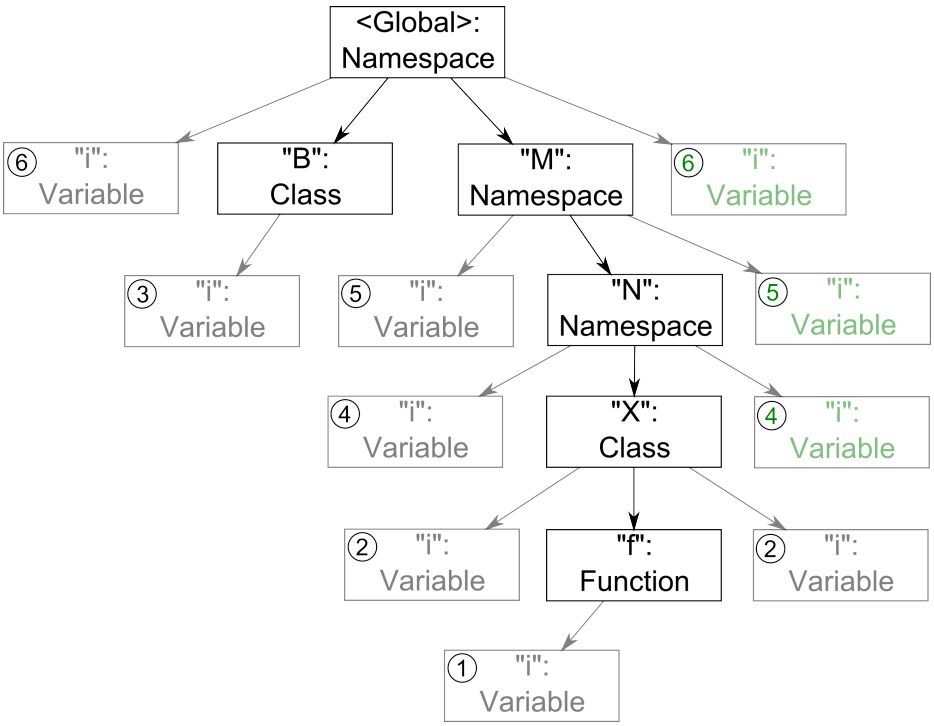

Figure 3.3.: Symbol Lookup in a Class Member

- Lookup for names in the declarator of a declared friend member function is first performed in the class, which the function is defined in (except if it is part of a template-argument in a template-id); if this lookup is not successful the name is looked up as if the function declaration was a member of the class granting friendship.

The following example is from the C++ standard draft [Dra10b], [basic.lookup.unqual] paragraph 10.

```
struct A {
  typedef int AT;
  void f1(AT);
  void f2(float);
};

struct B {
  typedef float BT;
  friend void A::f1(AT);
  friend void A::f2(BT);
};
```

Listing 3.7: Lookup in Friend Declarations

In the example, `AT` in the friend declaration of `f1` resolves as follows:

- `AT` is looked up in struct `A`
- As it is found, lookup is successful

In the example, `BT` in the friend declaration of `f2` resolves as follows:

- `BT` is looked up in struct `A`
- As it is not found, lookup continues at struct `B`
- There is a definition of `BT` in struct `B`, therefore, lookup is successful

We do not have any additional requirements to our scope tree through this procedure, since these lookup mechanisms are required anyway. The parser will need to know which lookups to perform.

Further Lookups

Hiding Default Arguments and Mem-Initializers Function parameter names hide the names of the surrounding blocks, as well as class and namespace scopes, in default arguments and mem-initializers of constructors. To cover this requirement we can either introduce a local scope for these rules or let the parser perform the lookup.

Hiding Enumerators In the definition of an enumeration, a defined enumerator is visible immediately after the definition and hides names declared in surrounding blocks, classes or namespaces. Here we could also introduce a local scope, which takes the declarations that will be preferred over the surrounding scopes.

Static Member Definitions Definitions of static data members outside of their class, perform lookup as if the definition was in the class. The parser will have to switch to the corresponding scope.

Namespace Member Definitions Similar to the definition of static members outside their class, if a variable of a namespace is defined outside its namespace, lookup in the definition happens as if it was inside the namespace. The following example is from the C++ standard draft [Dra10a], [basic.lookup.unqual] paragraph 14.

```
namespace N {
  int i = 4;
  extern int j;
}

int i = 2;

int N::j = i; //N::j := 4
```

Listing 3.8: Lookup in Qualified Variable Definitions

Function Try Blocks Names in handler of a function-try-block are looked up as if they were in the outermost block of the function definition. Parameter names should not be redeclared in the exception-declaration nor in the outermost block of a hander. Names declared in the try-block are not found in the handlers.

```
void foo(int i) try {
  int i = 2;
  throw 3;
}
catch(int &i) //Not allowed due to parameter declaration int i
{
  int i = 4; //Should not be allowed due to catch declaration int &i
}
```

Listing 3.9: Lookup in Function Try Blocks

Unnamed Namespaces

The examples above completely ignore the existence of unnamed namespaces. Due to the definition of an unnamed namespace, we expect, they are considered to belong to the surrounding namespace for lookup [Dra10a]([namespace.unnamed]):

```
inline? namespace <unique> { /* empty body */ }
using namespace <unique>;
namespace <unique> { namespace-body }
```

Listing 3.10: Interpretation of an Unnamed Namespace

`<unique>` is the same name for the three statements above, but does not occur anywhere else in the whole translation unit.

Summary

Our symbol table implementation will cover the standard as much as needed for satisfying our name lookup requirements. Special treatment of the forward lookup functionality is necessary, otherwise we cannot parse with a one-pass approach. To cope with this requirement we have some possibilities like forward lookup at the referenced position, deferred parsing of the corresponding sections or introduction of ambiguity results, which are resolved in a later step.
Lookup logic for unqualified names can mainly happen in one well known context. Thus, we do not expect the parser to be required to do lots of scope-crawling. Tracking the currently active context should suffice to perform the lookup in most cases.

3.2.5. Argument Dependent Lookup

Using an unqualified name for identification of a function, special lookup rules apply, due to so called argument dependent lookup. Let us start the explanation using an example, from the C++ standard draft [Dra10a], [basic.lookup.argdep] paragraph 1:

```
namespace N {
  struct S {};
  void f(S);
}

void g() {
  N::S s;
  f(s);
  (f)(s);
}
```

Listing 3.11: Argument Dependent Lookup

The declaration of function `f` is not accessible in the function `g`, at least not with an unqualified name. Nevertheless, `f(s)` is a semantically correct function call at that place. Due to the type `N::S` of argument `s` the function `f` can be resolved. On the other hand, the call to `(f)(s)` cannot be resolved as the parentheses around `f` prevent argument dependent lookup.
For argument dependent lookup the following rules are used, depending on argument type `T` – typedefs and using-declarations are not considered for this resolution:

- If `T` is a fundamental type (like `int`): No further lookup is required.
- If T is a class type: The class `T`, enclosing classes of `T`, base classes of `T` and namespaces which the classes before are members of. Template arguments, if `T` is a template-specialization, are also considered for lookup, as well as their associated classes and namespaces.
- If `T` is an enumeration type: The surrounding namespace and, if it is a class member, that class, will be considered for resolution.
- If `T` is a pointer to `U` or an array of `U`: Classes and namespaces associated with `U` are considered
- If `T` is a function type: Classes and namespaces associated with its arguments and its return type are considered.
- If `T` is a pointer to a member function: Classes and namespaces associated with its arguments and its return type as well as the class it is a member of are considered.
- If `T` is a pointer to a data member: Classes and namespaces associated with the class it is a member of, as well as the data member's type are considered.

These rules for argument dependent lookup are rather complex to implement and require lots of context information while lookup is performed. Probably, we will avoid them if it is possible to limit the symbol table to tracking types only, which might suffice for our purposes.

3.2.6. Qualified Lookup

While unqualified lookup is primarily performed upwards in the the scope tree, qualified lookup is a possibility to descend again into nested scopes. Using the scope operator (`::`) a qualified name consisting nested namespaces and scoped types can be denoted for local search.
There are several specific cases in the C++ standard defined for lookup. We will not elaborate on all of them in detail. We primarily stick with a resolution mechanism similar to the unqualified lookup, except that we can temporarily switch out of the currently active scope.
Most important for the whole lookup process, a property that is necessary for avoiding infinite loops, is the restriction to just visit every scope at most once. In the symbol table this requires some kind of tracking of the scopes already visited.
We have to decide whether the qualified lookup is mainly performed in the parser or the symbol table. Literally, we distinct whether the parser resolves the scope for lookup and lets the scope perform the resolution or if the parser collects information about the context and passes them to the symbol table for lookup.
Specific cases for lookup will be implemented when recognized as necessary, until that we perform lookup in the qualified scope provided by nested name specifiers. Eventually, we might decide to omit certain cases, as we accept going along with a superset of C++, that does not impair the AST generation capabilities.

3.2.7. Decltype Scopes

According to previous drafts of the C++0x standard the new keyword `decltype` has been introduced. Decltype could be used to omit typing complex type specifiers or even relieved the requirement to deduce the resulting type of an expression oneself. While this does come in handy for a programmer, it is quite difficult to deduce such a type at parse-time. It requires typing information and knowledge about conversions and operators available.
While, in previous revisions of the standard, `decltype` could only be used as a simple type specifier, in March additional uses for `decltype` have been voted into the standard [Van10b, Dra10a]. Theses changes contain the possibility to use `decltype` as a nested name specifier, in an unqualified id and destructor names. Before, the keyword `decltype`, followed by an expression in parentheses, was known to be a type for sure, without deducing the effective type. Now the resolution of the further nested name specifiers or names would require to know the exact type denoted by `decltype`. In the following example we illustrate the basic problem, not including any complex conversion or type deduction:

```
struct A {
  struct B {};
};

void foo() {
  A a;
  decltype(a)(x);      //Declares variable x of type A
  decltype(a)::B(y);   //Declares variable y of type A::B
}
```

Listing 3.12: New Application of `decltype`

According to the previous definition of `decltype` only the declaration of `x` would have been correct. This could be handled by simply assuming `decltype(a)` to be a known type, in the parser. The new definition also allows the declaration of `y`. But, the statement above that declares `y` could also be a function call, if B was a static member function of `A`. Subsequently, we have to be able to resolve the symbol `B` to disambiguate the statement. But this requires the capability to resolve the type of the expression used in `decltype` at parse-time. While this might be easy if the expression is just a single variable, like in our example `a`, it can become quite difficult for complex expressions. As long as we do not have the capability to resolve the type of such expressions, we cannot correctly determine whether a statement is a declaration or an expression if `decltype` is used as a nested name specifier.

3.3. Symbol Table

This section describes our implementation of the symbol table, for satisfying our needs to track symbols while parsing. The implementation of the symbol table happened hand in hand with its integration into the parser. Therefore, we have made some decisions based on the possibilities and restrictions of the parser implementation. Nevertheless, we have split up the documentation of the symbol table and the name resolution implementation into two distinct parts.

3.3.1. Requirements

The basic requirements to name resolution in our symbol table are taken from the C++ standard draft [Dra10b] and are summarized in the previous Section 3.2. We have several different types of symbols to be tracked:

- Namespaces
- Classes
- Structs
- Unions
- Enums
- Functions
- Variables
- Templates
- Typedefs

These symbols usually have a name, although, some of them can be anonymous. Although a symbol without a name cannot be referenced with identifiers, it needs to be tracked for proper resolution as a scope.
The first implementation, in the term project, tracked symbols in a monolithic scope, which is not sufficient at all. We need to have a structure of scopes for providing mechanisms like hiding and qualified name resolution. We have identified the symbols which are potential scope providers. They have already been described previously in Section 3.2.2.

Lookup

Considering unqualified lookup as the basic name resolution case, we can simply perform lookup in a single currently active scope. This also includes ascending in the scope structure in a very straight way. A symbol, which cannot be found locally, is resolved in the parent scopes, as described in Section 3.2.4.

```
namespace NS {
  typedef int TYPE;
  class K{
    TYPE i;
    //TYPE is resolved: in class K -> namespace NS -> global namespace
  };
}
```

Listing 3.13: Unqualified Lookup Example

We have also seen, that sometimes additional scopes are visible locally. For example, through derivation of classes or `using` directives, mapping namespaces into the local scope.

```
class L : public K {
  void foo(){
    using namespace NS;
    T t;
    //T is resolved: in class L -> class K -> namespace NS -> global namespace
  }
};
```

Listing 3.14: Example for Visible Non-Nested Scope

Qualified lookup includes descending in the scope tree, where a namespace or class scope can be entered on purpose. Such a qualifier hides the current scope for the resolution of the name, and resolves the succeeding identifier in the specified context locally. Therefore, it is possible to move resolution focus into a specific scope in the scope tree and ignore local identifiers with the same name in the current nesting level of the scope tree.

```
namespace NS1 {
  class K {};
}
namespace NS2 {
  class K {};
  NS1::K k; //NS1:: hides the current namespace NS2
}
```

Listing 3.15: Qualified Lookup Example

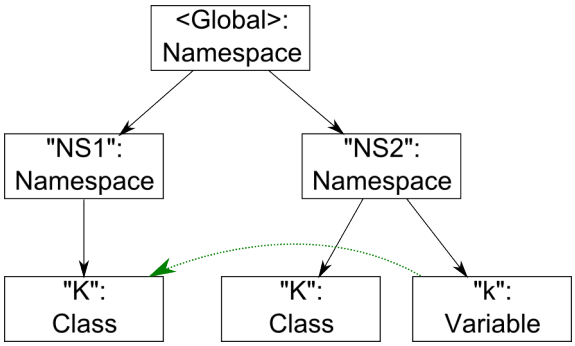

Figure 3.4.: Type of Variable k

Depending on the context, other scopes can be visible, but do not hide the surrounding scope completely. Symbols from the hidden scope, which are not declared in the hiding scope, are still visible. Nevertheless, if a symbol is defined in the hiding scope, it shadows symbols with the same name of the hidden scope.

```
struct S {
  typedef int I;
  void foo();
};
typedef float F;
void S::foo() {
  I i; //S:: is augmented in the local scope through the declarator S::foo
  F f; //F:: is resolved in the surrounding global namespace
}
```

Listing 3.16: Lookup with Augmented Scope

There are further situations where lookup is influenced. For example, the `using` declaration, which adds a qualified symbol to the local namespace, or the `using` directive, which maps a whole namespace into the scope of another namespace.

Symbol Types

For enabling our one-pass parse approach, we need to be able to track and switch contexts on the fly while parsing. Most important are operations for entering and leaving scopes as well as defining and resolving symbols. For avoiding to bloat the parser with resolution logic, we intent to implement as much as possible in the symbol table.
To represent the program elements we need several types of symbols. We distinct three core properties of those symbols:

Name Most of these symbols have a name for identification. There is one exception, the local scope, which never has a name and always is anonymous. Resolution requires a name, either for nested name specifiers to enter a scope or identifiers to name a specific symbol. Since local scopes cannot be accessed from outside, it is no problem not to have a specific name for them. On the other hand there exists the possibility to have anonymous classes or unscoped enums. As we need to retain the possibility to enter such scopes, we need to generate a name for later resolution.

Type Some symbols define a type, which can be used as a specifier for declarations. To know whether a certain symbol denotes a type is particularly important for performing several decisions in the parser. As mentioned above, there might be even types, which do not have a name, but specify a symbol anyway. Therefore, they have to be tracked as well.

Scope Most symbols somehow open a new scope. They can contain other symbols, which might be scopes themselves. From this the scope structure results and builds a tree structure of symbols. Symbols which do not represent a scope, are always leafs in this tree.

In Table 3.1 we have shown which symbols consist of which properties.

Symbol	Name	Type	Scope
Namespace	x		x
Class/Struct	x	x	x
Enum	x	x	x
Function	x		x
Variable	x		
Template	x	x	x
Typedef	x	x	x
Local Scope			x
Fundamental Type	x	x	

Table 3.1.: Symbol Properties

3.3.2. Implementation

The implementation of the symbol table is focused on the requirements of the parser and providing a comfortable interface to relief most of the resolution logic off the parser itself. This includes several actions for querying and defining symbols as well as switching scopes. We do not need to be able to determine every single symbol as exact as a compiler would. Our main goal is to find the correct rule in the grammar set for later defining the corresponding AST nodes, our eventual goal. Nevertheless, we will keep possible requirements for further semantic checks in mind, to enable later extensions.

Symbol Table Interface

The `C3P0Parser` primarily interacts with the symbol table though a set of well known operations. These operations are defined in the `IC3P0ParserContext` interface and can be categorized as follows:

- Symbol Definition
- Symbol Specification
- Symbol Query
- Scope Switch
- Scope Check

Symbol Definition These methods add new symbols to the symbol table. They create a new symbol of the corresponding type and assign it to the currently active scope.

```
public interface IC3P0ParserContext {
  ...
  ClassSymbol addClass(String className);
  EnumSymbol addEnum(String enumName);
  FunctionSymbol addFunction(String name, Scope lookupScope);
  TemplateSymbol declareTemplate();
  TypedefSymbol addTypedef(String typedefName, Type definedType);
  VariableSymbol addVariable(String varName, Type variableType);
  ...
}
```

Listing 3.17: Symbol Definition Methods

Symbol Specification These methods further specify existing symbols in the symbol table. Currently, this just includes methods for making other scopes or symbols visible in the active context and for adding base classes to existing classes. While the symbol table just checks for existence of the derived class at the moment, there might be further verification of visibilities added later.

```
public interface IC3P0ParserContext {
  ...
  void associateNamespace(NamespaceScope namespace);
  void defineUsing(Symbol s);
  void addBaseClass(Type symbol, ClassSymbol derived);
  ...
}
```

Listing 3.18: Symbol Specifying Methods

Symbol Query These methods check the existence of and resolve symbols. They are used in the parser for making decisions in semantic predicates and querying scopes for local lookups performed in the parser. At a glance there are only methods for querying type and scope symbols. In the first implementation these should suffice, to let the parser decide on the rules correctly.

```
public interface IC3P0ParserContext {
  ...
  boolean isKnownType(String type);
  boolean isKnownClass(String className);
  boolean isKnownTemplate(String templateName);
  boolean isKnownEnum(String enumName);
  boolean isKnownTypedef(String typedefName);
  boolean isKnownNamespace(String namespaceName);
  Scope resolveScope(String scopeName);
  Type resolveType(String string);
  NamespaceScope resolveNamespace(String namespaceName);
  Scope currentScope();
  Scope scopeOf(String scopeID);
  ...
}
```

Listing 3.19: Symbol Query Methods

Scope Switch These methods are for switching the currently active and temporarily included scopes. These methods are also used to track the active location in the nesting structure in the parser, which is used for unqualified and qualified name resolution.

```
public interface IC3P0ParserContext {
  ...
  void enterTemplateArg();
  void leaveTemplateArg();
  void enterParentheses();
  void leaveParentheses();
  void enterType(String typeName);
  void leaveType();
  void enterNamespace(String namespaceName);
  void leaveNamespace();
  void enterScope();
  void leaveScope();
  void enterLocalLookupScope(Scope scope);
  void leaveLocalLookupScope();
  void enterAugmentedLookupScope(Scope localScope);
  void leaveAugmentedLookupScope();
  ...
}
```

Listing 3.20: Scope Switch Methods

Scope Check These methods check the current environment. For example, whether the parser is in a template argument, which is used to distinct closing angle brackets from greater than operators. Or whether it is in a certain class definition, making the name of the surrounding class a valid type name.

```
public interface IC3P0ParserContext {
  ...
  boolean isInClass(String className);
  int inTemplateArg();
  ...
}
```

Listing 3.21: Scope Check Methods

Template Manipulation Due to the special way of template recognition in the parser, we have two further methods used to complete the parameter and declaration section of a template. Further information about template handling can be found in Section 3.4.4.

```
public interface IC3P0ParserContext {
  ...
  void finishTemplate();
  void finishTemplateParameterScope();
}
```

Listing 3.22: Template Manipulation Methods

Symbol Table Classes

The symbol table implementation consist of the following classes, also depicted in the class diagram in Figure 3.5. Interaction from the parser's point of view happens through the `IC3P0ParserContext` interface, implemented by the `SymbolTable` class. Together these classes build the `c3p0.parser.implementation.symboltable` package.

SymbolTable This is the main class for tracking the whole context of the parser. It maintains a reference to the currently active scope and manages temporarily active scopes for local resolution and hiding through qualified resolution. Furthermore, it tracks the nesting of template parameter declarations and template arguments of simple template ids. For accessing the `SymbolTable` from the parser it implements the complete `IC3P0ParserContext` interface.

AngleBracketCounter The `AngleBracketCounter` helps counting opening and closing angle brackets of template arguments. This might look simple at a glance, but requires also tracking of opening and closing parentheses, which reset the counting locally inside the enclosed region.

ScopeStack The `ScopeStack` is used when having one or several scopes hiding other scopes, or for having additional scopes augmented to the currently active scope. The topmost scope can be queried directly.

SymbolMap Every scope consists of a `ScopeMap` which is an implementation of a multi map. We require the capability to have one name to map to several symbols locally, even in well-formed programs, for example, when having an overload set for a certain function. Internally the symbols of the same name are kept in an ordered list. New symbols are added to the front of the list, as it is most likely that the last definition of a symbol is the most demanded, due to hiding.

Symbol Classes

The scope tree in the symbol table consists of three interfaces and several distinct classes. The structure of these classes is depicted in Figure 3.6. Different types of C++ program elements are represented by their corresponding node types of the scope tree. Defining

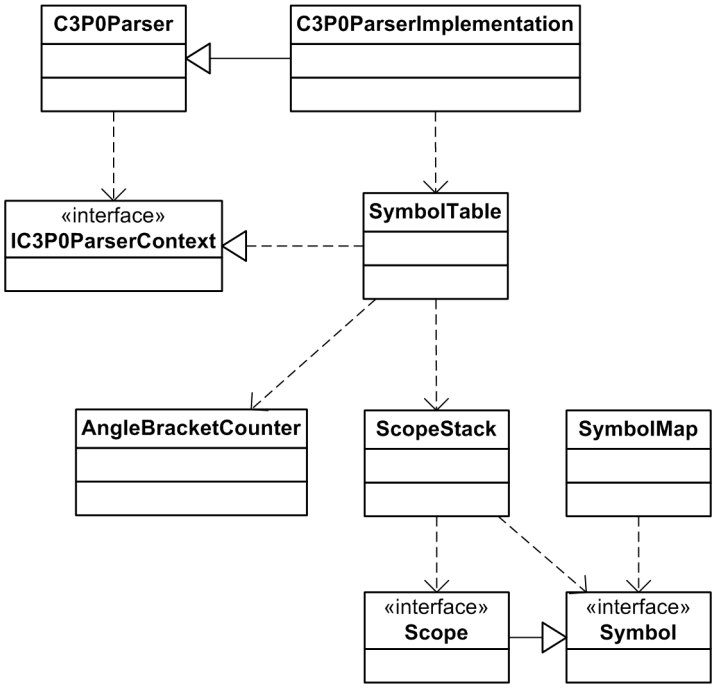

Figure 3.5.: Symbol Table Class Diagram

different types of node enables type specific behavior behind a common interface. Some types, though, do not specify this behavior explicitly and just inherit it from their ancestors.
Let us first have a look at the three interfaces `Symbol`, `Scope` and `Type`:

Symbol This is the basic interface implemented by all nodes of the scope tree. We have seen that most nodes have or can have a name, except for local scopes. Therefore, every `Symbol` can be queried for its name. Furthermore, it defines two methods which allow a symbol-specific transformation to `Scope` and `Type`. This is primarily used to avoid a bunch of `instanceof` queries and allow symbols like `typedef` to be interpreted as a scope, delegating the calls to the effective type behind the typedef.

```
String getName();

Scope asScope();
Type asType();
```

Listing 3.23: Interface: Symbol

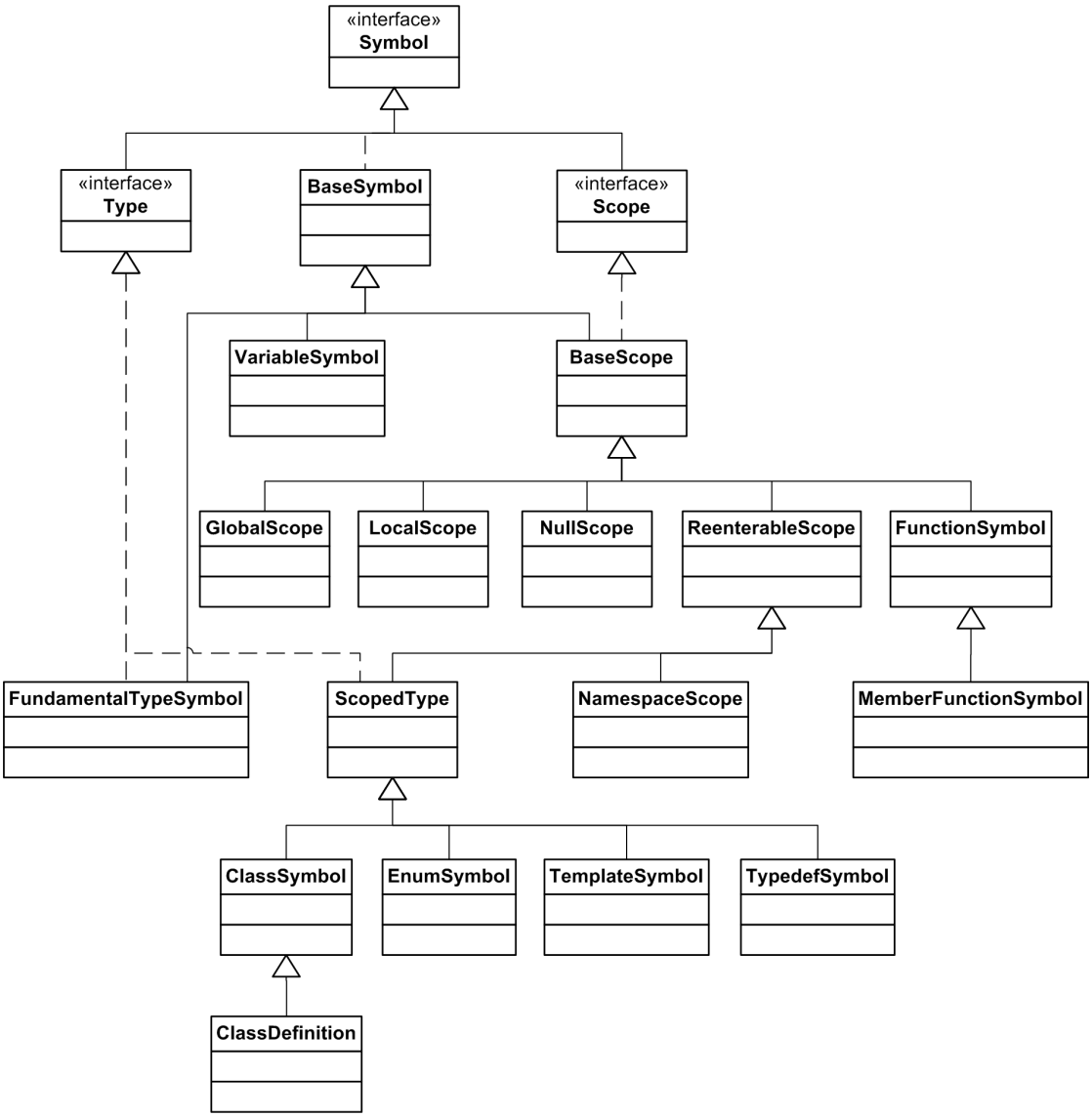

Figure 3.6.: Symbols Class Diagram

Scope Every node which can contain declarations of other symbols derives from `Scope`, which extends `Symbol`. Therefore, these nodes are possible inner nodes of the scope tree. All `Scope` nodes, except the node representing the global namespace, have an ancestor scope in the tree hierarchy. With `getParentScope` the surrounding scope can be accessed.

Symbols declared in a scope can be resolved through the `resolve` methods. These methods take several different parameters:

- The name of the sought-after symbol.
- A set of scopes already visited. The C++ standard specifies that every namespace is visited at most once during a single lookup. Furthermore, this prevents lifelocks in mutually including namespaces.
- A flag to indicate whether to check associated namespaces.
- To search the scopes for a symbol of a specific type, it is possible to pass a subtype of `Symbol`, to ignore all other symbol types.

The `resolve` methods with simpler signature are primarily intended to avoid long repetitive parameter lists when calling these methods from the parser. Additionally, we distinct between resolution methods which try to find a specific symbol starting at the current scope and methods which try to resolve a symbol locally: `resolveLocal`.

Beside querying symbols from a scope, new symbols can be defined, using the `define` method. We added a separate method for defining new namespaces, as even though a namespace can be split up into several parts, it remains the same namespace. At last we provide the possibility to map namespaces into current scopes, which is a functionality required due to the `using` directive.

```
Scope getParentScope();

Symbol resolve(String name);
Symbol resolve(String name, Set<Scope> visitedScopes, boolean checkNamespaces);
<T extends Symbol> T resolve(String name, Class<T> nodeType);
<T extends Symbol> T resolve(String name, Set<BaseScope> visitedScopes,
                             boolean checkNamespaces, Class<T> nodeType);

Symbol resolveLocal(String name);
<T extends Symbol> T resolveLocal(String name, Class<T> nodeType);

void define(Symbol symbol);
void define(NamespaceScope namespace);
void associateNamespace(NamespaceScope associatedNamespace);
```

Listing 3.24: Interface: Scope

Type `Type` derives from `Symbol`. Currently, it is a marker interface only. Therefore, it does not have any additional methods.

The structure of the classes effectively implementing the symbol types is a bit more complex and consists of several distinct types. The following classes do not posses the scope or type property:

BaseSymbol This is the base class for all scope tree nodes. It contains the general behavior of any symbol, which is the implementation of the `Symbol` interface, including the three methods `getName`, `asScope` and `asType`. We avoided to have `BaseSymbol` as the root of our symbol structure, as we would not have the possibility in Java to have diamond inheritance according to our requirements to have symbols implementing `Scope` and/or `Type`. The `BaseSymbol` contains the name of any symbol.

VariableSymbol The `VariableSymbol` represents a declared variable. Each variable declaration has a corresponding type assigned, which specifies the type of the variable. `VariableSymbol`s are always leaf nodes in the scope tree, as they cannot contain child nodes.

Most symbols open a new scope, which can contain symbol declarations themselves. In the following we have described the non-type scope symbols.

BaseScope `BaseScope` is the basic class implementing the general behavior of a scope, regarding symbol declaration and resolution. The `BaseScope` contains the parent relationship of scopes, has a set of associated namespaces and maintains a `SymbolMap` for managing the local symbols. Several symbols open a new scope. They derive from this class.

GlobalScope The `GlobalScope` represents the global namespace existing in every translation unit.

LocalScope Always when a compound statement is entered in the parser, a local scope is opened. `LocalScope`s primarily encapsulate local declarations from the surrounding context. Furthermore, they allow shadowing declarations in parent scopes. Local scopes are always anonymous.

NullScope The `NullScope` class reliefs the check whether scopes during resolution are `null` and allows the execution of symbol queries on non-existing scopes. Obviously a `NullScope` will never successfully resolve a symbol locally, as it cannot contain any. Subsequently, the methods for defining new symbols prohibit these operations.

ReenterableScope Some scopes, after definition, can be reentered again for symbol resolution. This is particularly important for qualified lookup. Scopes deriving from `ReenterableScope` represent such scopes. This class is a marker class, which does not add additional functionality. For example, `ClassDefinition` can be reentered for symbol resolution, while `LocalScopes` cannot.

NamespaceScope Namespaces are represented by `NamespaceScope`s. They are another example of a `ReenterableScope`. `NamespaceScope`s do not represent a type and do not add any additional functionality.

FunctionSymbol For a smooth integration into the scope tree, `FunctionSymbol`s derive from `BaseScope` too. We avoided implementing the function representation as a simple `BaseSymbol` containing a `LocalScope` field for the function body. This would imply unnecessary and complex logic for jumping (ascending) in the scope tree, when in this local context. Because the body is represented as a `compound_statement` it then just becomes a child node in the scope tree. Additionally, a `FunctionSymbol` contains a list of parameters and a return type, specifying the function signature.

MemberFunctionSymbol `MemberFunctionSymbol` adds the possibility to assign a member scope to a function symbol. Through this member scope, local lookup can access symbols which are not declared in the local or a parent scope, but are known due to the membership of a class or a namespace. According to this feature the `resolveLocal` method is overridden to include this scope.

Symbols which represent a type implement the corresponding `Type` interface, either directly or indirectly. These are the following symbols:

FundamentalTypeSymbol There are several fundamental types like `int`, which are available in almost any context. They do not open a new scope and can only be referenced. `FundamentalTypeSymbol`s represent these symbols for resolution only. Furthermore, the class contains a constant for each fundamental type, which can be used instead of creating new symbols. Since it is not expected that more fundamental types are introduced into the scope tree, its constructor is private.

ScopedType The `ScopedType` combines scopes and types. This is the general base class for symbols like `ClassSymbol`, which define a new type and provide a scope to enter, for name resolution.

ClassSymbol Classes add a little bit of complexity to the scope tree by forming cross-references that origin from their inheritance structure. Every `ClassSymbol` contains a set of other `Types` it inherits from. During resolution, the inherited scopes are preferred over the surrounding scopes. `ClassSymbol`s are intended to represent class declarations.

ClassDefinition `ClassDefinitions` inherit from `ClassSymbols`. They are used if a class has been previously declared, but not implemented. According to the context of the declaration symbol, resolution can return different results. Therefore, a `ClassDefinition` contains a reference to its `ClassSymbol` declaring the class and specifying the context.

EnumSymbol Enums are similar to classes, regarding their inheritance capabilities, except that they can only be derived from enums themselves. `EnumSymbol` do not allow association of namespaces in their bodies and therefore deny to react on such requests.

TemplateSymbol `TemplateSymbol`s are decorators for other declarations. This implementation is similar to the translation unit implementation in the CDT-AST and is also required by the implementation of the rules regarding templates in the `C3P0Parser`. Subsequently, a template symbol contains a to-be-decorated symbol and a list of template parameters, for local resolution. As the scope part of the `TemplateSymbol` is more or less equal to a local scope containing the template parameters, the decorated declaration would generally be hidden from its context. Because initially the template name cannot be known – due to the construction of the parser rule set – a temporary name is generated by the symbol table. The `TemplateSymbol` retains the obligation to register itself in its parent context, after its name becomes clear through parsing the declaration. Local resolution first happens in the template parameter list and, if unsuccessful, is delegated to the decorated symbol.

TypedefSymbol These symbols are primarily for mapping an existing symbol into the local scope using a name specified. `TypedefSymbol`s act as a mediator regarding resolution. They inherit the type and scope properties, which are accessed by the delegated `asScope` and `asType` methods, from the defining type.

3.3.3. Unnamed Symbols

There are some entities which not necessarily require a name in C++. For example unnamed namespaces or composite types which are instantiated immediately. As our symbol table requires a name to put them into the `SymbolMap`, we need to create one. `SymbolTable` provides the method `generateAnonymousIdentifier`, which takes a string as an argument, to identify the type by the name. The `SymbolTable` keeps track of the number of unnamed entities. To ensure the names of unnamed symbols do not clash with user defined symbols, the names generated are enclosed in angle brackets, which cannot occur at the beginning of an identifier name. The generated part of the name is the index number of the current unnamed entity. Subsequently, an anonymous class which is the fifth unnamed entity would get the following name:

```
"<Class5>"
```

Listing 3.25: Name Example of an Anonymous Class

Furthermore, we use these generated names to identify the `TemplateSymbols` as long as the declaration they contain is not parsed and therefore, the name is unknown.
The number of such anonymous names is limited to 18'446'744'073'709'551'616, by the size of Java `long`, which probably will suffice for all current applications of the parser.

Resolution Process

The symbol table maintains the scope tree and always knows the active scope. This scope should be consistent with the current nesting structure of the currently parsed C++ code. For special situations, where the scope is switched temporarily, there are two stacks for providing additional lookup contexts. For qualified lookup it is necessary to completely hide the current environment. Therefore, we maintain a stack of hidden scopes. When we enter such a hiding scope, we push the current scope onto the stack of hidden scopes and set the hiding scope as active. When leaving the hiding scope, the hidden scope is popped from the stack and set as the active scope again.
Additionally, to the active scope, there can be scopes that extend the active scope for some reason. We have called such scopes augmenting lookup scopes, which can be searched for symbols too, augmenting the current context virtually. This extension of the current scope is required, for example, when parsing implementations of previously declared classes. Let us have a look at a specific example:

```
namespace NS {
  typedef int I;
  struct S;
}
typedef float F;

struct NS::S {
  I i;
  F f;
};
```

Listing 3.26: Augmented Scope Example Code

In the C++ code above we have the definition of the type `I` and the declaration of struct `S` inside the namespace `NS`. Furthermore, we have defined the type `F` after the namespace `NS`. The implementation of `S` has references to type `I`, declared in `NS`, and type `F`, declared before the implementation of `S`. Therefore, while parsing the implementation of `S` we do not want to hide the surrounding namespace of the implementation completely. Nevertheless, the namespace `NS` needs to become visible in the body of `S` as well.
We also use this approach while parsing parameter lists of member function definitions. This is an implication of the parser rule implementation. While parsing the body of a member function, the function body scope automatically has the association to the scope that function is a member of. We do not have a member function object at the point where the parser has to decide whether the parameter list is valid. Subsequently, we need to enter the scope of the declaration temporarily, to ensure all parameters can be resolved correctly.
Resolution in the cases above could happen by hiding the current scopes with the augmented scope completely, as the definitions in the global namespace could be resolved through the global namespace, which is a parent of the namespaces anyway. We decided to distinct between scope hiding and scope augmentation for two reasons:

- When hiding the current scope completely, all new definitions are added to the hiding scope. In the case of class implementations of declared classes, we would like to have the class definition in place, where it really exists, instead of placed into the augmented scope.
- Currently, we do not have checks for relative positions of declarations to referenced symbols. If we implement such restrictions we could encounter problems, for example when approaching a symbol declared later in a scope, from the scope of a previous declaration.

Resolution Example

There is a short piece of C++ code in Listing 3.27. This example shows two structs `Base` and `Sub` deriving from `Base`. In the class `Base` we have defined type `F`. In the subclass we have a field of type `F`. While parsing the body of `Sub` the parser encounters the declaration of `f`, which implies that `F` can be resolved. As the declaration of `F` is not part of the branch in the scope tree containing `Sub`, we need to have the inheritance structure represented in the `ClassSymbol`.

```
struct Base {
  typedef float F;
};

struct Sub : Base {
  F f;
};
```

Listing 3.27: Resolution in Base Class Example

Figure 3.7 shows a sequence diagram, depicting the resolution process:

- The parser queries the symbol table, whether `F` is a known type, by calling the `isKnownType` method.
- The symbol table tries to resolve the symbol (`F`) in the augmented scopes first. As there are no augmented scopes, the search continues.
- Next the symbol table tries to resolve `F` in the current scope, by calling `resolve`.
- `ClassSymbol` overrides the `resolve` method, which first performs a local lookup.
- As `resolveLocal` (searching the local `SymbolMap`) does not return a known symbol, the base classes are queried, by calling `resolve` of each base class.
- `resolveLocal` of the base class `Base` finds a symbol in the local `SymbolMap` and is successful. The definition of `F` is returned, back to the symbol table.

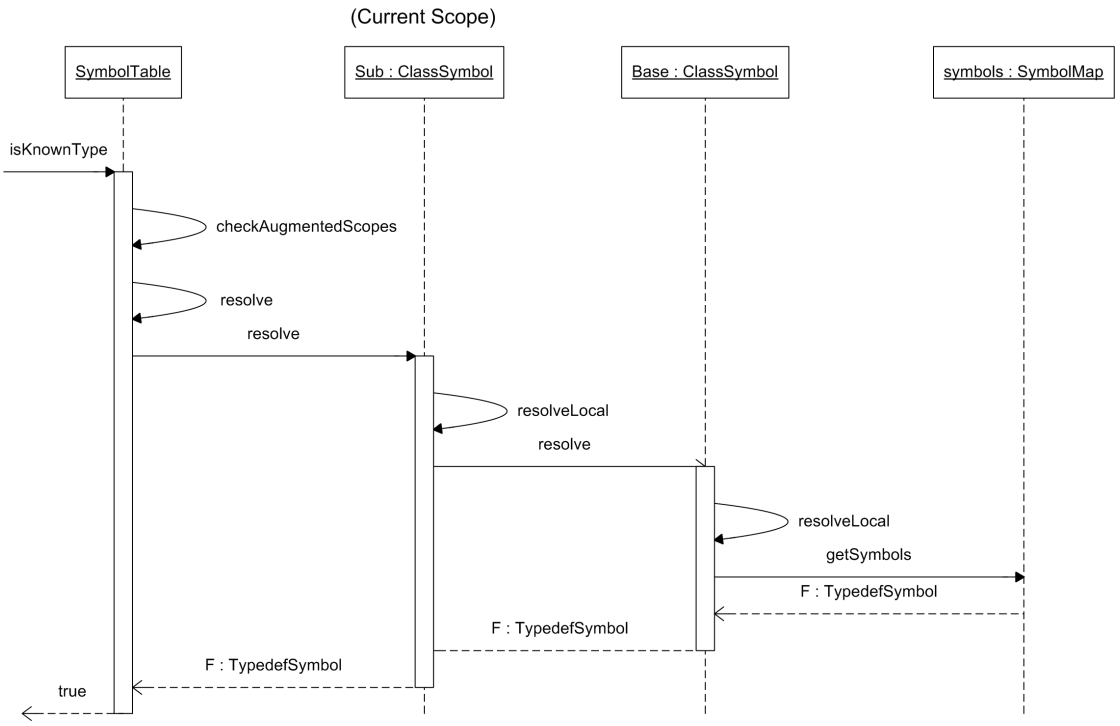

Figure 3.7.: Resolution Process Example

3.3.4. Testing

Our main focus for symbol table testing lies on the correct interaction from the parser's point of view. The most important parts of the symbol table tests are against the `IC3P0ParserContext` interface. The basic approach is to take a virtual piece of C++ code and call the methods, which should be called by the parser, in the correct order with the corresponding arguments. Below we have listed an example of such a test case. This is the C++ code segment, which shall be tested virtually:

```
class T{};
template<class T> class Templ {
  T t;
};
T k;
```

Listing 3.28: Symbol Table Test Example C++ Code

While the code above is parsed, the `C3P0Paser` should perform the following operations on the symbol table. We emulate this behavior by calling these methods directly. The results are verified at the end of the test case.

```
@Test
public void testHideWithTemplateParameter() {
  SymbolTable symbolTable = new SymbolTable();

  String klassName = "T";
  String templateName = "Templ";

  ClassSymbol klass = symbolTable.defineClass(klassName);

  symbolTable.declareTemplate();
  ClassSymbol templateParam = symbolTable.addClass(klassName);
  symbolTable.finishTemplateParameterScope();

  symbolTable.defineClass(templateName);
  symbolTable.enterClass(templateName);
  ClassSymbol resolvedTypeInTemplate = symbolTable.resolveClass(klassName);

  symbolTable.leaveType(templateName);
  symbolTable.finishTemplate();

  ClassSymbol resolvedTypeAfterTemplate = symbolTable.resolveClass(klassName);

  Assert.assertEquals(templateParam, resolvedTypeInTemplate);
  Assert.assertNotSame(klass, resolvedTypeInTemplate);
  Assert.assertEquals(klass, resolvedTypeAfterTemplate);
}
```

Listing 3.29: Symbol Table Test Implementation

In this particular case we assure that a template parameter shadows a class symbol with the same name, inside the template definition. First, we define the class `T`. Then we open the template declaration and add the template parameter. After closing the template parameter section, we add the template class `Templ`. Inside the template class we resolve the type `T`, which is expected to be the template parameter. Then we leave the template class definition and finish the template declaration. Outside the template we resolve `T` again to check whether it refers to the class `T`, defined before the template. Finally, we perform the checks to assure the symbols have been resolved as expected.
As we have implemented the symbol table and the name resolution side-by-side and this is a particularly laborious way of testing, especially in more complex cases, we implemented most tests as black box parser tests. They indirectly test the behavior of the symbol table.

3.3.5. Known Issues

The current implementation of the symbol table feels solid. During the implementation of the name resolution in the parser, we did not encounter any serious flaws or major deficiencies which revealed the scope tree as completely inappropriate. The basics of symbol resolution work fine and our test cases as well as a preprocessed hello world are completely satisfied with the current implementation. Nevertheless, we are aware of certain situations which are not covered yet. Some of them are the following:

Access Qualifiers The symbol table does not track access qualifiers yet. Subsequently, for members which have restricted visibility, the parser fails to recognize an ill-formed program as invalid. For example, the following C++ code is not valid as `F` should not be accessible in `Sub`, due to the `private` access qualifier:

```
struct Base {
private:
  typedef float F;
};

struct Sub : Base {
  F f;
};
```

Listing 3.30: Example for Restricted Member Access

When implementing a solution for this issue, we would have to decide where to perform this check. We have three possibilities: (1) To check during the resolution process in the symbol table and do not consider `F` as a resolvable declaration if not visible, which restricts the resulting error to: *F is not a known type.* or similar less helpful messages. (2) We could track the visibility in the symbol table and perform the check in the parser, which allows better error indication but bloats the parser with verification logic. (3) Member access could be checked as a post-processor, after parsing the translation unit. This is probably the most expensive way of verifying visibility as it adds further passes over the resulting AST. But it could be implemented as an optional feature, that can be disabled for certain contexts, depending on the requirements. Subsequently, we required a configurable parser which can perform very pedantic checks or be more forgiving.

Overload Sets Name resolution in C++ sometimes is not limited to a single result. Whereas a symbol eventually has to be resolvable unambiguously, there are intermediate steps, which return sets of symbols with the same name. For example, function names can have several implementations of the same overloaded function. The lookup in the symbol table should result in a set containing all those methods. The specific function, which should be called, is then determined through the parameter list. In the first version of C3P0 we are avoiding checks for type compatibility. Subsequently, we are not able to perform checks on parameter lists and cannot decide whether there exists a function with matching signature. Therefore, we do not need to return more than one symbol per resolution. There are other cases where ambiguities could occur. For example, when performing qualified lookup, the first name specifier could be ambiguous, perhaps through a `using` directive. The current implementation does not recognize such ambiguities and just resolves to the first matching symbol. In the worst case, we recognize an ill-formed program as correct.

Argument Dependent Lookup Similar to the lacking distinction of function symbols from the overload set, we are unable to perform argument dependent lookup yet. As long as we have not implemented the logic for deriving the type of an expression we cannot perform resolution on this level. Basically, it not very demanding to add the scopes of the parameters for argument dependent lookup to the lookup scope for function symbols, at least from the symbol tables point of view. However, here we encounter a problem in the parser. According to the current rule set, the declarator of a function is already parsed when encountering the parameter list, which might augment the lookup scope. We could not alter the lookup result retroactively. Therefore, we would need to check the existence of the function after parsing the whole signature. Thereby, the current implementation comes in handy, as we parse function calls without verification anyway and we can easily add verification afterwards, without changing much of existing semantic checks.

Expression Type Deduction As mentioned in Section 3.2.7, currently we cannot deduce the type of an expression. We have listed this as a drawback of the symbol table, where we expect this functionality to be implemented. Implementing it in the parser would probably bloat it up too much. For testability and readability reasons, an implementation in the symbol table is more desirable. Furthermore, type deduction heavily depends on the context and operator resolution, thus, it fits better to the parser context anyway. Currently, we can continue without this feature and expect CDT to be able to perform such checks, after we have created the AST. Due to recent changes to the C++ standard, a significant flaw popped up, because of the lack of this ability. In [Van10b] `decltype` has been extended to be a valid name specifier. While the syntactic change is minor, we would require to be able to determine the type of an expression, when using `decltype` as a name specifier for further resolving the qualified name.

Pointer Operator For a proper implementation of expression type deduction, as described before, we also need to track types of variables and return types of functions more exactly. Currently, we are just assigning the raw type to such declarations and do not distinct between pointers, references or values. While this is not required as long as we do not intend to perform type-compatibility checks, we do not need it. As soon as we want more pedantic semantic checks, we must extend the symbol table by this information. Probably the easiest way is to use decorators, which add the specifying type information.

We do not expect that the points above are complete show stoppers. In the worst case we end up with a lack of semantic verification, which is not covered in the parser yet, anyway. Basically, this only results in accepting a possibly ill-formed C++ program. As our main goal is to create a parser which is able to generate an AST for refactoring, we can live with these drawbacks and continue to focus on other basic challenges, which have to be solved. Furthermore, we expect it to be absolutely feasible to extend the current symbol table implementation for the feature above – it just takes additional time, which is not available during this master thesis.

3.4. Name Resolution

Beside implementing the symbol table we had to integrate it into the parser. As mentioned before, we did this more or less concurrently. Therefore, we knew immediately whether certain approaches worked out or not. Subsequently, the symbol table implementation is depending on some restrictions inflicted by the parser. In this section we describe the integration of the symbol table into the parser. We are not elaborating on every single rule interacting with the symbol table. We will rather introduce the basic ideas behind the concepts and describe how we understood and used them, to allow a reader understanding our parser's rule set.

3.4.1. Symbol Table Purpose

The idea behind the symbol table is quite simple. The parser cannot decide purely on syntactic information, which rule has to be processed next. Alternatives in syntactic rules might be ambiguous. In parsers for very simple languages, it is possible to decide which rule to take only by looking at the next token in the stream. ANTLR provides the possibility to match a rule alternative only when the whole alternative, or a certain subpart, would match. This is realized using syntactic predicates. For very complex languages like C++, even this possibility is not sufficient, as we encounter ambiguities which can only be resolved using context information. In ANTLR it is possible to integrate semantic checks, using semantic predicates.
For disambiguating token sequences with semantic information, we have to record the necessary information during the parse process. Here, the symbol table comes into play. In the following, we will focus on the integration into the parser using semantic predicates, which partially already have been implemented during the term project [Cor10a].

3.4.2. Semantic Checks

During the term project, when using our monolithic scope, we have already added several semantic predicates to the parser. Let us have a look at a quite simple example of a semantic check. We have inlined some of the sub rules and removed the actions of `simple_template_id` for having a more concise rule for explanations.

```
simple_template_id
  : {context.isKnownTemplate(input.LT(1).getText())}? IDENTIFIER
    '<' template_argument_list? '>'
  ;
```

Listing 3.31: Semantic Predicate in simple_template_id

The semantic predicate is an expression in Java evaluating to `boolean`, enclosed in `{...}?`. It checks whether there is a template defined in the active context, with the

name corresponding to the next token. This rule alternative is only possible, if this semantic check returns `true`.
It is also possible to use semantic predicates inside syntactic predicates, a syntactic restriction to enter the rule enclosed in `(...)=>`. For example, in the `member_declaration` rule we know that a function declaration which has a declarator name that is identical with the surrounding class, is a constructor declaration. As the syntactic predicate only checks whether the rule alternative would match, we have extracted only as much as needed from the `constructor_declaration` rule into the predicate. This saves parse time, opposed to a complete rule as a syntactic predicate. Like for example `(constructor_declaration)=>`, which had to be fully matched.

```
member_declaration
...
  | ({context.isInClass(input.LT(1).getText())}? IDENTIFIER '(') =>
    constructor_declaration
...
```

Listing 3.32: Semantic Predicate in Syntactic Predicate

The semantic predicate checks whether the `IDENTIFIER` corresponds to the name of the surrounding class. If the predicate fails to match, this alternative is not taken. Even if it matches, there can still occur a parse error in the `constructor_declaration` sub rule, which is eventually the same as if the parse error occurred without having a syntactic predicate.
In the current implementation of the parser we limit our semantic checks to tests whether an identifier is a known type, template or namespace. Further verifications could be implemented as well, but are not required for deciding which parser rule has to be taken next – which is the problem we intend to solve with the symbol table.

3.4.3. Actions

Beside implementing the semantic predicates we have to implement the actions for keeping the symbol table consistent with the current context. We use actions in the parser rules to perform the necessary operations. The action itself contains Java code, which is placed more or less as-is at the corresponding location in the parser. The following example shows the rule `compound_statement`.

```
compound_statement
  : {context.enterScope();} '{' statement_seq? '}' {context.leaveScope();}
  ;
```

Listing 3.33: Actions in compound_statement

Before parsing the alternative of the rule, a new local scope is opened through an action (`{...}`). Therefore, all symbol declarations contained in this compound statement will be local to this block. After parsing the rule is finished, the scope is left. It is particularly important that entering and leaving scopes is symmetric and opened scopes are always closed properly. It is desirable not to let scopes open across the boundaries of a rule. This should only be done if the there is certainly only one path through the rules which alway closes the scopes opened before.
Beside tracking scopes, we declare symbols, like for example class declarations. Because usually there are several rules involved in a declaration, we have to decide where the action to add the symbol fits best. In the case of class definitions we have decided to add the class symbol to the symbol table in the `class_head` rule, after the `IDENTIFIER`. Below we have a simplified version of the rule.

```
class_head
  : class_key nested_name_specifier?
    id=IDENTIFIER {context.addClass($id.getText())} base_clause?
  ;
```

Listing 3.34: Action for Declaring a Class Symbol

After adding the action for declaring the class symbol, we know that this class symbol is available in other rules having `class_head` as a sub rule – at least after parsing `class_head`. The only rule including `class_head` is `class_specifier`:

```
class_specifier
  : class_head '{' member_specification? '}'
  ;
```

Listing 3.35: Syntactic Parts of Rule `class_specifier`

We have already seen the rule `constructor_declaration`, which is reachable through the rule `member_specification`. As we have explained previously, there are semantic checks in the `constructor_declaration` rule, to verify whether the constructor name would match the name of the surrounding class. But, at the current state we are not inside the class added before, in the `class_head` rule. Therefore, a constructor could not be recognized yet – at least not as a constructor. To achieve this, we have to enter the class, previously declared, before parsing the `member_specification` sub rule. This is similar to entering a local scope in `compound_statement`:

```
class_specifier
  : head = class_head
    {context.enterType($head.name);}
    '{' member_specification? '}'
    {context.leaveType();}
  ;
```

Listing 3.36: Scope Switch Actions

As the symbol table, after executing the `enterType` method, virtually resides in the class definition, the constructor can be correctly resolved. `enterType` is parameterized with the name of the type to enter, which requires us to return the class name from the `class_head` rule.
From the beginning of this chapter, we know that there is a further case, which the implementation above does not cover yet. Consider the following C++ code snippet:

```
namespace N {
  typedef int I;
  class C;
}

class N::C {
  I i;
};
```

Listing 3.37: Qualified Class Definition

The class `C` has been declared in the namespace `N` and is defined outside, using the specified name `N::C`. So long, this works fine. A problem occurs in the body of the class definition. There is a declaration of the variable `i` referencing the type `I`. Unfortunately, `I` is not a known symbol in this scope. For being able to resolve `I` correctly, we have to extend our lookup scope by the namespace `N`. Therefore, we have to extend the current scope by the scope of namespace `N`. We do not want to switch the scope completely, as we want to retain the surrounding declarations of the class definition visible:

```
class_specifier
  : head = class_head
    {context.enterType($head.name);
      context.enterAugmentedLookupScope($head.lookupScope);}
    '{' member_specification? '}'
    {context.leaveAugmentedLookupScope();
      context.leaveType($head.name);}
  ;
```

Listing 3.38: Augmenting the Current Scope

There is information required about which namespace to take as augmenting scope. The name specifier `N::` is matched in the `class_head` rule and can only be resolved there. Subsequently, the namespace `N` can be returned from the `class_head` rule, back to its caller. We can then access this scope information through the name of the return value, `lookupScope`.

Scope Switch and Predicates

One major problem remains while switching scopes using actions as described above. Actions are not executed during evaluation of syntactic predicates. This actually makes sense, as this could imply inappropriate context switches. On the other hand this is a major problem for the evaluation of semantic predicates, which have to be successful during syntactic predicate evaluation as well. The following example illustrates the problem:

```
namespace N {
  struct S{
  };
}

N::S s;
```

Listing 3.39: Problem with Actions in Syntactic Predicates

The declaration of `s` requires to resolve the type `S` in namespace `N`. To be able to recognize `S` as a valid type we have to switch the scope, at least temporarily, to namespace `N`. Switching the scope requires actions, which would not be executed if the rule for matching the declaration specifier `N::S` is guarded with a syntactic predicate containing this switch. Nevertheless, the semantic predicate would be evaluated to check whether `S` is a known type. But, as we did not switch the scope due to the omitted action, the current scope (the global namespace in this particular case) is queried. As `S` is not a known type symbol, the syntactic predicate fails because of the inappropriate context during the evaluation.

Alternatives to Actions

As we cannot avoid having syntactic predicates for decision making an alternative approach is required. Due to the restrictions of syntactic predicates we are quite limited in our possibilities. Above we have stated that actions cannot be used for scope switches. Additionally, we cannot work with return values of rules and the `@after` parts of rules are not executed as well. There are two possibilities remaining: (1) We can still access the textual representation of a sub rule. (2) It is possible to perform actions in the `@init` and `finally` block of a rule. As it is the intention to setup the environment of the rule in these parts, they are executed even during predicate evaluation.

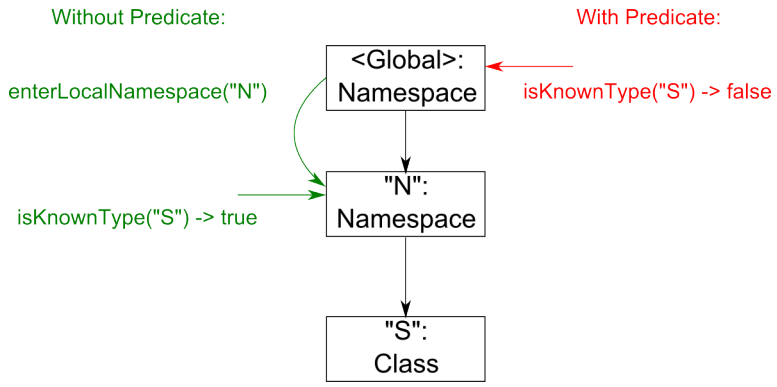

Figure 3.8.: Query Depending on Syntactic Predicate

We decided to primarily use the second option, as to split up a previously parsed text representation of a rule's match lets us virtually undo the recognition task in the symbol table. Furthermore, it can become particularly complex to break some constructs apart again. Nevertheless, we are still accessing the text representation of some tokens to perform the resolution of scopes. With the following example we outline the implementation of our approach. Let us consider the rule responsible for recognizing the type specifier in `N::S` from the example of 3.4.3:

```
simple_type_specifier
...
  | '::'? ((nested_name_specifier) => nested_name_specifier)?
    (type_name | TEMPLATE simple_template_id)
...
  ;
```

Listing 3.40: Nested Type in `simple_type_specifier`

If we added the action for switching scopes here after matching `nested_name_specifier` it would not necessarily be executed and `type_name` or `simple_template_id` would be resolved in the current scope. Therefore, we have replaced this alternative by the following rule:

```
simple_type_specifier
...
  | SCOPE? nested_type_specifier[context.currentScope()]
...
  ;
```

Listing 3.41: Adapted Rule `simple_type_specifier`

```
nested_type_specifier [Scope localScope]
@init{context.enterLocalLookupScope(localScope);}
  :   ( (TEMPLATE? IDENTIFIER '<') =>
        TEMPLATE? {context.isKnownTemplate(input.LT(1).getText())}?
        id=IDENTIFIER '<' template_argument_list? '>'
      | {context.isKnownType(input.LT(1).getText())
        || context.isKnownNamespace(input.LT(1).getText())}?
        id=IDENTIFIER )
      ((SCOPE) =>
        SCOPE nts=nested_type_specifier[context.resolveScope($id.getText())])?
  ;
  finally{context.leaveLocalLookupScope();}
```

Listing 3.42: New Rule `nested_type_specifier`

As we are not able to set the return values in actions, we had to inline some of the sub rules. Consequently, the rule `nested_type_specifier` looks a bit bloated, but is actually rather simple. As can be seen it takes an argument of type `Scope`, named `localScope`. This is the scope to perform our name resolution in. For setting the `localScope` as the active scope, we execute `enterLocalScope` in the `@init` block, which is always executed. Then, we either encounter a `simple_template_id`, a namespace or a type, which is recognized as known in the symbol table. If we encounter a following `SCOPE` operator we match it and call the rule recursively with the resolved scope as argument. After leaving the rule, we exit the local scope in the `finally` block. As `@init` and `finally` are always executed, we can be sure that after leaving the rule, either after predicate evaluation or the real parsing, the active context is reset to the context active before entering the rule.
There are three remarks to this solution: (1) This particular implementation is not as restrictive as it could be. It might be possible to match a single or nested namespace with this rule, but it actually should recognize a type. We defer handling of this issue to the point when we refine error handling, as such a construct would definitely indicate an ill-formed program. (2) Switching contexts using `@init` and `finally` blocks seems not necessary in this particular case, as we could just execute the lookup on the argument `localScope`. But, this actually bypasses the symbol table and a possible check for access specifiers regarding the current scope. (3) It would also be possible to force execution of actions while evaluating syntactic predicates, by changing the `@synpredgate` property in ANTLR. Unfortunately, this is a global property and cannot be overridden for a single rule. As this would imply that all actions would be executed during predicate evaluation we would have to do too much cleanup work, because most actions executed would have unwanted side-effects, which only are desired when the corresponding rule matches completely. Therefore, we do not consider this as a feasible alternative.

3.4.4. Template Handling

As the rules recognizing template declarations imply symbol table integration in a particular way which is not treated in one single rule, we describe template recognition

explicitly. Let us first have a look at the rule `template_declaration`:

```
template_declaration
  : TEMPLATE '<' template_parameter_list '>' declaration
  ;
```

Listing 3.43: `template_declaration` Rule

When the parser starts parsing this rule, we know that there will be a template declaration. Unfortunately, that is not very much knowledge. We are completely lacking information about template parameters and whether the template is a function or a class template. The declaration of these elements effectively happens in the sub rules `template_parameter_list` and `declaration`. This includes defining them in the currently active scope, in the symbol table. Subsequently, we need a solution to retain the symbol handling of the sub rules but prevent them to be declared in the current scope, that is active when entering `template_declaration`. For the template parameter we could open a new scope which catches the declared parameters. But this would not solve another major problem in the `declaration` rule. As when parsing the `declaration` of a template class, it already has to be known as a template. Therefore, a template with the corresponding name has to be resolvable at the point entering the template class body. To cope with both problems we have taken the following approach:

```
template_declaration
  : TEMPLATE {context.declareTemplate();}
    '<' template_parameter_list {context.finishTemplateParameterScope();} '>'
    declaration {context.finishTemplate();}
  ;
```

Listing 3.44: `template_declaration` Rule with Actions

As soon as we know that there is a template, we declare it - we could also say: we announce a template. This adds a `TemplateSymbol` to the current scope in the symbol table. The `TemplateSymbol` is a decorator for another declaration, which it does not contain at that point. According to the rule, the template parameters are parsed next. Thus, the `TemplateSymbol` is in a state where it expects template parameters, when new symbols are defined in its scope. All parameters declared in `template_parameter_list` are added to the template parameters in the `TemplateSymbol`. After we have finished parsing the template parameters, we close the parameter declaration section with `finishTemplateParameterScope`. After that, the `TemplateSymbol` expects the declaration to be decorated as a template. Therefore, the next symbol declaration in `TemplateSymbol` is added as the to-be-decorated declaration. Internally the template registers itself, after the declaration is added to the symbol table, in the surrounding scope, with its known name. After the declaration has been parsed, the template is closed

with `finishTemplate`. Further declarations before the `finishTemplate` call cause an error, as this should syntactically not be possible. The biggest advantage of this solution is the possibility to resolve the name of the template inside the template declaration as an identifier both with and without template arguments.

3.4.5. Forward References

In Section 3.2.4 we have already mentioned that, in member function definitions bodies, symbols of the surrounding class can be referenced before the parser passes their declaration.

```
struct S{
  void foo(){
    I i;
  }
  typedef int I;
};
```

Listing 3.45: Forward Reference Example

When parsing advances straight forward, the type `I` cannot be known when the declaration of `i` is encountered. The symbol table and name resolution described before cannot handle such code the way described. Subsequently, we need special handling of such cases. Below we describe three possible approaches to make the parser recognizing member function bodies correctly.

Post-Parse Verification

As mentioned at the beginning of this chapter, Terence Parr has described a way to deal with programming languages containing classes in [Par09b]. In two passes, first the declarations are parsed, second the references are resolved. To have a similar procedure in C3P0 we could parse the corresponding code without performing semantic checks and verify these semantic conditions after parsing has been completed. Following this approach would require us to change the rules responsible for recognizing member functions and their bodies. Unfortunately, it is a majority of the rules that can be reached while parsing the member function body. Subsequently, all these rules required either further version, which do not check the context, or an option to disable the checks. Even the latter would take considerable effort regarding grammar changes. Furthermore, we would not be able to disambiguate every statement without the required information about existing types.
We have decided to follow the one-pass parse approach, which basically makes Parr's pattern infeasible for C3P0. If we would implement forward referencing in this way it would most likely make sense to implement all semantic checks according to the approach described in [Par09b].

Forward Lookup

In [Par09b] Parr mentions another possibility to handle classes while parsing. An alternative approach could be to perform resolution on the fly when required, which implies some sort of lookahead. Parr does not elaborate on this approach but we expect it to work as follows.
If, in a member function body a rule gated with a certain semantic predicate is encountered, first the symbol table is checked as mentioned above. A symbol previously defined can be resolved as usual and no special treatment is required. On the other hand if a symbol cannot be resolved successfully, there must be some sort of lookahead. We expect this mechanism to become quite complex as the possible structures of such declarations consist of the complexity already implemented in the parser. But, as this lookahead must happen in the middle of the parse process it either requires to re-implement much of this resolution logic in the symbol table. This could be achieved by implementing a further resolver, consisting of its own rule set. Alternatively, we could jump out of the current context into the surrounding class, continue parsing at that point until all possible declarations are resolved. After that we could continue with the previously unresolvable predicate.
Jumping around in the code for deciding on specific rules might become quite difficult to follow and required much logic for maintaining the sections already parsed, as it would be desirable not to parse the same part several times. We expect this approach would be easier to implement if, like in the other approach described above, there already existed a complete AST, for syntactic representation, that can be crawled for subsequent declarations.

Deferred Parsing

As the previously described ideas for resolving forward references either are not feasible for our approach or rather laborious to implement, we have decided to implement a different forward reference strategy. Instead of two passes and forward lookup we defer parsing of regions affected by forward references. The approach is quite simple: If we encounter a member function definition in a class, represented by a certain alternative in the `member_declaration` rule, we just skip the body, which could contain the forward references and continue parsing after the function definition. Code that is not parsed cannot break the recognition. While an advantage on correct C++ code, it is inconvenient not to have such parts parsed at all. Therefore, we parse these blocks after the class definition has been parsed successfully. As at that point all declarations that could be referenced before in the code, have been passed and the corresponding symbols are known.
For implementing this feature, we have added some methods to the `C3P0Parser` and changed some rules. Our alternative in `member_declaration` to recognize member functions, needed an action that skips the body in the input stream. To make the rule more concise we have extracted the whole alternative into a separate sub rule. The result is the following `member_function_declaration` rule:

```
member_declaration
  ...
  | (member_function_declaration) => member_function_declaration
  ...
  ;

member_function_declaration
@after{context.leaveScope();}
  : attribute_specifier? function_signature
    ( ('{' | 'try') { skipMemberFunctionBody();}
      | ( '=' ( DEFAULT | DELETE) ';'))
  ;
```

Listing 3.46: Body Consuming Action for Member Functions

After matching `function_signature` we come to the part possibly containing forward references. It can either be introduced with an open curly brace (`'{'`) or the keyword `try`. Although we know, that after matching `function_signature` there is always a to-be-skipped body, if not having an equal sign `'='` as the next token, we cannot leave these two tokens out. Otherwise, the parser would not know that we are consuming some of the context, there occurred ambiguities if we left `('{' | 'try')` away. Subsequently, this has to be considered in the consuming method `skipMemberFunctionBody`. We still have to leave the scope, opened in the `function_signature`, after parsing this rule. Similarly, the alternatives for constructors and destructors have to be adapted as well. Completely avoiding member function bodies in this way, at least makes our test case from the beginning of this section turn into a green bar test. Unfortunately, we cannot detect syntax or semantic errors in the code skipped. Thus, we want to parse it after the region for possible declarations (the surrounding classes) has been passed. To invoke these checks we alter the `class_specifier` rule. Beyond adding the action for parsing the skipped member function bodies, we also need to track the nesting level of the classes in the parser. We will come to this later when explaining the implementation in the `C3P0ParserImplementation`. Currently, we just need to know that the `ClassSymbol` to perform the resolution in, has to be known when parsing the member function bodies. We have stripped the rule below, from actions regarding the symbol table, to emphasize the new actions:

```
class_specifier
  : head = class_head
    {...}
    {openMemberSpecification($head.classNode);}
    '{' member_specification? '}'
    {closeMemberSpecification(); parseMemberFunctionBodies();
    ...}
  ;
```

Listing 3.47: Changed `class_specifier` Rule

The implementation of the new actions in the rules above, are described below:

skipMemberFunctionBody This method creates a new task for parsing the member function body. It contains a marker from the input stream, pointing at the position where the body starts and the context for parsing the body, which corresponds to the surrounding class. And it consumes the rest of the function body from the token stream.

```
public void skipMemberFunctionBody() {
  DeferredParseTask deferredTask = new DeferredParseTask(input.mark(),
                                                         classNestingLevel.getLast());
  memberDefinitionsToParse.addLast(deferredTask);
  consumeFunctionBody();
}
```

Listing 3.48: Implementation of `skipMemberFunctionBody`

parseMemberFunctionBodies This method performs the post-class paring of the member function bodies. As parsing these function parts can throw recognition exceptions, this method can do this as well. Not catching this exception is okay, as the calling method (the rule `class_specifier`) is a possible origin of this exception as well. The member bodies are only parsed if we are at the outermost class definition.

It is mandatory to collect these tasks until the last enclosing class definition, because in member functions forward lookup can also access members of surrounding classes. Therefore, a possible declaration could probably not be parsed at the end of a nested class. Furthermore, before parsing the member function bodies, we need to mark the current position in the token stream to return to the current position after parsing the member function bodies.

```
public void parseMemberFunctionBodies() throws RecognitionException {
  if (classNestingLevel.isEmpty()) {
    int endMark = input.mark();
    while (!memberDefinitionsToParse.isEmpty()) {
        parseMember(memberDefinitionsToParse.removeLast());
    }
    input.rewind(endMark);
    input.release(endMark);
  }
}
```

Listing 3.49: Implementation of `parseMemberFunctionBodies`

parseMember This method configures the context for parsing the member function body and invokes the rule for parsing the function body. First, we enter the

class scope surrounding the member function body to be parsed. Then, the token stream is moved to the marked position, pointing at the body. The input has to be adjusted by one, as the parser has to match the first token of the body before. When at the correct position in the token stream, the method corresponding to the `function_body` rule can be called. As ANTLR generates such a method for every rule, we do not have any further effort for recognizing the body. After parsing the member function body, we step out of the class context and release the marker of the input stream as it is not used anymore.

```
protected void parseMember(DeferredParseTask deferredParseTask)
    throws RecognitionException {
  int mark = deferredParseTask.streamMark;
  context.enterLocalLookupScope(deferredParseTask.parseContext);
  input.rewind(mark);
  input.seek(input.index() - 1);
  function_body();
  context.leaveLocalLookupScope();
  input.release(mark);
}
```

Listing 3.50: Implementation of `parseMember`

This approach seems to work well and even more complex test cases including nested classes and forward references can be recognized correctly. Furthermore, it is conceptually quite simple and comprehensible. The implementation was easy and more or less straight forward. We just had one little side-effect while taking the `function_body` part out of the member function alternative. As we have syntactically simplified the whole `class_specifier` rule, our template declaration recognition got scrambled. The problem origins from the rule `decl_specifier`, which is used to specify several types of declarations, including function declarations. While syntactically in the rule set it does not depend on the concrete declaration, effectively not all specifiers are allowed for every declaration. For example a `class_specifier` or the keyword `typedef` shall not be part of a `decl_specifier_seq` in a function declaration, where it specifies the return type. But with the syntactical relief in `class_specifier` this became a possible match, which turned every template class declaration into a template function declaration. Eventually, resulting in failing tests. We engaged this issue by refining the `decl_specifier` for function declarations, by introducing the `function_decl_specifier`:

```
function_decl_specifier
  : STATIC |  EXTERN
  | trailing_type_specifier | function_specifier
  | FRIEND |  CONSTEXPR
  ;
```

Listing 3.51: New Rule `function_decl_specifier`

This change fixed the recognition of template classes and improved the overall recognition quality, as previously accepted qualifiers like `REGISTER` are not allowed for functions anymore; which is correct.

3.5. Deficiencies

While our approach of including the symbol table into the parser seems to work fine, we did not have enough time to implement and test all possible cases that would rely on the symbol table. Nevertheless, we are convinced more semantic context can be integrated into the parser with our approach as described above. In this section we collect the cases which we know the current implementation has lacking recognition capabilities for.

Tracking Data Fields

While the symbol table has a type defined to store a data field, we do not have the actions in the parser to track them. As the parser currently is focused on types to disambiguate its rules, we do not necessarily need to manage data fields.
This implies the following disadvantages:

- Local variables hiding a type are ignored and the hidden type can be resolved, although it should not. This might lead the parser to take the wrong rule. Letting the recognition fail, for a correct program or just having the parser recognize the code with the wrong rule and eventually generating an inappropriate AST.
- Access to members and local data fields is not checked for correctness. As long as we do not track them, we cannot implement the semantic predicates to check if they exist.
- Even if the symbol table would contain the functionality to deduce types of expressions, we also needed to track all entities to be able to derive them correctly.

Restrictive Syntactic Predicates

The incremental approach of developing the parser rules lacks slightly of the big picture of the whole rule set. Subsequently, syntactic predicates can become a bit too specific. For example, it is not necessary to include the whole `class_specifier` as a syntactic predicate for an alternative `class_specifier`. We already know, that in a well-formed C++ program a `class_head` followed by a `'{'` can only match a `class_specifier`, which suffices as a syntactic predicate.

```
type_specifier
  : (class_head '{') => class_specifier
  ...
  ;
```

Listing 3.52: `type_specifier` Rule

Optimizing such predicates improves performance of the parser. Sometimes shrinking the predicates can even avoid performing complex context switches for name resolution as explained before.
Probably, there are further predicates which can be optimized similar to the predicate for `class_specifier`.

Missing Lookup Scenarios

Some cases described in 3.2 have not been implemented yet. For example, we do not have an augmented scope for lookup in friend declarations. Furthermore, resolution when defining static data members does not perform lookup in the scope it is a member of and forward lookup is only possible for member function, constructor and destructor definitions.
Similar to variables and constants, enumerators are not tracked in the symbol table. Fortunately, this does not let the recognition fail, but currently we cannot perform a check for existence of the enumerator.

Inline Namespaces

In C++0x it is possible to optionally specify a namespace as inline. This maps the declarations of the inlined namespace into the surrounding namespace as well. Currently, we do not cover this functionality.

3.6. Conclusion

The first milestone of the master thesis has been completed successfully. We could continue the approach to implement a C++0x parser using ANTLR, started in the term project. We have replaced the dummy symbol table, providing a monolithic scope, with a more powerful scope-oriented implementation. It is able to track different types of symbols and can hide or augment scopes, depending on the context. Resolution dynamically adapts according to the active scope and even qualified names can be resolved. At the current state, the symbol table is not a perfect representation of the C++ standard and the currently known issues have been described in 3.3.5.
The symbol table has been integrated into the parser. It was possible to figure out an approach to handle the known problems considering syntactic predicates and action execution. For testing our implementation we have continued to extend our unit test suite, which contains almost 950 test cases and, furthermore, we are able to recognize a complete preprocessed hello world application, containing over 14'000 lines of C++0x code.
We have developed a simple but correct way of recognizing class parts that contain forward references correctly. It was possible to implement this approach without changing the general parse strategy – we can still stick with our one-pass approach – and do not require complex lookahead techniques.
There are cases remaining which cannot be recognized yet, but there are no known obstacles which prevent the implementation of this recognition completely. By continuing our approach, it should be feasible to create a parser recognizing C++0x at a satisfying precision level. Nevertheless, this implementation will take a lot of further time and more thorough testing.
Knowing that the symbol table and its integration is not a complete show stopper, we can continue focusing on the next critical task: Handling of preprocessor statements.

4. Preprocessing

The second milestone of this master thesis examines handling of preprocessing directives in C3P0. So far, we have been able to show solutions for problems expected while parsing already preprocessed C++ code. In this chapter, we describe the challenge when dealing with unpreprocessed code, containing preprocessing directives.

4.1. Requirements

The requirements to a preprocessor seem to be quite obvious: It should be able to preprocess C++ source code. But, our requirements reach beyond that. For creating an AST in the parser we need exact source position information. Although the `C3P0Lexer` already adds position information to the tokens created, these positions are not appropriate for our purpose. When creating an AST for performing automated refactorings on, we require position information about the original locations. For every token we must be able to figure out where it came from, including the source file and exact location therein. After just performing the replacement of the preprocessing directives, one usually has a large stream of tokens or characters, which have no direct relation to their original source.
Others have been dealing with the same problem. In [OMJ09] Overbey, Michelotti and Johnson have presented an approach to deal with the problem of representing preprocessed code when refactoring. Their focus is to retain the possibility to get back to the representation of the original source from the AST representation only. For retaining all information about the token origins, they have introduced pseudo-preprocessing, which replaces certain directives and keeps information about the replacement of the source code elements. This is similar to what we need as well. We have to figure out how close we need be to their approach, as currently the `C3P0Lexer` expects a completely preprocessed source stream.

4.2. Preprocessing Directives

There are several distinct types of preprocessing directives. These directives begin with a hash token (`#`), that must be the first token of a source line. The end of that line terminates the preprocessing directive. It is possible to span such a directive to several lines by putting a backslash (\) at the end of the source line.
In this section we will describe the different preprocessing directives and explain how they work.

4.2.1. Conditional Inclusion

It is possible to enable and disable code parts of the source code, depending on a condition, using the conditional preprocessing directives: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` and `#endif`.
It is possible to have several conditional inclusions nested, for which the syntax is free of ambiguity, as every `#if` has to be properly closed with an `#endif`. The interesting part about this preprocessing directive is the condition after `#if` and `#elif`. According to the standard [Dra10a] this condition is represented by a `constant-expression`, further specified to be an integral constant expression. This includes a set of additional rules of the standard ([expr.const]). Additionally, we have a major change in C++0x: Functions can be declared as `constexpr`, which makes them a possible part of a constant expression. This yields a challenging question: How can we resolve a function call to a `constexpr` function if we have not parsed any code at that point? As the standard has not been finished yet, we are expecting to get some refinement considering this specific case. We currently do not expect calls to `constexpr` functions to be a valid part of an integral constant expression during preprocessing, otherwise the translations unit steps get mixed as we are depending on later tasks (the recognition of function declarations) in earlier ones (preprocessing).
In either case, with or without `constexpr` functions as part of these constant expressions, it will be challenging to decide on all possible conditions. Precedence will follow the same rules as we have already implemented in C3P0 for expressions in general. But recognizing and structuring this expression will not suffice. We need to evaluate it for deciding on the conditionally included part.

```
#if defined NUM      //outer #if
#if NUM < 0          //inner #if
...
#else                //inner #if
...
#endif               //inner #if
#else                //outer #if
...
#endif               //outer #if
```

Listing 4.1: Nesting of Conditional Inclusion

4.2.2. Source File Inclusion

The `#include` directive allows to import other source files into the current file. A source file specified is inserted at the location of the directive. Usually, included files have a so called include guard, using the conditional inclusion to prevent duplicate definitions of symbols. For avoiding endless loops, which occur if the include guard is missing in mutually including files, there can be an implementation defined limit for the depth of inclusion levels.

There are three possibilities how an include directive can look like:

```
# include " q-char-sequence " new-line
# include < h-char-sequence > new-line
# include pp-tokens new-line
```

Listing 4.2: Syntax for Source File Inclusion

While the first two definitions are familiar to most C++ programmers, the third might look a bit confusing. We can encounter `include`s which do not directly have specified a file name. As the tokens following the `include` keyword, are target to macro replacement, described below, it is possible to have placeholders for filenames there. Nevertheless, after performing macro replacement, the `include` directive must have one of the first two forms.

4.2.3. Macro Replacement

There are two types of macros: *object-like* and *function-like* macros. Object-like macros specify a replacement of a certain macro name, represented by an identifier. This replacement will be inserted at the positions where that identifier occurs, after the definition. Function-like macros also define such a replacement, but they, additionally, contain parameters which get substituted by arguments in the replacement. Both macro types are specified using the `#define` directive. There is only one namespace for such macro definitions. It is also possible to undefine a macro using the `#undef` directive.

Object-like Macros

The syntax for object-like macros is quite simple:

```
# 'define' identifier replacement-list new-line
```

Listing 4.3: Syntax for Object-like Macros

The identifier denotes the name of the macro to be be defined. Every further occurrence is replaced by the tokens of the `replacement-list`.

```
#define ANSWER 42

int answerToTheUltimateQuestionOfLifeTheUniverseAndEverything(){
  return ANSWER;
}
```

Listing 4.4: Object-Like Macro Example

After preprocessing, the resulting C++0x code will look as follows:

```
int answerToTheUltimateQuestionOfLifeTheUniverseAndEverything(){
  return 42;
}
```

Listing 4.5: Object-Like Macro Example After Replacement

After replacing such an identifier, the replacement-list is checked for further macro replacements, including the rest of the source file. We emphasize that this rescan happens after a replacement and cannot be performed when the macro definition is encountered for saving processing time. If we would replace a macro in the replacement list, we would change the behavior. Look at the example below, which would have a different result if we replaced `A` in the replacement-list of the macro `B` while scanning the macro definition itself:

```
#define A 1
#define B A     // B will subsequently be replaced by A, not by 1!

#undef A

#if (B == 1)    // (B == 1) will become (A == 1), evaluating to false,
...             // as A becomes 0 in a condition if undefined.
#endif
```

Listing 4.6: Replacement Example

Function-like Macros

Very similar to functions, a macro definition can have parameters. These parameters can represent placeholders in the replacement list. The syntax for function-like macros is as follows:

```
# 'define' identifier '(' ident-list? ')' replacement-list new-line
# 'define' identifier '(' '...' ')' replacement-list new-line
# 'define' identifier '(' ident-list ',' '...' ')' replacement-list new-line
```

Listing 4.7: Syntax for Function-Like Macros

Every subsequent occurrence of the function-like macro name followed by opening parentheses, is replaced by the `replacement-list`. This is comparable to inlining the function defined by the macro name. So far, the replacement is similar to object-like macros. Additionally, the arguments passed to the macro invocation replace the corresponding refer-

ences to the parameters in the replacement list. Before argument substitution happens, the arguments are macro-replaced. The following example shows argument substitution in a function-like macro:

```
#define DOUBLE(a) 2 * a

int four = DOUBLE(2);
```

Listing 4.8: Function-Like Macro Example

The macro `DOUBLE` is replaced by its `replacement-list`, and the argument is inserted at the corresponding place of the parameter `a`. The argument `2` itself is unaffected by the macro replacement.

```
int four = 2 * 2;
```

Listing 4.9: Function-Like Macro Example After Replacement

The next example shows the behavior if a macro is passed as an argument. The argument gets macro-replaced (`DOUBLE(1) => (2 * 1)`) before the corresponding parameter (`a`) is substituted:

```
#define DOUBLE(a) (2 * a)

int four = DOUBLE(DOUBLE(1));  //expands to: int four = (2 * (2 * 1));
```

Listing 4.10: Argument is Macro-Replaced First

In C++0x, preprocessing has new features as well. In the new standard it is possible to have variable argument-lists in function-like macros. The variable parameter list is represented by the ellipsis (`...`). Variable arguments are accessed in the replacement list by `__VA_ARGS__`. All arguments exceeding the named arguments, before the ellipsis, are inserted at this placeholder. This includes the commas, separating the arguments and whitespaces, which get reduced to one single space. The following example illustrates the use of variable arguments in function-like macros:

```
#define DECLARE(type, ...) type __VA_ARGS__

DECLARE(int, i, j, k); //expands to: int i, j, k;
```

Listing 4.11: Variable Argument Example

It is possible to convert the tokens of an argument into a string literal, by putting the hash operator (#) before the corresponding parameter:

```
#define PRINT(expr) cout << "Result of '" << #expr << "' is: " << expr << endl

PRINT(1 + 2);
```

Listing 4.12: Argument to String Conversion

The preprocessor does not expand arguments that are converted into a string literal. After converting the argument into a string literal and substituting the parameter (expr), the resulting code looks as follows:

```
cout << "Result of '" << "1 + 2" << "' is: " << 1 + 2 << endl;
//Output: Result of '1 + 2' is: 3
```

Listing 4.13: Argument to String Conversion After Substitution

The ## operator can be used to concatenate two tokens of a replacement list together:

```
#define DEF_INT(arg) int var_##arg = arg

DEF_INT(1);  //Results in: int var_1 = 1;
```

Listing 4.14: Concatenation Using the ## Operator

As we have stated above, if a macro has been replaced, the inserted replacement-list is rescanned for further macros to replace. For avoiding endless loops, a macro, currently being replaced, is not replaced if encountered during this rescan. This also holds when transitively reaching the macro. The following example illustrates the behavior:

```
int x = 1;
#define x x + 2
foo(x); //expands to: foo(x + 2);

int a = 1;
#define a b + 1
#define b a + 2
foo(a); //Step 1: foo(b + 1);
        //Step 2: foo(a + 2 + 1);
```

Listing 4.15: Recursive Replacement

The definition of a macro lasts until the #undef directive of the corresponding macro.

4.2.4. Line Control

The `#line` directive has the purpose of setting file and position information for allowing a compiler generate expressive messages related to the correct position. Since we are not interested in any modifications of such position information, and consider the real files and offsets, this directive has no further impact on the positions in our AST, as long as we create it for refactoring purposes. We could implement a check for the syntax of this directive and emit an error or warning if it is erroneous:
The `#line` directive constructs as follows:

```
# 'line' digit-sequence new-line
# 'line' digit-sequence " s-char-sequence " new-line
# 'line' pp-tokens new-line
```

Listing 4.16: Syntax for the `#line` Directive

The `digit-sequence` defines the source line offset, the number shall be between 1 and 2'147'483'647 . The `s-char-sequence` defines the name of the source file to be expected in. Similar to the `#include` directive, the `#line` directive is subject to macro replacement. After this replacement has taken place, the resulting sequence must correspond to one of the first two definitions for the `#line` directive.

4.2.5. Error Directive

The `#error` directive emits an error while preprocessing. For example, it can be used if the environment fails to satisfy certain constraints. We must not necessarily stop the whole preprocessing process, when encountering such a directive, but we should inform the user about the error. With the integration into CDT, a warning or error could be added in the editor, when encountering a reachable `#error` directive.

```
# 'error' pp-tokens? new-line
```

Listing 4.17: Rule for the `#error` Directive

The optional pp-tokens compose the error message, possibly providing more information about the error for the user.

4.2.6. Pragma Directive

The `#pragma` directive specifies implementation defined behavior. In the first place we can ignore such directives, as it probably does not make sense to provide C3P0 specific `pragma` options. It will be necessary to reconsider dealing with `#pragma` if we adapt our implementation to existing compilers, for extended compatibility.

```
# 'pragma' pp-tokens? new-line
```

Listing 4.18: Rules for the `#pragma` Directive

4.2.7. Null Directive

The null directive is the hash token, followed by a `new-line` token. It simply does not have any effect.

```
# new-line
```

Listing 4.19: Rules for the Null Directive

4.2.8. Predefined Macros

There are some macros, which shall be predefined:

`__cplusplus` The specific value has not been defined for the new C++ standard yet.

`__DATE__` The date when preprocessing started, in the form `"Mmm dd yyyy"`. For example: `"Apr 12 2010"`

`__FILE__` The file name currently being preprocessed. This value depends on the `#line` directive.

`__LINE__` An integer representing the source line in the file currently being preprocessed. This value depends on the `#line` directive as well.

`__STDC_HOSTED__` Is set to 1 if the implementation is a hosted implementation. A hosted implementation supports the complete standard, including the standard library [GCC10].

`__TIME__` The time when preprocessing started, in the form `"hh:mm:ss"`. For example: `"15:51:30"`

`_Pragma(string-literal)` The pargma operator is similar to a function-like macro taking one argument, which has to be a string-literal. It ends up representing a `#pragma` directive, which we will ignore in the first place.

4.3. Implementation

This section describes the implementation of C2P0 (C++-PreProcessor-for-C++0x). To have a consistent solution when combining C2P0 and C3P0 we decided to implement the preprocessor in ANTLR as well. We have considered to extend an existing preprocessor by the features of C++0x and our requirements to position information. On the ANTLR website a C preprocessor has been published [KU06].
While analyzing this preprocessor we have made the following observations:

- The grammar delivered in the `tar` archive, had to be fixed syntactically, before the lexer and parser could be generated using ANTLR v3.2 [Par09a] (the current release).
- Running that preprocessor failed in some cases with a `StackOverflowError` when preprocessing recursive macros, like `#define f f`. This also included the example in paragraph 5 from the [cpp.scope] section of the C++ standard draft [Dra10a], as it includes recursion as well.
- Several parts of the source code and the grammar had been commented out and some actions in the grammar are up to 150 lines of code long.
- The C preprocessor on the ANTLR website [Par09a] did not contain any license information. Subsequently, we could not determine whether we were allowed to use it for our purposes or if there might be a conflict with EPL.

Eventually, we decided not to extend this preprocessor. It would have taken much effort to bring it into shape and we did not know whether we were allowed to use it under the EPL conditions. Furthermore, we have the following advantages when implementing C2P0 ourselves:

- The rules for implementing the lexer are pretty close to those of the C3P0Lexer. Therefore, we can reuse the existing lexer rules and add the preprocessing directives.
- Having a similar lexical analysis in the preprocessor and the C++0x parser, eases the concatenation of the two applications. Currently, we stick with a character-oriented representation as output of the preprocessor, which enables the preprocessor to be tested and used independently. Directly passing the AST, generated in C2P0, to C3P0 might be an option for future improvements.
- We can focus on our requirements to the preprocessor and do not have to fit them into existing structures.
- There will not be issues regarding legal aspects, as we can assure we have completely implemented the preprocessor ourselves.

Alternatively, we could have had a look at the preprocessor used in NetBeans, also published on the ANTLR website [Par09a]. But this would be impairing regarding legal aspects, as described in [Cor10a].

4.3.1. C2P0 and C3P0 Translation Steps

Getting from a set of source files to an abstract syntax tree takes several steps. In Figure 4.1 these steps are illustrated:

1. The preprocessor (C2P0) takes a set of source files as input. The output is preprocessed source code. Beside the common task of dealing with preprocessing directives, our preprocessor creates a position map. This position map can be used to determine the exact original position of the source code elements in the output, including source file, line and offset.

2. The (C3P0) lexer creates a token stream, representing the preprocessed code, for the parser.

3. The tokenized source is processed by the parser and transformed into a CDT-AST. The position map of the preprocessor can be used to deduce the original source position for every AST node.

Compared to the approach of representing preprocessed source code for refactoring by Overbey, Michelotti and Johnson [OMJ09], we have two main differences. First, we will handle the mapping of the source position in the parser. They have inserted an additional processing step between the lexer and the parser, called *Directive Recovery*. There they modify the tokens from the lexer by adding substitution and whitespace information retrieved from the preprocessor. As we do not intend to fully reproduce the original source files using our AST, we can omit most of this information and only require a mapping of the source positions. Second, our preprocessor consumes all preprocessing directives, as C3P0 is implemented to handle fully preprocessed code. While Overbey et al. only expand trigraphs, macros and file inclusions and pass other directives, like `#error` and `#line`, to the lexer. We can collect them in the preprocessor if necessary and add them to the AST afterwards.

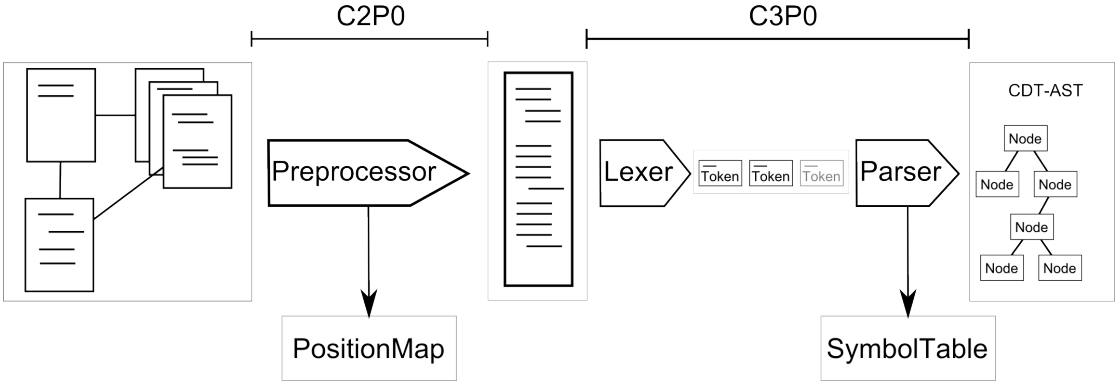

Figure 4.1.: Translation Steps

4.3.2. Components Overview

Terrence Parr in [Par09b] suggested to use a parser, generated using ANTLR, to create an AST and then walk this AST using tree grammars, as a generic approach when analyzing languages. As we had not been aware of this approach in the C3P0 term project we sticked with a more classical implementation to perform the tasks of the parser in actions, embedded into the parser rules. This is more efficient, but bloats the grammar rules with code. For getting familiar with Parr's approach, we decided to implement the preprocessing functionality using tree grammars. A tree grammar is used to walk an existing AST and to perform actions on specific nodes and/or to modify the structure of the AST, or to create a new one respectively.
For avoiding confusion about the different abstract syntax trees, in this chapter we will refer to the tree representation generated by ANTLR parsers, that can be walked using tree grammars as AST. If we refer to the abstract syntax tree that we finally create to be used in CDT, we will refer to it as CDT-AST. There is a detailed description of our ANTLR AST in Section 4.3.4
C2P0 is split up into the following components:

C2P0Lexer The lexer for the preprocessor is very similar to the C3P0Lexer. It consists of a similar rule set, augmented with rules for scanning preprocessing directives. The C2P0Lexer is generated with ANTLR, from the lexer rule set in the `C2P0.g` grammar.

C2P0Parser The tokens emitted by the lexer are structured in AST nodes by the parser. The generated AST represents the preprocessed input file. The rule set of the parser uses rewrite rules, which determine the parent relation of the AST nodes. The C2P0Parser is generated with ANTLR, from the parser rule set in the `C2P0.g` grammar.

C2P0Printer In the C2P0Printer, the AST, created by the C2P0Parser, is walked. It handles the preprocessing directives and creates the preprocessor output. The C2P0Printer is implemented as a tree grammar. It is generated with ANTLR from the rules in the `C2P0Printer.g` tree grammar.

C2P0GroupExpander A further tree grammar to scan groups of tokens for all macros and expanding them if necessary. It is generated with ANTLR, from the rules in the `C2P0GroupExpander.g` file.

C2P0MacroExpander Function-like macros are a bit more complex than a simple replacement of the name. In the `C2P0MacroExpander` parameters are substituted and arguments are macro-replaced before they are inserted. It is also implemented as a tree grammar. The C2P0MacroExpander is generated with ANTLR, from the rules in the `C2P0MacroExpander.g` grammar file.

C2P0ConditionEvaluator The fourth tree grammar, helps to determine the value of an integral constant expression used in a conditional inclusion. It is generated with ANTLR, from the rules in the `C2P0ConditionEvaluator.g` tree grammar.

4.3.3. C2P0Lexer

Like the C3P0Lexer, the C2P0Lexer is generated with ANTLR. Its rule set is defined in the corresponding grammar file, `C2P0.g`. A description of the C3P0Lexer rules and the structure of ANTLR grammar files can be found in [Cor10a]. In this section we will focus on the differences compared to the C3P0Lexer and describe the recognition of the preprocessing directives.

Output Tokens

The C2P0Lexer generates the following output tokens, additionally to the tokens known from C3P0:

- `IF_DIRECTIVE`
- `IFDEF_DIRECTIVE`
- `IFNDEF_DIRECTIVE`
- `ELIF_DIRECTIVE`
- `ELSE_DIRECTIVE`
- `ENDIF_DIRECTIVE`
- `INCLUDE_DIRECTIVE`
- `ERROR_DIRECTIVE`
- `DEFINE_DIRECTIVE`
- `UNDEF_DIRECTIVE`
- `PRAGMA_DIRECTIVE`
- `LINE_DIRECTIVE`
- `HASH`
- `DOUBLE_HASH`
- `NEW_LINE`

User Defined Literals

In C++0x, programmers have the possibility to define their own suffixes for literals. The implementation of such user defined literals in C3P0 is described in the documentation of the term project [Cor10a]. User defined literals have no meaning for a preprocessor, as they need to be combined with the corresponding operator to be interpreted. The separate handling of user defined literals implies some unnecessary complexity.
Lexically a user defined literal is always a common literal with a trailing `ud-suffix` [Dra10a]. For example, a user defined string literal has the following composition according to the standard:

```
USER_DEFINED_STRING_LITERAL
  : STRING_LITERAL UD_SUFFIX
  ;

UD_SUFFIX
  : IDENTIFIER
  ;
```

Listing 4.20: Rule for User Defined String Literal

The `UD_SUFFIX` represents an `IDENIFIER`. Subsequently, `"ud_str"suf` can also be recognized by the rules `STRING_LITERAL` and `IDENTIFIER` separately. Since user defined literals just occur as tokens that are passed through the preprocessor, we can simplify our

C2P0Lexer grammar to be unaware of user defined literals. This does not have any further implications. If whitespace-handling works correctly, every user defined literal will result in the corresponding representation, recognizable by the C3P0Lexer. Therefore, we have removed user defined literals from the C2P0 lexer rule set.

C++ Keywords

Opposed to a language-oriented parser, like C3P0, the preprocessor does not need to know about C++ keywords. For the preprocessor they are just `IDENTIFIER`s. All rules of the C3P0Lexer grammar regarding C++ keywords have been removed. `struct`, `class`, etcetera then are matched by the `IDENTIFIER` rule and treated as such. As a consequence, they can also be used as macro names. Subsequently, they will be replaced during macro expansion.

Preprocessing Directives

The preprocessor does not care about C++ keywords, but it has to recognize the preprocessing directives, listed below. For each directive recognized, a specific token has been defined:

- `if`
- `else`
- `line`
- `ifdef`
- `include`
- `error`
- `ifndef`
- `define`
- `pragma`
- `elif`
- `undef`
- The empty directive

According to the C++ standard ([Dra10a] [cpp]/1), the identifiers above are only recognized as a preprocessing directive if they have a preceding hash token (`#`) at the beginning of a file or following a new-line character. Therefore, a preprocessing directive cannot occur in the middle of a source code line. To decide whether a potential preprocessing directive is valid, we have introduced a semantic predicate, which enables or disables the rules dedicated for matching directives. For example, we have the rule for matching the `if` directive below:

```
IF_DIRECTIVE
@after{
  expectingDirective = false;
}
  : {expectingDirective}?=> 'if'
  ;
```

Listing 4.21: C2P0 Lexer Rule for `if` Directive

If the boolean `expectingDirective` is `true`, this rule can match. If not, `if` is matched as a common `IDENTIFIER`. It is necessary to have this distinction. Otherwise every `if` in the C++ code would be matched as an `if` directive. That would not be desirable, as it would change the behavior of the tree walkers in the preprocessor.
From the description above, we know that a preprocessing directive is only expected after a hash token (`#`) iff that hash token follows a new-line token or is at the beginning of a file. Intermediate whitespace characters between the hash and the new-line token are allowed. Thus, `expectingDirective` has to be set to `true` if we encounter a hash token satisfying this condition:

```
HASH
@after{
  if(lastVisibleToken == null || lastVisibleToken.getType() == NEW_LINE) {
    expectingDirective = true;
  }
}
  : '#'
  ;
```

Listing 4.22: C2P0 Lexer Rule for Hash

To have access to the last token emitted by the C2P0Lexer, we have to track it ourselves, as ANTLR does not provide access to tokens emitted previously. We can catch the output tokens by overriding the `nextToken` method of the `Lexer` class. If it is a visible token, we store it in the `lastVisibleToken` field. Visible tokens are all tokens, that are not on the `HIDDEN` channel.

```
@lexer::members {
  Token lastVisibleToken = null;
  public Token nextToken() {
    Token currentToken = super.nextToken();
    if(currentToken.getChannel() != HIDDEN){
      lastVisibleToken = currentToken;
    }
    return currentToken;
  }
}
```

Listing 4.23: `nextToken` Method in C2P0Lexer

Line Splicing

According to the C++ standard [Dra10a], in the second translation phase, lines ending with a backslash (\) are spliced together. Currently, we do not behave exactly like the standard demands. As such an escaped new-line could actually occur everywhere in the code, even identifiers or literals could be split to several lines. To implement

this possibility in the lexer we would need to add the sequences for escaped newline characters (like: '\\\n') optionally between any two characters of every token. This would make the grammar unreadable. As we do not have a generic solution yet, we have explicitly implemented line splicing for string literals, line comments and beyond token boundaries. At least if line splitting is not performed automatically, by a tool, at a specified line width, we expect to have the most common cases, programmers do by hand, covered. To handle line splicing in general, we have specified the following rules:

```
SKIPPED_NEWLINE
  : ESCAPED_NEWLINE {$channel = HIDDEN;}
  ;

fragment
ESCAPED_NEWLINE
  : BACKSLASH NEWLINE
  ;
```

Listing 4.24: Rule to Handle Line Splicing

The tokens matched by this rule are sent to the hidden channel, which means they will not be visible to the parser when creating the AST. As long as they are not accessed explicitly they will just vanish and the lines look like they were spliced together. This even enables splitting an identifier, which does not need to be processed in the preprocessor. For the C3P0Lexer, it will eventually look like one single identifier, instead of two.

```
int variable = 5;
int copy = var\
iable;

//results in:
//int variable = 5;
//int copy = variable;
```

Listing 4.25: Spliced Identifier

In the definition of line comments we have added the escaped newline from above as a possible part of the comment. Thus, the lexer consumes it as if it was a part of the comment content:

```
fragment
LINE_COMMENT
  options{greedy=false;}
  : '//' (ESCAPED_NEWLINE | ~(NEWLINE))*
  ;
```

Listing 4.26: Splicing of Line Comments

Special treatment of line splicing is encountered in string literals. As raw strings, which are not target of line splicing in translation phase two, have been introduced in C++0x, we just add the escaped newline as a possible alternative in the `S_CHAR` rule. This rule represents the possible characters in a string literal:

```
S_CHAR
  : COMMON_BASIC_CHARACTER_SET | '>' | '\''
  | ESCAPE_SEQUENCE
  | UNIVERSAL_CHARACTER_NAME
  | ESCAPED_NEWLINE
  ;
```

Listing 4.27: S_CHAR Rule

As a consequence the backslash and the new-line character belong to the string content. This includes raw strings as well. Subsequently, if the parser encounters a `STRING_LITERAL` token, its text representation is purged.

```
pp_token
  ...
  | sl = STRING_LITERAL {purgeString($sl);}
  ...
  ;
```

Listing 4.28: Purging String Literals

The `purgeString` method, defined in the `C2P0Parser`, removes all escaped newline characters of non-raw-string literal tokens:

```
private void purgeString(Token t) {
  if(!isRawString(t.getText())){
    String purgedText = t.getText().replaceAll("\\\\(\\n|\\r\\n|\\r)", "");
    t.setText(purgedText);
  }
}
```

Listing 4.29: `purgeString` Method

If the token does not represent a raw string, the method replaces its text representation with a spliced version of the string.
This implementation is advantageous regarding raw strings. During the second translation phase, the standard does not make a difference between common string literals and raw strings regarding line splicing. But, in the third phase this replacement is undone in raw strings. To be able to perform this revocation all such modifications need to be marked. As C2P0 is aware of raw strings from the beginning, we can avoid this entirely.

Hidden Channel

In C3P0 we have not been interested in whitespace characters at all. In the C3P0Lexer we have just skipped all whitespace characters, by calling the `skip` method in the corresponding rule:

```
WS
  : (' '|'\t'|'\n'|'\r')+ {skip();}
  ;
```

Listing 4.30: C3P0 Rule for Handling Whitespace Characters

Calling the `skip` method lets the characters matched by the rule vanish. While parsing C++, whitespace characters are syntactically insignificant. But, they are important for the preprocessor, at least when having a character-oriented output. Therefore, in C2P0 we must not skip them like we did in C3P0. Instead, we send them to the `HIDDEN` channel.

```
WS
  : (' '|'\t')+ {$channel = HIDDEN;}
  ;
```

Listing 4.31: C3P0 Rule for Handling Whitespace Characters

Tokens on the hidden channel are skipped in the token stream, that is passed to the parser. Subsequently, the parser will not need to handle them. But they are still available if the stream is accessed directly. We harness this while creating our AST nodes for the C2P0Parser.

4.3.4. C2P0Parser

The C2P0Parser takes the token stream created by the C2P0Lexer and generates an AST as output. The parser rule set of the preprocessor is significantly simpler than the rules for C++ in C3P0. The grammar for the C2P0Parser is derived from chapter 16 of the C++ standard draft [Dra10a]. How we implement such parser rules in general has been described thoroughly in the term project documentation [Cor10a], Section 5.4 Parser. Thus, we will not repeat that here in this document. But, we will describe AST construction and the corresponding rewrite rules.
According to [Par07], rewrite rules, among other possible ways, are the recommended way to construct ASTs. The AST is available as part of the result of a C2P0Parser method that represents a parser rule. It can be accessed with the `getTree` method. Furthermore, it is also possible to access the AST created by another rule, inside the ANTLR grammar, through the `tree` attribute:

```
object_like_macro
  : ^(OBJECT_MACRO name = IDENTIFIER rep = replacement_list
      {objectMacros.put($name.text, new ObjectMacro($name.text, $rep.tree));}
    )
  ;
```

Listing 4.32: Accessing a Tree of Another Rule

AST Generation

In ANTLR parser grammars, we have the possibility to generate ASTs from the token input streams, using rewrite rules. First, AST generation has to be enabled for the parser. To do this the `output` option needs to be set to `AST`:

```
options {
  output = AST;
}
```

Listing 4.33: Output Option

If not further specified, the resulting AST consists of `CommonTree` nodes, provided by the ANTLR runtime.
The syntax for the rules of the parser do not change. They remain the same as described in [Cor10a], section 5.2 Grammar Overview. Additionally, to the functionality we have used so far in C3P0, we use the operators and rewrite rules to make the rule alternatives create and structure the AST nodes we need [Par10c, Par07].
It is possible to leave the parser rules as they are, without any tree construction operators or rewrite rules. Then the resulting AST of that rule is just a flat list of AST nodes. As every rule returns a tree as a result, these nodes are all added as children to a new parent node. This parent node does not have a token assigned. Therefore, it does not represent a specific source element. All such nodes are called a `nil` nodes.
This can be very useful when implementing rules that gather nodes collected by a repetition. For example, the rule `pp_tokens`:

```
pp_tokens
  : pp_token+
  ;
```

Listing 4.34: `pp_tokens` Rule

This rule matches one or more `pp_token` and creates a `nil` node, to which all matched `pp_token` are added as children. Let us consider the following example:

```
int i = 1;
```

Listing 4.35: Code for Tree Example

The AST generated when executing the `pp_tokens` rule is depicted in Figure 4.2.

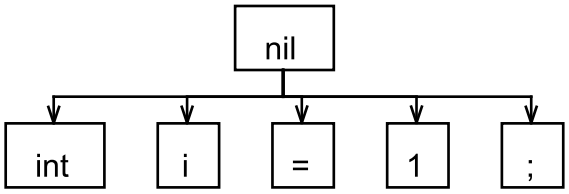

Figure 4.2.: AST for `int i = 1;`

While the resulting AST is flat, behind the scenes several rules have been matched in the parser. The corresponding parse tree, a tree showing the call hierarchy of the parser rules, is shown in Figure 4.3.

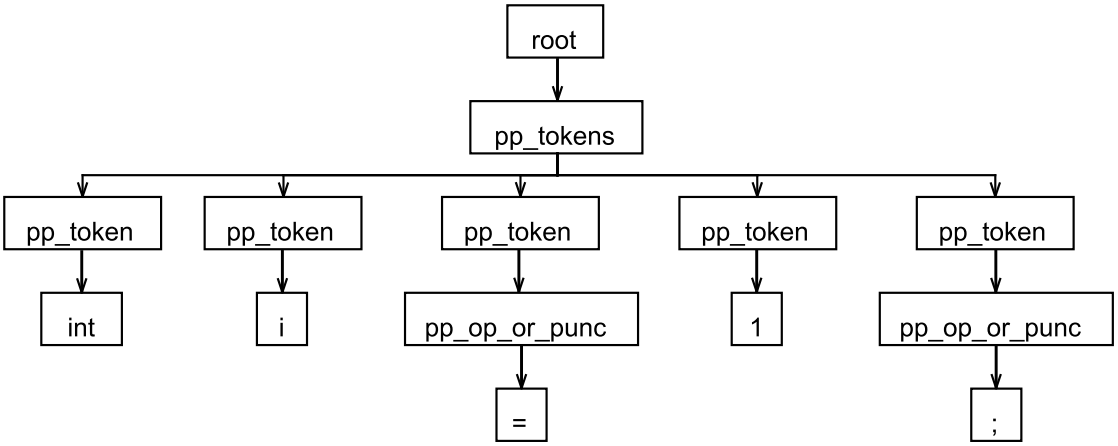

Figure 4.3.: Parse Tree for `int i = 1;`

While the AST can be modified using rewrite operators and rules, the parse tree never changes for the same rule set. The AST is intended to have a representation that is focused on significant parts of the syntax. To structure the AST the following operators are available:

- Exclamation Mark (`!`) - For skipping a node in the AST.
- Circumflex (`^`) - To designate a node to become root of the local subtree.

With the `!` operator, we can remove unnecessary nodes from the AST. For example, a comma, separating identifiers in an identifier list in function-like macro definitions, is not necessary in the AST, because every identifier in the parameter list represents a parameter by itself. Adding an `!` after the token to be matched by the rule, excludes this token when generating the AST for that rule.

```
identifier_list
  : IDENTIFIER (','! IDENTIFIER)*
  ;
```

Listing 4.36: `identifier_list` Rule

To structure the input, matched by a rule, the `^` operator can be used. For example, when parsing mathematical expressions it can be desirable to have the operator being the root of a sub-expression. Let us consider the rule `additive_expression`, used in `C2P0ConditionEvaluator` (stripped off the actions for calculating the result):

```
additive_expression
  : multiplicative_expression
    (
      '+'^ multiplicative_expression
    | '-'^ multiplicative_expression
    )*
  ;
```

Listing 4.37: `additive_expression` Rule

In Figure 4.4 the AST for the expression `5 + 3 * 2` is illustrated. The operators `+` and `*` become the root nodes of the corresponding subtrees.

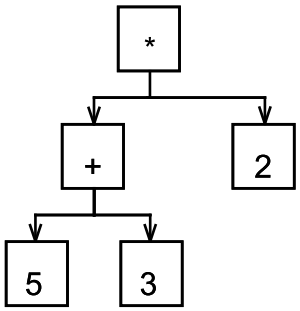

Figure 4.4.: AST for `5 + 3 * 2`

In some cases it is not enough to just remove AST nodes or define a root node for a subtree. For example, reordering is not possible with the operators above. For complex

rules, it might be desirable to separate the AST construction part from the recognition part. This is possible with rewrite rules. A rewrite rule starts with the `->` operator and can be appended to any rule alternative [Par07].

```
rule: <<alt1>> -> <<build-this-from-alt1>>
    | <<alt2>> -> <<build-this-from-alt2>>
    ...
    | <<altN>> -> <<build-this-from-altN>>
    ;
```

Listing 4.38: Rewrite Rule Syntax

The following example shows the rule `object_like_macro` (which matches definitions of object-like macros). The corresponding rewrite rule is included.

```
object_like_macro
  : IDENTIFIER replacement_list NEW_LINE
    -> ^(OBJECT_MACRO IDENTIFIER replacement_list)
  ;
```

Listing 4.39: `object_like_macro` Rule

In the rewrite rule the `^` operator can also be used, but with a slightly different syntax. Instead of putting it after a token, it is placed before a list of tokens or rules in parentheses. The first element of this list becomes the root node of that list. In Figure 4.5 we see the AST generated by the `object_like_macro` rule when the following preprocessing directive is recognized:

```
#define MACRO_NAME this is the replacement of MACRO_NAME
```

Listing 4.40: Object-like Macro Definition

This example also shows the possibility to create new nodes in rewrite rules. Although, `OBJECT_MACRO` is not a part of the rule alternative, there can be such a node in the rewrite rule. The parser creates a `CommonTree` node for it. As there has to be a corresponding token, for defining the type of a node, we either need to define an `OBJECT_MACRO` rule in the lexer or, if we do not have such a rule, we must add `OBJECT_MACRO` to the set of valid tokens explicitly. In this specific case we did the latter. The purpose of these tokens is to ease the task of implementing tree grammars, as we can specifically match them in the tree grammar rules. For C2P0 we have defined several artificial nodes, defined in the `tokens` section of the C2P0 grammar, displayed in Listing 4.41. We will come across these tokens again in the sections explaining the tree grammars.

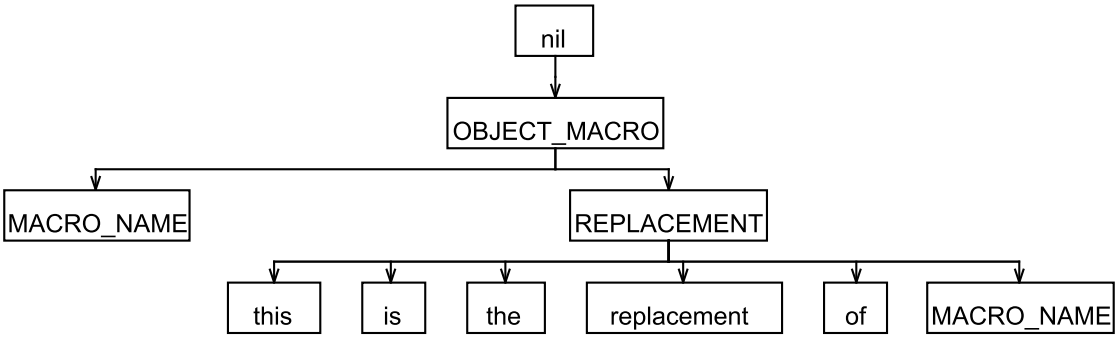

Figure 4.5.: AST for an Object-like Macro Definition

```
tokens{
  FILE; BLOCK; LINE; CODELINE; CONDITIONAL_INCLUSION; CONDITION;
  PARAMETER; REPLACEMENT; OBJECT_MACRO; FUNCTION_MACRO; ARGUMENT;
}
```

Listing 4.41: Tokens of the C2P0 Grammar

Function-like Macros

The rule set for the preprocessor is mostly straight forward and can be implemented very close to the definition in the C++ standard [Dra10a]. There is only one textual restriction, that needs a semantic predicate: In the definition of function-like macros it is important not to have an intermediate whitespace character between the macro name and the parameter list of the macro. Otherwise the macro is considered an object-like macro instead, which cannot take arguments:

```
#define SWAP(a, b) (b, a) // A function-like macro
#define FIRST (a, b) a    // An object-like macro
```

Listing 4.42: Function-like Macro Example

As whitespace characters are on the hidden channel, they are not visible to the parser. Subsequently, we have to check adjacency separately. To ensure there is no space between the macro name and the opening parentheses in the function-like macro definitions, we have added a semantic predicate to check whether they are adjacent. As a consequence the rule `function_like_macro` does not have a valid alternative if there is anything between the macro name and the open parentheses.

```
function_like_macro
  : {getDistance(input.LT(1), input.LT(2)) == 0}? IDENTIFIER
    '(' (identifier_list (',' '...')? | '...')? ')' replacement_list NEW_LINE
    -> ^(IDENTIFIER ^(PARAMETER identifier_list? '...'?) replacement_list)
  ;
```

Listing 4.43: `function_like_macro` Rule

If further flexibility is required to form a rule's tree, it is even possible to define an action that creates a `C2P0Tree` node, which will be set as the rule result. The following example shows the implementation of the `include` rule in the `C2P0Printer`. This action, containing Java code, creates a complete AST, which is inserted instead of the AST for the include directive:

```
include
  : name = HEADER_NAME -> {handleInclude($name.text)}
  ;
```

Listing 4.44: `include` Rule

The `handleInclude` method takes a filename, parses that included file and returns the AST representing that file's content. This AST becomes the rule result tree. This tree is inserted instead of the include statement, where previously an `#include` directive had been in the AST.

C2P0Tree

After scanning and parsing the source code containing preprocessing directives, we have an abstract representation of the input file. This abstraction is free of whitespace information. As the task of preprocessing is rather character- and string- oriented, it is quite important to handle whitespace characters correctly. These facts are obviously conflicting. According to the C++ standard [Dra10a] ([lex.phases]), during the third translation phase, whitespace sequences can be reduced to one single space character. Despite skipping all the whitespace characters again, during lexical analysis in C3P0, retaining them in the preprocessor is important for two reasons: First, if we just removed them, we would splice identifiers together, probably resulting in changed behavior or even syntactically incorrect programs. We could avoid this deficiency by inserting a space between every token reprinted in the preprocessor. Second, when converting a parameter to a string in macro expansion, expanding `__VA_ARGS__` can result in incorrect behavior. But the standard restricts the possible behaviors, for whitespace handling in general, to either leave all whitespace characters as they are or to reduce them to a single space. If we add a whitespace between every token, we could end up having superfluous spaces.

Consider the following example:

```
#define STR(...) #__VA_ARGS__
STR(1,2   , 3) //Shall expand to "1,2 , 3"
```

Listing 4.45: Whitespace Handling Example

Putting a whitespace between every token would not satisfy the form allowed by the standard.
For performing whitespace-handling correctly, we need to somehow retain them. As we cannot store them in the default AST nodes (of type `CommonTree`), we have decided to create our own implementation for representing AST nodes, the `C2P0Tree`. With our own AST node type we have further advantages, as we can configure the node whether to be further expandable during macro replacement. The implementation itself is not complex, it is just an extension of the `CommonTree` class:

```
public class C2P0Tree extends CommonTree implements Cloneable {
  protected boolean hasLeadingWhitespace = false;
  protected boolean expand = true;

  public C2P0Tree();
  public C2P0Tree(Token tok);

  public void setLeadingWhitespace(boolean hasLeadingWhitespace);
  public void clearLeadingWhitespace();
  public boolean hasLeadingWhitespace():

  public void setExpansion(boolean expansion);
  public boolean canBeExpanded();

  protected C2P0Tree dupNode();
}
```

Listing 4.46: C2P0Tree Class

The C2P0Parser has to be configured to expect and work with this new type of tree nodes. We have to set the name of our node class as `ASTLabelType` in the options section:

```
options {
  ...
  ASTLabelType = C2P0Tree;
}
```

Listing 4.47: C2P0 Grammar Options

Otherwise ANTLR expects to be working with nodes of type `Object`, which can imply a lot of casting in parser rule actions. As the generated parser does not create the nodes itself, we have to implement a TreeAdaptor, that works as an abstract factory [GHJV95] for our `C2P0Tree` nodes.

```
public class C2P0TreeAdaptor extends CommonTreeAdaptor {
  protected TokenStream stream;

  public C2P0TreeAdaptor(TokenStream source);
  public C2P0Tree create(Token payload);

  private void determineLeadingWhitespace(C2P0Tree node);
}
```

Listing 4.48: C2P0TreeAdaptor Class

The implementation of the concrete factory, `C2P0TreeAdator`, overrides the `create` method for instantiating our `C2P0Tree` nodes. To figure out whether a specific `C2P0Tree` node has leading whitespace characters, we have the `determineLeadingWhitespace` method. It directly looks at the token stream, `stream`, for checking the preceding tokens. If any are found, the corresponding flag is set in the `C2P0Tree` instance.
Here, we could also collect the specific whitespace tokens and add them to the node, like Overbey et al. [OMJ09] do in their representation. If we intend to later extend our preprocessor to be able to reproduce the complete original source code, we could to this as well.

4.3.5. C2P0Printer

The output of our preprocessor is a source file free of preprocessing directives. Depending on conditional inclusions, certain parts belong to that source file, others are excluded and do not. Through `#include` directives, other source files are mapped into the resulting file. Writing the output character stream from an AST created by the `C2P0Parser` is the task of the C2P0Printer.
The AST needs to be walked to handle the preprocessing directives. This traversal is implemented using a tree walker generated by ANTLR from a tree grammar. A tree grammar defines the structure that it expects the AST to look like. The syntax for the tree grammar rules is very close to the syntax of the rewrite rules. Therefore, a tree grammar for a complete AST can be very similar to the rule set of the parser reduced to the tree construction parts.

Printer Overview

In Figure 4.6, an overview of the classes related to the `C2P0Printer` is shown. The classes and their functionality are described below or in the corresponding sections.

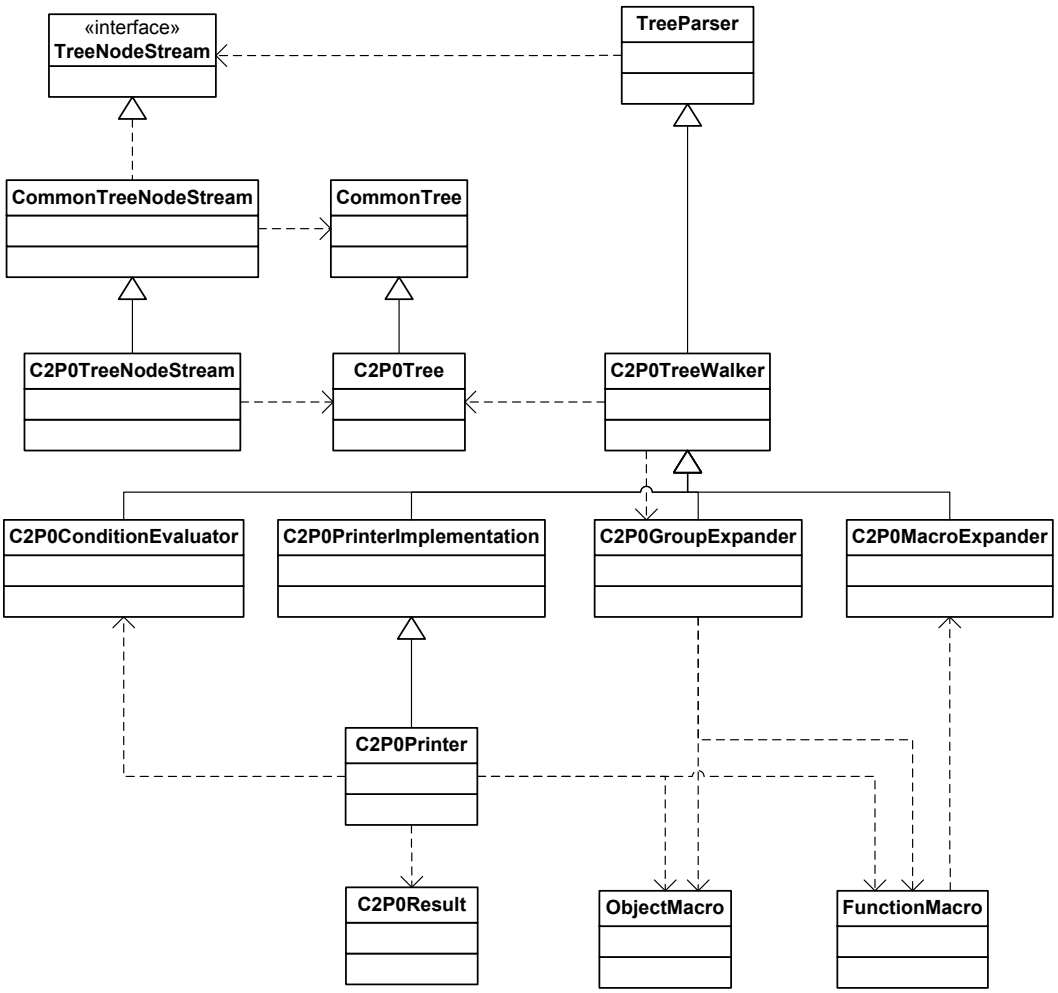

Figure 4.6.: Class Overview of the C2P0 Tree Walkers

Options

The `C2P0Printer` tree grammar needs the following options to be set:

```
options {
  tokenVocab = C2P0;
  output = AST;
  ASTLabelType = C2P0Tree;
  superClass = C2P0PrinterImplementation;
}
```

Listing 4.49: `tokenVocab` Options for the `C2P0Printer`

tokenVocab Lexer rules are not required, as walking the AST just depends on the tree nodes. Nevertheless, the tree walker has to be aware of the tokens used to generate the AST, as the tree node types depend on them. To make the tree walker aware of the token vocabulary of the AST, it needs to be imported into the grammar. To import these tokens, the `tokenVocab` option has to be set to `C2P0`, which configures the tree grammar to import the C2P0 tokens defined in the `C2P0.tokens` file.

output While the result of the C2P0Printer primarily consists of a string representation of the output source, there are some modifications to the AST performed, like source file inclusion. Subsequently, every rule must be able to return an AST which eventually might get inserted into the resulting source file, in string form. To enable the corresponding functionality, we need to set the `output` option to `AST`.

ASTLabelType Like in the `C2P0Parser`, we need to make the tree grammar expect a specific type of tree nodes. The `ASTLabelType` has to be set to the corresponding class name: `C2P0Tree`

superClass The `superClass` option allows to specify a super class for the `C2P0Printer`, generated from the grammar. As ANTLR does not provide the option to define abstract methods, which then are implemented in a derived class, it is recommended to define a super class to have a possibility for an external implementation [Par10a]. Otherwise all methods had to be defined in the `@members` section of the grammar. This can be tedious if the grammar is edited in a grammar-oriented application like ANTLRWorks, that is unaware of Java code. We have defined `C2P0PrinterImplementation` to be the super class. It contains the methods that are required to perform the necessary tree manipulation.

Rules

We have picked the important rules of the `C2P0Printer` for explaining the syntax and its functionality. A tree walker works like a parser, except that it takes a tree as input

instead of tokens. There are two additional input nodes, which are not defined in the vocabulary but required to walk a tree: `UP` and `DOWN`. They represent the structure of the AST when the nodes are serialized, which is necessary for the parser to be able to distinct the levels of the tree. For illustrating the use of these two nodes let us consider the expression `5 + 3 * 2`. The corresponding AST can be seen in Figure 4.4. The stream form of this AST is as follows:

```
* DOWN + DOWN 5 3 UP 2 UP
```

Listing 4.50: Tree Node Sequence for `5 + 3 * 2`

If a node has child nodes, after that parent node a `DOWN` node is emitted and then all its child nodes are visited. After the last child node an `UP` node is emitted. The grammar for a tree walker has to be written correspondingly to expect these nodes. The syntax hides their existence, but they have to be specified implicitly by the rule. In ANTLRWorks these `UP` and `DOWN` nodes are shown in the syntax diagram of the tree grammar rules. For example, the rule `control_line` specifies all preprocessing directives, except conditional inclusions:

```
control_line[boolean active]
  : ^(INCLUDE_DIRECTIVE include[$active])
  | ^(UNDEF_DIRECTIVE id=IDENTIFIER) {if($active) undefineMacro($id.text);}
  | ^(DEFINE_DIRECTIVE macro_definition[$active])
  | ^(LINE_DIRECTIVE pp_tokens)
  | ^(ERROR_DIRECTIVE (tok=pp_tokens {if($active)errors.add($tok.tree);})?)
  | ^(PRAGMA_DIRECTIVE pp_tokens?)
  ;
```

Listing 4.51: C2P0Printer Rule `control_line`

The train graph in Figure 4.7 shows where the `UP` and `DOWN` tokens that are expected. When comparing `control_line` of the `C2P0Printer` to the corresponding rule in the `C2P0Parser`, we see the similarities (highlighted) between the rewrite parts and the tree grammar rules. The only difference is the boolean parameter `active`, which indicates whether a certain part needs to be included or not:

```
control_line
  : HASH INCLUDE_DIRECTIVE include NEW_LINE -> ^(INCLUDE_DIRECTIVE include)
  | HASH UNDEF_DIRECTIVE IDENTIFIER NEW_LINE -> ^(UNDEF_DIRECTIVE IDENTIFIER)
  ...
  ;
```

Listing 4.52: C2P0Parser Rule `control_line`

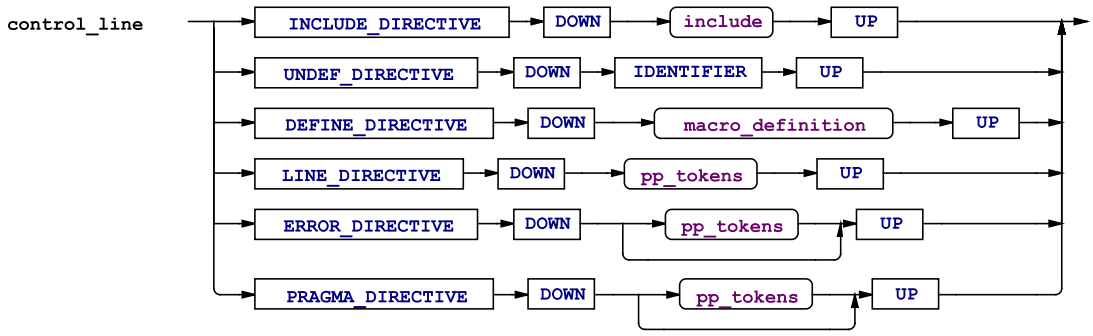

Figure 4.7.: Syntax Diagram for the `control_line` Rule

Below, we have described the rules for handling the basic structure of the AST created from a source file. Figure 4.8 shows the AST created for the simple input `int i = 1;` and which rules match the subtrees.

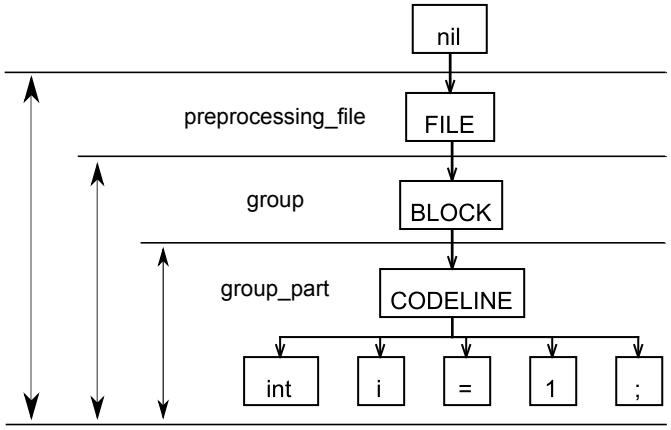

Figure 4.8.: AST for Directive-Free Code

preprocessing_file This is the entry rule, which is called for an AST representing a complete source file. The `C2POParser` puts a `FILE` node as root node of the whole AST. There is one child of this `FILE` node, represented by the `group` rule.

```
preprocessing_file
  : ^(FILE group[true])
  ;
```

Listing 4.53: `preprocessing_file` Rule

group The `group` rule consists of a `BLOCK` node as root, which has `group_parts` as children. The boolean parameter `active` indicates whether the content of this group has to be preprocessed or if it had been excluded through a surrounding `if_section`, for which the condition evaluates to `0` (`false`).

```
group [boolean active]
  : ^(BLOCK group_part[$active]*)
  ;
```

Listing 4.54: `group` Rule

group_part In `group_part`, we have the distinction between preprocessing directives for controlling the active content (`if_section`), other preprocessing directives (`control_line`) and the source code lines, which will form the content of the output file (`code_lines`). The rewritten AST of `code_lines` is converted back into a string representation and becomes part of the preprocessed source file. The alternative that only contains a `LINE` node, represents empty lines which are not relevant for the output.

```
group_part [boolean active]
  : ^(LINE if_section[$active])
  | ^(LINE control_line[$active])
  | cl = code_lines[$active] {if($active)result.printLine($cl.tree);}
  | LINE
  ;
```

Listing 4.55: `group_part` Rule

code_lines This rule collects the source code lines and replaces the macros. As function macro calls can be spanned across several lines, it is necessary to gather the lines of one block and than expand them all together. The fully macro-expanded subtree is the resulting AST of this rule or just an empty line, depending on whether those lines are "`active`" code.

```
code_lines [boolean active]
  : ((CODELINE) => ^(CODELINE tls += pp_tokens))+
      -> {$active}? {expandLines($tls)}
      -> LINE
  ;
```

Listing 4.56: `code_line` Rule

The rules above form the structure of the source file represented by the AST. The following rules control the content, handle source file inclusion and define macros:

if_section The `if_section` describes the subtree for conditional inclusion. The root of that subtree is always a `CONDITIONAL_INCLUSION` node, which has an `if_group` as its first child, optionally followed by `elif_groups` and an `else_group`. The last child consists of an `endif_line` subtree to terminate the conditional inclusion. The `boolean` parameter `active` indicates whether the code inside the groups is part of the output. Furthermore, it is important for the `elif_groups` and `else_group` rules whether the previous conditions have evaluated to non-null (`true`).

```
if_section[boolean active]
  : ^(CONDITIONAL_INCLUSION
      ifg = if_group[$active]
      elifg = elif_groups[$active && !$ifg.wasTrue]?
      else_group[$active && !$ifg.wasTrue && !$elifg.wasTrue]?
      endif_line)
  ;
```

Listing 4.57: `if_section` Rule

Figure 4.9 shows a possible AST, which could be matched by the `if_section` tree grammar rule. This AST represents the input code of Listing 4.58. The code blocks have been colored green if they belong to the output or orange if not. In the example, `version` is expected to be replaced by `2`. We have not included the macro definition for `version` in Figure 4.9 on purpose, as the whole AST would have been too large to be illustrated properly.

```
//#define version 2

#if version == 1
v1
#elif version == 2
v2
#else
v3
#endif
```

Listing 4.58: Example Code for `if_section`

constant_expression Every `if_group` and `elif_group` contains a condition to be evaluated. The root of a subtree representing a `constant_expression` is an artificial `CONDITION` node. The child nodes are a sequence of `condition_token`s, which can be any preprocessing token (`pp_token`). There is one extension to this token set

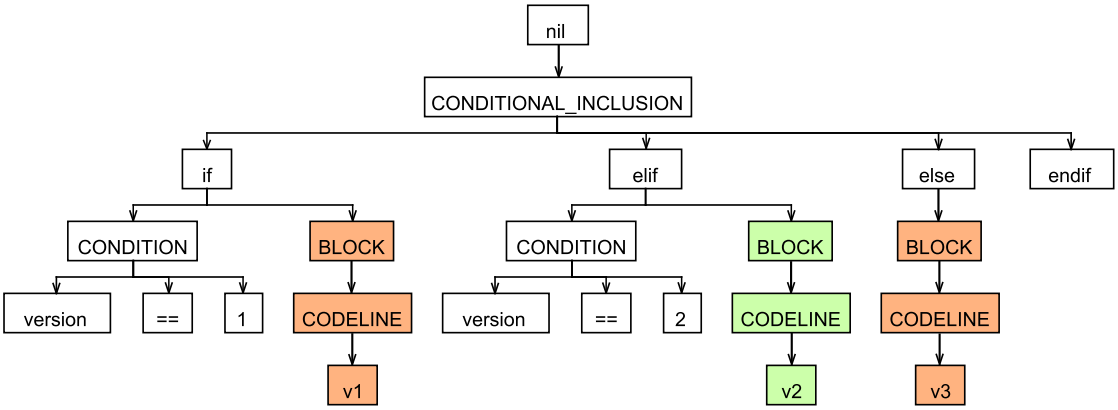

Figure 4.9.: Possible AST matched by `if_section`. `version` is expected to be 2.

though: It is allowed to check whether a macro has been defined at that point. If an `IDENTIFIER` node is encountered, which represents the `defined` keyword, followed by a further `IDENTIFIER`, the C2P0Printer checks if the macro represented by the second `IDENTIFIER` has been defined. These two nodes are replaced by an `INTEGER_LITERAL` node either representing `1`, if it has been defined, or `0`, if not.

The complete subtree matched by `condition_tokens` is passed to the `evaluate` method, which uses the `C2P0ConditionEvaluator` (See Section 4.3.6) to determine the value of this condition expression.

```
constant_expression returns [long value]
  : ^(CONDITION tok=condition_tokens) {$value = evaluate($tok.tree);}
  ;
```

Listing 4.59: `constant_expression` Rule

```
condition_token
  : (IDENTIFIER) => id=IDENTIFIER
    ( {$id.text.equals("defined")}?
        (macro = IDENTIFIER | '(' macro = IDENTIFIER ')')
          -> {isDefined($macro.text)}? INTEGER_LITERAL["1"]
          -> INTEGER_LITERAL["0"]
    | -> $id)
  | pp_token
  ;
```

Listing 4.60: `condition_token` Rule

include There are three possibilities how an `#include` directive can look like. It is either has a path embraced by `<...>` or `"..."`, or consists of a sequence of `pp_tokens`, which have to expand to one of the previous two forms. The distinction between implementation and user includes has been consolidated in the `C2POParser`. In the `C2POPrinter` we just have to determine whether we have received `pp_tokens`, which must be expanded first. The overloaded `handleInclude` method expands the `pp_tokens` if necessary, loads and preprocesses the file specified. The resulting content is added to the output source code.

As not every `#include` directive is in an active section, it can also be in a group for which a surrounding `condition` evaluates to `false`. Subsequently, if a file does not have to be included, it is not. Which saves much processing time.

```
include [boolean visit]
  : name = HEADER_NAME ({$visit}? -> {handleInclude($name.text)})?
  | tok = pp_tokens ({$visit}? -> {handleInclude($tok.tree)})?
  ;
```

Listing 4.61: `include` Rule

To illustrate the steps, let us consider the following example.

```
//before include
#include "header.hpp"
//after include

// --- header.hpp ---
header content
```

Listing 4.62: Example Code for `#include` Directive

From the source code of the including source file, the AST, containing the subtree for the include, is generated. When the `C2POPrinter` encounters that subtree, it loads the included source file, creates an AST for it and generates the resulting source string. That string, instead of the `#include` directive, is appended to the resulting source file (See Figure 4.10).

macro_definition When a `#define` directive is encountered, a new macro has to be defined. We distinct two types of macros: (1) object-like macros and (2) function-like macros. Similar to `#include`, macro definition directives are not always required to be executed. If a macro definition is encountered in a group that is excluded, its definition is not added to the macros known in the `C2POPrinter`.

Object-like macro subtrees have an `OBJECT_MACRO` as root node with two children. The first child is an `IDENTIFIER` representing the macro name. The second child is a subtree of the form defined by the rule `replacement_list`.

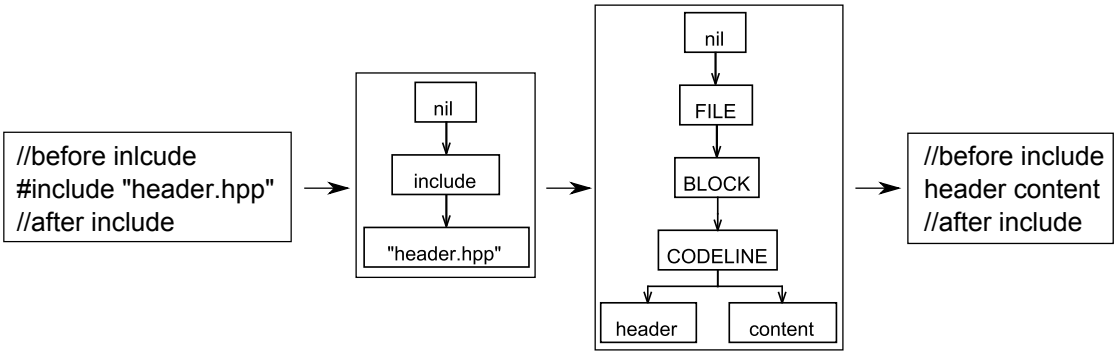

Figure 4.10.: Steps for File Inclusion

A function-like macro is represented by `function_like_macro`, which is similar to `object_like_macro`, but has a `FUNCTION_MACRO` node as root. It also has a further child, `parameter_list`, specifying the parameters of the function-like macro.

```
macro_definition [boolean active]
  : ^(OBJECT_MACRO object_like_macro[$active])
  | ^(FUNCTION_MACRO function_like_macro[$active])
  ;

object_like_macro [boolean active]
  : name = IDENTIFIER rep = replacement_list
      ({$active}? {
        define(new ObjectMacro($name.text, $rep.tree));
      })?
  ;

function_like_macro [boolean active]
  : ^(name = IDENTIFIER args = parameter_list rep = replacement_list)
      ({$active}? {
        define(new FunctionMacro($name.text, $args.tree, $rep.tree));
      })?
  ;
```

Listing 4.63: Macro Definition Rules

A `replacement_list` is a subtree with a `REPLACEMENT` node as root. The children are a, possibly empty, list of `pp_tokens`:

```
replacement_list
  : ^(REPLACEMENT pp_token*)
  ;
```

Listing 4.64: `replacement_list` Rule

The `parameter_list` of a `function_macro_definition` has a `PARAMETER` node as root. The children are a, possibly empty, list of `IDENTIFIER`s and an optional, trailing ellipsis (`...`) for taking variable arguments.

```
parameter_list
  : ^(PARAMETER IDENTIFIER* '...'? )
  ;
```

Listing 4.65: `parameter_list` Rule

Figure 4.11 shows an AST that is generated by the `C2P0Parser` for the function-like macro definition in Listing 4.66. If the `C2P0Printer` encounters such an AST, it creates the corresponding macro definition for it.

```
#define SUM(a, b) (a + b)
```

Listing 4.66: Function-like Macro Definition Example

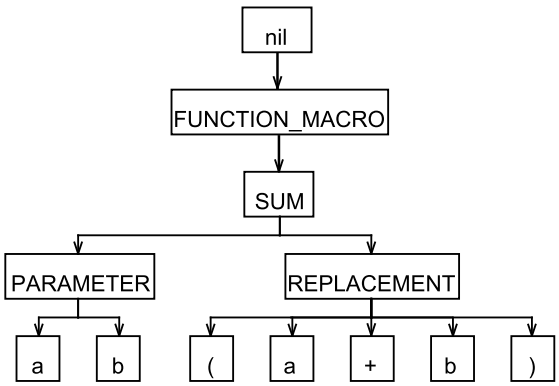

Figure 4.11.: AST for Function-like Macro Definition

4.3.6. C2P0ConditionEvaluator

Conditional inclusion always depends on at least one condition to be evaluated. In the C++ standard draft [Dra10a]([cpp.cond]/1) this condition is defined to be an integral constant expression. As mentioned in Section 4.2.1, in the new standard it is possible to have invocations of defined `constexpr` functions being a part of a constant expression [Dra10a]([expr.const]/2). This problem gets resolved by a further specification, in [Dra10a]([cpp.cond]/1), which designates all identifiers in such conditions to be treated

as macro names. The `C2P0ConditionEvaluator` is a tree grammar, written for determining the result of a constant expression in a conditional inclusion. Currently, it is just used by the `C2P0Printer` for this condition evaluation. But the grammar could also be used to structure such conditions. The rule set is similar to the rules for evaluating expressions `NewestDraft`([expr]). As the condition has to be integral, the constant values are limited to `INTEGER_LITERAL`s, `CHARACTER_LITERAL`s and `IDENTIFIER`s, which are either macro-expanded or evaluate to `0`.

Options

The `C2P0ConditionEvaluator` tree grammar has the following options set:

```
options {
  tokenVocab = C2P0;
  output = AST;
  ASTLabelType = C2P0Tree;
  superClass = C2P0TreeWalker;
}
```

Listing 4.67: Options of the `C2P0ConditionEvaluator`

These settings are the same as for the `C2P0Printer`, except for the `superClass` option. The `C2P0TreeWalker` manages the state of a tree walker and provides common methods, for defining and expanding macros.

Rules

Below we describe the grammar rules of the `C2P0ConditionEvaluator`. As the implementation is similar for most operators, we focus on some specific rules. The following operators are allowed:

- (`?:`) Ternary
- (`||`) Logical Or
- (`&&`) Logical And
- (`|`) Inclusive Or
- (`^`) Exclusive Or
- (`&`) And
- (`==`) Equality
- (`!=`) Inequality
- (`<`) Less Than
- (`>`) Greater Than
- (`<=`) Less Than or Equal
- (`>=`) Greater Than or Equal
- (`«`) Left Shift
- (`»`) Right Shift
- (`+`) (Unary) Plus
- (`-`) (Unary) Minus
- (`!`) Logical Negation
- (`~`) Bit Complement

Every rule returns a value, which is calculated according to the operator specified. As the type of our return value is `long`, a signed 64 bit integer value, it is not possible to

represent unsigned 64 bit integers correctly. Actually, we do not have the distinction between singed and unsigned integer values at all. We could use `java.math.BigInteger`s to handle integer values larger than $2^{63}-1$, but this would not satisfy all requirements either. It would solve the problem of handling larger integer values, but `BigInteger` would not behave like `intmax_t`, or `uintmax_t` respectively, regarding overflow, as required by [Dra10a]([cpp.cond]/4). Because `BigInteger` is designed to avoid overflows, there should never occur one. Since the limits are implementation defined [Dra10a]([support.limits]), we do not expect encountering much real world code to exploit these overflows in conditions. Therefore, we consider this behavior not to be a serious drawback.

multiplicative_expression This is a common representative for most of the rules in `C2P0GroupEvaluator`. It has a left part, an `unary_expression`, optionally followed by either the `*`, `/` or `%` operator with the corresponding `unary_expression`. Depending on the operator the return value is calculated. The modulo operator (`%`) needs to be escaped as ANTLR otherwise tries to interpret it as part of a string template rule.

```
multiplicative_expression returns [long value]
  : e1 = unary_expression {$value = $e1.value;}
    ( '*'^  e2 = unary_expression {$value *= $e2.value;}
    | '/'^  e2 = unary_expression {$value /= $e2.value;}
    | '%'^ e2 = unary_expression {$value \%= $e2.value;}
    )*
  ;
```

Listing 4.68: `multiplicative_expression` Rule

logical_or_expression Some operators never result in values possibly reaching the limits describes above, as they evaluate to a boolean value, represented by either `0` (`false`) or `1` (`true`). The following operators have a boolean result: Logical Or, Logical And, Equality, Inequality, Less Than, Greater Than, Less Than or Equal, Greater Than or Equal and Logical Negation.

```
logical_or_expression returns [long value]
  : e1 = logical_and_expression {$value = $e1.value;}
    ('||'^ e2 = logical_and_expression
      {$value = ($value != 0 || $e2.value != 0) ? 1 : 0;})*
  ;
```

Listing 4.69: `logical_or_expression` Rule

conditional_expression The rule `conditional_expression`, for the ternary operator, has a `logical_or_expression` and two `conditional_expressions`. Other rules usually have the same rule left and right to the operator. As seen in listing 4.70 this is different in `conditional_expression` for the following reasons: According to the C++ standard [Dra10a] ([expr.cond]) the ternary operator is right-associative. Thus, the condition part before the `?` must not be a `conditional_expression` itself, otherwise the rule would be self-recursive. Furthermore, we cannot use the approach of the other rules, to have a `logical_or_expression`, optionally followed by a sequence of repetitive ternary operators `('?' ':')*` (with further `logical_or_expression` as sub expressions). Because this would imply left associativity and denied the use of inner ternary operations between `'?'` and `':'`.

```
conditional_expression returns [long value]
  : e1 = logical_or_expression {$value = $e1.value;}
     ('?'^ e2 = conditional_expression ':'! e3 = conditional_expression
     {$value = ($value != 0) ? $e2.value : $e3.value;})?
  ;
```

Listing 4.70: `condition_expression` Rule

Figure 4.12 illustrates the difference between left- and right-associativity, representing the following code:

```
#if true ? true : false ? false : false
Expected Code
#else
Unexpected Code
#endif
```

Listing 4.71: Example Code for Nested `conditional_expressions`

While the right-associative approach evaluates to `true`, the left-associative approach would evaluate to `false`, yielding an incorrect result.
We expect, in real preprocessing code nested `conditional_expressions` hardly occur, at least not without explicit association, using parentheses. As both, the Microsoft Visual C++ compiler (version 16.00.30319.01 for 80x86) as well as the GNU CPP (version 4.3.4), do not handle the preprocessing code above correctly. GNU CPP (version 4.3.4) has nesting of the ternary operator implemented, but it behaves left-associative and the Microsoft Visual C++ compiler just cuts the line at the end of the first ternary operation. In GNU CPP version 4.4.0 this problem does not occur anymore, it behaves correctly in this case.
Remark: In the documentation for the MS VC++ there is no explanation of the operators allowed [Mic10a]. The documentation for the GNU CPP actually does not specify the ternary operator as valid for constant expressions in conditional inclusions [Pro10].

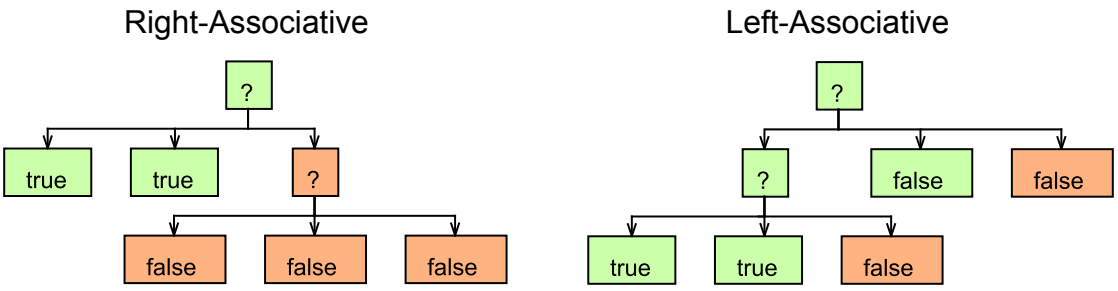

Figure 4.12.: AST for Right- and Left-Associative Conditional Inclusion

postfix_expression This rule determines the values of the conditions operands. In the first place, this is parsing the values of `INTEGER_LITERALS`. The constraints to these value has been described above. Furthermore, the rule covers expression parts embraced by parentheses, which is only important for the precedence and does not imply computations in the rule itself. According to the C++ standard [Dra10a] ([cpp.cond]/4) almost all remaining `IDENTIFIER`s evaluate to `0`. There is one exception though: `true` evaluates to non-zero, `1` in our implementation. The last alternative handles `CHARACTER_LITERALS`, which are converted to their corresponding integer value, according to the Java implementation of `char`.

```
postfix_expression returns [long value]
  : val = INTEGER_LITERAL {$value = parseLongValue($val.text);}
  | '('! expr = conditional_expression ')'! {$value = $expr.value;}
  | id = IDENTIFIER  {if($id.text.equals("true")) $value = 1; else $value = 0;}
    ({$id.text.equals("defined")}?
      macro = IDENTIFIER {$value = isDefined($macro.text) ? 1 : 0;})?
  | cl = CHARACTER_LITERAL {$value = getCharValue($cl);}
  ;
```

Listing 4.72: `postfix_expression` Rule

4.3.7. C2P0GroupExpander

Macro replacement is a difficult topic of preprocessing, at least when doing it AST-oriented instead of stream-oriented. There is one basic problem when performing macro expansion using ANTLR: According to the C++ standard [Dra10a] ([cpp.rescan]/1), after a macro has been replaced, the resulting sequence is rescanned for further replacement. This rescan includes all subsequent preprocessing tokens. The problem we encounter here, origins from the way a parser usually processes the tokens, or nodes respectively in the case of a tree walker. After matching a macro name, which has to be expanded, we can easily take the replacement and insert it into the result. But, the

parser is not intended to directly parse that result again, as part of the input. While this could easily be performed recursively (on the result only), it would not yield the correct behavior, as we need to include the rest of the node sequence in this rescan. Furthermore, recursively rescanning the replacement, including the remaining nodes, could result in a deep call structure for source code consisting of many macros, which could become a limitation regarding scalability.
To overcome this problem, we have moved control over macro replacement and rescanning out of the ANTLR-generated tree grammars. The `expandTree` method handles the node stream and the replacement to ensure correct behavior. We still use a tree grammar for expansion, as it is easier to implement the recognition and understand the replacement defined in grammar form, especially regarding argument lists for function-like macros. The `C2P0GroupExpander` expects to walk a sequence of preprocessing tokens. It processes the nodes one by one and creates the replacement if it encounters a macro.

Options

The `C2P0GroupExpander` tree grammar has the following options set:

```
options {
  tokenVocab = C2P0;
  output = AST;
  ASTLabelType = C2P0Tree;
  superClass = C2P0TreeWalker;
}
```

Listing 4.73: Options of the `C2P0GroupExpander`

The `C2P0GroupExpander` has the same options as the `C2P0ConditionEvaluator`. It is derived from the `C2P0TreeWalker` to inherit its macro handling capabilities. The `output` option here is very important, as the `C2P0TreeWalker` depends on the AST created during expansion.

Rules

Below, we have described the rules of the `C2P0GroupExpander`. The most important rule is `try_expand` as it checks whether an `IDENTIFIER` is a macro that has to be replaced and invokes the macro expansion if necessary. Furthermore, we have a ruled for recognizing arguments of function-like macros.

try_expand This rule rewrites the AST passed as input, by performing the necessary lookahead and invoking macro replacement. It has three alternatives, which eventually match all possible node sequences, as long as it does not contain `UP` and `DOWN` nodes:

- The first alternative matches function-like macro calls, starting with the macro name, an `IDENTIFIER`, followed by `arguments`. This rule only matches,

if the macro specified by the `IDENTIFIER` is a known function-like macro, that has not already been expanded in the active replacement procedure and if this macro name has not been explicitly excluded during a preceding macro expansion. These conditions are checked by the `shallExpandFunction` method in the syntactic predicate. The arbitrary number of `NEW_LINE` nodes allows to span function-like macro calls across several lines. If the rule matches, the flag `expanded` is set, to indicate that there has been an expansion. The rewrite part of the rule retrieves the corresponding `FunctionMacro`, which is expanded along with the parameters matched by the `argument` rule. Eventually, the rule returns whether the replaced macro call had leading whitespace characters. That is necessary as the node representing the call will vanish during replacement and together with that node the information about leading whitespace.

- The second alternative matches single `IDENTIFIER`s. It works very similar to the first alternative, for expanding function-like macros. After matching the `IDENTIFIER`, an action determines whether it is a macro that shall be expanded or not. Opposed to the first alternative, this one always matches, whether the `IDENTIFIER` is an object-like macro to be expanded or not. The distinction of the result lies in the rewrite rules: The first rewrite rule alternative is gated by the `expanded` predicate. If this is `true` the macro is retrieved and expanded, if not the rule returns just the matched `IDENTIFIER` as a resulting AST. Again, whitespace information is retained.
- The third alternative matches any other node type on the stream.

```
try_expand returns [boolean expanded, boolean remainingWhitespace]
@init{
  $expanded = false;
}

  : (id = IDENTIFIER {shallExpandFunction($id.text)}? NEW_LINE* arguments) =>
    id = IDENTIFIER NEW_LINE* args = arguments
    {
      $expanded = true;
      $remainingWhitespace = $id.hasLeadingWhitespace();
    }
      -> {functionMacros.get($id.text).expand($args.arguments, this)}
  | (IDENTIFIER) => id = IDENTIFIER
    {
      $expanded = shallExpandObject($id.text);
      $remainingWhitespace = $id.hasLeadingWhitespace();
    }
      ->  {$expanded}? {objectMacros.get($id.text).expand(this, $id)}
      ->  $id
  | unexpanded_pp_token?
  ;
```

Listing 4.74: `try_expand` Rule

arguments This rule matches the argument list for a function-like macro replacement.

- The `argument` rule returns a list of `C2POTree`s, each representing one argument. The arguments are embraced by parentheses and separated by commas. We do not have any limitation to the number of arguments expected. They are just parsed and then passed to the `FunctionMacro` for expansion.
- An argument can be empty or consist of several parts. It is possible to have a sequence of several tokens as one single argument, as long as there is no intervening comma.
- An argument part can be almost any preprocessing token. There are two special alternatives though: (1) It can match a `NEW_LINE` token, which is not included in the rewritten AST. This allows to span the arguments of a function-like macro replacement across several lines. (2) It is possible to have parentheses in arguments, but only if they occur pairwise. If not, this would either leave the argument list unterminated, in the case of open parentheses or cut the argument list, in case of close parentheses. Furthermore, it is allowed to have commas in arguments iff embraced by parentheses.

```
arguments returns[List arguments]
@after{
  $arguments = $args;
}
  : '('(args+=argument) (',' (args+=argument))* ')'
  ;

argument
  : argument_part*
  ;

argument_part
  : INTEGER_LITERAL
  | FLOATING_LITERAL
  | CHARACTER_LITERAL
  | STRING_LITERAL
  | IDENTIFIER
  | NEW_LINE!
  | '(' (argument_part | ',')* ')'
  | preprocessing_op_or_punc
  ;
```

Listing 4.75: Rules for Matching Argument Lists

4.3.8. C2P0MacroExpander

The `C2P0MacroExpander` is used in `FunctionMacro`s during replacement. Compared to object-like macros the replacement of function-like macros is more complex. There are four additional tasks:

1. The parameters of the function-like macro in the replacement have to be substituted with the corresponding arguments.
2. Before an argument replaces a parameter, it has to be macro-replaced first.
3. The arguments for parameters following the hash (`#`) token, are converted into a corresponding `STRING_LITERAL`.
4. Tokens with an intervening `##` token, have to be concatenated to form a new token.

The resulting AST represents the fully expanded function-like macro replacement.

Options

The `C2P0MacroExpander` tree grammar has the following options set:

```
options {
  tokenVocab = C2P0;
  output = AST;
  ASTLabelType = C2P0Tree;
  superClass = C2P0TreeWalker;
}
```

Listing 4.76: Options of the `C2P0MacroExpander`

The `C2P0MacroExpander` has the same options as the `C2P0GroupExpander`. It is derived from the `C2P0TreeWalker` to inherit its macro handling capabilities.

Rules

We have two rules worth mentioning in the `C2P0macroExpander`: First, there is the rule `expand_function_macro`, used by `FunctionMacro`s for expanding their replacement list. Second, `replacement_token`, which handles each token of the replacement, by performing substitution, string replacement and concatenation.

expanded_function_macro This rule matches an AST, having a `REPLACEMENT` node as root with a list of `replacement_token`s as children. It collects whitespace information, which is required if the replacement of nodes leaves unassigned whitespaces. Remaining whitespaces are added to the next possible node. If there are unassigned whitespaces left, they are returned to the caller. The resulting AST has the replaced nodes as children.

```
expand_function_macro returns [boolean remainingWhitespace]
@init{
  $remainingWhitespace = false;
}
  : ^(REPLACEMENT (tok = replacement_token
    {
      $remainingWhitespace |= $tok.remainingWhitespace;
      if($tok.tree != null && $remainingWhitespace)
      {
        $tok.tree.setLeadingWhitespace(true);
        $remainingWhitespace = false;
      }
    }
    )*) -> replacement_token*
  ;
```

Listing 4.77: `expand_function_macro` Rule

replacement_token This rule processes the input nodes of the replacement list. The first three alternatives are interesting:

1. A hash (`#`) node followed by an `IDENTIFIER` has to be replaced by a new `STRING_LITERAL` node. According to the standard [Dra10a] ([cpp.stringiz]/1) the `#` operator is expected to be followed by a parameter. Currently, the `C2POMacroExpander` accepts any `IDENTIFIER`, regardless whether it is a parameter or not. If it is a parameter, the `createStringTree` method creates the `STRING_LITERAL` node for the corresponding argument. This argument is not macro replaced and subsequently, taken as is for determining the string representation. If it is not a parameter we just rewrite the `IDENTIFIER`, letting the `#` vanish. This alternative cannot yield remaining whitespaces, as there will always be a node which they are appended to.

```
replacement_token returns[boolean remainingWhitespace]
  : hash = '#' id = IDENTIFIER
      -> {isParameter($id)}?
          {createStringTree(arguments.get($id.text), $id, $hash)}
      -> IDENTIFIER
```

Listing 4.78: String Operator Alternative

2. The second alternative handles the concatenation (`##`) operator. The preceding and succeeding nodes together form a new node. If the original nodes are parameters they are replaced with the corresponding arguments. Like in string conversion, the arguments are not macro replaced. For allowing chains of concatenations there can be an arbitrary number of following concatenations after the first one. The additional node just gets appended. Like the

first alternative, this one always yields a node. Subsequently, there will never be remaining whitespaces.

```
replacement_token returns[boolean remainingWhitespace]
  ...
  | id1 = pp_token '##' id2 = pp_token
    {
      C2P0Tree arg1 = isParameter($id1.tree) ?
        arguments.get($id1.start.toString()) : $id1.tree;
      C2P0Tree arg2 = isParameter($id2.tree) ?
        arguments.get($id2.start.toString()) : $id2.tree;
      C2P0Tree res = joinArguments(arg1, arg2, $id1.tree);
    }
    ('##' id3 = pp_token
    {
      C2P0Tree arg3 = isParameter($id3.tree) ?
        arguments.get($id3.start.toString()) : $id3.tree;
      res = joinArguments(res, arg3, res);
    } )* -> {res}
  ...
```

Listing 4.79: Concatenation Operator Alternative

3. The third alternative handles `IDENTIFIER`s, which are possibly parameters. If it effectively is a parameter, determined by the predicate in the first rewrite rule, the corresponding argument is expanded, using the `expandArgument` method. The nodes of the expanded argument replace the parameter.

   If the argument is empty, the leading whitespace of the parameter becomes the remaining whitespace. The whitespace information is returned to the caller of the rule. The parameter vanishes.

   If the `IDENTIFIER` is not a parameter, the alternative just rewrites to the `IDENTIFIER` itself.

```
replacement_token returns[boolean remainingWhitespace]
  ...
  | id = IDENTIFIER
    {
      if(isParameter($id) && arguments.get($id.text) == null){
        $remainingWhitespace = $id.hasLeadingWhitespace();
      }
    }
      -> {isParameter($id)}?
          {expandArgument(arguments.get($id.text), $id, this)}
      -> IDENTIFIER
  ...
```

Listing 4.80: Parameter Alternative

4. The remaining alternatives are just for advancing in the node list and do not get substituted.

```
replacement_token returns[boolean remainingWhitespace]
  ...
  | INTEGER_LITERAL
  | FLOATING_LITERAL
  | CHARACTER_LITERAL
  | STRING_LITERAL
  | preprocessing_op_or_punc
  ;
```

Listing 4.81: Remaining Alternatives

4.3.9. PositionMap

For being able to create a CDT-AST with source positions related to the original location of the code in the corresponding files, the character stream as result of C2P0, is not sufficient. We need a possibility to map the positions of the tokens created from the preprocessed stream, by the `C3P0Lexer`, back to the original source positions. These positions are available while preprocessing in the tokens associated with the AST nodes. When a tree gets converted into its string representation, the positions can hardly be included in this representation. We have decided to create a mapping between the offsets in the string representation of the code and the original source positions, provided by the tokens in C2P0.
Instead of just appending the string representation of a token to the result string in the `C2P0Result`, we also put the token into a map with the current offset as key. As every element that has a string representation will have a token representation in C3P0, we can query the original (C2P0) token with the offset of the new (C3P0) token.
The following example illustrates the position mapping. Let us assume we have two C++ source files, one including the other.

```
#include "incl.hpp"

int j = i;


// ----- incl.hpp ------

int i = 5;
```

Listing 4.82: Unpreprocessed Example Code

After C2P0 has processed to source files we receive the following preprocessed source code as output:

```
int i = 5;
int j = i;
```

Listing 4.83: Preprocessed Example Code

It does not contain any relation to the original source code and therefore, we cannot create a CDT-AST with correct position information - which is crucial for automated refactorings. In Figure 4.13, we have illustrated the position mapping for the `INTEGER_LITERAL` (`5`). From the original source file `incl.hpp` we have the corresponding token, including the source line, start and stop position. At that moment, the node containing this token is converted into its string representation for the output, the token is put into the position map, together with the offset in the preprocessed code (`8`) as key. When the `C3P0Lexer` tokenizes the preprocessed code, it creates a token for this `INTEGER_LITERAL`, with offset `8`. Then it is possible to query the position map to get the original token, containing the necessary information, for this offset.

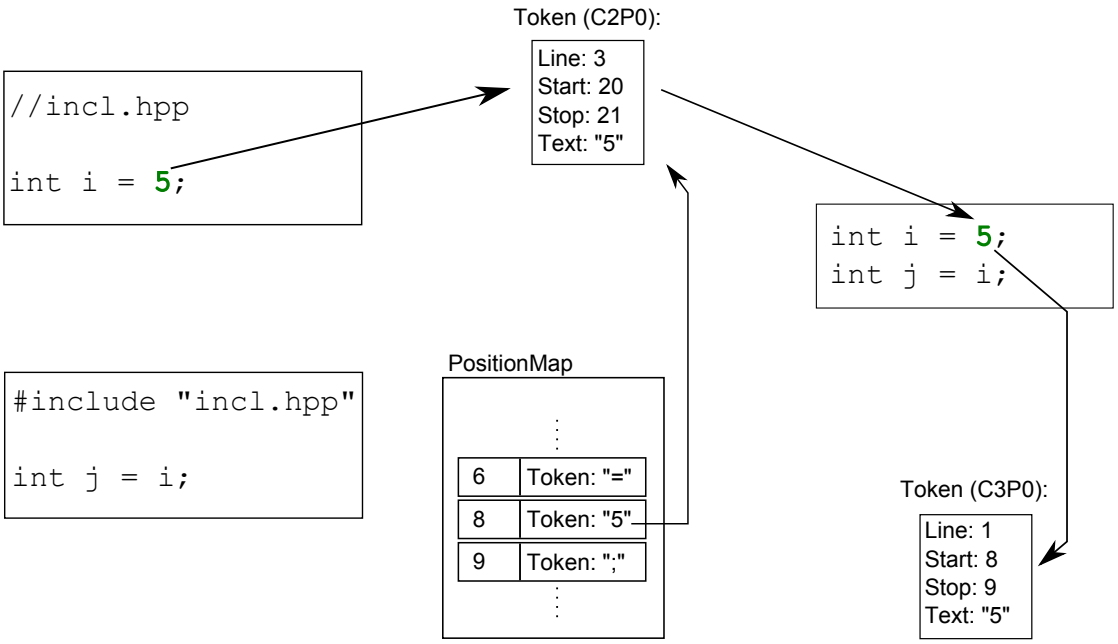

Figure 4.13.: Position Map Example

In the future, we will probably need a further mapping of positions for setting positions of macros and source file inclusions in the CDT-AST. This problem will be engaged when dealing with macros and inclusions in the CDT-AST. This will likely be close to the approach of Overby et al. [OMJ09], including references to nodes that were replaced. **Remark:** The tokens, in C2P0, contain the source file name they belong to.

4.3.10. Macro Replacement

Replacing macros can be more complex than expected at a glance. Especially, rescanning after replacement and exclusion for further expansions required some effort to get macro replacement work correctly in C2P0. As explained in Section 4.3.7, rescanning after macro expansion cannot be solved in an ANTLR generated tree walker directly. This functionality is located in the `expandTree` method of `C2P0TreeWalker`.

Control During Expansion

To explain control of the token stream, let us consider the following example:

```
#define step 5
#define increase(a) a + step
increase(1)  // expands to: 1 + 5
```

Listing 4.84: Expansion Example

When the `C2P0Printer` encounters the group containing the line `increase(1)` it calls the method `expandTree` for expanding that subtree. Therein, the tree is converted into a node stream, which serves as input for a `C2P0GroupExpander`. In Figure 4.14, on the left side we see the stream. It contains pointer to the next node and the corresponding `C2P0GroupExpander`. By invoking `try_expand` (the method for the corresponding rule in the `C2P0GroupExpander` grammar), it tries to expand the macro at the current position of the stream, if there is any.

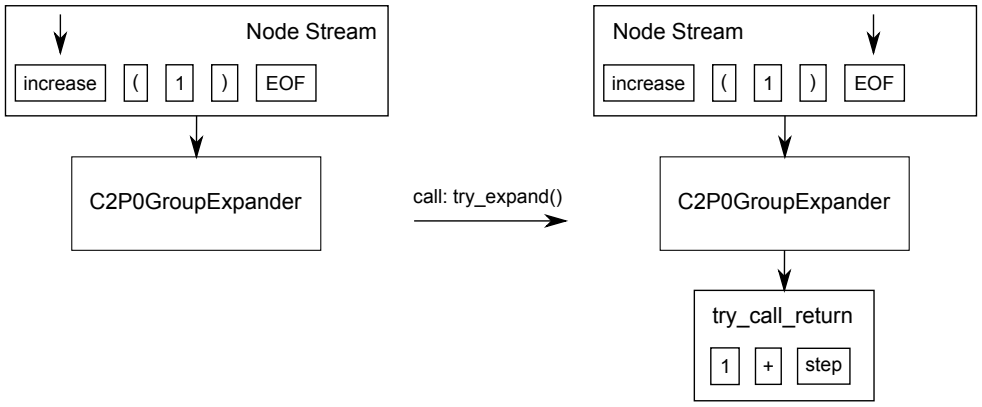

Figure 4.14.: Result of Expansion Using `try_expand`

On the right side of Figure 4.14 we see the state after calling `try_expand`. The node stream is consumed until the end of the macro call and the return value contains the nodes created by the `C2P0MacroExpander` for the `increase` macro. While the parameter

has been replaced, other macros, like `step`, are still untouched. So macro replacement is not finished. While in this case we could just invoke macro expansion, using the `expandTree` for the returned replacement (`1 + step`), this is not sufficient for all cases. As the standard includes the remaining tokens for this rescan, this can yield a different result. Consider the following example:

```
#define f(a) a + g
#define g(b) b + 3
f(1)(2)  // expands to: 1 + 2 + 3
```

Listing 4.85: Example Including Remaining Tokens

This result could not be achieved by just rescanning the result of the `try_expand` call, which yields `1 + g`, because `g` for itself is not a macro call. On the stream we have `(2)` remaining, as `f(1)` has been consumed.
In the first place, we had an approach which reads all remaining tokens of the node stream and created a new stream from the returned replacement and the remaining stream. For this new stream the `C2P0GroupExpander` was invoked again. When it did not expand a macro it just continued. While this worked it had not been a very clever approach, as consuming the remaining stream and creating a new one implies a lot of unnecessary effort.
To avoid this flaw, we have implemented our own node stream, `C2P0TreeNodeStream`.

C2P0TreeNodeStream

Our stream implementation is derived from `CommonTreeNodeStream` and inherits all functionality from it. Additionally, it provides the possibility to insert new tree nodes at the current position of the stream.
The `CommonTreeNodeStream`, provided by ANTLR, takes an AST and iterates over this structure. As a parser needs lookahead capabilities for backtracking, this stream has a buffer which stores nodes for reading ahead and rewinding the stream if necessary. The buffer is required, as the iterator used to walk the tree cannot go backwards. We use this buffer to insert our nodes from the replacement.
The implementation is very simple. We have one method which inserts a `C2P0Tree` at the current position of the stream. This virtually rewinds the stream to the point of the replacement, yielding a state where next on the stream comes the whole replacement followed by the remaining tokens.
There are three cases of possible replacements:

1. If the macro expands to nothing, the replacement is `null`. In other words the macro call vanishes, which already happened as it had been consumed and no replacement is inserted instead. Therefore, we do not need to insert anything into the stream.

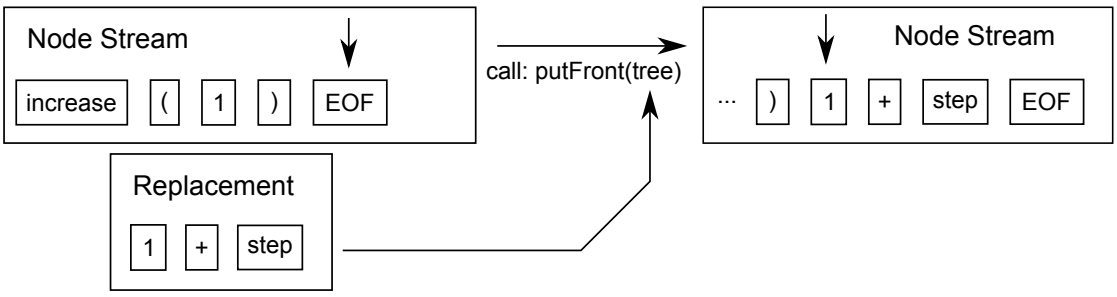

Figure 4.15.: Stream State Before and After Insertion

2. If the replacement just consists of a single node, we have to insert that node at the current position of the stream. Which makes the replacement the next node to be returned by the stream.

3. If the replacement consists of several nodes, they are all expected to be children of the root node representing the tree. Subsequently, all children are inserted at the current position of the stream. Then the next nodes on the stream are the replacement, followed by all subsequent nodes of the stream.

After inserting the replacement, the end of file index has to be adjusted, as the position of the `EOF` in the buffer might have changed.

```
@SuppressWarnings("unchecked")
public void putFront(C2P0Tree tree) {
  if(tree == null){
    return;
  }
  if(tree.getChildCount() == 0) {
    data.add(p, tree);     // data is a buffer for the stream
  }                        // p is the current index in that buffer
  else{
    data.addAll(p, tree.getChildren());
  }
  if ( data.get(data.size()-1)==eof ) {
    eofElementIndex = data.size()-1;
  }
}
```

Listing 4.86: `putFront` Method

Node Exclusion for Future Expansion

After a macro has been replaced, the replacement is rescanned for further macros. There is an exclusion to this expansion though. A macro name that had previously been

expanded in this recursive replacement procedure, is not further expanded. This prevents endless loops during macro replacement:

```
#define f(a) a + f(a)
f(1) //expands to: 1 + f(1)
```

Listing 4.87: Example for Exclusion

This exclusion just holds for the replacement of a macro and not for the whole rescan, so the "remaining" nodes are not affected. Therefore, the exclusion cannot be depending on the macro name only. Tracking the position until which an exclusion is applied can be quite difficult. As an absolute position cannot be taken in the stream, because the number of nodes is increased and decreased during the expansion. As described in Section 4.3.4 every node can be marked as not being a valid target for expansion. Thus, after creating a replacement for a macro, we scan it for further macros, which had been expanded in the current chain of replacement and disable further expansion for those nodes.

4.3.11. Testing

To verify the functionality of our preprocessor, we had several parts to be tested:

C2P0Lexer First, we had to ensure the lexer can tokenize unpreprocessed C++ code as input. To have an independent test set, we have used the boost C++ library, version 1.42 [DNR+10], as a basis for generating lexer tests. To have a simple management of test cases, we have a designated directory containing all source files to be tested in C2P0 tests: `source/resources/C2P0_testcases`

We have added all boost headers to this directory for building the set of files to be tested. For every single file, a test case is created. The content of this file is the input to run the lexer with. After running the test, which includes testing the parser, the state of the lexer is checked and it is ensured that the test ran without errors. All boost headers build a test set of about 7'000 files. Furthermore, any other test of C2P0 depends on the C2P0Lexer, which directly tests the lexer as well.

C2P0Parser The lexer and parser tests are combined, as for every parser test we would have needed to invoke the lexer anyway. Therefore, the output of a lexer test, a token stream, is passed to the parser to create an AST from it. Then, the state of the parser is checked for errors. As the tests are the same for the parser and the lexer we have about 7'000 test cases for testing the recognition capability. In the whole boost library (version 1.42), there are only seven cases which cannot be recognized by C2P0 correctly:

- interprocess/sync/xsi/simple_xsi_semaphore.hpp - this file contains an unclosed `#if` directive.

- iostreams/detail/broken_overload_resolution/stream.hpp - the `#endif` directive of the include guard is followed by the macro name (of the include guard), instead of a comment containing that macro name.
- mpl/and.hpp - defines a macro with name `and`. But `and` is a digraph [Dra10a] ([lex.digraph]) and actually shall not be used as a macro name. This is even mentioned in a comment inside that header file.
- mpl/or.hpp - here we have the same problem as in the `and.hpp` header.
- spirit/home/lex/lexer/lexertl/functor_data.hpp - does not end with a *newline* character
- typeof/typeof.hpp - while there can be parentheses surrounding a macro name when following the `defined` keyword, the standard does not allow to have a macro name embraced by parentheses in the `#ifndef` directive.
- spirit/home/classic/utility/rule_parser.hpp - at the end of line 422, there is a whitespace character after the backslash for line splicing.

The problems that occur while parsing the files above are all caused because they are not conforming to the C++ standard. We have just exclude these cases, but keep them in mind for possible extensions cases to make the parser more forgiving.

Beside the test cases generated from the boost library, we have further cases defined for specifically checking AST construction in the `C2POParser`. These cases derive from `PreprocessorTreeTest` and compare input source code to the text representation of the resulting AST. In Listing 4.88 we have an example tree test case.

```
@Test
public void testElifDirective()
{
  String sourceCode =
            "#if 1\n" +
            "int i = 1;\n" +
            "#elif 2\n" +
            "int i = 2;\n" +
            "#else\n" +
            "int i = 3;\n" +
            "#endif\n";
  String expectedTree =
            "(FILE (BLOCK (LINE (CONDITIONAL_INCLUSION " +
            "(if (CONDITION 1) (BLOCK (CODELINE int i = 1 ;))) " +
            "(elif (CONDITION 2) (BLOCK (CODELINE int i = 2 ;))) " +
            "(else (BLOCK (CODELINE int i = 3 ;))) endif))))";
  createAndCompareTree(sourceCode, expectedTree);
}
```

Listing 4.88: Tree Test Case

The string representation of a tree is quite simple: The first node embraced by parentheses is the root node of that tree, every further node is a child. For creating a tree structure, a child can be a tree itself. Everyone familiar with LISP might recognize that kind of representation.

C2P0Printer While the lexer and the parser can be tested easily, the `C2P0Printer`, which performs most preprocessing tasks, is much harder to test. Every feature, that manipulates the input source files has to be tested with the expected output specified. Nevertheless, it is important to have this functionality tested thoroughly. According to private conversation with Prof. Peter Sommerlad, the supervisor of this master thesis, people of the C++ Standard Committee suggested to use the boost preprocessor metaprogramming library [KM10] for testing our preprocessor. We did so by extracting over 100 test cases from the documentation of that library, crosschecking the result with the corresponding output of the GNU CPP [Pro]. Through these tests, we expect to have covered a very wide variety of possible preprocessing test cases.

The implementation of the `C2P0Printer` tests is similar to the file-based tests of `C3P0`. They have a section for input and a section for the expected output, separated by a certain comment. This allows to use the same file in another preprocessor without any further modifications, as both sections should contain preprocessable code. Listing 4.89 shows such a test case.

```
#include <boost/preprocessor/list/append.hpp>

#define L1 (a, (b, (c, BOOST_PP_NIL)))
#define L2 (x, (y, (z, BOOST_PP_NIL)))

BOOST_PP_LIST_APPEND(L1, L2)
   // expands to (a, (b, (c, (x, (y, (z, BOOST_PP_NIL))))))

/***test-input-end***/
(a, (b, (c, (x, (y, (z, BOOST_PP_NIL))))))
```

Listing 4.89: `C2P0Printer` Test Example

Beside the test cases derived from the boost preprocessor documentation, we have created further tests, also in the form of file tests, from the description of the preprocessing behavior and the examples in the preprocessing section of the C++ standard draft [Dra10a]. Currently, we have over 200 test cases verifying correct handling of preprocessing directives which invoke more than 250'000 macro expansions during execution. Additionally, to the unit tests described above, we have also tested C2P0 with the hello world program, including `iostream`. We had been able, after setting the paths and predefined values, to successfully preprocess it, using the MinGW [Min10] include headers. We had been able to compile the resulting preprocessed file using MinGW GCC, version 3.4.5. The resulting program could be executed, greeting the world enthusiastically!

Predefined Macros

In Section 4.2.8, we have described the predefined macros of the standard. Below we have a short description of the implementation:

- The macros `__cplusplus` and `__STDC_HOSTED__` just take an integer value, which is passed on construction. They are implemented as `IntegerMacro`s, which take a name and an integer value. Expanding these macros creates a node representing that value.
- The macros `__TIME__`, `__DATE__` and `__FILE__` take the string representation of their value. They are implemented as `StringMacro`s, which take a name and string as value. Expanding these macros creates a node representing that string literal.
- The function-like `__Pragma()` macro, has its own class. When it is expanded it returns an empty node, letting the macro vanish.
- The `__LINE__` macro is a bit special. It requires the corresponding node that represents the line macro to be expanded in the source code. From this node the current line number is deduced and the corresponding node is created.

4.3.12. Deficiencies

The capabilities of C2P0 are quite reliable and through the large amount of test cases, including real code, we are confident to have a preprocessor being able do deal with most input. Nevertheless, there are deficiencies we are aware of, that might need to be improved in the future.

Performance

The approach to tokenize a file while preprocessing and use a parser and tree grammars for manipulations is not as efficient as a character-oriented approach could possibly be. For example, preprocessing the hello world code, mentioned in Section 4.3.11, takes almost 2 seconds. Creating the whole structure takes a lot of time and modifying the tree structure using tree grammars, in the way we do this, is quite slow. We have a lot of unnecessary nodes created in the tree processors. To give a rough estimation for the hello world code:

- The resulting code consists of 425'000 characters.
- Over 1'600'000 `C2P0Tree` nodes are created.

Even if we assume there is a token for every character in the output file, we still need almost four nodes per token to create the final representation.
There are possible targets for optimization:

- ANTLR, when using tree grammars like we do, duplicates the nodes when rewriting them, to avoid modifications with side-effects at unexpected locations. For example, when having the same node in two different subtrees and that node receives a new child, this affects both subtrees. It is possible to turn the duplication off (by setting the `rewrite` option), but this requires changing other parts of the infrastructure, like the type of node stream used. Furthermore, we need to make sure this optimization does not suffer from the side-effects mentioned before.
- Perhaps the grammars can be optimized, by avoiding predicates through restructuring. Lookahead in this way takes time, as the target input has to be evaluated at least twice.
- In some cases, we might even avoid tree grammars completely. But in our opinion, this is a trade-off regarding understandability, as the modifications in tree grammars are easier to understand than modifications in source code.
- The combination of C2P0 and C3P0 will profit of avoiding the character-oriented intermediate representation from the preprocessor, by directly passing the AST as input to C3P0. Some filtering will be necessary though, to transform the nodes, which will take time as well.

C3P0 Purpose

The output of our preprocessor is clearly focused on C3P0 for further processing steps. While it is possible to compile the resulting code, at least for our hello world application, it does not have the best suited form. The C2P0 result is free of all directives, like the output of other preprocessors as well. But for example, the output of CPP for the GNU compiler still contains compiler-specific file and position offset information for creating expressive error messages. Currently, we do not provide such extensions and therefore, are of limited use for other. Nevertheless, it is not impossible to implement further features for adapting to such requirements.

Duplicate Macros

The C++ standard allows to define the same macro twice, if the replacements, and for function-like macros, the parameter lists, are identical [Dra10a]([cpp.replace]/2). In the current version, we do not check whether two definitions of the same macro are identical or not. We just check whether the name of a macro has already been defined and then take the latter definition. The user is informed about the redefinition.

Lexical Deficiencies

As mentioned before, line splicing is not possible everywhere yet. Most common cases, when splitting lines between tokens should be treated correctly. Furthermore, we are not treating trigraph sequences yet. While line splicing can result in tokens that are currently recognized as two, which is a major flaw of the token-oriented preprocessing

approach, it can be an advantage, when implementing trigraph handling. Because these transformations shall not affect raw-strings, we can exclude them and do not need to undo such replacements.
It is possible that we receive a node with an incorrect node type when using the concatenation operator in function-like macro replacements, because we do not rescan the resulting string representation, to determine the effective type. This it is not a serious flaw, as long as we are creating a character-oriented output stream.
The last known lexical deficiency in C2P0, is the lacking concatenation of two adjacent string literals.

4.3.13. Whitespace Handling

In the last week of the master thesis, we had again worked through the code and the test cases. For the preprocessor we could discover a minor flaw remaining. Regarding whitespace handling we had been able to create a failing test case:

```
#define f g h
#define g
#define h

(f) // should actually expand to ( )
```

Listing 4.90: Failing Whitespace Test

The preprocessor expands the code above to `()`. The space, expected to be between the parentheses, is lost. Unfortunately, we had not been able to fix this problem quickly.

4.4. Conclusion

Although not being a simple topic, we have managed to implement a preprocessor for C++0x. Our approach to do so is probably quite out of the ordinary. We do not expect it to be common that a preprocessor works token-oriented like ours does. The main reason seems to be performance, as a character-oriented C application can hardly be topped regarding efficiency. Nevertheless, we think our implementation more is understandable and, considering future work, it will be a better solution regarding integration into C3P0.
The capabilities of our preprocessor satisfy our known requirements. There is a test suite containing tricky cases, including examples from the boost metaprogramming library and the C++ standard. So far, we had been able to satisfy all requirements emerging from these code snippets. To cover macro expansion, including the special properties regarding recursion and exclusion, we had to make quite some effort. Working with example code has revealed some mistakes in the documentation of the boost metaprogramming library and the C++ standard draft [Dra10a] as well. In Chapter A, we have described the mistakes, which we also have reported.
The preprocessor provides the position mapping to the original location of code, which is essential for constructing an AST for automated refactorings. Unfortunately, we had not been able to come to the point in AST construction where we really needed this feature.
At the beginning of the second milestone, we had underestimated this whole task and it took more time than initially expected. But, we are confident with the result. During the implementation, we have learned a lot about the benefits and the use of tree grammars. Some deficiencies still exist. We do not consider them show stoppers though. Thus, we can continue our work on C3P0, which is constructing a CDT-AST.

5. AST Construction

In this chapter, we describe the third milestone of the C3P0 master thesis. As mentioned in the introduction, our focus lies on providing a CDT-AST to make the automated refactorings support the upcoming C++ standard. We have overcome most obstacles encountered during parser and preprocessor development so far. Now, we are at the point where we adapt C3P0 to create a representation of a C++ program, which can be used for further analysis. The effective implementation, the problems occurred and the outlook for this part of C3P0 are documented below.

5.1. Introduction

For implementing automated refactorings used in integrated development environments, it is essential to have an expressive abstract representation of the target source code. The specific requirements to such a representation depends on the type of refactoring being performed. From our practical experience in creating such refactoring tools we know that abstract syntax trees, like the one available in CDT, significantly ease this task. Therefore, we consider it crucial to have the capabilities to represent the new features of the C++0x standard, as soon as it is released. Otherwise the automated refactorings currently available in CDT, will not work for new program elements and could not effectively be adapted to work with the new standard.
Extending C3P0 to create an abstract syntax tree also serves as a test to see how difficult the adaptation of the current infrastructure, to solve a real world task, is. First, we will try to achieve the construction of an AST that is comparable to the current representation available in CDT. If we have enough time, we will also add nodes for new features of C++0x, that are not available in CDT yet.

5.1.1. Target Abstraction

Creating a CDT-AST in C3P0 will mainly consist of identifying the rules for creating the corresponding CDT-AST node. We can illustrate this with a small example. Let us consider the following declaration:

```
int i;
```

Listing 5.1: Simple Declaration Code

The best possible visualization to show what happens in C3P0, when encountering this very simple piece of code, is the resulting parse tree, stripped by the parts showing the evaluation of the predicates. In Figure 5.1 we see our starting position regarding C3P0, showing the whole parse tree.

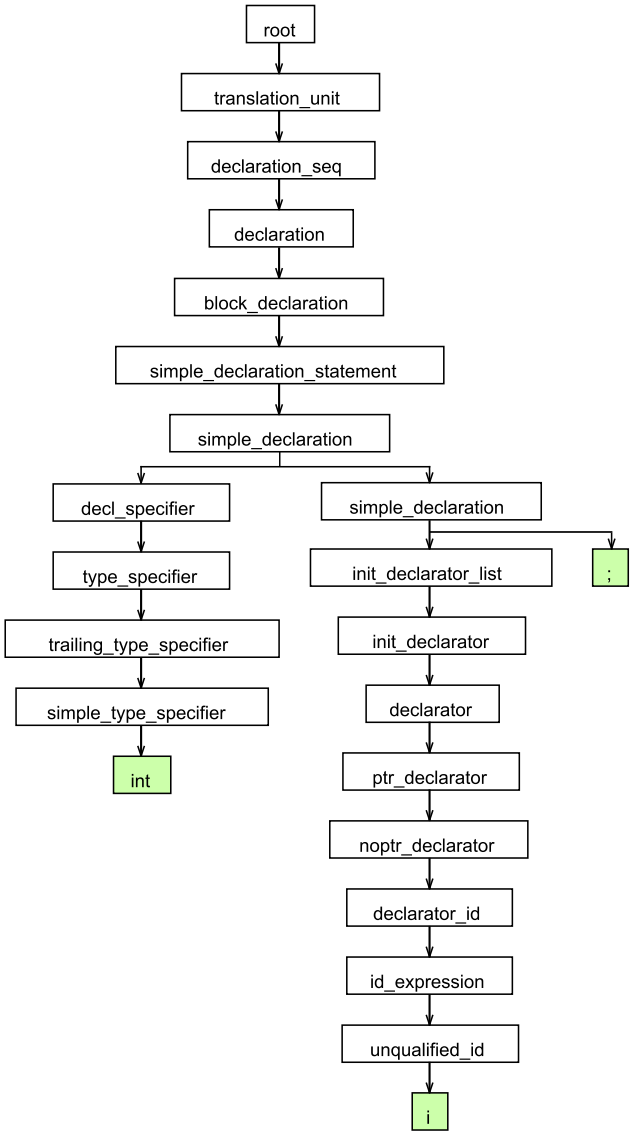

Figure 5.1.: Parse Tree for `int i;`

While the parse tree is far too complex and very close to the rule set of C3P0, CDT creates a much more abstract representation for the declaration above. In Figure 5.2 the CDT-AST for this declaration is illustrated.

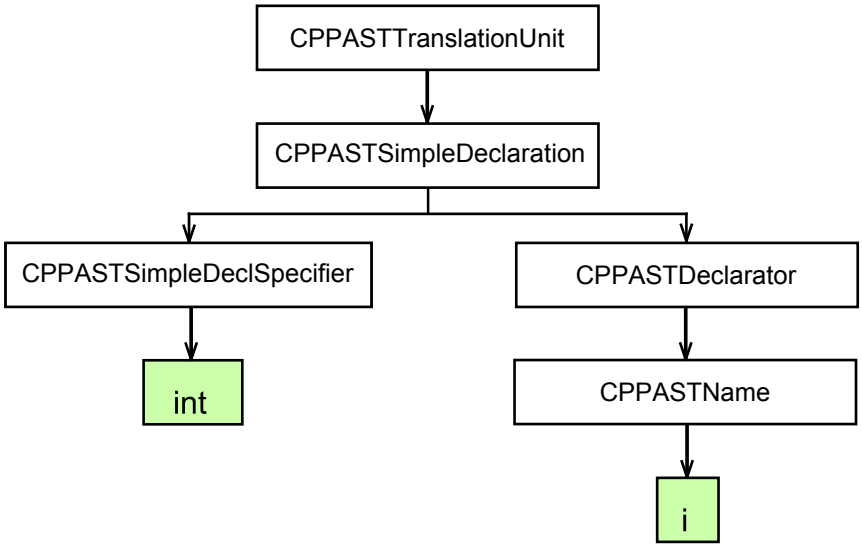

Figure 5.2.: CDT-AST for `int i;`

To identify which C3P0 rules provide the required information, and to construct the corresponding CDT-AST node, is the first task of extending C3P0. In the example above, we define the following responsibilities:

- `translation_unit` creates `CPPASTTranlationUnit`
- `simple_declaration` creates `CPPASTSimpleDeclaration`
- `simple_type_specifier` creates `CPPASTSimpleDeclSpecifier`
- `declarator_id` creates `CPPASTDeclarator`
- `unqualified_id` creates `CPPASTName`

Here we can see, that it is not necessarily the rule with the corresponding name – in the case of `CPPASTSimpleDeclSpecifier` – which must create the CDT-AST node, but the rule that has the required information. This avoids passing specific information for creating a node along the rule hierarchy. We are expecting the most difficulties where the CDT Parser, and the AST respectively, structurally diverges from our implementation of C3P0 and the C++ standard.
As seen in Figures 5.1 and 5.2, we also get rid of further syntactic elements like semicolons and braces when creating the CDT-AST. If necessary, these elements will be generated by the ASTWriter when creating source code from the abstract representation again.
The next section further describes the implementation of the AST construction and the integration into the parser rules.

5.2. Implementation

This section describes the implementation of the AST construction in C3P0. Furthermore, we summarize the challenges encountered and the limitations inherited from the current CDT-AST node infrastructure.

5.2.1. CDT Version

When we started the C3P0 term project, in September 2009, we have analyzed the infrastructure available from CDT. It was possible to create a translation unit and the corresponding nodes, using the public CDT infrastructure. As CDT, and especially the parts we are interested in, have been improved significantly since then, we have switched to newer releases during this last part of the master thesis. At the end of our development work, we have adapted C3P0 to the latest release of CDT, version 7.0.0, available at [Sch] This version already supports some features of C++0x.

5.2.2. CDTASTCreator

In the previous chapter we have described how C2P0, our preprocessor, has been implemented, with a very small set of parser rules. Most tree rewriting and actions producing the effective result, has been realized using tree grammars and the tree walkers generated from them. As we did not have C3P0 in the corresponding form for working on a resulting AST in similar ways, we went along with a more classical approach, and implemented the construction of the CDT-AST using actions in the parser rules.
Unfortunately, these actions bloat the grammar file and sometimes also obfuscate the relevant parts of the grammar. Additionally, it is tedious to develop the actions, written in Java code, for constructing CDT-AST nodes, without having code completion available in ANTLRWorks. Subsequently, we have introduced a super class for `C3P0Parser`, the `CDTASTCreator`. This allowed us to implement the corresponding actions for each rule mainly in Eclipse. To have most code for creating CDT-AST nodes outside the grammar, also keeps the actions in the rules as concise as possible.

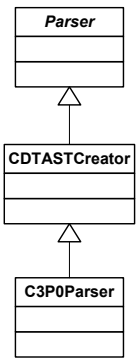

Figure 5.3.: C3P0 Hierarchy Including CDTASTCreator

5.2.3. Rules and Actions

The integration of the actions for creating the CDT-AST nodes, happens quite straight forward. A parser rule returns a CDT-AST node, which is created inside the rule itself or gets returned as a result of one of the rule parts. Below we look at some example rules for further explanations. We start with the root rule `translation_unit`.

Translation Unit

```
translation_unit returns [ICPPASTTranslationUnit unit]
@after{
  $unit = createTranslationUnit($decls.declarations);
}
  : decls = declaration_seq?
  ;
```

Listing 5.2: AST Construction Rule `translation_unit`

Compared to the previous version of the `translation_unit` rule we have made the following changes:

- Added a return value: `ICPPASTTranslationUnit unit`
- Added the `@after` action, for creating the returned `ICPPASTTranslationUnit`.
- Added a variable `decls` for accessing all declarations matched by the parser.

Below we see the implementation of the method creating the translation unit node, `createTranslationUnit` of the `CDTASTCreator`:

```
protected ICPPASTTranslationUnit createTranslationUnit(
                                  List<IASTDeclaration> declarations ) {
  ICPPASTTranslationUnit tu = nodeFactory.newTranslationUnit();
  if (declarations != null) {
    for (IASTDeclaration declaration : declarations) {
      tu.addDeclaration(declaration);
    }
  }
  return tu;
}
```

Listing 5.3: Method `createTranslationUnit`

In this rule it was easy to implement the construction of the corresponding CDT-AST node (`ICPPASTTranslationUnit`) in a neat way. The information necessary can be accessed through the variable assigned in the rule part (`decls`) and the whole construction is handled in the `@after` section, which is separate from the rule's alternatives.

Expression List

In the simple declaration example above, we have seen that there are several rules that do not create new nodes. They just pass the return values from parts of the rule, back to the caller. For example, `expression_list` is such a rule:

```
expression_list returns [ICPPASTInitializerList exprList]
@after{$exprList = $init_list.initList;}
  : init_list = initializer_list
  ;
```

Listing 5.4: `expression_list` Rule

Simple Declaration

The composition of some rules, like `simple_declaration`, had to be split up into several distinct parts. Unfortunately, it is not possible to conveniently handle the whole node creation in one single method call, in the `@after` part of the rule. As a result every alternative of the rule has its own action, for creating the corresponding CDT-AST node.

```
simple_declaration returns [String name, IASTSimpleDeclaration decl]
  : (init_declarator_list ';') => initDecl = init_declarator_list ';'
    { $name = $initDecl.name;
      $decl = createSimpleDeclaration($initDecl.declarators);
    }
  | declSpec = decl_specifier sDecl = simple_declaration
    { if($sDecl.name != null){
        $name = $sDecl.name;
      }
      $decl = $sDecl.decl;
      addDeclSpecifier($decl, $declSpec.declSpecifier);
    }
  | ';'
    { $decl = createSimpleDeclaration();
    }
  ;
```

Listing 5.5: `simple_declaration` Rule

The rule `simple_declaration` has a special composition, compared to the definition in the C++ standard. We have thoroughly described the reason, in the documentation of the term project [Cor10a], Section 5.4.3 Post Implementation Changes. Because the alternatives are completely different in what they create, which is also implied through the recursive implementation of the rule, we have decided to implement the actions in the alternatives.

Here, we also have the first disparity of the CDT-AST implementation and C3P0, or the C++ standard respectively. While the standard [Dra10a] defines a sequence of `decl_specifiers` to be part of a `simple_declaration`, CDT only defines one node for specifying the `decl-specifiers` in a`simple-declaration`.

```
simple-declaration:
  attribute-specifier? decl-specifier-seq? init-declarator-list? ;
```

Listing 5.6: `simple-declaration` Rule in the C++ Standard

This makes sense, as all `decl-specifiers` should be distinct – with the exception of the `long long` `simple-type-specifier`. But C3P0 does not, respectively cannot [Cor10a], summarize the single `decl_specifiers` to one single CDT-AST `decl-specifier` node in one common rule. Thus we have to merge a sequence of `decl-specifiers` with the `addDeclSpecifier` method, in the `CDTASTCreator`.

Labeled Statement

In certain cases, it had not been possible to just create the CDT-AST nodes and return them to the calling rule. One example for this problem was the distinct representation of labeled statements and the `case` and `default` labels of a switch statement. While the C++ standard [Dra10a] does not separate those two cases, they are represented differently in the CDT-AST.

```
labeled-statement:
  attribute-specifier? identifier : statement
  attribute-specifier? case constant-expression : statement
  attribute-specifier? default : statement
```

Listing 5.7: Standard Rule for `labeled-statement`

In the excerpt from the C++ standard above, we see that a labeled statement decorates another statement. In every alternative we have another trailing statement. Looking at the corresponding nodes of the CDT-AST, reveals that the node representing the labeled statement, which corresponds to the first alternative, indeed decorates a statement:

```
public class CPPASTLabelStatement extends ASTNode implements
        IASTLabelStatement, IASTAmbiguityParent {
  private IASTName name;
  private IASTStatement nestedStatement;
  ...
}
```

Listing 5.8: Class CPPASTLabelStatement

So far C3P0 is consistent with the CDT-AST. Looking at the switch statement labels, `case` and `default`, shows the incongruity. In the CDT-AST they are not represented by a `CPPASTLabelStatement`, but they have their own node types. They do not decorate another statement:

- `CPPASTCaseStatement`
- `CPPASTDefaultStatement`

This by itself is not a major problem, as we have the alternatives. The rule could create the corresponding node in each alternative. The problem arises from the fact that these two nodes are a statement by their own, and do not decorate another statement.
To cover this case we have adapted C3P0 to be closer to the CDT-AST and added an alternative to the `statement` rule. This separately covers the labels of a switch statement:

```
statement returns [IASTStatement statement]
  ...
  | sls = switch_label_statement
    {$statement = $sls.statement;}
  | (labeled_statement) => label_stmt = labeled_statement
    {$statement = $label_stmt.statement;}
  ...
  ;

switch_label_statement returns [IASTStatement statement]
  : 'case' const_expr = constant_expression ':'
    {$statement = createCaseStatement($const_expr.expression);}
  | 'default' ':'
    {$statement = createDefaultStatement();}
  ;
```

Listing 5.9: Alternative for Switch Labels

Member Function Bodies

The last special case we want to mention, regards our special way to handle forward references, described in Section 3.4.5. As we do not parse a member function body, defined inside the type specification, we cannot create the nodes for the body directly, while recognizing the function:

```
member_function_declaration returns [IASTDeclaration decl]
@after{ $decl = $fun_sig.funDef; context.leaveScope();}
  :   attribute_specifier? fun_sig = function_signature
      ('{' | 'try') { skipMemberFunctionBody($fun_sig.funDef);}
  ;
```

Listing 5.10: Changed Rule `member_function_declaration`

For overcoming this obstacle, we have extended the `DeferredParseTask` to take an `ICPPASTFunctionDefinition`, which represents the corresponding function. To check whether the function body can be recognized correctly, we use C3P0 again, by invoking the `function_body` rule, at the skipped position. The return value of this rule contains the statement node representing the body. This node can then be set as the function body of the `ICPPASTFunctionDefinition`.

5.2.4. CDT Limitations

Several alternatives in the C3P0 rule set could not have been adapted to create correct CDT-AST nodes, for the very simple reason that there had not been a corresponding node, to represent the feature. Below we have listed the parts missing in the CDT-AST to fully cope with the new standard:

Lambdas Currently, there are no nodes to cover the representation of lambda expressions available in CDT. It will probably not be very difficult to define new nodes for implementing lambda expressions. It might require the following nodes for adding lambdas to the CDT-AST:

- `CPPASTLambdaExpression` for representing a lambda expression in general, in a CDT-AST.
- `CPPASTLambdaDeclarator` for representing the declarator of a lambda expression, declaring its parameters, indicating whether it is mutable, recording all possible exceptions thrown and to contain the return value.
- `CPPASTLambdaCapture` for representing the lambda capture. This extra node type could probably be omitted by storing the capture information in the `CPPASTLambdaExpression` directly.

Decltype The released version 7.0.0 of CDT is already aware of declaration specifiers having type `decltype`. There is also the infrastructure for storing the corresponding expression in the `CPPASTSimpleDeclSpec`. But the CDT ASTWriter is unable to create code for them. Furthermore, CDT is currently unable to use `decltype` as a part of a nested name specifier, like described in [Van10b]. If treated similar to other nested names, there might be a new type of `IASTName` node, representing such a name part: `CPPASTDecltype`, which is aware of the expression enclosed.

Literal Operator Operators of the form `operator "" identifier`, for defining user defined literals, are not supported by CDT yet. Other operators are represented by either `CPPASTConversionName`s or `CPPASTOperatorName`s. In our opinion a subtype of `CPPASTOperatorName`, augmented with the corresponding suffix, seems to be the most suitable solution. As according to the C3P0 rule set, a literal operator id can occur in the same context as an operator-function-id.

Range-Based For CDT does not provide a node for representing a range-based `for` statement. A `CPPASTRangeForStatement` would need to take a declaration of

the loop variable (`IASTDeclaration`), and an expression (`IASTExpression`) or a braced initializer list (`ICPPASTInitializerList`). Furthermore, it would require the body (`IASTStatement`), which is common to the existing form of `for` loops.

Attributes Currently, CDT is unable to recognize attributes at all. How to handle them will require some fundamental decisions about the integration. In the first place, it might be easy to define the representation. A `CPPASTAttribute` node could take the attribute definition, consisting of an attribute token, represented by scope-separated identifiers and an argument clause. Attributes can occur at many different places, one ought even say they are similar to comments. Therefore, it is debatable where they should be placed in the AST. We know the following possibilities:

- Let a proxy node decorate existing nodes
- Introduce a field in the uppermost type of the node hierarchy, `ASTNode`. The recognition can be integrated into the parser, like we did in C3P0.
- Collect them while preprocessing and create a map with the corresponding nodes as keys.

Compared to comments, there is a significant advantage regarding attributes: It is defined to which nodes they syntactically belong to.

Alias Declarations At the moment, CDT does not support alias declarations of the form `using identifier = type-id`. A typedef declaration in a CDT-AST is a declaration, containing a declaration specifier, with the storage class set to the `typedef` value. While alias declarations are semantically similar to a typedef declaration, it has a different syntax and most likely cannot be represented in the same way. We suggest to create a new declaration node type: `CPPASTAliasDeclaration`, which knows about the alias name (`ASTName`) and the declared type (`CPPASTTypeId`).

Constexpr CDT does not recognize the keyword `constexpr` yet. The `constexpr` keyword represents an additional modifier for C++ declarations, represented by a declaration specifier. Subsequently, the `ICPPASTDeclSpecifier` interface could be extended by `setConstexpr` and `isConstexpr` methods.

Thread-local Specifier In C++0x there is a new storage-class specifier `thread_local`. In the CDT-AST storage-class specifiers are also represented by declaration specifier nodes. Therefore, the `IASTDeclSpecifier` contains a constant defining the value for each storage-class. A new constant for the `thread_local` storage-class specifier would be required: `sc_thread_local`

Inline Specifier for Namespaces Currently, in CDT `inline` is a function specifier only. In C++0x `inline` can be a namespace specifier as well. As namespace definitions do not have declaration specifiers like function declarations have, the `inline` specifier has to be attached to the `ICPPASTNamespaceDefinition` interface. We suggest adding the methods `setInline` and `isInline`.

Default/Delete Specifier for Functions At the moment CDT cannot recognize explicitly defaulted or deleted functions. As `= default` and `= delete` relief the requirement to have a body statement in the corresponding function definition, we consider the `CPPASTFunctionDefinition` to be the right place to specify defaulted and deleted functions. As `= default` and `= delete` are mutually exclusive it could make sense to create constants for flagging the state in CDT manner.

nullptr Literal Although, test cases using the `nullptr` literal, like `int *pi = nullptr;`, create correct source code when the corresponding CDT-AST is written by an `ASTWriter`, `nullptr` is not recognized as the corresponding literal. Currently, it is represented by just a name (`ASTName`). CDT should introduce a new keyword for `nullptr` and extend the `ICPPASTLiteralExpression` interface by a constant for `nullptr`.

noexcept In the latest version of the C++ standard draft [Dra10a] `noexcept` has been introduced. CDT does not recognize that keyword yet. Here, we suggest the `ICPPASTFunctionDeclarator` to be extended by methods for checking this specifier: `setNoexcept` and `isNoexcept`. C3P0 currently creates a CDT-AST when encountering `noexcept`, which represents a function declaration stating `throw ()`. This is equivalent to `noexcept`.

5.3. Testing

To have verification of the results achieved, while implementing the actions to create the CDT-AST in C3P0, we have created a set of about 250 new test cases.

5.3.1. Test Depth

Testing whether a CDT-AST, generated by C3P0, is as expected, could happen at different detail levels:

Create a Writable AST The most superstitious way to test the functionality of the CDT-AST creation in C3P0 is create an AST, while parsing the target code and check whether the resulting CDT-AST can be rewritten, using the CDT `ASTWriter`. This assures that our ASTs do not have missing child nodes and its nodes are valid, regarding their hierarchy.

Compare String Representation After creating, and writing the AST, we could additionally compare the resulting string representation to the string representation of the CDT-AST, created by the CDT parser for the same code. In this way, we can check, with the limitations of the `ASTWriter`, that we can recreate the same code representation, using the CDT parser and C3P0, for the tested C++ source code. And, as a further advantage, our tests are not prone to changes in the ASTWriter, like comparing generated source code to the input source code would be.

Check Node Details To have a complete check for identity of two translation units, we would need to compare the CDT-ASTs, of the CDT parser and C3P0, recursively. This would include supplement information about preprocessing statements, in the translation unit, as well as location information, in every single node. This ensured exact identity of two ASTs.

Due to lacking time remaining in this master thesis, we have not been able to implement the specification of every single detail of a CDT-AST node, like correct position information. Thus, we decided not to let our tests compare the CDT-AST constructed by C3P0, to a CDT parser generated AST in every detail. We consider the comparison of the written code to provide enough equality, at the current state of C3P0.

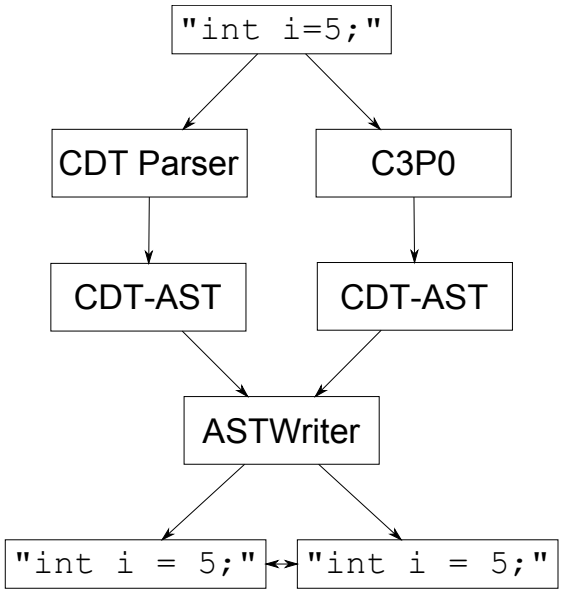

Figure 5.4.: AST Construction Test

5.3.2. New Tests

Unfortunately, it had been necessary to create new test cases, as it was not possible to use the existing C3P0 tests, which are directly testing the C3P0 rules. Our test set, for verifying the functionality of C3P0, could not be used, because the CDT parser, for creating a reference CDT-AST, always needs to be able to parse a complete translation unit. A translation unit always consists of any number of declarations. Thus, it is not possible to test expressions or statements directly.
In the example below, we have test code for the `delete_expression`. While the test for the corresponding parser rule in C3P0 would target the delete expression only (`delete a`), it is impossible to use this test as is with the CDT parser. Subsequently, our AST creation test needs some kind of framework that allows testing a delete expression,

without creating too much overhead. The simple declaration, taking an expression as an initializer comes in handy here.

```
void v = delete a;
```

Listing 5.11: Delete Expression Test Code

Although, it does not make sense semantically to have a variable, declared void, getting initialized by a delete expression. But, this is an possible way to create a test for a delete expression.

5.3.3. ASTWriter Problems

As CDT is catching up with the new features of C++0x, we already had some of them available in the CDT-AST. Unfortunately, the `ASTWriter`, for creating code from a CDT-AST, has not been updated to cope with these new possibilities. Furthermore, we encountered some cases, independent of new features, which were not written correctly. If possible, we have provided patches for those flaws of the writer, to get it C++0x-ready. Below we have summarized the problems occurred:

auto The `auto` keyword, as a type specifier, gets recognized by CDT. But, when writing the corresponding declaration specifier, the type value `t_auto`, indicating the auto type, is not recognized by the `DeclSpecWriter`. We have created a patch to fix this issue.

long long The `DeclSpecWriter` would be able to write `long long` specifier, that are a possible part of every `IASTSimpleDeclSpecifier`. But, it only prints it if the declaration specifier is of type `ICASTSimpleDeclSpecifier`, which a CPPASTSimpleDeclSpecifier, used for C++, never is. We have created a patch, which removes this restriction and always prints the `long long` qualifier, if it is set.

Pack Parameters With the introduction of variadic templates, parameters for template declarations can be made a pack parameter, taking a variable number of arguments. The CDT-AST is already capable of flagging a parameter as a so called pack parameter. Currently, the `TemplateParameterWriter` is not aware of pack parameters and subsequently does not print the corresponding ellipsis. We have created a patch, which extends this functionality of the `TemplateParameterWriter`.

decltype `decltype` as a simple type specifier is already implemented in the CDT-AST, but cannot be written by the `DeclSpecWriter`. We have created a patch for making it aware of the `decltype` specifier. This does not include writing of the `decltype` expression.

Scopes Although not a new feature to C++, CDT does not write the leading scope qualifier (`::`) in fully qualified name node. Our patch for the `NameWriter` fixes that.

Trailing Return Type The `DeclaratorWriter` had not been able to print trailing return types, which are already implemented in the CDT-AST as well. Our patch fixed this.

Further Patches During our work with the CDT-AST and the `ASTWriter` we have created further patches fixing typos, ugly naming and other minor flaws.

5.4. Deficiencies

At the current state C3P0 cannot compete with the CDT parser in every aspect of recognizing C++ code. We have managed to create a parser, focused on the new C++ standard as a whole. Regarding recognition of new features, C3P0 is superior for the moment. Nevertheless, the CDT Parser has been developed and proven to work over years, bearing experience that cannot easily be caught up with during the relatively short development time of C3P0.
Eventually, C3P0 is limited to the existing CDT-AST infrastructure, regarding usefulness of its recognition power. Meaning we could actually cope with new C++0x code, but do not have a suitable representation at hand. As we ran out of time in this master thesis we had not been able to invest great effort into creating and integrating new CDT-AST nodes.
There are some further issues regarding CDT-AST construction that are not completely satisfying for us.

Restrictive Recognition The basic concept of C3P0 is to have some decisions relying on the known types and namespaces as well. This has been implemented in the previous term project and the first milestone of this master thesis. In some cases, these preconditions can prevent a CDT-AST being created for incomplete pieces of source code. Look at the following example:

```
typedef int T;
T t;
```

Listing 5.12: Simple Declaration Example

C3P0 can correctly create the CDT-AST for this translation unit including both declarations. But if we remove the definition of `T`, `T t;` cannot be recognized anymore. The CDT parser on the other hand assumes `T` to be a type and creates an AST for that code anyway. Therefore, C3P0 is more restrictive and less forgiving regarding lacking type information.

While it actually would be correct not to recognize `T t;` alone as correct, we consider the recovery in such cases to be part of proper error-handling. Which might report an error during static analysis, but create a CDT-AST node in this case anyway. Although such a node can only be a guess in an incomplete program, it might at least retain the possibility for modifications and other AST-related tasks.

Action-Bloated Rules In the current form, most functionality of C3P0 is implemented in the parser grammar and the related classes. During the implementation of C2P0 we have seen the advantages of tree grammars, when executing actions on specific AST substructures. For constructing the CDT-AST, a tree grammar would be perfectly suitable. Unfortunately, this is not possible in the current state of C3P0, as the parser does not create an AST and rather performs all tasks in the embedded actions. The result is a rule set that has actions bloated with CDT-AST construction statements and also an extension of the rule signature for returning the nodes created. It is particularly cumbersome if, additionally to the CDT-AST construction, context management for the symbol table is mixed in the actions. Although it is not hard to distinct the actions from the rule parts, the overall purpose of the rule gets separated by these actions, obfuscating the other parts.

```
nested_name_specifier returns [Scope lookupScope, List<IASTName> names]
@init{
  $names = new ArrayList<IASTName>();
}
  : ( type = type_name
      {
        $lookupScope = context.resolveScope($type.typeName);
        $names.add($type.nameNode);
      }
    | namespace = namespace_name
      {
        $lookupScope = context.resolveScope($namespace.namespaceName);
        $names.add($namespace.nameNode);
      }
    )
    SCOPE
    ( ( (IDENTIFIER | TEMPLATE? unverified_simple_template_id) SCOPE )=>
      { context.enterLocalLookupScope($lookupScope); }
      nnp = nested_name_part
      {
        context.leaveLocalLookupScope();
        if($nnp.lookupScope != null) {
          $lookupScope = $nnp.lookupScope;
        }
        $names.add($nnp.nameNode);
      }
      SCOPE
    )*
  ;
```

Listing 5.13: Nested Name Specifier Rule

Having C3P0 split up into a parser creating an AST, and tree walkers for symbol resolution and CDT-AST construction, would improve cohesion for the distinct tasks. Furthermore, we probably could modularly decide whether to use the static verification of types or not, before applying the CDT-AST constructor, relieving the problem of not being forgiving on incomplete code.

Declaration Ambiguity In some specific cases, C3P0 is unable to create the correct CDT-AST without further adaptations of the grammar. The following example illustrates one:

```
void foo() {
  typedef int T;
  T(i);
}
```

Listing 5.14: Confound Recognition

As `T` is a known type and `i` is undefined, `T(i);` declares the variable `i` of type `T`. Meanwhile CDT recognizes this case correctly and creates a CDT-AST representing a simple declaration. C3P0 actually does recognize this code as correct, but does not use the correct rules for recognition. Subsequently, it creates incorrect CDT-AST nodes. The actual problem origins from the parser implementation in the term project, but remained unrecognized as previous test mainly tested the recognition power of C3P0 as a whole.

Analyzing the behavior of the CDT parser in this case revealed that it is taking a decision based on the knowledge about `T` and `i`. We have examined the outcome of the following combinations of kinds for `T` and `i` in the statement `T(i);`, getting the following result:

All declarations were declaring variable `i` of type `T`. The table indicates that as soon as we have a variable in play, the statement is interpreted as a function call by the CDT parser. Unfortunately, C3P0 is not yet able to resolve variable symbols. Subsequently, we can hardly reproduce the behavior of CDT in the same way.

To overcome this specific drawback, we have added an additional alternative to `simple_declaration_statement`. If we have a leading simple type specifier and a declarator, the new alternative is matched. It solves this problem, but does still not provide completely the same behavior as the CDT parser.

Kind of T	Kind of i	Resulting Node
unknown	unknown	Declaration
unknown	variable	Function Call
unknown	type	Declaration
type	unknown	Declaration
type	variable	Function Call
type	type	Declaration
function	unknown	Declaration
function	variable	Function Call
function	type	Declaration
variable	unknown	Function Call
variable	variable	Function Call
variable	type	Function Call

Table 5.1.: CDT Parse Result of `T(i)`

5.5. Conclusion

The last milestone, constructing a CDT-AST, was the first application of C3P0 to use the parsers capabilities. Eventually, we are able to construct CDT-ASTs that are close to the trees produced by the CDT parser itself. Subsequently, we have successfully used the previous implementation of our parser's infrastructure and shown that it works in most cases. We also know from this milestone that there are deficiencies we should improve regarding the recognition power and the selection of the correct rules. Mostly, these improvements rely on the fundamental feature to have a complete symbol table at hand, for resolving every possible name in the source code.
Regarding completeness of the CDT-AST and all information contained therein, there are clear goals for further improvements.

5.5.1. Outlook

Due to limited time, we have left out the following parts while constructing the CDT-AST:

Position Information The CDT-AST created by C3P0 is lacking all position information yet. In the previous part of this master thesis, the development of the preprocessor, we have prepared position resolution. The position map created can be used to set position information while creating the CDT-AST nodes, even in preprocessed code containing source file inclusion and conditionals, as well as macro expansions.

Preprocessor Directives In the CDT-AST all preprocessor directives are collected in the `CPPASTTranslationUnit` node. C3P0 does not add such information yet. It would be necessary to collect the directives encountered while preprocessing, for retaining them for CDT-AST construction. This also includes comments, which

are commonly treated as superfluous whitespaces, that get thrown away in the preprocessor.

We doubt it makes sense to further push CDT-AST construction with C3P0. The CDT parser and the corresponding indexer have been significantly improved. There is ambition to cope with the new standard and several steps have been taken to achieve this goal in CDT. Subsequently, there will not be much benefit in having an additional parser being able to create a CDT-AST too.

6. Further Improvements

Beside the milestones of the master thesis we have implemented some little extensions, which did not have a relation to any of the tasks planned. These modifications are described in this chapter.

6.1. Resetting Memoization

This is the realization of an idea we have already had during the term project. Memoization in ANTLR, when turned on, allows the parser to store the result of the evaluation of a syntactic predicate. This results in a tradeoff, increasing parse speed significantly, but also increasing memory usage of the parser. As the current implementation, due to many syntactic predicates, is terribly slow if memoization is turned off, we need to be able to activate it. Unfortunately, we cannot turn it on for every single rule, as in certain cases during lookahead, some syntactic predicates might fail, as specific actions, for adding new symbols to the symbol table, did not get executed at that point. After these actions are executed in the parse process, due to memoization the, then succeeding, predicates are not evaluated again, as the result is already known. Unfortunately, this yields an incorrect result.
Subsequently, our idea was to turn memoization on, but to reset the map containing the previously evaluated results every time we define a new symbol. A first attempt to implement this behavior was successful.
We have used the observer pattern to notify the parser about significant changes in the symbol table, which triggers a reset of the memoization storage (See Figure 6.1). Every time a new symbol is defined, the symbol table is set as changed and the observers, in our case just the parser, is notified about this change. As reaction, the parser resets the memoization map.

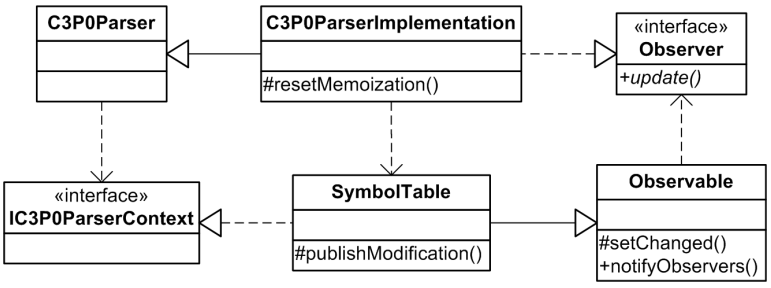

Figure 6.1.: Observer Pattern Introduced to Parser / SymbolTable

6.1.1. Performance Improvement

Performance is slightly impaired by this change, but removing the requirement to figure out for each rule whether memoization has to be turned on or off is worth the impact on performance – at least while developing the parser, which could imply several changes to these settings. Compared to a setup having memoization disabled, to ensure every condition is evaluated for the actual state, we still have a significant improvement.
The actual numbers depend on the test case. Most significant improvement was observable in `testNestedParameterList`. With memoization disabled, parsing the member function declaration, in Listing 6.1 takes more than 8 seconds. Activating memoization, including our reset mechanism, reduces the test execution time to about 0.001 seconds.

```
class A{
public:
  int foo( int (*a)(int b,int (*c)(int d) ) );
};
```

Listing 6.1: Test Code for Nested Parameter List

On average the improvement is smaller. But we can still reduce the execution time of the whole test suite, containing more than 1'200 test cases, from over 20 to about 1.5 seconds.

6.2. File-Based Tests

With the preprocessed hello world program as input the failing cases got more and more context related. Previously, we had reproduced the failing cases in common JUnit test files, added to test suites. Each case had first defined the code under test and then set up the whole environment required for test execution. Depending on the case we had to vary the amount of context, which could be taken out of the source code string and added to the setup part as direct symbol table manipulations. An example for such a test case is listed below:

```
Test
public void testDeriveStructInTemplateClass() {
  //Test Code
  String sourceCode =
      "template<typename T>\n" +
      "struct Tpl{" +
      "  struct Base{};\n" +
      "  struct Derived : Base{" +
      "    typedef int _S_empty_rep_storage\n;"+
      "    static Derived& _S_empty_rep(){\n" +
      "      void* __p = reinterpret_cast<void*>(&_S_empty_rep_storage);\n" +
      "      return *reinterpret_cast<Derived*>(__p);\n" +
      "    }\n" +
      "  };\n" +
      "};";
  //Setup
  C3P0ParserImplementation parser = createParser(sourceCode);
  predefineTemplate(parser, "reinterpret_cast");
  //Execute & Assert
  executeTranslationUnitTest(parser);
}
```

Listing 6.2: Example Test Case

Especially when having C++ code that consists of several lines, this approach can be quite annoying. Opposed to C++0x, Java does not support raw strings or a comparable feature to have large chunks of structured characters, which is particularly cumbersome here. Furthermore, the formatter in Eclipse also affects such string arrangements. In previous projects we had used test generators which took structured ascii files and created test cases from them [CFS06]. As this came in handy, we decided to implement this here as well in a simpler form. Currently, we do not need to have a configuration section. If we come to the point where this is required, we can easily add it.
Subsequently, it is much easier to create test cases with source code consisting of many lines. Furthermore, these source files can be taken as is and passed as input for a compiler to check their validity. There is only one drawback: We cannot easily rerun a specific test from the JUnit window in Eclipse, as the test cases are created dynamically during test suite execution. As a simple workaround we have created a separate test suite which executes the cases of a specific folder, where we can drop in cases, to have a small test suite containing only what we are currently interested in.

7. Project Results

This chapter summarizes the results this our master thesis. Furthermore, we describe open tasks and have defined possible follow-up projects.

7.1. Achievements

Starting with the outcome of the preceding term project, we have continued to work on C3P0 during the 21 weeks of this master thesis.

7.1.1. Symbol Tracking

In the first milestone we have developed a concept to handle symbol tracking successfully. Although we could not implement every single detail of the new C++ standard, due to time restrictions, we have shown our approach to work and be applicable to C3P0. We have encountered several difficulties, especially regarding the integration into the ANTLR grammar, due to predicate evaluation and action execution. With minor changes to the rule set we have overcome these obstacles. Eventually, symbol tracking and lookup is possible with our approach.

7.1.2. Forward Lookup

One part of the first milestone has been dealing with forward references in class members. Our solution to postpone the parsing of their bodies works very well and does not imply significant overhead, as we can stick with our one-pass approach. There are no known deficiencies regarding this way to handle member function bodies.

7.1.3. Preprocessing

It had been an open task to deal with preprocessing directives, since the end of the term project. In the second milestone, we have successfully solved this problem for C3P0. Although it is not an integral part of C3P0, we have the possibility to preprocess C++ code, in a separate component. The resulting output can then be parsed using C3P0. The preprocessor is capable of satisfying most requirements of the C++ standard. Beside writing preprocessed code, it creates a position map that allows retrieving the original source code position, including offset and file name, for every token in the output stream. If possible, we would like to avoid the intermediate string representation and directly advance from the AST created in C2P0 to any further task C3P0 might do.

We have a wide range of test cases that are satisfied by the functionality of C2P0. They ensure the current and future functionality of C2P0, and make the preprocessor application reliable and open for changes by others.

7.1.4. AST Construction

In the last milestone of this master thesis, we have changed C3P0 to construct a CDT-AST while parsing. For the integration of the constructing actions, we had to solve minor problems. Eventually, we have a parser that can construct CDT-ASTs for most code. We have seen that, regarding C++0x features, C3P0 is ahead of the CDT parser. Unfortunately, this advantage is of limited use, as the infrastructure provided by CDT to represent the new features does not include all of them yet.
This milestone has shown that C3P0 can be used for a real application. Nevertheless, it will hardly make sense to follow the idea to construct the CDT-AST using C3P0. It would take much additional effort to achieve the reliability and the complete feature set already provided by the CDT parser, which has evolved greatly recently.

7.1.5. Performance

Thanks to an improved reset mechanism of the ANTLR memoization feature, we could improve parse time significantly. Although heavily reducing the evaluation of syntactic predicates, our solution does not impair the recognition capabilities of C3P0. The problems, occurring during the term project, that selectively disabling memoization for certain rules could leave unrecognized cases left, have also been avoided.

7.1.6. Just Did It

After we started the C3P0 project, experts had called it an impossible endeavor to implement a C++0x parser like we did. Even though we do not have a perfect parser yet, we have achieved more than initially expected. Probably, C3P0 is the first C++0x parser implemented using ANTLR. Although, we are not claiming it was easy. Especially, finding a good point to start was challenging for grammar-related tasks, as almost everything is connected and relies on other parts. Thus, one can hardly isolate a problem to solve it independently.
We have successfully shown that it is absolutely possible to develop an application of this kind in test-driven manner. About this approach, we have been glad very often while changing existing functionality in C3P0, as we always had reliable indication whether our ideas worked out or broke existing behavior unexpectedly.
During the term project and even more in this master thesis, we have learned very much about the C++ standard and parser implementation using ANTLR. If we had to start over again we would start completely different, but we do not consider this a bad thing. Much more, this is an indication for increased experience.
All in all, we are proud of what we have achieved so far and hope to be able to push this project further to become useful in one or another way.

7.2. Outlook

C3P0 is not finished yet. There remains a lot to do on the parser. Below, we have described what we are next up to and what possible extension and improvement tasks for continuing the project could be.

7.2.1. CDT Contribution

The effort in CDT to cope with the new C++ standard as well reduces the need for a new parser. Subsequently, we are not sure whether it makes sense to continue following th initial goal to create a new parser for CDT. Although we are ahead in recognizing the features of the new standard C3P0 is lacking the long experience though real-world use compared to the existing CDT parser. It will probably less effort to add the missing features in CDT, than bringing C3P0 into this IDE. Furthermore, we are not sure whether we could compete regarding performance, as long as C3P0 is implemented for general use and not specifically optimized to create a CDT-AST. Nevertheless, we have some insight about the required AST nodes regarding the new features, which we will like to share with the CDT developers.
In the last milestone we have created several patches for adapting the ASTWriter of CDT closer to the new feature, already implemented. Thus, we have already a small contribution from our side.

7.2.2. Follow-Up Projects

The time available in the term project and this master thesis had not been enough to implement all possible features for C3P0. Below we have a list of extensions and improvements for our parser, ranging from small tasks to full assignments that might take several weeks to accomplish.

Integration of C2P0 into C3P0

In Chapter 4 we have described the implementation of C2P0, our preprocessor. Its lexer grammar has been implemented very close to the lexer grammar of C3P0. This had two reasons: First, it was not necessary to develop the rules for the `C2P0Lexer` from scratch, which saved much time. Second, when both lexers are alike, we have similar or even identical tokens. Subsequently, the nodes of the AST created in C2P0 have a close relation to the nodes in C3P0. Currently, we have a strict separation of C2P0 and C3P0 which interact only through the output of the preprocessor and the position map. This intermediate step could be avoided, which could make the `C3P0Lexer` obsolete. Eventually, the `C3P0Parser` could work on the AST created in the `C2P0Printer` directly. Changing that name then probably would make sense. We expect this task to be dividable into the following steps:

1. Decide how the new intermediate representation shall look like. Currently, we are creating an AST for the preprocessed code in the `C2P0Parser`, but it would be

required to define the exact content for using it in C3P0. For example, define how to represent inactive if-sections.

2. Adapt the tree construction (or rewriting) in C2P0 to create this AST.
3. Remove the `C3P0Lexer`.
4. Create a filter: As the C++ keywords are not recognized as such in a preprocessor, some `IDENTIFIER`s need to be transformed to the corresponding keyword tokens, that currently exist in C3P0. ANTLR provides the possibility to perform such a conversion using a filter grammar, that does not need to match a complete structure, but executes actions for the parts that match only and ignores the rest [Par07].
5. As there is no lexer in C3P0 anymore, there must be a token stream, created from the C2P0 output AST. Alternatively, the `C3P0Parser` grammar could be converted into a tree grammar, directly working on that AST as well.

When regarding the last part, changing C3P0 to become a tree grammar, this whole task can become quite an effort, taking about five to eight weeks to realize.

Error Handling

Neither C2P0 nor C3P0 are very expressive when they recognize an error in the source code, as we did not set the focus on providing helpful messages. Usually an error during recognition yields a message in the form: *"No viable alternative at input: XYZ"*, possibly including a position. If lucky, the grammars recover from the error by themselves and continue their work, in the worst case they cannot and stop. This behavior was okay while developing the grammars having unit tests, as one always knew where to look for the mistake. For a user of the parser, this is not acceptable.
ANTLR provides the possibility to catch exceptions at the end of a rule, where the parser can react appropriately and create expressive messages to inform about the error that occurred.
Our approach to improve error messages:

1. Hit the parser with a lot of different source code examples, that contain errors.
2. Collect information about each case regarding the error, locate the closest point in the grammar and define an expressive feedback.
3. Implement the handling of that errors and if possible try to recover the state of the parser to be able to continue in a sensible way.

Depending on the ambition one can spend an almost arbitrary amount of time for locating and handling further error cases.

Symbol Table: Further Symbols

C3P0 depends on information about known types to decide on certain rules. At the moment, we are just tracking templates and types and ignore any other symbol, like variables or functions. This limitation causes problems in certain cases where a local name would hide a type name. Extending the symbol table for further kinds of symbols that can be known, allows to further extend the recognition capabilities.
Furthermore, C3P0 currently does not distinct overloaded functions and templates. This does not impair the possibility to accept correct code, but denies to recognize mistakes regarding wrong number or types of parameters.
The resolution in our symbol table currently ignores visibilities. Everything is considered to be public and can be accessed from outside if declared. Having detailed knowledge about every symbol allows to implement further verification capabilities, or is even required for certain features.
Estimating the time required for possible extensions of the symbol table is difficult. Several parts of C3P0 have to be adapted:

- The grammar rules for recognizing the corresponding declarations.
- Integrating the symbols into the symbol table and recognizing the relations (hiding, overloading, conflicts, etcetera).
- Adding capabilities for the resolution of the new symbols.

Small extensions, like recognizing visibilities probably do not take more than one or two weeks, while complete tracking of variables and functions, including type, parameters and membership, will take several weeks to implement properly.

Type Deduction

The parser could check type compatibility. To do so it needs to be able to deduce the type of expressions, which can be rather easy if it just regards access to one variable. On the other hand it might become quite complex in expressions relying on implicit conversions and user defined operators. This could also solve the current problem of `decltype` scopes, described in Section 3.2.7.
There are patterns supporting to implement this functionality [Par09b], explaining the approach to perform type conversion. This extension also depends on certain improvements of the symbol table, described above.
After types of expressions can be deduced it is also possible to perform further checks like function and template overload resolution or verifying type compatibility of assignments. The overall effort to implement such functionality includes the extensions of the symbol table. As it relies on several feature, like accessing the type of a variable, knowing possible conversions and resolving entities of an overload set, improving the symbol table to be suitable for type deduction will take some time. Depending on the problems encountered, we are expecting this to take six to nine weeks.

Constexpr Evaluation

A further feature, depending on the symbol table extensions, is the evaluation of constant expressions. It is possible to evaluate certain expressions, constant expressions, at compile-time [Dra10a]([expr.const]/1). Subsequently, it should be feasible for a parser like C3P0 to determine the value of such an expression.
In an IDE, like Eclipse CDT, this could be used to show the value of such an expression in a tool-tip when hovering with the mouse cursor.
If the infrastructure from the symbol table is available, the evaluation itself should not be that complex. This feature should be implementable in one to two weeks.

Template Checks

There has been a feature discussed for the new C++ standard called *concepts* [DRS05]. C++ concepts were supposed to specify a type system for templates, which enables expressive messages for template-related mistakes. Unfortunately, concepts did not make it into the standard. Either way, better information about problems regarding templates would be desirable.
If C3P0 was able to deduce types at parse-time, or later in static analysis, and if it could create code for a template instance, compatibility of the template arguments to the template could be checked. In a further step, one could try to deduce constraints on the entities passed as template arguments.
We are not sure how many obstacles are encountered when implementing such functionality, but we are expecting this endeavor to be rather complex. Especially, as it depends on a lot of information about the symbols used and the specific templates. We are expecting to need at least several weeks to achieve something useful.

Performance Optimization

A major flaw in the implementation of C2P0 and C3P0 is performance. While ANTLR is not the perfect solution regarding this aspect, we are sure there is much potential for improvement even when using ANTLR. There might be syntactic predicates, evaluating whole rule alternatives for deciding, which could be optimized. There also is potential for optimization regarding the number of nodes created in C2P0 as well.
Although we did not focus on performance in the master thesis, we have been able to increase the performance of C3P0 significantly by enabling our modified version of memoization. There definitely are further possibilities for improvement.

- Analyze the runtime characteristics of C2P0 and C3P0 to figure out which operations are time-consuming.
- Assess the possible means to improve performance.
- Netbeans uses a performance-optimized version of ANTLR v2 for their grammar [Mic10b]. Perhaps ANTLR v3 can be optimized similarly.

- Replace certain parts of the tree grammars with code, avoiding to use a whole parser - Trade-off: Understandability

Analyzing the current implementation thoroughly and developing suitable improvements will take some time. Depending on the extent, it can take tree to eight weeks to achieve a remarkable improvement. This task is independent of others.

Introduction of a Tree Grammar for C3P0 Actions

Beside coupling C2P0 and C3P0, the `C3P0Parser` has lots of embedded actions for tracking types and creating the CDT-AST. It would be nice to have AST construction separated, at least, from the CDT-AST generation. Making the `C3P0Parser` just create an AST, a possible target for further tree grammars, allowed to implement further structure related tools. Therefore, the infrastructure could be used for several other tasks when working with C++0x code, without depending on the tasks currently performed by C3P0.
The separation of these elements will take time, as it implies creating a new rule set for the parser to structure the input tokens and to extract the current functionality to tree grammars. It will take approximately six to nine weeks, as this change will need fundamental changes in the general approach.

Further Ideas

Beside the tasks scratched above, several other C3P0-related follow-up projects are possible.

Conversion to ANTLR v4 Terence Parr is reworking ANTLR completely [Par10b]. Currently, there is no official release date for the new version of ANTLR, version 4. But as soon as it is released, one should think of porting the current parser to ANTLR v4. There are several interesting features and improvements planned, which could ease the task of writing ANTLR grammars.

Static Analysis If we are able to create an AST for C++0x code, either a CDT-AST or an ANTLR-AST, this structure can be used for further static analysis. Especially, if we have further information available about the existing symbols, many further checks and code analysis could be implemented.

Compatibility While adapting C3P0 to be able to recognize the hello world example application, we had to deal with non-standard-compliant code parts, like the use of `__attribute__`. There is a variety of different toolchain-specific syntax, which must be dealt with, when analyzing source code. Creating extensions to have a configurable set of additional features would be very desirable in productive environments.

A. Side-effects

Developing a parser like C3P0 makes one look at documentations in a different way compared to other readers. Probably, some parts are read more thoroughly than others would. While working with the specifications of the C++ standard [Dra10a] and the boost preprocessor metaprogramming library [KM10], we came across certain mistakes and inconstancies. We gave feedback about these issues, to get those things improved.

A.1. Boost Metaprogramming Library

For implementing macro expansion, there was the suggestion to take the boost metaprogramming library as reference to test our preprocessor. We worked through the samples of the documentation and derived test cases from them. While implementing those tests, we came across three mistakes in the documentation. The flaws were minor, nevertheless, worth to be reported:

1. Documentation of BOOST_PP_LIMIT_ITERATION_DIM:

   The documentation states that this macro expands to 5, but it actually expands to 3.

2. Documentation of BOOST_PP_IIF:

   In the sample code, a closing angle bracket is missing in the second include directive.

3. Documentation of BOOST_PP_ENUM_PARAMS_WITH_A_DEFAULT:

   In the sample code, "class" is missing before T0, T1 and T2 in the comment describing the expected result.

We have created a ticket for these issues and our suggestions have been adopted [Cor10b].

A.2. Standard Examples

Creating a preprocessor required intensive studies of the corresponding section in the C++ standard draft [Dra10a]([CPP]). To verify the explanations have been understood correctly, we have analyzed the examples carefully. Thereby, we came across the example for the concatenation operator. It contained curly braces in the unprocessed code where we had not expected them:

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,{,}7), t(8,9,),
t(10,,), t(,11,), t(,,12), t(,,) };
```

Listing A.1: Example of Paragraph 7 in section [cpp.scope]

Trying to process this example using several compilers failed. We expected there could be some change to the standard for C++0x that we had not been aware of, which would allow the highlighted curly braces – we could not find any. Subsequently, we tracked down the origin of this example to figure out whether there was a change in the standard that explained it. The very first time this example occurred, it did not contain the curly braces in question. Going forward in the standard drafts revealed that there had been a version of this example containing curly braces around every comma. In later versions these had been removed again, except for the pair in our example. Thus, we expected this to be a residue that got lost during cleanup of the document. Further inquiry on the *c++std core* mailing-list confirmed this assumption. It will be corrected in the next version of the C++ standard draft.
It is funny, but we have realized how it probably came to the issue of having curly braces around a specific character when writing this very section. Below we have the first line for the listing showing the example code – the curly braces around the comma probably had been required for the kind of listing used:

```
\begin{lstlisting}[caption=Example of Paragraph 7 in section [cpp.scope{]}]
```

Listing A.2: LaTeXCode of the Listing Containing the Example

Additionally, we have suggested to adapt the result of the examples in paragraphs 5 and 9 as well:

- There should be a space between ~ and 5.

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };

--- to ---

f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

Listing A.3: Example Paragraph 5

- There should be no space after `"Flag"`.
- There should be no space after `x`, in the second line.
- There should be no spaces around `"The first, second, and third items."`

```
fprintf(stderr, "Flag" );
fprintf(stderr, "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));

--- to ---

fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

Listing A.4: Example Paragraph 9

A.3. Inconsistency in Changes to Attributes

In March, the suggestions for the core issue 951 have been published in N3067 [Van10a]. In this document several changes to the positions of attributes have been decided. While weaving these changes into the C3P0 grammar, we came across an inconsistency, caused by this change. In 8.3.5 [dcl.fct] paragraphs 1 and 2, the optional `attribute-specifier` had been moved after the optional `exception-specification`. The listing below shows the new construction syntax for a function declarator:

```
D1 ( parameter-declaration-clause ) cv-qualifier-seq?
ref-qualifier? exception-specification? attribute-specifier?
```

Listing A.5: Changed [dcl.fct] Definition

While this change by itself is okay, it creates an inconsistency regarding lambda declarators:

```
lambda-declarator :
  ( parameter-declaration-clause ) attribute-specifier? mutable?
  exception-specification? trailing-return-type?
```

Listing A.6: Unchanged [expr.prim.lambda] Definition

In our opinion, the declarators of functions and lambda expressions shall be consistent in their definition regarding the position of the optional `attribute-specifier`:

```
lambda-declarator :
  ( parameter-declaration-clause ) mutable?
  exception-specification? attribute-specifier? trailing-return-type?
```

Listing A.7: Suggestion for [expr.prim.lambda] Definition

This inconsistency was not intended and the change to the lambdas was forgotten. This has already been confirmed by William M. Miller and an issue will be opened for it.

A.4. Mandatory Constant Expression

In the latest draft of the C++ standard [Dra10a]([dcl.name]), we have found an editorial error in the grammar for `noptr-abstract-declarator`. According to the definition there, a constant expression is always mandatory in square brackets for array declarations:

```
noptr-abstract-declarator :
  noptr-abstract-declarator? parameters-and-qualifiers
  noptr-abstract-declarator? [ constant-expression ] attribute-specifier?
  ( ptr-abstract-declarator )
```

Listing A.8: Rule for `noptr-abstract-declarator`

If that definition was correct, it would not be possible to have specific declarations, which are valid. Consider the following example:

```
void foo(int[]);
```

Listing A.9: Abstract Array Parameter Example

As the abstract parameter `int[]` should be matched by `noptr-abstract-declarator`. A parser implementing the rule as denoted above, could not recognize that declaration with empty square brackets. Therefore, the `constant-expression` in the rule must be optional.
This editorial error is reported and will be fixed.

A.5. ASTVisualizer

During the third milestone, constructing the CDT-AST, we had to create and compare lots of AST nodes and their hierarchy. We took the AST created by the CDT parser as reference to see whether C3P0 created correct, or at least the same, results. Initially there were two possibilities to look at a CDT-AST:

Debugger One possibility, which is always available when working with Eclipse, is to take the debugger and look at the nodes in the variable view. But this is hardly favorable, as navigating to the point of interest is cumbersome. Furthermore, there is much information we are not interested in, as the debugger displays every possible field. We only used this way to analyze a CDT-AST if we needed that specific information, which is not available in other tools.

DOM AST Viewer A more appropriate way to get information about a CDT-AST is the DOM AST Viewer, a plug-in for the CDT IDE. It can display a CDT-AST reduced to specific information like the node types and the textual representation of certain elements. The structure of the representation is an expandable tree view, similar to the variable view in the debugger. Figure A.1 shows the DOM AST Viewer.

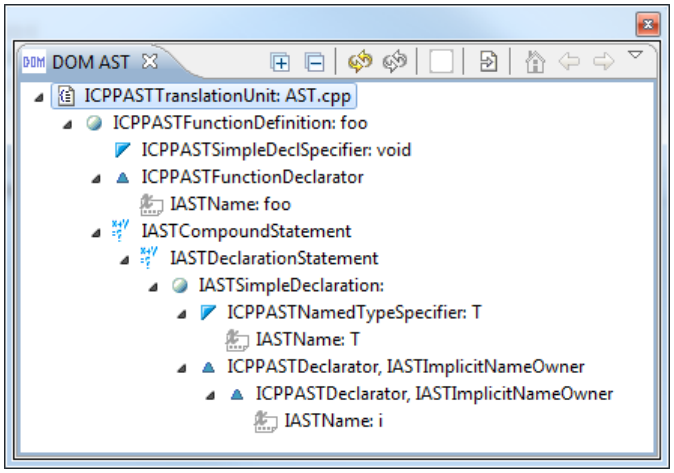

Figure A.1.: DOM AST Viewer

Neither view was appropriate nor comfortable for ongoing checks. Both included clicking though the nodes for analyzing their children, without having the big picture. There is no tool to see a CDT-AST in similar way like ANTLRWorks displays its ASTs. Thus we decided to create one on our own. We did not want to invest too much effort in developing a complete tool from scratch, thus we have adapted an example Java applet of an existing graph library, `JGraphT` [Nav10].

To have an existing example display a CDT-AST we have implemented our own vertex and edge classes as well as an AST visitor for creating the corresponding elements of the graph. In Figure A.2 the resulting graph can be seen.

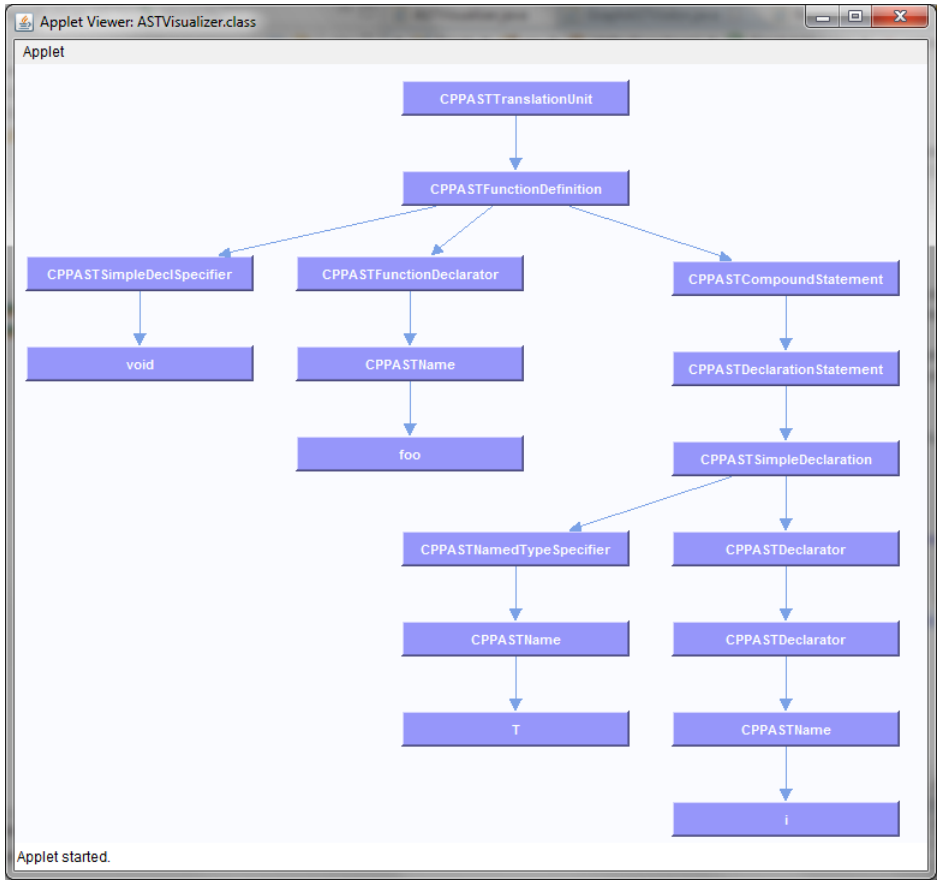

Figure A.2.: ASTVisualizer Example

B. Build Server

B.1. Memory Issues

Having all boost headers as test cases for the `C2P0Parser` creates large output for displaying the results. Hudson could handle the initial 7'000 cases, but we encountered a problem when reaching over 7'250 cases. The ANT script containing the whole build finished successfully, but creating the test results failed with an `OutOfMemoryError`:

```
BUILD SUCCESSFUL
Total time: 1 minute 32 seconds
ln failed: -1
ln failed: -1
Recording test results
FATAL: Java heap space
java.lang.OutOfMemoryError: Java heap space
```

Listing B.1: C2P0 Console Output

Increasing heap space memory for the build itself did obviously not fix this problem. It was necessary to increase the heap memory space available for the JVM running Hudson. This is only possible in the Hudson configuration file: `/etc/default/hudson`

```
# arguments to pass to java
JAVA_ARGS="-Xmx128m"
```

Listing B.2: Hudson Memory Configuration

C. Project Management

This chapter summarizes the progress on C3P0 during the master thesis. First, we have the project plan roughly depicting the semester plan. Second, there is an overview of the time spent on the thesis broken up into weeks.

C.1. Project Plan

Below we have the estimated plan for this master thesis and a chart showing which part we have been working on, split up into weeks.

C.1.1. Target

Although we knew that it will be hardly possible to plan 20 weeks of parser development exactly, we have tried to roughly estimate the time planning for the semester.

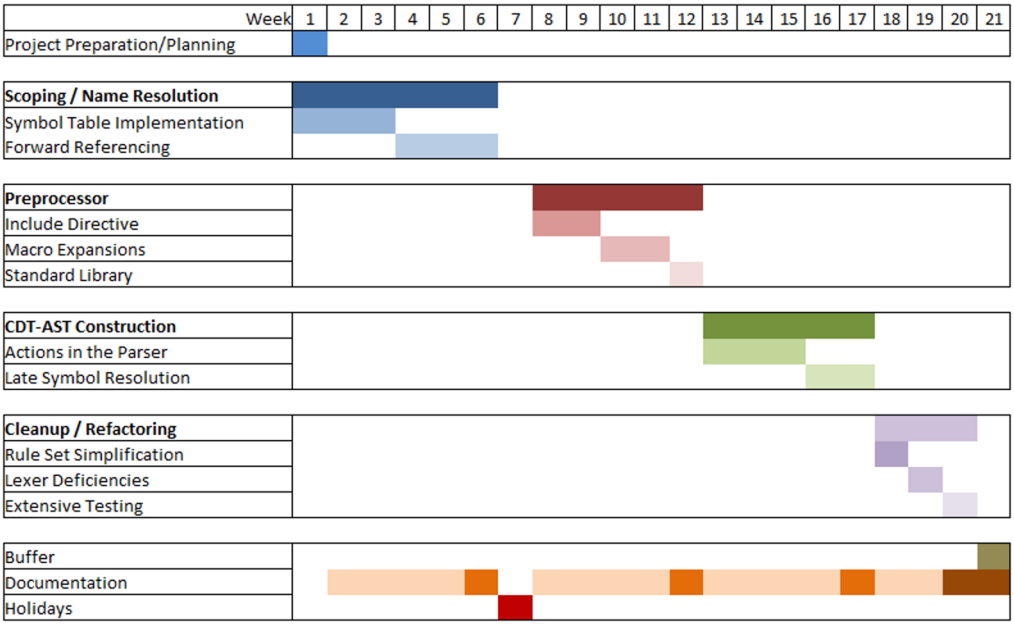

Figure C.1.: Target Plan at the Beginning of the Master Thesis

C.1.2. Actual

Below we see the actual time spent to the tasks of the master theses, divided into weeks.

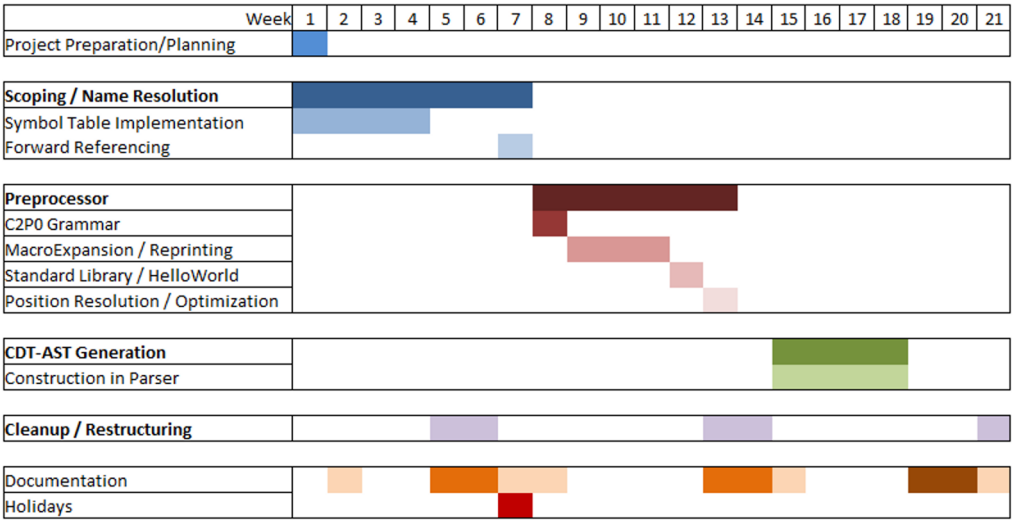

Figure C.2.: Actual Work on the Tasks

Compared to the initial planning, we see that it took slightly longer to work on our first two milestones than expected. In the first milestone, we needed a bit more time to implement our current symbol table, but had been able to solve the problem regarding forward referencing particularly fast. For the second milestone, we have expected to spend more time on the include directives, which eventually did not even require one week, as a small part of *Reprinting*. Macro expansion on the other hand took longer than expected. Actually, we had planned to invest more time on documentation during a milestone, to have fewer work at the end. Eventually, we ended up documenting most at the end of a milestone though.

Week	Hours	Week	Hours
1	50	12	39
2	46	13	44.5
3	50	14	36.5
4	46	15	44
5	45	16	44
6	37	17	38
7	36	18	40
8	46	19	39
9	41	20	48
10	44	21	45
11	45		
Total: 904			

Table C.1.: Time Sheet - Week Summary

C.2. Time Sheet

This section summarizes the time spent on the master thesis during the semester. Actually a master thesis span across 20 weeks. Due to easter holiday we effectively have 21 week from the beginning to the end. During this time, a master student is expected to work 810 hours, for earning 27 ECTS credit points. This results in an expected average workload of 38.6 hours per week.
In Figure C.3 the time spent on the master thesis, split up into weeks, is depicted. The red line shows the average work time expected.

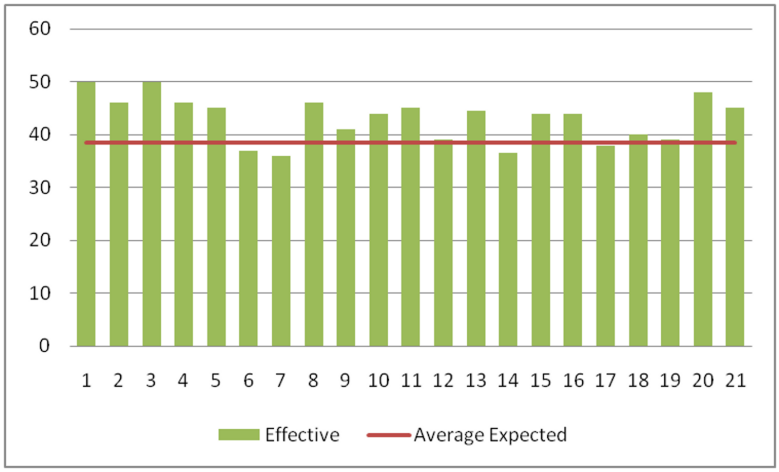

Figure C.3.: Time Spent per Week

Figure C.4 shows the accumulated work time, the red dots, and the accumulated expected work time, in blue dots. The break between week 6 and 8 origins from the easter holidays.

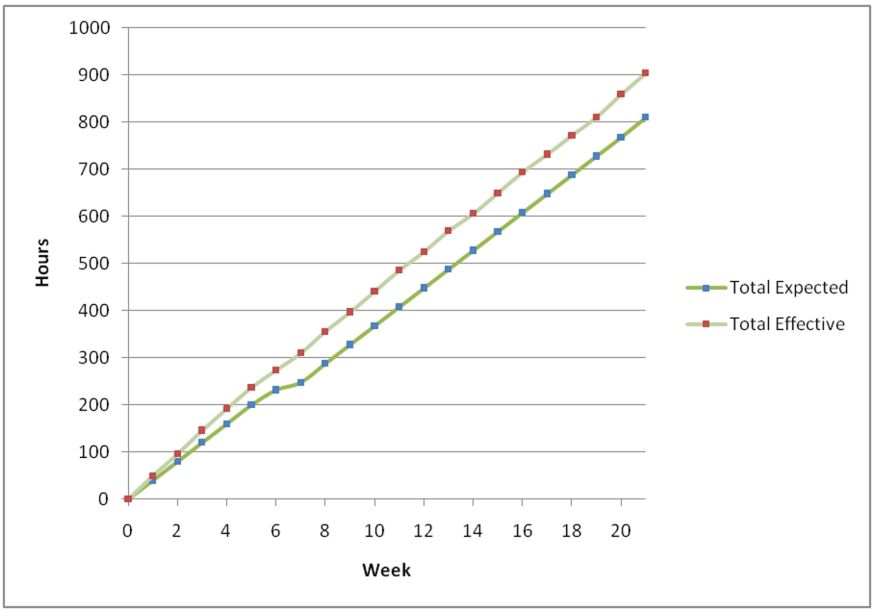

Figure C.4.: Time Spent Accumulated

We are expecting to have approximately 20 additional hours of work designing the poster, creating the presentation and preparing the oral exam.

C.3. Content from Term Project

To have a more consistent documentation for the master thesis we have included some sections of the term project documentation. They shall not be part of the assessment. We have listed the corresponding sections below:

- Chapter 1 (*Introduction*) - Copied from term project documentation, except Section 1.4 (*About This Document*). Some refinements in the other sections of this chapter have been made too.
- Chapter D

C.4. Personal Impression

About one year ago, when we had to decide on the topic of the upcoming term project, and the following master thesis, creating a parser for the new C++ standard was an outstanding option in every aspect. The assignment to implement a new parser from scratch cannot be easy at all. On one side, that scared me a little bit, on the other side, the challenge to do so was very tempting. Even though I was excited about the start of the new semester, to begin with the project, I had been extremely nervous. This situation lasted into the semester for weeks. Usually the nights were short, as I often was lying in bed, pouring over a complex problem of the grammar.
Unfortunately, the term project had been interrupted by four weeks of military service – four exhaustive weeks. Subsequently, my semester had to be prolonged by those four weeks into February. The results of the term project had been considered a success and we decided to continue C3P0 in the next semester, as my master thesis.
After a short holiday, I continued my work, engaging more difficult parts of the parser. Working on the project as a master thesis is extremely intensive. During the term project, I had spent about three, or slightly more, days on C3P0, still working part-time the other two days. This provided some balance. Then sitting five, or sometimes more, days on tasks known to be assessed at the end, is extremely demanding. It had been similar to the diploma thesis, working full time for eight weeks. With the particular difference, that a master thesis must be done alone and it lasted 21 weeks.
Now, looking back at that time, I know that the effort was worth it.

D. Nomenclature

ANTLR A tool for creating language recognizers, developed by Terence Parr [Par09a]

AST Abstract Syntax Tree – An abstract representation of a program or source code, usually focusing on domain specific information.

C3P0 C-Plus-Plus-Parser for C++0x – A parser implementation, generated by ANTLR, for the new C++ standard.

C++0x The name of the new C++ standard, originating from the planned release in 2009 (C++09). Currently the release is expected to become C++0A or C++0B.

CDDL Common Development and Distribution License – A license for open-source software, created by Sun Microsystems [Mic04].

CDT Eclipse C++ Development Tooling – A C++-IDE plug-in for the Eclipse platform [Sch].

EBNF Extended Backus-Naur Form – A metasyntax for describing formal languages.

EPL Eclipse Public License – A license for open-source software, created by the Eclipse Foundation [Fou04].

GPL GNU General Public License – A license for open-source software, created by the GNU Project [Pro91].

IDE Integrated Development Environment – A tool for supporting programmer of a certain programming language in developing software.

Lexer A program for transforming a sequence of characters into a token-stream.

LL(*) Denotes a family of parsers with the property of parsing the input left-to-right and constructing a left-most derivation. The asterisk stands for an arbitrary lookahead.

Parser A program for determining the grammatical structure of an input token-stream.

Semantic Predicate A guard enabling or disabling alternatives in a parser rule according to certain semantic information.

Syntactic Predicate A guard enabling or disabling alternatives in a parser rule according to syntactic information.

Bibliography

[CFS06] Thomas Corbat, Lukas Felber, and Mirko Stocker. Refactoring support for the eclipse ruby development tools. `http://r2.ifs.hsr.ch/rubyrefactoring.pdf`, 2006.

[Cor10a] Thomas Corbat. C3P0 Term Project Documentation. `http://c3p0.ifs.hsr.ch/C3P0Files/C3P0.pdf`, 2010.

[Cor10b] Thomas Corbat. Ticket 4224. `https://svn.boost.org/trac/boost/ticket/4224`, 2010.

[DNR+10] Beman Dawes, Eric Niebler, Rivera Rene, Daniel James, and Vladimir Prus. Boost c++ libraries, version 1.42.0. `http://www.boost.org`, 2010.

[Dra09] Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n3000.pdf`, November 2009.

[Dra10a] Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n3090.pdf`, March 2010.

[Dra10b] Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n3035.pdf`, February 2010.

[DRS05] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1886.pdf`, 2005.

[Fou04] Eclipse Foundation. Eclipse Public Liecense - v 1.0. `http://www.eclipse.org/legal/epl-v10.html`, 2004.

[GCC10] GCC. Language standards supported by gcc. `http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Standards.html`, 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GZS07] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring support for the c++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 781–782, New York, NY, USA, 2007. ACM.

[KM10] Vesa Karvonen and Paul Mensonides. Boost - preprocessor metaprogramming tools. http://www.boost.org/doc/libs/1_42_0/libs/preprocessor/doc/index.html, 2010.

[KU06] Youngki KU. C Preprocessor. http://www.antlr.org/grammar/list, 2006.

[Mic04] Sun Microsystems. Common Development and Distribution License, version 1.0. http://www.sun.com/cddl/cddl.html, 2004.

[Mic10a] Microsoft. Documentation for conditional inclusion in visual c++ preprocessor. http://msdn.microsoft.com/en-us/library/ew2hz0yd.aspx, 2010.

[Mic10b] Sun Microsystems. C/C++ Grammar by Sun Microsystems. http://hg.netbeans.org/main/file/tip/cnd.modelimpl/src/org/netbeans/modules/cnd/modelimpl/parser, 2010.

[Min10] MingGW. Minimalist gnu for windows - gcc. http://www.mingw.org, 2010.

[Nav10] Barak Naveh. Jgrapht. http://www.jgrapht.org, 2010.

[OMJ09] Jeffrey Overby, Matthew Michelotti, and Ralph Johnson. Toward a Language-Agnostic, Syntactic Representation for Preprocessed Code. http://jeff.over.bz/papers/2009/wrt2009.pdf, 2009.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[Par09a] Terence Parr. ANother Tool for Language Recognition, version 3.2. http://www.antlr.org, 2009.

[Par09b] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2009.

[Par10a] Terence Parr. antlr-interest: Abstract tree parser. http://markmail.org/message/6s3nicw52442mc5u, 2010.

[Par10b] Terence Parr. Antlr v4 plans. http://www.antlr.org/wiki/display/~admin/ANTLR+v4+plans, 2010.

[Par10c] Terence Parr. ANTLR Wiki - Tree Construction. http://www.antlr.org/wiki/display/ANTLR3/Tree+construction, 2010.

[Pro] GNU Project. GNU, the GNU Compiler Collection. http://gcc.gnu.org/.

[Pro91] GNU Project. GNU General Public License, version 2. http://www.gnu.org/licenses/gpl-2.0.html, 1991.

[Pro10] GNU Project. Documentation for conditional inclusion in cpp. http://gcc.gnu.org/onlinedocs/cpp/If.html, 2010.

[Sch] Doug Schaefer. Eclipse CDT. http://www.eclipse.org/cdt.

[Van10a] Daveed Vandevoorde. Core issue 951: Various attribute issues (revision 1). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3067.pdf, 2010.

[Van10b] Daveed Vandevoorde. Core issues 743 and 950: Additional decltype(...) uses (revision 1). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3049.pdf, 2010.

[WG2] C++ Standards Committee. http://www.open-std.org/jtc1/sc22/wg21.