Separate Compilation in SugarC

Separate Kompilierung in SugarC
Bachelor-Thesis von Carina Oberle
April 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

Separate Compilation in SugarC
Separate Kompilierung in SugarC

Vorgelegte Bachelor-Thesis von Carina Oberle

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Manuel Weiel, M.Sc.

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 15. April 2015

(Carina Oberle)

Abstract

C is a general-purpose programming language that has its origins in the early 1970s. Despite its age and low-level nature, it still is one of the most widely used programming languages nowadays. Yet, the language lacks a lot of helpful features that can be found in modern programming languages. We argue that syntactic extensibility is a sensible solution to achieving a language based on C that both remains intuitive to C programmers and, in addition, allows to introduce domain specific abstractions that help to achieve higher productivity. For this purpose we present SugarC, an extensible language based on C, which we implemented as an instantiation of the Sugar* framework. Specifically, we provide an approach to integrate a linking stage into our language's build process, which is required in the context of separate compilation.

Inhaltsangabe

C ist eine Allzweck-Programmiersprache, welche ihren Ursprung in den frühen 1970er Jahren hat. Trotz des Alters und niedrigen Abstraktionslevels ist C heute nach wie vor eine der weitest verbreiteten Programmiersprachen. Allerdings mangelt es der Sprache an einigen nützlichen Features, welche in modernen Sprachen vorhanden sind. Wir argumentieren, dass syntaktische Erweiterbarkeit eine geeignete Lösung ist, um eine C-basierte Sprache zu erhalten welche sowohl intuitiv für C Programmierer ist und zudem ermöglicht, Domänen-spezifische Abstraktionen einzuführen, welche helfen eine höhere Produktivität zu erreichen. Für diesen Zweck präsentieren wir SugarC, eine erweiterbare Sprache basierend auf C, welche wir als Instanziierung des Sugar* Frameworks implementiert haben. Im Speziellen liefern wir einen Ansatz zur Integration einer Linking-Phase in den Build-Prozess unserer Sprache, welche im Rahmen separater Kompilierung benötigt wird.

Contents

1 Introduction

This section describes the motivation behind this thesis, states the contributions that we make and outlines the structure of the thesis.

1.1 Motivation

C [17] is a general-purpose programming language that has its origins in the early 1970s. Originally designed as a systems programming language for Unix, it quickly found its way into everyday programming practice. Despite its age and low-level nature, it still is one of the most widely used programming languages nowadays [1]. Yet, the language lacks a lot of helpful features that can be found in modern programming languages. Missing language constructs in C are commonly emulated using lexical macros provided by the C preprocessor. However, lexical macros must be handled with great care, as they fail to reliably preserve the lexical structure of a program.

In this thesis we present *SugarC*, an extensible language based on C, which allows to define new *syntactic sugar* within the language itself. The term *syntactic sugar* refers to syntax extensions that internally are mapped back onto constructs of the base syntax. Consequently, neither expressiveness nor functionality of the language are extended. Yet, syntactic sugar is essential for augmenting a language with domain abstractions that can lead to a considerable increase in productivity. Unlike the text-based macro approach implemented by the C preprocessor, SugarC ensures that its extensions always preserve the syntactical correctness of a program.

We implemented SugarC as an instantiation of the *Sugar** framework [5], which is a language extensibility framework featuring syntax extensions based on library imports. Besides C, SugarC comprises SDF [21], a language for syntax definition, and the term rewriting language Stratego [23]. Specific to SugarC is that it inherits some characteristics from its base-language C which are not directly supported by the framework. One specialty is the concept of header files as interfaces between different translation units. The Sugar* framework is not designed to allow multiple base file extensions, such as both **.c** and **.h**. We thus examine approaches of circumventing this issue.

A major feature specific to C is *separate compilation*. Every C source file represents a distinct translation unit that is compiled independently of any other modules. Necessary forward declarations of functions or variables defined in related modules are typically provided in header files and included via preprocessor directives. The resulting object files are then linked together to obatin an executable program or a library. This is a notable difference to languages like Java, Scala or Haskell, where we do not have *true* separate compilation as we have it in C, and consequently have no need for a separate linking stage. Yet, the Sugar* framework originates from an extensible version of Java and hence is tailored towards languages that possess a similar build process. Therefore, we examine ways of integrating the missing linking stage into SugarC, without forgoing the concepts and benefits of the framework. The Sugar* framework promotes a modular extension activation mechanism, which is the result of its incremental way of processing source files. To conform to this concept, we also need to recompile and link files only if a dependent module has changed.

[1] http://www.langpop.com

1.2 Contributions

The main contributions of this thesis are:

- We build on prior work to provide a modular grammar specification of C99 in the syntax definition metalanguage SDF2 [2].
- We provide a tree traversal strategy in the term rewriting language Stratego to resolve ambiguities intrinsic to C's grammar [3].
- We implement SugarC as Sugar* language plug-in, exploring the possibilities and boundaries of the Sugar* framework with respect to separate compilation [4].
- We provide a case study to demonstrate the practicability of SugarC [5].

1.3 Structure

This thesis is structured as follows: In Section 2 we give an introduction to the main concepts behind SugarC. As a base for SugarC, in Section 3 we describe the specification of the C grammar in a modular way, such that it can be composed with grammar extensions, and show how we can resolve ambiguities intrinsic to C's grammar using a term rewriting language. In Section 4 we outline how we implemented SugarC as instance of the Sugar* framework and present how we resolved encountered challenges, with special focus on separate compilation. We demonstrate SugarC's practicability by providing a case study in Section 5, and examine related approaches to achieving syntactic extensibility for C in Section 6. Finally, we conclude this thesis by resuming our work and outlining future work that may build upon it in Section 7.

[2] https://github.com/c-oberle/spoofax-c/blob/master/C/lib/C.def
[3] https://github.com/c-oberle/spoofax-c/blob/master/C/trans/disamb.str
[4] https://github.com/c-oberle/lang-c
[5] https://github.com/c-oberle/fsm-casestudy

2 Preliminaries

In this section, we introduce the main concepts which constitute the backbone of SugarC. This includes the meta-languages SDF and Stratego, the Spoofax/IMP tool set, and notably the Sugar* language extensibility framework.

2.1 SDF

The *Syntax Definition Formalism* (SDF) [21] is a meta-language that enables a natural and precise description of a formal language. Notably, it promotes syntax specification in a modular fashion, which distinguishes it from other formalisms such as common Backus-Naur Form (BNF). Besides its modularity, other important features that contribute to SDF's expressiveness are its integration of both lexical and context-free syntax, regular expression shorthands, and declarative disambiguation constructs, as sketched in Figure 2.1. Those features in summary allow for a more natural and concise description of a language's syntax, compared to more restrictive formalisms such as BNF, or Yacc as well. Yacc, for instance, a widely used parser generator, is restricted to grammars that fall within the LALR (*Look-Ahead left-to-right*) subclass and thus fails to handle grammars containing ambiguities [3]. With SDF, we do not have such restrictions. SDF's natural way of specifying syntax, however, gives rise to syntactic ambiguities which require *generalized parsing*. This means we have to examine every possible interpretation of a given input text, and hence obtain a *parse forest* in case of ambiguities, rather than a single parse tree. Nonetheless, SDF is supported by a parser generator, which is another important point in which it sets itself apart from BNF. Specifically, it is supported by *scannerless generalized LR parsing* (SGLR), which we will describe in section 3.2.

```
lexical syntax
[A-Za-z][A-Za-z0-9]*
    -> ID
context-free syntax
ID   -> Id
```

(a) Integration of lexical and context-free syntax.

```
Stm*      -> Prog
Id "(" Args? ")"
             -> Call
{E ","}+     -> Args
(Id | Lit)   -> E
```

(b) Regular expression shorthands.

```
E "+" E -> E {left}
E "*" E -> E {left}
priorities
{left: E "*" E -> E} >
{left: E "+" E -> E}
```

(c) Declarative disambiguation mechanisms.

Figure 2.1.: Features contributing to SDF's expressiveness.

2.2 Stratego

Stratego [23] is a program transformation language that provides programmable rewriting strategies. It is especially well-suited for traversing and transforming tree structures, as it is tailored towards processing programs in the form of syntax trees. Stratego programs operate on terms, which essentially are trees representing program code. Such a term may look like `Add(Int(5), Int(2))`, which represents the abstract syntax for the addition of two integer literals. Stratego allows to define transformations on terms by means of *conditional rewrite rules* that make use of pattern matching and **`where`**/**`with`** clauses to determine whether they can be applied to a given subterm. A basic rewriting rule is of the form `rule-name : term-pattern -> term` and may be applied to a term `t` via `<rule-name> t`. For instance, we may define a rule called *eval* that performs simple constant folding by transforming an additive term with two integer literals as child nodes into an integer node containing the sum of both.

```
eval : Add(Int(l), Int(r)) -> Int(n)
  where
    n := <add> (l, r)
```

Figure 2.2 visualizes the application of this rule to our example term mentioned before.

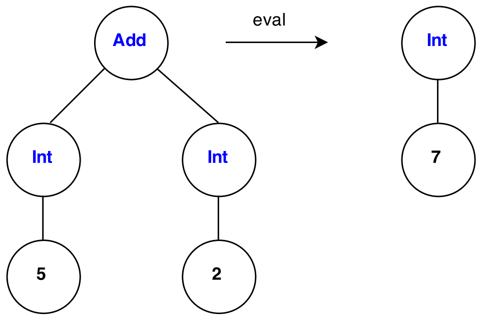
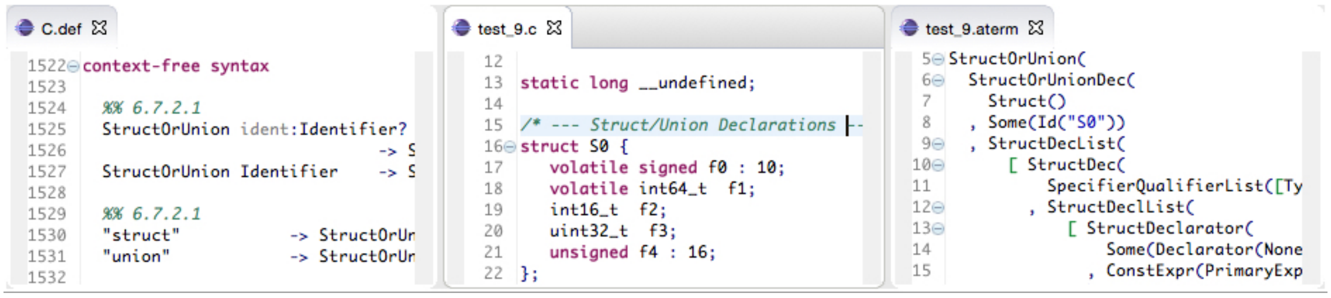


Figure 2.2.: Simple example of a program transformation.

For more complex tasks, we may compose rewriting rules to obtain *rewriting strategies*. Stratego allows to compose rules sequentially as in `s = s1 ; s2` , or via the *deterministic choice* operator `<+`. A useful application of the latter is the strategy `try(s) = s <+ id`, which first tries to apply its argument strategy `s` and, in case this fails, applies the built-in identity function `id` which leaves the original term unchanged.

2.3 Spoofax/IMP

Spoofax/IMP [15] is a language workbench that facilitates the implementation of domain-specific languages (DSLs). The term *language workbench* was coined by Martin Fowler in 2005 [6] and describes a set of programming tools combining language oriented programming with appropriate IDE tooling. A language workbench like Spoofax allows to define a whole new programming language from scratch, without the need to write a parser and with full support for seamless IDE integration. Spoofax is implemented as Eclipse plug-in and makes use of the IMP Meta-tooling Platform to provide customized editor services. It integrates several meta-programming tools, among which SDF and Stratego are of most relevance for us. We use Spoofax mainly to specify the C programming language, which constitutes the base of SugarC, and to generate a corresponding parse table. Figure 2.3 shows a small excerpt of our Spoofax language project for C, presenting from left to right the syntax definition, a successfully parsed test file and the corresponding abstract syntax tree (AST).



```
context-free syntax

  %% 6.7.2.1
  StructOrUnion ident:Identifier?
                                   -> S
  StructOrUnion Identifier         -> S

  %% 6.7.2.1
  "struct"               -> StructOrUn
  "union"                -> StructOrUn

```

```

static long __undefined;

/* --- Struct/Union Declarations --
struct S0 {
   volatile signed f0 : 10;
   volatile int64_t  f1;
   int16_t  f2;
   uint32_t  f3;
   unsigned f4 : 16;
};
```

```
StructOrUnion(
  StructOrUnionDec(
    Struct()
  , Some(Id("S0"))
  , StructDeclList(
      [ StructDec(
          SpecifierQualifierList([Ty
        , StructDeclList(
            [ StructDeclarator(
                Some(Declarator(None
              , ConstExpr(PrimaryExp
```

Figure 2.3.: Implementing the C programming language in Spoofax.

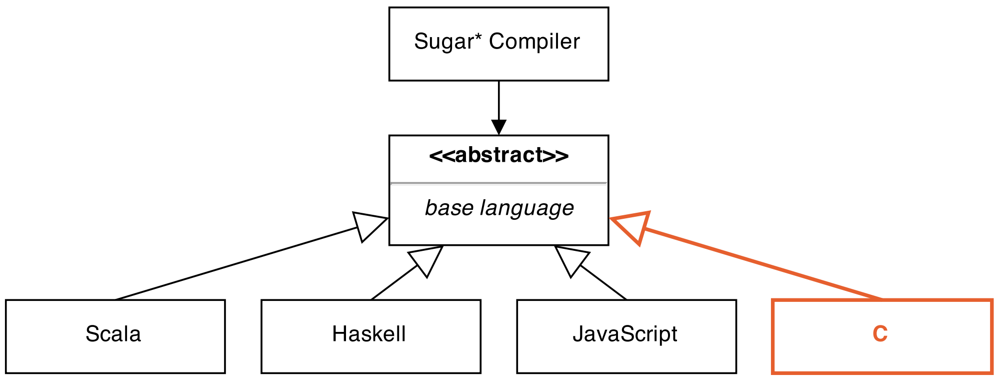


Figure 2.4.: Sugar* language plug-in architecture.

2.4 Sugar*

The Sugar* framework [5] aims at making programming languages extensible. It originated from *SugarJ* [4], an extensible language based on Java, which introduces the concept of *library-based language extensibility*. Library-based extensibility describes the possibility of activating language extensions via library imports. Such *sugar libraries* may contain the definition of new syntactical constructs and their mapping to the base language. In order to define syntactic sugar, the framework integrates the meta-languages SDF and Stratego. Additionally, it builds upon Spoofax to provide optimal IDE support of extensions. Whenever an import of a sugar library is encountered during parsing, Sugar* composes the current grammar with the new syntax and adapts the parser correspondingly. Hence, Sugar* adapts its parser "on the fly", extending the parser itself during the process of parsing. This incremental way of parsing source files refletcs throughout large parts of the framework.
Besides its intuitive appeal and seamless integration into the language, the library-based approach of managing and activating language extensions ensures high modularity, reuse and composability of extensions. The framework is the result of abstracting away from the concrete language Java by introducing an abstract base-language component as additional level of indirection, and eliminating all other dependencies from the SugarJ compiler to the concrete base language. This allows to parameterize the compiler over different base languages by providing language plug-ins, as Figure 2.4 shows. Several language plug-ins like SugarScala [1], SugarHaskell [2] or SugarJS [3] already witness the practicability of the framework. We provide such a plug-in for the C programming language.

1 https://github.com/sugar-lang/lang-scala
2 https://github.com/sugar-lang/lang-haskell
3 https://github.com/sugar-lang/lang-javascript

3 C Grammar in SDF

In order to specify an extensible language, we need to provide both a specification of its base syntax and a specification of an extension syntax, which can be used to introduce new syntactical constructs. Additionally, we need means to compose our syntax specifications to finally achieve a uniform language. This imposes some requirements to the syntax specification:

- The specification should be modular.
- The specification should be uniform (i.e., base and extension syntax should be specified using the same formalism).
- The specification formalism should be supported by a parser generator.

In Section 3.1 we outline the specification of SugarC's base grammar in SDF, a metalanguage that meets all of the listed requirements. In Section 3.2 we show how ambiguities in the grammar specification can be resolved using SDF's built-in disambiguation mechanisms and the term rewriting language Stratego. Finally, we evaluate our specified grammar in Section 3.3.

3.1 Grammar Specification

As a starting point of our extensible version of C, we specify the grammar of the C standard C99 using the *Syntax Definition Formalism* (SDF). More specifically, we use SDF2. SDF is a metalanguage that enables the definition of context-free grammars in a modular way, which distinguishes this approach from specifying syntax in common Backus-Naur Form (BNF). Modularity is a highly important aspect, as we later want to compose this base grammar with grammar extensions to allow the introduction of new syntactical constructs. With mere BNF, which is commonly used to specify the syntax of a programming language, such composability is not given. In addition, unlike BNF, SDF is directly supported by a parser generator.

The complete grammar specification consists of 342 production rules together with 47 rules specifying lexical restrictions, and is subdivided into 57 grammar modules. It can be found online as part of our Spoofax language project for C at `https://github.com/c-oberle/spoofax-c`.

Listing 3.1 exemplarily shows how we can specify the syntax for additive expressions in SDF. Note that, specifically to SDF, the direction of the production rules is inverted compared to common syntax specification in BNF. In this example, the associativity and precedence of the production rules is implicitly given by separating into additive and multiplicative expressions. Specifically, we have left-associative rules, where multiplicative expressions have higher precedence than additive expressions. Likewise, this could be modeled explicitly by using special SDF constructs. In addition to specifying plain production rules, we decorate them with constructor attributes indicated by **`cons`**. This attribute has no effect on the defined syntax, but specifies the name of the node in the abstract syntax tree that is constructed if the given production rule is applied. For instance, a term like $5+7-2$ is implicitly handled as $((5+7)-2)$ due to left-associativity, and would result in a tree similar to the one shown in Figure 3.1.

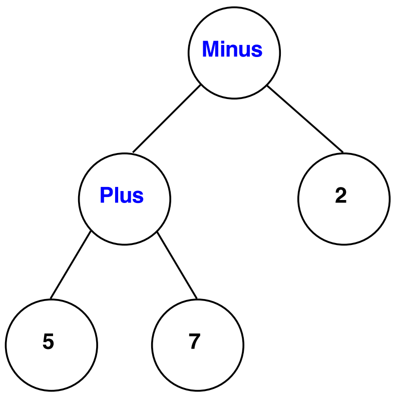


Figure 3.1.: Simplified AST for the term $5 + 7 - 2$.

```
context-free syntax
  MultiplicativeExpr                     -> AdditiveExpr
  AdditiveExpr "+" MultiplicativeExpr -> AdditiveExpr {cons("Plus")}
  AdditiveExpr "-" MultiplicativeExpr -> AdditiveExpr {cons("Minus")}
```

Listing 3.1: Syntax for additive expressions in SDF.

Characteristic to C, the only way of imposing some kind of module structure to a program consists in splitting it into separate files, which then are included by means of preprocessor directives. For this reason, a language that only allows plain C syntax is highly impractical. The macro language implemented by the C preprocessor, however, allows for conditional compilation and thus requires variability-aware parsing [7, 13]. For instance, a code fragment surrounded by the directives `#ifdef FLAG` and `#endif` is only processed in case the symbol `FLAG` has previously been defined. Yet, the enclosed code fragment may affect the parse result. For this reason we decided to augment our C grammar only with a limited subset of existing preprocessor directives. Mainly, we support the syntax of `#include` directives, as they are fundamental for achieving modularity and enable the use of standard libraries. In addition, we support the non-standard but widely supported directive `#pragma once` as alternative to `#include` guards.

3.2 Grammar Disambiguation

With SDF we have a single formalism to specify both the lexical and the phrase level grammar. Correspondingly, SDF uses a scannerless generalized LR parser (SGLR) [20], which means that lexing and parsing are no distinct steps in the process of parsing a source file, but are rather closely intertwined. As a trade-off for SDF's concise and natural way of specifying syntax, we need to handle resulting ambiguities by using a generalized version of an LR parser.

A generalized LR (GLR) parser proceeds similar to a common LR parser. However, given a particular grammar, it processes all possible interpretations of a given input text by applying breadth-first search. The parse table generated by a GLR parser generator thus allows for multiple transitions, given a specific state and input token. In case of conflicting transitions, the parse stack is forked into multiple parallel parse stacks, resulting in so called *parse forests* that consequently have to be transformed into single *parse trees*. In order to do so, we need to discard unwanted trees according to certain rules. The described parse architecture of SDF is depicted in Figure 3.2.

[1] cf. http://www.meta-environment.org/Meta-Environment/SDF

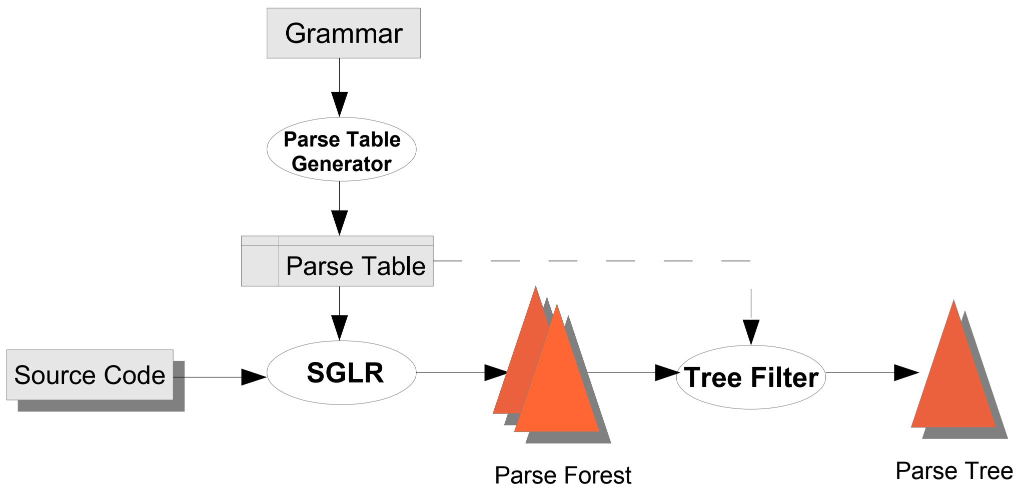


Figure 3.2.: SDF's parse architecture. [1]

3.2.1 SDF Disambiguation Filters

For the most common cases of ambiguities, SDF already provides built-in disambiguation filters. This makes it easy to handle common ambiguous constructs like the *dangling else*. If we consider the following C code fragment, we see that there are two possible ways of parsing it:

```
if (a) if (b) stm1; else stm2;
```

This could be parsed as if it were either one of the following:

```
if (a) {                                  if (a) {
  if (b)                                    if (b)
    stm1;                                     stm1;
  else                                    }
    stm2;                                 else
}                                           stm2;
```

Using simple preference attributes, we can easily eliminate such ambiguities by specifying which derivation we want to choose if there is more than one possibility.

```
"if" "(" Expr ")" Stm                -> SelectionStm {cons("If"), prefer}
"if" "(" Expr ")" st1:Stm "else" st2:Stm  -> SelectionStm {cons("If")}
```

As in C we want to associate an `else` with the nearest `if`, we give preference to the first of the listed production rules by adding a `prefer` tag.

Besides preference attributes, SDF provides some more features to resolve grammar ambiguities. This includes means to reject certain productions, assign associativity to productions, introduce precedence rules and to introduce follow restrictions, i.e., prohibit sequences to be followed by certain symbols. In summary those features enable disambiguation of most of the common critical constructs encountered in practice.

3.2.2 The C Typedef Problem

The possibility of defining type aliases in C using the `typedef` construct induces ambiguities that are more difficult to resolve and cannot simply be handled by SDF's disambiguation filters. The typedef parsing problem, usually referred to as "*typedef-name: identifier*" problem due to the critical production

```
amb(
  [ Dec(
      DecSpecifierSeq([TypeSpecifier(TypedefName(Id("x")))])
    , Some(
        InitDeclaratorList(
          [Declarator(Some(PointerSeq([Pointer(None())])), Id("p"))]
        )
      )
    )
  , ExprStm(
      Some(
        Expr(
          [Mul(PrimaryExpr(Id("x")), PrimaryExpr(Id("p")))]
        )
      )
    )
  ]
)
```

Figure 3.3.: Ambiguity node in the ATerm representation for x * p;.

rule, is a well-known problem that occurrs when parsing C code [2]. It unveils the hidden context sensitivity of C's grammar that has to be handled in order to be able to parse C code correctly. The problematic production rule in our specified SDF grammar is the following:

```
Identifier              -> TypedefName
```

This production rule in a way "collides" with other productions such as:

```
Identifier              -> PrimaryExpr
```

An *Identifier* can thus be derived either as *TypedefName* or as *PrimaryExpr*. As there is lexically no distinction between an identifier that represents a type alias and an identifier that, e.g., represents a simple integer variable, we can encounter ambiguities that can only be resolved by providing contextual information. In the following, we are going to examine this problem by taking a look at concrete examples. Consider the following C statement:

```
int *p;
```

Obviously, this line of code declares a pointer variable *p* pointing to an integer. We know this because `int` is a reserved keyword that indicates a type specifier and thus cannot be the name of a variable. In our grammar this is explicitly modeled by the following rules:

```
"int"              -> TypeSpecifier {cons("Int")}
"int"              -> Identifier {reject}
```

Now we examine the following syntactically correct C code snippet:

```
typedef int x;
x *p;
```

This is the point where we encounter the first ambiguity caused by our critical production rule. Intuitively, it is clear that the second line should semantically be the same as the example declaration given above. However, if we want to parse it we have no knowledge about the context. As we don't know whether *x* has been defined as a type alias or a variable, the statement could either be a multiplication of two variables, where the result is discarded, or otherwise a declaration of a pointer variable pointing to a value of the type represented by *x*. Hence, SGLR outputs a parse forest which is represented in the ATerm (*Annotated Term*) format by a tree-like structure containing ambiguity nodes, as depicted in Figure 3.3.

An ambiguity node in the ATerm format contains a list of all possible ways of parsing a specific part of code. Thus, to resolve ambiguities we first need some context-sensitive rules to determine the correct parsing option for a given ambiguity node. Secondly, we need means to substitute the ambiguity node in our abstract syntax tree by the correct derivation. For this we chose the term rewriting language Stratego, which is directly integrated into the Spoofax language workbench together with SDF and therefore allows seamless interoperability.

Stratego is especially well-suited for traversing and transforming tree structures, as it is tailored towards processing programs in the form of syntax trees. Thus, we can use it to perform transformations directly on the abstract syntax trees produced by the SGLR parser. Basic transformation steps in Stratego can be specified using *conditional rewrite rules*, which make use of *pattern matching* and `where`/`with` clauses to determine whether they can be applied to a given subterm. For example, we can define a simple rule called *Eval* which rewrites an *Add* tree with two integer constants as subtrees to an *Int* node containing the sum of both subtrees [2].

```
Eval: Add(l, r) -> Int(n) where n := <add> (l, r)
```

However, conditional rewrite rules alone are not sufficient to resolve our typedef-related ambiguities, as they do not provide the necessary contextual information. A single rule only has knowledge of the term to which it is applied and possibly of its subterms. What we have to do is traverse the complete syntax tree in a top-down manner and collect all typedef-names we pass on our way down. As typedefs have the same scoping rules as common variable declarations, we also need to consider this. Then, whenever we encounter an ambiguity node, we can either prune branches containing non-existing typedef-names, or otherwise branches containing common identifiers that in fact are type aliases.

To do so, we make use of two more features of Stratego:

- **Rewriting Strategies** allow to define precisely when and how certain rewrite rules shall be applied, which allows us to define a tree traversal strategy that meets our requirements.
- **Dynamic Rules** bring in the context-sensitivity we need in order to resolve typedef-related ambiguities. Dynamic rules in fact are normal rewrite rules, but are generated at run-time and can access information from their generation context. Importantly, the live range of dynamic rules can be limited by rule scopes [22].

Now we can define a disambiguation strategy `disamb`, which we want to apply to the root node of our abstract syntax tree:

```
disamb =
  rules(td-names := [])
  ; traverse(try(disamb-typedef))
```

First, we dynamically create a rule `td-names` that does not consume any input and simply produces an empty list. This rule allows us to maintain a list of all typedef-names in our current scope. Subsequently, we apply a strategy `traverse` that traverses the complete AST and, where possible, applies the rule `disamb-typedef`. We define the traversal strategy as follows:

```
traverse(s) =
  s; all(enter-scope(s) <+ traverse(s))
```

This strategy first applies its argument strategy `s` and then on every child node tries to apply `enter-scope`, or otherwise again `traverse`.

The strategy `enter-scope` makes use of dynamic rule scopes and succeeds only on nodes introducing a new C scope, i.e., function definitions, blocks and **for**-loops with local variable declarations.

[2] Example from https://strategoxt.org/Stratego/StrategoLanguage

```
enter-scope(s) =
    (?FunDef(_, _, _, _) <+ ?Block(_) <+ ?ForDec(_, _, _, _))
    ; {| td-names : traverse(s) |}
```

Whenever we enter a new C scope, we traverse the current sub-AST with the rule scope for `td-names` limited to this traversal. This guarantees that local changes to `td-names` are reverted after leaving the local scope and our list of typedef-names thus stays in sync with the valid type aliases of the current scope.

The strategy `disamb-typedef` resolves ambiguities when applied to an ambiguity node, or otherwise collects new type aliases or variable names and updates `td-names` correspondingly. We need to examine variable names as well, as they can locally shadow type aliases defined in an outer scope. In this case, the name is locally removed from our list of typedef-names. Ambiguities are finally resolved by discarding a branch in case it contains

a) a typedef-name that is not in `td-names`.

b) a variable name that is in `td-names`.

The complete disambiguation module is provided in Appendix A.

3.3 Evaluation

The general problem whether a given grammar is ambiguous or not is undecidable, as it can be shown that it is equivalent to the undecidable *Post correspondence problem* [11]. This means we can detect ambiguities and resolve them, yet we do not have any guarantee that after doing so, our grammar is free of any further ambiguities. Therefore, the best practice of evaluating the correctness of a grammar consists in parsing large program files, preferably of real projects. Unfortunately, parsing code of real projects turns out to be no adequate solution for our C grammar. As we only provide a limited subset of existing preprocessor directives, we may only parse arbitrary C files after preprocessing them. Not only for large software projects written in C, like the version control system Git [3] or the GNU compiler collection (GCC) [4], but also for the most minimalistic program which only prints “Hello World” on the command line, we see that the preprocessed source code still contains constructs which our grammar does not support. Those are mainly optimized inline assembler constructs and may look like the following:
`__asm("_""fopen").`

Inline assembly is an extension to standard C supported by most C compilers, but no actual part of C. To circumvent inline assembly code and preprocessor directives, we use the command line tool Csmith [25] to randomly generate C programs conforming to the C99 standard. Csmith is mainly used for stress-testing compilers and has successfully exposed numerous bugs in C compilers like GCC [5] and LLVM [6]. We generated an initial test set of 10 random C programs that sum up to 13779 lines of code. The result is promising, as all of the test files could be parsed successfully and were free of ambiguities after passing our `typedef` disambiguation filter.

Yet, in order to show that our disambiguation module behaves as desired, it is not sufficient to show that we can parse code unambiguously. In particular, we want to see that the parse result actually is correct. We thus implemented additional unit tests in the *Spoofax Testing Language* (SPT) [14] to cover a variety of different test scenarios. The main purpose of those tests is to examine correctness of our `typedef`

[3] https://github.com/git/git
[4] https://www.gnu.org/software/gcc/releases.html
[5] https://embed.cs.utah.edu/csmith/gcc-bugs.html
[6] https://embed.cs.utah.edu/csmith/llvm-bugs.html

disambiguation module, but we also cover some general language tests. Besides the actual unit tests, a test module in the SPT testing language allows to specify a start symbol for parsing and a test setup. Without loss of generality, we use nonterminal `ExternalDeclarationSeq` as start symbol. Furthermore, we specify a minimal test setup as follows:

```
setup [[
  typedef int a;
]]
```

We implemented pairs of test cases, where each pair contains identical code snippets in different contexts. For instance, to cover an ambiguous code snippet like `a * b;` we first implement one test case where we put the snippet inside the body of a function with no arguments. As in our initial test setup we defined `a` as a type alias, the desired parse result contains a pointer declaration. The SPT language allows us to specify the expected parse result in terms of an AST pattern via **`parse to`** `<pattern>` and matches it with the actual parse result. To check whether the disambiguation strategy chose the right parsing option, it suffices to check for a declaration node (`Dec`) inside the function's body. Irrelevant details are abstracted away by means of the wildcard pattern '_' which matches any term.

```
test Pointer Declaration [[
  void f(void){
    a* b;
  }
]] parse to ExtDecSeq([_,FunDef(_,_,_,Block(Some(BlockItems([Dec(_,_)])))))])
```

In the second test case we put our ambiguous code fragment inside a function with integer arguments, shadowing the **`typedef`** declaration of our test setup. As a consequence, we now expect a multiplicative expression statement (`ExprStm`) instead of a pointer declaration.

```
test Multiplication [[
  void f(int a, int b) {
    a* b;
  }
]] parse to ExtDecSeq([_,FunDef(_,_,_,Block(Some(BlockItems([ExprStm(_)])))))])
```

The test results suggest that our grammar specification, in combination with our **`typedef`** disambiguation strategy, constitutes an adequate base for parsing C code. Still, we are facing the problem that we may only resolve **`typedef`**-related ambiguities in full generality in case we apply it to a program's entire AST. That is, not only to the AST of the main source file, which possibly contains **`#include`** directives, but to the AST that we obtain after preprocessing it. This is a restriction which is unpleasant, albeit quite characteristic to C. We consequently also need to handle this issue in SugarC, which we will discuss further in Section 4.4.

4 SugarC as Instantiation of the Sugar* Framework

We implemented SugarC as an instantiation of the *Sugar** language extensibility framework. The Sugar* framework originated from *SugarJ* [4], which is an extensible language based on Java introducing the concept of *library-based language extensibility*. Library-based extensibility describes the possibility of activating language extensions via library imports. Such *sugar libraries* contain the definition of new syntactical constructs and their mapping to the base language. A core feature of Sugar* is *incremental parsing*, which reflects throughout large parts of the framework. Whenever an import of a sugar library is encountered during parsing, the current grammar is composed with the new syntax and the parser is adapted correspondingly.

In section 4.1 we show how we implemented SugarC as an instantiation of the Sugar* framework. We lay focus on the implementation of header files in SugarC in section 4.2 and present our solution to separate compilation in section 4.3. Finally, we outline possibilities and limitations regarding the integration of our **`typedef`** disambiguation module into our language in section 4.4.

4.1 Implementation

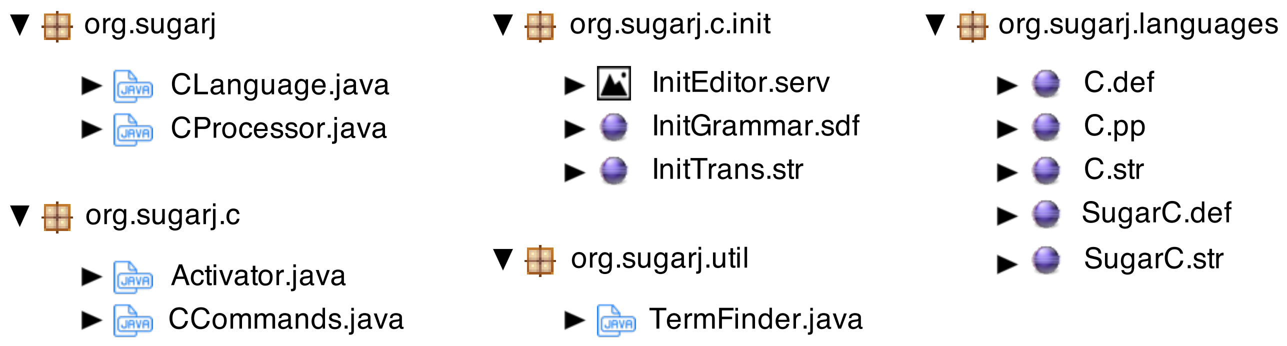
To implement a Sugar* language plug-in, we basically need to provide an implementation of two language-specific interfaces (*IBaseLanguage* and *IBaseProcessor*). In addition, we require the following resources:

- a specification of the base language's grammar (`C.def`)
- a specification of the extension syntax (`SugarC.def`)
- a pretty-printing table for unparsing ASTs (`C.pp`)
- modules specifying initial editor services, the initial grammar and initial transformations

The resulting project structure of SugarC is shown in Figure 4.1. In the previous section we already described how we specified the base grammar for SugarC. In the following sections we illustrate our implementation of SugarC's extension syntax, show how we may unparse transformed syntax trees back into readable code, and sketch our implementation of the interfaces *IBaseLanguage* and *IBaseProcessor*. The remaining initialization modules are implemented straight-forward and will not be discussed in further detail. The modules specifying the initial grammar and initial transformations simply compose the SugarC grammar or parse signature, respectively, with modules of the Sugar* standard library. The initial editor services, on the other hand, mainly specify specific colorings and code foldings.

4.1.1 Extension Syntax

In order to be able to add new syntactic sugar in SugarC, we need to define another grammar module which specifies the syntax of extensions. The Sugar* framework performs *incremental parsing*, which means that it processes only one top-level declaration at a time. This is necessary for activating extensions encountered during parsing, as in such case the parser needs to be adapted correspondingly. Hence, we first need to specify what a top-level declaration is. Top-level declarations are represented by the nonterminal `ToplevelDeclaration`, which Sugar* uses as start symbol for parsing. For SugarC, a top-level declaration is either an external C declaration (`CExternalDeclaration`, cf. Listing 4.1, Line



- ▼ org.sugarj
  - ▶ CLanguage.java
  - ▶ CProcessor.java
- ▼ org.sugarj.c
  - ▶ Activator.java
  - ▶ CCommands.java
- ▼ org.sugarj.c.init
  - ▶ InitEditor.serv
  - ▶ InitGrammar.sdf
  - ▶ InitTrans.str
- ▼ org.sugarj.util
  - ▶ TermFinder.java
- ▼ org.sugarj.languages
  - ▶ C.def
  - ▶ C.pp
  - ▶ C.str
  - ▶ SugarC.def
  - ▶ SugarC.str

Figure 4.1.: SugarC project structure.

10), an extension definition (`CExtension`, Line 11), an extension import (`CExtensionImport`, Line 12) or a declaration of a module dependency (`CDependency`, Line 13). We will explain the latter in section 4.3.

The syntax for external C declarations is already defined in our base grammar (`org/sugarj/languages/C`). What remains to specify is how new syntactic sugar can be declared, and how it can be imported. We specify a SugarC extension to be composed of an extension head and an extension body (cf. Listing 4.1, Line 17). Conforming to most existing Sugar* language plug-ins, we indicate the beginning of a sugar declaration with the keyword **`sugar`**. It is followed by a C identifer, which is the name of the new extension and hence the name of the grammar module that is created therefor. An extension body consists of a possibly empty sequence of extension elements (`ExtensionElem`) framed by curly braces (Line 19). The syntax for such extension elements is specified in the module `org/sugarj/languages/Sugar` and fully comprises SDF and Stratego, which makes it possible to even write extensions employing other imported extensions.

We define the syntax for importing sugar modules analogous to preprocessor includes, but with keyword **`import`** to point out its different semantics (cf. Listing 4.1, Line 23). Unlike common preprocessor includes such as **`#include`** `<stdio.h>`, an import statement of the form **`#import`** `"Sugar"` first imports extensions contained in the SugarC module `Sugar` and then, during pretty-printing, resolves to a **`#include`** directive which includes the desugared file. In addition to sugar imports, we introduce a second language construct indicated by keyword **`#needs`** (Line 27), which allows us to establish dependencies between otherwise unrelated modules. We describe the purpose of this feature in detail in section 4.3.

```
 1  definition
 2
 3  module org/sugarj/languages/SugarC
 4  imports org/sugarj/languages/C
 5          org/sugarj/languages/Sugar
 6
 7  exports
 8    %% top-level declaration
 9    context-free syntax
10      CExternalDeclaration  -> ToplevelDeclaration
11      CExtension            -> ToplevelDeclaration
12      CExtensionImport      -> ToplevelDeclaration
13      CDependency           -> ToplevelDeclaration
14
15    %% sugar declaration
16    context-free syntax
17      CExtensionHead CExtensionBody   -> CExtension {"CExtension", prefer}
18      "sugar" CIdentifier             -> CExtensionHead {"CExtensionHead"}
```

```
    "{" ExtensionElem* "}"          -> CExtensionBody {"CExtensionBody"}

  %% sugar import
  context-free syntax
    "#" "import" "\"" CIncludeFile "\"" -> CExtensionImport {"CExtensionImport"}

  %% (indirect) module dependency
  context-free syntax
    "#" "needs" "\"" CIncludeFile "\"" -> CDependency {"CDependency"}

  lexical restrictions
    "sugar" -/- [a-zA-Z0-9\'\-\_]
```

Listing 4.1: Syntax for SugarC extensions.

4.1.2 Pretty Printing

Every syntactical construct we introduce in SugarC is mapped back to its corresponding C counterpart. This transformation process takes place at the level of abstract syntax trees produced by the SGLR parser. To be able to use existing C compilers to compile the result into an executable file, we need to *unparse* the AST, i.e., transform it back into concrete C syntax. *Pretty printing*, in addition to mere unparsing, attempts to make the code human readable by giving it a structured and clear visual appearance. We make use of the Spoofax language workbench to generate a plain unparsing table for our C grammar, which consists of 218 table entries. For SugarC, we add one more entry by hand, which in addition allows to unparse sugar import statements. This additional rule allows us to resolve an import statement like `#import "Sugar"` to a common `#include` directive that includes the desugared file corresponding to module *Sugar*. The entries of the unparsing table are structured like the ones presented in Listing 4.2, which exemplarily shows how we can unparse subtrees representing an `if` statement.

```
If  --  KW["if"] KW["("] _1 KW[")"] _2
If  --  KW["if"] KW["("] _1 KW[")"] _2 KW["else"] _3
```

Listing 4.2: Table entries for unparsing an If statement.

An **If** node in the abstract syntax tree may have either two or three child nodes, depending on whether an `else` case has been specified. Thus, we have two corresponding table entries. Using the Box markup language [19], we can augment entries of the generated table with appropriate markup to describe the desired layout of program text. We augmented 102 entries with Box markup, resulting in considerable improvement of readability of the resulting program text. The remaining entries did not require additional formatting and were left unchanged. Listing 4.3 shows the table entries from Listing 4.2 augmented with formatting information, which allows us to pretty-print an **If** subtree.

```
If  -- V vs=0 is=2 [H hs=0 [KW["if"] KW["("] _1 KW[")"]] _2]
If  -- V vs=0 [V vs=0 is=2 [H hs=0 [KW["if"] KW["("] _1 KW[")"]] _2] V vs=0 is=2
      [KW["else"] _3]]
```

Listing 4.3: Table entries augmented with Box markup.

The basic idea of Box markup consists in composing sub-boxes. Every child node, as well as every keyword or relevant symbol, represents a sub-box that can be arranged. We use the Box-operators `V` and `H` with additional spacing information to specify the relative ordering of these boxes in a vertical or horizontal way. For our **If** subtree, this yields the ordering depicted in Figure 4.2. The plain unparsing rules generated by Spoofax, on the contrary, would result in single-line programs, where contents are simply strung together in a vast sequence of characters.

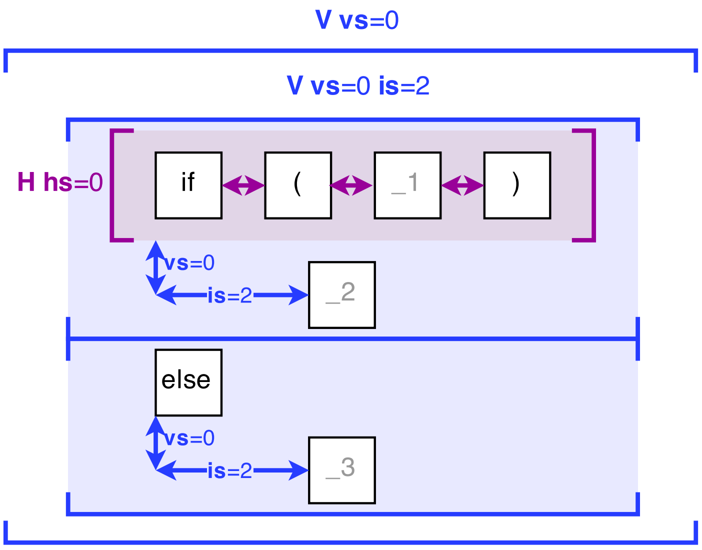


Figure 4.2.: Pretty printing an If statement using Box markup.

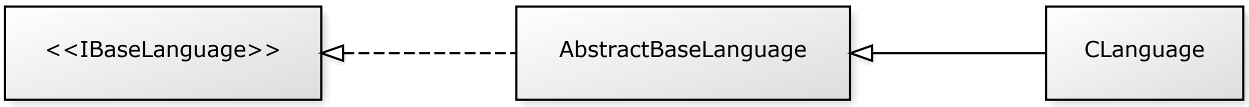


Figure 4.3.: Class hierarchy of CLanguage.

4.1.3 Language Interface

The interface *IBaseLanguage* abstracts away from the concrete base-language. It is stateless and provides methods to reveal general information on the language, such as the name of the language or its base file extension. It is implemented by the abstract class *AbstractBaseLanguage* which contains some additional helper methods. This class is referenced by the interface *IBaseProcessor* and thus needs to be extended in order to create a new Sugar* instance. Our base-language *CLanguage* therefore extends *AbstractBase-Language*, which gives the class hierarchy presented in Figure 4.3. While being independent of the actual processing of source files, the language interface allows to obtain a fresh base-language processor via the factory method `createNewProcessor`. The Sugar* compiler calls this method once for every source file it compiles. Hence, every source file has its own processor. Besides two methods to get the language name and version respectively (`getVersion` and `getLanguageName`), the remaining methods can be separated into three categories, which we will explain in the following.

File extensions. Every Sugar* instantiation has three kinds of file extensions. The method `getSugarFileExtension` returns the extension of source files that may contain syntactic sugar. In the case of SugarC this is `"sugc"`. The extension of files containing desugared, plain base-language code (`getBaseFileExtension`) and the extension of compiled base-language source files (`getBinaryFileExtension`) in our case are `"c"` and `"o"`, respectively. Additionally, in C we have a file extension for header files (`"h"`) which is returned by `getHeaderFileExtension`.

Initialization. For initialization, every base-language implementation needs to provide information on where required resources are located. This includes the location of an SDF module defining the initial grammar (`getInitGrammar`), which typically depends on other packaged SDF modules (`getPackagedGrammars`). Our initial grammar for SugarC simply composes its base grammar (`org/sugarj/languages/SugarC`) with a module of the Sugar* standard library (`org/sugarj/stdlib/`

Common). We furthermore provide a Spoofax editor-service module (getInitEditor) which specifies initial editor services such as code folding, code completion or the coloring of keywords. Finally, we locate a Stratego module specifying the initial transformation (getInitTrans), which includes initial analyses and desugarings. The language user later can extend all of the listed specifications with custom rules.

AST predicates. At last, every base-language must implement language-specific predicates over abstract syntax trees. The Sugar* framework distinguishes three kinds of top-level declarations: extension declarations, import declarations and base-language declarations. Those are language-specific and captured by corresponding predicates which need to be implemented. For any given Stratego term, the predicate isExtensionDecl in *CLanguage* determines whether the term is an application of constructor CExtension. Likewise, isImportDecl decides whether a given term is an application of CExtensionImport. Our dependency construct **#needs**, which we will describe in detail in Section 4.3, technically is neither import nor base declaration. Yet, we have to opt for one of them, as the Sugar* framework currently provides no other means to establish additional dependencies. We chose to handle it as a base declaration, since we do not actually want the framework to import syntactic sugar at this point. A term hence is a base declaration (isBaseDecl) in case it is an external C declaration (ExtDec), a function definition (FunDef), a preprocessor include (Include or StdInclude), or a dependency (CDependency). Optional predicates introduced in *AbstractBaseLanguage* by default return **false** and thus need only be implemented in case the language requires it. For SugarC, none of them are of relevance.

4.1.4 Processor Interface

The interface *IBaseProcessor* provides language-specific methods necessary for the processing of a source file. Like the language interface, it is implemented by an abstract class (*AbstractBaseProcessor*) which contains additional language-independent helper functions and must be extended in order to implement a Sugar* language plug-in. Hence, *CProcessor* extends *AbstractBaseProcessor*, resulting in the class hierarchy presented in Figure 4.4. Unlike the base-language representation, a base-language processor is stateful. For every source file, the Sugar* compiler uses exactly one base-language processor. The compiler initializes this processor (init) by providing the path to the Sugar* source file and the compiler's environment, which contains common information like the source path or the include path.

A base-language processor is responsible for processing base-language declarations (processBaseDecl) and module imports (processModuleImport). Both mainly consist in pretty-printing the abstract declaration term and appending it to a list that maintains all encountered base-declarations or imports, respectively. This way, a base-processor accumulates all desugared source file fragments during compilation of a Sugar* source file. For pretty-printing (prettyPrint), we use commands provided by the Sugar* framework to read our pretty-print table (org/sugarj/languages/C.pp) and use it for unparsing abstract terms. The processing of extension declarations and extension imports is independent of the concrete language and is taken care of by the framework.

When the source file is completely processed, Sugar* calls getGeneratedSource on the processor to assemble the desugared source file from the collected imports and body declarations. The Sugar* compiler then requires the processor to compile the resulting source file (compile), which only contains plain base-language code. For that purpose, *CProcessor* calls the static method gcc provided by utility class *CCommands*, which invokes the C compiler *GCC* [18] with appropriate options.

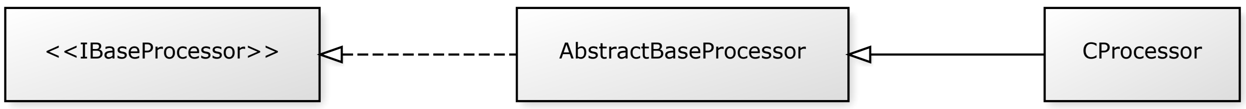


Figure 4.4.: Class hierarchy of CProcessor.

4.2 Header Files in SugarC

The only way of achieving some kind of module structure in C consists in splitting a program into several files. Related program code is bundled in a distinct file to make it easier to maintain and reuse it. To be able to compile each file separately, we require forward declarations of used functions or variables in order to prevent compile errors. Header files are used to provide such forward declarations and thus act as an interface between different translation units. They are distinguished from common C files by their file extension (**.h**) and typically have the same name as the corresponding C file which implements its functions. The concept of header files is quite specific to C-like languages and not directly supported by the Sugar* framework. Sugar* expects a language to have exactly one sugar file extension, as well as to desugar source files into exactly one base file extension. In the following, we examine two possible ways of circumventing this problem.

Header flags. One option consists in introducing the syntax for a header flag (`#header`), which internally marks a SugarC module as header file. This header flag is handled as if it were a base declaration by adapting the predicate `isBaseDecl` in *CLanguage*. However, we treat a header flag specially when processing it in `processBaseDecl`. Unlike actual base-language declarations, we do not want to pretty-print header flags, as they in fact are no base-language constructs. In addition, we set an internal flag in our current instance of *CProcessor*, indicating that the source file we are currently processing contains a header flag. Consequently, we update the name of the target file, such that SugarC modules containing a header flag are desugared into header files instead of C files. This approach, however, does not comply with the typical header file format, in which we can directly recognize header files as such by only taking a look at their file name. Even if we assume that, by convention, a header flag is always on top of a source file, we first need to open it to see whether it is a header or a common source file.

Naming conventions. The second approach, which is more conforming to C's header file format, is to impose naming conventions. As we cannot have two different file extensions like `"sugc"` and `"sugh"`, we declare files with suffix `"_h.sugc"` to be header files, and other files to be common source files. For instance, a file `test.sugc` can have its forward declarations in a file `test_h.sugc`. Like with the header flag approach, we adapt the name of the target file in *CProcessor*, to desugar files with header suffix into header files instead of C files. In principle, this approach allows us to desugar a file `test.sugc` into `test.c`, and equally `test_h.sugc` into `test.h`. Unfortunately, this conflicts with how Sugar* checks whether a module is named correctly. A sugar module, i.e., an extension defined in a SugarC file, must have the same name as the desugared source file. This means if we desugar `test_h.sugc` into `test.h`, an extension defined in `test_h.sugc` must have the name `test`. If we now want to import this module in `test.sugc`, we run into a circular dependency, as the module corresponding to `test.sugc` is named `test` as well. Therefore, we desugar a header file `test_h.sugc` into `test_h.h` to avoid this problem.

In favor of conformity with C's header file format, we chose the second approach to implement header files in SugarC. Yet, SugarC header files are mainly provided for cosmetical reasons and are no compulsory feature of our language. This is because we may as well implement forward declarations in common C files and include them in another file via `#include "file.c"`. If we put aside programming conventions, the only notable difference between header files and common C files is the fact that we usually do not need to compile header files. The Sugar* framework, however, requires us to compile header files as well. For instance, if we process an import of a SugarC header like `#import "test_h"`, the framework expects that there exists a corresponding binary file `test_h.o`. If this is not the case, Sugar* does not trigger further processing of the module import.

As a result, we treat header files basically the same way as we do with common SugarC files. The only difference lies in the file extension of desugared files. Hence, the reason for providing declared header files in SugarC mainly consists in conformity to C and its programming conventions.

4.3 Separate Compilation

In C, programs typically consist of several source files that each contain related program code. C source files may contain preprocessor directives, such as `#include <stdio.h>`, which are resolved before the actual compilation. Such preprocessed C source files are commonly referred to as *translation units*, as they each are compiled independently of any other modules. This is called *separate compilation* and is possible by providing forward declarations of functions or variables defined in other files, in order to inform the compiler about their existence. For instance, if we want to use a function `helloWorld` which is defined in another source file, we need to provide a forward declaration like the following to prevent compile errors.

```
void helloWorld(void);
```

This function prototype informs the compiler about the function's name and type signature, i.e., its arity, parameter types and return type. It may be seen as guarantee to the compiler that somewhere in the program there exists a corresponding definition, which legitimates the use of the function in our code. We may directly place this forward declaration in our current file. It is, however, more convenient to bundle it in a header file together with related code and simply include the header file.

What distinguishes separate compilation in C to how most other languages are compiled, is the fact that C files are processed by merely relying on the interfaces of required modules. In Java, for instance, every source file forms a compilation unit that is compiled on its own, as well. Yet, if we want to compile class C, which depends on classes $C_1, \ldots, C_n$, those dependent classes $C_1, \ldots, C_n$ must at least be available in binary form, as Java has no separate interface files [1]. In addition, in case of circular imports it is necessary to pass all involved source files to the compiler at once. Java files are compiled into class files which are directly loaded and executed by the Java Virtual Machine (JVM), hence omitting a separate linking stage. The Java build process and execution concept is presented in Figure 4.5. We will see that also in SugarC we may not achieve *true* separate compilation as we have it in C. This is by no means a drawback and simply follows from the general design of the Sugar* framework.

4.3.1 C Build Process

The build process, i.e., the process of turning source files into an executable program, in C consists of two major phases: compiling and linking. At first, C source files are separately compiled into *object files* (**.o**). Using GCC, this is done via a command like the following, where the `-c` option requires GCC to only execute the compilation stage and perform no linking.

```
$ gcc -c file_1.c file_2.c .. file_n.c
```

This command results in the generation of files `file_1.o`, `file_2.o`, ..., `file_n.o`. Those object files contain relocatable machine code that is not directly executable. Several object files refer to each other by means of symbols, which can be viewed as placeholders for program code contained in other modules. In order to fill in the gaps and hence to connect the unrelated object files, we execute an additional linking stage. In this stage, the linker combines object files to an executable program or library by resolving

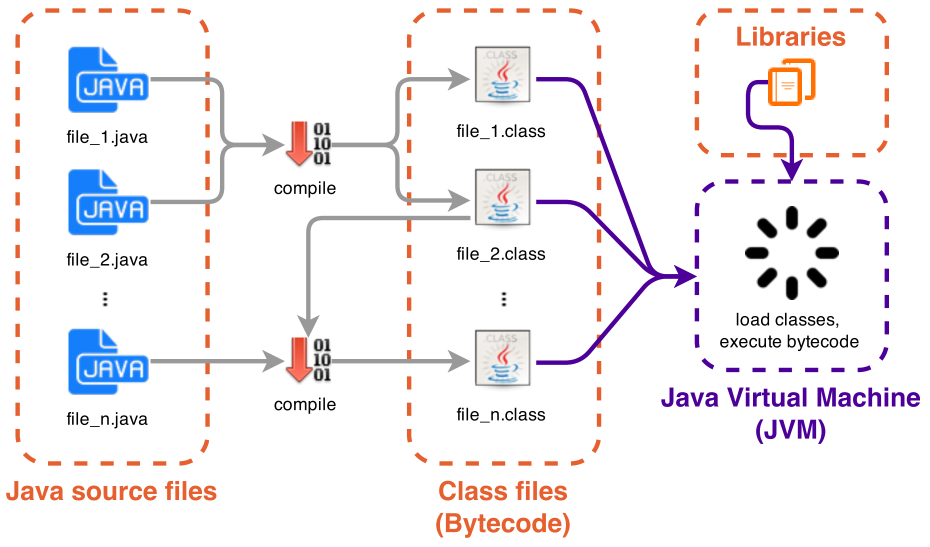


Figure 4.5.: Java classes are directly loaded and executed by the JVM.

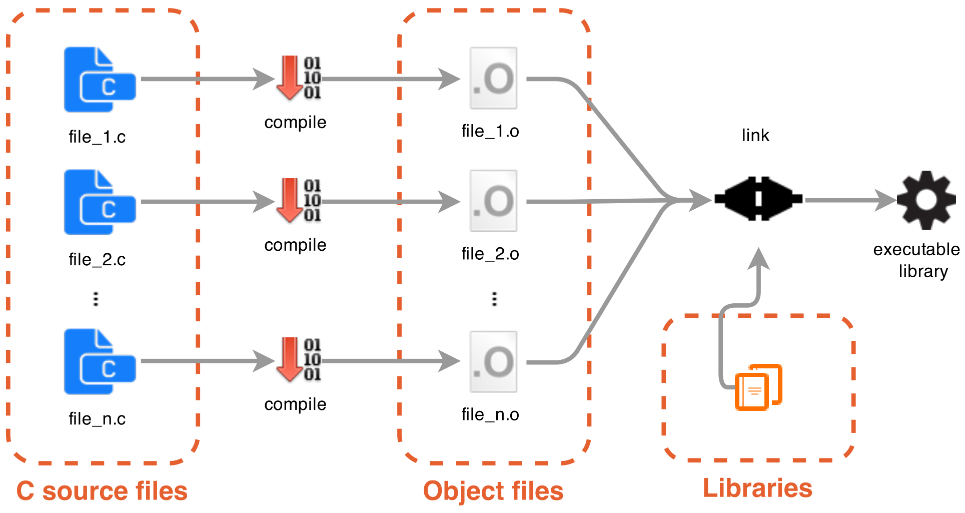


Figure 4.6.: Separate compilation in C requires a separate linking stage.

symbols and *relocating* program code, i.e., assigning run time addresses. This may be done by simply passing the object files to GCC. In addition, if we want to specify the name of the resulting binary, we may use GCC's output flag `-o` and prepend the desired file name.

```
$ gcc file_1.o file_2.o .. file_n.o -o HelloWorld
```

In case one of the listed files contains a `main` function, the given command results in the generation of an executable program `HelloWorld`. An overview of the general build process is given in Figure 4.6.

4.3.2 Linking in SugarC

The separate linking stage is quite specific to C-like languages and constitutes one of the major challenges of our implementation of SugarC. As the Sugar* framework originates from an extensible version of Java, it is tailored to languages like Java, Scala or Haskell that do not require such a linking stage. By this time, the Sugar* framework does not provide support for a separate linking stage. Moreover, performing global linking in some kind of post-processing phase would even contradict the framework's modular way of activating extensions. We thus initiate linking only when processing special modules,

(a) Main.sugc

```
#import "Hello_h"

int main()
{
  helloWorld();
  return 0;
}
```

(b) Hello_h.sugc

```
void helloWorld();
```

(c) Hello.sugc

```
#include <stdio.h>

void helloWorld()
{
  printf("Hello!\n");
}
```

Figure 4.7.: Example program in SugarC.

which we want to refer to as *linker modules*.

We identify a linker module by simply examining whether it contains a `main` function, which represents the entry point of any C program. For this purpose we make use of utility class *TermFinder* (`org/sugarj/util/TermFinder.java`), to check for every base declaration term we process (`processBaseDecl`) whether it contains a function definition with identifier `main`. In such case, we set an internal flag in our current instance of *CProcessor* to mark the current module as linker module. Whenever the framework invokes compilation on a linker module, we execute an additional linking stage. In this stage, we collect all modules on which the linker module depends and link the corresponding binaries. Obviously, to do so we need to ensure that the dependent modules are already available in binary form at the time we want to perform linking. For this we make use of the dependency management system provided by the Sugar* framework.

4.3.3 Dependency Management

Especially when a program gets more complex, it can be time-consuming to recompile the whole project every time we modify a single file. Yet, if we make adjustments in a specific module, we ideally want to recompile every module which depends on it. The Sugar* framework thus takes care of collecting dependencies between modules in order to trigger recompilation only if necessary. The framework identifies dependencies of a given Sugar* module by examining which modules it imports, as well as the resulting transitive imports. However, this approach is tailored towards languages that do not employ separate compilation, i.e., languages that do not make use of separate interface files.

We demonstrate this by the following scenario: An example SugarC program consists of three modules, as Figure 4.7 shows. The module **Main** contains the main function and imports the interface file **Hello_h**, which is implemented by module **Hello**. When Sugar* processes **Main**, it identifies **Hello_h** as only module dependency, as **Main** imports it. Consequently, if necessary, the framework triggers compilation of the imported header module to make sure it is available in binary form [1]. However, it cannot establish a connection to the corresponding implementation file. This is adequate for separate compilation, but prevents us from linking files appropriately, as we cannot be sure whether **Hello** has already been compiled at the time we want to compile and link **Main**. We may only perform linking if we know that every involved module is actually available in binary form. Thus, we need to inform the Sugar* framework about the additional dependencies, in order to make use of its recompilation concept.

For this purpose, we introduce an additional language construct that may be used to declare further module dependencies. The syntax for this construct is similar to the syntax of sugar imports, but headed

[1] As we already mentioned earlier, there is no actual need to compile header files. Still, the framework does not know about the concept of header files and thus treats them like common source files.

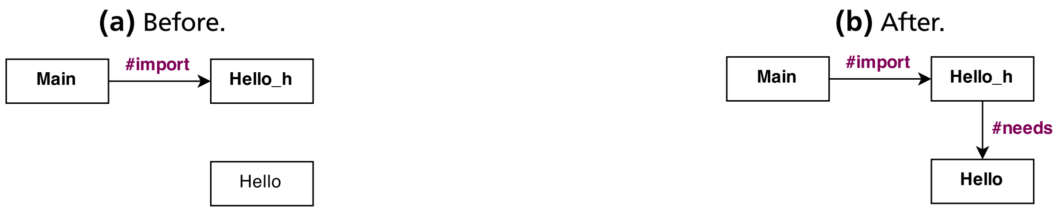


Figure 4.8.: Establishing dependencies between header and implementation files.

by the keyword **#needs**. As a result, we may declare a module dependency via a statement of the form `#needs "module"`. In order to implement the semantics for this construct, the Sugar* framework allows for two different options. The first option consists in providing it with the same semantics as common sugar imports, but leaving out the pretty-printing step that we have for **#import** statements. As desired, the framework consequently registers a dependency when processing such a statement. At the same time, however, it imports extensions from the declared module. This is both an unexpected side-effect and unnecessary processing overhead. For this reason, we chose an alternative option that Sugar* provides to establish module dependencies.

Sugar* permits base-language constructs to establish further module dependencies as well, as for instance Java classes may contain qualified names that reference external classes (e.g. *Java.util.String*). Hence, we handle our **#needs** construct as if it were a base-language declaration by adapting the predicate `isBaseDecl` in *CLanguage* correspondingly. In addition, we adapt the method `processBaseDecl` in *CProcessor*, in order to implement the specific handling for dependency declarations, as shown in Listing 4.4. In case a base-declaration term is a dependency declaration (`CDependency`), we extract the module name and return a list which contains it. The framework consequently registers the additional dependency and takes care of triggering compilation of the corresponding source files, whenever necessary.

```
@Override
public List<String> processBaseDecl(IStrategoTerm toplevelDecl)
    throws IOException {

  if (isApplication(toplevelDecl, "CDependency")) {
    List<String> dependencies = new ArrayList<String>();
    String module = getModulePath(toplevelDecl);
    dependencies.add(module);

    return dependencies;
  }

  /* Otherwise, pretty-print term
   * and return empty list. */
  return Collections.emptyList();
}
```

Listing 4.4: Handling of dependency declarations.

This way, we may ensure that all modules required for linking are already available in binary form after processing the main module. Referring to our example, we may resolve the linking problem by establishing a connection between **Hello_h** and **Hello**, as illustrated in Figure 4.8. For this we simply insert the following line into **Hello_h**.

```
#needs "Hello"
```

Still, to perform linking we need to know exactly which files to pass to the linker. In order to be able to collect those object files that are relevant for linking, we create auxiliary import files for SugarC modules.

To write those files, we call the method `writeDepdendencyFile` provided by helper class *CCommands*, directly before invoking compilation of the desugared source file. The resulting file contains information on both imports and plain dependencies, which may look like the following.

```
Import:Hello_h.h
Dep:Hello.c
```

This information allows to trace back all dependencies of a given module. Beginning with the main module, we recursively follow the links to dependent modules, while collecting direct or transitive dependencies. The concrete implementation of the responsible method `getDependencies` is presented in Listing 4.5. Notably, we do not collect direct or transitive imports, as those files will be textually included by the C preprocessor and consequently would result in duplicate symbols.

```
private static Set<AbsolutePath> getDependencies(Path outFile,
    List<Path> includePaths) {
  HashMap<String, Set<AbsolutePath>> refMap = readDependencyFile(outFile,
      includePaths);
  Set<AbsolutePath> deps = new HashSet<AbsolutePath>();
  /* collect direct dependencies */
  deps.addAll(refMap.get(DEP_PREFIX));

  Set<AbsolutePath> refs = new HashSet<AbsolutePath>();
  refs.addAll(refMap.get(IMPORT_PREFIX));
  refs.addAll(deps);

  /* recursively collect dependencies for all
   * direct dependencies and imports */
  for (AbsolutePath ref : refs) {
    deps.addAll(getDependencies(ref, includePaths));
  }

  return deps;
}
```

Listing 4.5: Collecting dependent modules that are relevant for linking.

Via calling `getDependencies`, we may obtain all relevant base files. The last step consists in replacing the file extensions to obtain the corresponding object files, which can finally be handed over to the linker.

4.4 Integrating the Disambiguation Module

The presence of the C preprocessor and its close ties into the C programming language poses a serious problem when it comes to processing C code. Common tasks like static analyses, automated refactorings, or even general parsing are obstructed by textual includes and compile-time variability. In order to bypass the latter, we only provided our language with a minimal subset of existing preprocessor directives, omitting features like **`#define`** or **`#ifdef`**. Textual includes, however, constitute a major issue regarding the integration of our **`typedef`**-disambiguation strategy, as presented in Section 3.2. To reliably resolve **`typedef`**-related ambiguities like `a * b;` we need to apply our disambiguation filter to the entire program's AST. In return, this means we first need to resolve **`#include`** directives by passing the source files to the C preprocessor. Otherwise, it is not possible for our disambiguator to know whether a given identifier is a type alias or not, as there might be **`typedef`** declarations contained in included header files.

In the context of the Sugar* framework, it constitutes no sensible solution to preprocess files for the purpose of resolving ambiguities. A solution to this problem might be accomplished by either using unsound heuristics, or by making use of some kind of *reversible* preprocessor. The latter might allow to preprocess files in order to resolve ambiguities, and include AST annotations that make it possible to transform it back into unpreprocessed code [9]. Yet, this approach is coupled with possibly vast processing overhead.

Even small programs quickly result in hundreds of lines of code after preprocessing them, in case they include a standard library.

Notwithstanding this problem, we may integrate our disambiguation strategy into our language and use it with certain limitations. This requires to add a disambiguation phase to Sugar*'s processing pipeline, and to make a few adaptions in *CProcessor* that allow Sugar* to locate the disambiguation module. The resulting limitations are the following:

- We may only resolve ambiguities reliably if we restrict the use of type aliases to those declared in the current source file.
- Due to the incremental nature of the Sugar* framework, we may only disambiguate one top-level declaration at a time. Yet, if we always place sugar imports on top of a source file, the remaining part may be disambiguated at once.

5 Case Study: A DSL for Finite-State Machines

In this section, we present a case study to demonstrate the benefits and functionality of SugarC. Particularly, we show how we implemented a small domain-specific language in SugarC, which allows us to implement finite-state machines in a concise, declarative way. The full source code of this case study can be found online at `https://github.com/c-oberle/fsm-casestudy`.

A *domain-specific language* (DSL), as opposed to general-purpose languages like C or Java, is a programming language specialized to a particular application domain. This means, a DSL raises the abstraction level to allow a natural, high-level description of problems related to its domain. As a consequence, a DSL is able to increase productivity, reduce boilerplate code, and even may reduce required programming expertise [16]. Extensible languages like SugarC provide a perfect base for embedding DSLs, as we can directly extend our base language with the domain-specific syntax. In the following, we show how we can implement a small DSL in SugarC, which facilitates the implementation of finite-state machines.

A *finite-state machine* (FSM) is an abstract model of computation which can be used to model simple sequential logic circuits, as well as whole computer programs. It constitutes a quite fundamental concept which may be observed in many areas of everyday life. Simple examples include devices like vending machines, ATMs, elevators or traffic lights. But also more sophisticated concepts like communication protocols, compiler front ends or neurological systems are frequently modeled by means of finite state machines. As the name suggests, a finite-state machine consists of a finite number of states, with declared start and end states. Transitions between two states are commonly formalized as tuples of the form (S, E, S'), which expresses that event E in state S causes a transition to state S'.

As an example, we construct a state machine modeling a simple vending machine. Our vending machine only sells a single product which may be purchased by inserting a coin and pushing a button to confirm. After confirming the purchase, the machine gives out the product and spare coins. It then waits until the product has been removed until it is ready again to process the next purchase. While the product is still in the output area, the machine does not accept inserted coins. The machine is illustrated in Figure 5.1 and consists of three states:

- S_0: Idle, waiting for input.
- S_1: Coin inserted, wait for confirmation.
- S_2: Purchase confirmed, give out product.

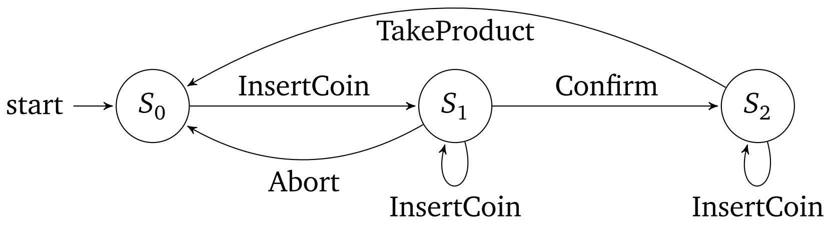


Figure 5.1.: FSM representing a simple vending machine.

If we want to implement our vending machine in C, we may model both states and events as `enum`, as shown in Figure 5.2. In order to implement the actual state machine logic, we may read incoming events

```c
typedef enum {
    Idle,
    CoinInserted
    GiveProduct
} State;
```

(a) States

```c
typedef enum {
    InsertCoin,
    Abort,
    Confirm,
    TakeProduct
} Event;
```

(b) Events

Figure 5.2.: Modelling states and events as enumeration.

in an infinite loop and switch over the current state and event in order to execute the correct transition action. However, even for very basic state machines this quickly results in expansive switch-case blocks, as Listing 5.1 shows.

```c
int main(void)
{
  State state = Idle;

  while(1)
  {
    Event event = readEvent();

    switch(state)
    {
      case Idle:
        switch(event)
        {
          case InsertCoin:
            state = CoinInserted;
            break;
          default:
            break;
        }
        break;

      case CoinInserted:
        switch(event)
        {
          case Abort:
            state = Idle;
            break;
          case Confirm:
            state = GiveProduct;
            break;
          case InsertCoin:
            state = CoinInserted;
          default:
            break;
        }
        break;
      case GiveProduct:
        switch(event)
        {
          case takeProduct:
            state = Idle;
            break;
          case InsertCoin:
            state = GiveProduct;
            break;
          default:
            break;
        }
        break;
    }
  }
}
```

Listing 5.1: Basic state machine logic in C.

This way of specifying the behavior of our state machine is unpleasant, as it involves repetitive code patterns and prevents us from focusing on the actual behavior instead of implementation details. We thus add high-level syntax for state machines, which allows us to specify our vending machine in a natural, precise and less error-prone way, as Listing 5.2 demonstrates.

```
statemachine {
  initial state Init

  input { ... }

  events InsertCoin, Abort, Confirm, TakeProduct

  state Idle {
    InsertCoin => CoinInserted
    Abort => Idle
  }
  state CoinInserted {
    Confirm => GiveProduct
    InsertCoin => CoinInserted
    Abort => Idle
  }
  state GiveProduct {
    TakeProduct => Idle
  }
}
```

Listing 5.2: FSM in SugarC. [1]

Our state machine specification, indicated by the keyword `statemachine`, consists of four compulsory parts:

- A declaration of the initial state (`initial state`).
- A specification of how we get the next event (`input`).
- A declaration of possible events (`events`).
- A declaration of available states and corresponding transitions (`state`).

In addition, we also provide language constructs for specifying data on which the state machine may operate, as well as entry and exit actions for states. For instance, we may add a counter to our state machine which counts the number of sold products. This only requires minor additions to our specification in Listing 5.2. First, we add a data block to specify and initialize our counter.

```
data { int productsSold = 0; }
```

Then we specify an entry action for state `GiveProduct` to increment the counter whenever the vending machine gives out a product.

```
state GiveProduct {
  enter { productsSold++; }
  TakeProduct => Idle
}
```

	Syntax	Desugaring	Editor services
Rules	19	36	6
SLOC	26	235	9

Table 5.1.: Implementation effort for our SugarC state machine extension.

Internally, we create entry and exit functions for every state and surround statements referring to a state transition with corresponding function calls. For example, if we are in state `Idle`, we surround a transition to state `CoinInserted` as follows:

```
exit_Idle();
state = CoinInserted;    // state transition
enter_CoinInserted();
```

This may seem like a trivial task, and in fact can be achieved with relatively low efforts in SugarC. In C, on the contrary, this task is highly error-prone as it is easy to misplace or completely miss function calls, resulting in inconsistencies that are hard to trace back in a larger setting.

We now want to lay focus on how we implemented the transformations that take on the writing of the desired C program. Basically, our main desugaring rule rewrites a state machine construct into a list of top-level declarations. This list contains all necessary declarations, such as the data declarations, entry and exit functions for states, and of course the main function which contains the actual state machine logic. It is important that those declarations are ordered properly, as C requires declaration before use. In order to obtain the desired list of top-level declarations, we apply several minor desugarings to specific parts of the state machine. We exemplarily describe a desugaring that allows us to get the entry function for a given state, as shown in Listing 5.3. This rule transforms a `State` subtree into a **`void`** function, which has a name depending on the state's name and a function body corresponding to the state's entry action. If we apply this transformation rule to state `GiveProduct`, we get the result as presented in Figure 5.3.

```
state-to-entry-fun :
     State(name, StateBody(entry, _, _))
          -> VoidFun(fun-name, block)

     with
        fun-name := <get-entry-fun-id> name;
        block := <state-entry-to-block> entry
```

Listing 5.3: Transforming a state into a corresponding entry function.

(a) Before transformation.

```
state GiveProduct {
  enter { productsSold++; }
  TakeProduct => Idle
}
```

(b) After transformation.

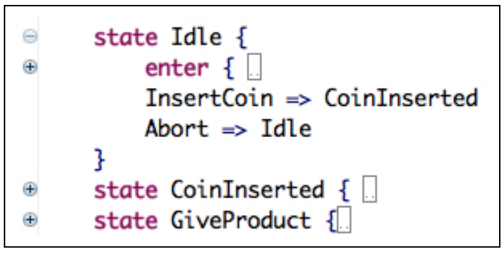
```
void enter_GiveProduct() {
  productsSold++;
}
```

Figure 5.3.: Transforming state `GiveProduct` into a corresponding entry function.

Table 5.1 shows the overall implementation efforts for our state machine extension in SugarC. A side-by-side comparison of a concrete state machine implementation in SugarC and its corresponding C counterpart illustrates the benefits of this extension. Our full vending machine example [2], which simulates

[2] https://github.com/c-oberle/fsm-casestudy/blob/master/test/VendingMachine.sugc

a FSM on the command line, consists of 64 source lines of code (SLOC). The plain C implementation, on the contrary, sums up to 142 SLOC. In addition, SugarC's support for editor extensions allows us to make the implementation of state machines even more convenient. Code folding, as shown in Figure 5.4, allows to hide implementation details, which makes it easier to focus on relevant aspects.



```
state Idle {
    enter { 
    InsertCoin => CoinInserted
    Abort => Idle
}
state CoinInserted { 
state GiveProduct {
```

Figure 5.4.: Code folding for state machines.

To conclude our case study, we briefly want to refer to an existing C framework [3] that is meant to facilitate the implementation of event-driven state machines. This framework promotes the reduction of implementation overhead by enabling the definition of state transitions in terms of look-up tables. Yet, it makes heavy use of unsafe macros which give rise to unpleasant errors. In general, any such framework implemented in plain C falls short of achieving support for domain abstractions the way SugarC does.

[3] http://www.block-net.de/Programmierung/cpp/fsm/fsm.html

[3] Syntax based on https://github.com/seba–/sugarj/blob/master/case-studies/statemachine/

6 Related Work

There exist quite a few approaches that aim at providing C with high-level features, which reflects the wish of programmers for C makeovers like SugarC. Most approaches, however, do not provide language extensibility as integral language feature like SugarC does, but rather build on extensibility of compilers. The most prominent example of an extended version of C probably is Objective-C [12], which is a strict superset of C and extends it with object orientation. In the following, we present some notable approaches to achieve extensibility for C and put them face-to-face with SugarC. Table 6.1 provides a direct comparison of the presented approaches.

Cello

Cello [10] is a common C library that is meant to introduce higher level programming to C. Hence, it allows to include syntactic sugar with the same ease as SugarC, via a simple library import. Among its features are interfaces, exceptions, constructors and destructors for aided memory management, *duck typing*, which allows for generic functions, and some more syntactic sugar to increase readability of code. As all of those features are implemented in plain C, with the help of lexical macros provided by the C preprocessor, they come with several major drawbacks. For instance, to achieve generic functions Cello uses void pointers, which makes functions work for all pointer types. Consequently, we forgo the compiler's type checking, which is an unsatisfactory result. Yet, Cello's macro-based approach to emulate high-level language features constitutes common practice among C programmers. If programmers wish to have a `foreach` construct to loop over data sets, they define a macro that emulates the corresponding functionality, accepting the loss of basic guarantees on syntactical correctness. SugarC, on the contrary, always provides such guarantees, as it operates on abstract syntax trees rather than lexical tokens. In general, Cello's slogan of *"hacking C to its limits"* seems like an adequate description, as it tries to make the most of what C and its preprocessor allow for. Still, it definitely falls short when comparing it to more sophisticated approaches to achieve higher level programming in C.

xtc

A more sophisticated approach to extending C is represented by *xtc* [8] (originally *e**xt**ensible **C***, later *e**xt**ensible **C**ompiler*), which is an extensible compiler framework for C and Java. xtc shares SugarC's concept of transforming extended program code in the form of syntax trees into semantically equivalent C programs. Extensions may be implemented in terms of hygienic macros, which specify grammar modifications, AST transformations or type constraints. xtc makes use of so-called *parsing expression grammars* (PEGs) and employs a *packrat parser generator*. A packrat parser is a recursive descent parser that may perform backtracking and ensures linear-time performance by memoizing intermediate results. Notably, PEGs are closed under composition, which implies modularity and composability of extensions as we have it in SugarC. Unlike context-free grammars specified in SDF, PEGs cannot be ambiguous, since they employ a choice operator that always selects the first match. The `typedef` problem is addressed by adding stateful transaction attributes to production rules that allow to store context information in global hash tables. Still, like SugarC, it falls short as soon as preprocessor includes are involved.

mbeddr

The modular C-variant *mbeddr* [24] is based on the JetBrains Meta Programming System (MPS) and focusses on embedded software engineering. It features language extensions with regard to likewise syntax, type system, semantics and IDE and thus represents one of the richest approaches to make C extensible that we are aware of. mbeddr avoids processing problems related to the C preprocessor by completely omitting preprocessor directives. Instead, mbeddr features special *exports* of symbol declarations which obviates the need for header files. The underlying language workbench MPS uses a

Approach	Extensible Syntax	Extensible static Analyses	Extensible Editor Services	Modular Reasoning	Self-Extensibility	Extensible Debugger
SugarC	●	●	●	●	●	○
mbeddr	●	●	●	●	●	●
xtc	●	○	○	●	○	○
Cello	○	○	○	○	○	○

Table 6.1.: Comparison of the presented approaches (● supported, ○ not supported)

projectional editor that allows to directly operate on an abstract model that is projected to human-friendly views. As a result, non-textual notations like tables or mathematical symbols may increase programming comfort considerably. Like SugarC, mbeddr translates extended source code to standard C and uses GCC for compilation. On top of that, it integrates the debugger GDB to even provide support for extensible debugging.

7 Conclusion and Future Work

We presented the extensible C-based language SugarC as solution to leverage C programming to a higher abstraction level, while retaining the full range of its low-level capabilities. As instance of the Sugar* extensibility framework and with the help of the Spoofax language workbench, the implementation efforts could be kept within reasonable bounds. Table 7.1 shows a comparison of the implementation efforts for realizing SugarC and SugarJ, respectively. Besides the grammar definition, the overhead for SugarC mainly lies in the processing part, for which we employ several helper classes. The main challenges for SugarC were constituted by C-specific peculiarities, most notably by its interface-based program design to enable separate compilation. We compensated Sugar*'s missing support for separate compilation by integrating a separate linking stage into our language. To conform to the incremental nature of the Sugar* framework, we only execute this linking stage when processing a program's main module, which we identify by examining whether it contains a `main` function. In order to be able to link programs properly, we ensure that required modules are available in binary form by establishing additional module dependencies between header and implementation files.

	Base Grammar	Initial Grammar	IBaseLanguage	IBaseProcessor	Helper Classes
SugarC	1359	28	96	187	277
SugarJ	1165	58	177	203	150 [1]

Table 7.1.: SLOC for realizing SugarC in comparison to SugarJ.

We gave an insight into the possibilities of SugarC by providing a language extension for finite state machines. An example SugarC program which simulates a basic vending machine on the command line demonstrates the practicability and benefits of this extension. Likewise, other extensions like **`foreach`** loops or aided memory management may be implemented to increase programming comfort. Yet, SugarC inherits some C-specific properties which pose drawbacks that are hard to resolve. The hidden context-sensitivity of C's grammar gives rise to syntactic ambiguities that may only be resolved by providing context information. Our proposed disambiguation strategy is capable of resolving such ambiguities, yet applies a closed-world assumption. This means we may only expect it to always choose the right parsing option in case we apply it to the *whole* program's AST. In the presence of textual **`#include`** directives, this requires to first pass the program to the C preprocessor, which is no sensible solution in the context of the Sugar* framework. This problem could possibly be resolved by means of some kind of *reversible* preprocessor, as proposed in [9]. Still, we may perfectly use our disambiguator on unpreprocessed source files, as long as we restrict our use of type aliases to those declared in the current file.

For future work, our investigations regarding the support for separately compiled base-languages pave the way for extensibility of related languages like C++ or Objective-C. As Objective-C is a strict superset of C, our modular C grammar even provides a perfect base for an extensible version of it. Furthermore, one could think of integrating support for separately compiled languages directly into the Sugar* framework, in order to simplify and unify their implementation. Specifically, this includes the integration of concepts like our proposed **`#needs`** construct to establish additional module dependencies, with the mere purpose of influencing the recompilation process. But also the support for header and implementation files and their corresponding extensions might be integrated into the framework.

[1] SugarJ's language-specific helper classes are directly integrated into the Sugar* framework.

A Full Typedef Disambiguation

```
module disamb

imports

  libstratego-lib
  libstratego-aterm
  libstratego-sglr

  include/C

strategies

  disamb =
    rules(td-names := [])
     ; traverse(try(disamb-typedef))

  traverse(s) =
    s; all(enter-scope(s) <+ traverse(s))

  disamb-typedef =
    resolve-amb ; try(collect-name)
    <+ collect-name

  collect-name =
    collect-td-name
    <+ collect-id

  enter-scope(s) =
    (?FunDef(_, _, _, _) <+ ?Block(_) <+ ?ForDec(_, _, _, _))
     ; {| td-names : traverse(s) |}

  collect-td-name =
    ?Dec(DecSpecifierSeq([StorageClassSpecifier(Typedef()), _]), Some(inits))
      ; update-td-names(union | <collect-decl-ids> inits)

  collect-id =
    ?Dec(_, Some(inits))
      ; update-td-names(diff | <collect-decl-ids> inits)
    <+ ?ParamDec(_, decl)
      ; update-td-names(diff | [<extract-decl-id> decl])

  update-td-names(s | ids) =
    rules(td-names := <s> (<td-names>, ids))

  collect-decl-ids =
    ?InitDeclaratorList(decls)
    ; <foldl(\(decl, ids) ->
        <union> (ids, [<extract-decl-id> decl]) \)> (decls, [])

  extract-decl-id =
    (?Declarator(_, dd) <+ ?InitDeclarator(Declarator(_, dd), _))
    ; <collect-one(?Id(_))> dd
  contains-invalid-name =
    collect-all(?TypedefName(n); not(<elem> (n, <td-names>)))
       ; not(?[])
    <+ collect-all(not(?TypedefName(_)); one(<elem> (<id>, <td-names>)))
       ; not(?[])


rules

  resolve-amb :
    a@amb([x, y])    ->  res

    with
      amb([x', y']) := <rec x(all(resolve-amb <+ x))> a;
      res := <if(<contains-invalid-name> x', !y', !x')>
```

Bibliography

[1] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of java classes. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, pages 189–200, New York, NY, USA, 2002. ACM.

[2] Eli Bendersky. The context sensitivity of c's grammar. `http://eli.thegreenplace.net/2007/11/24/the-context-sensitivity-of-cs-grammar`, 2007. [Online; retrieved November 16 2014].

[3] Nigel P. Chapman. *LR Parsing: Theory and Practice*. Cambridge University Press, New York, NY, USA, 1987.

[4] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, October 2011.

[5] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. *SIGPLAN Not.*, 49(3):3–12, October 2013.

[6] Martin Fowler. Language workbenches: The killer-app for domain specific languages? `http://www.martinfowler.com/articles/languageWorkbench.html`, 2005. [Online; retrieved March 27 2015].

[7] Paul Gazzillo and Robert Grimm. Superc: Parsing all of c by taming the preprocessor. *SIGPLAN Not.*, 47(6):323–334, June 2012.

[8] Robert Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[9] Quentin Hocquet and Benoit Sigoure. revcpp: A reversible c++ preprocessor. `https://www.lrde.epita.fr/~sigoure/revcpp.pdf`, 2007. [Online; retrieved April 8 2015].

[10] Daniel Holden. Higher level programming in c. `libcello.org`, 2015. [Online; retrieved March 31 2015].

[11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.

[12] Apple Inc. Programming with objective-c. `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/`, 2014. [Online; retrieved March 31 2015].

[13] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[14] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: enabling test-driven language development. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM, 2011.

[15] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[16] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[18] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[19] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, January 1996.

[20] Eelco Visser. Scannerless generalized-lr parsing. Technical report, University of Amsterdam, August 1997.

[21] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[22] Eelco Visser. Scoped dynamic rewrite rules. In *Rule Based Programming (RULE?01), volume 59/4 of Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[23] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 13–26, New York, NY, USA, 1998. ACM.

[24] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.