A

A simple, possibly correct LR parser for C11

Jacques-Henri Jourdan, Inria Paris, MPI-SWS
François Pottier, Inria Paris

The syntax of the C programming language is described in the C11 standard by an ambiguous context-free grammar, accompanied with English prose that describes the concept of "scope" and indicates how certain ambiguous code fragments should be interpreted. Based on these elements, the problem of implementing a compliant C11 parser is not entirely trivial. We review the main sources of difficulty and describe a relatively simple solution to the problem. Our solution employs the well-known technique of combining an LALR(1) parser with a "lexical feedback" mechanism. It draws on folklore knowledge and adds several original aspects, including: a twist on lexical feedback that allows a smooth interaction with lookahead; a simplified and powerful treatment of scopes; and a few amendments in the grammar. Although not formally verified, our parser avoids several pitfalls that other implementations have fallen prey to. We believe that its simplicity, its mostly-declarative nature, and its high similarity with the C11 grammar are strong informal arguments in favor of its correctness. Our parser is accompanied with a small suite of "tricky" C11 programs. We hope that it may serve as a reference or a starting point in the implementation of compilers and analysis tools.

CCS Concepts: •**Software and its engineering** → **Parsers;**

Additional Key Words and Phrases: Compilation; parsing; ambiguity; lexical feedback; C89; C99; C11

ACM Reference Format:
Jacques-Henri Jourdan and François Pottier, 2016. A simple, possibly correct LR parser for C11 *ACM Trans. Program. Lang. Syst.* V, N, Article A (January YYYY), 37 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

This paper explains how to build a compliant C11 parser by combining: an LALR(1) automaton, produced out of a context-free grammar by an LALR(1) parser generator; a lexical analyzer or "lexer", also generated from a declarative description; and a "lexical feedback" mechanism.

Naturally, this is by no means the only approach to parsing C11. Several other approaches are discussed in Section 4. However, this technique seems quite appealing, for at least two reasons. First, it requires comparatively little effort, as a large part of the code is generated. Second, it makes it comparatively easy to convince oneself that the resulting parser complies with the C11 standard. Indeed, our lexer and parser together are less than a thousand (nonblank, noncomment) lines of code, and are expressed in a mostly declarative style.

Instead of immediately delving into the low-level details of LALR(1) automata, lexical feedback, and so on, we give in Section 2 a high-level overview of the various sources of syntactic ambiguity in the C11 standard. As illustrations, we present a collection of "tricky" yet syntactically valid C11 program fragments. (Unless otherwise indicated, every program fragment in the paper is syntactically valid.) This collection, which we make available online [Jourdan and Pottier 2017], forms a small benchmark suite.

This work is supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003.

DOI: http://dx.doi.org/10.1145/0000000.0000000

It can be used to test whether a compiler appears to comply with the C11 standard, especially as concerns the classification of identifiers.

In Section 3, we describe our solution to the problem of parsing C11. It involves a twist on lexical feedback that allows a smooth interaction with lookahead; a simplified and powerful treatment of scopes; and amendments to the C11 grammar. It has been implemented and is available online [Jourdan and Pottier 2017]. We believe that it complies with the C11 standard. We hope that it may serve as a reference or as a starting point in the implementation of compilers, analysis tools, and so on.

In what sense, if any, is our parser "correct"? In a formal sense, this claim cannot easily be stated (let alone proven), as there is no mathematical specification of the C11 language. The informal prose in the standard does not have unequivocal formal meaning; in fact, we point out two ambiguities in it (see Sections 2.2.1 and 2.4). One might wish to come up with a formal specification of the syntax of the C11 language and prove (either on paper or with machine assistance) that our parser complies with this specification. However, a simple, formal specification of the C11 syntax is elusive: the shortest and simplest specification that we know of is precisely our parser itself. As a result, our parser is not verified, and cannot formally be claimed correct. In an experimental sense, we have evidence that it is working: a slightly extended version of it is used in the CompCert C compiler [Leroy 2009; Leroy 2017]. Although it might seem desirable to test it more extensively (both against real-world C11-compliant code bases and against examples of invalid code), we have not undertaken this effort.

2. CHALLENGES IN PARSING C11

To a first approximation, the syntax of the C11 programming language is described by a context-free grammar, which appears in the C11 standard [ISO 2011, A.2]. This grammar is unfortunately ambiguous. The ambiguity is in principle eliminated by a set of rules, expressed in plain English prose, and somewhat scattered through the C11 standard.

In order to implement a correct parser for C11, one must understand why there is ambiguity and how the rules eliminate it, or attempt to eliminate it. As we will see, a few rules are worded in such a way that several interpretations are possible!

In the following, we present the sources of ambiguity, as well as the disambiguation rules and their subtleties, via a series of illustrating examples. It forms a small suite of tricky cases, which can be used as a test suite for a C11 parser.

2.1. Ambiguity in the meaning of identifiers

The most problematic source of ambiguity in the C11 grammar lies in the various roles of identifiers. An identifier may belong to one of four name spaces [ISO 2011, 6.2.3], namely "label names", "tags", "members", and "ordinary identifiers". Furthermore, an ordinary identifier can play one of two roles: it can be viewed either as a "variable" or as a "typedef name" [ISO 2011, 6.2.1, §1]. (We use the word "variable" to mean an "object", a "function", or an "enumeration constant" paragraph 6.2.1, §1.) Unfortunately, the distinction between these two roles requires contextual information. For example, consider the following two valid code fragments:

```
// typedef_star.c
typedef int T;
void f(void) {
  T * b;
}
```

```
// variable_star.c
int T, b;
void f(void) {
  T * b;
}
```

In the left-hand fragment, the function `f` contains a declaration of the local variable `b` as a pointer to an object of type `T` (which has been declared as a synonym for `int`). On the

other hand, in the right-hand fragment, the body of `f` does not contain any declaration: it contains one statement, namely an expression statement, a multiplication of the global variable `T` by the global variable `b`.

Thus, the manner in which one should interpret the body of `f` depends on the context in which this declaration appears. In particular, its interpretation depends on whether the identifier `T` denotes a variable or a typedef name. Some mechanism is therefore needed in order to classify identifiers.

Before discussing how such a mechanism might be described and implemented (which we do in Section 3), let us recall the rules that this classification process must obey. These rules are relatively simple, but exhibit a few subtleties (such as noncontiguous scopes; see Section 2.1.4) and interactions with other language features (such as the "dangling else" ambiguity; see Section 2.2.2) which further complicate the problem.

2.1.1. Name spaces. As noted above, there are four name spaces of identifiers [ISO 2011, 6.2.3]: label names, tags, members, and ordinary identifiers. An identifier may play independent roles in independent name spaces. For instance, the following code is valid:

```
// namespaces.c
typedef int S, T, U;
struct S { int T; };
union U { int x; };
void f(void) {
  // The following uses of S, T, U are correct, and have no
  // effect on the visibility of S, T, U as typedef names.
  struct S s = { .T = 1 };
  T: s.T = 2;
  union U u = { 1 };
  goto T;
  // S, T and U are still typedef names:
  S ss = 1; T tt = 1; U uu = 1;
}
```

2.1.2. Visibility and scopes. A declaration introduces either new variables or new typedef names, depending on whether the **`typedef`** keyword appears among the declaration specifiers. A declaration may contain multiple declarators. A declarator declares exactly one entity, and introduces one identifier [ISO 2011, 6.7.6, §2]. It is "visible" within a certain region of the program text [ISO 2011, 6.2.1, §2]. Where the declarator is visible, the declared entity can be referred to via its identifier.

Visibility begins where a declarator is complete. Visibility stops at the end of the scope that the declaration inhabits. The program text is organized in "scopes", that is, regions that begin and end at well-defined points, such as the beginning and end of a block (see Section 2.1.4). Scopes can overlap: more precisely, they can be nested [ISO 2011, 6.2.1, §4]. A declaration of an identifier in an inner scope hides a declaration of the same identifier in the same scope or in an outer scope. Thus, the region where a declaration is visible is not necessarily contiguous:

```
// local_scope.c
typedef int T;
void f(void) {
  T y = 1; // T is a type
  if(1) {
    int T;
    T = 1; // T is a variable
```

```
  }
  T x = 1; // T is a type again
}
```

Both variable declarations and type declarations may appear in an inner scope. A variable declaration can hide a type declaration, and vice-versa:

```
// local_typedef.c
typedef int T1;      // Declaration of type T1 as int
void f(void) {
  typedef int *T2;   // Declaration of type T2 as pointer to int
  T1 x1;             // Declaration of x1 of type T1
  T2 x2;             // Declaration of x2 of type T2
  x1 = 0;
  x2 = 0;
}
```

A variable or typedef name becomes visible just after the completion of its declarator [ISO 2011, 6.2.1, §7]. In C terminology, a declarator is a fragment of a declaration that concerns a single identifier. The syntax for declaring pointers, functions, and arrays is part of a declarator, whereas declaration specifiers (such as `int` and `const`) and optional initializers are not. Thus, the following code is valid:

```
// declarator_visibility.c
typedef int T, T1(T);   // T is visible when declaring T1.
void f(void) {
  int (*T)(T x) = 0;
    // This declaration declares T as being a pointer to a
    // function taking one parameter, x, of type T, and
    // returning an integer. It initializes T to the null pointer.
    // The declaration is valid, since in the declarator of the
    // parameter x, T is still a typedef name, as the declarator
    // of T has not yet ended.

  int T1 = sizeof((int)T1);
    // In the initializer sizeof((int)T1), the declarator of T1 has
    // ended (it is constituted solely by the identifier T1), so T1
    // denotes a variable.
}
```

2.1.3. Enumeration specifiers modify the current scope. A somewhat counter-intuitive feature of enumeration specifiers is that they declare enumeration constants [ISO 2011, 6.7.2.2, §3], which are ordinary identifiers [ISO 2011, 6.2.3, §1]. This phenomenon takes place regardless of the context in which the enumeration specifier occurs. The newly declared enumeration constants remain visible (except where temporarily hidden) until the end of the current scope, whatever that scope might be.

For instance, mentioning an enumeration specifier as an argument to the `sizeof` operator, or as part of a cast expression, declares new enumeration constants. Because an expression does not form a scope, these constants remain visible past the end of the expression. In the following example, the typedef name `T` is hidden by a local declaration of `T` as an enumeration constant:

```
// enum_shadows_typedef.c
typedef int T;
void f(void) {
  int x = (int)(enum {T})1;
  // T now denotes an enumeration constant,
```

```
  // and behaves syntactically like a variable:
  x = (int)T;
}
```

An enumeration constant becomes visible in the current scope just after the completion of its enumerator [ISO 2011, 6.2.1, §7]. Thus, the newly declared constant can be referred to in the declaration of further enumeration constants and in the remainder of the current scope:

```
// enum_constant_visibility.c
typedef int T;
void f(void) {
  int x;
  x = (enum {T, U = T+1})1 + T;
  int y = U - T;
}
```

2.1.4. Where scopes begin and end. According to the C11 standard [ISO 2011, 6.2.1, §2], there are three kinds of scopes: file, block, and function prototype. (The fourth kind, function scope, does not concern ordinary identifiers. In our case, it can be ignored.) For our purposes, though, it is not necessary to keep track of the various kinds of scopes; it suffices to know where they are opened and closed.

A "file" scope is opened at the beginning of a translation unit, and closed at the end.

The C11 standard [ISO 2011, 6.8.2, §2] states that "a compound statement is a block", which means that a new "block" scope must be created at the beginning of a compound statement, and closed at the end:

```
// block_scope.c
typedef int T;
int x;
void f(void) {
  { T T;
    T = 1;
    typedef int x;
  }
  x = 1; // x as a type is no longer visible
  T u;   // T as a variable is no longer visible
}
```

A selection statement forms a block. Furthermore, each of its immediate substatements also forms a block [ISO 2011, 6.8.4, §3]:

```
// if_scopes.c
typedef int T, U;
int x;
void f(void) {
  if(sizeof(enum {T}))
    // The declaration of T as an enumeration constant is
    // visible in both branches:
    x = sizeof(enum {U}) + T;
  else {
    // Here, the declaration of U as an enumeration constant
    // is no longer visible, but that of T still is.
    U u = (int)T;
  }
  switch(sizeof(enum {U})) x = U;
  // Here, T and U are typedef names again:
```

```
  T t; U u;
}
```

The same rule applies to iteration statements [ISO 2011, 6.8.5, §5]:

```
// loop_scopes.c
typedef int T, U;
int x;
void f(void) {
  for(int T = 0; sizeof(enum {U}); ) x = U+T;
  for(sizeof(enum {U}); ; ) x = U + sizeof(enum {T});
  while(sizeof(enum {U})) x = U;
  // A declaration in the body of a do ... while loop
  // is not visible in the loop condition.
  do x = sizeof(enum {U}) + U;
  while((U)1 + sizeof(enum {U}));
  // The above declarations of T and U took place in inner scopes
  // and are no longer visible.
  T u3; U u4;
}
```

On the other hand, expression statements, return statements, and labelled statements do not give rise to new scopes:

```
// no_local_scope.c
typedef int T, U, V;
int x;
int f(void) {
  x = sizeof(enum {T});
  label: x = sizeof(enum {U});
  return sizeof(enum {V});
  // T, U and V now denote enumeration constants:
  x = T + U + V;
}
```

The parameter list of a function prototype gives rise to a new "function prototype" scope, which must be opened at the beginning of the parameter list and closed at the end of the parameter list. Thus, a name introduced somewhere in the parameter list is visible in the remainder of the parameter list and is no longer visible after the parameter list ends. We emphasize that such a name can be either a formal parameter or an enumeration constant that appears in the type of a formal parameter. In the following example, the names introduced by the parameter list of `f` are `T`, `U`, `y`, `x`:

```
// function_parameter_scope.c
typedef long T, U;
enum {V} (*f(T T, enum {U} y, int x[T+U]))(T t);
  // The above declares a function f of type:
  // (long, enum{U}, ptr(int)) -> ptr (long -> enum{V})
T x[(U)V+1]; // T and U again denote types; V remains visible
```

The parameter list of a function definition gives rise to a new "block" scope, which must be opened at the beginning of the parameter list and closed at the end of the function body. Indeed, the C11 standard [ISO 2011, 6.2.1, §4] prescribes: "If the declarator or type specifier that declares the identifier appears [...] within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block." Thus, in the following example, which begins like the previous one, the scope of the names `T`, `U`, `y`, `x`, which are introduced in the parameter list of `f`, includes the function body:

```
// function_parameter_scope_extends.c
typedef long T, U;
enum {V} (*f(T T, enum {U} y, int x[T+U]))(T t) {
  // The last T on the previous line denotes a type!
  // Here, V, T, U, y, x denote variables:
  long l = T+U+V+x[0]+y;
  return 0;
}
```

This example is particularly challenging. It shows that a scope need not be contiguous. Indeed, the block scope created at the beginning of the parameter list `(T T, enum {U} y, int x[T+U])` encompasses this parameter list as well as the function body, but does not include the rest of the declarator of `f`. In particular, the identifier `T` in the parameter list `(T t)` must be interpreted as a typedef name introduced by the initial **`typedef`** declaration.

In an implementation, when reaching the end of a parameter list, one must close the scope that was opened for this parameter list. Yet, one must not discard it entirely, as one may have to re-open it at the beginning of the function body, if this parameter list turns out to be part of a function definition. One cannot "fake" this by opening a new scope at the beginning of the function body and introducing the names of the formal parameters into it, as this new scope would then be missing the enumeration constants that may have been declared in the parameter list.

2.2. Ambiguity in `if` versus `if-else`

An **`if`** statement may or may not have an **`else`** branch. This creates an ambiguity, which is well-known as the "dangling else" problem: an **`else`** branch that follows two nested **`if`** constructs can be attached either to the farthest or to the nearest **`if`** construct. For example, according to the C11 grammar, the following code may give rise to two different syntax trees, depending on which **`if`** statement the **`else`** branch is associated with:

```
// dangling_else.c
int f(void) {
  if(0)
    if(1) return 1;
   else return 0;
  return 1;
}
```

2.2.1. An arguably ambiguous disambiguating sentence. The C11 standard attempts to explicitly eliminate this ambiguity by stating: "An **`else`** is associated with the lexically nearest preceding **`if`** that is allowed by the syntax" [ISO 2011, 6.8.4.1, §3]. Thus, in the previous example, the **`else`** branch must be attached to the second **`if`** construct.

However, this disambiguating sentence is itself somewhat unclear, in a subtle way, and could arguably be viewed as ambiguous or misleading. The following *invalid* code fragment illustrates the problem:

```
// dangling_else_misleading.fail.c
typedef int T;
void f(void) {
  if(1)
    for(int T; ;)
      if(1) {}
      else {
        T x;
      }
}
```

Here, if the `else` branch is attached to the second `if` construct, then the last occurrence of `T` must denote a variable, which implies that the program is syntactically invalid. One might therefore interpret paragraph 6.8.4.1, §3 as dictating that the `else` branch be attached to the first `if` construct. Under this interpretation, the last occurrence of `T` would denote a typedef name, and the program would be syntactically valid.

Such an interpretation seems exotic: we know of no C compiler that behaves in this manner. Therefore, our informal interpretation of paragraph 6.8.4.1, §3 is that "an `else` is associated with the lexically nearest preceding `if` that is allowed by the syntax, considering only what has been read so far". In other words, the text beyond the `else` keyword must not be taken into account when one commits its association with a certain `if` keyword.

Nevertheless, we believe it is interesting to point out that the short informal sentence in paragraph 6.8.4.1, §3 does not have clear formal meaning. We point out another such sentence in Section 2.4.

2.2.2. An interaction between the `if`-`else` ambiguity and the scoping rules. Another subtle issue caused by the fact that `else` branches are optional is its interaction with the scoping rules (which we have recalled in Section 2.1.4). Consider an open-ended `if` statement, that is, one that does not (yet) have an `else` branch, such as "`if(1);`". After such a statement has been read, two cases arise:

— If the token that follows this `if` statement is the keyword `else`, then this `if` statement is in fact not finished. Its scope must extend all the way to the end of the `else` branch.
— If, on the other hand, the token that follows this `if` statement is not `else`, then this `if` statement is finished. Its scope ends immediately before this token.

Although this specification is clear, it may be somewhat tricky to implement correctly in a deterministic parser, such as an LALR(1) parser with lexical feedback. When one reads the first token beyond an open-ended `if` statement, one does not know yet whether the scope of the `if` statement has ended. Thus, if this token happens to be an identifier, and if one attempts too early to classify this identifier as a variable or a typedef name, one runs a risk of classifying it incorrectly. The potential danger appears in the following valid example:

```
// dangling_else_lookahead.c
typedef int T;
void f(void) {
  for(int T; ;)
    if(1);
  // T must be resolved outside of the scope of the
  // "for" statement, hence denotes a typedef name:
  T x;
  x = 0;
}
```

This particular example is incorrectly rejected by GCC 5.3 (and older versions). We have reported this bug to the developers of GCC [Jourdan 2015]. It has been acknowledged and fixed in the development version.

2.3. Ambiguity in declarations and struct declarations

The C11 grammar exhibits an ambiguity in the syntax of declarations. This is illustrated by the following code fragment:

```
// declaration_ambiguity.c
typedef int T;
void f (void) {
```

```
  unsigned int;   // declares zero variables of type "unsigned int"
  const T;        // declares zero variables of type "const T"
  T x;            // T is still visible as a typedef name
  unsigned T;     // declares a variable "T" of type "unsigned"
  T = 1;
}
```

In the C11 grammar [ISO 2011, 6.7, §1], the syntax of declarations is defined by the production *declaration: declaration-specifiers init-declarator-list*$_{opt}$ **;**. With this in mind, there is only one way of reading the third line above, "`unsigned int`". Indeed, `unsigned` and `int` are type specifiers, hence also declaration specifiers; the *init-declarator-list* in this case must be absent. Hence, this declaration declares zero variables of type `unsigned int`. However, there are two ways of reading each of the two declarations "`const T`" and "`unsigned T`". On the one hand, since `T` is a typedef name, it is also a type specifier, so these declarations could be interpreted, like the previous one, as declarations of zero variables of type `const T` and `unsigned T`, respectively. On the other hand, since `T` is an identifier, it forms a declarator, so these declarations can be interpreted as declarations of a variable `T` of type `const` and `unsigned`, respectively.

This ambiguity also appears in the syntax of parameter declarations, where it is caused by the fact that a parameter may be named or anonymous [ISO 2011, 6.7.6, §1]. An analogous ambiguity appears in the syntax of struct declarations [ISO 2011, 6.7.2.1, §1], where it is caused by the fact that the *struct-declarator-list* is optional.

Struct declarations exhibit another (similar) ambiguity, which is caused by the fact that a bit-field can be anonymous, as prescribed by the production *struct-declarator: declarator*$_{opt}$ **:** *constant-expression* [ISO 2011, 6.7.2.1, §1]. This is illustrated by the following code fragment:

```
// bitfield_declaration_ambiguity.c
typedef signed int T;
struct S {
  unsigned T:3; // bit-field named T with type unsigned
  const T:3;    // anonymous bit-field with type const T
};
```

This declares a structure `S` with two bit-fields. There are, however, two ways of interpreting each of the above two bit-field declarations. Indeed, in each of them, one may consider `T` either as a type specifier (in which case an anonymous bit-field of type `T` is being defined) or as a member name (in which case a bit-field named `T` is being defined).

All of the ambiguities discussed above are eliminated by paragraph 6.7.2, §2, which limits the combinations of type specifiers that may be used in a declaration or struct declaration:

> At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. Each list of type specifiers shall be one of the following multisets [...]: – **void**; – **char**; – **signed char**; [...]

In light of this paragraph, the declaration "`const T`" in `declaration_ambiguity.c` must be interpreted as follows. Because every declaration must contain at least one type specifier, `T` must be interpreted in this declaration as a type specifier. Therefore, this declaration does not declare any variable.

By the same reasoning, the bit-field "`const T:3`" in the previous example must be interpreted as a constant anonymous bit-field of type `T`.

The declaration "`unsigned T`" in `declaration_ambiguity.c` must be interpreted as follows. Among the combinations of type specifiers *not* permitted by paragraph 6.7.2, §2, one

finds `unsigned T`, where `T` is a typedef name. Thus, this cannot be a declaration of zero variables. Therefore, this must be a declaration of a variable `T` of type `unsigned`.

By the same reasoning, the bit-field "`unsigned T:3`" in the previous example must be interpreted as a bit-field named `T` whose type is `unsigned`. (It is worth noting that, therefore, this occurrence of `T` lies in the name space of "members". Thus, it does not hide the earlier declaration of `T` as a typedef name. This declaration remains visible on the following line.)

2.4. Ambiguity in parameter declarations

In addition to the ambiguity discussed above (Section 2.3), the C11 grammar exhibits a further ambiguity in the syntax of parameter declarations. Consider the following code fragment:

```
// parameter_declaration_ambiguity.c
typedef int T;
void f(int(x), int(T), int T);
// First parameter: named x, of type int
// Second parameter: anonymous, of type int(T) (i.e., T -> int)
// Third parameter: named T, of type int
```

According to the C11 grammar, in this fragment, the parameter declaration `int(T)` can be interpreted in two ways. One can read it as introducing a parameter whose name is `T` and whose type is `int`. One can also read it as introducing an anonymous parameter whose type is `int(T)`, that is, "function of `T` to `int`". In the first reading, the name of the parameter is parenthesized, as allowed by the syntax of declarators. The identifier `T`, which happens to denote a typedef name, is re-declared as a variable. In the second reading, because `T` denotes a typedef name, `(T)` forms a valid abstract declarator.

The C11 standard attempts to eliminate this ambiguity as follows [ISO 2011, 6.7.6.3, §11]:

> If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

In particular, in the previous example, the second reading must be preferred. The parameter declaration `int(T)` introduces an anonymous parameter whose type is `int(T)`.

Just like the sentence that pretends to eliminate the "dangling else" ambiguity (Section 2.2.1), this sentence is arguably unclear: it must be carefully interpreted. Indeed, consider the following code fragment, which we consider *invalid*:

```
typedef int T;
int f(int(T[const*]));
```

Here, one might argue that the identifier `T` "cannot" be treated as a typedef name, because this leads (a little later) to a syntax error: indeed, `[const*]` is not allowed by the syntax of abstract declarators[1]. Therefore, one might conclude that `T` must be treated as a parameter name. Under this interpretation, the following two declarations would be considered equivalent:

```
int f(int(T[const*]));
int f(int T[const*] );
```

[1] The production *direct-abstract-declarator*: *direct-abstract-declarator*$_{opt}$ `[ * ]` [ISO 2011, 6.7.7, §1] does not allow `const` to appear here. This is an asymmetry with the production *direct-declarator*: *direct-declarator* `[` *type-qualifier-list*$_{opt}$ `* ]`, which allows it. We do not have an explanation for this asymmetry. It would be possible to relax the syntax of abstract declarators without introducing a conflict.

As in Section 2.2.1, we argue that such an interpretation seems counter-intuitive and difficult or costly to implement. Moreover, in practice, Clang and GCC both accept `[const*]` as part of an abstract declarator, and view `T` as a typedef name in this example. Therefore, we follow Clang and GCC: when one finds an identifier `T` in this ambiguous position, one should commit to the "typedef name" interpretation before one reads further ahead. In our parser, this is done by appropriately resolving a reduce/reduce conflict (Section 3.5).

2.5. Ambiguity in the use of `_Atomic`

C11 introduces the keyword `_Atomic`. This keyword can be used in two different syntactic fashions:

— It can be used as a type qualifier, like `const`, `restrict`, and `volatile` [ISO 2011, 6.7.3, §1].
— It can be used as part of an atomic type specifier [ISO 2011, 6.7.2.4, §1]. More precisely, if τ is a type name, then `_Atomic`(τ) is a type specifier.

At first sight, this overloading of the `_Atomic` keyword appears as if it might create an ambiguity. When `_Atomic` is followed by an opening parenthesis, one does not know whether `_Atomic` is used as a type qualifier (in which case the declaration specifier list is finished, and this parenthesis is the beginning of a parenthesized declarator) or whether this is the beginning of an atomic type specifier. In order to eliminate this potential ambiguity, the C11 standard states [ISO 2011, 6.7.2.4, §4]:

> If the `_Atomic` keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.

However, this restriction is not really necessary, as paragraph 6.7.2, §2 (quoted and discussed in Section 2.3) already removes this ambiguity. Indeed, upon encountering an `_Atomic` keyword followed by an opening parenthesis, one may reason as follows:

— If a type specifier has been read already as part of the current declaration, then this `_Atomic` keyword cannot be viewed as the beginning of another type specifier. Indeed, according to the list of permitted combinations of type specifiers [ISO 2011, 6.7.2, §2], an atomic type specifier cannot be accompanied by another type specifier. Thus, in this case, `_Atomic` must be viewed as a type qualifier. (In this case, paragraph 6.7.2.4, §4 would impose rejecting the code.)
— If no type specifier has been read yet, then this `_Atomic` keyword cannot be viewed as a type qualifier. Indeed, that would imply that the parenthesis is the beginning of a parenthesized declarator, so that the declaration specifier list is finished and contains no type specifier. This is forbidden: a declaration specifier list must contain at least one type specifier. Thus, in this case, `_Atomic` must be viewed as the beginning of a type specifier.

This may seem a somewhat complicated and informal proof of unambiguity. Fortunately, the absence of ambiguity is automatically proved by LR parser generators (see Section 3.5).

3. AN LALR(1) PARSER FOR C11

Our C11 parser [Jourdan and Pottier 2017] is produced by an LALR(1) parser generator out of a grammar which we strive to keep as close as possible to the C11 grammar and as simple as possible. We assume that its input is preprocessed source code.

3.1. In search of a suitable grammar

What grammar should be provided as an input to an LALR(1) parser generator? It seems natural to first examine the C11 grammar [ISO 2011, A.2]. Let us refer to it as

$\mathcal{G}_0$. As explained earlier (Section 2.1), it is an ambiguous context-free grammar, whose most problematic source of ambiguity has to do with the various roles of identifiers. An identifier may belong to one of four name spaces [ISO 2011, 6.2.3], namely "label names", "tags", "members", and "ordinary identifiers". Furthermore, an ordinary identifier can play one of two roles: it can be viewed either as a "variable" or as a "typedef name" [ISO 2011, 6.2.1, §1]. In $\mathcal{G}_0$, *identifier* is a terminal symbol, whereas *typedef-name* is a nonterminal symbol, defined by the production *typedef-name: identifier* [ISO 2011, 6.7.8, §1]. This creates an ambiguity: $\mathcal{G}_0$ does not indicate when this production should be used, that is, when an identifier should be viewed as a typedef name and when (on the contrary) it should be viewed as a variable (or a label name, or a tag, or a member). This is done in the most part by a set of informal "scoping rules" [ISO 2011, 6.2.1] and in a lesser part by informal restrictions enunciated in other places, such as paragraph 6.7.2, §2, paragraph 6.7.6.3, §11, and paragraph 6.9.1, §6.

In order to eliminate this source of ambiguity, it is customary[2] to remove from $\mathcal{G}_0$ the production *typedef-name: identifier* and view *identifier* and *typedef-name* as two distinct terminal symbols. By doing so, one ostensibly assumes that ordinary identifiers are somehow classified prior to parsing as variables or typedef names. Let us refer to this new grammar as $\mathcal{G}_1$ [Degener 2012].

The grammar $\mathcal{G}_1$ seems remarkably well-behaved. It is "almost" LALR(1). More precisely, the automaton built for $\mathcal{G}_1$ by an LALR(1) parser generator exhibits only two conflicts: a shift/reduce conflict caused by the "dangling else" ambiguity (Section 2.2) and a shift/reduce conflict caused by the "`_Atomic(`" ambiguity (Section 2.5). In both cases, the desired behavior can be obtained by instructing the parser generator to always prefer shifting.

Unfortunately, the grammar $\mathcal{G}_1$ is too restrictive: there are valid C11 programs that it rejects. Indeed, whereas in $\mathcal{G}_0$ the terminal symbol *identifier* means "an arbitrary identifier", in $\mathcal{G}_1$ it has more restricted conventional meaning: it represents "an identifier that has been classified as a variable" (as opposed to a typedef name). In two places, namely *primary-expression* [ISO 2011, 6.5.1, §2] and *identifier-list* [ISO 2011, 6.9.1, §6], this restricted meaning is in fact desired. Therefore, in these two places, it is fine to use *identifier*. Everywhere else, the unrestricted meaning of "an arbitrary identifier" is desired. In those places, $\mathcal{G}_1$ is too restrictive and must be amended by replacing *identifier* with a choice between *identifier* and *typedef-name*. This yields a new grammar, which we refer to as $\mathcal{G}_2$.

The grammar $\mathcal{G}_2$ is slightly less well-behaved than $\mathcal{G}_1$. In addition to the two shift/reduce conflicts mentioned above, it exhibits a reduce/reduce conflict caused by the "parameter declaration" ambiguity (Section 2.4) and a shift/reduce conflict caused by the "declaration" ambiguity (Section 2.3). The former conflict can be solved by instructing the parser generator to always prefer a certain production (Section 3.5). Unfortunately, the latter conflict cannot be statically solved: neither "always shift" nor "always reduce" is the correct behavior (Section 3.4).

In summary, the grammar $\mathcal{G}_2$ is our starting point. At least two problems remain to be solved. First, one must implement a mechanism by which ordinary identifiers are (correctly) classified as variables or typedef names. Although it is well-known that such a classification must be performed on the fly via a form of "lexical feedback", the problem is trickier than it may at first seem, as we have argued earlier (Section 2).

[2]This idea probably goes back to the early days of the B and C programming languages, circa 1969–70 [Ritchie 1993]. In any case, it explicitly appears in Kernighan and Ritchie's 1988 book [Kernighan and Ritchie 1988, §A.13], which describes "K&R C". They write: "With one further change, namely deleting the production *typedef-name: identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the **if-else** ambiguity."

Our solution, which requires several changes in the grammar, is described in Section 3.3. Second, one must further modify the grammar so as to eliminate the problematic shift/reduce conflict described above. Our solution is described in Section 3.4.

3.2. Tools

We build our lexer and parser with the programming language OCaml, the lexer generator ocamllex, and the parser generator Menhir [Pottier and Régis-Gianas 2016]. The latter two can be described as the OCaml analogues of lex and yacc. Minsky *et al.* [Minsky et al. 2013, Chapter 16] give a good tutorial introduction to these tools.

The implementation of the lexer does not present any significant difficulty, except in its interaction with the parser, which is described in detail in the following.

Menhir supports several parser construction methods, including LALR(1), Pager's method [Pager 1977] and Knuth's canonical method [Knuth 1965]. For our purposes, LALR(1) is sufficient. We exploit two specific features of Menhir that improve conciseness and readability, namely parameterized nonterminal symbols [Pottier and Régis-Gianas 2016, §5.2] and inlined nonterminal symbols [Pottier and Régis-Gianas 2016, §5.3]. Both features can be expanded away (by Menhir itself), producing a version of the grammar that does not exploit these features.

This expanded grammar, with empty semantic actions, forms a valid `.y` file, which yacc and Bison [Donnelly and Stallman 2015] can read and process, and which we distribute [Jourdan and Pottier 2017]. The LALR(1) automata produced by Bison and Menhir appear to be identical. (We have checked that they have the same number of states.)

3.3. Lexical feedback

As explained in Section 3.1, we follow an approach where variables and typedef names must give rise to distinct tokens (or, in our case, distinct sequences of tokens). In an ideal world, this distinction would be made prior to parsing, in a logically separate phase. It would take the form of extra logic inserted into the lexer (or between the lexer and parser). Unfortunately, in reality, this is impossible. Indeed, as explained in Section 2, in order to tell whether an identifier is a variable or a typedef name, one must know which declarations are visible in the current scope. This in turn requires being aware of where scopes are opened and closed and where declarations take effect. To obtain this information requires parsing. Thus, a form of feedback is required: the parser must provide information to the lexer. To this end, some form of global state, which the lexer reads and the parser updates, is required. This is known elegantly as "lexical feedback", or, rather more prosaically, as "the lexer hack".

In all implementations of lexical feedback known to us, including ours, the lexer has read access to some global state, which the parser is in charge of updating. The updates take place in the semantic actions associated with certain productions. They are executed when these productions are reduced.

In all previous implementations of lexical feedback known to us, upon encountering an identifier in the source code, the lexer immediately consults the global state so as to determine whether (in the current scope) this identifier represents a variable or a typedef name, and, depending on the outcome of this test, emits distinct tokens: say, `VARIABLE` versus `TYPE`. In our approach, in contrast, upon encountering an identifier in the source code, the lexer immediately produces the token `NAME`, followed with either `VARIABLE` or `TYPE`. This delays the access by the lexer to the global state, thus eliminating the risk of incorrect classification that was illustrated in Section 2.2.2.

In the following, we first describe what global state is shared between our lexer and parser, and through which API this global state is updated and read (Section 3.3.1). Then, we explain when this global state is updated by the parser. This involves sav-

ing and restoring contexts (Section 3.3.2) and declaring new identifiers in the current context (Section 3.3.3). Finally, we explain when the lexer accesses the global state (Section 3.3.4).

3.3.1. Context API. Let us use the word "context", in a precise technical sense, to mean a "set of identifiers". A context represents a set of identifiers that should be viewed as typedef names (whereas all other identifiers should be viewed as variables). It is an immutable set.

In Our OCaml implementation, the management of contexts is encapsulated in a single module, baptized `Context`. This module is extremely simple: it is implemented in 8 lines of code. It contains one mutable global variable, namely `current`, which holds a pointer to an immutable set of identifiers. The set `!current` is "the current context"[3].

This module is used by the lexer (which consults the global state) and by the parser (which updates the global state), thus allowing information to flow from the parser to the lexer. It offers the following API:

```
val declare_typedefname : string -> unit
val declare_varname     : string -> unit
val is_typedefname      : string -> bool
type context
val save_context        : unit -> context
val restore_context     : context -> unit
```

The functions `declare_typedefname` and `declare_varname` are invoked by the parser when a new variable or typedef name is introduced: they change the current context. More precisely, the call `declare_typedefname id` adds the identifier `id` to the current context, whereas `declare_varname id` removes `id` from the current context. The function `is_typedefname` is invoked by the lexer when it needs to decide whether to emit `VARIABLE` or `TYPE`; it consults the current context. These three functions have logarithmic time complexity: OCaml's persistent set library exploits balanced binary search trees.

The function `save_context` is invoked by the parser when opening a new scope. This function returns `!current`, an immutable set of identifiers: in other words, a snapshot of the current context. This snapshot can be later re-installed (if and when desired) by invoking `restore_context`. This function simply writes its argument into `current`. These two functions have constant time complexity.

The module `Context` does not internally maintain a stack. Nor does its API impose a stack discipline: calls to `save_context` and `restore_context` need not be balanced. Thus, the API is very simple and powerful.

3.3.2. Where contexts are saved and restored. For convenience, we define a nonterminal symbol, `save_context`, which generates the empty sentence, as follows:

```
save_context: (* empty *) { save_context() }
```

When this production is reduced, its semantic action is executed and returns a snapshot of the current context. We use this nonterminal symbol at any point in the grammar where we need to save the current context in order to prepare for a possible later re-installation. We may do so quite freely, for the following reasons:

— Because `save_context` generates the empty sentence, its use has no effect on the language that is accepted by the parser.
— As noted earlier, the runtime cost of saving the context is very low.

[3] In OCaml, `!` dereferences a mutable variable.

— Since calls to `save_context` and `restore_context` need not be balanced, we may use `save_context` wherever we please, without worrying about matching calls to `restore_context`.

One possible adverse effect of inserting occurrences of `save_context` in the grammar is that this can create new shift/reduce conflicts. Such a conflict is typically between reducing the production `save_context: (*empty *)` (which means: take a snapshot now!) and shifting (which means: make progress in some other productions $p_1, p_2, \ldots$). We avoid all such conflicts by adding redundant occurrences of `save_context` in the productions $p_1, p_2, \ldots$ By doing so, we instruct the parser to always take a snapshot now, even in situations where it is not known yet whether this snapshot will be useful in the future.

Most calls to `save_context` and `restore_context` come in matching pairs. In that case, for documentation and for conciseness, it is desirable to phrase the grammar in terms of a high-level notion of "scope", instead of low-level function calls. To this end, we define a parameterized nonterminal symbol, `scoped(X)`:

```
scoped(X):
  ctx = save_context
    x = X
    { restore_context ctx; x }
```

The parameter `X` stands for a (terminal or nonterminal) symbol. By definition, `scoped(X)` generates the same language as `save_context X`, which itself generates the same language as `X`. The difference between `scoped(X)` and `X` is in the semantic actions: when one uses `scoped(X)`, the current context is saved initially and restored at the end[4]. More precisely, when the production `scoped(X): save_context X` is reduced, the top two cells of the parser's stack respectively contain a saved context `ctx` and an abstract syntax tree `x` for the symbol `X`. The semantic action `restore_context ctx; x` re-installs the context `ctx` and returns `x`. This explains why we do not need to explicitly maintain a stack of contexts: saved contexts are naturally stored in the parser's stack. An analogous idea appears in Bison's manual [Donnelly and Stallman 2015, 3.4.8.1]. It is unfortunately expressed in a much lower-level style that makes it more difficult to understand and notationally heavier.

We use this construction, among other places, in the syntax of statements:

```
statement:
  labeled_statement
| scoped(compound_statement)
| expression_statement
| scoped(selection_statement)
| scoped(iteration_statement)
| jump_statement
    {}
```

This elegantly reflects the fact that compound statements, selection statements, and iteration statements form a "block" (Section 2.1.4) and therefore require opening and closing a new scope.

Function definitions are more difficult to deal with, in part because one cannot distinguish a declaration and a function definition until one reaches a function body, and in part because in a function definition the scope where the parameters are visible is noncontiguous. (Both phenomena have been illustrated in Section 2.1.4.)

[4]Again, `ctx` and `x` are OCaml variables: they are bound to the semantic values returned by `save_context` and `X`, respectively. One could replace them in the semantic action with `$1` and `$2`.

To deal with these difficulties, we build (simplified) abstract syntax trees for declarators. An abstract syntax tree of OCaml type `declarator` stores the identifier that is declared and (if this is a function declarator) a snapshot of the context at the end of the parameter list. The API offered by these abstract syntax trees is as follows:

```
type declarator
(* Constructors. *)
val identifier_declarator: string -> declarator
val function_declarator: declarator -> context -> declarator
val other_declarator: declarator -> declarator
(* Accessors. *)
val identifier: declarator -> string
val reinstall_function_context: declarator -> unit
```

The three constructors above are used in the semantic actions associated with the nonterminal symbols `declarator` and `direct_declarator`. For instance, the following production recognizes a function declarator and builds an abstract syntax tree for it:

```
direct_declarator:
  ...
| d = direct_declarator LPAREN ctx = scoped(parameter_type_list) RPAREN
    { function_declarator d ctx }
| ...
```

The variable `ctx` above represents a snapshot of the context at the end of the parameter list. This stems from the following definition:

```
parameter_type_list:
  parameter_list option(COMMA ELLIPSIS {}) ctx = save_context { ctx }
```

The accessor function `identifier` retrieves the identifier that is declared by a declarator. It is used in the definition of the nonterminal symbols `declarator_varname` and `declarator_typedefname`, which recognize the same language as `declarator` and have the added side effect of declaring this identifier (as a variable or as a typedef name, respectively) in the current scope:

```
declarator_varname:
  d = declarator { declare_varname     (identifier d); d }

declarator_typedefname:
  d = declarator { declare_typedefname (identifier d); d }
```

The accessor function `reinstall_function_context` expects its argument `d` to be a function declarator. It retrieves the context that was saved at the end of the parameter list and re-installs it (by invoking `restore_context`). On top of that, it declares the function's name as a variable, by calling `declare_varname (identifier d)`.

Putting the pieces together, a function definition is described as follows:

```
function_definition1:
  declaration_specifiers d = declarator_varname
    { let ctx = save_context () in
      reinstall_function_context d;
      ctx }

function_definition:
  ctx = function_definition1 declaration_list? compound_statement
    { restore_context ctx }
```

Assume, for the sake of simplicity, that the parser is reading a function definition whose `declaration_list?` is empty. Then, the nonterminal symbol `function_definition1` is

reduced when the parser looks ahead at the opening brace that forms the beginning of the `coumpound_statement`. At this point, the current context is saved under the name `ctx`. (This context includes the function's name, declared as a variable, because we have used `declarator_varname`.) Then, the call `reinstall_function_context d` restores the context that existed at the end of the parameter list (which has been saved as part of the abstract syntax tree `d`) and, on top of it, declares the function's name. The function body is parsed in this scope. Upon completion, this scope is closed by restoring the context `ctx`.

3.3.3. Where declarations take effect. In the C11 grammar, "variable" declarations and "type" declarations are described by a single nonterminal symbol, *declaration* [ISO 2011, 6.7]. In our parser, however, we must know, at the end of each declarator, whether this declarator introduces a new variable or a new typedef name. Indeed, we must make the lexer aware of this new name, by updating the current context.

For this reason, we distinguish between two kinds of declarations: those that contain one occurrence of the **`typedef`** keyword, and those that do not. (The C11 standard forbids multiple occurrences of **`typedef`**.) A list of declaration specifiers that does not contain **`typedef`** is described by the nonterminal symbol `declaration_specifiers`, whereas a list that contains one occurrence of **`typedef`** is described by the nonterminal symbol `declaration_specifiers_typedef`. (The definitions of these symbols are given and explained in Section 3.4.)

This allows us to distinguish between "variable" declarations and "type" declarations, as follows:

```
declaration:
  declaration_specifiers
  init_declarator_list(declarator_varname)?     SEMICOLON
| declaration_specifiers_typedef
  init_declarator_list(declarator_typedefname)? SEMICOLON
| static_assert_declaration
    {}
```

The symbols `init_declarator_list` and `init_declarator` are defined as in the C11 grammar, except they are parameterized over `declarator`. This parameter can be instantiated, as desired, with `declarator_varname` or `declarator_typedefname` (whose definitions have been shown in Section 3.3.2). Thus, we obtain the desired effect: the absence or presence of **`typedef`** among the declaration specifiers influences the manner in which the declarators update the current context.

3.3.4. Token sequences for identifiers. The lexer uses the current context in order to classify identifiers and emit appropriate tokens. However, an LALR(1) parser may request one token from the lexer, known as a "lookahead" token, *before* it decides whether a production should be reduced and which production should be reduced. As noted earlier (Section 2.2.2), this introduces a danger of mis-classification of an identifier by the lexer. Indeed, the lexer classifies this identifier in the current scope, whereas perhaps the parser is just about to perform a reduction that affects the current scope (by declaring a new identifier in this scope, or by closing this scope). This is not just a theoretical possibility: it is the root of the bug that we have identified in GCC [Jourdan 2015].

Some parser generators, such as Bison, guarantee that, in a "defaulted state", the parser will not request a lookahead token from the lexer[5]. By relying on this feature, one might hope to cleverly avoid the danger. We have no such luck,

[5]The Bison manual [Donnelly and Stallman 2015, §5.8.2] notes: "The presence of defaulted states is an important consideration [...]. That is, if the behavior of yylex can influence or be influenced by the semantic actions [...], then the delay of the next yylex invocation until after those reductions is significant. For exam-

though: in the example of Section 2.2.2, the problem arises in a state where the parser will either shift the token `ELSE` or reduce the production `selection_statement: IF LPAREN expression RPAREN scoped(statement)`. (This is the very state where the "dangling else" shift-reduce conflict arises.) This is not a defaulted state: hence, the guarantee offered by Bison does not help. Furthermore, Menhir does not at present offer this guarantee, so we would rather not rely on it.

To work around this danger, we introduce an original idea. Upon encountering an identifier, our lexer emits a sequence of *two* tokens. The first token is always `NAME`. It is produced by the lexer without consulting the global state. The second token is either `VARIABLE` or `TYPE`, depending on the current status of this identifier. Because the distinction is delayed until the second token is demanded by the parser, the danger of mis-classification disappears. By the time the second token is demanded, the global state must have been updated already (if necessary) and cannot be outdated. Indeed, one can check[6] that no production with a side effect can be reduced when the lookahead token is "fragile", that is, when it is `VARIABLE` or `TYPE`.

Our grammar has three nonterminal symbols that correspond to an identifier[7]:

```
typedef_name:       i = NAME TYPE                   { i }
var_name:           i = NAME VARIABLE               { i }
general_identifier: i = typedef_name | i = var_name { i }
```

Where the C11 grammar uses *typedef-name*, we use `typedef_name`. Where the C11 grammar uses *identifier*, we use `var_name` or `general_identifier`. More precisely, `var_name` is used when referring to a variable in an expression (`primary_expression: var_name` [ISO 2011, 6.5.1, §2]) or in the identifier list of a K&R function declarator[8] (`identifier_list: var_name | identifier_list COMMA var_name`), whereas `general_identifier` is used everywhere else.

3.4. Controlling type specifiers in declarations and struct declarations

As explained in Section 2.3, the C11 grammar exhibits an ambiguity in the syntax of declarations. This ambiguity gives rise in the grammar $\mathcal{G}_2$ to a shift/reduce conflict (Section 3.1). In short, this conflict can be explained as follows: when the parser has read a list of declaration specifiers and finds that the next input symbol is a typedef name, it must choose between shifting (which means that the list of declaration specifiers is not finished, and this typedef name is a type specifier) and reducing (which means that the list of declaration specifiers is finished, and this typedef name is the beginning of a declarator). This conflict cannot be solved via a precedence declaration. Indeed, neither "always shift" nor "always reduce" is the correct behavior.

As explained in Section 2.3, in order to eliminate the ambiguity, the C11 standard imposes a well-formedness constraint on lists of declaration specifiers [ISO 2011,

---

ple, the semantic actions might pop a scope stack that yylex uses to determine what token to return. Thus, the delay might be necessary to ensure that yylex does not look up the next token in a scope that should already be considered closed."

[6] In our grammar, `VARIABLE` and `TYPE` always follow `NAME`. In other words, they never follow a nonterminal symbol, and they are never the first symbol in a right-hand side. Therefore, no reduction can possibly take place when `VARIABLE` or `TYPE` is the lookahead symbol. This reasoning holds even in the presence of default reductions.

[7] In Menhir, the syntax "`i =`" introduces an OCaml variable `i` and binds it to the semantic value of the symbol that follows. (An analogous feature in Bison is known as a "named reference" [Donnelly and Stallman 2015, §3.6].) Here, one could also remove all occurrences of "`i =`" and write "`$1`" instead of "`i`" in the semantic actions.

[8] The C11 standard [ISO 2011, 6.9.1, §6] does not allow a typedef name to occur in the identifier list of a K&R function declarator. Indeed, it prescribes: "An identifier declared as a typedef name shall not be redeclared as a parameter".

6.7.2, §2]. Similarly, in order to eliminate the shift/reduce conflict, we must encode this constraint into our grammar. Fortunately, we need not enforce this constraint exactly. We may enforce a weaker constraint, provided it is still strong enough to eliminate the conflict. Checking that the C11 standard is respected is deferred to a later semantic analysis. Our aim is to keep our grammar as simple as possible.

We split type specifiers in two disjoint categories:

— "Unique" type specifiers cannot occur together with any other type specifiers in a declaration. They are **`void`**, **`_Bool`**, atomic type specifiers, struct or union specifiers, enumeration specifiers, and typedef names.
— "Nonunique" type specifiers can occur together with other (nonunique) type specifiers in a declaration. They are **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`signed`**, **`unsigned`** and **`_Complex`**.

It follows from paragraph 6.7.2, §2 that a list of declaration specifiers either contains exactly one unique type specifier or contains one or more nonunique type specifiers.

Fortunately, lists restricted by simple numeric constraints can be described in the formalism of LR(1) grammars. For instance, the parameterized nonterminal symbol `list_eq1(A, B)` describes a list whose elements are `A`'s or `B`'s and which contains exactly one element of type `A`:

```
list_eq1(A, B):
  A B* | B list_eq1(A, B) {}
```

This definition states that such a list either begins with `A`, in which case the remainder of the list must match `B*`, or begins with `B`, in which case the remainder of the list must match `list_eq1(A, B)` again. Similarly, we define `list_ge1(A, B)`, which describes a list whose elements are `A`'s or `B`'s and which contains at least one element of type `A`:

```
list_ge1(A, B):
  A B* | A list_ge1(A, B) | B list_ge1(A, B) {}
```

Equipped with these tools, we can easily require every list of declaration specifiers to either contain exactly one unique type specifier or contain at least one nonunique type specifier:

```
declaration_specifiers:
  list_eq1(type_specifier_unique,    declaration_specifier)
| list_ge1(type_specifier_nonunique, declaration_specifier)
    {}
```

The nonterminal symbols `type_specifier_unique` and `type_specifier_nonunique` are as described above. The nonterminal symbol `declaration_specifier` corresponds to a declaration specifier that is neither a type specifier nor the **`typedef`** keyword. (Our handling of **`typedef`** is discussed below.)

This grammar fragment is amenable to LR(1) parsing. As long as it finds `declaration_specifier`s, the LR automaton accumulates them on its stack, exploring in parallel the possibility that this could be a `list_eq1(type_specifier_unique, ...)` and the possibility that this could be a `list_ge1(type_specifier_nonunique, ...)`. As soon as it finds a type specifier (which must be either unique or nonunique, but cannot be both), one of these possibilities is discarded.

The nonterminal symbol `declaration_specifiers_typedef`, which was mentioned earlier (Section 3.3.3), is defined in an analogous manner. Like `declaration_specifiers`, it requires the list to either contain exactly one unique type specifier or contain at least one nonunique type specifier. In addition, it requires the list to exhibit exactly one occurrence of **`typedef`**.

```
declaration_specifiers_typedef:
  list_eq1_eq1(TYPEDEF,
               type_specifier_unique,    declaration_specifier)
| list_eq1_ge1(TYPEDEF,
               type_specifier_nonunique, declaration_specifier)
    {}
```

The parameterized nonterminal symbol `list_eq1_eq1(A, B, C)` describes a list of `A`'s, `B`'s, and `C`'s that contains exactly one `A` and exactly one `B`. Similarly, `list_eq1_ge1(A, B, C)` describes a list of `A`'s, `B`'s, and `C`'s that contains exactly one `A` and at least one `B`. Their definitions are omitted.

3.5. Wrapping up

We have presented the main two ingredients of our approach, namely the original manner in which we set up lexical feedback (Section 3.3) and the manner in which we avoid the conflict caused by the "declaration" ambiguity (Section 3.4). There remain two conflicts in the grammar. They are statically solved, as follows.

The "dangling else" ambiguity (Section 2.2) gives rise to a well-known shift-reduce conflict. This conflict is solved via a suitable precedence declaration [Aho et al. 1986, Chapter 4], which causes shifting to be preferred to reduction when the lookahead symbol is `ELSE`:

```
selection_statement:
  IF LPAREN expression RPAREN scoped(statement) ELSE scoped(statement)
| IF LPAREN expression RPAREN scoped(statement) %prec below_ELSE
| SWITCH LPAREN expression RPAREN scoped(statement)
```

The "parameter declaration" ambiguity (Section 2.4) gives rise in our grammar to a reduce/reduce conflict. Initially, this conflict is between the productions `general_identifier: typedef_name` (Section 3.3.4) and `type_specifier_unique: typedef_name` (Section 3.4). It appears in only one state of the automaton, and only when the lookahead symbol is '`(`', '`)`', or '`[`'. By studying how one may reach this situation, one finds that it is reachable only via a parameter declaration. Thus, paragraph 6.7.6.3, §11 applies, and prescribes that the typedef name on top of the parser's stack should be viewed as a type specifier. Thus, the parser generator must be instructed to always reduce the second of the above two productions. Menhir, like Bison [Donnelly and Stallman 2015, §5.6], always favors the production that appears first in the grammar. In our grammar, the production `general_identifier: typedef_name` happens to come first, and we do not wish to change that, as we want to keep the rules in the same order as they appear in the C11 grammar [ISO 2011, A.2]. So, we replace `type_specifier_unique: typedef_name` with `type_specifier_unique: typedef_name_spec` and add the production `typedef_name_spec: typedef_name` at the beginning of the grammar. This has the desired effect.

The "`_Atomic(`" ambiguity (Section 2.5) gives rise to a shift/reduce conflict in the grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ (Section 3.1). Fortunately, in our grammar, this conflict is gone. As explained in Section 2.5, the restriction imposed by paragraph 6.7.2, §2 eliminates the ambiguity; and we are fortunate enough that our implementation of this restriction (Section 3.4) eliminates the conflict.

Because the restriction of paragraph 6.7.2.4, §4 is not necessary, our parser does not, by default, implement it. Therefore, it accepts a slightly larger language than prescribed by the C11 standard. For instance, it accepts the following noncompliant code:

```
// atomic_parenthesis.c
int _Atomic (x);
```

In case strict compliance is needed, our parser can be configured to enforce the restriction of paragraph 6.7.2.4, §4 and reject the above code. To do this, we let the lexer emit the special token `ATOMIC_LPAREN`, instead of `LPAREN`, when it encounters an opening parenthesis that immediately follows an **`_Atomic`** keyword. Furthermore, we add the production `atomic_type_specifier: ATOMIC ATOMIC_LPAREN type_name RPAREN`. Thus, a parenthesis that follows **`_Atomic`** must be part of an atomic type specifier; it cannot be interpreted in any other way.

4. RELATED WORK

Our approach to parsing C11 combines a deterministic parser (produced by an LALR(1) parser generator, based on a slightly modified version of the C11 grammar) and a deterministic "lexical feedback" mechanism, which distinguishes between variables and typedef names. This approach follows a well-established tradition, which we discuss in Section 4.1. However, there are other approaches to parsing C (and, more generally, C++), which we discuss in Section 4.2.

4.1. LR parsers and lexical feedback

Lexical feedback in an LR parser is a well-known technique, and has been in use for a long time. Mason, Brown, and Levine's book [Brown et al. 1992, Chapter 7] offers a brief description of it. It is however a rather subtle and fragile technique, due in particular to a possible adverse interaction between lexical feedback and lookahead. There is a danger that the lexer reads one token ahead, before the parser has updated the global state that informs the lexer. This would cause the lexer to deliver an incorrect token. The issue is complicated by the fact that, in the presence of default reductions, it can be hard to understand exactly when the parser requests a new token from the lexer.

This issue, as well as the problem of classifying identifiers in a C parser, is discussed at length in a very interesting thread on the newsgroup `comp.compilers` [lex 1992].

Deep in this thread, Jim Roskind explains how to prove the absence of adverse interaction between lexical feedback and lookahead: "[one] must show that [no] reductions involved with annotation of [the] symbol table take place while [a fragile token] is waiting in the lookahead buffer". We have used this argument in Section 3.3.4, where (in our case) no reduction at all can take place when the lookahead token is a fragile token (that is, `VARIABLE` or `TYPE`). Roskind also asserts that "the typedef-ness of an identifier can change *only* after a declarator is complete". Technically, this is false: it can change also when a scope is closed and when defining a new enumeration constant. In ANSI C, at the end of a scope, the lookahead token must be '`}`' or '`)`', neither of which is fragile, so all is well. In C99, however, a selection statement forms a block, so a scope is closed at the end of such a statement. Due to the "**`if-else`**" ambiguity, determining whether an **`if`** statement is finished requires looking ahead at the next token, which could be fragile. Therefore, there is a danger of misclassification, which we have discussed in Section 2.2.2, identifying a bug in GCC [Jourdan 2015].

In the same thread, Dale R. Worley puts forth the idea that, in order to eliminate the ambiguity in declarations, "the grammar must keep track of whether a type-specifier has been seen in the declaration-specifiers". His changes to the grammar are analogous in spirit to those we present in Section 3.4. We keep track of more information (namely, the distinction between unique and nonunique type specifiers, and the absence or presence of **`typedef`**) and we use parameterized nonterminal symbols to preserve readability.

Roskind is the author of an LALR(1) grammar for ANSI C and for C++. These grammars are available online [Roskind 1990], together with detailed explanations, which unfortunately seem to concern mostly C++. Roskind goes to great lengths to eliminate ambiguity: his ANSI C grammar exhibits only one shift/reduce conflict, namely

the benign "dangling else" conflict, and no reduce/reduce conflict. The subgrammars of declarations, parameter declarations, and declarators appear to have been quite deeply re-worked. Unfortunately, Roskind's published grammar does not contain semantic actions; it would take some work to reconstruct the lexical feedback mechanism.

Up until version 3.4.4, GCC used an LALR(1) parser [The GNU project 2004]. Like Roskind's [Roskind 1990] and Worley's (cited above), this parser has (deeply) re-worked subgrammars of declarations, declarators, declaration specifiers, etc. It uses explicit auxiliary stacks to keep track of which scopes have been opened and (in a declaration) which declaration specifiers have been seen. It also stores information on the parser's stack, in the same way as we store saved contexts on the parser's stack (Section 3.3.2). We are perhaps the first to point out that no auxiliary stack is needed. Since version 4.0.0, GCC uses a hand-written recursive descent parser, which today is about 16Kloc. This parser accepts a much larger language than C11: it accepts Objective C as well as many GNU extensions.

Clang, too, uses a hand-written recursive descent parser. Even though the recursive descent parsers in GCC and Clang are hand-written, they use a lexical feedback mechanism, or "lexer hack". Some claim that Clang's approach does not use a "hack" [Bendersky 2012]. In our eyes, whether the global state resides in the lexer itself, or in a separate "symbol table" module (such as our `Context` module, presented in Section 3.3.1) makes relatively little difference: the on-the-fly classification of identifiers remains difficult to get right, as evidenced by the bug that we found in GCC [Jourdan 2015].

The CompCert C compiler [Leroy 2009; Leroy 2017] contains two parsers. The CompCert "pre-parser" (in `cparser/pre_parser.mly`) is an LALR(1) parser equipped with a lexical feedback mechanism. The C11 parser presented in this paper is based on it. The pre-parser is in charge of detecting and reporting syntax errors, following a technique proposed by Jeffery [Jeffery 2003] and implemented in Menhir by the second author [Pottier 2016]. The CompCert "parser" (in `cparser/Parser.vy`) is also an LALR(1) parser, but does not need lexical feedback: it parses the input again, exploiting the classification of identifiers that has been performed already by the pre-parser. As a result, it is very simple. Its grammar is very close to the C11 grammar. It is LALR(1): it has no conflicts whatsoever. In particular, the "dangling else" conflict is resolved by distinguishing "open-ended" and "non-open-ended" `if` statements (Section 2.2.2). This is done without actually duplicating the subgrammar of statements, thanks to parameterized terminal symbols. Out of this grammar, Menhir produces an executable parser, written in the Coq programming language, whose correctness can be verified by the Coq proof assistant [Jourdan et al. 2012].

Van Wyk and Schwerdfeger [Van Wyk and Schwerdfeger 2007] propose a "context-aware" variant of the LR algorithm where the parser passes to the lexer a "valid lookahead set", that is, a set of valid symbols that the lexer is permitted to return at this point. However, this does not solve the "typedef name" ambiguity, so, when parsing C, lexical feedback is still required [Van Wyk and Schwerdfeger 2007, §6.1]. Furthermore, because the computation of the lookahead symbol depends on the current control state of the parser, the lookahead symbol must be re-computed after each reduction [Van Wyk and Schwerdfeger 2007, §6.2.1]. Van Wyk and Schwerdfeger use this technique to develop an extensible variant of ANSI C.

A short paper by Scarpazza [Scarpazza 2007] presents an approach to parsing ANSI C using flex, Bison, and lexical feedback. When parsing a declaration, Scarpazza insists that the identifier that is being declared should be classified by the lexer as a variable. (In contrast, we let the lexer randomly classify this identifier, and compensate by accepting `general_identifier` in our grammar.) This leads Scarpazza to set up an elaborate form of lexical feedback, whereby the parser keeps track of "type stacks" and uses them to inform the lexer.

4.2. Other approaches

Nondeterministic parsing algorithms support general (possibly ambiguous) context-free grammars. Examples of such algorithms include various forms of backtracking LR, as implemented for instance in BtYacc [Dodd and Maslov 1999]; GLR [Grune and Jacobs 2008, §11.1]; Earley's algorithm [Grune and Jacobs 2008, §7.2]; and many more. These algorithms explore several potential parses (either one after the other or in parallel), discarding those that lead to a syntax error.

Because they support ambiguous grammars, nondeterministic parsing algorithms can be applied, *without* lexical feedback, to the C11 grammar, as it appears in the C11 standard. In the newsgroup thread cited earlier [lex 1992], Jan Rekers argues in favor of using GLR, without lexical feedback, for parsing C. Baxter *et al.* [Baxter et al. 2004] also report using GLR to parse many languages, including C and C++. McPeak and Necula present a hybrid GLR/LR parsing algorithm, and use it in a C++ parser [McPeak and Necula 2004]. However, this approach does not magically eliminate the difficulty of parsing C11. Because the C11 grammar is ambiguous, a nondeterministic parsing algorithm produces an exponential number of potential parse trees. Well-formedness filters (applied either a posteriori or on the fly) are necessary so as to discard those trees that do not respect the informal prose in the C11 standard. These filters must implement the scoping rules as well as several paragraphs of the standard that have been cited in Section 2. They could be just as difficult to implement correctly as lexical feedback.

A nondeterministic parsing algorithm can also be used *with* lexical feedback. In that case, the side effects of the semantic actions must be undone when backtracking. For instance, Thurston and Cordy develop a backtracking LR algorithm and apply it to C++ [Thurston and Cordy 2006]. Their paper contains a good survey of the related work in this area.

A way of avoiding lexical feedback, which may be used in a deterministic or nondeterministic parser, is to modify the grammar so that it does not require a distinction between variables and typedef names, yet is not ambiguous. This may require unifying several syntactic categories that are considered distinct in the official grammar. For instance, a parser might recognize "`T * b`" as a "*declaration-or-expression*", letting a later semantic analysis determine whether (in the context where it occurs) it should be interpreted as a declaration or as an expression. Willink [Willink 2001, Chapter 5] adopts this approach, which he calls the "superset grammar" approach, in a nondeterministic parser for C++. This approach may well be applicable to C11. Unfortunately, it imposes pervasive changes on the grammar and requires implementing an ad hoc semantic analysis. It may be difficult to argue that the modified grammar, in conjunction with the semantic analysis, complies exactly with the C11 standard.

Grimm [Grimm 2006] describes *Rats!*, a parser generator based on parsing expression grammars (PEGs), as opposed to context-free grammars. Instead of symmetric choice, PEGs offer an ordered choice operator, therefore eliminating all ambiguity by construction. They also support syntactic predicates, which match the input without consuming it. PEG parsers are backtracking parsers. They achieve linear time and space complexity via memoization. Grimm discusses the use of *Rats!* to parse C [Grimm 2006, §5]. He notes that the added power of PEGs makes it easier to deal with certain aspects of C. For instance, there is no need to alter the grammar so as to track the presence of type specifiers in a sequence of declaration specifiers, as we did in Section 3.4. To classify identifiers, Grimm uses a standard lexical feedback mechanism. In the presence of backtracking, the side effects required by this mechanism must be performed within (nested) transactions.

Semantic predicates makes it possible to invalidate a production based on an arbitrary predicate evaluated at parse time. They are available in several parser generators, including *Rats!*, ANTLR and in the GLR mode of Bison. They could, in principle, lift the need for a lexical feedback mechanism by disabling the reductions containing an identifier token when this identifier does not have the right status in the current environment. However, to the best of our knowledge, we are not aware of any C11 parser that would use semantic predicates.

We have assumed that the C source code has been preprocessed and therefore contains no preprocessor directives. In contrast, some authors attempt to parse without expanding macros, without reading #included files, and/or without resolving static conditionals.

Padioleau [Padioleau 2009], for instance, performs neither of these three tasks. He extends the grammar of C with new terminal symbols, including `#include`, `#ifdef`, etc., as well as several classes of identifiers. Indeed, in addition to typedef names and variables, he wishes to recognize identifiers that denote macros. Furthermore, he wishes to distinguish between several typical families of macros, such as macros that expand to statements, macros that expand to loop headers, and so on. This classification involves heuristic aspects (for instance, it exploits indentation). It takes the form of a transformation of the token stream, which is inserted between the lexer and the parser. Padioleau's work is used in Coccinelle [Padioleau et al. 2008], a tool that helps analyze and evolve C code.

Gazzillo and Grimm [Gazzillo and Grimm 2012] expand macros and read #included files, but do not resolve static conditionals, as they wish to parse all possible "configurations" of the software simultaneously. They introduce a new LR engine, Fork-Merge-LR, which relies on standard LALR(1) parse tables. Like GLR, it is a nondeterministic parsing algorithm. However, the need for nondeterminism does not stem from ambiguity in the grammar; it stems from the desire to parse all possible configurations simultaneously. Gazzillo and Grimm's use Roskind's grammar, extended with support for common gcc extensions. Their parser uses a form of lexical feedback [Gazzillo and Grimm 2012, §5.2]. When it encounters an identifier that denotes a typedef name in one configuration and a variable in another configuration, it forks two subparsers.

5. CONCLUSION

In this paper, we have reviewed several of the difficulties that arise in writing a C11 parser, and we have presented such a parser. It is a slightly simplified version of the "pre-parser" found in the CompCert C compiler.

It is very concise: the grammar, including (just) the semantic actions necessary to perform lexical feedback, is under 500 (nonblank, noncomment) lines of code; the lexer and the support code are less than 400 lines.

To the best of our knowledge, this parser complies with the C11 standard. However, it has not been extensively tested. (Running it on large existing code bases would require extending it with support for the most commonly used extensions of C, such as the GNU extensions.) We offer only a small suite of "tricky" C11 programs that can be used as a "torture test".

There exist many commercial and academic tools that can parse C. Some of them have been discussed in Section 4. Unfortunately, these parsers are often either designed for an older version of the language (such as C89) or plain incorrect. The C11 parsers found in popular compilers, such as GCC and Clang, are very likely correct, but their size is in the tens of thousands of lines. Therefore, we believe that there is a need for a simple reference implementation that is easy to adopt and extend. We hope that our parser can play this role.

Although we discuss only C11 in this paper, our parser can also parse C99, which is a subset of C11. C89, on the other hand, is not a subset of C11. There are differences concerning the scoping rules and, more importantly, the "implicit `int`" rule, which allows declarations without a type specifier, in which case the default type `int` is used [ANSI 1989, 3.5.2]. This makes C89 more difficult to parse. Our distribution [Jourdan and Pottier 2017] includes a C89 parser, which is separate from our C11 parser. (It is based on a different grammar.) It supports the "implicit `int`" rule, handles all C11 constructs, and supports either set of scoping rules, depending on its configuration. Therefore, when configured to use the C89 scoping rules, this parser accepts every C89 program; and, when configured to use the C99/C11 scoping rules, it accepts every C99 or C11 program. For the sake of brevity, it is not discussed in the paper.

REFERENCES

1992. Lookahead vs. Scanner Feedback. https://groups.google.com/forum/#!topic/comp.compilers/gqeQy3mXqnA. (Jan. 1992).

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

ANSI. 1989. ANSI X3.159-1989 Programming Language C. (1989).

Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. 2004. DMS®: Program Transformations for Practical Scalable Software Evolution. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 625–634.

Eli Bendersky. 2012. How Clang handles the type / variable name ambiguity of C/C++. (July 2012). http://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc/.

Doug Brown, John Levine, and Tony Mason. 1992. *Lex & Yacc, 2nd Edition*. O'Reilly Media.

Jutta Degener. 2012. C11 Yacc grammar. http://www.quut.com/c/ANSI-C-grammar-y.html. (Dec. 2012).

Chris Dodd and Vadim Maslov. 1999. *BtYacc*.

Charles Donnelly and Richard Stallman. 2015. *Bison*.

Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 323–334.

Robert Grimm. 2006. Better extensibility through modular syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 38–51.

Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing techniques: a practical guide, second edition*. Springer.

ISO. 2011. ISO/IEC 9899:2011 – Programming languages – C. (2011).

Clinton L. Jeffery. 2003. Generating LR syntax error messages from examples. 25, 5 (2003), 631–640.

Jacques-Henri Jourdan. 2015. GCC bug #67784: Incorrect parsing when using declarations in for loops and typedefs. (Sept. 2015).

Jacques-Henri Jourdan and François Pottier. 2017. A simple, possibly correct LR parser for C11 – implementation and test suite. https://github.com/jhjourdan/C11parser. (Feb. 2017).

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 397–416.

Brian W. Kernighan and Dennis Ritchie. 1988. *The C Programming Language, Second Edition*. Prentice Hall.

Donald E. Knuth. 1965. On the translation of languages from left to right. *Information & Control* 8, 6 (1965), 607–639.

Xavier Leroy. 2009. Formal verification of a realistic compiler. 52, 7 (2009), 107–115.

Xavier Leroy. 2017. The CompCert C verified compiler. https://github.com/AbsInt/CompCert. (2017).

Scott McPeak and George C. Necula. 2004. Elkhound: a Fast, Practical GLR Parser Generator. In *International Conference on Compiler Construction (CC) (Lecture Notes in Computer Science)*, Vol. 2985. Springer, 73–88.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml: Functional programming for the masses*. O'Reilly.

Yoann Padioleau. 2009. Parsing C/C++ Code Without Pre-processing. In *International Conference on Compiler Construction (CC) (Lecture Notes in Computer Science)*, Vol. 5501. Springer, 109–125.

Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*. 247–260.

David Pager. 1977. A Practical General Method for Constructing LR(k) Parsers. *Acta Informatica* 7 (1977), 249–268.

François Pottier. 2016. Reachability and error diagnosis in LR(1) parsers. In *International Conference on Compiler Construction (CC)*.

François Pottier and Yann Régis-Gianas. 2016. The Menhir parser generator. (2016).

Dennis M. Ritchie. 1993. The Development of the C Language. In *Second ACM SIGPLAN Conference on History of Programming Languages (HOPL)*. 201–208.

Jim Roskind. 1990. A grammar for ANSI C. http://www.ccs.neu.edu/research/demeter/tools/master/doc/headers/C++Grammar/. (July 1990).

Daniele Paolo Scarpazza. 2007. Practical Parsing for ANSI C. *Dr. Dobb's* 392 (Jan. 2007), 48–55.

The GNU project. 2004. GCC 3.4.4, `gcc/c-parse.in`. (Oct. 2004).

Adrian D. Thurston and James R. Cordy. 2006. A Backtracking LR Algorithm for Parsing Ambiguous Context-dependent Languages. In *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*.

Eric R. Van Wyk and August C. Schwerdfeger. 2007. Context-aware Scanning for Parsing Extensible Languages. In *International Conference on Generative Programming and Component Engineering (GPCE)*. 63–72.

Edward D. Willink. 2001. *Meta-Compilation for C++*. Ph.D. Dissertation. University of Surrey.

A. GRAMMAR

```
%{
  open Context
  open Declarator
%}

%token<string> NAME
%token VARIABLE TYPE
%token CONSTANT STRING_LITERAL
%token
   ALIGNAS ALIGNOF ATOMIC BOOL COMPLEX IMAGINARY GENERIC NORETURN STATIC_ASSERT
   THREAD_LOCAL AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM
   EXTERN FLOAT FOR GOTO IF INLINE INT LONG REGISTER RESTRICT RETURN SHORT
   SIGNED SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE
%token
   PTR INC DEC LEFT RIGHT LEQ GEQ EQEQ EQ NEQ LT GT ANDAND BARBAR PLUS MINUS
   STAR TILDE BANG SLASH PERCENT HAT BAR QUESTION COLON AND MUL_ASSIGN
   DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN
   AND_ASSIGN XOR_ASSIGN OR_ASSIGN LPAREN ATOMIC_LPAREN RPAREN LBRACK RBRACK
   LBRACE RBRACE DOT COMMA SEMICOLON ELLIPSIS
%token EOF

%type<context> save_context parameter_type_list function_definition1
%type<string> typedef_name var_name general_identifier enumeration_constant
%type<declarator> declarator direct_declarator

(* There is a reduce/reduce conflict in the grammar. It corresponds to the
   conflict in the second declaration in the following snippet:

     typedef int T;
     int f(int(T));

   It is specified by 6.7.6.3 11: 'T' should be taken as the type of the
   parameter of the anonymous function taken as a parameter by f (thus,
   f has type (T -> int) -> int).

   The reduce/reduce conflict is solved by letting menhir reduce the production
   appearing first in this file. This is the reason why we have the
   [typedef_name_spec] proxy: it is here just to make sure the conflicting
   production appears before the other (which concerns [general_identifier]). *)

(* These precedence declarations solve the dangling else conflict. *)
%nonassoc below_ELSE
%nonassoc ELSE

%start<unit> translation_unit_file

%%

(* Helpers *)

(* [option(X)] represents a choice between nothing and [X].
   [ioption(X)] is the same thing, but is inlined at its use site,
   which in some cases is necessary in order to avoid a conflict.
   By convention, [X?] is syntactic sugar for [option(X)]. *)

%inline ioption(X):
| /* nothing */
| X
    {}
```

```
option(X):
| o = ioption(X)
    { o }

(* By convention, [X*] is syntactic sugar for [list(X)]. *)

list(X):
| /* nothing */
| X list(X)
    {}

(* A list of A's and B's that contains exactly one A: *)

list_eq1(A, B):
| A B*
| B list_eq1(A, B)
    {}

(* A list of A's and B's that contains at least one A: *)

list_ge1(A, B):
| A B*
| A list_ge1(A, B)
| B list_ge1(A, B)
    {}

(* A list of A's, B's and C's that contains exactly one A and exactly one B: *)

list_eq1_eq1(A, B, C):
| A list_eq1(B, C)
| B list_eq1(A, C)
| C list_eq1_eq1(A, B, C)
    {}

(* A list of A's, B's and C's that contains exactly one A and at least one B: *)

list_eq1_ge1(A, B, C):
| A list_ge1(B, C)
| B list_eq1(A, C)
| B list_eq1_ge1(A, B, C)
| C list_eq1_ge1(A, B, C)
    {}

(* Upon finding an identifier, the lexer emits two tokens. The first token,
   [NAME], indicates that a name has been found; the second token, either [TYPE]
   or [VARIABLE], tells what kind of name this is. The classification is
   performed only when the second token is demanded by the parser. *)

typedef_name:
| i = NAME TYPE
    { i }

var_name:
| i = NAME VARIABLE
    { i }

(* [typedef_name_spec] must be declared before [general_identifier], so that the
   reduce/reduce conflict is solved the right way. *)

typedef_name_spec:
| typedef_name
    {}
```

```
general_identifier:
| i = typedef_name
| i = var_name
    { i }

save_context:
| (* empty *)
    { save_context () }

scoped(X):
| ctx = save_context x = X
    { restore_context ctx; x }

(* [declarator_varname] and [declarator_typedefname] are like [declarator]. In
   addition, they have the side effect of introducing the declared identifier as
   a new variable or typedef name in the current context. *)

declarator_varname:
| d = declarator
    { declare_varname (identifier d); d }

declarator_typedefname:
| d = declarator
    { declare_typedefname (identifier d); d }

(* Actual grammar *)

primary_expression:
| var_name
| CONSTANT
| STRING_LITERAL
| LPAREN expression RPAREN
| generic_selection
    {}

generic_selection:
| GENERIC LPAREN assignment_expression COMMA generic_assoc_list RPAREN
    {}

generic_assoc_list:
| generic_association
| generic_assoc_list COMMA generic_association
    {}

generic_association:
| type_name COLON assignment_expression
| DEFAULT COLON assignment_expression
    {}

postfix_expression:
| primary_expression
| postfix_expression LBRACK expression RBRACK
| postfix_expression LPAREN argument_expression_list? RPAREN
| postfix_expression DOT general_identifier
| postfix_expression PTR general_identifier
| postfix_expression INC
| postfix_expression DEC
| LPAREN type_name RPAREN LBRACE initializer_list COMMA? RBRACE
    {}

argument_expression_list:
| assignment_expression
| argument_expression_list COMMA assignment_expression
```

```
    {}

unary_expression:
| postfix_expression
| INC unary_expression
| DEC unary_expression
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF LPAREN type_name RPAREN
| ALIGNOF LPAREN type_name RPAREN
    {}

unary_operator:
| AND
| STAR
| PLUS
| MINUS
| TILDE
| BANG
    {}

cast_expression:
| unary_expression
| LPAREN type_name RPAREN cast_expression
    {}

multiplicative_operator:
  STAR | SLASH | PERCENT {}

multiplicative_expression:
| cast_expression
| multiplicative_expression multiplicative_operator cast_expression
    {}

additive_operator:
  PLUS | MINUS {}

additive_expression:
| multiplicative_expression
| additive_expression additive_operator multiplicative_expression
    {}

shift_operator:
  LEFT | RIGHT {}

shift_expression:
| additive_expression
| shift_expression shift_operator additive_expression
    {}

relational_operator:
  LT | GT | LEQ | GEQ {}

relational_expression:
| shift_expression
| relational_expression relational_operator shift_expression
    {}

equality_operator:
  EQEQ | NEQ {}

equality_expression:
| relational_expression
```

```
| equality_expression equality_operator relational_expression
    {}

and_expression:
| equality_expression
| and_expression AND equality_expression
    {}

exclusive_or_expression:
| and_expression
| exclusive_or_expression HAT and_expression
    {}

inclusive_or_expression:
| exclusive_or_expression
| inclusive_or_expression BAR exclusive_or_expression
    {}

logical_and_expression:
| inclusive_or_expression
| logical_and_expression ANDAND inclusive_or_expression
    {}

logical_or_expression:
| logical_and_expression
| logical_or_expression BARBAR logical_and_expression
    {}

conditional_expression:
| logical_or_expression
| logical_or_expression QUESTION expression COLON conditional_expression
    {}

assignment_expression:
| conditional_expression
| unary_expression assignment_operator assignment_expression
    {}

assignment_operator:
| EQ
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
    {}

expression:
| assignment_expression
| expression COMMA assignment_expression
    {}

constant_expression:
| conditional_expression
    {}

(* We separate type declarations, which contain an occurrence of [TYPEDEF], and
   normal declarations, which do not. This makes it possible to distinguish /in
```

```
   the grammar/ whether a declaration introduces typedef names or variables in
   the context. *)

declaration:
| declaration_specifiers         init_declarator_list(declarator_varname)?     SEMICOLON
| declaration_specifiers_typedef init_declarator_list(declarator_typedefname)? SEMICOLON
| static_assert_declaration
    {}

(* [declaration_specifier] corresponds to one declaration specifier in the C11
   standard, deprived of TYPEDEF and of type specifiers. *)

declaration_specifier:
| storage_class_specifier (* deprived of TYPEDEF *)
| type_qualifier
| function_specifier
| alignment_specifier
    {}

(* [declaration_specifiers] requires that at least one type specifier be
   present, and, if a unique type specifier is present, then no other type
   specifier be present. In other words, one should have either at least one
   nonunique type specifier, or exactly one unique type specifier.

   This is a weaker condition than 6.7.2 2. Encoding this condition in the
   grammar is necessary to disambiguate the example in 6.7.7 6:

     typedef signed int t;
     struct tag {
     unsigned t:4;
     const t:5;
     };

   The first field is a named t, while the second is unnamed of type t.

   [declaration_specifiers] forbids the [TYPEDEF] keyword. *)

declaration_specifiers:
| list_eq1(type_specifier_unique,    declaration_specifier)
| list_ge1(type_specifier_nonunique, declaration_specifier)
    {}

(* [declaration_specifiers_typedef] is analogous to [declaration_specifiers],
   but requires the [TYPEDEF] keyword to be present (exactly once). *)

declaration_specifiers_typedef:
| list_eq1_eq1(TYPEDEF, type_specifier_unique,    declaration_specifier)
| list_eq1_ge1(TYPEDEF, type_specifier_nonunique, declaration_specifier)
    {}

(* The parameter [declarator] in [init_declarator_list] and [init_declarator]
   is instantiated with [declarator_varname] or [declarator_typedefname]. *)

init_declarator_list(declarator):
| init_declarator(declarator)
| init_declarator_list(declarator) COMMA init_declarator(declarator)
    {}

init_declarator(declarator):
| declarator
| declarator EQ c_initializer
    {}
```

```
(* [storage_class_specifier] corresponds to storage-class-specifier in the
   C11 standard, deprived of [TYPEDEF] (which receives special treatment). *)

storage_class_specifier:
| EXTERN
| STATIC
| THREAD_LOCAL
| AUTO
| REGISTER
    {}

(* A type specifier which can appear together with other type specifiers. *)

type_specifier_nonunique:
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| COMPLEX
    {}

(* A type specifier which cannot appear together with other type specifiers. *)

type_specifier_unique:
| VOID
| BOOL
| atomic_type_specifier
| struct_or_union_specifier
| enum_specifier
| typedef_name_spec
    {}

struct_or_union_specifier:
| struct_or_union general_identifier? LBRACE struct_declaration_list RBRACE
| struct_or_union general_identifier
    {}

struct_or_union:
| STRUCT
| UNION
    {}

struct_declaration_list:
| struct_declaration
| struct_declaration_list struct_declaration
    {}

struct_declaration:
| specifier_qualifier_list struct_declarator_list? SEMICOLON
| static_assert_declaration
    {}

(* [specifier_qualifier_list] is as in the standard, except it also encodes the
   same constraint as [declaration_specifiers] (see above). *)

specifier_qualifier_list:
| list_eq1(type_specifier_unique,    type_qualifier)
| list_ge1(type_specifier_nonunique, type_qualifier)
    {}
```

```
struct_declarator_list:
| struct_declarator
| struct_declarator_list COMMA struct_declarator
    {}

struct_declarator:
| declarator
| declarator? COLON constant_expression
    {}

enum_specifier:
| ENUM general_identifier? LBRACE enumerator_list COMMA? RBRACE
| ENUM general_identifier
    {}

enumerator_list:
| enumerator
| enumerator_list COMMA enumerator
    {}

enumerator:
| i = enumeration_constant
| i = enumeration_constant EQ constant_expression
    { declare_varname i }

enumeration_constant:
| i = general_identifier
    { i }

atomic_type_specifier:
| ATOMIC LPAREN type_name RPAREN
| ATOMIC ATOMIC_LPAREN type_name RPAREN
    {}

type_qualifier:
| CONST
| RESTRICT
| VOLATILE
| ATOMIC
    {}

function_specifier:
| INLINE
| NORETURN
    {}

alignment_specifier:
| ALIGNAS LPAREN type_name RPAREN
| ALIGNAS LPAREN constant_expression RPAREN
    {}

declarator:
| d = direct_declarator
    { d }
| pointer d = direct_declarator
    { other_declarator d }

(* The occurrences of [save_context] inside [direct_declarator] and
   [direct_abstract_declarator] seem to serve no purpose. In fact, they are
   required in order to avoid certain conflicts. In other words, we must save
   the context at this point because the LR automaton is exploring multiple
   avenues in parallel and some of them do require saving the context. *)
```

```
direct_declarator:
| i = general_identifier
    { identifier_declarator i }
| LPAREN save_context d = declarator RPAREN
    { d }
| d = direct_declarator LBRACK type_qualifier_list? assignment_expression? RBRACK
| d = direct_declarator LBRACK STATIC type_qualifier_list? assignment_expression RBRACK
| d = direct_declarator LBRACK type_qualifier_list STATIC assignment_expression RBRACK
| d = direct_declarator LBRACK type_qualifier_list? STAR RBRACK
    { other_declarator d }
| d = direct_declarator LPAREN ctx = scoped(parameter_type_list) RPAREN
    { function_declarator d ctx }
| d = direct_declarator LPAREN save_context identifier_list? RPAREN
    { other_declarator d }

pointer:
| STAR type_qualifier_list? pointer?
    {}

type_qualifier_list:
| type_qualifier_list? type_qualifier
    {}

parameter_type_list:
| parameter_list option(COMMA ELLIPSIS {}) ctx = save_context
    { ctx }

parameter_list:
| parameter_declaration
| parameter_list COMMA parameter_declaration
    {}

parameter_declaration:
| declaration_specifiers declarator_varname
| declaration_specifiers abstract_declarator?
    {}

identifier_list:
| var_name
| identifier_list COMMA var_name
    {}

type_name:
| specifier_qualifier_list abstract_declarator?
    {}

abstract_declarator:
| pointer
| ioption(pointer) direct_abstract_declarator
    {}

direct_abstract_declarator:
| LPAREN save_context abstract_declarator RPAREN
| direct_abstract_declarator? LBRACK ioption(type_qualifier_list) assignment_expression? RBRACK
| direct_abstract_declarator? LBRACK STATIC type_qualifier_list? assignment_expression RBRACK
| direct_abstract_declarator? LBRACK type_qualifier_list STATIC assignment_expression RBRACK
| direct_abstract_declarator? LBRACK STAR RBRACK
| ioption(direct_abstract_declarator) LPAREN scoped(parameter_type_list)? RPAREN
    {}

c_initializer:
| assignment_expression
```

```
| LBRACE initializer_list COMMA? RBRACE
    {}

initializer_list:
| designation? c_initializer
| initializer_list COMMA designation? c_initializer
    {}

designation:
| designator_list EQ
    {}

designator_list:
| designator_list? designator
    {}

designator:
| LBRACK constant_expression RBRACK
| DOT general_identifier
    {}

static_assert_declaration:
| STATIC_ASSERT LPAREN constant_expression COMMA STRING_LITERAL RPAREN SEMICOLON
    {}

statement:
| labeled_statement
| scoped(compound_statement)
| expression_statement
| scoped(selection_statement)
| scoped(iteration_statement)
| jump_statement
    {}

labeled_statement:
| general_identifier COLON statement
| CASE constant_expression COLON statement
| DEFAULT COLON statement
    {}

compound_statement:
| LBRACE block_item_list? RBRACE
    {}

block_item_list:
| block_item_list? block_item
    {}

block_item:
| declaration
| statement
    {}

expression_statement:
| expression? SEMICOLON
    {}

selection_statement:
| IF LPAREN expression RPAREN scoped(statement) ELSE scoped(statement)
| IF LPAREN expression RPAREN scoped(statement) %prec below_ELSE
| SWITCH LPAREN expression RPAREN scoped(statement)
    {}
```

```
iteration_statement:
| WHILE LPAREN expression RPAREN scoped(statement)
| DO scoped(statement) WHILE LPAREN expression RPAREN SEMICOLON
| FOR LPAREN expression? SEMICOLON expression? SEMICOLON expression? RPAREN scoped(statement)
| FOR LPAREN declaration expression? SEMICOLON expression? RPAREN scoped(statement)
    {}

jump_statement:
| GOTO general_identifier SEMICOLON
| CONTINUE SEMICOLON
| BREAK SEMICOLON
| RETURN expression? SEMICOLON
    {}

translation_unit_file:
| external_declaration translation_unit_file
| external_declaration EOF
    {}

external_declaration:
| function_definition
| declaration
    {}

function_definition1:
| declaration_specifiers d = declarator_varname
    { let ctx = save_context () in
      reinstall_function_context d;
      ctx }

function_definition:
| ctx = function_definition1 declaration_list? compound_statement
    { restore_context ctx }

declaration_list:
| declaration
| declaration_list declaration
    {}
```