29. Resolving Typedefs in a Multipass C Compiler

W.M. McKeeman  
Digital Equipment Corporation  
110 Spitbrook Road  
Nashua, NH 03062

Abstract  
A C compiler must resolve the ambiguity between variables and typedef names during parsing. This requires the parser take into account extrasyntactic information. The information is typically held in the compiler symbol table. This paper outlines a solution where the compiler symbol table is not available. The solution is to build a minimal symbol table in the parser itself.

Introduction  
C fails to be LR(1) because of a conflict between identifier and typedef-name. The situation is illustrated by the following fragment:

```
static X(Y)
```

This text starts a declaration of Y if X is a typedef-name and Y is not. It starts a function prototype for X if Y is a typedef-name and X is not. It starts an old-style function-definition if neither X nor Y is a typedef-name. There are similar conflicts for casts and parenthesized expressions and function calls.

C programmers have little difficulty resolving these conflicts—the declared attributes of the names are sufficient. A syntax-driven parser, on the other hand, makes all decisions based on the immediate syntactic context. A previous type definition is not part of that context, thus something additional must be done.

The following description assumes the reader is familiar with compiler structure and parsing methods.

The typical C compiler scans source text, parses it, and builds a symbol table in a single pass. The typedef-name resolution takes place in an enhanced scanner which builds a typedef-name token instead of an identifier token when the scanner finds that token in the symbol table marked as a typedef-name. The parser is unaffected by this collusion between the scanner and symbol table [2]. Alternatively, the parser can interrogate the symbol table in those places where the conflict arises [3]. In this case the scanner is unaffected.

259
Where parsing is done on a separate pass from interpreting declarations, the symbol table is not available to either the scanner or the parser. A solution for unavailable symbol information is presented here in terms of Standard C as defined in the American National Standard X3.159-1989 \(^3\) [1]. The basis of the solution is a private symbol table built into the parser, capable only of resolving the \texttt{typedef-name/identifier} ambiguity.

It turns out that the parser needs to know only in a small number of places whether an \texttt{identifier} is a \texttt{typedef-name} [§A.2.2, 3.5.6] so as to apply the grammar rule

\begin{verbatim}
typedef-name:
  identifier
\end{verbatim}

The issue must be resolved in deciding between rules [§3.3.4]

\begin{verbatim}
cast-expression:
  unary-expression
  ( type-name ) cast-expression
\end{verbatim}

because a \texttt{unary-expression} might start with a parenthesized \texttt{identifier} and also between the two cases for each of the rules [§3.5]

\begin{verbatim}
declaration-specifiers:
  storage-class-specifier declaration-specifiers\_opt
  type-specifier declaration-specifiers\_opt
  type-qualifier declaration-specifiers\_opt
\end{verbatim}

because if the last \texttt{type-specifier} is an \texttt{identifier} it might instead be meant to be redeclared as belonging to the following declarator. The choice between \texttt{declarator} and \texttt{abstract-declarator} is formally ambiguous and therefore requires a special elaboration in the standard [§3.7.1]. There is a similar ambiguity between new and old-style \texttt{function-definition}.

Finally, the issue must be resolved between rules [§3.6.2]

\begin{verbatim}
compound-statement:
  declaration-list\_opt statement-list\_opt
\end{verbatim}

because of the ambiguity between function calls and declarations mentioned for \texttt{X(Y)} at the start of this section.

The parser accesses its private symbol table to resolve the local ambiguity rather than relying on the scanner. With this solution separation of concerns is cleaner. There is no question of the feasibility of the proposed solution since a complete symbol table could be built in the parser. The problem to be solved is keeping the parser's private symbol table simple, small, and efficient. The

\(^3\)Throughout this paper, the standard is implicitly referenced using the notation §x.y.z
idea has appeared in a C compiler [3]; it does not appear to be documented in the open literature.

Extra-syntactic decision-making in a parser requires ad hoc modifications to the parser. The modifications can be ugly if the parsing method is inflexible, as is the case if the parser is table-driven and parser sources are not available. The parsers in which these ideas have been tested have used recursive descent.

There are a number of simplifying assumptions in this presentation. It is assumed that the only objective of the parser is to report the shift/reduce sequence implied by the grammar. It is assumed that there is a lexical process that produces actions of type Token. It is assumed that the parser produces actions of type Rule. The time-merged sequence of Token and Rule actions is the shift/reduce sequence. These assumptions are neither theoretically nor practically limiting: any intermediate form can be efficiently constructed from the shift/reduce sequence. It is also straightforward to produce this same output from LALR-based parsers [4].

The requirements on the solution are that all syntactically correct C programs can be parsed and that all syntactically incorrect C programs can be diagnosed. The code added to resolve \texttt{typedef-name} need issue a diagnostic only when it cannot be assured that other diagnostic facilities will come into play. Specifically, the correct parse must be provided up to the point where a nonsyntactic error will be diagnosed, and some parse continuation must be provided so that later phases of the compiler will actually be invoked—the output of the parser must always reflect the parse for some correct program.

The Parser Symbol Table

Symbol tables for C are required to reflect a number of detailed requirements and constraints of the language definition. Where there is a common solution for the special parser symbol table and the general table, no details are given here based on the presumption that there are other sources of this information.

The actions for the private symbol table are interrogation, entry of a new \texttt{typedef-name}, obscuring a \texttt{typedef-name} with some other use of its name, entering and leaving a scope.

Scoping is complicated by the requirement that names in a \texttt{parameter-type-list} and \texttt{identifier-list} [§3.5.4] are in the scope associated with the \texttt{compound-statement} of the \texttt{function-definition} even though they are outside the opening '{' [§3.1.2.1]. The solution presented here has a second kind of scope entry which reopens a just-closed scope frame. This makes six functions altogether in the typedef-resolving symbol table mechanism.

Typedef names are in ordinary name space [§3.1.2.3]. If typedef is encountered in a \texttt{translation-unit}, its identifier becomes a \texttt{typedef-name} until either the current scope is finally left, or another declaration for the same name in an inner scope temporarily obscures it. There can be at most one \texttt{typedef-name} entered in any one scope. There can be multiple entries for other ordinary name
space uses of identifiers (because of the rules for linkage [§3.1.2.2] and old-style function definitions).

Only the grammar rules

\[
\text{direct-declarator:} \\
\text{identifier}
\]

\[
\text{enumeration-constant:} \\
\text{identifier} \\
\text{identifier} = \text{constant-expression}
\]

\[
\text{primary-expression:} \\
\text{identifier}
\]

can introduce names into ordinary name space [§3.5.4, 3.1.3.3, 3.3.1, 3.3.2]. These three cases can be treated one at a time.

A direct-declarator introduces a typedef-name only when it eventually participates in the grammar rule

\[
\text{declaration:} \\
\text{declaration-specifiers} \text{ init-declarator-list}_{\text{opt}} ;
\]

and reserved word typedef is among the declaration-specifiers. If, on the other hand, typedef was not in the declaration-specifiers, direct-declarator obscures any use for that identifier in enclosing scopes.

The rule above for init-declarator-list leads to

\[
\text{init-declarator:} \\
\text{declarator} \\
\text{declarator} = \text{initializer}
\]

Via the above nonterminal declarator, nonterminal direct-declarator participates in four other rules in C:

\[
\text{function-definition:} \\
\text{declaration-specifiers}_{\text{opt}} \text{ declarator declaration-list}_{\text{opt}} \text{ compound-statement}
\]

\[
\text{parameter-declaration:} \\
\text{declaration-specifiers} \text{ declarator}
\]

\[
\text{struct-declarator:} \\
\text{declarator} \\
\text{declarator}_{\text{opt}} : \text{constant-expression}
\]

\[
\text{direct-declarator:} \\
( \text{declarator} )
\]
In function-definition, typedef is syntactically allowed but never valid in either declaration-specifiers or declaration-list [§3.7.1]. This occurrence of declarator therefore cannot enter a typedef-name. And, since function-definition is always in the outermost scope, neither can it obscure a typedef-name.

In parameter-declaration, typedef is syntactically allowed but never valid [§3.5.4.3, 3.7.1]. A typedef-name cannot be entered, but one can be locally obscured by a parameter.

In struct-declarator ordinary name space names cannot be defined. Therefore, all parse symbol table activity must be suspended for struct-declarator.

Whatever is said about direct-declarator also applies to the parenthesized declarator. Thus it may behave as any of the four uses of declarator. This concludes what must be done for direct-declarator.

The situation for enumeration-constant is much simpler—whenever it appears it obscures any other use of its identifier in outer scopes. There is no need to check for multiple definition in the current scope—later phases of the compiler will do that.

The occurrence of a primary-expression can introduce a local ordinary name space object with external linkage when the primary-expression is immediately used in the rule

\[
\text{postfix-expression:} \\
\text{primary-expression ( argument-expression-listopt )}
\]

and the identifier is not previously declared. The effect is to obscure other local uses of that name from the point of implicit declaration. In fact the parser can ignore implicit declarations. They cannot obscure a typedef-name because there can be no previous declaration (of any kind) for that name.

**Implementation Details**

While it violates a constraint to have two declarations for the same name in one scope [§3.5], it is not necessary to check this constraint to parse correct programs. In essence, C is extended during parsing so that, instead of diagnosing multiple declarations, the last declaration wins. This trick always results in a decision on typedef-name and makes the diagnostic correct with respect to the nearest declaration. This decision only affects incorrect programs, thus there is no substantial impact on efficiency of parsing.

Six parser-specific routines need to be implemented:

```c
void ParseEnterScope(void);
void ParseExitScope(void);
void ParseReenterScope(void); /* undo Exit */
void ParseEnterTypedef(Token t);
void ParseObscureTypedef(Token t);
bool ParseIsTypedef(Token t);
```
Function ParseIsTypedef(t) is what is needed by the parser to distinguish between ordinary identifier and typedef-name. Everything else is just support for this function. ParseIsTypedef(t) is called just in the situations where both a typedef-name and object are syntactically acceptable and the parse depends on which is actually found.

Functions ParseEnterScope() and ParseExitScope() are usually paired. The scopes in C are associated with one of file-scope, function body, compound statement, or prototype. These two functions are called as each scope is entered, and as it is exited.

The call of function ParseReenterScope() is immediately preceded (in time) by ParseExitScope() and restores the parse symbol table to the state it had just before it did the exit [5]. The trick is to merely set a global delay flag in ParseExitScope() and not do the exit action. If the next call is ParseReenterScope(), it has no effect except for the clearing of the delay flag. All other actions check the delay flag and if it is set, actually do the exit action prior to doing their own functions. The re-enter situation occurs for function definitions. Function definition is detected syntactically when a file-scope declarator is immediately followed by other than one of ',', ';', or ' '. The effect is to include the formal parameters of the function in the scope of the function body.

ParseEnterTypedef(t) pushes t on the parse symbol table and marks it as a typedef-name. ParseObscureTypedef(t) pushes t on the parse symbol table and marks it as not a typedef-name. The obscuring action is effective until the end of scope occurs, causing ParseExitScope() to be called.

Invoking the typedef symbol table functions can be done in any of the conventional ways for complete symbol tables.

**Scope Management**

```c
void p ( ) function prototype
void p (int ) function prototype
void p (int a) function prototype
void (*p)(int ) pointer to function
void (*p)(int ) pointer to function
void (*p)(int a) pointer to function
void p (int a) { } function definition
void p ( ) { } function definition
void p (a ) int a; {} function definition
    { }
    t exit scope (final)
    reenter scope
    t exit scope (perhaps temp)
    t enter scope
```

Scopes are associated with parameter-type-list, identifier-list and compound-
statement. They interact [§3.1.2.1]. There are also some additional rules that allow the former status to be ignored [§3.5.6]. After a function header scope is closed (perhaps temporarily) scope is reopened for the function body since the parameters are in the same scope as the body. This forces ParseEnterScope() or ParseReenterScope() to be called before parsing compound statements.

Test Case

The syntax for C declarations permits writing declarations that are hard for programmers to decipher. Compilers have the same problem. The following C program compiles and runs. It contains multiple definitions of name p, each using a different combination of typedef-names, to be analyzed for compatibility. At the same time the type definitions are being obscured in local scopes because the same names are used for objects. J appears 8 times, once as the name of a parameter. I appears 10 times, 4 as the name of a parameter. The reader might want to predict the number of times ‘in p’ gets printed.

```c
#include <stdio.h>

typedef int I;    /* I is a typedef-name */
typedef I J(I());

I(i);             /* i is an int variable */

extern J p;       /* p takes an int function */
extern I p(J), p(J I), p(J J);
extern int q(register J), q(J register), r(const J J);

I p(J I) {        /* I becomes a local name */
    puts("in p");
    return i++<0 ? I((J*)p(I)) : -17;
}

main() {
    i = -2;
    printf("%d\n", p(p));
}
```

Error Behavior

This mechanism must behave correctly in the face of source program errors. The principal issue is avoiding incorrectly classifying an identifier (typedef-name or not). The situation arises when the user incorrectly declares an identifier twice in the same scope. Since the only report from this mechanism is the single bit
ParseIsTypedef(), there are exactly two problems: TRUE overriding FALSE and vice versa.

The consequence of an incorrect value is a parsing error later in the source text. The compiler cannot know which declaration was intended. It is barely acceptable to report a parsing error at some point well beyond the declaration that caused the problem. The mechanism outlined here could be extended to detect and report redeclarations that could affect the value of ParseIsTypedef() without affecting the interface defined above.

Acknowledgements

The central idea of this paper was brought to the attention of the author by John Hamby in relation to the DEC C implementation. Since then many of my co-workers have contributed ideas and suggestions. The implementation described here is from a prototype incremental compiler for Standard C.

References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.