A Research C# Compiler

DAVID R. HANSON AND TODD A. PROEBSTING
Microsoft Research, 1 Microsoft Way, Redmond, WA 98052 USA
drh@microsoft.com toddpro@microsoft.com

Summary

C# is the new flagship language in the Microsoft .NET platform. C# is an attractive vehicle for language design research not only because it shares many characteristics with Java, the current language of choice for such research, but also because it's likely to see wide use. Language research needs a large investment in infrastructure, even for relatively small studies. This paper describes a new C# compiler designed specifically to provide that infrastructure. The overall design is deceptively simple. The parser is generated automatically from a possibly ambiguous grammar, accepts C# source, perhaps with new features, and produces an abstract syntax tree, or AST. Subsequent phases—dubbed visitors—traverse the AST, perhaps modifying it, annotating it or emitting output, and pass it along to the next visitor. Visitors are specified entirely at compilation time and are loaded dynamically as needed. There is no fixed set of visitors, and visitors are completely unconstrained. Some visitors perform traditional compilation phases, but the more interesting ones do code analysis, emit non-traditional data such as XML, and display data structures for debugging. Indeed, most usage to date has been for tools, not for language design experiments. Such experiments use source-to-source transformations or extend existing visitors to handle new language features. These approaches are illustrated by adding a statement that switches on a type instead of a value, which can be implemented in a few hundred lines. The compiler also exemplifies the value of dynamic loading and of type reflection.

Keywords: Compiler architecture, abstract syntax trees, .NET, C# programming language, visitor pattern, object-oriented programming

Introduction

C# [1, 2] is the preeminent programming language in the Microsoft .NET platform. The .NET platform includes tools, technologies, and methodologies for writing internet applications [3]. It includes programming languages, tools that support XML web services, and new infrastructure for writing HTML pages and Windows applications. At its core are a new virtual machine and an extensive runtime environment. Compilers for C# and other .NET languages generate code for this virtual machine, called the .NET Common Intermediate Language or MSIL for short. MSIL provides a low-level, executable, type-safe program representation that can be verified before execution, much in the same way as the Java VM [4] provides a verifiable representation for Java programs. It is, however, designed specifically to support multiple languages on modern processors.

C# is a high-level, type-safe, object-oriented programming language. It has many of the same features as Java, but it also has language-level support for properties, events, attributes, and interoperability with other languages. C# also has operator overloading, enumerations, value types, and language constructs for iterating over collections.

Java is often the language of choice for experimental programming language research. Research focuses either on the Java VM or on changes to Java itself. Adding generics to Java is an example of the latter focus [5]. C# is an attractive platform for language research because it is in the same language 'family' as Java and because it is likely to become used widely. Microsoft's C# is available on Windows and on FreeBSD as part of the Rotor [6] distribution, and the Mono Project [7] is developing a C# compiler for Linux, so language researchers seeking wide impact for their results may want to use C#. Also, C# will undoubtedly evolve over time and is thus open to future additions, so language research results might

find their way into wide use. Again, adding generics to C# and to MSIL [8] is an example of the possibilities.

Programming language research on new language features and their implementation requires a significant investment in a suitable compilation infrastructure. At the minimum, such work needs a compiler that accepts the full language, is easily changed, and can compile significant programs quickly. Besides the usual complexity inherent in all non-trivial compilers, there is a natural tension between flexibility and performance, both of the compiler itself and of the generated code. Wonderfully flexible compilers that accept only a subset of the language, are too slow, or produce incorrect or very inefficient code don't get used. The same fate befalls compilers that generate highly optimized code but that are too complex to understand and modify easily.

This note describes a new C# compiler designed specifically for use in language research. The goal of this compiler, named lcsc (for local C sharp compiler), is to facilitate experiments with a wide range of language-level features that require new syntax and semantics. Examples include simple additions like a Modula-3 TYPECASE statement [9] to more exotic features like Icon iterators [10], program histories [11], and dynamically scoped variables [12]. Lcsc is not designed for code generation research, per se, because it has no support specifically for native code generation, but there's nothing in its design that precludes such work. By default, it emits MSIL.

The design of lcsc is particularly simple and it is easy to modify and to extend, and it's fast enough for experimental purposes, albeit perhaps an order of magnitude slower than the production C# compiler in .NET. However, implementation languages account for some of the speed difference: lcsc is written in C# and makes heavy use of automatic memory management and the .NET compiler is written in C++. Early experience with lcsc confirms that it is easy to extend and to modify, particularly if the new features can be modeled in C# itself.

Perhaps surprisingly, most of the use of lcsc to date is for C# language tools and program analysis, and not for language research. Of course, there are more tool developers than language researchers, but this usage was not anticipated and was not factored into the design. The end result is, however, that lcsc's design, which is based on abstract-syntax trees, is a good infrastructure for language-level tools. Using a similar approach for other languages might harvest similar benefits.

Design

From 30,000 feet, lcsc's design is dead simple: The parser reads the C# source input and builds an abstract syntax tree [13], or AST, and subsequent phases, such as type checking and code generation, traverse the AST, perhaps leaving annotations on its nodes. The details are, of course, more involved, but the basic design dictates little beyond the ASTs.

The design is intentionally lean: there is no explicit support for parsing fragments of the grammar, incremental compilation, etc., and there is no GUI interactive development or support for using one. Some of these features could be provided by adding new start symbols to the grammar and specialized AST traversals, and the Eclipse Platform [14] may be an appropriate development environment infrastructure.

To date, lcsc has been used only for experiments and tools involving C#. While its components are specific to C#, they could, in principle, be revised for other, similar languages.

Parsing

The parser is generated automatically from a C# BNF grammar that is nearly identical to the grammar given in the language specification [1]. This grammar is ambiguous, and the home-grown parser generator accepts ambiguous grammars. The generated parser is a generalized LR parser [15, 16]; for inputs that have ambiguous parses, the parser builds multiple parse trees. The number of parse trees is theoretically unbounded, but for practical programming language grammars, there are few alternatives, which can usually be resolved by inspecting the alternative subtrees, as described below.

The parser generator also emits code to build an AST bottom-up from the parse trees. Code fragments that return AST nodes are associated with each production in the grammar, as exemplified by the productions for the if-statement:

if-statement **`if (`** *boolean-expression* **`)`** *embedded-statement*
`statement new if_statement(a3,a5,null)`

if-statement **`if (`** *boolean-expression* **`)`** *embedded-statement* **`else`** *embedded-statement*
`statement new if_statement(a3,a5,a7)`

In the productions, nonterminals appear in *italics* and terminals in a **`typewriter`** font. The code fragment—the 'action' in grammar parlance—appears on the far right in the lines following the productions. The occurrences of **`statement`** identify the abstract type of the AST nodes returned by the **`new`** expressions. The **`a3`**, **`a5`**, and **`a7`** refer to the AST values returned by the corresponding grammar symbols in the rule in the order in which they occur, e.g., *boolean-expression*, and the two occurrences of *embedded-statement*.

Ambiguities are resolved during AST construction by examining alternative parse trees. If a parse tree node has more than one alternative, the set of alternatives is passed to the user-defined method **`resolve`**, which inspects the alternatives and perhaps the context in which they occur, chooses one, and returns the appropriate AST node. While this ad hoc approach does require some iteration to discover and handle all the ambiguities, its cost is too small to warrant more sophisticated mechanisms [17]. Per-node resolution code is short, usually less than a dozen lines; the C# if-statement takes 17 lines and is one of the most complicated. The entire body of **`resolve`** for C# is only 114 lines.

A novelty of the parser generator is that it reads the grammar specification from Excel spreadsheets, which eliminates much of the code that parses the grammar specification in other generators, provides some error checking, and serves as a simply GUI to the grammar. Nonterminals, rules, types, and actions each appear in separate columns, one production per row. The type column, which holds **`statement`** in the if-statement example above, is computed using Excel formulas. A separate 'sheet' in the spreadsheet lists the AST type for each nonterminal, and formulas are used to propagate those types into the third column of the 476 productions in the grammar sheet. Separate sheets are also used to list keywords and operator tokens.

The types sheet also gives the default AST expression for each nonterminal, which is used when an optional occurrence of a nonterminal appears in a rule. For example,

block **`{`** *statement-list*$_{opt}$ **`}`** **`statement new block_statement(a2)`**

specifies that a *block* is an optional *statement-list* enclosed in braces. If *statement-list* is omitted, its default from the types sheet (a zero-length **`statementList`**) is supplied as the value of **`a2`**. Incidentally, optional elements are written exactly as this example shows, with a subscript 'opt'.

Using an Excel spreadsheet makes the grammar easy to augment. Additional columns can be added for, say, comments or annotations for other tools, without modifying the parser generator, which inspects only specific columns. Finally, the parser generator accepts one or more spreadsheets, so language extensions can be specified in a separate spreadsheet without modifying the core C# grammar.

There are, of course, disadvantages to using Excel. Even viewing the grammar requires running Excel. While it's easy to extract the spreadsheet data as plain text in several formats, some are idiosyncratic. The 330-line parser-generator module that extracts the grammar is thus Excel-specific and cannot be reused easily. Packaging the grammar and its associated data as XML is an attractive alternative, in part because XML is ubiquitous and platform independent, and there are many XML-related tools. A good XML editor could, for example, provide a GUI for grammar and structured editing much as Excel does.

Semantics

The parser returns a single AST for each input source file. For multiple source files, these ASTs are stitched together into a single AST for subsequent processing. Semantic processing is specified entirely by command-line options, which specify AST 'visitors' and the order in which they are invoked. Visitors are packaged as separate classes in .NET dynamically linked libraries, DLLs for short. For example,

```
lcsc –visitor:XML foo.cs
```

parses **foo.cs** and passes the resulting AST to the **XML** class in **XML.dll**. By convention, the AST is passed to the static method **visit**; there are provisions for passing additional string arguments, when necessary, and for specifying the DLL file name.

Visitor classes [18] traverse the AST, perhaps annotating it, transforming it, or emitting output, and return the AST for subsequent passes. For example, traditional compilation is accomplished by

```
lcsc –visitor:bind –visitor:typecheck
            –visitor:rewrite –visitor:ilgen foo.cs
```

which parses **foo.cs**, binds names, type checks, rewrites some ASTs, and generates and assembles MSIL code. The bind visitor hangs symbols on the AST nodes and collects symbols into symbol tables, and typecheck uses these data to compute the types for expressions and drops types on those nodes. Rewrite modifies some ASTs to simplify code generation, e.g., it turns post-increment expressions into additions and assignments. Finally, most of ilgen simply emits MSIL code, but even ilgen annotates some nodes: it assigns internal label numbers to loop and iteration statements.

Dynamic loading permits visitors to access data structures built by other visitors directly. Without dynamic loading, the AST and its associated data structures would have to serialized and passed via files or pipes to subsequent visitors. This approach has been used often by us and others (see, for example, Ref. [19]), but it's slower, can induce restrictions on data structures, and may be prone to serialization errors. From a packaging viewpoint, dynamic loading is much simpler.

There are 164 AST node classes, 16 of which are abstract classes. For example, **statement** and **expression** are abstract classes. The ASTs describe the source-level structure of the input program; the class for an if-statement is typical:

```
public class if_statement: statement {
   public if_statement(expression expr, statement thenpart,
         statement elsepart) {
      this.expr = expr;
      this.thenpart = thenpart;
      this.elsepart = elsepart;
   }
   public expression expr;
   public statement thenpart;
   [MayBeNull] public statement elsepart;
   public override void visit(ASTVisitor prefix, ASTVisitor postfix) {…}
}
```

An **if_statement** node is created by a **new** expression, which calls the constructor to fill in the fields. The **visit** method is described below. Nodes with multiple children use type-safe lists that can be indexed like arrays, e.g., the –**List** types in

```
public class class_declaration: declaration {
   public class_declaration(IList attrs, IList mods, InputElement id,
              IList bases, IList body) { … }
   public attribute_sectionList attrs;
   public typeList bases;
   public declarationList body;
```

```
    public InputElement id;
    …
}
```

InputElements are tokens and identify the token type, the specific token instance, and its location in the source code.

Much of the code in a visitor is boilerplate traversal code. Included with lcsc is mkvisitor, a tool that uses type reflection to generate a complete visitor that can be subsequently edited to suit its specific task. For example,

```
mkvisitor -args "SymbolTable bindings" >visitor.cs
```

generates the following C# code for the **if_statement** class shown above.

```
void if_statement(if_statement ast, SymbolTable bindings) {
    expression(ast.expr, bindings);
    statement(ast.thenpart, bindings);
    if (ast.elsepart != null)
        statement(ast.elsepart, bindings);
}
```

The **[MayBeNull]** annotation on the field **elsepart** in the declaration of **if_statement** above is a C# attribute, and it indicates that the field may be null. These kinds of attributes are used in mkvisitor and other program generation tools to emit appropriate guards to avoid traversing valid null ASTs. Typical visitors run around 2000 lines of C# of which about 830 are generated by mkvisitor.

Mkvisitor uses type reflection to discover the AST vocabulary, but this approach is not, of course, the only one. Mkvisitor could read the AST source code, or it could read some other specification of the ASTs. But using reflection is perhaps the easiest of these approaches and it fits well into the lcsc build process: Whenever the AST source code is revised and recompiled, mkvisitor gets the revised definitions automatically.

While the full visitor machinery is used for most compilation passes, there is a simpler mechanism that is useful for one-shot applications. Each AST node includes a **visit** method that implements both a prefix and postfix walk. For instance, the **visit** method in **if_statement** is

```
public override void visit(ASTVisitor prefix, ASTVisitor postfix) {
    prefix(this);
    expr.visit(prefix, postfix);
    thenpart.visit(prefix, postfix);
    if (elsepart != null)
        elsepart.visit(prefix, postfix);
    postfix(this);
}
```

The arguments **prefix** and **postfix** are instances of **ASTVisitor** *delegates*, which are essentially type-safe function pointers. These delegates are particularly useful for tools that need to examine only parts of the AST or that look for specific patterns. For example, C# statements of the form

```
if (…) throw …
```

mimic assertions. The expression

```
ast.visit(new ASTVisitor(doit), new ASTVisitor(donothing));
```

finds occurrences of this pattern in an AST rooted at **ast**, where **doit** and **donothing** are defined as follows.

```
public static void doit(AST ast) {
    if (ast is if_statement
    && ((if_statement)ast).thenpart is throw_statement
```

```
      && ((if_statement)ast).elsepart == null)
         Console.WriteLine("{0}: Possible assertion", ast.begin);
   }

   public static void donothing(AST ast) { }
```

The **begin** and **end** fields in an AST give the beginning and ending locations in the source code spanned by the AST.

Applications

Finding code fragments that look like assertions typifies the use of lcsc to find patterns in C#. Elaborations of this usage are used for code-auditing tools, for example. Pattern matching on an AST instead of text makes it easy to consider context and to fine tune matches to avoid voluminous output.

XML is an increasing popular way to exchange data, and there are numerous tools available for editing and viewing XML. The XML visitor emits an AST as XML for consumption by XML-based tools or external compilation tools that accept XML. C# includes extensive support for reflection, which makes it possible to discover class information during execution. The 70-line XML visitor uses reflection to discover the details of the AST classes necessary to emit XML and is thus automatically upward compatible with future additions to the AST vocabulary.

The visitor architecture helps write metaprogramming tools, e.g., tools that write programs. The .NET platform includes a tree-based API for creating programs, typically those used in web services applications. This API defines a language-independent code document object model, also known as CodeDOM. It's possible, for example, to build a CodeDOM tree and pass it to C#, Visual Basic, or to any language that offers a 'code provider' interface. A common approach to writing CodeDOM applications is to write, say, C# source code and translate it by hand into the API calls that build the CodeDOM tree for that C# code. The lcsc codedom visitor automates this process: Given a C# program *P*, it emits a C# program that, when executed, builds the CodeDOM tree for *P*.

The source visitor is similarly useful: It emits C# source code from an AST. When coupled with visitors that modify the AST, source provides a source-to-source transformation facility. As detailed in the next section, source is useful for C# language extensions that can be modeled in C# itself. It's also useful for writing code reorganization tools. A simple example is sortmembers, which alphabetizes the fields and methods in all of the classes in its input C# program. For instance, the command

```
lcsc –visitor:sortmembers –visitor:source old.cs >new.cs
```

sorts the class members in **old.cs** and writes the C# output to **new.cs**. The sortmember visitor simply rearranges the declaration subtrees in each class declaration and lets source emit the now-sorted results.

A limitation of the source visitor is that it cannot reproduce the C# input exactly, because the ASTs do not include comments and white space. So, source can't be used as a pretty printer for example, or for applications that demand full comment fidelity, e.g., systems that generate code documentation from structured comments. Comments and white space could, in principal, be associated with AST nodes.

Incidentally, source turns out to document the ASTs nicely, because, for each node type, it shows how to emit the corresponding C# source. Writers of new visitors often start with a copy of source and edit to suit their own purposes. At any point, the output shows exactly how much progress has been made—C# source appears for those node types whose visitor methods remain to be edited.

Visitors are also used for diagnostic purposes, e.g., to help debug other visitors. The display visitor renders its input AST as HTML and launches the web browser to display the result. Using reflection, display lists the fields in each class instance with hyperlinks to those fields that hold references to other classes. Figure 1 shows the **class_declaration** AST node for the following prototypical 'hello world' C# program.

```
class Hello {
   public static void Main() {
      System.Console.WriteLine("Hello, world");
   }
}
```

The hyperlinks, shown underlined, make it easy to traverse the data structure by clicking on the links. Display handles lists as well as AST types, and it omits empty lists, which occur frequently.

```
class_declaration#22:
        attrs=attribute_sectionList#33 (empty)
        bases=typeList#35 (empty)
        body=declarationList#37
        id=InputElement{coord=hello.cs(3,7),str=Hello,tag=identifier}
        sym=ClassType#43
        mods=InputElementList#45 (empty)
        parent=compilation_unit#12
```

Figure 1. Sample display output.

Because display uses reflection, it can display *any* class type; for example, the **`sym`** field in Figure 1 refers to a **`ClassType`**, which is a class type used to represent C# types. This capability helps debug visitors that annotate the AST. For instance, the command

```
lcsc –visitor:bind –visitor:display
            –visitor:typecheck –visitor:display foo.cs
```

builds the AST for **`foo.cs`**, runs bind, displays the AST with bind's annotations (which includes the **`sym`** field shown in Figure 1), runs typecheck, then displays the AST again with typecheck's additions.

For even medium size C# programs, display generates a large amount of HTML. While navigating the AST is straightforward, it is often difficult to correlate a specific AST node with its associated source text. The browser visitor is a more ambitious variant of display that provides a more natural 'navigation' mechanism. Browser displays the C# source text in one window and AST nodes in another one. The AST nodes are rendered as done by display. Highlighting a fragment of source text causes the root of smallest enclosing AST subtree to appear in the AST window. Just that subtree can be explored by clicking the field values. Another variant of browser provides a similar mechanism for exploring the generated MSIL code: Highlighting a source code fragment displays the corresponding MSIL code in another window.

Adding Language Features

Implementing new language features has much in common with writing code analysis tools, and lcsc's visitor-based design facilitates such additions. For example, the source visitor often does half the work for new features that can be modeled in vanilla C#. More ambitious features can be implemented by building on the existing visitors, either by calling them explicitly or by subclassing them.

Adding a typeswitch statement provides an example of both approaches. The typeswitch statement is a case statement that branches on the type of an expression instead of on its value. For example,

```
typeswitch (o) {
   case Int32  (x): Console.WriteLine(x); break;
   case Symbol (s): Symbols.Add(s); break;
   case Segment: popSegment(); break;
   default: throw new ArgumentException();
}
```

switches on the type of **o**. The typeswitch cases can also introduce locals of the case label type, as illustrated by the **Int32** and **Symbol** cases, which introduce locals **x** and **s**. Figure 2 gives the grammar for the typeswitch statement, which is based on the Modula-3 TYPECASE statement [9].

typeswitch-statement	
typeswitch (*expression* **)** *typeswitch-block*	**new typeswitch_statement(a3,a5)**
typeswitch-block	
{ *typeswitch-sections*$_{opt}$ **}**	**a2**
typeswitch-sections	
typeswitch-section	**typeswitch_sectionList.New(a1)**
typeswitch-sections typeswitch-section	**List.Cons(a1,a2)**
typeswitch-section	
case *type* **(** *identifier* **) :** *statement-list*	**new typeswitch_section(a2,a4,a7)**
typeswitch-labels statement-list	**new typeswitch_section(a1,a2)**
default : *statement-list*	**new typeswitch_section(a3)**
typeswitch-labels	
typeswitch-label	**switch_labelList.New(a1)**
typeswitch-labels typeswitch-label	**List.Cons(a1,a2)**
typeswitch-label	
case *type* **:**	**new typeswitch_label(a2)**

Figure 2. Typeswitch syntax.

Typeswitch can be implemented in C# using if and goto statements, local variables, and typecasts to convert the typeswitch expression to the types indicated. For example, the typeswitch fragment above could be translated mechanically as follows.

```
{
   object yy_1 = o;
   if (yy_1 is Int32) {
      Int32 x = (Int32)yy_1;
      Console.WriteLine(x);
      goto yy_1_end;
   }
   if (yy_1 is Symbol) {
      Symbol s = (Symbol)yy_1;
      Symbols.Add(s);
      goto yy_1_end;
   }
   if (yy_1 is Segment) {
      popSegment();
      goto yy_1_end;
   }
   throw new ArgumentException();
yy_1_end: ;
}
```

A source-to-source transformation is thus one way to implement typeswitch. There are several alternatives. One approach is to write a visitor that transforms the typeswitch subtrees into the corresponding block and if statement trees as suggested by the output above, and use the source visitor to emit the trans-

formed AST. A simpler approach, however, is to extend the source visitor to accept typeswitch subtrees and emit the if statement implementation directly.

In either case, the first step is to define the typeswitch grammar, which appears in Figure 2. This grammar resides in its own Excel spreadsheet and is passed to the parser generator along with the original C# grammar. The second step is to define the AST types needed to represent typeswitch statements and to add the appropriate types and actions to the grammar spreadsheet. There are three new types, and the actions are shown to the right of the productions in Figure 2. The type column is omitted.

The final step is to add methods to the source visitor to traverse the three typeswitch AST types, emitting C# code as suggested above. It is not necessary to change the source visitor class itself; it can simply be *extended* by a derived class, typeswitch_source, that implements three new methods. It also overrides the methods in source for break statements and default labels, because these constructs can now appear in both switch and typeswitch statements. These steps take a total of only 153 lines:

19 lines	typeswitch grammar
63	typeswitch AST nodes
71	extend source
153	Total

More exotic language features and those that cannot be modeled in C# require more implementation effort. The effort can sometimes be reduced by transforming a part of the feature into existing C# AST nodes to make use of existing visitor code. But many additions require all of the typical compilation steps, including binding, typechecking, and code generation. Typeswitch again provides an example.

The first two steps are the same as in the source-to-source approach: specify the grammar and define the typeswitch-specific AST node types.

Each of the four traditional compilation visitors must be extended to traverse the three typeswitch nodes. Again, this extension can be done by subclassing, and defining new methods and overriding existing methods. The complete implementation takes 333 lines:

19 lines	typeswitch grammar
63	typeswitch AST nodes
100	extend bind
37	extend typecheck
39	extend rewrite
65	extend ilgen
333	Total

Bind requires the most code because it makes three passes over the AST and defines a class for each pass. Code generation is performed by rewrite and ilgen, which could be combined into a single visitor.

Note that in both approaches the additional code required is approximately proportional to the ‘size’ of the typeswitch addition. For many additions, lcsc provides scalable extensibility [20]; that is, the addition requires effort proportional to its syntactical and semantic size relative to the base C# language.

Of course, lcsc’s unconstrained approach does not guarantee scalable extensibility, so other language features could take much more effort. Substantial features typically add 1000s of lines instead of 100s. For example, an implementation of futures for C# took about 1000 lines. But even a complete visitor is much less effort than implementing a complete compiler or modifying a production compiler, and the visitor-based design enforces a modularity that helps avoid errors.

Lcsc’s design is not a solution to the extension problem, viz., the code-modification conflicts that can arise when adding both language features and tools. As the typeswitch example shows, inheritance helps when an addition requires new AST classes, but more complicated additions may require more complex subclasses. Most lcsc visitors annotate the AST nodes with analysis values, e.g., types, so new visitors

often require new fields in some AST classes. Luckily, additions do not invalidate previously compiled visitors, but the additions can lead to confusion. Factory classes [18] would help, but not solve the problem completely, because they would need to be modified when new AST classes were added. Another approach, used in SableCC [21], is to store all visitor-specific analysis data in the visitor class. Finally, there's no easy way to compose several visitors into a single visitor beyond writing a visitor class whose sole purpose is to invoke other visitors. To date, this omission has caused no problems, but as the visitor vocabulary grows, visitor combinators [22] may be prove useful.

Discussion

There are many tools for writing compilers. Parser generators, exemplified by Yacc, and code-generator generators, such as iburg [23], are now used routinely to help build the 'ends' of compilers. Lcsc includes a parser generator, but its design focuses most on a compiler's 'middle' phases, because those phases are often the most intricate, and on compiler-related tools. SableCC [21] and Polyglot [20] are recent compiler infrastructure systems that are closest in spirit and intention to lcsc. Both use Java and use visitors for semantic passes over ASTs. Given an annotated grammar, SableCC generates a lexer, parser, AST definitions, and AST traversal classes. Semantic analysis and code-generation passes are written by extending these traversal classes. Polyglot uses the JFlex [24] lexer generator and the CUP [25] parser generator. Polyglot emits Java code and is thus intended for experiments on extensions to Java. It comes with an AST definition and semantic passes for Java, which the experimental extends to accommodate the language extension under investigation. Interestingly, Polyglot's passes rewrite ASTs applicatively: passes do not modify their input trees.

At just under 14,000 lines, lcsc is a relatively small compiler, but lcsc lacks some of the components in production compilers, such as an optimization phase, and it has no interactive development environment. The visitor-based design makes it easy to see the relative sizes of the various passes. Table I gives the line counts for the core of lcsc, for the visitors, and for the program generators. Everything is written in C# except the parser generator, which is written in Icon [10].

Table I. Source line counts.

Line Count	*Item*
6779 lines	lexer, parser, data structures (required)
6942	traditional semantic passes (required for 'normal' compiler)
1972	bind.cs
2628	typecheck.cs
2342	rewrite.cs, ilgen.cs
3908	utility passes (optional)
1023	browser
1644	codedom
1015	source
156	sortmembers
70	xml
1594	program generation tools
966	parser generator (written in Icon)
329	excel2gram.cs
198	mkvisitor.cs
101	list generator

Writing visitors does require substantial understanding of the compiler's innards. Simple visitors require understanding only the AST vocabulary, which is easily digested because it follows the language closely. More complex visitors and visitors that subclass existing visitors require understanding the symbol-table and type data structures and some of the existing visitors, such as bind. Fortunately, these data

structures are straightforward, and most visitors can be understood in isolation, because the design induces a natural decoupling. In any case, getting a grip on even several visitors is much less onerous than digesting an entire monolithic compiler.

The visitor approach is incredibly flexible, and the design encourages writing several simple visitors that each do one task well to accomplish an overall goal rather than a single, complex visitor. And simple visitors are easier to debug. All visitors written to date pass along an AST, possibly modified or annotated. While not immediately obvious, there's nothing in the design that dictates this convention. A set of visitors could collaborate to pass along an instance of *any* object type. So, for example, a visitor could accept an AST, build a completely different representation of the program, such as a flow graph of basic blocks, and pass this representation along to its successors.

This flexibility does come at a price, however. The value passed to visitors is an object, so visitors must downcast this object to the expected type, e.g., AST. Thus, some type errors are caught only at runtime. Within a visitor's code, however, all types are known at compile-time and are checked statically.

C# and the .NET platform provide excellent support for dynamic loading and for reflection, and lcsc and its build tools make heavy use of them. The value of these features is easy to underestimate a priori. Once they seep into an application's design, however, they become invaluable. Reflection simplified several visitors and made mkvisitor, source, and browser possible. The visitor approach would not have been nearly as useful, or even possible, without dynamic loading. In an environment that supports only static linking, adding new visitors would require at least relinking and probably some recompilation. For example, multiple code generators are linked into a single executable for the retargetable C compiler, lcc [26], precisely because dynamic loading as supported in .NET and Java was not then universally available. If lcc were rewritten today, it would use dynamic loading to access its code generators.

References

1. ECMA International, 'C# Language Specification', Standard ECMA-334, Geneva, Dec. 2001 www.ecma.ch/ecma1/STAND/ecma-334.htm.
2. E. Gunnerson, *A Programmer's Introduction to C#* (second edition), Apress, Berkeley, CA, 2001.
3. D. S. Platt, *Introducing Microsoft .NET*, second edition, Microsoft Press, Redmond, WA, 2002.
4. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, second edition, Addison-Wesley, Palo Alto, CA, 1999.
5. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, 'Making the Future Safe for the Past: Adding Genericity to the Java Programming Language', *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Application*, Vancouver, Oct. 1998, pp. 183–200. http://doi.acm.org/10.1145/286936.286957.
6. D. Stutz, T. Neward, and G. Shilling, *Shared Source CLI Essentials*, O'Reilly & Associates, Sebastopol, CA, 2003.
7. The Mono Project, http://www.go-mono.com, 2003.
8. A. Kennedy and D. Syme, 'Design and Implementation of Generics for the .NET Common Language Runtime', *Proceedings of the SIGPLAN'01 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001, pp. 1–12. http://doi.acm.org/10.1145/378795.378797.
9. G. Nelson, *Systems Programming with Modula-3*, Prentice-Hall, Englewood Cliffs, NJ, 1991. www.research.compaq.com/SRC/m3defn/html/m3.html.
10. R. E. Griswold and M. T. Griswold, *The Icon Programming Language* (third edition), Peer-to-Peer Communications, San Jose, CA, 1997. www.cs.arizona.edu/icon/.
11. T. A, Proebsting and B. G. Zorn, 'Tangible program histories', Technical Report MSR-TR-2000-54, Microsoft Research, Redmond, WA, May 2000. ftp://ftp.research.microsoft.com/pub/tr/tr-2000-54.ps.
12. D. R. Hanson and T. A. Proebsting, 'Dynamic Variables'. *Proceedings of the SIGPLAN'01 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001, pp. 264–273. http://doi.acm.org/10.1145/378795.378857.

13. A, W. Appel, *Modern Compiler Implementation in Java*, second edition, Cambridge University Press, New York, 2002.
14. Object Technology International, Inc., *Eclipse Platform Technical Overview*, Feb. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.
15. E. Visser, *Syntax Definition for Language Prototyping*, PhD Dissertation, Univ. Amsterdam, 1997. http://www.cs.uu.nl/people/visser/ftp/Vis97.ps.gz.
16. E. Visser, 'Scannerless Generalized-LR Parsing', Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. http://www.cs.uu.nl/people/visser/ftp/P9707.ps.
17. M. van den Brand, J. Scheerder, J. Vinju and E. Visser, 'Disambiguation Filters for Scannerless Generalized LR Parsers', *Compiler Construction (CC'02)*, Grenoble, April 2002, pp. 143–158.
18. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
19. D. R. Hanson, 'Early Experience with ASDL in lcc', *Software—Practice & Experience*, **29** (5), 417–435 (1999).
20. N. Nystrom, M. R. Clarkson, and A. C. Myers, 'Polyglot: An Extensible Compiler Framework for Java'. *Proceedings of the 12th International Conference on Compiler Construction*, Warsaw, Poland, April 2003, pp. 138–152. Published as Lecture Notes in Computer Science, vol. 2622, Springer Verlag, Heidelberg, Jan. 2003. http://www.cs.cornell.edu/andru/papers/polyglot.pdf.
21. E. M. Gagnon and L. J. Hendren, 'SableCC, an Object-Oriented Compiler Framework', *Proceedings of TOOLS 26: Technology of Object-Oriented Languages*, Santa Barbara, CA, Aug. 1998, pp. 140–154. http://www.sable.mcgill.ca/publications/papers/1998-1/sable-paper-1998-1.ps.
22. J. Visser, 'Visitor Combination and Traversal Control', *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Application*, Tampa Bay, FL, Oct. 2001, pp. 270–282. http://doi.acm.org/10.1145/504282.504302.
23. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. 'Engineering a Simple, Efficient Code-generator Generator', *ACM Letters on Programming Languages and Systems*, **1** (3), 213–226 (1992). http://doi.acm.org/10.1145/151640.151642.
24. G. Klein, *JFlex—The Fast Scanner Generator for Java*, http://jflex.de/, 2001.
25. S. E. Hudson, F. Flannery, C. S. Ananian, D. Wang, and A. W. Appel, *CUP Parser Generator for Java*, http://www.cs.princeton.edu/~appel/modern/java/CUP/, 1999.
26. C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, Menlo Park, CA, 1995. www.cs.princeton.edu/software/lcc/.