Create a working compiler with the LLVM framework, Part 2

Use clang to preprocess C/C++ code

Arpan Sen June 19, 2012

The LLVM compiler infrastructure provides a powerful way to optimize your applications regardless of the programming language you use. Learn to instrument code in LLVM, using the clang API to preprocess `C/C++` code in this second article of a two-part series.

Other articles in this series

View more articles in the Create a working compiler with the LLVM framework series.

The first article in this series explored LLVM intermediate representation (IR). You hand-crafted a "Hello World" test program; learned some of LLVM's nuances, like type casting; and finished it off by creating the same program using LLVM application programming interfaces (APIs). In the process, you also learned about LLVM tools such as `llc` and `lli` and figured out how to use llvm-gcc to emit LLVM IR for you. This second and concluding part of the series explores some of the other cool things that you can do with LLVM. Specifically, it looks at instrumenting code—that is, adding information to the final executable that is generated. It also explores a bit of clang, a front end for LLVM supporting `c`, `c++`, and Objective-C. You use the clang API to preprocess and generate an abstract syntax tree (AST) for `c/c++` code.

LLVM passes

LLVM is known for the optimization features it provides. Optimizations are implemented as *passes* (for high-level details about LLVM passes, see Related topics). The thing to note here is that LLVM provides you with the ability to create utility passes with minimum code. For example, if you don't want your function names to begin with "hello," you can have a utility pass to do just that.

Understanding the LLVM opt tool

From the man page for `opt`, the "`opt` command is the modular LLVM optimizer and analyzer." Once you have the code for the custom pass ready, you compile it into a shared library and load it using

`opt`. If your LLVM installation went well, `opt` should already be available in your system. The `opt` command accepts both LLVM IR (the .ll extension) and LLVM bit-code formats (the .bc extension) and can generate the output as either LLVM IR or bit-code. Here's how you use `opt` to load your custom shared library:

```
tintin# opt –load=mycustom_pass.so –help –S
```

Also note that running `opt –help` from the command line generates a laundry list of passes that LLVM performs. Using the `load` option with `help` generates a help message that includes information about your custom pass.

Creating a custom LLVM pass

You declare LLVM passes in the Pass.h file, which on my system is installed under /usr/include/llvm. This file defines the interface for an individual pass as part of the `Pass` class. Individual pass types, each derived from `Pass`, are also declared in this file. Pass types include:

- **`BasicBlockPass` class.** Used to implement local optimizations, with optimizations typically operating on a basic block or instruction at a time
- **`FunctionPass` class.** Used for global optimizations, one function at a time
- **`ModulePass` class.** Used to perform just about any unstructured interprocedural optimizations

Because you intend to create a pass that should object to any function names beginning with "Hello," it's imperative that you create your own pass by deriving from `FunctionPass`. Copy the code in Listing 1 from Pass.h.

Listing 1. Overriding the runOnFunction class in FunctionPass

```
Class FunctionPass : public Pass {
  /// explicit FunctionPass(char &pid) : Pass(PT_Function, pid) {}
  /// runOnFunction - Virtual method overridden by subclasses to do the
  /// per-function processing of the pass.
  ///
  virtual bool runOnFunction(Function &F) = 0;
  /// …
};
```

Likewise, the `BasicBlockPass` class declares a `runOnBasicBlock`, and the `ModulePass` class declares a `runOnModule` pure virtual method. The child class needs to provide a definition for the virtual method.

Coming back to the `runOnFunction` method in Listing 1, you see that the input is an object of type `Function`. Dig into the /usr/include/llvm/Function.h file to easily see that the `Function` class is what LLVM uses to encapsulate the functionality of a `C/C++` function. `Function` in turn is derived from the `Value` class defined in Value.h and supports a `getName` method. Listing 2 shows the code.

Listing 2. Creating a custom LLVM pass

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
class TestClass : public llvm::FunctionPass {
public:
virtual bool runOnFunction(llvm::Function &F)
  {
    if (F.getName().startswith("hello"))
    {
      std::cout << "Function name starts with hello\n";
    }
    return false;
  }
};
```

The code in Listing 2 misses out on two important details:

- The `FunctionPass` constructor needs a `char`, which is used internally by LLVM. LLVM uses the address of the `char`, so what you use to initialize it should not matter.
- You need some way for the LLVM system to understand that the class you created is a new pass. This is where the `RegisterPass` LLVM template comes in. You declare the `RegisterPass` template in the PassSupport.h header file; this file is included in Pass.h, so you don't need additional headers.

Listing 3 shows the complete code.

Listing 3. Registering the LLVM Function pass

```
class TestClass : public llvm::FunctionPass
{
public:
  TestClass() : llvm::FunctionPass(TestClass::ID) { }
  virtual bool runOnFunction(llvm::Function &F) {
    if (F.getName().startswith("hello")) {
      std::cout << "Function name starts with hello\n";
    }
    return false;
  }
  static char ID; // could be a global too
};
char TestClass::ID = 'a';
static llvm::RegisterPass<TestClass> global_("test_llvm", "test llvm", false, false);
```

The `template` parameter in the `RegisterPass` template is the name of the pass to be used on the command line with `opt`. That's it: All you need to do now is create a shared library out of the code in Listing 3, and then run `opt` to load the library followed by the name of the command you registered using `RegisterPass`—in this case, `test_llvm`—and finally a bit-code file on which your custom pass will run along with the other passes. The steps are outlined in Listing 4.

Listing 4. Running the custom pass

```
bash$ g++ -c pass.cpp -I/usr/local/include `llvm-config --cxxflags`
bash$ g++ -shared -o pass.so pass.o -L/usr/local/lib `llvm-config --ldflags -libs`
bash$ opt -load=./pass.so –test_llvm < test.bc
```

Now look at the other side of the coin—the front end to the LLVM back end: clang.

Introducing clang

Notes before you begin

Clang is work in progress, and as is obvious with any project of this scale, the documentation is typically behind the code base. As such, it's best to check out the developer mailing list (see Related topics for a link). You might want to have clang sources built and installed: To do so, follow the instructions in the clang getting started guide (see Related topics). Note that you need to issue the `make install` command after the build is complete for installation to default system folders. The rest of this article assumes that the clang headers and libraries reside in system folders similar to /usr/local/include and /usr/local/lib, respectively.

LLVM has its own front end—a tool (appropriately enough) called *clang.* Clang is a powerful `C/C++`/Objective-C compiler, with compilation speeds comparable to or better than GNU Compiler Collection (GCC) tools (see Related topics for a link to more information). More importantly, clang has a hackable code base, making for easy custom extensions. Much like the way you used the LLVM back-end API for your custom plug-in in Part 1, in this article you use the API for the LLVM front end and develop some small applications for preprocessing and parsing.

Common clang classes

You need to familiarize yourself with some of the most common clang classes:

- `CompilerInstance`
- `Preprocessor`
- `FileManager`
- `SourceManager`
- `DiagnosticsEngine`
- `LangOptions`
- `TargetInfo`
- `ASTConsumer`
- `Sema`
- `ParseAST` is perhaps the most important clang method.

More on the `ParseAST` method shortly.

For all practical purposes, consider `CompilerInstance` the compiler proper. It provides interfaces and manages access to the AST, preprocesses the input sources, and maintains the target information. Typical applications need to create a `CompilerInstance` object to do anything useful. Listing 5 provides a sneak peek into the CompilerInstance.h header file.

Listing 5. The CompilerInstance class

```
class CompilerInstance : public ModuleLoader {
  /// The options used in this compiler instance.
  llvm::IntrusiveRefCntPtr<CompilerInvocation> Invocation;
```

```
  /// The diagnostics engine instance.
  llvm::IntrusiveRefCntPtr<DiagnosticsEngine> Diagnostics;
  /// The target being compiled for.
  llvm::IntrusiveRefCntPtr<TargetInfo> Target;
  /// The file manager.
  llvm::IntrusiveRefCntPtr<FileManager> FileMgr;
  /// The source manager.
  llvm::IntrusiveRefCntPtr<SourceManager> SourceMgr;
  /// The preprocessor.
  llvm::IntrusiveRefCntPtr<Preprocessor> PP;
  /// The AST context.
  llvm::IntrusiveRefCntPtr<ASTContext> Context;
  /// The AST consumer.
  OwningPtr<ASTConsumer> Consumer;
 /// \brief The semantic analysis object.
  OwningPtr<Sema> TheSema;
 //… the list continues
};
```

Preprocessing a C file

There are at least two ways to create a preprocessor object in clang:

- Directly instantiate a `Preprocessor` object
- Use the `CompilerInstance` class to create a `Preprocessor` object for you

Let's begin with the latter approach.

Helper and utility classes needed for preprocessing

The `Preprocessor` alone won't be of much help: You need the `FileManager` and `SourceManager` classes for reading files and tracking source locations for diagnostics. The `FileManager` class implements support for file system lookup, file system caching, and directory search. Look into the `FileEntry` class, which defines the clang abstraction for a source file. Listing 6 provides an excerpt from the FileManager.h header file.

Listing 6. The clang FileManager class

```
class FileManager : public llvm::RefCountedBase<FileManager> {
  FileSystemOptions FileSystemOpts;
   /// \brief The virtual directories that we have allocated.  For each
  /// virtual file (e.g. foo/bar/baz.cpp), we add all of its parent
  /// directories (foo/ and foo/bar/) here.
  SmallVector<DirectoryEntry*, 4> VirtualDirectoryEntries;
  /// \brief The virtual files that we have allocated.
  SmallVector<FileEntry*, 4> VirtualFileEntries;
 /// NextFileUID - Each FileEntry we create is assigned a unique ID #.
  unsigned NextFileUID;
  // Statistics.
  unsigned NumDirLookups, NumFileLookups;
  unsigned NumDirCacheMisses, NumFileCacheMisses;
 // …
  // Caching.
  OwningPtr<FileSystemStatCache> StatCache;
```

The `SourceManager` class is typically queried for `SourceLocation` objects. From the SourceManager.h header file, information about `SourceLocation` objects is provided in Listing 7.

Listing 7. Understanding SourceLocation

```
/// There are three different types of locations in a file: a spelling
/// location, an expansion location, and a presumed location.
///
/// Given an example of:
/// #define min(x, y) x < y ? x : y
///
/// and then later on a use of min:
/// #line 17
/// return min(a, b);
///
/// The expansion location is the line in the source code where the macro
/// was expanded (the return statement), the spelling location is the
/// location in the source where the macro was originally defined,
/// and the presumed location is where the line directive states that
/// the line is 17, or any other line.
```

Clearly, `SourceManager` depends on `FileManager` behind the scenes; indeed, the `SourceManager` class constructor accepts a `FileManager` class as the input argument. Finally, you need to keep track of errors you might encounter while processing the source and report the same. You do so using the `DiagnosticsEngine` class. As with `Preprocessor`, you have two options:

- Create all the necessary objects on your own
- Let the `CompilerInstance` do everything for you

Let's stick with the latter option. Listing 8 shows the code for the `Preprocessor`; everything else has already been explained.

Listing 8. Creating a preprocessor with the clang API

```
using namespace clang;
int main()
{
    CompilerInstance ci;
    ci.createDiagnostics(0,NULL); // create DiagnosticsEngine
    ci.createFileManager();  // create FileManager
    ci.createSourceManager(ci.getFileManager()); // create SourceManager
    ci.createPreprocessor();  // create Preprocessor
    const FileEntry *pFile = ci.getFileManager().getFile("hello.c");
    ci.getSourceManager().createMainFileID(pFile);
    ci.getPreprocessor().EnterMainSourceFile();
    ci.getDiagnosticClient().BeginSourceFile(ci.getLangOpts(), &ci.getPreprocessor());
    Token tok;
    do {
        ci.getPreprocessor().Lex(tok);
        if( ci.getDiagnostics().hasErrorOccurred())
            break;
        ci.getPreprocessor().DumpToken(tok);
        std::cerr << std::endl;
    } while ( tok.isNot(clang::tok::eof));
    ci.getDiagnosticClient().EndSourceFile();
}
```

Listing 8 uses the `CompilerInstance` class to serially create the `DiagnosticsEngine` (the `ci.createDiagnostics` method call) and `FileManager` (`ci.createFileManager` and `ci.CreateSourceManager`) for you. Once the file association is done using `FileEntry`, continue processing each token in the source file until you reach end of file (EOF). The preprocessor's `DumpToken` method dumps the token on screen.

To compile and run the code in Listing 8, use the makefile provided in Listing 9, with appropriate adjustments for your clang and LLVM installation folders. The idea is to use the llvm-config tool to provide any necessary LLVM include paths and libraries: You should never attempt to hand link those on a g++ command line.

Listing 9. Makefile for building preprocessor code

```
CXX := g++
RTTIFLAG := -fno-rtti
CXXFLAGS := $(shell llvm-config --cxxflags) $(RTTIFLAG)
LLVMLDFLAGS := $(shell llvm-config --ldflags --libs)
DDD := $(shell echo $(LLVMLDFLAGS))
SOURCES = main.cpp
OBJECTS = $(SOURCES:.cpp=.o)
EXES = $(OBJECTS:.o=)
CLANGLIBS = \
    -L /usr/local/lib \
    -lclangFrontend \
    -lclangParse \
    -lclangSema \
    -lclangAnalysis \
    -lclangAST \
    -lclangLex \
    -lclangBasic \
    -lclangDriver \
    -lclangSerialization \
    -lLLVMMC \
    -lLLVMSupport \
all: $(OBJECTS) $(EXES)
%: %.o
        $(CXX) -o $@ $< $(CLANGLIBS) $(LLVMLDFLAGS)
```

After compiling and running the above code, you should get the output in Listing 10.

Listing 10. Crash while running the coding in Listing 7

```
Assertion failed: (Target && "Compiler instance has no target!"),
   function getTarget, file
   /Users/Arpan/llvm/tools/clang/lib/Frontend/../..
   /include/clang/Frontend/CompilerInstance.h,
   line 294.
Abort trap: 6
```

What happened here is that you missed out on one last piece of the `CompilerInstance` settings: the target platform for which this code should be compiled. This is where the `TargetInfo` and `TargetOptions` classes come in. According to the clang header TargetInfo.h, the `TargetInfo` class stores the necessary information about the target system for the code generation and has to be created before compilation or preprocessing can ensue. As expected, `TargetInfo` seems to have information about integer and float widths, alignments, and the like. Listing 11 provides an excerpt from the TargetInfo.h header file.

Listing 11. The clang TargetInfo class

```
class TargetInfo : public llvm::RefCountedBase<TargetInfo> {
  llvm::Triple Triple;
protected:
  bool BigEndian;
  unsigned char PointerWidth, PointerAlign;
  unsigned char IntWidth, IntAlign;
  unsigned char HalfWidth, HalfAlign;
  unsigned char FloatWidth, FloatAlign;
  unsigned char DoubleWidth, DoubleAlign;
  unsigned char LongDoubleWidth, LongDoubleAlign;
  // …
```

The `TargetInfo` class takes in two arguments for its initialization: `DiagnosticsEngine` and `TargetOptions`. Of these, the latter must have the `Triple` string set to the appropriate value for the current platform. This is where LLVM comes in handy. Listing 12 shows the addition to Listing 9 to make the preprocessor work.

Listing 12. Setting the target options for the compiler

```
int main()
{
    CompilerInstance ci;
    ci.createDiagnostics(0,NULL);
    // create TargetOptions
    TargetOptions to;
    to.Triple = llvm::sys::getDefaultTargetTriple();
    // create TargetInfo
    TargetInfo *pti = TargetInfo::CreateTargetInfo(ci.getDiagnostics(), to);
    ci.setTarget(pti);
    // rest of the code same as in Listing 9…
    ci.createFileManager();
    // …
```

That's it. Run this code and see what output you get for the simple hello.c test:

```
#include <stdio.h>
int main() {  printf("hello world!\n"); }
```

Listing 13 shows the partial preprocessor output.

Listing 13. Preprocessor output (partial)

```
typedef 'typedef'
struct 'struct'
identifier '__va_list_tag'
l_brace '{'
unsigned 'unsigned'
identifier 'gp_offset'
semi ';'
unsigned 'unsigned'
identifier 'fp_offset'
semi ';'
void 'void'
star '*'
identifier 'overflow_arg_area'
semi ';'
void 'void'
star '*'
identifier 'reg_save_area'
semi ';'
```

```
r_brace '}'
identifier '__va_list_tag'
semi ';'

identifier '__va_list_tag'
identifier '__builtin_va_list'
l_square '['
numeric_constant '1'
r_square ']'
semi ';'
```

Hand-crafting a Preprocessor object

One of the good things about clang libraries, you can achieve the same result in multiple ways. In this section, you craft a `Preprocessor` object but without making a direct request to `CompilerInstance`. From the Preprocessor.h header file, Listing 14 shows the constructor for the `Preprocessor`.

Listing 14. Constructing a Preprocessor object

```
Preprocessor(DiagnosticsEngine &diags, LangOptions &opts,
               const TargetInfo *target,
               SourceManager &SM, HeaderSearch &Headers,
               ModuleLoader &TheModuleLoader,
               IdentifierInfoLookup *IILookup = 0,
               bool OwnsHeaderSearch = false,
               bool DelayInitialization = false);
```

Looking at the constructor, it's clear that you need to create six different objects before this beast can start up. You already know `DiagnosticsEngine`, `TargetInfo`, and `SourceManager`. `CompilerInstance` is derived from `ModuleLoader`. So you must create two new objects—one for `LangOptions` and another for `HeaderSearch`. The `LangOptions` class lets you compile a range of `C/C++` dialects, including `C99`, `C11`, and `C++0x`. Refer to the LangOptions.h and LangOptions.def headers for more information. Finally, the `HeaderSearch` class stores an `std::vector` of directories to search, amidst other things. Listing 15 shows the code for the `Preprocessor`.

Listing 15. Hand-crafted preprocessor

```
using namespace clang;
int main()  {
    DiagnosticOptions diagnosticOptions;
    TextDiagnosticPrinter *printer =
      new TextDiagnosticPrinter(llvm::outs(), diagnosticOptions);
    llvm::IntrusiveRefCntPtr<clang::DiagnosticIDs> diagIDs;
    DiagnosticsEngine diagnostics(diagIDs, printer);
    LangOptions langOpts;
    clang::TargetOptions to;
    to.Triple = llvm::sys::getDefaultTargetTriple();
    TargetInfo *pti = TargetInfo::CreateTargetInfo(diagnostics, to);
    FileSystemOptions fsopts;
    FileManager fileManager(fsopts);
    SourceManager sourceManager(diagnostics, fileManager);
    HeaderSearch headerSearch(fileManager, diagnostics, langOpts, pti);
    CompilerInstance ci;
    Preprocessor preprocessor(diagnostics, langOpts, pti,
      sourceManager, headerSearch, ci);
    const FileEntry *pFile = fileManager.getFile("test.c");
    sourceManager.createMainFileID(pFile);
    preprocessor.EnterMainSourceFile();
    printer->BeginSourceFile(langOpts, &preprocessor);
```

```
    // … similar to Listing 8 here on
}
```

Note a few things about the code in Listing 15:

- You have not initialized `HeaderSearch` to point to any specific directories. You should do so.
- The clang API requires that `TextDiagnosticPrinter` be allocated on the heap. Allocating on the stack causes a crash.
- You have not been able to get rid of `CompilerInstance`. Because you are using `CompilerInstance` anyway, why bother to hand-craft it at all other than to be more comfortable with the clang API?

Language option: C++

You have worked with `c` test code so far: How about a bit of `c++`, then? To the code in Listing 15, add `langOpts.CPlusPlus = 1;`, and try it with the test code in Listing 16.

Listing 16. C++ test code for the preprocessor

```
template <typename T, int n>
struct s {
  T array[n];
};
int main() {
  s<int, 20> var;
}
```

Listing 17 shows the partial output from your program.

Listing 17. Partial preprocessor output from the code in Listing 16

```
identifier 'template'
less '<'
identifier 'typename'
identifier 'T'
comma ','
int 'int'
identifier 'n'
greater '>'
struct 'struct'
identifier 's'
l_brace '{'
identifier 'T'
identifier 'array'
l_square '['
identifier 'n'
r_square ']'
semi ';'
r_brace '}'
semi ';'
int 'int'
identifier 'main'
l_paren '('
r_paren ')'
```

Creating a parse tree

The `ParseAST` method defined in clang/Parse/ParseAST.h is one of the more important methods that clang provides. Here's one of the declaration of the routine, copied from ParseAST.h:

```
void ParseAST(Preprocessor &pp, ASTConsumer *C,
       ASTContext &Ctx, bool PrintStats = false,
       TranslationUnitKind TUKind = TU_Complete,
       CodeCompleteConsumer *CompletionConsumer = 0);
```

`ASTConsumer` provides you with an abstract interface from which to derive. This is the right thing to do, because different clients are likely to dump or process the AST in different ways. Your client code will be derived from `ASTConsumer`. The `ASTContext` class stores—among other things—information about type declarations. Here's the easiest thing to try: Print a list of global variables in your code using the clang ASTConsumer API. Many tech firms have strict rules about global variable usage in `C++` code, and this could be the starting point of creating your custom lint tool. The code for your custom consumer is provided in Listing 18.

Listing 18. A custom AST consumer class

```
class CustomASTConsumer : public ASTConsumer {
public:
 CustomASTConsumer () :  ASTConsumer() { }
    virtual ~ CustomASTConsumer () { }
    virtual bool HandleTopLevelDecl(DeclGroupRef decls)
    {
        clang::DeclGroupRef::iterator it;
        for( it = decls.begin(); it != decls.end(); it++)
        {
            clang::VarDecl *vd = llvm::dyn_cast<clang::VarDecl>(*it);
            if(vd)
               std::cout << vd->getDeclName().getAsString() << std::endl;;
        }
        return true;
    }
};
```

You are overriding the `HandleTopLevelDecl` method (originally provided in `ASTConsumer`) with your own version. Clang is passing the list of globals to you; you iterate over the list and print the variable names. Excerpted from ASTConsumer.h, Listing 19 shows several other methods that client consumer code could override.

Listing 19. Other methods you could override in client code

```
/// HandleInterestingDecl - Handle the specified interesting declaration. This
/// is called by the AST reader when deserializing things that might interest
/// the consumer. The default implementation forwards to HandleTopLevelDecl.
virtual void HandleInterestingDecl(DeclGroupRef D);

/// HandleTranslationUnit - This method is called when the ASTs for entire
/// translation unit have been parsed.
virtual void HandleTranslationUnit(ASTContext &Ctx) {}

/// HandleTagDeclDefinition - This callback is invoked each time a TagDecl
/// (e.g. struct, union, enum, class) is completed.  This allows the client to
/// hack on the type, which can occur at any point in the file (because these
/// can be defined in declspecs).
virtual void HandleTagDeclDefinition(TagDecl *D) {}

/// Note that at this point it does not have a body, its body is
  /// instantiated at the end of the translation unit and passed to
  /// HandleTopLevelDecl.
  virtual void HandleCXXImplicitFunctionInstantiation(FunctionDecl *D) {}
```

Finally, Listing 20 shows the actual client code using the custom AST consumer class that you developed.

Listing 20. Client code using a custom AST consumer

```
int main() {
    CompilerInstance ci;
    ci.createDiagnostics(0,NULL);
    TargetOptions to;
    to.Triple = llvm::sys::getDefaultTargetTriple();
    TargetInfo *tin = TargetInfo::CreateTargetInfo(ci.getDiagnostics(), to);
    ci.setTarget(tin);
    ci.createFileManager();
    ci.createSourceManager(ci.getFileManager());
    ci.createPreprocessor();
    ci.createASTContext();
    CustomASTConsumer *astConsumer = new CustomASTConsumer ();
    ci.setASTConsumer(astConsumer);
    const FileEntry *file = ci.getFileManager().getFile("hello.c");
    ci.getSourceManager().createMainFileID(file);
    ci.getDiagnosticClient().BeginSourceFile(
       ci.getLangOpts(), &ci.getPreprocessor());
    clang::ParseAST(ci.getPreprocessor(), astConsumer, ci.getASTContext());
    ci.getDiagnosticClient().EndSourceFile();
    return 0;
}
```

Conclusion

Other articles in this series

View more articles in the Create a working compiler with the LLVM framework series.

This two-part series covered a lot of ground: It explored LLVM IR, offered ways to generate IR through hand-crafting and LLVM APIs, showed how to create a custom plug-in for the LLVM back end, and explained the LLVM front end and its rich set of headers. You also learned how to use this front end for preprocessing and AST consumption. Creating a compiler and extending it, particularly for complex languages like `C++`, had seemed like rocket science earlier in the history of computing, but with LLVM, life has been simplified. Documentation is where LLVM and clang still need work, but until that's sorted out, I recommend a fresh cup of brew and VIM/doxygen to browse the headers. Have fun!

Related topics

- Learn the basics of the LLVM in Create a working compiler with the LLVM framework, Part 1: Build a custom compiler with LLVM and its intermediate representation (Arpan Sen, developerWorks, June 2012). Optimize your applications regardless of the programming language you use with the powerful the LLVM compiler infrastructure. Building a custom compiler just got easier!
- Learn more about LLVM passes.
- Read Getting Started: Building and Running Clang for detailed information on building and installing clang.
- Take the official LLVM Tutorial for a great introduction to LLVM.
- Dig into the LLVM Programmer's Manual, an indispensable resource for the LLVM API.
- Visit the LLVM project site and download the latest version.
- Find details about clang from the LLVM site.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.