LLVM Technologies in Depth

Ted Kremenek
Manager, Compiler Frontend Team

Roadmap

- Clang in Xcode 4
  - Code completion
  - Fix-it
  - Indexing and Edit-all-in-scope
- Clients of LLVM
  - LLDB
  - Integrated assembler

Using Clang Inside Xcode 4

Building Great C/C++ Source Tools is Hard

- C/C++ are complex languages:
  - Macros
  - Function and operator overloading
  - Namespaces
  - C++ templates
- Great source tools?
  - Few tools act like they really "understand" your code

Xcode 3 Tools

A plethora of C parsers

- Most source tools use their own parser
- Why all this replication?
  - Imprecise and error prone
  - Inconsistencies with compiler
- Design limitations lead to a suboptimal user experience

Why Not Reuse the Compiler's Parser?

Benefits

- Precise results
- Consistent with the compiler
- New language features immediately recognized by source tools

Challenges

- Compilers are often monolithic
- Compiler frontends typically not engineered for general tools support
- Parser needs to be fast

Clang and Xcode 4

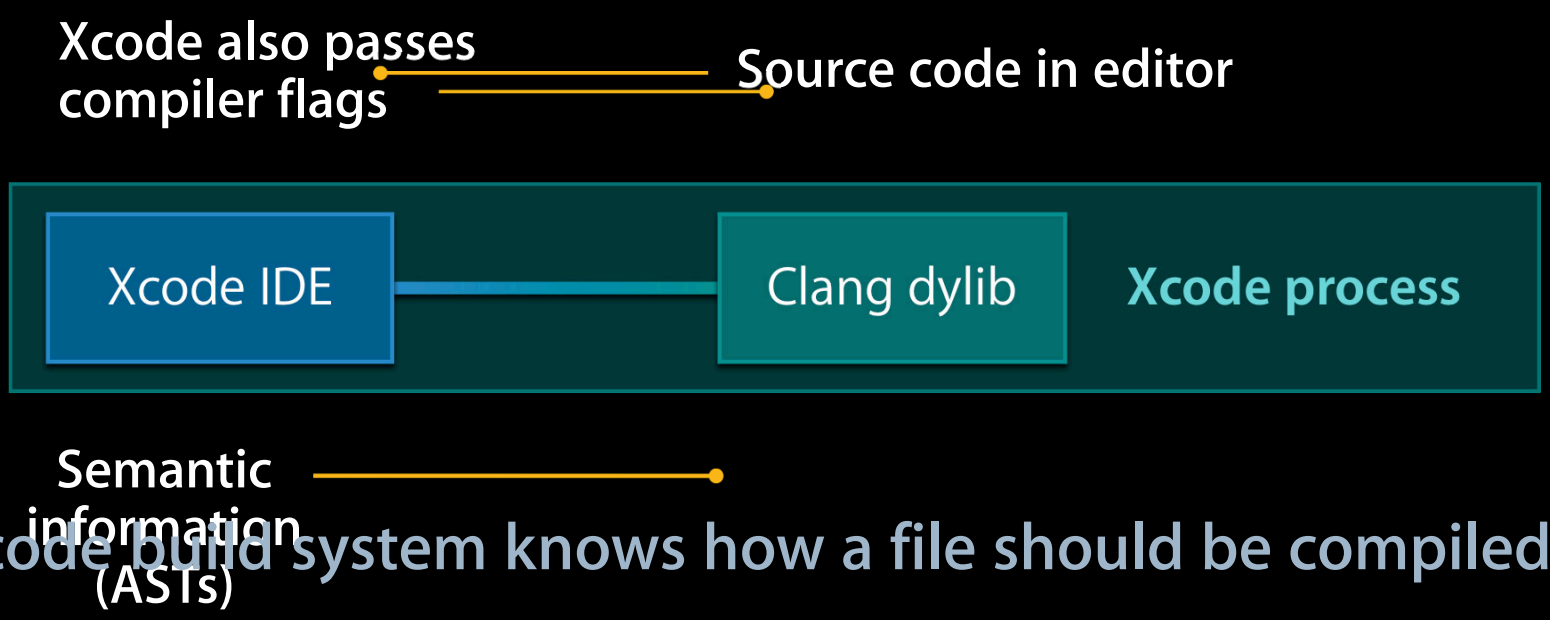
Bring the compiler's precision into the IDE

- Source code indexing
- Syntax highlighting
- Code completion
- Edit-all-in-scope
- Live warnings and Fix-it hints

How Does It Work?

Step 1—Tight Integration with Xcode

Step 1—Tight Integration with Xcode



Xcode build system knows how a file should be compiled

- Headers and preprocessor definitions
- Language flags
- Crucial for extracting meaningful information from Clang

Clang-Based Code Completion

Goals

- Accurate type information for expressions
- AST accurately represents the language (“C++ is hard”)
- Handle overloaded operators, functions, templates

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->
```

code completion

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I-> code completion
```

What has Clang done up to this point?

- **Parsed** the definition of `Wow` and `Foo`
- **Instantiated** the template `std::list<Foo>`
- **Resolved** the definition of `std::list<Foo>::iterator`

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->
```

code completion

What is `I`?

- A type name?
- If not, look it up in the current scope, namespace, …
- **Result:** `I` is a variable in the current scope

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->
```

code completion

Resolving '->'

- If this was C, this is always a pointer dereference
- For C++, we must consider operator overloading
- **Result:** `->` is a call to `std::list<Foo>::iterator::operator->()`

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->
```

code completion

Possible completions?

- Call to `operator->()` returns a `Foo *`
- What are the methods of `Foo`?
- **Results:** { `Wow *bar()`, `~Foo()` }

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->bar()->
```

code completion

Possible completions?

- Use of `->` is standard pointer dereference of a `Wow *`
- **Results:** `{ member, ~Wow() }`

Anatomy of a Code Completion

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->bar()->member++;
```

Fix-it

Fix-it suggestions

- Address small typos and omissions
- Happen when the compiler is fairly sure of what you mean
- Are part of the compiler's parser recovery logic

Note!

- Not refactoring
- Local in scope

Fix-it

Goals

- Error recovery in parser needed to determine Fix-it
- Accurate line and column information for edits

Anatomy of a Fix-it

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->bar()->member++;
}
```

Anatomy of a Fix-it

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->ba->member++;
}
```

```
error: no member named 'ba' in 'Foo'; did you mean 'bar'?
    I->ba->member++;
       ^~
       bar

error: base of member reference has function type 'Wow *()'; perhaps you
meant to call this function with '()'?
    I->ba->member;
         ^
         ()
```

Anatomy of a Fix-it

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->ba->member++;
}
```

How to handle `ba`?

- Doesn't resolve to an identifier
- Closest match via typo correction is `bar`
- Suggest Fix-it of `bar` to user
- Pretend that we saw `bar` and continue parsing

Anatomy of a Fix-it

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->bar->member++;
}
```

How to handle `bar` followed by `->` ?

- Referencing `bar` means we are referring to a method
- `->` on `bar` doesn't make sense ... but it does on `bar()`
- Suggest Fix-it of `()` to user
- Pretend we saw `()` and continue parsing

Anatomy of a Fix-it

```
struct Wow { int member; };
struct Foo { Wow *bar(); };

void call_bar(std::list<Foo> &V) {
  for (std::list<Foo>::iterator I=V.begin(), E=V.end(); I!=E; ++I)
    I->bar()->member++;
}
```


Field access to **member** is valid. No errors!

Clang-Based Source Code Indexing

What is it?

- Xcode indexes the "symbols" in your source code
  - Navigation ("Jump to Definition")
  - Quick help
  - Edit-all-in-scope
- Clang-based indexing uses the symbol information from Clang
  - Far more precise than Xcode 3
  - Aids in understanding large codebases

Clang-Based Code Completion

Goals

- Precision
  - Resolve ambiguities (e.g., overloaded functions)
- Good indexing results despite malformed code
- Accurate line and column information
- Lucid understanding of macros

Indexing Code with “Ambiguities”

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Overloaded functions

Call to overloaded function

Overloaded methods

Another `draw()` in a different class

Call to overloaded method via `const` pointer

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Results with Xcode 3

Indexing Code with “Ambiguities”

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Overloaded functions

- Both `print(int)` and `print(float)` are not distinguished by the index

Indexing Code with "Ambiguities"

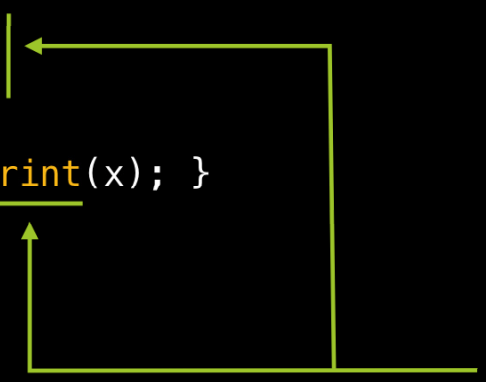
```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```



Target of call ambiguously resolved

- "Jump to Definition" in Xcode 3 for this call gives you both `print(int)` and `print(float)`

Indexing Code with "Ambiguities"

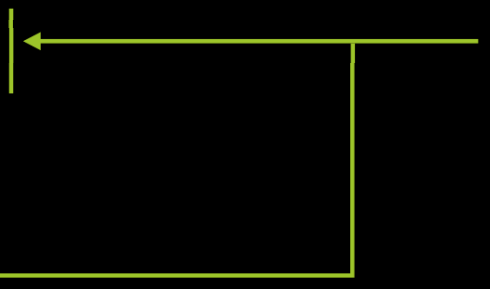
```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```



Methods with same name are not distinguished

- Function overloading ignored
- Enclosing class ignored

Indexing Code with "Ambiguities"

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

"Jump to Definition" returns all three definitions of draw()

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Results with Xcode 4

Indexing Code with "Ambiguities"

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Overloaded functions

- Both `print(int)` and `print(float)` are given different "symbol resolutions"
- Symbol resolution takes into account parameter types, namespaces, ...

Indexing Code with “Ambiguities”

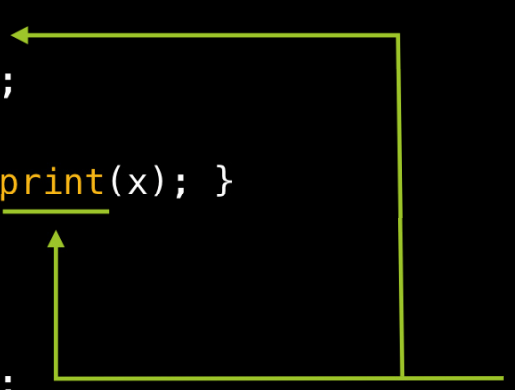
```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```



Target of call uniquely resolved

- Call to `print(int)` is unambiguous
- “Jump to Definition” will show one result
- Xcode 3 would show two results

Indexing Code with "Ambiguities"

```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```

Methods have unique symbol resolutions

Symbol resolution takes into account...

- Qualifiers
- Static versus non-static methods
- Enclosing class

Indexing Code with “Ambiguities”

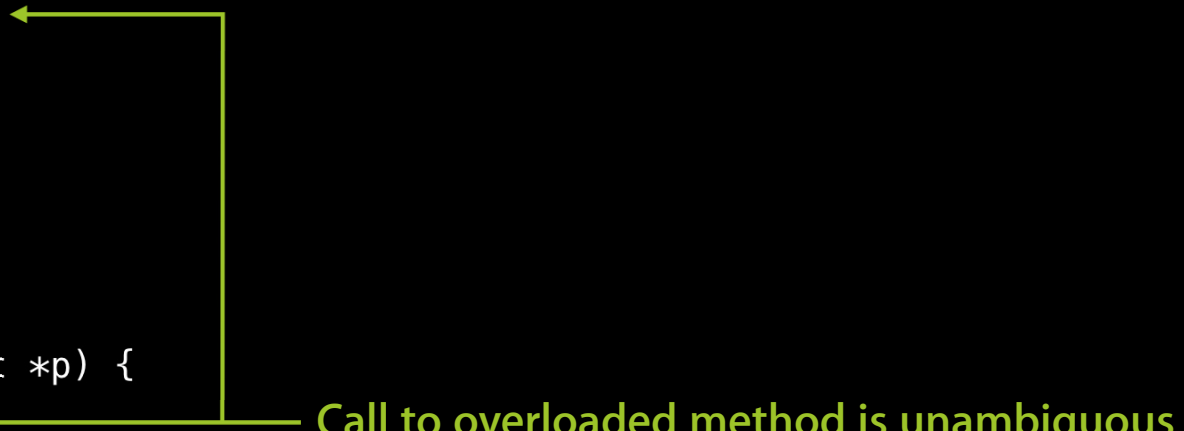
```
void print(int x);
void print(float x);

void test(int x) { print(x); }

struct Shape {
  void draw();
  void draw() const;
};

struct Cowboy {
  void draw();
};

void test(Shape const *p) {
    p->draw();
}
```



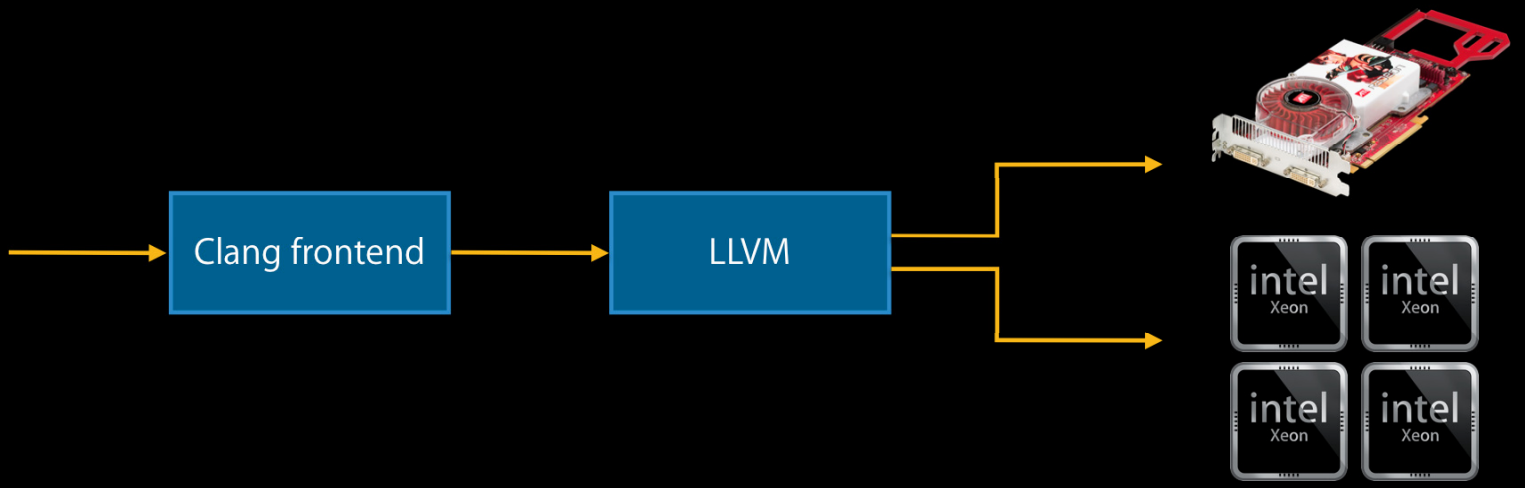
Clients of LLVM

Evan Cheng
Manager, Compiler Code Generator Team

Mac OS X

OpenCL

- Clang frontend with LLVM optimizer and code generator



Speed up CoreImage by 25%!

Other Mac OS X LLVM Clients

OpenGL

MacRuby

LLDB

Brief Introduction to LLDB

LLDB—A modern, modular, and speedy LLVM-based debugger

- Performance
- Handle all C++ constructs
- Multithreading
- Reuse existing compiler technologies
- http://lldb.llvm.org/

Why Not GDB?

- Large, fragile, and aging codebase
- Difficult to add new features and maintain
- Requires its own C / C++ parser, disassembler!

Expression Printing

How do debuggers print expressions?

```
p my_shape->scale(4)
```

Expression Printing in GDB

- GDB includes its own:
  - C/C++ expression parser
  - C type system
  - Type checking logic

```
p my_shape->scale(4)
```

- Costs of yet another parser:
  - Not always correct
  - Difficult to test
  - Implementing new language features

How About LLDB?

LLDB Leverages LLVM Technologies

- Clang frontend
- LLVM code generator and JIT
- LLVM disassembler

LLVM-Based Expression Printing

Goals

- Higher fidelity expression parsing and evaluation
- Support all language constructs
- Less platform specific knowledge in the debugger

Expression Printing in LLDB

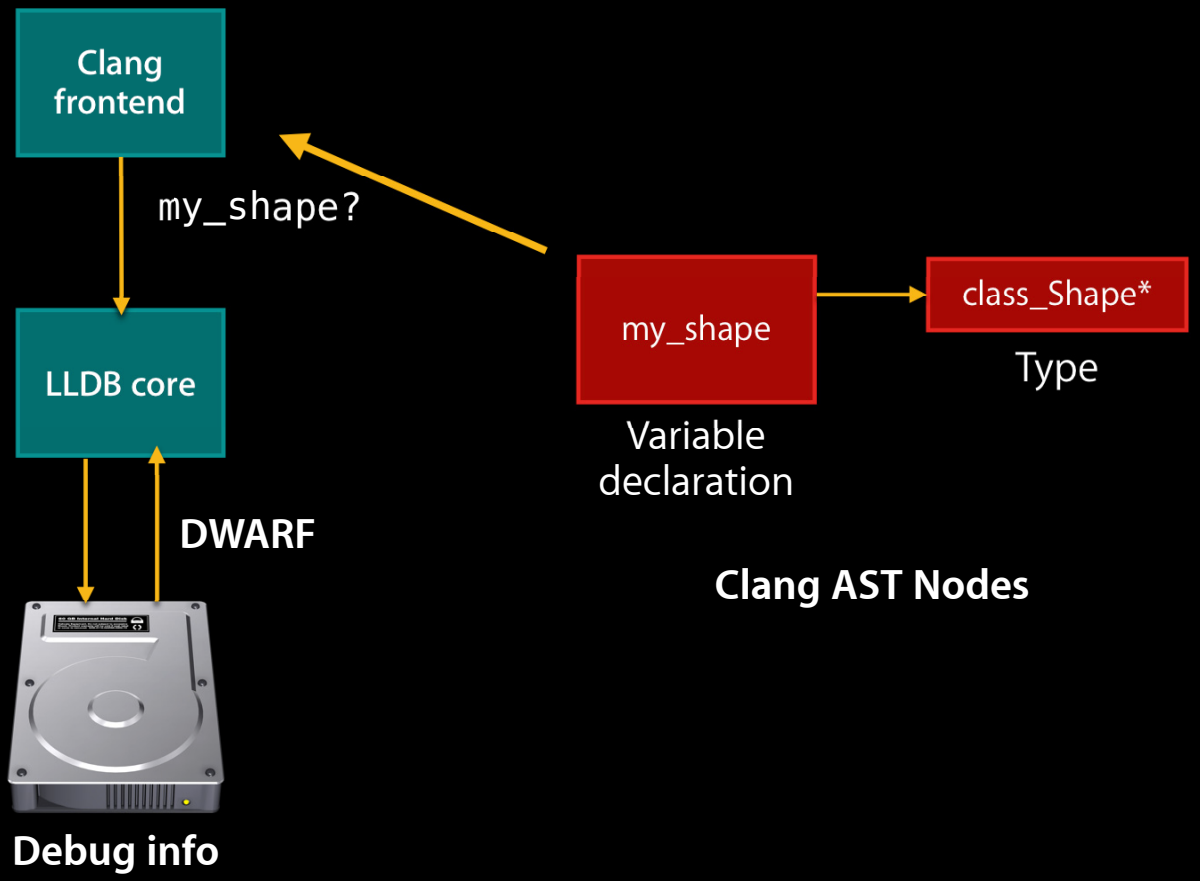
- Clang provides parsing and language semantics
- LLDB provides information about the program that is being run

```
my_shape->scale(4)
```

Lookup "my_shape" to find type of declaration

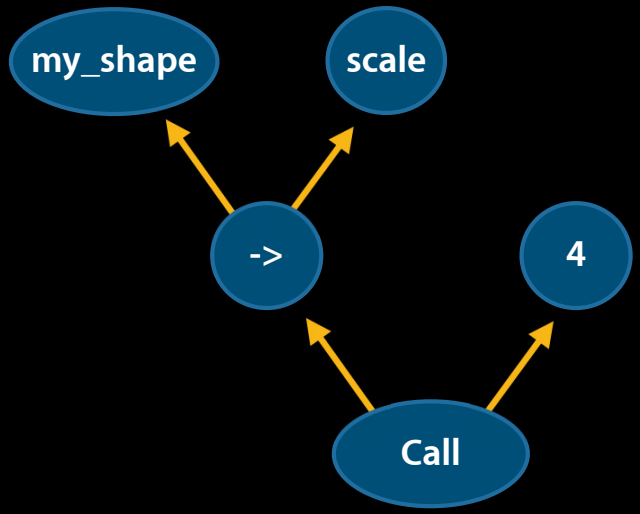
Name Lookup

`my_shape->scale(4)`



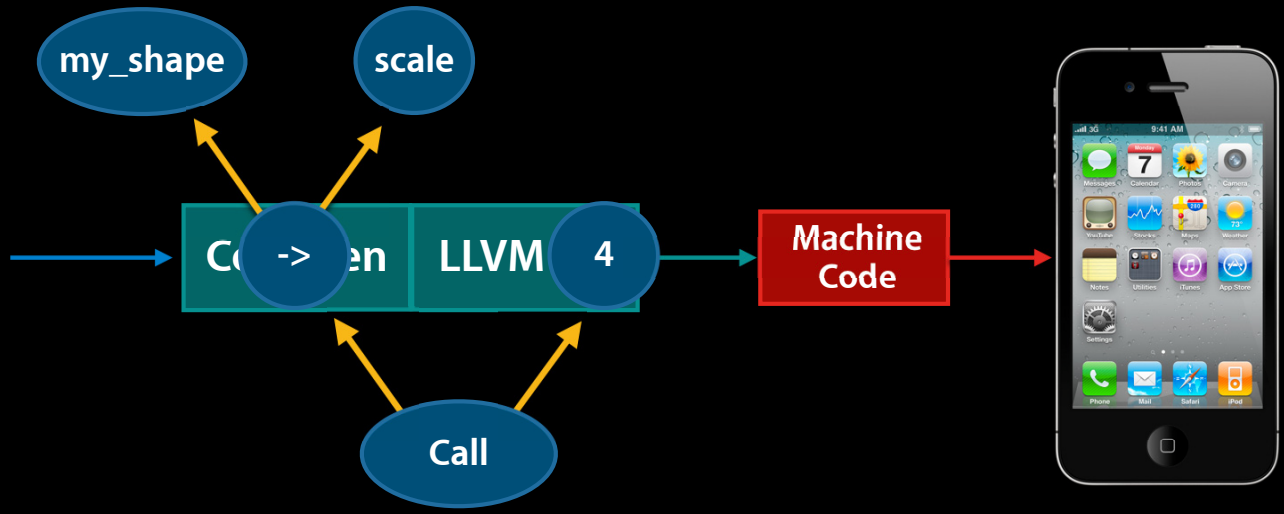
Parser Finishes, Creates Syntax Tree

```
my_shape->scale(4)
```



- Computing the result requires a function call on the device!
  - Calling conventions? Platform specific ABI? Yuck.

LLVM Just-in-Time Code Generation

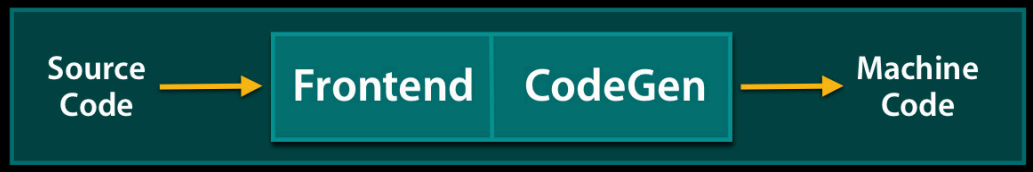


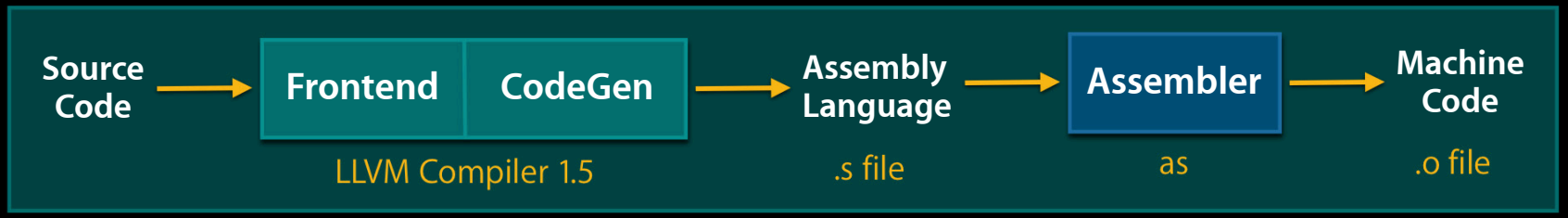
Benefits of LLDB Design

- Higher fidelity expression parsing and evaluation
- Support all language constructs
  - Inline function, template instantiation
  - Debugger gets new language features
- Less platform specific knowledge in the debugger
- Can pull in other compiler features: Fix-it, code completion

Assembler

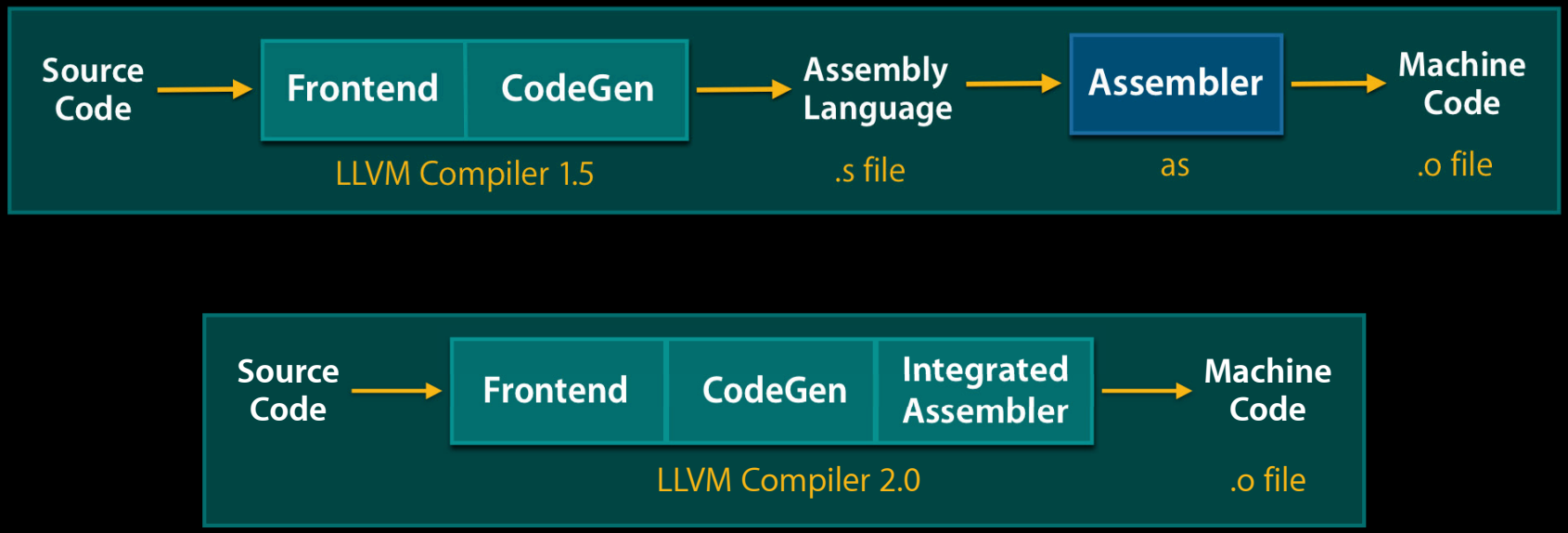
Stages of Compilation





- Why spend time formatting and reparsing a large text file?

LLVM Integrated Assembler



- From source code to .o file with LLVM technologies

Better Error Messages

```
$ gcc test.c
/var/folders/n7/n7Yno9ihEm894640nJdSQU+++TI/-Tmp-//ccJ5wv0H.s:11:no such
instruction: `movr %eax'

$ clang test.c
<inline asm>:1:2: error: unrecognized instruction
        movr    %eax
        ^
test.c:2:11: note: generated from here
 __asm__ ("movr    %0" : "+r" (X));
          ^
```

Better Assembly Dumps

Making assembly dumps more useful

```
LCPI0_0:                                ## constant pool double
   .long 0                              ## double 5.000000e-01
   .long 1071644672

   ...

LBB0_2:                                 ## =>This Inner Loop Header: Depth=1
   movss (%edi,%ebx,4), %xmm0
   movss 20(%ebp), %xmm1
   subss %xmm0, %xmm1
   cvtss2sd %xmm1, %xmm0
   movsd -48(%ebp), %xmm1               ## 8-byte Reload
   andpd %xmm1, %xmm0
   divsd -32(%ebp), %xmm0               ## 8-byte Folded Reload
   cvtsd2ss %xmm0, %xmm0
   movss -52(%ebp), %xmm1               ## 4-byte Reload
   xorps %xmm1, %xmm0
   cvtss2sd %xmm0, %xmm0
```

LLVM Integrated Assembler

- Benefits
  - 10% faster builds (for debug builds of many applications)
  - Better error messages for inline assembly
  - More useful assembly dumps
- Issues with the new integrated assembler?
  - File a bug and use: **-no-integrated-as**

Summary

LLVM playing a foundational role in Apple technologies

- Clang in Xcode 4
  - Code completion
  - Fix-it
  - Indexing and Edit-all-in-scope
- Clients of LLVM
  - OpenCL, OpenGL, MacRuby
  - LLDB: a modern, modular, and speedy LLVM-based debugger
  - Integrated assembler

More Information

Michael Jurewitz
Developer Tools Evangelist
jurewitz@apple.com

LLVM Project
Open-Source LLVM Project Home
http://llvm.org

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Debugging with Xcode 4 and LLDB	Mission Friday 9:00AM

Labs

LLVM Lab	Developer Tools Lab B Thursday 4:30PM
Xcode 4 Lab	Developer Tools Lab B Friday 9:00AM

Q&A


WWDC10