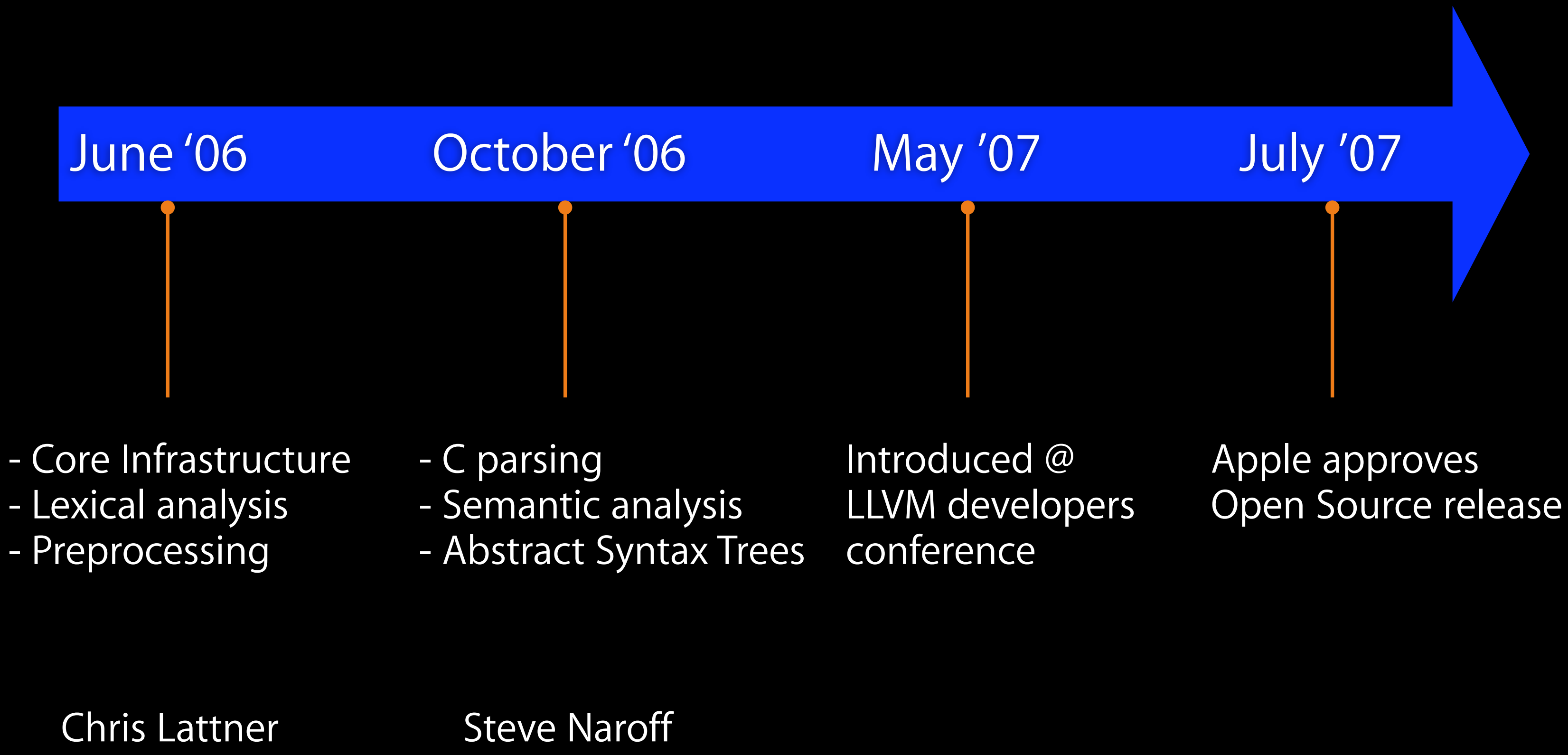
Clang Intro

Steve Naroff
snaroff@apple.com

Clang: What is it?

- C, Objective-C, C++ front-end for llvm
- Drop-in Replacement for GCC
    - Compatibility is critical!
- Part of Open Source LLVM Project
    - Same design approach and organization
    - LLVM UIUC "BSD" License

Clang Timeline



June ‘06

- Core Infrastructure
- Lexical analysis
- Preprocessing

Chris Lattner

October ‘06

- C parsing
- Semantic analysis
- Abstract Syntax Trees

Steve Naroff

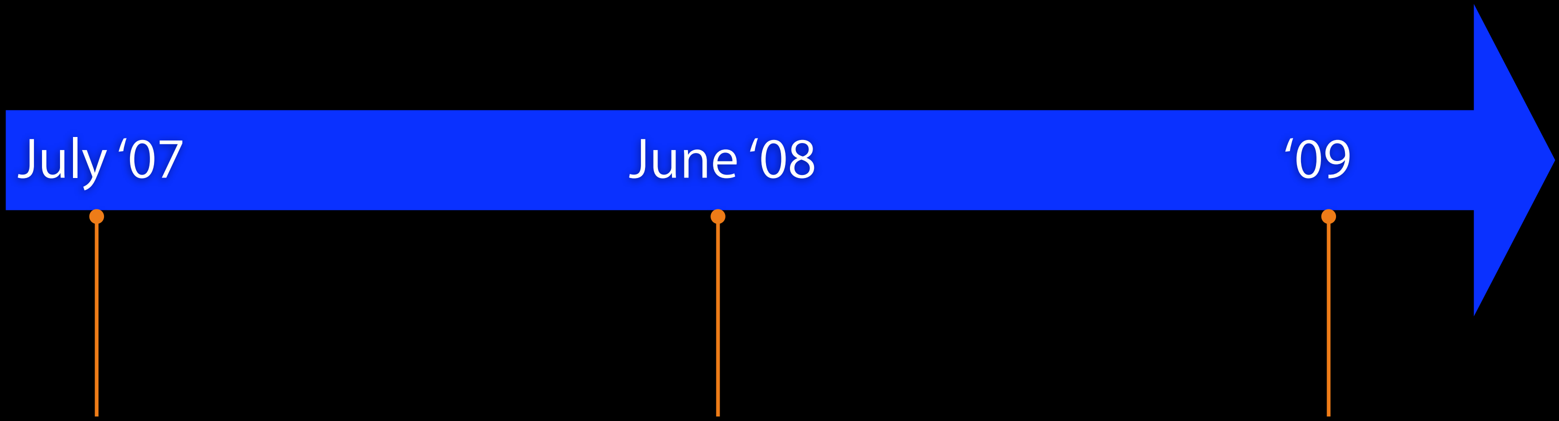
May ’07

Introduced @
LLVM developers
conference

July ’07

Apple approves
Open Source release

Clang Timeline



July '07

June '08

'09

Apple approves
Open Source release

Apple introduces clang at
World Wide Developer Conference

clang native
compiler for C/ObjC

Key Contributors

- Ted Kremenek
- Fariborz Jahanian
- Devang Patel
- Anders Carlsson
- Eli Friedman
- Argiris Kirtzidis
- David Chisnall
- Nate Begeman

Motivation

- ✓ Fast compilation
- ✓ Expressive error messages
- ✓ Foundation for new programming tools

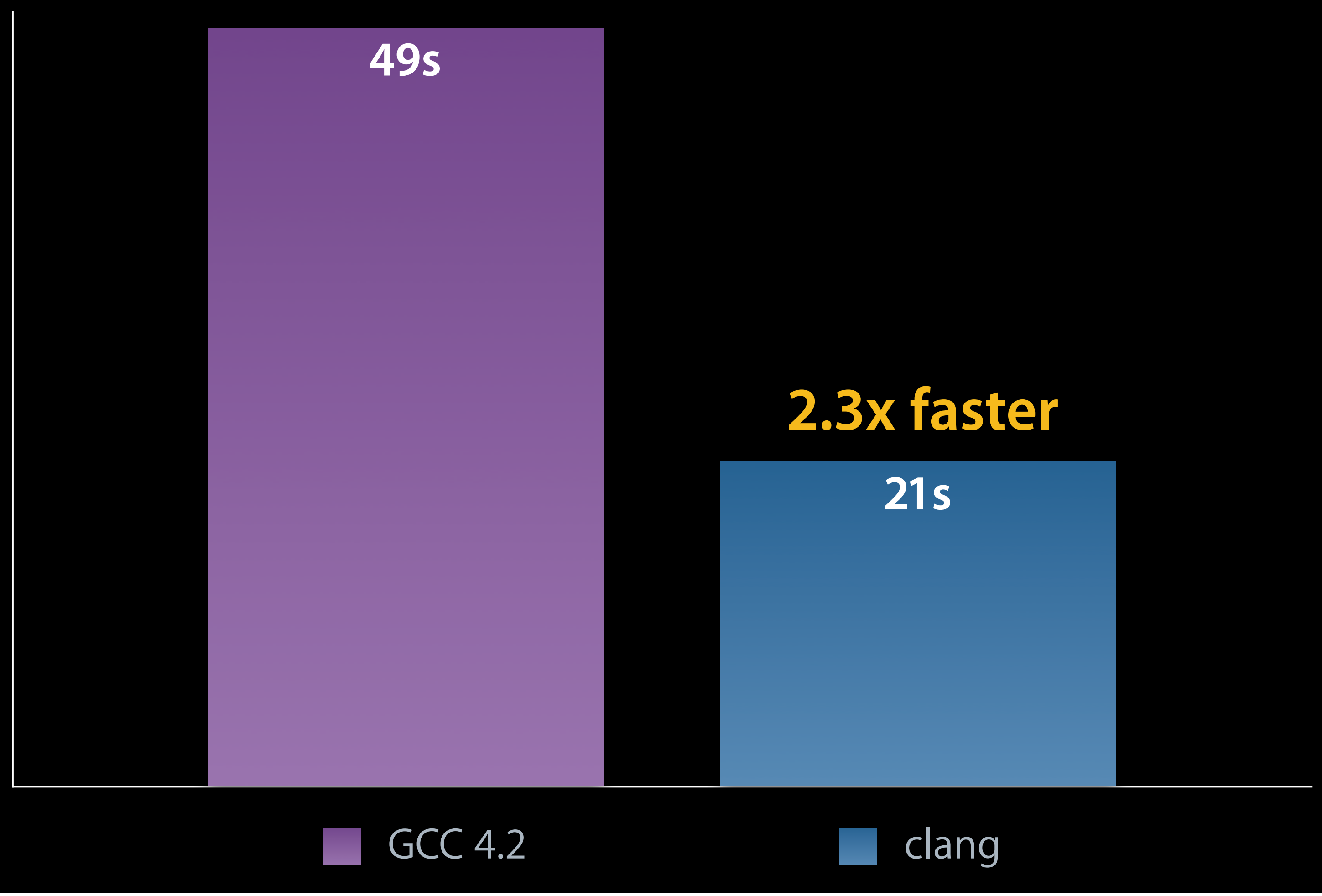
Spur Innovation for the Next Decade

- Progressive Open Source development model
- Modular, LLVM-inspired architecture
- Some specific features we'd like to enable
  - Static analysis / bug finding
  - Refactoring
  - Cross referencing
  - Incremental compilation

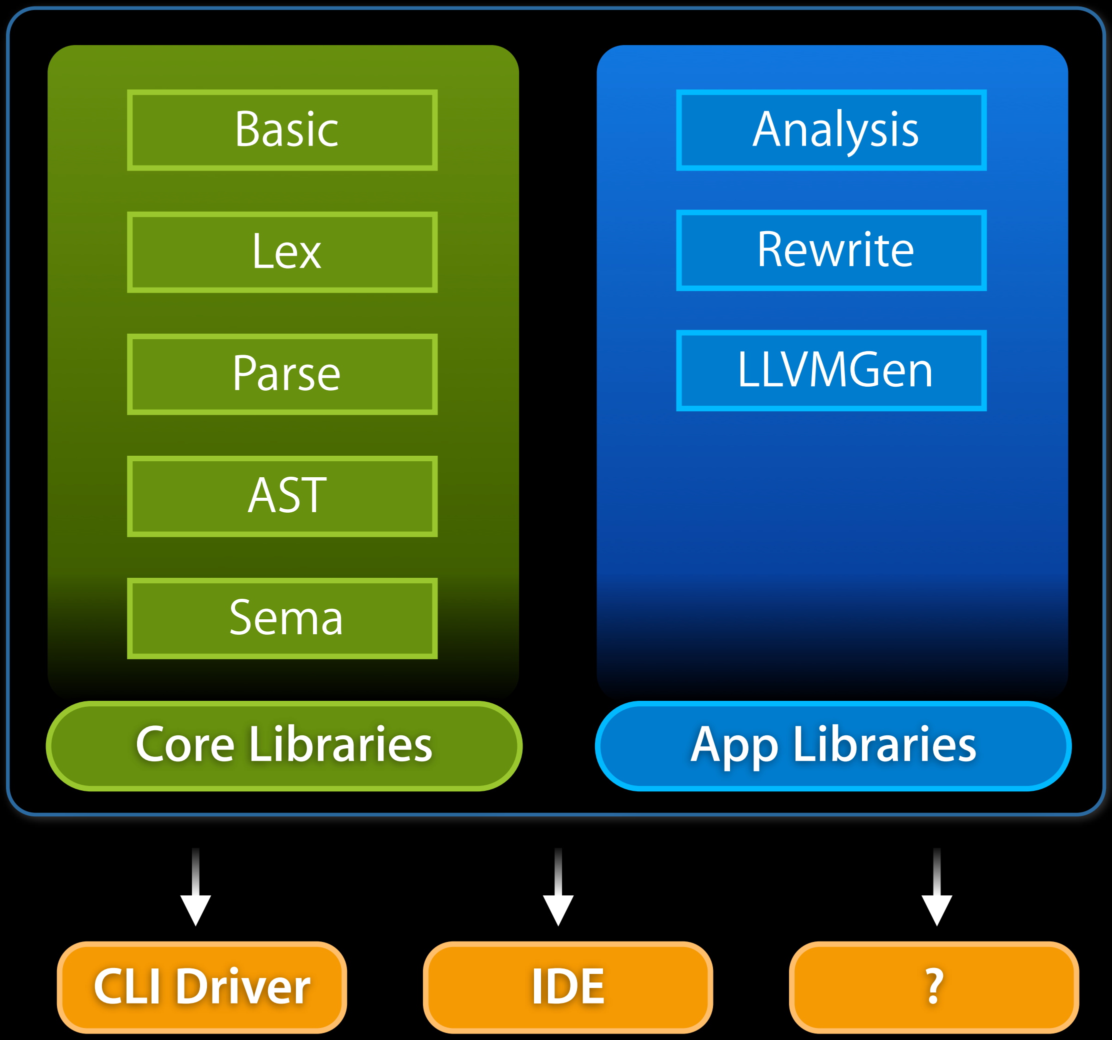
PostgreSQL Front-end Times

Intel Core Duo (2.66 Ghz)—Real-time, in seconds

619 C Files in 665K lines



Clang Libraries



Basic

Lex

Parse

AST

Sema

Core Libraries

Analysis

Rewrite

LLVMGen

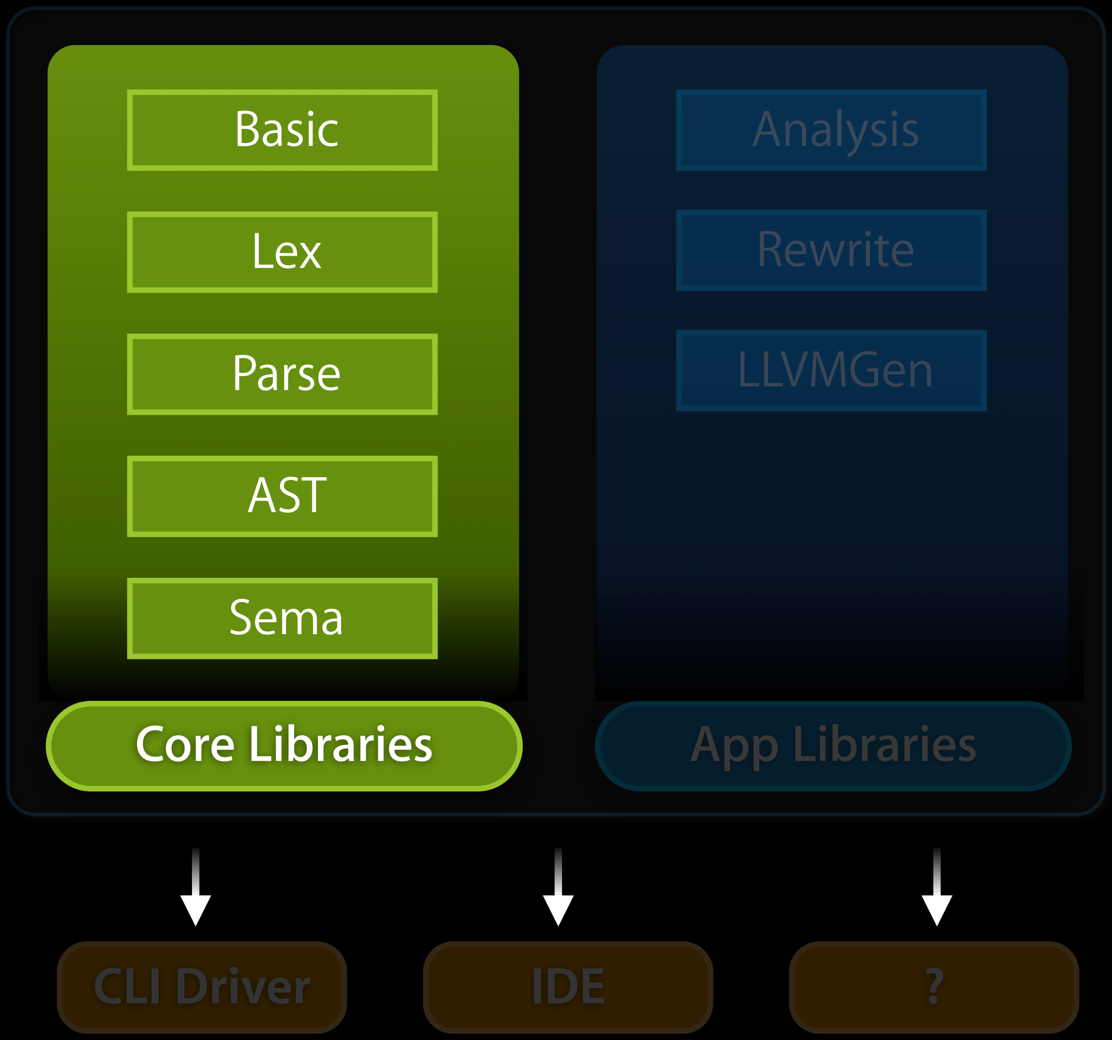
App Libraries

CLI Driver

IDE

?

Clang Libraries



Basic

Lex

Parse

AST

Sema

Core Libraries

Analysis

Rewrite

LLVMGen

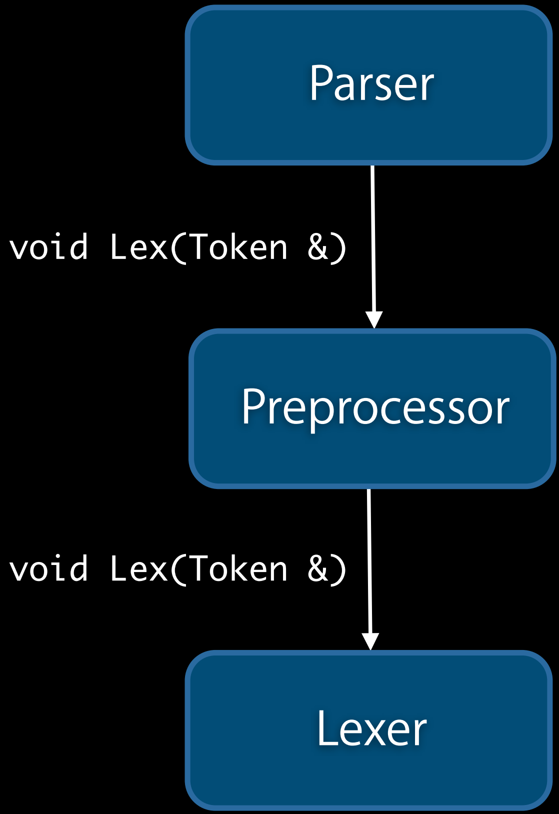
App Libraries

CLI Driver

IDE

?

Clang Components



Parser

void Lex(Token &)

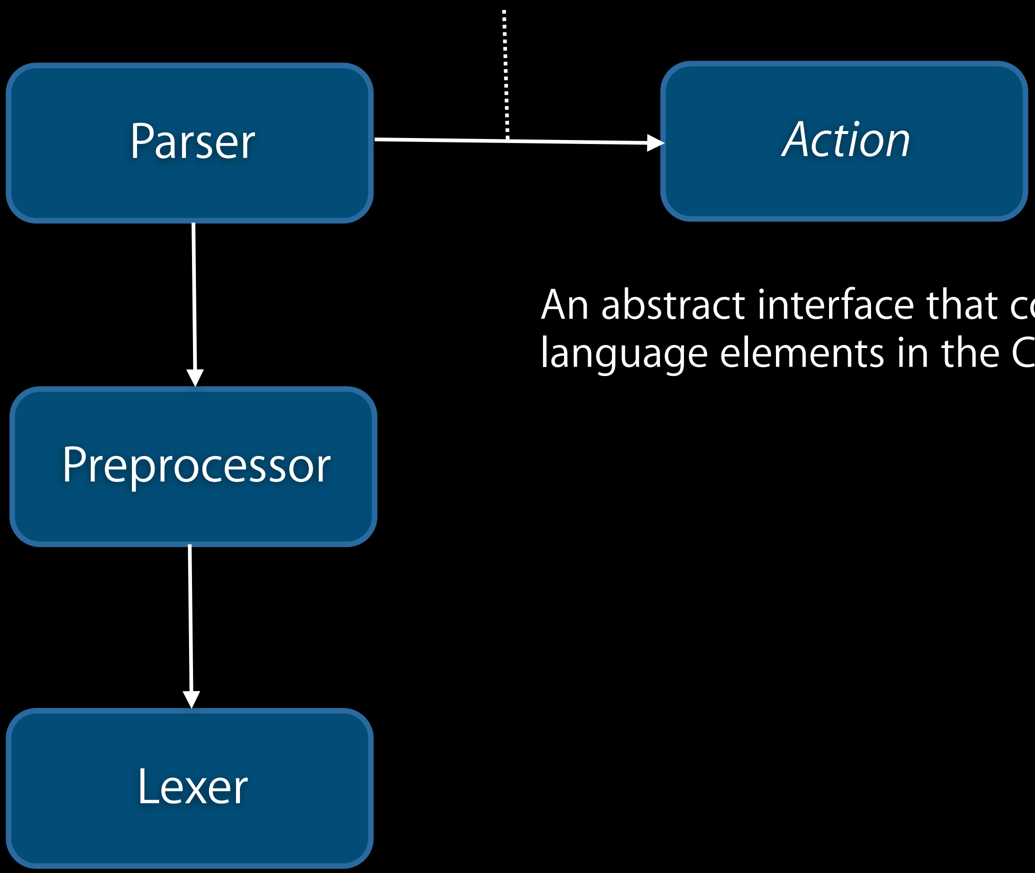
Preprocessor

void Lex(Token &)

Lexer

Clang Parser Actions

“ActOn” delegate methods (45 Declaration, 35 Expression, 25 Statement)



Parser

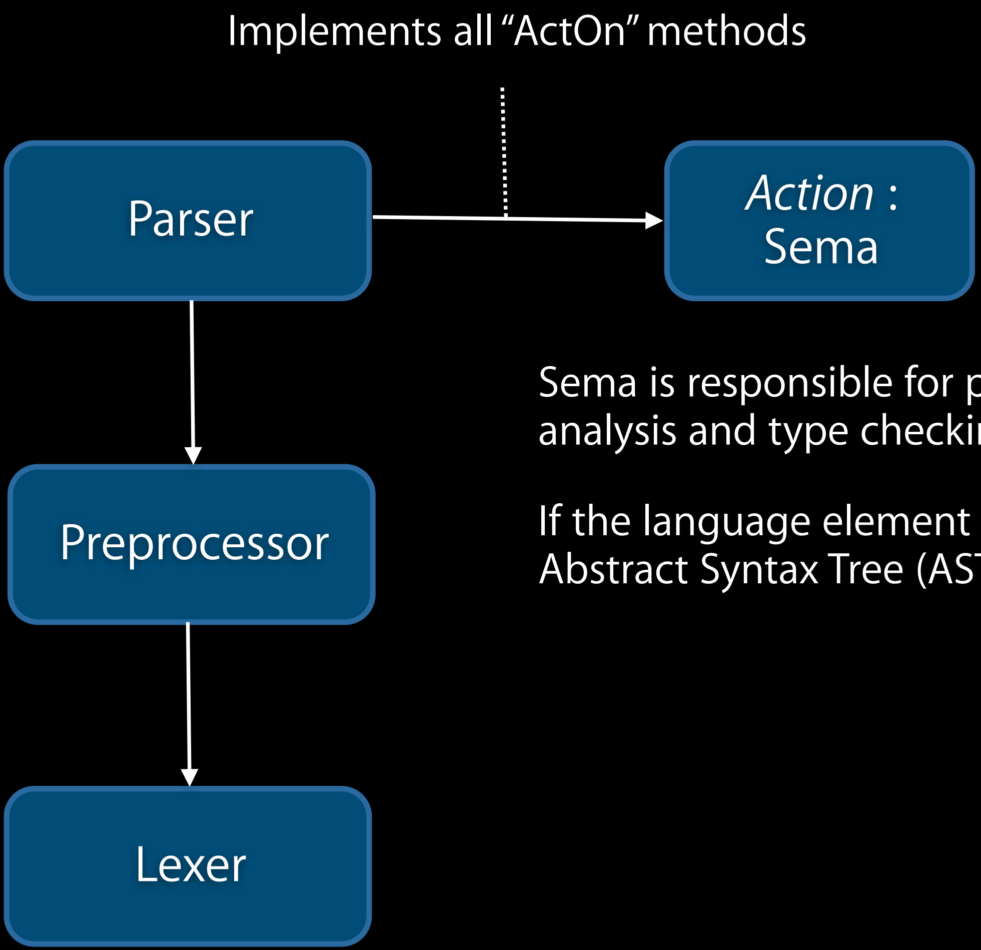
Action

Preprocessor

Lexer

An abstract interface that corresponds to the language elements in the C/ObjC grammar

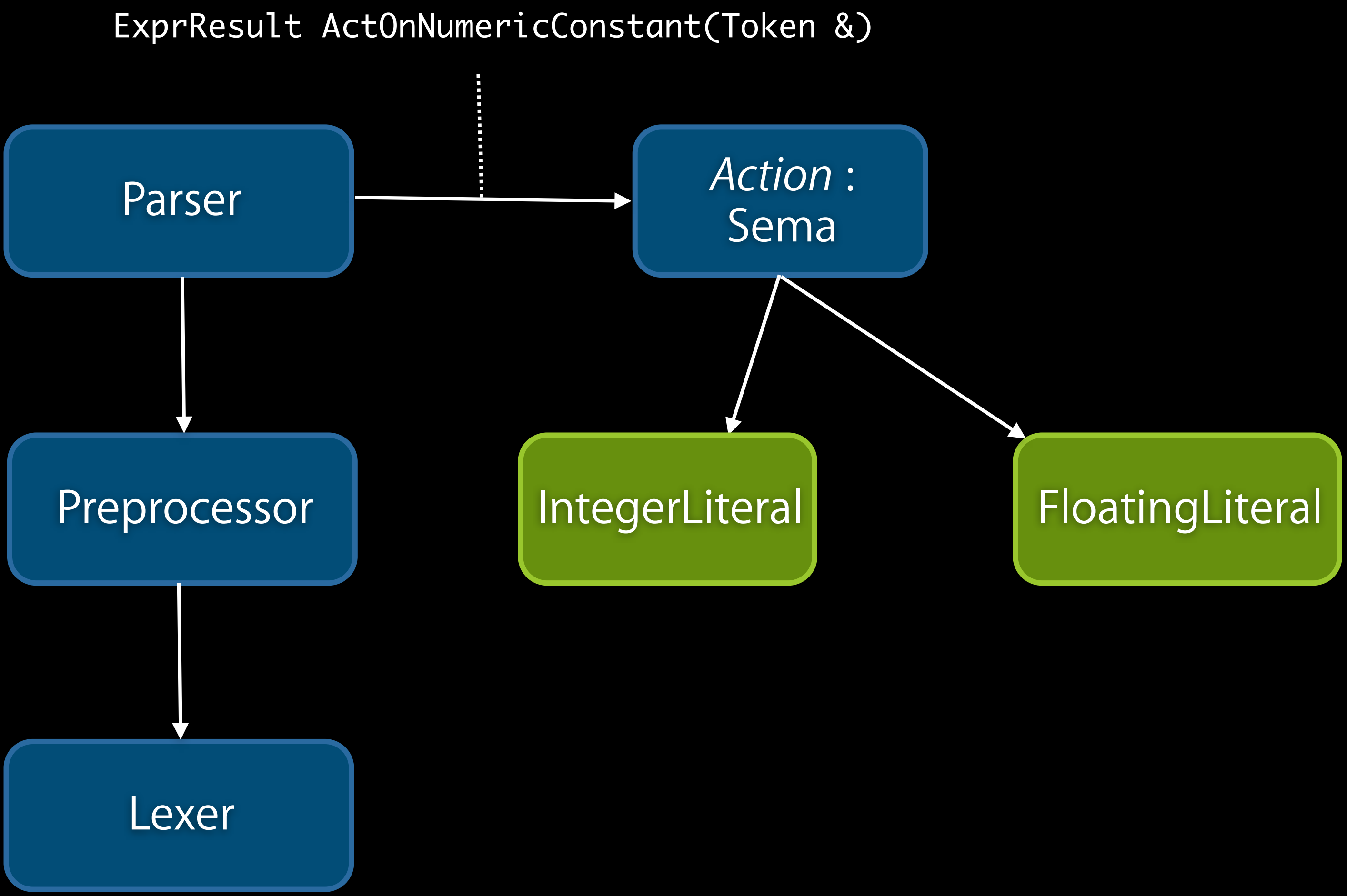
clang -fsyntax-only



Sema is responsible for performing semantic analysis and type checking.

If the language element is well formed, an Abstract Syntax Tree (AST for short) is produced.

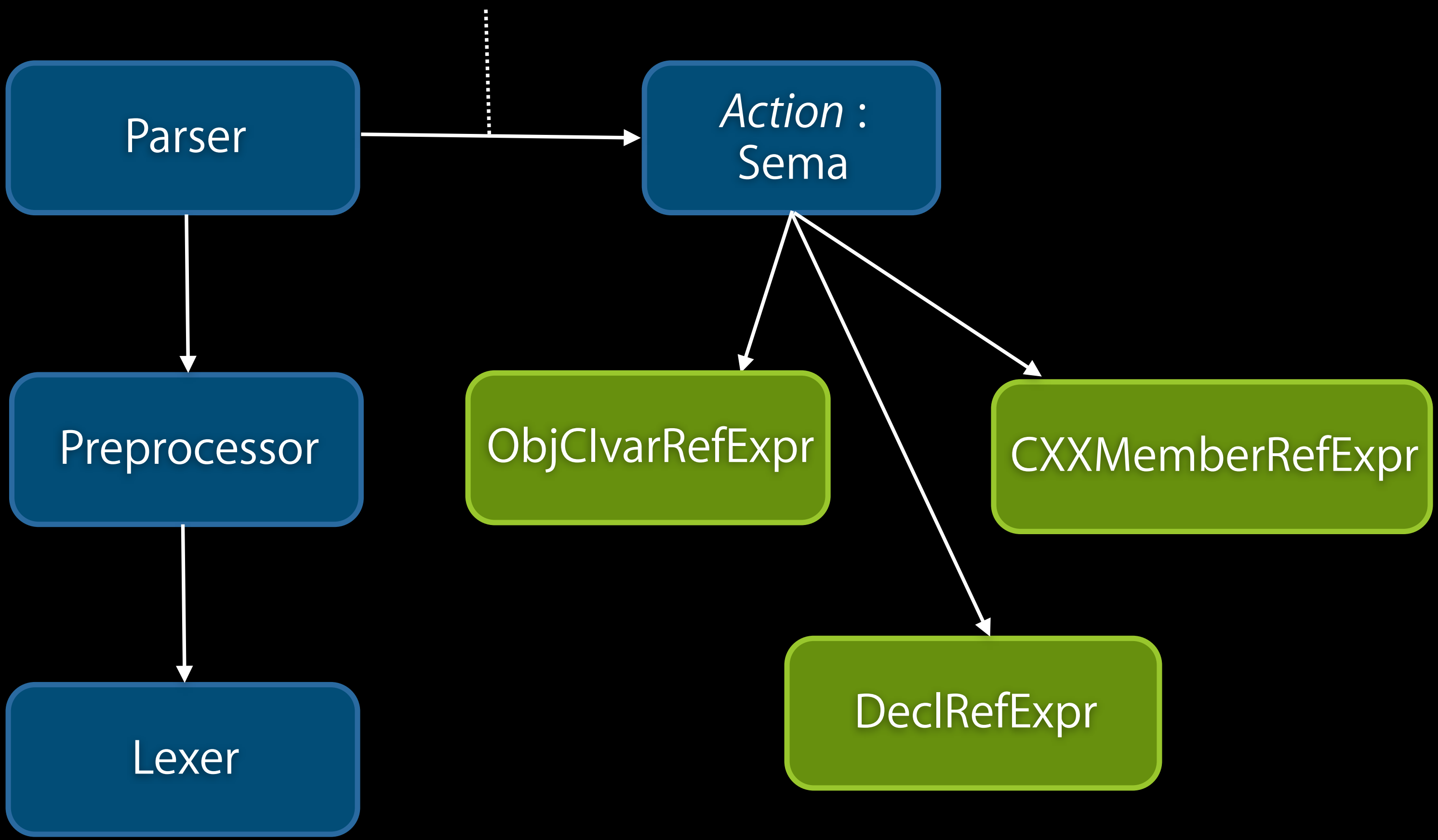
clang -fsyntax-only



```
ExprResult ActOnNumericConstant(Token &)
```

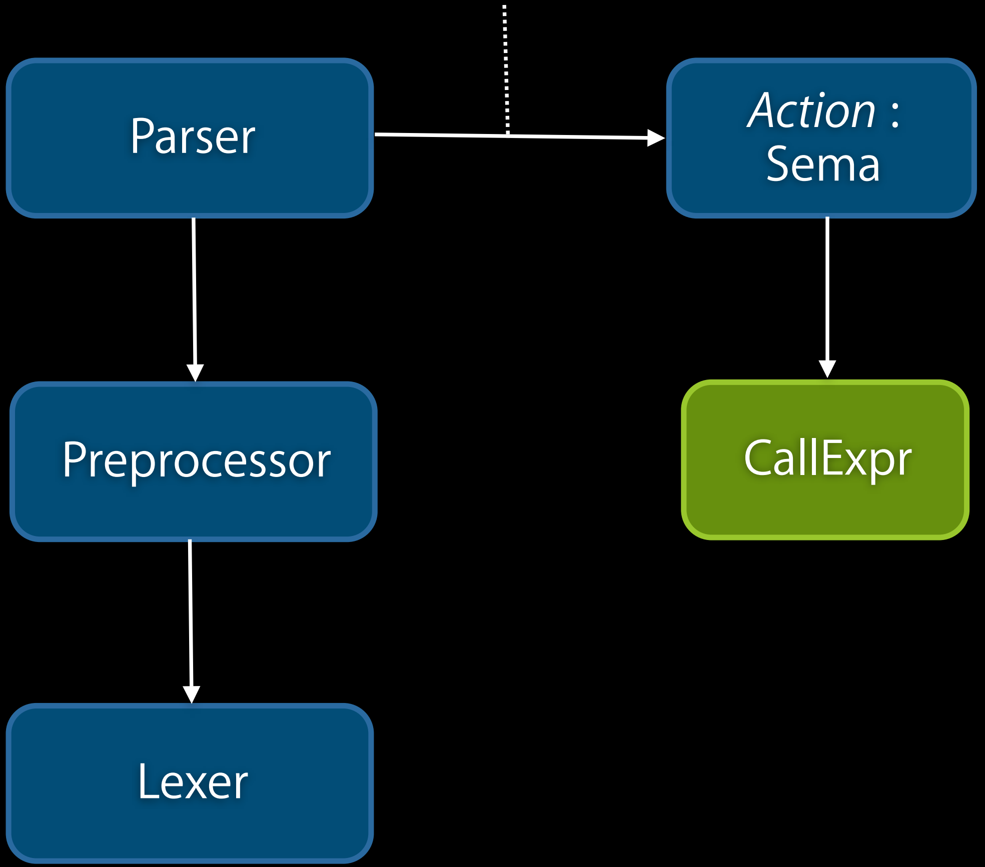
clang -fsyntax-only

```
ExprResult ActOnIdentifierExpr(Scope *S, SourceLocation Loc,
                               IdentifierInfo &II, bool HasTrailingParen)
```

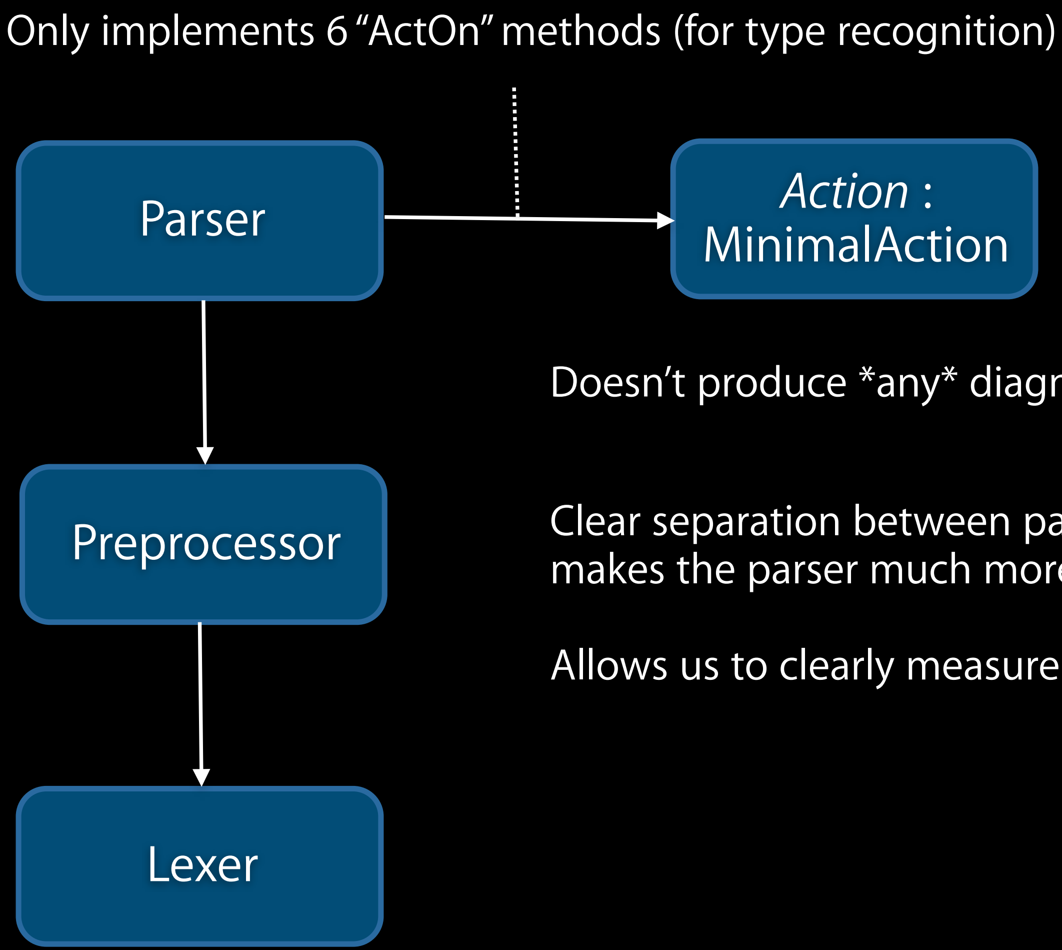


clang -fsyntax-only

```
ExprResult ActOnCallExpr(ExprTy *fn, SourceLocation LParenLoc,
                         ExprTy **args, unsigned NumArgs,
                         SourceLocation *CommaLocs, SourceLocation RParenLoc)
```



clang -parse-noop



Only implements 6 “ActOn” methods (for type recognition)

Parser

Action :
MinimalAction

Preprocessor

Lexer

Doesn’t produce *any* diagnostics or objects!

Clear separation between parsing and semantic makes the parser much more “reusable”.

Allows us to clearly measure the cost of each phase.

AST Requirements

- Support multiple, diverse clients
  - Reflect the source code (e.g. typedef preservation)
  - Preserve source location information (diagnostics, rewriting)
- High performance (both time & space)
  - IdentifierInfo’s and Type’s are uniqued
  - QualType, a smart-pointer class for storing C type qualifiers
  - Selectors, a smart-pointer class for storing ObjC selectors

AST Anatomy

```
class IntegerLiteral : public Expr {
  llvm::APInt Value;
  SourceLocation Loc;
public:
  // Constructor - type should be IntTy, LongTy, LongLongTy,
  // UnsignedIntTy, UnsignedLongTy, or UnsignedLongLongTy
  IntegerLiteral(const llvm::APInt &V, QualType type, SourceLocation l)
    : Expr(IntegerLiteralClass, type), Value(V), Loc(l)
    { assert(type->isIntegerType() && "Illegal type in IntegerLiteral"); }

  // Getters
  const llvm::APInt &getValue() const
    { return Value; }
  virtual SourceRange getSourceRange() const
    { return SourceRange(Loc); }
```

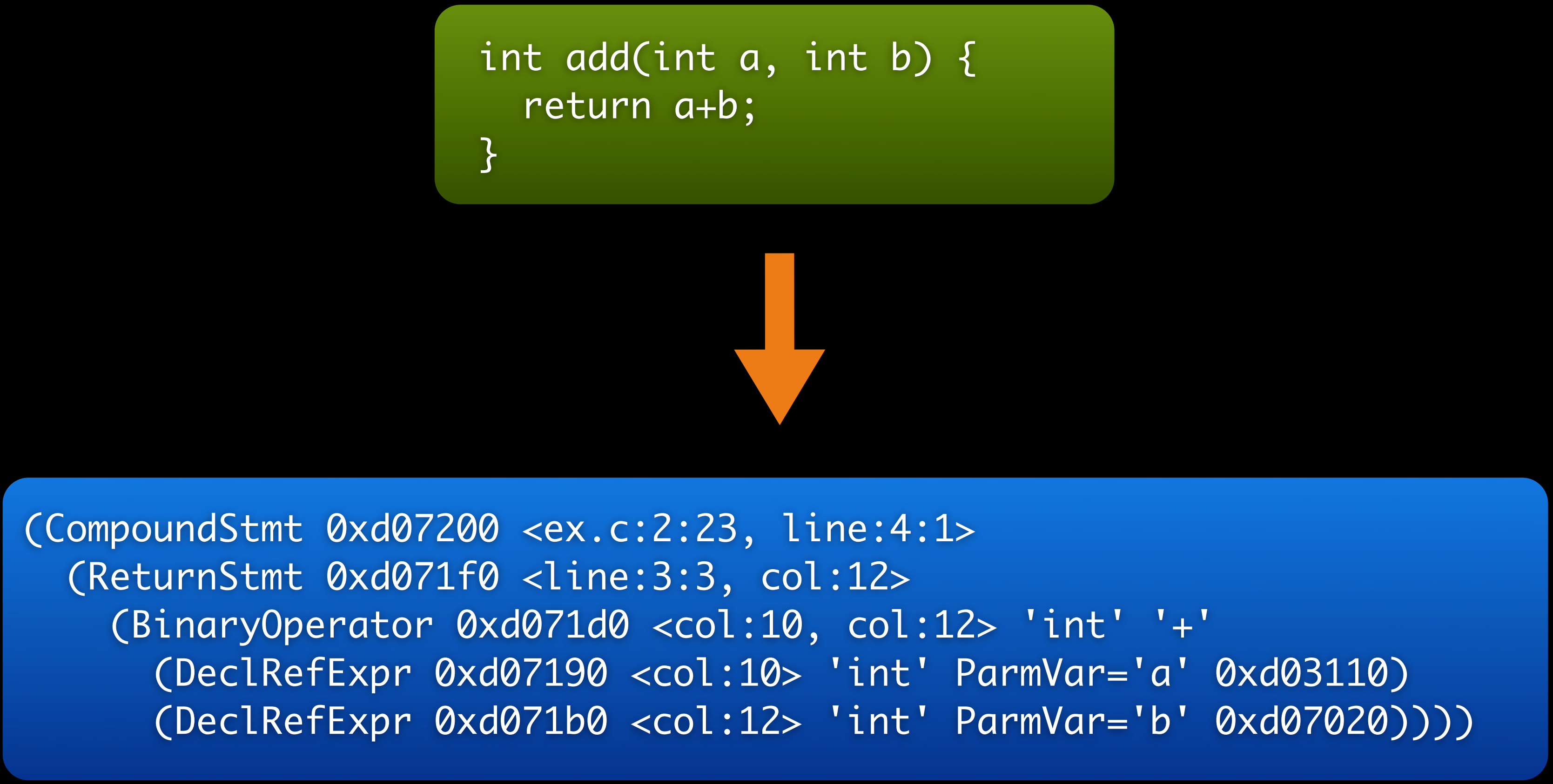
AST Anatomy

```
  // Iterators
  virtual child_iterator child_begin();
  virtual child_iterator child_end();

  // Serialization
  virtual void EmitImpl(llvm::Serializer& S) const;
  static IntegerLiteral* CreateImpl(llvm::Deserializer& D, ASTContext& C);

  // ‘isa’ dynamic type support
  static bool classof(const Stmt *T)
    { return T->getStmtClass() == IntegerLiteralClass; }
  static bool classof(const IntegerLiteral *)
    { return true; }
};
```

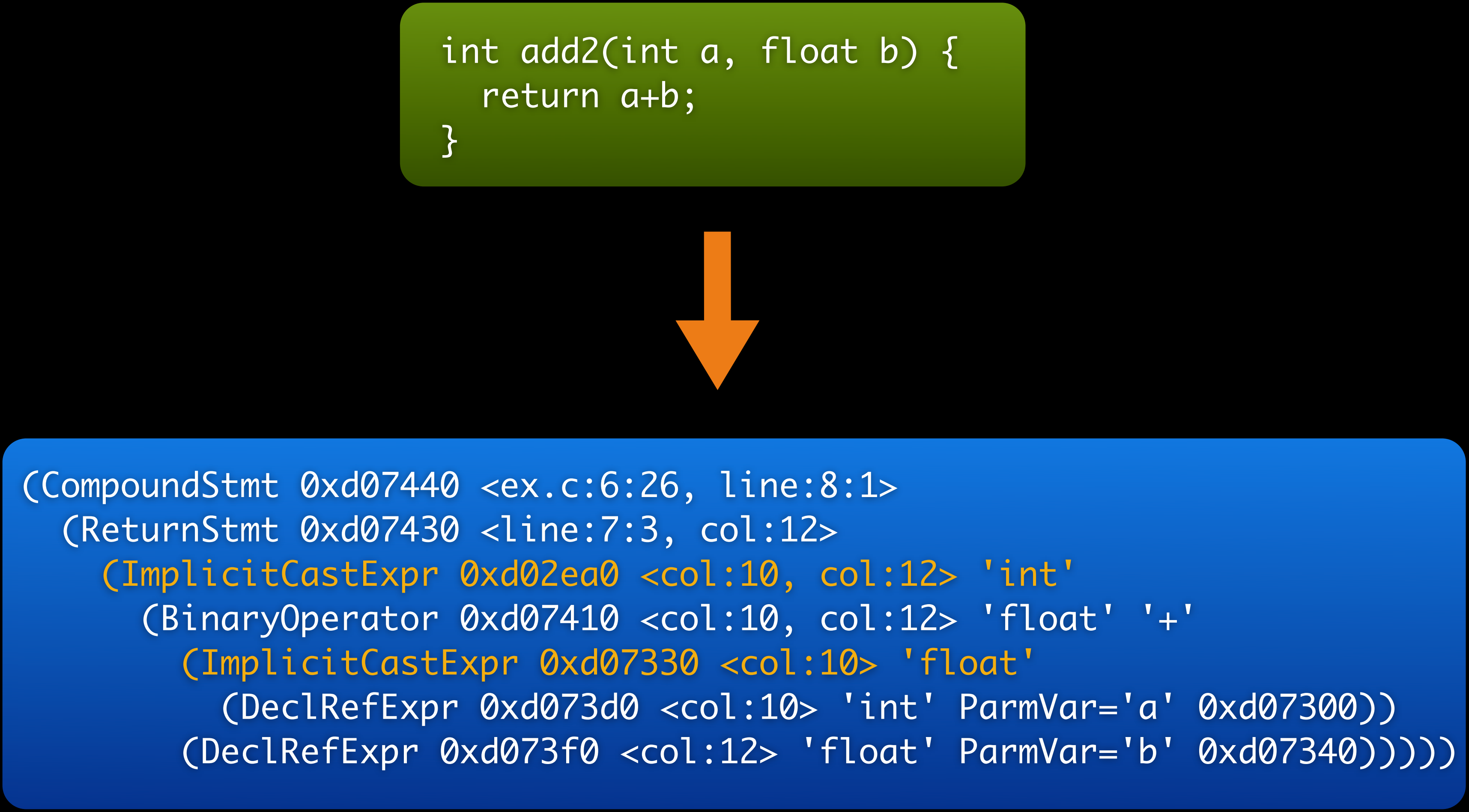
clang -ast-dump



```
int add(int a, int b) {
  return a+b;
}
```

```
(CompoundStmt 0xd07200 <ex.c:2:23, line:4:1>
  (ReturnStmt 0xd071f0 <line:3:3, col:12>
    (BinaryOperator 0xd071d0 <col:10, col:12> 'int' '+'
      (DeclRefExpr 0xd07190 <col:10> 'int' ParmVar='a' 0xd03110)
      (DeclRefExpr 0xd071b0 <col:12> 'int' ParmVar='b' 0xd07020))))
```

clang -ast-dump



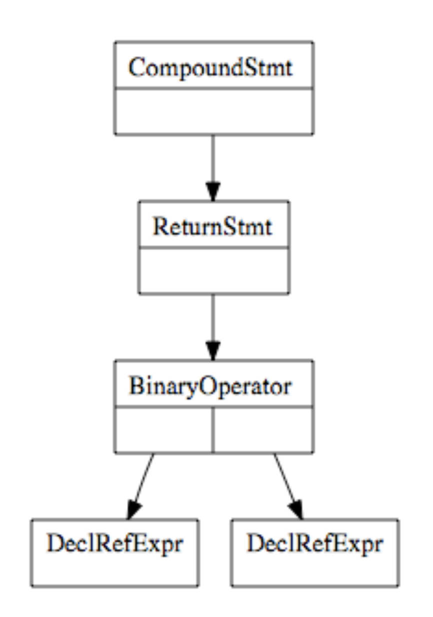
```
int add2(int a, float b) {
  return a+b;
}
```

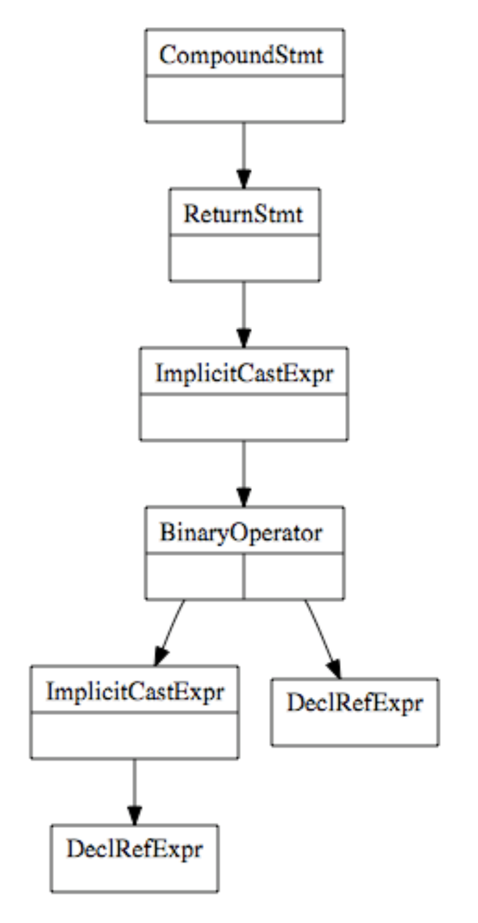
```
(CompoundStmt 0xd07440 <ex.c:6:26, line:8:1>
  (ReturnStmt 0xd07430 <line:7:3, col:12>
    (ImplicitCastExpr 0xd02ea0 <col:10, col:12> 'int'
      (BinaryOperator 0xd07410 <col:10, col:12> 'float' '+'
        (ImplicitCastExpr 0xd07330 <col:10> 'float'
          (DeclRefExpr 0xd073d0 <col:10> 'int' ParmVar='a' 0xd07300))
        (DeclRefExpr 0xd073f0 <col:12> 'float' ParmVar='b' 0xd07340)))))
```

clang -ast-view

```
int add(int a, int b) {
  return a+b;
}
```

```
int add2(int a, float b) {
  return a+b;
}
```





AST Traversal

```
void Example::HandleTopLevelDecl(Decl *D) { // ASTConsumer hook
  if (Stmt *Body = D->getBody())
    CallDumper(Body);
}

void Example::CallDumper(Stmt *S) {
  // Visit all children.
  for (Stmt::child_iterator CI = S->child_begin(), E = S->child_end();
       CI != E; ++CI)
    if (*CI)
      CallDumper(*CI);

  // Dump all AST’s that represent C function calls.
  if (CallExpr *CE = dyn_cast<CallExpr>(S))
    CE->dump();
}
```

Extending clang

- Build a tool for C/ObjC:
  - Traverse AST’s, CFG’s
  - Use built-in Dataflow analysis
  - Subclass MinimalAction (if semantic analysis isn’t desired)

- Add a new language feature:
  - Hack lexer, parser, sema, codegen to add your extension

Status

- C/ObjC Parsing:
  - Mostly complete, missing details of VLAs, minor GNU extensions
  - Can parse huge source bases without problems
- C/ObjC Code Generation:
  - Can compile many simple C apps with LLVM codegen
  - ObjC supports GNU ObjC runtime
  - Aiming for solid C/ObjC support in 2009
- C++ Parsing:
  - Can parse namespaces, classes, inline functions
  - Much is still missing, but we’re making progress
  - See http://clang.llvm.org/cxx_status.html

Related Projects: Help!

- Debugger support
- New “libgcc”
- New standard headers
  - float.h
  - xmmintrin.h
  - …
- Compiler driver: llvmc2?

Questions?