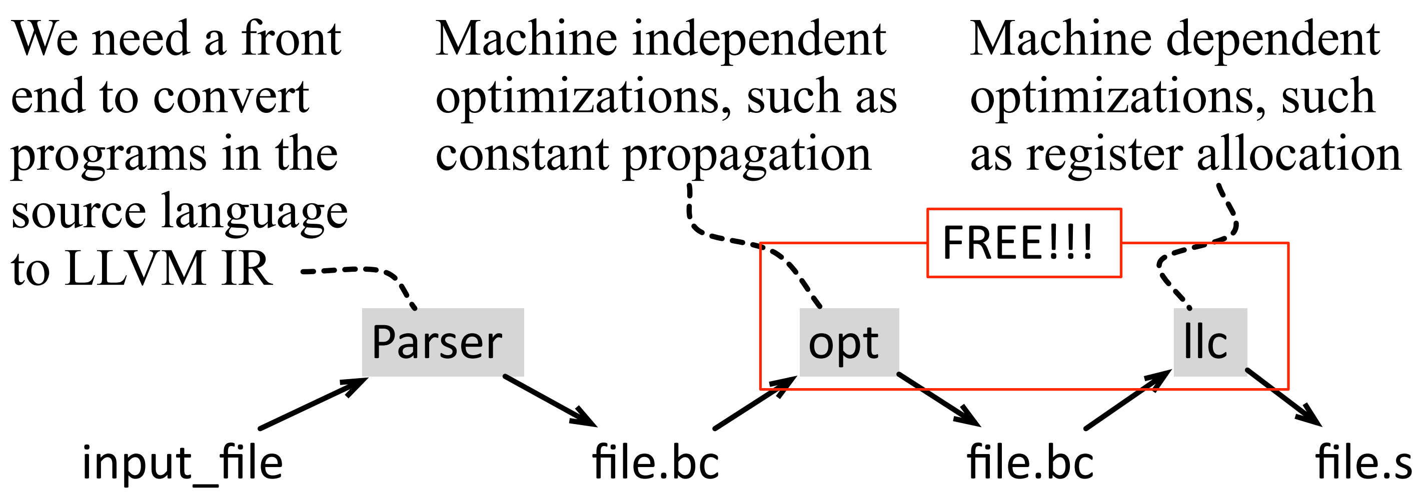
COMPILING A LANGUAGE

DCC 888

Dealing with Programming Languages

- LLVM gives developers many tools to interpret or compile a language:
  - The intermediate representation
  - Lots of analyses and optimizations
- We can work on a language that already exists, e.g., C, C++, Java, etc
- We can design our own language.

When is it worth designing a new language?



The Simple Calculator

- To illustrate this capacity of LLVM, let's design a very simple programming language:
  - A program is a function application
  - A function contains only one argument x
  - Only the integer type exists
  - The function body contains only additions, multiplications, references to x, and integer constants in polish notation:

$$(\lambda x\ .\ \times\ x\ x)\ 4 = 16$$

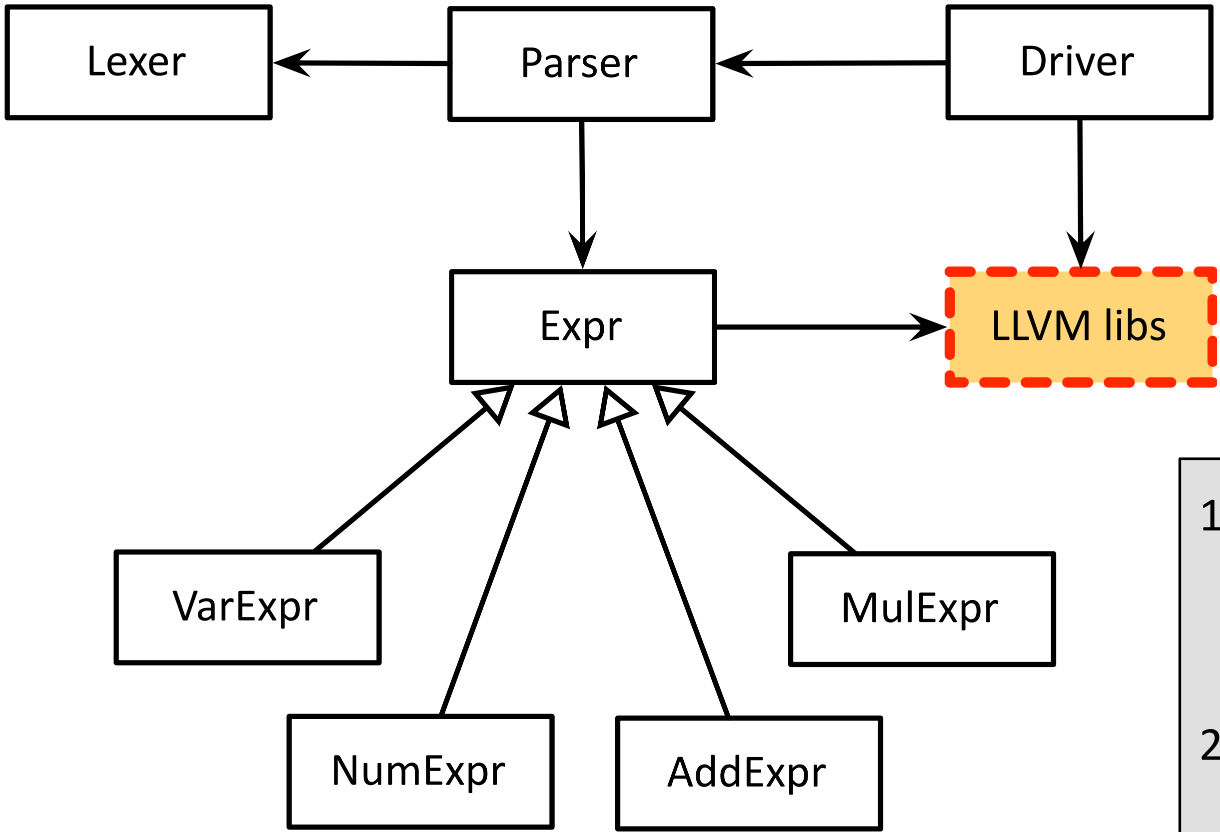
$$(\lambda x\ .\ +\ x\ \times\ x\ 2)\ 4 = 12$$

$$(\lambda x\ .\ \times\ x\ +\ x\ 2)\ 4 = 24$$

1) Can you understand why we got each of these values?

2) How is the grammar of our language?

The Architecture of Our Compiler



1) Can you guess the meaning of the different arrows?
2) Can you guess the role of each class?
3) What would be a good execution mode for our system?

The Execution Engine

```
$> ./driver 4
* x x
Result: 16

$> ./driver 4
+ x * x 2
Result: 12

$> ./driver 4
* x + x 2
Result: 24
```

Our execution engine parses the expression, converts it to a function written in LLVM IR, JIT compiles this function, and runs it with the **argument** passed to the program in command line.

```
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %addtmp = add i32 %x, 2
  %multmp = mul i32 %x, %addtmp
  ret i32 %multmp
}
```

Let's start with our lexer. Which tokens do we have?

Lexer.h

The Lexer

- A lexer is a program that divides a string of characters into tokens.
  - A token is a terminal in our grammar, e.g., a symbol that is part of the alphabet of our language.
  - Lexers can be easily implemented as finite automata.

```
#ifndef LEXER_H
#define LEXER_H
#include <string>
class Lexer {
 public:
   std::string getToken();
   Lexer() : lastChar(' ') {}
 private:
   char lastChar;
   inline char getNextChar() {
     char c = lastChar;
     lastChar = getchar();
     return c;
   }
};
#endif
```

1) Again: which kind of tokens do we have?

2) Can you guess the implementation of the getToken() method?

Implementation of the Lexer

```
#include "Lexer.h"
std::string Lexer::getToken() {
  while (isspace(lastChar)) { lastChar = getchar(); }
  if (isalpha(lastChar)) {
    std::string idStr;
    do { idStr += getNextChar(); } while (isalnum(lastChar));
    return idStr;
  } else if (isdigit(lastChar)) {
    std::string numStr;
    do { numStr += getNextChar(); } while (isdigit(lastChar));
    return numStr;
  } else if (lastChar == EOF) {
    return "";
  } else {
    std::string operatorStr;
    operatorStr = getNextChar();
    return operatorStr;
  }
}
```

1) Would you be able to represent this lexer as a state machine?

2) We must now define the parser. How can we implement it?

Parsing

- Parsing is the act to transform a string of tokens in a syntax tree♤.

```
#ifndef PARSER_H
#define PARSER_H

#include <string>
class Expr;
class Lexer;

class Parser {
  public:
    Parser(Lexer* argLexer) : lexer(argLexer) {}
    Expr* parseExpr();
  private:
    Lexer* lexer;
};

#endif
```

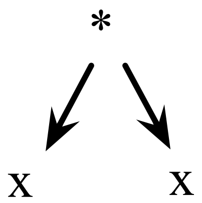
1) What are these forward declarations good for?
2) Do you understand this syntax?
3) What does the parser return?

♤: it used to be one of the most important problems in computer science.

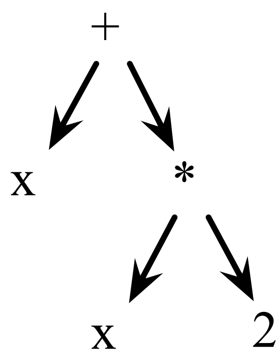
Syntax Trees

- The parser produces syntax trees.

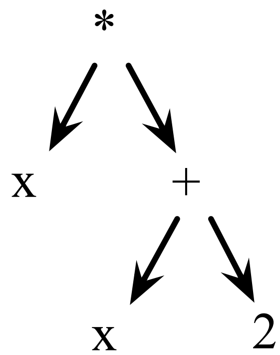
* x x



\+ x * x 2



* x + x 2



How can we implement these trees in C++?

The Nodes of the Tree

```
#ifndef AST_H
#define AST_H

#include "llvm/IR/IRBuilder.h"

class Expr {
  public:
    virtual ~Expr() {}
    virtual llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const = 0;
};

class NumExpr : public Expr {
  public:
    NumExpr(int argNum) : num(argNum) {}
    llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const;
    static const unsigned int SIZE_INT = 32;
  private:
    const int num;
};

class VarExpr : public Expr {
  public:
    llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const;
    static llvm::Value* varValue;
};

class AddExpr : public Expr {
  public:
    AddExpr(Expr* op1Arg, Expr* op2Arg) :
            op1(op1Arg), op2(op2Arg) {}
    llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const;
  private:
    const Expr* op1;
    const Expr* op2;
};

class MulExpr : public Expr {
  public:
    MulExpr(Expr* op1Arg, Expr* op2Arg) :
            op1(op1Arg), op2(op2Arg) {}
    llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const;
  private:
    const Expr* op1;
    const Expr* op2;
};

#endif
```

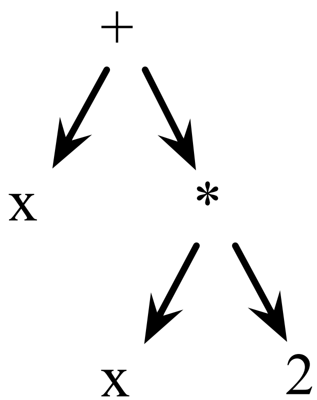
There is a **gen** method that is a bit weird. We shall look into it later.

Going Back into the Parser

- Our parser will build a syntax tree.

+ x * x 2 → Parser →

```
new AddExpr(
   new VarExp(),
   new MulExpr(
      new VarExpr(),
      new NumExpr(2)
   )
)
```



Jan Łukasiewicz, father of the Polish notation

The polish notation really simplifies parsing. We already have the tree, and without parentheses!

So, how can we implement our parser?

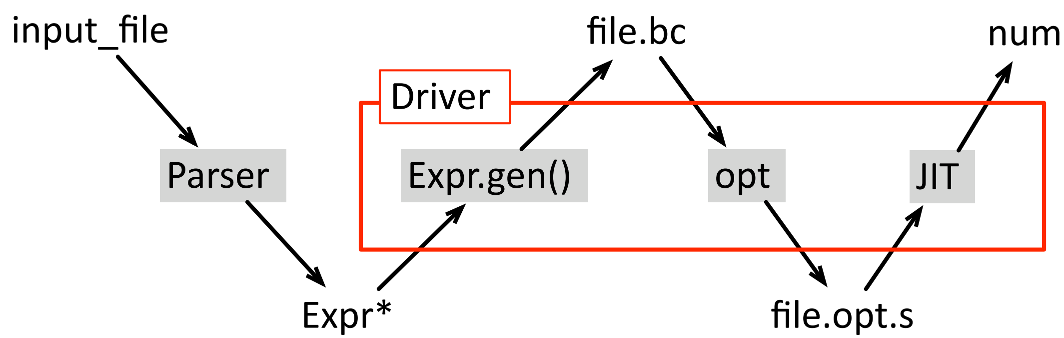
The Parser's Implementation

```
Expr* Parser::parseExpr() {
 std::string tk = lexer->getToken();
 if (tk == "") {
  return NULL;
 } else if (isdigit(tk[0])) {
  return new NumExpr(atoi(tk.c_str()));
 } else if (tk[0] == 'x') {
  return new VarExpr();
 } else if (tk[0] == '+') {
  Expr *op1 = parseExpr();
  Expr *op2 = parseExpr();
  return new AddExpr(op1, op2);
 } else if (tk[0] == '*') {
  Expr *op1 = parseExpr();
  Expr *op2 = parseExpr();
  return new MulExpr(op1, op2);
 } else {
  return NULL;
 }
}
```

```
#include "Expr.h"
#include "Lexer.h"
#include "Parser.h"
```

1) Why checking the first character of each token is already enough to avoid any ambiguity?

2) Now we need a way to translate trees into LLVM IR. How to do it?



The Translator

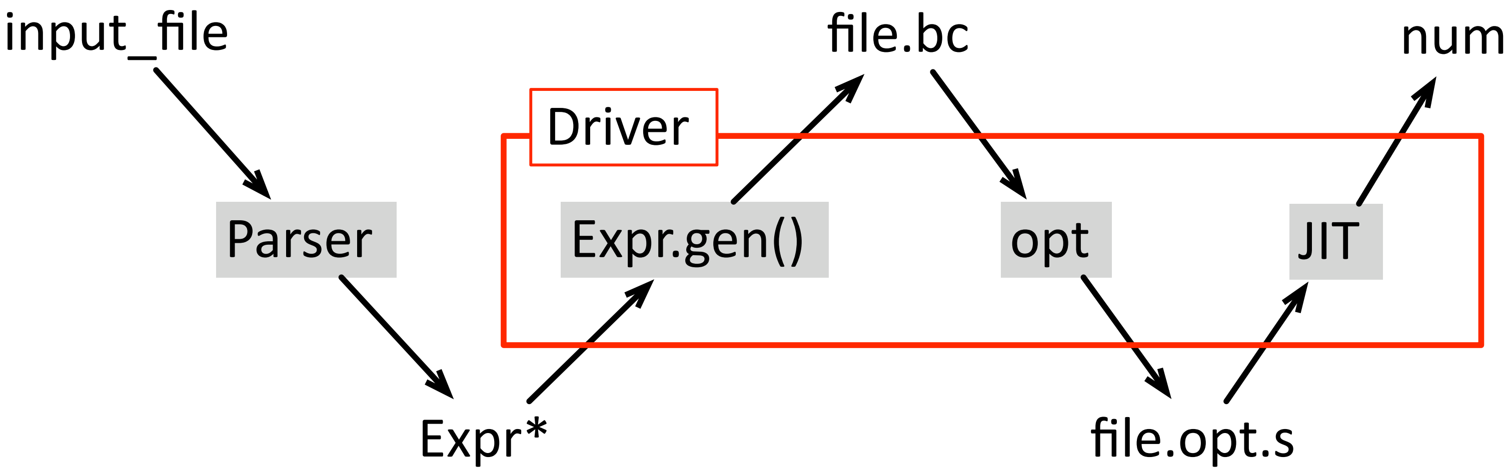
```
#include "Expr.h"

llvm::Value* VarExpr::varValue = NULL;

llvm::Value* NumExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  return llvm::ConstantInt::get
    (llvm::Type::getInt32Ty(context), num);
}
llvm::Value* VarExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  llvm::Value* var = VarExpr::varValue;
  return var ? var : NULL;
}
llvm::Value* AddExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  llvm::Value* v1 = op1->gen(builder, context);
  llvm::Value* v2 = op2->gen(builder, context);
  return builder->CreateAdd(v1, v2, "addtmp");
}
llvm::Value* MulExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  llvm::Value* v1 = op1->gen(builder, context);
  llvm::Value* v2 = op2->gen(builder, context);
  return builder->CreateMul(v1, v2, "multmp");
}
```

Our implementation has a small hack: our language has only one variable, which we have decided to call 'x'. This variable must be represented by an LLVM value, which is the argument of the function that we will create. Thus, we need a way to inform the translator this value. We do it through a static variable **varValue**. That is the only static variable that we are using in this class.

The Driver's Skeleton

```
int main(int argc, char** argv) {
  if (argc != 2) {
    llvm::errs() << "Inform an argument to your expression.\n";
    return 1;
  } else {
    llvm::LLVMContext context;
    llvm::Module *module = new llvm::Module("Example", context);
    llvm::Function *function = createEntryFunction(module, context);
    module->dump();
    llvm::ExecutionEngine* engine = createEngine(module);
    JIT(engine, function, atoi(argv[1]));
  }
}
```

The procedure that creates an LLVM function is not that complicated. Can you guess its implementation?



Creating an LLVM Function

```
llvm::Function *createEntryFunction(
  llvm::Module *module,
  llvm::LLVMContext &context) {
 llvm::Function *function =
   llvm::cast<llvm::Function>(
     module->getOrInsertFunction("fun", llvm::Type::getInt32Ty(context),
      llvm::Type::getInt32Ty(context), (llvm::Type *)0)
     );
 llvm::BasicBlock *bb = llvm::BasicBlock::Create(context, "entry", function);
 llvm::IRBuilder<> builder(context);
 builder.SetInsertPoint(bb);
 llvm::Argument *argX = function->arg_begin();
 argX->setName("x");
 VarExpr::varValue = argX;
 Lexer lexer;
 Parser parser(&lexer);
 Expr* expr = parser.parseExpr();
 llvm::Value* retVal = expr->gen(&builder, context);
 builder.CreateRet(retVal);
 return function;
}
```

This code is not "that" complicated, but it is not super straightforward either, so we will go a bit more carefully over it.

Let's start with **this** humongous call. What do you think it is doing?

Creating an LLVM Function

```
llvm::Function *createEntryFunction(
    llvm::Module *module,
    llvm::LLVMContext &context) {
  llvm::Function *function =
    llvm::cast<llvm::Function>(
        module->getOrInsertFunction("fun", llvm::Type::getInt32Ty(context),
          llvm::Type::getInt32Ty(context), (llvm::Type *)0)
        );
  llvm::BasicBlock *bb = llvm::BasicBlock::Create(context, "entry", function);
  llvm::IRBuilder<> builder(context);
  builder.SetInsertPoint(bb);
  llvm::Argument *argX = function->arg_begin();
  argX->setName("x");
  VarExpr::varValue = argX;
  Lexer lexer;
  Parser parser(&lexer);
  Expr* expr = parser.parseExpr();
  llvm::Value* retVal = expr->gen(&builder, context);
  builder.CreateRet(retVal);
  return function;
}
```

Here we are creating a function called "fun" that returns an integer, and receives an integer as a parameter. This cast has a variable number of arguments, and so we use a sentinel, e.g., NULL, to indicate the end of the list of arguments.

And **here**, what are we doing?

Creating the Body of the Function

```
llvm::Function *createEntryFunction(
   llvm::Module *module,
   llvm::LLVMContext &context) {
 llvm::Function *function =
   llvm::cast<llvm::Function>(
      module->getOrInsertFunction("fun", llvm::Type::getInt32Ty(context),
       llvm::Type::getInt32Ty(context), (llvm::Type *)0)
      );
 llvm::BasicBlock *bb = llvm::BasicBlock::Create(context, "entry", function);
 llvm::IRBuilder<> builder(context);
 builder.SetInsertPoint(bb);
 llvm::Argument *argX = function->arg_begin();
 argX->setName("x");
 VarExpr::varValue = argX;
 Lexer lexer;
 Parser parser(&lexer);
 Expr* expr = parser.parseExpr();
 llvm::Value* retVal = expr->gen(&builder, context);
 builder.CreateRet(retVal);
 return function;
}
```

This code creates a basic block, where we will insert instructions. We are attaching this block to a IRBuilder. This object is an LLVM helper to create new instructions.

1) Before we move on, do you remember what is a basic block?

2) And **this** code sequence here, what is it doing? That is a consequence of our hack...

Going Back to the Hack

Expr.h:

```
class VarExpr : public Expr {
  public:
    llvm::Value *gen(llvm::IRBuilder<> *builder,
            llvm::LLVMContext& con) const;
    static llvm::Value* varValue;
};
```

Again: our hack is a way to return an evaluation to a variable. Our language only has one variable, and its value never changes. This variable is the argument of the function that we are creating. We set its value upon creating this argument.

Expr.cpp:

```
llvm::Value* VarExpr::varValue = NULL;
llvm::Value* VarExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  llvm::Value* var = VarExpr::varValue;
  return var ? var : NULL;
}
```

Driver.cpp:

```
llvm::Argument *argX = function->arg_begin();
argX->setName("x");
VarExpr::varValue = argX;
```

Driver.cpp

A Few Final Remarks on Function Creation

```
llvm::Function *createEntryFunction(
    llvm::Module *module,
    llvm::LLVMContext &context) {
  llvm::Function *function =
    llvm::cast<llvm::Function>(
        module->getOrInsertFunction("fun", llvm::Type::getInt32Ty(context),
          llvm::Type::getInt32Ty(context), (llvm::Type *)0)
        );
  llvm::BasicBlock *bb = llvm::BasicBlock::Create(context, "entry", function);
  llvm::IRBuilder<> builder(context);
  builder.SetInsertPoint(bb);
  llvm::Argument *argX = function->arg_begin();
  argX->setName("x");
  VarExpr::varValue = argX;
  Lexer lexer;
  Parser parser(&lexer);
  Expr* expr = parser.parseExpr();
  llvm::Value* retVal = expr->gen(&builder, context);
  builder.CreateRet(retVal);
  return function;
}
```

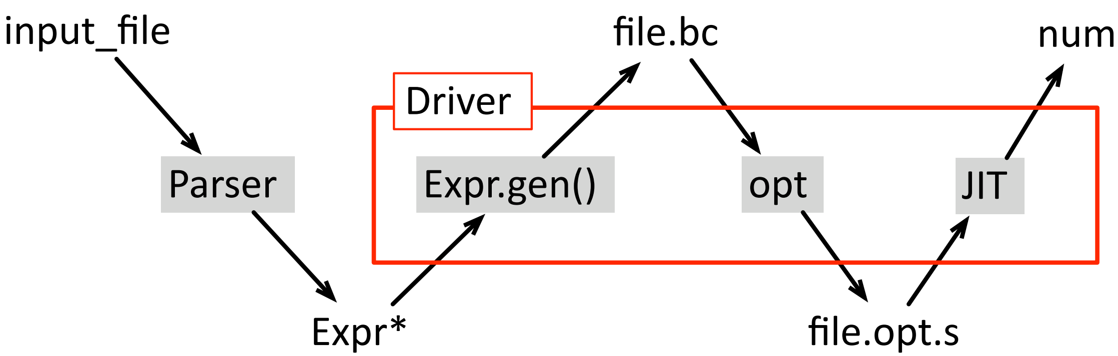
1) Easy one: what are we doing **here**?

2) And what are we doing in **this code snippet**?

Now, the JIT

```
int main(int argc, char** argv) {
  if (argc != 2) {
    llvm::errs() << "Inform an argument to your expression.\n";
    return 1;
  } else {
    llvm::LLVMContext context;
    llvm::Module *module = new llvm::Module("Example", context);
    llvm::Function *function = createEntryFunction(module, context);
    module->dump();
    llvm::ExecutionEngine* engine = createEngine(module);
    JIT(engine, function, atoi(argv[1]));
  }
}
```

What do you think the method **createEngine** is doing?



Now, we need a way to execute programs. We can interpret these programs, using lli, a tool that comes in the LLVM distro. If a JIT compiler is available for your architecture (usually it is), then we can JIT compile the code, as we will show next.

Creating an Engine to Execute Programs

- Engine is how we call the program that is in charge of executing other programs, e.g., the JavaScript engine in the Firefox browser, the C# engine in .NET, etc

```
llvm::ExecutionEngine* createEngine(llvm::Module *module) {
  llvm::InitializeNativeTarget();
  std::string errStr;
  llvm::ExecutionEngine *engine =
    llvm::EngineBuilder(module)
    .setErrorStr(&errStr)
    .setEngineKind(llvm::EngineKind::JIT)
    .create();
  if (!engine) {
    llvm::errs() << "Failed to construct ExecutionEngine: " << errStr << "\n";
  } else if (llvm::verifyModule(*module)) {
    llvm::errs() << "Error constructing function!\n";
  }
  return engine;
}
```

These are the sequence of method calls necessary to create a JIT engine. This engine can, later, receive a function, and execute it.

Invoking the JIT

Invoking the engine over a function is very easy. We just need a bit of setup to pass arguments to this function. After the JIT is done executing the function, we have the function's return value, which we can use as we wish.

```
void JIT(llvm::ExecutionEngine* engine, llvm::Function* function, int arg) {
  std::vector<llvm::GenericValue> Args(1);
  Args[0].IntVal = llvm::APInt(32, arg);
  llvm::GenericValue retVal = engine->runFunction(function, Args);
  llvm::outs() << "Result: " << retVal.IntVal << "\n";
}
```

Can you identify the code that sets the arguments up, and the code that gets the return value back?

Compiling Everything

- We can compile these programs using the LLVM standard Makefile.
- In fact, LLVM comes with a folder, "examples", which we can use to build our application:

```
~$ cd Programs/llvm/examples/DCC888/

~/Programs/llvm/examples/DCC888$ make
llvm[0]: Compiling Driver.cpp for Debug+Asserts build
llvm[0]: Compiling Expr.cpp for Debug+Asserts build
llvm[0]: Compiling Lexer.cpp for Debug+Asserts build
llvm[0]: Compiling Parser.cpp for Debug+Asserts build
llvm[0]: Linking Debug+Asserts executable driver
ld warning: ...
llvm[0]: ======= Finished Linking Debug+Asserts Executable driver

~/Programs/llvm/examples/DCC888$ cd ../../Debug+Asserts/examples/

~/Programs/llvm/Debug+Asserts/examples$ ./driver 4
* x 3
Result: 12
```

Using the standard Makefile makes it easy to link our code with all the LLVM libraries.

Quick Look in our Makefile

```
LEVEL = ../..
TOOLNAME = driver
EXAMPLE_TOOL = 1

# Link in JIT support
LINK_COMPONENTS := jit interpreter nativecodegen

include $(LEVEL)/Makefile.common
```

We can specify the **name of the executable** that we shall be creating, and we can point out which **libraries** will be necessary to compile our program.

Running

Example 1:

```
$> ./driver 4
* 3 + x * 5 + x 1

; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %addtmp = add i32 %x, 1
  %multmp = mul i32 5, %addtmp
  %addtmp1 = add i32 %x, %multmp
  %multmp2 = mul i32 3, %addtmp1
  ret i32 %multmp2
}

Result: 87
```

Example 2:

```
* x + * x 4 + * x 3 + x + * x x * 3 x

; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %multmp = mul i32 %x, 4
  %multmp1 = mul i32 %x, 3
  %multmp2 = mul i32 %x, %x
  %multmp3 = mul i32 3, %x
  %addtmp = add i32 %multmp2, %multmp3
  %addtmp4 = add i32 %x, %addtmp
  %addtmp5 = add i32 %multmp1, %addtmp4
  %addtmp6 = add i32 %multmp, %addtmp5
  %multmp7 = mul i32 %x, %addtmp6
  ret i32 %multmp7
}

Result: 240
```

Can you draw these two syntax trees?

Optimizing the Programs

- One of the nice things of LLVM is that it comes with many optimizations, which we can apply on its intermediate representation.

As an example, if our input program has only constants, LLVM folds all of them into a single value:

```
./driver 4
+ 3 * 4 + 5 6

; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  ret i32 47
}

Result: 47
```

1) How do you think this optimization works?

2) Where do you think this optimization is implemented?

3) And what about **this** program below: will LLVM optimize it?

```
./driver 4
+ * x 3 * x 3
```

The Need for Global Optimizations

Constant folding is implemented by the IRBuilder class. This is a local optimization. In other words, this optimization can only look into the parameters of the instruction that will be constructed. Naturally, this is not enough to catch, for instance, the redundancy between the two occurrences of "* x 3" in our example.

```
+ * x 3 * x 3

; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %multmp = mul i32 %x, 3
  %multmp1 = mul i32 %x, 3
  %addtmp = add i32 %multmp, %multmp1
  ret i32 %addtmp
}
```

1) Which compiler optimizations do you know?

2) How could we optimize the program on the left?

3) Which optimizations do you think the compiler could use to optimize this program?

```
llvm::Value* AddExpr::gen
(llvm::IRBuilder<> *builder, llvm::LLVMContext &context) const {
  llvm::Value* v1 = op1->gen(builder, context);
  llvm::Value* v2 = op2->gen(builder, context);
  return builder->CreateAdd(v1, v2, "addtmp");
}
```

The LLVM Tool Belt

1) Can you guess what each of these optimizations will do?

2) How do we use this new method?

```
void optimizeFunction(
  llvm::ExecutionEngine* engine,
  llvm::Module *module,
  llvm::Function* function
) {
  llvm::FunctionPassManager passManager(module);
  passManager.add(new llvm::DataLayout(*engine->getDataLayout()));
  passManager.add(llvm::createInstructionCombiningPass());
  passManager.add(llvm::createReassociatePass());
  passManager.add(llvm::createGVNPass());
  passManager.add(llvm::createCFGSimplificationPass());
  passManager.doInitialization();
  passManager.run(*function);
}
```

Better not to forget:

```
#include "llvm/Analysis/Passes.h"
#include "llvm/PassManager.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/Transforms/Scalar.h"
```

Driver.cpp

The New Driver

```
int main(int argc, char** argv) {
  if (argc != 2) {
    llvm::errs() << "Inform an argument to your expression.\n";
    return 1;
  } else {
    llvm::LLVMContext context;
    llvm::Module *module = new llvm::Module("Example", context);
    llvm::Function *function = createEntryFunction(module, context);
    llvm::errs() << "Module before optimizations:\n";
    module->dump();
    llvm::errs() << "Module after optimizations:\n";
    llvm::ExecutionEngine* engine = createEngine(module);
    optimizeFunction(engine, module, function);
    module->dump();
    JIT(engine, function, atoi(argv[1]));
  }
}
```

Just for fun, we are printing the function before and after we run the optimizations.

The Optimizations in Action

```
+ * x 3 * x 3

Module before optimizations:

; ModuleID = 'Example'
define i32 @fun(i32 %x) {
entry:
  %multmp = mul i32 %x, 3
  %multmp1 = mul i32 %x, 3
  %addtmp = add i32 %multmp, %multmp1
  ret i32 %addtmp
}

Module after optimizations:

; ModuleID = 'Example'
define i32 @fun(i32 %x) {
entry:
  %addtmp = mul i32 %x, 6
  ret i32 %addtmp
}

Result: 24
```

The optimized program has only one arithmetic instruction, whereas the original program had three such operations.

Different programming languages may require different kinds of optimizations. Can you think about optimizations that are specific to particular languages?

Final Remarks

- LLVM gives programmers several tools to build their programming languages:
  - Nice intermediate representation
  - Several optimizations
  - Several back-ends