Code generation
Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
Motivation
Intermediate code (representation; IR) facilitates retargeting (addressing a new back end) and machine-independent optimization.
Position of a Code Generator in the Compiler Model

- **Front-End**
- **Code Optimizer**
- **Code Generator**

Source program → Intermediate code → Intermediate code → Target program

Lexical error
Syntax error
Semantic error

Symbol Table
Different options for target code

- Absolute machine code (directly executable code)
- Relocatable machine code (subject to linking step)
- Assembly code (which is suitable for debugging)
- Bytecode (P code, intermediate representation)

The focus in this course
Obvious requirements

- Intermediate and final code must be correct.
  - That is, code generation must be semantics-preserving.
- Code must be fast and resource-friendly.
  - We need a cost model of the code languages.
  - We may need weights in code generation.
  - We may use optimizations to reduce costs.
Intermediate representations targeted by code generation

- **Graphical representations** (e.g. AST)
- **Postfix notation**: operations on values stored on operand stack (similar to JVM bytecode)
- **Three-address code**: (e.g. *triples* and *quads*)
  \[ x := y \text{ op } z \]
- **Two-address code**:
  \[ x := \text{ op } y \]
  which is the same as \[ x := x \text{ op } y \]
Postfix Notation

\[ a := b \times -c + b \times -c \]

\[ \text{a b c uminus } \times \text{ b c uminus } \times + \text{ assign} \]

Postfix notation represents operations on a stack

Pro: easy to generate
Cons: stack operations are more difficult to optimize

Bytecode (for example)

```
iload 2 // push b
iload 3 // push c
ineg // uminus
imul // *
iload 2 // push b
iload 3 // push c
ineg // uminus
imul // *
iadd // +
istore 1 // store a
```
Three-Address Code

\[ a := b \times -c + b \times -c \]

\[
\begin{align*}
t1 &= -c \\
t2 &= b \times t1 \\
t3 &= -c \\
t4 &= b \times t3 \\
t5 &= t2 + t4 \\
a &= t5
\end{align*}
\]

Linearized representation of a syntax tree

Linearized representation of a syntax DAG
Big questions of code generation

- What is the “language” of intermediate representations?
- How “context-free” is the language?
- What is the distance between high-level and IR language?
- How to define a mapping between the languages?
- What sort of optimizations are possible on IR?
- What backends are available for IR?
- ...

Syntax-directed translation

A typical means of mapping higher-level to lower-level code
Three-Address Statements

- Assignment statements: $x := y \ op \ z$, $x := op \ y$
- Indexed assignments: $x := y[i]$, $x[i] := y$
- Pointer assignments: $x := &y$, $x := *y$, $*x := y$
- Copy statements: $x := y$
- Unconditional jumps: goto lab
- Conditional jumps: if $x \ relop \ y$ goto lab
- Function calls: param $x$… call $p$, $n$
  return $y$

Various such “P code machines” exist …
Syntax-Directed Translation into Three-Address Code

Productions

\[
S \rightarrow \text{id} := E \\
| \text{while } E \text{ do } S \\
E \rightarrow E + E \\
| E \times E \\
| - E \\
| (E) \\
| \text{id} \\
| \text{num}
\]

Synthesized attributes:

\[
\begin{align*}
S.\text{code} & \quad \text{three-address code for } S \\
S.\text{begin} & \quad \text{label to start of } S \text{ or nil} \\
S.\text{after} & \quad \text{label to end of } S \text{ or nil} \\
E.\text{code} & \quad \text{three-address code for } E \\
E.\text{place} & \quad \text{a name holding the value of } E
\end{align*}
\]

Code generation

\[
t3 := t1 + t2
\]
## Syntax-Directed Translation into Three-Address Code (cont’d)

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} := E$</td>
<td>$S$.code := $E$.code $</td>
</tr>
<tr>
<td>$S \rightarrow \text{while } E \text{ do } S_1$</td>
<td>(see next slide)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E$.place := newtemp(); $E$.code := $E_1$.code $</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \ast E_2$</td>
<td>$E$.place := newtemp(); $E$.code := $E_1$.code $</td>
</tr>
<tr>
<td>$E \rightarrow -E_1$</td>
<td>$E$.place := newtemp(); $E$.code := $E_1$.code $</td>
</tr>
<tr>
<td>$E \rightarrow (E_1)$</td>
<td>$E$.place := $E_1$.place $</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E$.place := $\text{id}$.name $</td>
</tr>
<tr>
<td>$E \rightarrow \text{num}$</td>
<td>$E$.place := newtemp(); $E$.code := gen($E$.place ‘:=’ $\text{num}$.value)</td>
</tr>
</tbody>
</table>
Production

\[ S \to \textbf{while} \ E \ \textbf{do} \ S_1 \]

Semantic rule

\[
\begin{align*}
S.\text{begin} & := \text{newlabel}() \\
S.\text{after} & := \text{newlabel}() \\
S.\text{code} & := \text{gen}(S.\text{begin} \ ':') \parallel \\
& \quad E.\text{code} \parallel \\
& \quad \text{gen}(\text{‘if’} \ E.\text{place} \text{‘=} \text{‘0’} \text{‘goto’} \ S.\text{after}) \parallel \\
& \quad S_1.\text{code} \parallel \\
& \quad \text{gen}(\text{‘goto’} \ S.\text{begin}) \parallel \\
& \quad \text{gen}(S.\text{after} \ ':')
\end{align*}
\]

<table>
<thead>
<tr>
<th>S.begin:</th>
<th>E.code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textbf{if} \ E.\text{place} \text{=} \text{0} \textbf{goto} S.\text{after}</td>
</tr>
<tr>
<td>S.code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textbf{goto} S.\text{begin}</td>
</tr>
</tbody>
</table>

| S.after: | ... |

Example

\[ i := 2 \times n + k \]

while \( i \) do

\[ i := i - k \]

\[
\begin{align*}
  t1 & := 2 \\
  t2 & := t1 \times n \\
  t3 & := t2 + k \\
  i & := t3 \\
\end{align*}
\]

\[ \text{L1: if } i = 0 \text{ goto L2} \]
\[ t4 := i - k \]
\[ i := t4 \]
\[ \text{goto L1} \]

\[ \text{L2:} \]
Symbol Tables for Scoping

```c
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ ...
  swap(s.a, s.b);
  ...
}
```

We need a symbol table for the fields of struct S

Need symbol table for arguments and locals for each function

Need symbol table for global variables and functions

Check: s is global and has fields a and b

Using symbol tables we can generate code to access s and its fields
Additional aspects

- Register allocation
- Array operations
- Procedure calls
- Exceptions
- ...

No details provided on this matter. See any textbook on compiler construction. This would be another course.
Important kind of target for code generation
Examples of P code machines

- Java Virtual Machine
- MATLAB precompiled code
- The machine of UCSD Pascal
public static int gcd(int x, int y) {
    while (x != y) {
        if (x > y)
            x = x - y;
        else
            y = y - x;
    }
    return x;
}
Characteristics of P code

- Portability of code generation
- Simplicity of code generation
- Simplicity of interpretation
- Compact size of P code
- Debugging of P code
- Efficiency by JITing or hardware implementation
Variation points in P code machines

- The extent of using registers
- Available machine registers (such as SP)
- The amount of stacks
- Availability of heap
- Supported datatypes
- Calling conventions
- ...

A quick look at compiler optimization

This is a huge area making up for the bigger part of compiler construction which we cannot cover here in any useful way. Some context is provided for motivation.
Major classification

- Peephole optimization (pretty local)
- Optimizations requiring global program analysis
- Optimizations requiring IR in “normal” form
Peephole optimization

- Perform optimization over small window of code.
- Typical examples:
  - Constant folding (evaluate constant subexpressions)
  - Strength reduction (replace slow by fast operations)
  - Common sub-expression elimination (simple forms thereof)
  - Remove redundant operations
  - Algebraic transformations
  - Special case operations (inc vs. add 1)
  - ...

An example for strength reduction for Java Bytecode

```java
... 
aload 1
aload 1
mul
... 
``` 

```java
... 
aload 1
dup
mul
... 
``` 

An example for removing redundant operations

Source

\[
\begin{align*}
    a &= b + c; \\
    d &= a + e;
\end{align*}
\]

Bytecode

```plaintext
MOV b, R0  # Copy b to the register
ADD c, R0  # Add c to the register
MOV R0, a  # Copy the register to a
MOV a, R0  # Copy a to the register
ADD e, R0  # Add e to the register
ADD R0, d  # Copy the register to d
```

 Eliminate past regular generation

An example for removing redundant operations for Z80 :-)

PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL _ADDR1
POP HL
POP DE
POP BC
POP AF

PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL _ADDR1
POP HL
POP DE
POP BC
POP AF

PUSH AF
PUSH BC
PUSH DE
PUSH HL
CALL _ADDR1
POP HL
POP DE
POP BC
POP AF


An example for an algebraic transformation on three-address statements

\[
\begin{align*}
    t1 & := a - a \\
    t2 & := b + t1 \\
    t3 & := 2 \times t2
\end{align*}
\]

\[
\begin{align*}
    t1 & := 0 \\
    t2 & := b \\
    t3 & := t2 \ll 1
\end{align*}
\]

Bit shift being cheaper than multiplication with 2.
SSA form: a key concept in compiler optimization
Static single assignment form (SSA form)

A property of IR: each variable is assigned exactly once!

SSA enables or enhances compiler optimizations

Regular

\[
\begin{align*}
    y & := 1 \\
    y & := 2 \\
    x & := y
\end{align*}
\]

It requires some analysis to see that the first assignment is “dead” and that the third assignment relies on the second.

SSA

\[
\begin{align*}
    y_1 & := 1 \\
    y_2 & := 2 \\
    x_1 & := y_2
\end{align*}
\]

These facts are more obvious in SSA form.

SSA conversion

Replace the target of each assignment with a new variable.
Replace each reference with the right “version”.

Regular
\[
\begin{align*}
  y & := 1 \\
  y & := 2 \\
  x & := y
\end{align*}
\]

SSA
\[
\begin{align*}
  y_1 & := 1 \\
  y_2 & := 2 \\
  x_1 & := y_2
\end{align*}
\]
Optimizations benefiting from SSA

- constant propagation
- value range propagation
- sparse conditional constant propagation
- dead code elimination
- global value numbering
- partial redundancy elimination
- strength reduction
- register allocation

Control-flow graphs
Flow Graphs

- A flow graph is a graphical depiction of a sequence of instructions with control flow edges.

- A flow graph can be defined at the intermediate code level or target code level.

```plaintext
MOV 1,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```plaintext
MOV 0,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```
Basic Blocks

• A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)

```plaintext
MOV 1, R0
MOV n, R1
JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: JMPNZ R1, L1
```

```
MOV 1, R0
MOV n, R1
JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: JMPNZ R1, L1
```
Basic Blocks and Control Flow Graphs

- A control flow graph (CFG) is a directed graph with basic blocks $B_i$ as vertices and with edges $B_i \rightarrow B_j$ iff $B_j$ can be executed immediately after $B_i$. 

```
MOV 1,R0
MOV n,R1
JMP L2
L1: MUL 2,R0
SUB 1,R1
L2: JMPNZ R1,L1
L1: MUL 2,R0
SUB 1,R1
L2: JMPNZ R1,L1
```
Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block $B_1$ is a predecessor of $B_2$
  - Basic block $B_2$ is a successor of $B_1$

```
MOV 1,R0
MOV n,R1
JMP L2

L1: MUL 2,R0
    SUB 1,R1

L2: JMPNZ R1,L1
```
Partition Algorithm for Basic Blocks

*Input:* A sequence of three-address statements

*Output:* A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program
Loops

• A *loop* is a collection of basic blocks, such that
  – All blocks in the collection are *strongly connected*
  – The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry
Loops (Example)

Strongly connected components:

\[ \text{SCC}=\{\{B2,B3\},\{B4\}\} \]

Entries:

B3, B4
Transformations on Basic Blocks

- A code-improving transformation is a code optimization to improve speed or reduce code size.
- *Global transformations* are performed across basic blocks.
- *Local transformations* are only performed on single basic blocks.
- Transformations must be safe and preserve the meaning of the code.
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form.
Common-Subexpression Elimination

• Remove redundant computations

\[
\begin{align*}
a & := b + c \\
b & := a - d \\
c & := b + c \\
d & := a - d \\
\end{align*}
\]

\[
\begin{align*}
a & := b + c \\
b & := a - d \\
c & := b + c \\
d & := b \\
\end{align*}
\]

\[
\begin{align*}
t1 & := b \times c \\
t2 & := a - t1 \\
t3 & := b \times c \\
t4 & := t2 + t3 \\
\end{align*}
\]

\[
\begin{align*}
t1 & := b \times c \\
t2 & := a - t1 \\
t4 & := t2 + t1 \\
\end{align*}
\]
Dead Code Elimination

- Remove unused statements

\[
\begin{align*}
  b & := a + 1 \\
a & := b + c \\
  & \ldots \\
\end{align*}
\]

Assuming \( a \) is dead (not used)

\[
\begin{align*}
  \text{if true goto L2} \\
  b & := x + y \\
  & \ldots \\
\end{align*}
\]

Remove unreachable code
Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed

\[
\begin{align*}
  t_1 &:= b + c \\
  t_2 &:= a - t_1 \\
  t_2 &:= t_1 * d \\
  d &:= t_2 + t_1
\end{align*}
\]

Normal-form block
Interchange of Statements

• Independent statements can be reordered

\[
\begin{align*}
t_1 & := b + c \\
t_2 & := a - t_1 \\
t_3 & := t_1 * d \\
d & := t_2 + t_3 \\
\end{align*}
\]

Note that normal-form blocks permit all statement interchanges that are possible
Algebraic Transformations

• Change arithmetic operations to transform blocks to algebraic equivalent forms

\[
\begin{align*}
t1 & := a - a \\
t2 & := b + t1 \\
t3 & := 2 \times t2 \\
t1 & := 0 \\
t2 & := b \\
t3 & := t2 \ll 1
\end{align*}
\]
Alias analysis
Knowing whether or not two pointer variables alias each other may heavily influence optimizations.

There are three possible alias cases here:

1. The variables p and q cannot alias.
2. The variables p and q must alias.
3. It cannot be conclusively determined at compile time if p and q alias or not.

If p and q cannot alias, then \( i = p.foo + 3 \); can be changed to \( i = 4 \). If p and q must alias, then \( i = p.foo + 3 \); can be changed to \( i = 5 \). In both cases, we are able to perform optimizations from the alias knowledge. On the other hand, if it is not known if p and q alias or not, then no optimizations can be performed and the whole of the code must be executed to get the result. Two memory references are said to have a *may-alias* relation if their aliasing is unknown.

Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location.

Alias analysis techniques are usually classified by flow-sensitivity and context-sensitivity. They may determine may-alias or must-alias information. The term alias analysis is often used interchangeably with term points-to analysis, a specific case.

Concluding remarks

Basic concepts:
- P code, SSA, CFG, alias analysis

Typical optimizations:
- constant folding, common subexpression elimination

Compiler architecture:
- lexer, parser, AST, IR, optimization, code generation

Compiler implementation frameworks:
- LLVM
Code generation and intermediate representations in practice – The LLVM experience. (Overview and Demo)
LLVM overview

[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
LLVM Compiler System

• LLVM = Low Level Virtual Machine
• The LLVM Compiler Infrastructure
  – Provides reusable components for building compilers
  – Reduce the time/cost to build a new compiler
  – Build static compilers, JITs, trace-based optimizers, ...

• The LLVM Compiler Framework
  – End-to-end compilers using the LLVM infrastructure
  – C and C++ are robust and aggressive:
    • Java, Scheme and others are in development
  – Emit C code or native code for X86, Sparc, PowerPC
Three primary LLVM components

• The LLVM *Virtual Instruction Set*
  – The common language- and target-independent IR
  – Internal (IR) and external (persistent) representation

• A collection of well-integrated libraries
  – Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, ...

• A collection of tools built from the libraries
  – Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer, ...
The LLVM C/C++ Compiler

• From the high level, it is a standard compiler:
  – Compatible with standard makefiles
  – Uses GCC 4.2 C and C++ parser

  \[ \begin{align*}
  \text{C file} & \rightarrow \text{llvmgcc} \rightarrow .o \text{ file} \\
  \text{C++ file} & \rightarrow \text{llvmg++} \rightarrow .o \text{ file} \\
  \text{.o file} & \rightarrow \text{llvm linker} \rightarrow \text{executable}
  \end{align*} \]

<table>
<thead>
<tr>
<th>Compile Time</th>
<th>Link Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>llvmgcc</td>
<td>.o file</td>
</tr>
<tr>
<td>llvmg++</td>
<td>.o file</td>
</tr>
<tr>
<td>.o file</td>
<td>llvm linker</td>
</tr>
<tr>
<td>llvm linker</td>
<td>executable</td>
</tr>
</tbody>
</table>

• Distinguishing features:
  – Uses LLVM optimizers, not GCC optimizers
  – .o files contain LLVM IR/bytecode, not machine code
  – Executable can be bytecode (JIT’ed) or machine code

Chris Lattner—lattner@cs.uiuc.edu
The LLVM C/C++ Compiler (cont)

Standard compiler organization, which uses LLVM as midlevel IR:
- Language specific front-end lowers code to LLVM IR
- Language/target independent optimizers improve code
- Code generator converts LLVM code to target (e.g. IA64) code

Key LLVM Feature:
IR is small, simple, easy to understand, and is well defined

Chris Lattner – lattner@cs.uiuc.edu

[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
Running example: arg promotion

Consider use of by-reference parameters:

```
int callee(const int &X) {
    return X+1;
}
int caller() {
    return callee(4);
}
```

compiles to

```
int callee(const int *X) {
    return *X+1;  // memory load
}
int caller() {
    int tmp;    // stack object
    tmp = 4;    // memory store
    return callee(&tmp);
}
```

We want:

```
int callee(int X) {
    return X+1;
}
int caller() {
    return callee(4);
}
```

✓ Eliminated load in callee
✓ Eliminated store in caller
✓ Eliminated stack slot for ‘tmp’
Why is this hard?

• Requires interprocedural analysis:
  – Must change the prototype of the callee
  – Must update all call sites \( \rightarrow \) we must \textbf{know} all callers
  – What about callers outside the translation unit?

• Requires alias analysis:
  – Reference could alias other pointers in callee
  – Must know that loaded value doesn’t change from function entry to the load
  – Must know the pointer is not being stored through

• Reference might not be to a stack object!
Looking into events at compile-time

- C file → llvmgcc → .o file
- C++ file → llvm++ → .o file

C to LLVM Frontend → Compile-time Optimizer
- C++ to LLVM Frontend
- "cc1plus" → "gccas"

LLVM IR Parser → LLVM Verifier
- LLVM IR as text file
- Lowers C AST to LLVM

- 40 LLVM Analysis & Optimization Passes
- LLVM .bc File Writer
- Modified version of GCC
- Emits LLVM IR as text file
- Lowers C++ AST to LLVM

Dead Global Elimination, IP Constant Propagation, Dead Argument Elimination, Inlining, Reassociation, LICM, Loop Optns, Memory Promotion, Dead Store Elimination, ADCE, ...
Looking into events at link-time

- .o file
- LLVM Linker
- Link-time Optimizer
- LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner
- Perfect place for argument promotion optimization!
- Native Code Backend
  - “llc”
- C Code Backend
  - “llc -march=c”
- Native executable
- .bc file for LLVM JIT
- Link in native .o files and libraries here
- Native executable
- Chris Lattner – lattner@cs.uiuc.edu
Goals of the compiler design

- **Analyze and optimize as early as possible:**
  - Compile-time opts reduce modify-rebuild-execute cycle
  - Compile-time optimizations reduce work at link-time (by shrinking the program)

- **All IPA/IPO make an open-world assumption**
  - Thus, they all work on libraries and at compile-time
  - “Internalize” pass enables “whole program” optzn

- **One IR (without lowering) for analysis & optzn**
  - Compile-time optzn can be run at link-time too!
  - The same IR is used as input to the JIT

*IR design is the key to these goals!*

[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
Goals of LLVM IR

- Easy to produce, understand, and define!
- Language- and Target-Independent
  - AST-level IR (e.g. ANDF, UNCOL) is not very feasible
    - Every analysis/xform must know about ‘all’ languages
- One IR for analysis and optimization
  - IR must be able to support aggressive IPO, loop opts, scalar opts, … high- \textit{and} low-level optimization!
- Optimize as much as early as possible
  - Can’t postpone everything until link or runtime
  - No lowering in the IR!

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu
LLVM Instruction Set Overview #1

- **Low-level and target-independent semantics**
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers

- **IR has text, binary, and in-memory forms**

```llvm
loop:
  %i.1 = phi int [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, int %i.1
  call void %Sum(float %AiAddr, %pair* %P)
  %i.2 = add int %i.1, 1
  %tmp.4 = setlt int %i.1, %N
  br bool %tmp.4, label %loop, label %outloop
```

---


[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
LLVM Instruction Set Overview #2

- High-level information exposed in the code
  - Explicit dataflow through SSA form
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
    - Preserve array subscript and structure indexing

```llvm
loop:
  %i.1 = phi int [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, int %i.1
  call void %Sum(float %AiAddr, %pair* %P)
  %i.2 = add int %i.1, 1
  %tmp.4 = setlt int %i.1, %N
  br bool %tmp.4, label %loop, label %outloop
```

http://llvm.cs.uiuc.edu/
LLVM Type System Details

- The entire type system consists of:
  - Primitives: void, bool, float, ushort, opaque, ...
  - Derived: pointer, array, structure, function
  - No high-level types: type-system is language neutral!

- Type system allows arbitrary casts:
  - Allows expressing weakly-typed languages, like C
  - Front-ends can implement safe languages
  - Also easy to define a type-safe subset of LLVM

See also: docs/LangRef.html

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu
Lowering source-level types to LLVM

- **Source language types are lowered:**
  - Rich type systems expanded to simple type system
  - Implicit & abstract types are made explicit & concrete

- **Examples of lowering:**
  - References turn into pointers: `T& → T*
  - Complex numbers: `complex float → { float, float }
  - Bitfields: `struct X { int Y:4; int Z:2; } → { int }
  - Inheritance: `class T : S { int X; } → { S, int }
  - Methods: `class T { void foo(); } → void foo(T*)

- Same idea as lowering to machine code

http://llvm.cs.uiuc.edu/  Chris Lattner – lattner@cs.uiuc.edu
LLVM Program Structure

- **Module contains Functions/GlobalVariables**
  - Module is unit of compilation/analysis/optimization

- **Function contains BasicBlocks/Arguments**
  - Functions roughly correspond to functions in C

- **BasicBlock contains list of instructions**
  - Each block ends in a control flow instruction

- **Instruction is opcode + vector of operands**
  - All operands have types
  - Instruction result is typed
Our example, compiled to LLVM

```c
int callee(const int *X) {
    return *X+1; // load
}
int caller() {
    int T; // on stack
    T = 4; // store
    return callee(&T);
}
```

```c
internal int %callee(int* %X) {
    %tmp.1 = load int* %X
    %tmp.2 = add int %tmp.1, 1
    ret int %tmp.2
}
int %caller() {
    %T = alloca int
    store int 4, int* %T
    %tmp.3 = call int %callee(int* %T)
    ret int %tmp.3
}
```

Linker “internalizes” most functions in most cases

[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
Our example, desired transformation

```c
internal int %callee(int * %X) {
    %tmp.1 = load int* %X
    %tmp.2 = add int %tmp.1, 1
    ret int %tmp.2
}
int %caller() {
    %T = alloca int
    store int 4, int* %T
    %tmp.3 = call int %callee(int* %T)
    ret int %tmp.3
}

internal int %callee(int %X.val) {
    %tmp.2 = add int %X.val, 1
    ret int %tmp.2
}
int %caller() {
    %T = alloca int
    store int 4, int* %T
    %tmp.1 = load int* %T
    %tmp.3 = call int %callee(%tmp.1)
    ret int %tmp.3
}

int %caller() {
    %tmp.3 = call int %callee(int 4)
    ret int %tmp.3
}
```

Other transformation (-mem2reg) cleans up the rest.
LLVM Coding Basics

- Written in modern C++, uses the STL:
  - Particularly the vector, set, and map classes

- LLVM IR is almost all doubly-linked lists:
  - Module contains lists of Functions & GlobalVariables
  - Function contains lists of BasicBlocks & Arguments
  - BasicBlock contains list of Instructions

- Linked lists are traversed with iterators:
  ```
  Function *M = ...
  for (Function::iterator I = M->begin(); I != M->end(); ++I) {
    BasicBlock &BB = *I;
    ...
  }
  ```

See also: [docs/ProgrammersManual.html](http://llvm.cs.uiuc.edu/docs/ProgrammersManual.html)

http://llvm.cs.uiuc.edu/
LLVM Pass Manager

- Compiler is organized as a series of ‘passes’:
  - Each pass is one analysis or transformation

- Four types of Pass:
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time

- Constraints imposed (e.g. FunctionPass):
  - FunctionPass can only look at “current function”
  - Cannot maintain state across functions

See also: docs/WritingAnLLVMPass.html

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu

[The LLVM Compiler Framework and Infrastructure, LCPC Tutorial, 2004 by Chris Lattner]
Services provided by PassManager

- **Optimization of pass execution:**
  - Process a function at a time instead of a pass at a time
  - Example: If F, G, H are three functions in input pgm:
    “FFFFGGGGHHHH” not “FGHFGHFGHFGH”
  - Process functions in parallel on an SMP (future work)

- **Declarative dependency management:**
  - Automatically fulfill and manage analysis pass lifetimes
  - Share analyses between passes when safe:
    - e.g. “DominatorSet live unless pass modifies CFG”

- **Avoid boilerplate for traversal of program**

See also: [docs/WritingAnLLVMPass.html](http://llvm.cs.uiuc.edu/)

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu
Pass Manager + Arg Promotion #1/2

- **Arg Promotion is a CallGraphSCCPass:**
  - Naturally operates bottom-up on the CallGraph
    - Bubble pointers from callees out to callers

```cpp
24: #include "llvm/CallGraphSCCPass.h"
47: struct SimpleArgPromotion : public CallGraphSCCPass {

- **Arg Promotion requires AliasAnalysis info**
  - To prove safety of transformation
    - Works with any alias analysis algorithm though

```cpp
48: virtual void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<AliasAnalysis>(); // Get aliases
    AU.addRequired<TargetData>();     // Get data layout
    CallGraphSCCPass::getAnalysisUsage(AU); // Get CallGraph
```
Pass Manager + Arg Promotion #2/2

- Finally, implement `runOnSCC` (line 65):

```cpp
bool SimpleArgPromotion::runOnSCC(const std::vector<CallGraphNode*> &SCC) {
    bool Changed = false, LocalChange;
    do {  // Iterate until we stop promoting from this SCC.
        LocalChange = false;
        // Attempt to promote arguments from all functions in this SCC.
        for (unsigned i = 0, e = SCC.size(); i != e; ++i)
            LocalChange |= PromoteArguments(SCC[i]);
        Changed |= LocalChange;  // Remember that we changed something.
    } while (LocalChange);
    return Changed;           // Passes return true if something changed.
}
```

```cpp
static int foo(int ***P) {
    return ***P;
}
```

```cpp
static int foo(int P_val_val_val) {
    return P_val_val_val;
}
```

http://llvm.cs.uiuc.edu/  

Chris Lattner – lattner@cs.uiuc.edu
LLVM Dataflow Analysis

- LLVM IR is in SSA form:
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions

- Control Flow Graph is always available:
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG

- Higher-level info implemented as passes:
  - Dominators, CallGraph, induction vars, aliasing, GVN, ...

See also: docs/ProgrammersManual.html

http://llvm.cs.uiuc.edu/

Chris Lattner – lattner@cs.uiuc.edu
Arg Promotion: safety check #1/4

#1: Function must be “internal” (aka “static”)

```c
88: if (!F || !F->hasInternalLinkage()) return false;
```

#2: Make sure address of F is not taken

- In LLVM, check that there are only direct calls using F

```c
99: for (Value::use_iterator UI = F->use_begin();
    UI != F->use_end(); ++UI) {
    CallSite CS = CallSite::get(*UI);
    if (!CS.getInstruction()) // "Taking the address" of F.
        return false;
```

#3: Check to see if any args are promotable:

```c
114: for (unsigned i = 0; i != PointerArgs.size(); ++i)
    if (!isSafeToPromoteArgument(PointerArgs[i]))
        PointerArgs.erase(PointerArgs.begin()+i);
    if (PointerArgs.empty()) return false; // no args promotable
```

http://llvm.cs.uiuc.edu/ Chris Lattner – lattner@cs.uiuc.edu
#4: Argument pointer can only be loaded from:

- No stores through argument pointer allowed!

```c
// Loop over all uses of the argument (use-def chains).
for (Value::use_iterator UI = Arg->use_begin();
    UI != Arg->use_end(); ++UI) {
    // If the user is a load:
    if (LoadInst *LI = dyn_cast<LoadInst>(*UI)) {
        // Don’t modify volatile loads.
        if (LI->isVolatile()) return false;
        Loads.push_back(LI);
    } else {
        return false; // Not a load.
    }
}
```
Arg Promotion: safety check #3/4

#5: Value of "*P" must not change in the BB

- We move load out to the caller, value cannot change!

```
// Get AliasAnalysis implementation from the pass manager.
156: AliasAnalysis &AA = getAnalysis<AliasAnalysis>();

// Ensure *P is not modified from start of block to load
169: if (AA.canInstructionRangeModify(BB->front(), *Load,
                                        Arg, LoadSize))
      return false; // Pointer is invalidated!
```

See also: docs/AliasAnalysis.html

Chris Lattner – lattner@cs.uiuc.edu

http://llvm.cs.uiuc.edu/
#6: "*P" cannot change from Fn entry to BB

175: for (pred_iterator PI = pred_begin(BB), E = pred_end(BB);
        PI != E; ++PI) // Loop over predecessors of BB.
    // Check each block from BB to entry (DF search on inverse graph).
    for (idf_iterator<BasicBlock*> I = idf_begin(*PI);
        I != idf_end(*PI); ++I)
        // Might *P be modified in this basic block?
        if (AA.canBasicBlockModify(**I, Arg, LoadSize))
            return false;

http://llvm.cs.uiuc.edu/
Various omissions

- Exception handling, string literals
- Tools, applications, benchmarks
- ...

The LLVM Compiler Infrastructure (Homepage)

Getting Started with the LLVM System (How to install ...?)

Kaleidoscope: Implementing a Language with LLVM

llvm/build/examples/Kaleidoscope
Implementing a language with LLVM

Sample code (extracted from LLVM user guide) available online: https://github.com/slecourse/slecourse/tree/master/sources/llvm
Kaleidoscope: Implementing a Language with LLVM

LLVM Tutorial: Table of Contents

- Tutorial Introduction and the Lexer
- Implementing a Parser and AST
- Implementing Code Generation to LLVM IR
- Adding JIT and Optimizer Support
- Extending the language: control flow
- Extending the language: user-defined operators
- Extending the language: mutable variables / SSA construction
- Conclusion and other useful LLVM tidbits

Source: http://llvm.org/docs/tutorial/index.html
Kaleidoscope
Recursive Fibonacci

# Compute the x'th fibonacci number.
def fib(x):
    if x < 3 then
        1
    else
        fib(x-1)+fib(x-2)

# This expression will compute the 40th number.
fib(40)
Iterative Fibonacci

def binary : 1 (x y) y;

def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
    c = a + b :
    a = b :
    b = c) :
  b;

User-defined sequencing operator

Use of mutable variables.
A simple parser for Kaleidoscope
A simple parser for Kaleidoscope

- A simple lexer
- A recursive descent parser
- Plain old C++ objects for AST
if (isdigit(LastChar) || LastChar == '.') { // Number: [0-9]+
    std::string NumStr;
    do {
        NumStr += LastChar;
        LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), 0);
    return tok_number;
}
A recursive descent parser

```c
static ExprAST *ParsePrimary() {
    switch (CurTok) {
    default: return Error("unknown token when expecting an expression");
    case tok_identifier: return ParseIdentifierExpr();
    case tok_number: return ParseNumberExpr();
    case '(': return ParseParenExpr();
    }
}
```
Plain old C++ objects for AST

```cpp
/// NumberExprAST – Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
    double Val;

public:
    NumberExprAST(double val) : Val(val) {};

};

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
    ExprAST *Result = new NumberExprAST(NumVal);
    getNextToken(); // consume the number
    return Result;
}
```
A simple code generator for Kaleidoscope
A simple code generator for Kaleidoscope

- Add virtual code generation methods to AST classes
- Expressions return SSA registers ("Value" in LLVM)
- Use some house-keeping data structures
- Use uniqued IR, e.g., for literals
- Use of name hints for intermediate results
- Functions return, well, functions ("Function" in LLVM)
- Use an error method to report, for example, missing decls
Add virtual code generation methods to AST classes

```cpp
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
    virtual ~ExprAST() {}
    virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
    double Val;
public:
    NumberExprAST(double val) : Val(val) {}
    virtual Value *Codegen();
};
...
```
Use some house-keeping data structures

```cpp
static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
```
Floating point literals

Value *NumberExprAST::CodeGen() {
    return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Arbitrary precision numbers

Uniqued IR for numbers; no "new"!
Variable references

```cpp
Value *VariableExprAST::Codegen() {
    // Look this variable up in the function.
    Value *V = NamedValues[Name];
    return V ? V : ErrorV("Unknown variable name");
}
```

Mapping to be initialized by code for function declaration.
Binary expressions

Value *BinaryExprAST::Codegen() {
    Value *L = LHS->Codegen();
    Value *R = RHS->Codegen();
    if (L == 0 || R == 0) return 0;

    switch (Op) {
    case '+': return Builder.CreateFAdd(L, R, "addtmp");
    case '-': return Builder.CreateFSub(L, R, "subtmp");
    case '*': return Builder.CreateFMul(L, R, "multmp");
    case '<':
        L = Builder.CreateFCmpULT(L, R, "cmptmp");
        // Convert bool 0/1 to double 0.0 or 1.0
        return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                    "booltmp");
    default: return ErrorV("invalid binary operator");
    }
}

Use of name hints for intermediate results
Function calls

Value *CallExprAST::Codegen() {
    // Look up the name in the global module table.
    Function *CalleeF = TheModule->getFunction(Callee);
    if (CalleeF == 0)
        return ErrorV("Unknown function referenced");

    // If argument mismatch error.
    if (CalleeF->arg_size() != Args.size())
        return ErrorV("Incorrect # arguments passed");

    std::vector<Value*> ArgsV;
    for (unsigned i = 0, e = Args.size(); i != e; ++i) {
        ArgsV.push_back(Args[i]->Codegen());
        if (ArgsV.back() == 0) return 0;
    }

    return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
Function *PrototypeAST::Codegen() {
    // Make the function type:  double(double,double) etc.
    std::vector<Type*> Doubles(Args.size(),
                               Type::getDoubleTy(getGlobalContext()));
    FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                           Doubles, false);
    
    Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

    ... error handling omitted ...

    // Set names for all arguments.
    unsigned Idx = 0;
    for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
         ++AI, ++Idx) {
        AI->setName(Args[Idx]);

        // Add arguments to variable symbol table.
        NamedValues[Args[Idx]] = AI;
    }
    return F;
}
Function definitions

Function *FunctionAST::Codegen() {
    NamedValues.clear();
    Function *TheFunction = Proto->Codegen();
    if (TheFunction == 0) return 0;
    BasicBlock *BB = BasicBlock::Create(
        getGlobalContext(), "entry", TheFunction);
    Builder.SetInsertPoint(BB);

    if (Value *RetVal = Body->Codegen()) {
        // Finish off the function.
        Builder.CreateRet(RetVal);
        verifyFunction(*TheFunction);
        return TheFunction;
    }
    TheFunction->eraseFromParent();
    return 0;
}
Some optimizations for Kaleidoscope
Trivial constant folding

Kaleidoscope

```python
def test(x) 1+2+x;
```

IR

```c
define double @test(double %x) {
  entry:
  %addtmp = fadd double 3.000000e+00, %x
  ret double %addtmp
}
```

Relies on basic optimization during IR construction
Missing constant folding

Kaleidoscope  \[
\text{def test}(x) \ (1+2+x)^*(x+(1+2));
\]

IR  \[
define\ double\ \text{@test}(\text{double} \ \%x) \ 
\entry:\
\quad \%addtmp = \text{fadd double} \ 3.000000e+00, \ \%x \\
\quad \%addtmp1 = \text{fadd double} \ %x, \ 3.000000e+00 \\
\quad \%multmp = \text{fmul double} \ %addtmp, \ %addtmp1 \\
\quad \text{ret double} \ %multmp
\]
Missing constant folding

Kaleidoscope

```
def test(x) (1+2+x)*(x+(1+2));
```

IR

```
define double @test(double %x) {
    entry:
    %addtmp = fadd double 3.000000e+00, %x
    %multmp = fmul double %addtmp, %addtmp
    ret double %multmp
}
```
Register per-function optimizations

FunctionPassManager OurFPM(TheModule);
    // Basic setup logic.
OurFPM.add(new DataLayout(*TheExecutionEngine->getDataLayout()));
    // Provide basic AliasAnalysis support for GVN.
OurFPM.add(createBasicAliasAnalysisPass());
    // Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
    // Reassociate expressions.
OurFPM.add(createReassociatePass());
    // Eliminate Common SubExpressions.
OurFPM.add(createGVNPass());
    // Simplify the control flow graph (deleting unreachable blocks, etc).
OurFPM.add(createCFGSimplificationPass());

OurFPM.doInitialization();
TheFPM = &OurFPM; // Inform code generation via global.
MainLoop(); // Run the main "interpreter loop" now.
if (Value *RetVal = Body->Codegen()) {
   // Finish off the function.
   Builder.CreateRet(RetVal);

   // Validate the generated code, checking for consistency.
   verifyFunction(*TheFunction);

   // Optimize the function.
   TheFPM->run(*TheFunction);

   return TheFunction;
}
JIT-based execution/evaluation for Kaleidoscope
$ ./toy
called()
ready> 4+5;
ready> Read top-level expression:
define double @0() {
  entry:
    ret double 9.000000e+00
}

Evaluated to 9.000000
ready>
static ExecutionEngine *TheExecutionEngine;
...
int main() {
    ..
    // Create the JIT. This takes ownership of the module.
    TheExecutionEngine = EngineBuilder(TheModule).create();
    ..
}
Original main loop

```c
/// top ::= definition | external | expression | ';
static void MainLoop() {
    while (1) {
        fprintf(stderr, "ready> ");
        switch (CurTok) {
        case tok_eof:    return;
        case ';':        getNextToken(); break;  // ignore top-level ";".
        case tok_def:    HandleDefinition(); break;
        case tokExtern:  HandleExtern(); break;
        default:         HandleTopLevelExpression(); break;
        }
    }
}
```
static void HandleTopLevelExpression() {
    // Evaluate a top-level expression into an anonymous function.
    if (FunctionAST *F = ParseTopLevelExpr()) {
        if (Function *LF = F->Codegen()) {
            LF->dump();  // Dump the function for exposition purposes.

            // JIT the function, returning a function pointer.
            void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

            // Cast it to the right type (no args, returns a double).
            // Thus, call it as a native function.
            double (*FP)() = (double (*)()) (intptr_t) FPtr;
            fprintf(stderr, "Evaluated to %f\n", FP());
        }
    }
    ... continue with parsing ...
}
Control flow for Kaleidoscope
IR

define double @fib(double %x) {
    entry:
    %cmptmp = fcmp ult double %x, 3.000000e+00
    br i1 %cmptmp, label %ifcont, label %else

    else: ; preds = %entry
    %subtmp = fadd double -1.000000e+00, %x
    %calltmp = call double @fib(double %subtmp)
    %subtmp1 = fadd double -2.000000e+00, %x
    %calltmp2 = call double @fib(double %subtmp1)
    %addtmp = fadd double %calltmp, %calltmp2
    br label %ifcont

    ifcont: ; preds = %entry, %else
    %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]
    ret double %iftmp
}
Extend lexer (trivial)

tok_if = -6, tok_then = -7, tok_else = -8,
...
if (IdentifierStr == "def") return tok_def;
if (IdentifierStr == "extern") return tokExtern;
if (IdentifierStr == "if") return tok_if;
if (IdentifierStr == "then") return tok_then;
if (IdentifierStr == "else") return tok_else;
return tok_identifier;
/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
    ExprAST *Cond, *Then, *Else;
public:
    IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *else)
        : Cond(cond), Then(then), Else(_else) {}
    virtual Value *Codegen();
};
Extend parser (trivial)

```c
/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
    getNextToken();  // eat the if.
    ExprAST *Cond = ParseExpression();
    if (!Cond) return 0;
    if (CurTok != tok_then) return Error("expected then");
    getNextToken();  // eat the then
    ExprAST *Then = ParseExpression();
    if (Then == 0) return 0;
    if (CurTok != tok_else) return Error("expected else");
    getNextToken();
    ExprAST *Else = ParseExpression();
    if (!Else) return 0;
    return new IfExprAST(Cond, Then, Else);
}
```
static ExprAST *ParsePrimary() {
    switch (CurTok) {
        default: return Error("unknown token when expecting an expression");
        case tok_identifier: return ParseIdentifierExpr();
        case tok_number: return ParseNumberExpr();
        case '(' : return ParseParenExpr();
        case tok_if : return ParseIfExpr();
    }
}

Value *IfExprAST::Codegen() {
    Value *CondV = Cond->Codegen();
    if (CondV == 0) return 0;

    // Convert condition to a bool by comparing equal to 0.0.
    CondV = Builder.CreateFCmpONE(CondV,
                                      ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                      "ifcond");
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create blocks for the then and else cases.
// Insert the 'then' block at the end of the function.
BasicBlock *ThenBB = BasicBlock::Create(
    getGlobalContext(), "then", TheFunction);
BasicBlock *ElseBB = BasicBlock::Create(
    getGlobalContext(), "else");
BasicBlock *MergeBB = BasicBlock::Create(
    getGlobalContext(), "ifcont");

Builder.CreateCondBr(CondV, ThenBB, ElseBB);
// Emit then value.
Builder.SetInsertPoint(ThenBB);

Value *ThenV = Then->Codegen();
if (ThenV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Then' can change the current block.
// Update ThenBB for the PHI.
ThenBB = Builder.GetInsertBlock();
// Emit else block.
TheFunction->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseV = Else->Codegen();
if (ElseV == 0) return 0;

Builder.CreateBr(MergeBB);
// Codegen of 'Else' can change the current block.
// Update ElseBB for the PHI.
ElseBB = Builder.GetInsertBlock();
// Emit merge block.
TheFunction->getBasicBlockList().push_back(MergeBB);
Builder.SetInsertPoint(MergeBB);
PHINode *PN = Builder.CreatePHI(
    Type::getDoubleTy(getGlobalContext()), 2,
    "iftmp");

PN->addIncoming(ThenV, ThenBB);
PN->addIncoming(ElseV, ElseBB);
return PN;
}
User-defined operators for Kaleidoscope
User-defined operators

- Treat operators like functions
- Use of precedence parsing
- Use of external functions

Demo: [http://llvm.org/docs/tutorial/LangImpl6.html](http://llvm.org/docs/tutorial/LangImpl6.html)
Mutable variables for Kaleidoscope
def binary : 1 (x y) y;

def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
   c = a + b :
   a = b :
   b = c) :
  b;

User-defined sequencing operator

Use of mutable variables.
SSA revisited

We need SSA for optimizations.

LLVM IR supports SSA.

Functional Kaleidoscope implied SSA trivially.

Let’s add mutable variables and see how IR is retained.
Illustrative C example

```c
int G, H;
int test(_Bool Condition) {
    int X;
    if (Condition)
        X = G;
    else
        X = H;
    return X;
}
```
The value of X obviously depends on the executed path.
Desired SSA

@G = weak global i32 0 ; type of @G is i32*
@H = weak global i32 0 ; type of @H is i32*

define i32 @test(i1 %Condition) {
  entry:
    br i1 %Condition, label %cond_true, label %cond_false
  
  cond_true:
    %X.0 = load i32* @G
    br label %cond_next
  
  cond_false:
    %X.1 = load i32* @H
    br label %cond_next
  
  cond_next:
    %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
    ret i32 %X.2
}
Desired SSA

- SSA required (by LLVM) for registers.
- Too difficult to generate in a frontend.
- SSA not required (by LLVM) for memory:
  - Model mutable variables via stack.
  - Rely on optimization `mem2reg`. 
Mutable variables based on stack alloca

@G = weak global i32 0 ; type of @G is i32*
@H = weak global i32 0 ; type of @H is i32*

define i32 @test(i1 %Condition) {
  entry:
  %X = alloca i32 ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32 %X ; Read X
  ret i32 %X.2
}

1. Each mutable variable becomes a stack allocation.
2. Each read of the variable becomes a load from the stack.
3. Each update of the variable becomes a store to the stack.
4. The variable represents the stack address.
static std::map<std::string, Value*> NamedValues;

static std::map<std::string, AllocaInst*> NamedValues;

Add allocations at the entry block of each function.

Codegen for variables reference needs to load from stack.

Codegen for assginments needs to store on stack.