Intermediate Code Generation

Transforming the abstract syntax tree (AST) into intermediate representation is called lowering.

We will modify the stack/register machine that we saw already, by adding types to it.

The resulting intermediate representation is similar to LLVM (Low Level Virtual Machine), but not completely the same. In the end, we decided not to base the complete course on LLVM, because some of its features are ugly. It is close enough to our representation, so you will have no problems understanding it.

The big advantage of teaching LLVM is that it is used by real compilers.

Requirements on Intermediate Representation

Most compilers translate different languages into different machines.

In order to avoid quadratic explosion, one should use a single common intermediate representation.

The intermediate representation must be far enough from the languages to be suitable for all of them, and must be also far enough from the machines to enable translation into all of them.

The representation should have a simple type system, and be suitable for optimizations.

Simple/Non-simple Types

We distinguish between simple and non-simple types.

Simple types are types that fit into a register. Primitive types are always simple.

Classes that have non-trivial copy-constructor/assignment/destructor are always non-simple.

Non-trivial means: Doing something different than just byte copying.

Other **struct** types can be simple in principle if they are not too big. In CLANG, a complex number is still simple, while a quaternion is not simple anymore. In our framework, **struct**s are always non-simple.

Typed Register Machine

Before we can discuss any form of typed translation, we need to develop a typed version of the stack/register machine.

We add types to register assignments, so that they now have form:

$$\begin{aligned}
\alpha_1\text{: } &\textbf{int} &\leftarrow\ & 3\\
\alpha_2\text{: } &\textbf{real} &\leftarrow\ & 2.7182818\\
\alpha_3\text{: } &\textbf{real} &\leftarrow\ & \textbf{conv}(\alpha_1)\\
\alpha_4\text{: } &\textbf{real} &\leftarrow\ & \textbf{sum}(\alpha_2, \alpha_3)
\end{aligned}$$

Note that, even though we attached the types to the registers, we effectively typed the operations (as would be in assembly language). This is because every operation works on registers, and creates a register.

Explicit Addresses, More Complicated Calculations

In the untyped register machine, load and store had the following forms:

$$\alpha_1 \leftarrow \text{load } \#1$$

$$\text{store } \alpha_1 \ \#2$$

Because C has pointers, which can be calculated at run time, the address of the load/store is not always fixed in advance. We will use # to create a pointer to the local variable, and let load/store use the pointer:

$$\alpha_1\text{: pntr(int)} \leftarrow \#1$$

$$\alpha_2\text{: int} \leftarrow \text{load } \alpha_1$$

$$\alpha_3\text{: pntr(int)} \leftarrow \#2$$

$$\text{store } \alpha_3 \ \alpha_4$$

Real Hardware is Complicated

On real hardware, the address calculations are more complicated:

- Different data types have different lengths. If the first variable is **int**, the second variable starts at position 4. If the first variable is **double**, the second variable starts at position 8.
- Some types have alignment requirements. On most hardware, **int** and **double** must start at an address that is a multiple of 4. This implies that if the first local variable is **char**, 3 bytes will be wasted.

This also applies to fields in structs.

We ignore these problems, because in every meaningful program, the sequence of local variables is always fixed at each point in the program. Hence, the exact address can be computed, if one wishes to produce real assembly. Also, sizes are hardware dependent, so we don't want to include them.

Alloc and Dealloc

We need to add a type to the alloc statement. Otherwise, it would be impossible to translate to hardware.

There is no need to specify types to dealloc, because the sequence of allocated variables is always fixed at each program point (see next slide):

alloc **double**

alloc **int**

alloc **char**

...

dealloc 3

Sequence of Local Variables is Always Fixed

We require that, when two possible computation paths merge, the types of the allocated variables on these paths are the same.

This means that the following situation is not allowed:

$$
\begin{array}{ll}
 & \text{iffalse } \alpha \text{ goto } L_1 \\
 & \text{alloc } \textbf{double} \\
 & \text{goto } L_2 \\
L_1: & \text{alloc } \textbf{int} \\
L_2: & \alpha{:}?? \leftarrow \#0 \\
 & \text{dealloc } 1
\end{array}
$$

At L_2, we don't know the type of #, and we don't know how much space to deallocate.

Memcopy

We assumed that registers are limited in size. They can only hold built-in types and pointers.

In order to assign **struct**s and pass them to functions, we need a **memcopy** statement. It has form **memcopy** α_1, α_2, where α_1 α_2 must have form **pntr**(T). It copies from left to right.

On the following slides, we give some examples. The function call/return mechanism is not changed.

The translation function will be a bit more complicated (but not much) because of possible **struct**s that don't fit into a register.

Consider the `fact` function again, this time properly typed:

```
double fact( unsigned n )
{
   double res = 1.0;
   while( n != 0 )
   {
      res = res * n;
      n = n - 1;
   }
   return res;
}
```

It starts with

unsigned	**double**
n	(uninitialized)

Translation of `fact`

Create local variable `res` and initialize it with 1.0 :

alloc **double**

α_0: **pntr**(**double**) $\leftarrow$ #0

α_1: **double** $\leftarrow$ 1.0

store α_1 α_0

The condition: `while( n != 0 )`

loop : α_2: **pntr**(**unsigned**) $\leftarrow$ #1

α_3: **unsigned** $\leftarrow$ load α_2

α_4: **unsigned** $\leftarrow$ 0

α_5: **bool** $\leftarrow$ neq α_3 α_4

iffalse α_5 goto exit

Statement `res = res * n`

α_6: **pntr**(**double**) $\leftarrow$ #0

α_7: **double** $\leftarrow$ load α_6

α_8: **pntr**(**unsigned**) $\leftarrow$ #1

α_9: **unsigned** $\leftarrow$ load α_8

α_{10}: **double** $\leftarrow$ conv α_9

α_{11}: **double** $\leftarrow$ mult α_7 α_{10}

α_{12}: **pntr**(**double**) $\leftarrow$ #0

store α_{11} α_{12}

Statement `n = n - 1`

α_{13}: **pntr**(**unsigned**) $\leftarrow$ #1

α_{14}: **unsigned** $\leftarrow$ load α_{13}

α_{15}: **unsigned** $\leftarrow$ 1

α_{16}: **unsigned** $\leftarrow$ minus α_{14} α_{15}

α_{17}: **pntr**(**unsigned**) $\leftarrow$ #1

store α_{16} α_{17}

goto loop

Copy local variable `res` into the result, and clean up `res` and `n`:

exit : α_{18}: **pntr**(**double**) $\leftarrow$ #0

α_{19}: **double** $\leftarrow$ load α_{18}

dealloc 2

α_{20}: **pntr**(**double**) $\leftarrow$ #0

store α_{19} α_{20}

return

strlen and lstlen

We study two more examples, which are given on this and the next slide. The aim is to explain how pointer arithmetic works.

```
unsigned int strlen( char* p )
{
   unsigned int len = 0;
   while( *p )
   {
      ++ p;                 // pointer arithmetic
      ++ len;
   }
   return len;
}
```

lstlen

```
struct list { int first;   struct list* next; };

unsigned int listlen( struct list* p )
{
   unsigned int len = 0;
   while(p)
   {
      p = ( p -> next );   // pointer arithmetic
      ++ len;
   }
   return len;
}
```

The while condition `while( *p )` is not difficult to translate, it contains an implicit conversion to **bool**:

For simplicity, we assume that counting of registers starts at 0.

α_0: **pntr**(**pntr**(**char**)) $\leftarrow$ #1

α_1: **pntr**(**char**) $\leftarrow$ load α_0

α_2: **char** $\leftarrow$ load α_1

α_3: **bool** $\leftarrow$ conv α_2

iffalse α_3 goto exit

Translation of Control Statements

Let's start doing systematic translations.

Statements of form **if** B **then** S_1 **else** S_2 are translated as follows:

$$
\begin{array}{ll}
 & \alpha\text{: } \mathbf{bool} \leftarrow \cdots \\
 & \text{iffalse } \alpha \text{ goto } L_1 \\
 & \text{translation of } S_1 \\
 & \text{goto } L_2 \\
L_1 : & \text{translation of } S_2 \\
L_2 : &
\end{array}
$$

Statements of form **while** B **do** S are translated into:

L_1 α: **bool** $\leftarrow \cdots$

iffalse α goto L_2

translation of S

goto L_1

L_2 :

Understanding Pointers

1. A pointer can point to an array element. In that case, one can add integers to it, which will move the pointer to another position in the same array. This type of addition does not change the type of the pointer. As a consequence, array pointers can be subtracted.
2. A pointer can point to a struct. In that case, one can add an unsigned int to it, which will move the pointer into a field. This addition will change the type of the pointer.
3. A pointer can point to an array. In that case, one can add an unsigned to it, which will move the pointer to an array element. This addition will change the type of the pointer.

Due to implicit conversions, (3) never occurs in *C*, but it exists in reality, and other languages may have it.

I believe that the intended type of pointer addition, can always derived from the type. Additions of type 2,3, will always change the type, while addition of type 1 does not.

Hence, it is possible to use the name **sum** for all of them. We will also allow **diff** on type 1 pointers.

LLVM, which we will see later, uses another approach: It defines a ternary (or higher) operator `getelementpntr` (`gep`), where the first operation is always of type 1, and later arguments are type 2,3 additions. If you want only a 2 or 3, you have to insert 0 as first argument.

Statement `p ++`

$$\alpha_0\text{: } \mathbf{pntr}(\mathbf{pntr}(\mathbf{char})) \leftarrow \#1$$

$$\alpha_1\text{: } \mathbf{pntr}(\mathbf{char}) \leftarrow \text{load } \alpha_0$$

$$\alpha_2\text{: } \mathbf{unsigned} \leftarrow 1$$

$$\alpha_3\text{: } \mathbf{pntr}(\mathbf{char}) \leftarrow \text{sum } \alpha_1\ \alpha_2$$

$$\alpha_4\text{: } \mathbf{pntr}(\mathbf{pntr}(\mathbf{char})) \leftarrow \#1$$

$$\text{store } \alpha_3\ \alpha_4$$

The sum in α_3 is a type 1 addition. Because of this, α_1 and α_3 have the same type.

Note that in concrete hardware, type 1 addition must take the length of the data into account.

Translation of `p = ( p -> next )`

α_0: **pntr**(**pntr**(**list**)) $\leftarrow$ #1

α_1: **pntr**(**list**) $\leftarrow$ load α_0

α_2: **unsigned** $\leftarrow$ 1

α_3: **pntr**(**pntr**(**list**)) $\leftarrow$ sum α_1 α_2

α_4: **pntr**(**list**) $\leftarrow$ load α_3

α_5: **pntr**(**pntr**(**list**)) $\leftarrow$ #1

store α_4 α_5

The addition in α_3 is a type 2 addition. It changed the type of α_1, and α_1 is a pointer to a struct type.

Two Translation Functions

We are ready to translate C-expressions. Since we have to deal with non-simple **struct** types that do not fit in a register, we need two versions.

The first version **translate**(t) returns a register, and emits code that writes the value of t into this register as before. The second version **memtranslate**(t) emits code that pushes the value of t on the memory stack, and returns nothing.

In principle **memtranslate**(t) would be sufficient to obtain a complete translation function, but it would be very inefficient on simple operations on simple data, because of too many **load**s and **store**s.

Treatment of Lvalues

Lvalues are in fact pointers, so we define the following function:

Let t be an AST. If $t.\text{lr} = \textbf{lval}$, then $t.\text{truetype} = \textbf{pntr}(t.\text{type})$.
Otherwise, $t.\text{truetype} = t.\text{type}$.

Beginning is always hard

We have to start the translation of t. If t.truetype is non-simple, or t has form $f(t_1, \ldots, t_n)$ with f a user defined function, then call **memtranslate**(t).

If t.truetype is not **void**, then emit **dealloc** 1.

Otherwise, just call **translate**(t) and ignore the returned register.

If t is the condition of an **if** or **while** statement we assume that t.truetype = **bool**.

We assume that **return** t has been replaced by an assignment to the return variable.

Translation Function

Translation into a register is not really different from the untyped translation function, but C-operators need to be replaced by their definitions in the process.

As before, **translate**(AST t) returns a register, and emits code that writes the value of t into this register.

Let $T = t$.truetype.

- If t is a constant c, then create a new register name α. Emit α: $T \leftarrow c$ and return α.
- If t is a variable v, then create a new register name α. Let i be the current position of v on the stack. Emit: α: $T \leftarrow \#i$, and return α.

Otherwise, the AST has form $t = f(t_1, \ldots, t_n)$, where f is either a user defined function, or a built-in function.

If f is a user defined function, we proceed as follows:

First call **memtranslate**($f(t_1, \ldots, t_n)$). This emits code that pushes the value of $f(t_1, \ldots, t_n)$ on the stack.

But we need it in a register. Invent two register names α, β.

Emit

$$\beta\text{: } \mathbf{pntr}(T) \leftarrow \#0$$

$$\alpha\text{: } T \leftarrow \text{load } \beta$$

$$\text{dealloc } 1$$

Now we can return α.

- If t has form $\mathbf{op}(t_1, \ldots, t_n)$ with **op** a built-in operator, we will replace **op** by its definition. First, recursively call

$$\alpha_1 = \mathbf{translate}(t_1), \ldots, \alpha_n = \mathbf{translate}(t_n).$$

For each t_i, define $T_i = t_i$.truetype.

Find a definition for **op** in the list that we give on the following slides. This definition has form

$$\beta\colon T\ \mathbf{op}(\ \beta_1\colon T_1, \ldots, \beta_n\colon T_n\ ) \Rightarrow\ \text{(implementation of } \mathbf{op}).$$

Emit the implementation of **op**, with the following replacements: Each register β_i is replaced by its corresponding α_i. If the implementation contains more registers, these are replaced by new, unused registers.

After that, we return the register by which β was replaced.

Definitions of Built-In Functions

We assume that special function definitions always have form

$$\beta\colon U\ \mathbf{op}(\beta_1\colon U_1, \ldots, \beta_n\colon U_n) \Rightarrow \text{ implementation of } \mathbf{op}.$$

We list the main ones:

- For every simple type U, we have a definition for the load operator

$$\beta\colon U\ \text{load}(\ \beta_1\colon \mathbf{pntr}(U)\ ) = \quad \beta\colon U \leftarrow \text{load}\ \beta_1.$$

- For every primitive operator (like for example $+$), we have definitions of form:

$$\beta\colon U + (\ \beta_1\colon U,\ \beta_2\colon U'\ ) \Rightarrow \beta\colon U \leftarrow \text{sum}\ \beta_1\ \beta_2.$$

Definitions of Built-In Functions

- For **conv**, we have the following definitions:

$$\begin{array}{lll} \beta_1\text{: } U \text{ conv}(\ \beta_1\text{: } U_1\ ) & \Rightarrow & \text{if } U = U_1 \\ \beta\text{: } U \text{ conv}(\ \beta_1\text{: } U_1\ ) & \Rightarrow \quad \beta\text{: } U \leftarrow \textbf{conv } \beta_1 & \text{if } U \neq U_1 \end{array}$$

  A conversion between identical types emits no code, and returns the register of the original value.

- For **xpp**, we have the following definitions:

$$\begin{array}{lll} \beta\text{: } U \text{ xpp}(\ \beta_1\text{: } \textbf{pntr}(U)\ ) \Rightarrow & \beta\text{: } U \leftarrow \text{load } \beta_1 \\ & \beta'\text{: } \textbf{unsigned} \leftarrow 1 \\ & \beta''\text{: } U \leftarrow \text{sum } \beta\ \beta' \\ & \text{store } \beta''\ \beta_1 \end{array}$$

- Definitions for xmm are similar.

Definitions of Assignment Operators

- For every simple type U, we have an assignment operator

$$\beta_2\text{: } U \ =(\ \beta_1\text{: } \mathbf{pntr}(U),\ \beta_2\text{: } U\ ) \Rightarrow \ \text{store } \beta_2\ \beta_1.$$

- For every simple type U to which **unsigned**s can be added, we have a preincreasement operator ppx:

$$\begin{aligned} \beta_1\text{: } \mathbf{pntr}(U)\ \text{ppx}(\ \beta_1\text{: } \mathbf{pntr}(U)\ ) \Rightarrow \quad & \beta'\text{: } U \leftarrow \text{load } \beta_1 \\ & \beta''\text{: } \mathbf{unsigned} \leftarrow 1 \\ & \beta'''\text{: } U \leftarrow \text{sum } \beta'\ \beta'' \\ & \text{store } \beta'''\ \beta_1 \end{aligned}$$

- The definition of **mmx** is similar.

Definitions of Assignment Operators

- For every simple type U on which addition is defined, we define an adding assignment operator:

$$\beta'' \colon U \mathrel{+}= (\ \beta_1 \colon \mathbf{pntr}(U),\ \beta_2 \colon U\ ) \Rightarrow \quad \begin{array}{l} \beta' \colon U \leftarrow \text{load } \beta_1 \\ \beta'' \colon U \leftarrow \text{sum } \beta' \ \beta_2 \\ \text{store } \beta'' \ \beta_1 \end{array}$$

- The definitions of the other modifying assignment operators are similar.

Lvalue Field Selection

We give the definition for lvalue field selection. Unfortunately rvalue field selection does not fit into the schedule, so we treat it separately on the next slide.

- For every struct type U_1, for each of its fields f: U with relative position i, we have a definition

$$\beta\text{: } \mathbf{pntr}(U)\ \mathrm{select}_f(\ \mathbf{pntr}(\beta_1)\text{: } U_1\ ) \Rightarrow \quad \begin{array}{l} \beta'\text{: } \mathbf{unsigned} \leftarrow i \\ \beta\text{: } U \leftarrow \mathrm{sum}\ \beta_1\ \beta' \end{array}$$

  Field selection adds the offset of the field to the pointer. The addition is of type 2.

Rvalue field selection

Rvalue field selection does not fit into the pattern of the previous slides, because the **struct** is in memory, while the result has to be in a register. Therefore, we need to give a separate definition for **translate**($\text{select}_f(t_1)$), for the case when $\text{select}_f(t_1).\text{lr} = \textbf{rval}$.

Let $T = \text{select}_f(t_1).\text{type}$. Let i be the relative position of field f in T, and let U be its type.

Rvalue field selection (2)

First call **memtranslate**(t_1). This creates code that pushes the value of t_1 on the stack.

Create new register names $\alpha, \alpha_1, \alpha_2, \alpha_3$ and emit:

α_1: **pntr**$(T) \leftarrow \#0$

α_2: **unsigned** $\leftarrow i$

α_3: **pntr**$(U) \leftarrow$ sum α_1 α_2

α: $U \leftarrow$ load α_3

dealloc 1

Return α.

Translation into Memory

We define **memtranslate**(t).

It emits code that pushes the value of t on the stack.

Let $T = t$.truetype.

If t has form $f(t_1, \ldots, t_n)$ with f a user defined function, then emit

$$\text{alloc } T.$$

After that, for $i := n$ to 1, call **memtranslate**(t_i). This creates code that pushes the values of t_i on the stack.

Finally, emit

$$\text{call } f.$$

Translation into Memory (2)

If t has form $\text{select}_f(t_1)$, with $t.\text{lr} = \textbf{rval}$, and $t.\text{type}$ non-simple, then let i be the index of field f in T. Let U be its type.

Emit

$$\text{alloc } U.$$

Call **memtranslate**(t_1), and emit

$$\alpha_1\text{: } \textbf{pntr}(T) \leftarrow \#0$$

$$\alpha_2\text{: } \textbf{unsigned} \leftarrow i$$

$$\alpha_3\text{: } \textbf{pntr}(U) \leftarrow \text{sum } \alpha_1\ \alpha_2$$

$$\alpha_4\text{: } \textbf{pntr}(U) \leftarrow \#1$$

$$\text{memcopy } \alpha_3\ \alpha_4$$

$$\text{dealloc } 1$$

Translation into Memory (3)

If t has form $=(t_1, t_2)$, with $t.\text{type} = t_2.\text{type}$ a non-simple type T, call **memtranslate**(t_2).

After that call, $\alpha =$ **translate**(t_1). This is possible, because $t_1.\text{lr} =$ **lval**.

Create a new register name α_1, and emit

$$\alpha_1\text{: } \mathbf{pntr}(T) \leftarrow \#0$$

$$\text{memcopy } \alpha_1 \ \alpha$$

We do not deallocate t_2, because it must be returned by $=$.

Translation into Memory (4)

If t has form load(t_1), with t.type = t_1.type a non-simple type T, t.lr = **rval**, and t_1.lr = **lval**, then call α = translate(t_1).

Create a new register β and emit

alloc T

β: **pntr**(T) $\leftarrow$ #0

memcopy α β

Translation into Memory (5)

If the previous cases cannot be applied, then t.truetype must be simple. We call **translate** and store the result in memory.

t must have form $\mathbf{op}(t_1, \ldots, t_n)$ with **op** a simple operator.

Call $\alpha = \mathbf{translate}(t)$.

Create a new register β and emit:

alloc T

β: $\mathbf{pntr}(T) \leftarrow \#0$

store α β

Translation of Boolean ? Expression1 : Expression2

Assume that the AST has form $t = ?(t_1, t_2, t_3)$. We have t_1.truetype = **bool**, and t.truetype = t_1.truetype = t_2.truetype. Define $T = t$.truetype.

Translation of $t_1 ? \; t_2 : t_3$

We first define **translate** : Let $\alpha =$ **translate**(t_1). Create two new labels L_1, L_2, and emit

$$\text{iffalse } \alpha \text{ goto } L_1.$$

Call $\alpha_1 =$ **translate**(t_2). Emit

$$\text{goto } L_2$$
$$L_1 :$$

Call $\alpha_2 =$ **translate**(t_3). Emit

$$\alpha_1 \colon T \leftarrow \alpha_2$$
$$L_2 :$$

Return α_1.

Translation of $t_1 ?\ t_2 : t_3$

We define **memtranslate**($?(t_1, t_2, t_3)$) :

Let $\alpha =$ **translate**(t_1). Create two new labels L_1, L_2. Emit

$$\text{iffalse } \alpha \text{ goto } L_1$$

Call $\alpha_1 =$ **memtranslate**(t_2). Emit

$$\text{goto } L_2$$

$$L_1 :$$

Call **memtranslate**(t_3). Emit

$$L_2 :$$

Return.

Examples

This is a possible implementation of **strcpy**:

```
void strcopy( char* p, const char* q )
{
   size_t i = 0;
   while( q[i] )
   {
      p[i] = q[i];
      i ++ ;
   }
   p[i] = 0;
}
```

Length of a C-string

```
size_t length( char* s )
{
   size_t length = 0;
   while( *( s++ ))
      ++ length;
   return length;
}
```

Example with Twodimensional Arrays

```
void fill( p[][10], unsigned int n )
{
   for( unsigned int i = 0; i < n; ++ i )
      for( unsigned int j = 0; j < 10; ++ j )
         p[i][j] = i * j;
}

main( )
{
   int squares[5][10];
   fill( squares, 5 );
   return 0;
}
```

C^{++}

We have shown how to translate expresson AST for C into intermediate language.

If one wants to translate C^{++}, then one needs to make the following modifications:

- **class** types can have overloaded copy constructors and assignment operators. For such types, **memcopy** cannot be used. Instead on needs to call the user defined constructors and assignment operator.
- In addition, **class** types can have destructors. When a type has a destructor, it cannot be simply deallocated. One first has to call the destructor.

Since constructors, assignment and destructors are user defined, one has to follow the usual calling convention.

C^{++} : Explicit References

In C^{++}, the lr-value distinction was replaced by the reference/value distinction.

This means that **ref** becomes a type constructor, similar to **pntr**.

References complicate things a bit, but not much.

The real problem comes on the next slides.

C^{++} : Presence of Temporaries

When a value is returned but a reference is needed, C^{++} creates a temporary:

```
std::ostream& operator << ( std::ostream& , const X& x );
X( int x ); // constructor
std::cout << X(3) << "\n";
```

The C^{++} standard guarantees that temporaries exist until the expression is completely evaluated. This is needed for example in:

```
const X& max( const X& x1, const X& x2 )
   { return either x1 or x2; }


std::cout << max( X(i), max( X(j), X(k) ) ) << "\n";
```

There are three temporaries. Since we don't know in advance which one will be printed, all three of them have to exist to the end.

Presence of Temporaries (2)

In order to add to the pleasure, temporaries can have destructors, and may be conditionally created:

```
X x(1)
std::cout << ( randombool ? x : X(2) ) << "\n";
```

Here is a sketch of the modifications needed:

1. During semantic analysis, create a new conversion, which converses T to **constref**(T).
2. Before translation, create, for each conversion to a temporary, a short-lived stack variable. If the type has a destructor, then also create a register of type **bool**. Initialize it with false.
3. Translate the temporary conversions into an assignment to the corresponding local variable. At the same time make the corresponding **bool** true.

Differences with LLVM

Our register/stack machine is close to LLVM, which is used by the Clang compiler. The main differences are:

1. LLVM allows direct function calls, e.g. without writing and reading parameters on the stack.
2. LLVM strictly forbids multiple assignments to the same variable, even on different branches. This results in a normal form, called static single assignment (SSA). So called ϕ-functions are introduced at merge points.
3. LLVM uses one single pointer instruction `getelementpntr` for all pointer calculations.
4. In LLVM, the instructions have typed names, instead of the registers. It makes no difference.

Forgotten Topics

- I did not say anything about inheritance. It doesn't have impact on the compilation algorithm, as far as I can see. It makes method definitions a bit more complicated.
- The exception mechanism in C^{++} is very complicated. It has has to be implemented in such a way that exeptions that are not thrown result in zero cost.

  As a consequence, one cannot check exceptions by **ifs** after returning. The exception mechanism has to unwind the call stack and call destructors, when an exception is thrown.

  B.t.w. many people, I also, believe that the exception mechanism (and the concept of exception) in C^{++} is a failed design.