Physically Based Simulation

Alexander Schlögl

June 12, 2018

Contents

This is **my interpretation** of the lecture slides. I tried to be very verbose and explain everything, all while removing irrelevant parts from the lecture. Using this you should be able to pass the lecture easily. **However, I do not take responsibility for any bad results and will not be blamed from anyone. This was a lot of work and I did it to save others (especially students of following semesters) from having to do this themselves. Use this summary at your own responsibility.** If you have any feedback, feel free to create an issue on the git. I don't promise I will fix anything, but I will try.

1 Introduction to Compilers

A compiler is a program that takes code written in a source language, which is usually a high-level language, and transforms it into a target language, often object code or machine code. In the toolchain that transforms high level code to machine code, there also are other, compile-related programs, which may or may not work together with a compiler:

- **Interpreters & just-in-time compilers** often used for scripting languages (and Java)
- **Assemblers** translate the assembly language into machine code
- **Linkers** combine different object files into executable code
- **Loaders** load shared librabries (relocatable code)
- **Preprocessors** perform macro substitutions
- **Editors** are used to edit the code
- **Debuggers** allow step-by-step execution of the executable
- **Profilers** create memory and runtime profiles of the executable
- **Binary Inspection** allow inspection of the target code in the executable

1.1 Language Description

As a compiler needs to be tailored to the source and target language, describing languages is an essential part of building a compiler. Languages are usually defined at three levels:

- **Lexical level**: The lexical level of a language is defined by a dictionary. The dictionary contains a list of keywords and formats for the different data types, as well as valid variable names, usually defined using regular expressions.
- **Syntactical level**: The syntax of a language is defined by a grammar, describing valid control structures of the language.
- **Semantic level**: This describes the meaning of well-defined sentences in the language, and is often defined (in prose) in the language documentation.

1.2 Phases of a Compiler

A compiler operates in phases, split according to the tasks performed. Common phases of a compiler are shown in Figure 1. While the distinction between the phases is not always clear cut, keeping a degree of modularity is often beneficial.

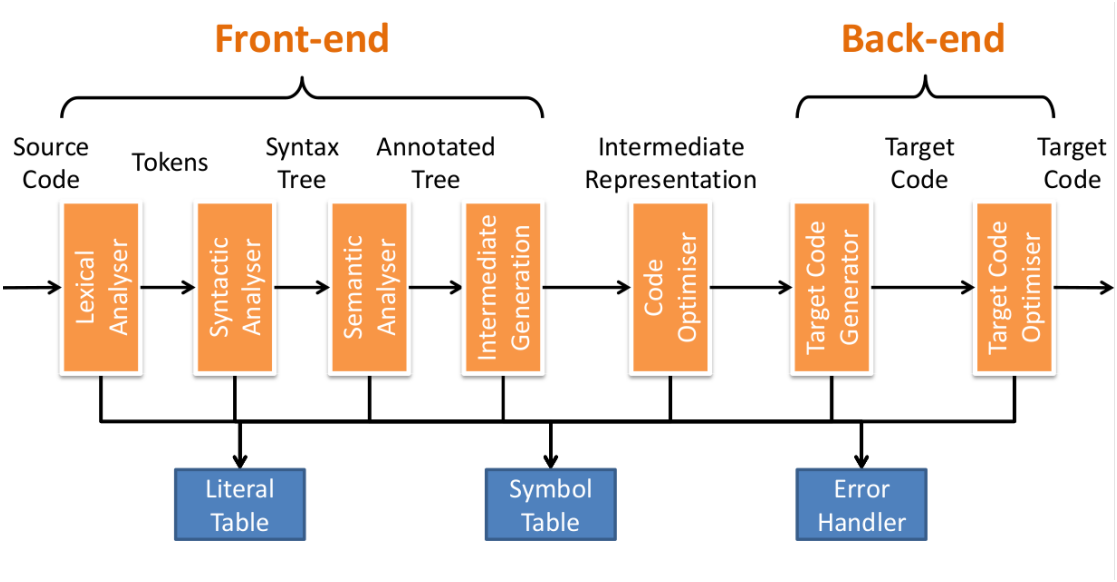

Figure 1: The phases of a compiler

Lexical Analysis (Scanning)

Scanning is the process of taking in a stream of characters and outputting a stream of tokens. This means splitting the source code into variable names, literals, etc. During this phase a compiler can also enter *identifiers* into the *symbol table* and *literals* into the *literal table*.

Syntactic Analysis (Parsing)

During parsing the stream of tokens is used together with a grammar to create a *syntax tree* and report syntax errors to the user.

Semantic Analysis

Semantic analysis checks the meaning of the program, and annotates the syntax tree with *attributes*, e.g. declarations and data types. Some semantics can only be checked while the programming is running (think dynamically allocated arrays), so not all errors are caught here.

Intermediate Code Generation

If multiple source and target languages or platforms are going to be supported it can be very beneficial to generate an intermediate representation that is independent of platform and language. Using an intermediate representation removes the need of creating a compiler for every combination of source and target platform. This reduces

the number of parts that need to be written from $m * n$ to $m + n$, where m is the number of source platforms and n is the number of target platforms. The required for adding a new source or target platform also drops from m or n to 1.

Intermediate representations also have the benefit of making optimization through multiple passes easier. A good example of intermediate representations being used is the LLVM compiler.

Code Optimizer

The code optimizer works on the intermediate representation by applying optimizations. An optimization is a transformation that improves performance of the code in one or more metric. Examples are dead code eleminition, consant folding or propagation, etc.

Code Generator

During this phase the actual target code is generated. This can be Assembler, or any other target language. Memory management, register allocation and instruction scheduling are the main challenges here.

Target Code Optimizer

In the last phase optimizations that are target specific are done. This includes replacing instructions with faster ones, data pre-fetching and code parallelization where possible.

1.3 T-Diagrams

A compiler is defined by three languages:

- **Host Language**: This is the language in which the compiler itself runs.
- **Source Language**: This is the language of the input.
- **Target Language**: This is the language the compiler produces.

Any and all of these three languages can be the same. If a compiler produces code in a language that cannot be run on the host machine (the one doing the compilation), it is called a *cross-compiler*.

Compilers are often represented using T-Diagrams, with the letters denoting the different languages. An example is shown in Figure 2.

Bootstrapping

In order to create a compiler for a new language, one can save some work by employing a process called *bootstrapping*. During this process, a compiler for the new language is written in the new language, as it can then make use of the many neat features that were included in the new totally not bloated language that is super awesome and will

end all other programming languages (/s). The language creators then write a quick and dirty compiler in a different language. This compiler doesn't have to be powerful, it only needs to be able to compile the "good" compiler. By combining the two we then get a (hopefully) correct, but inefficient compiler. Then we can recompile the "good" compiler with the minimal one to get the final version of the compiler, which can then compile all future versions of itself (until you include new language features). The full workflow is shown in Figure 2

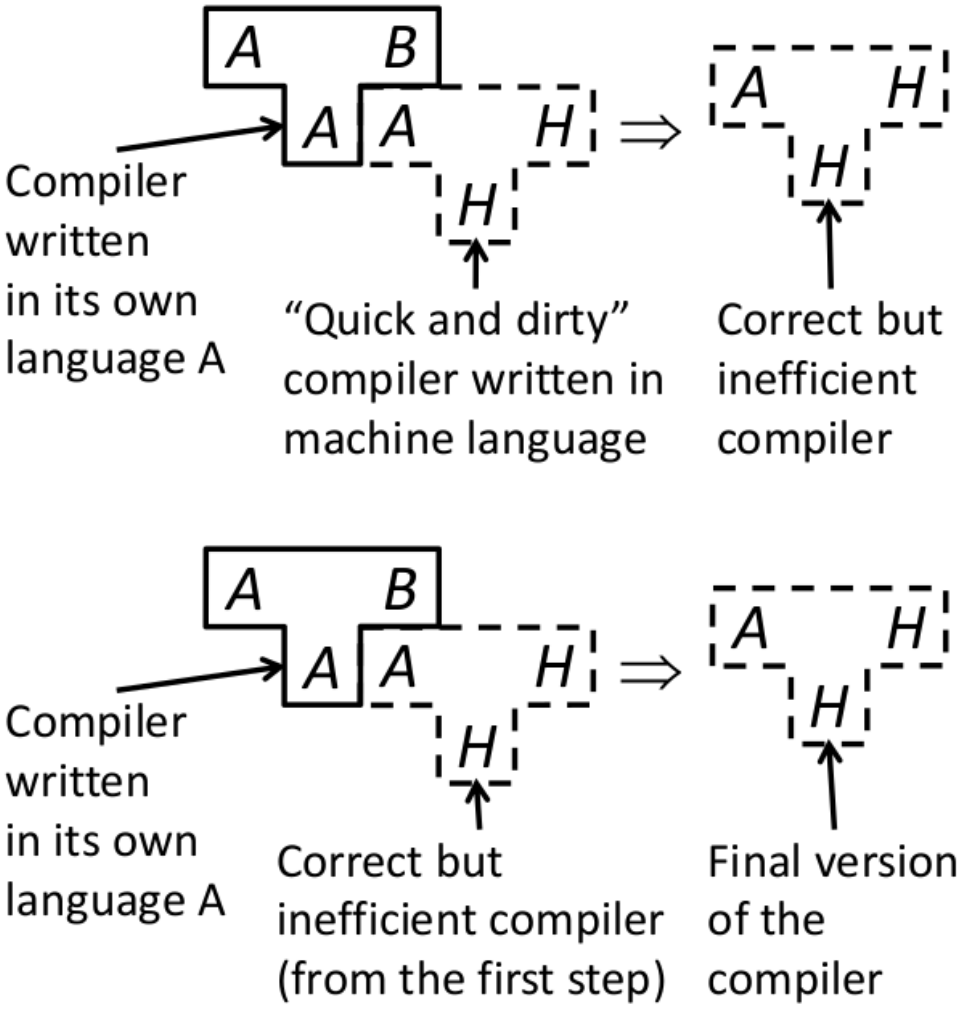

Figure 2: The bootstrapping process in less text and more images

Porting

Porting is the process of moving a compiler written it its own source language A from machine H to machine K. In order to do this, a compiler is written in the source language A with target language K, called a retargeted compiler. This is then compiled with the original compiler and produces a cross-compiler. The cross-compiler runs in language H and produces language K from source language A. The retargeted compiler is then compiled with the cross compiler to create a compiler for language A that runs in language K. The entire workflow is shown in Figure 3

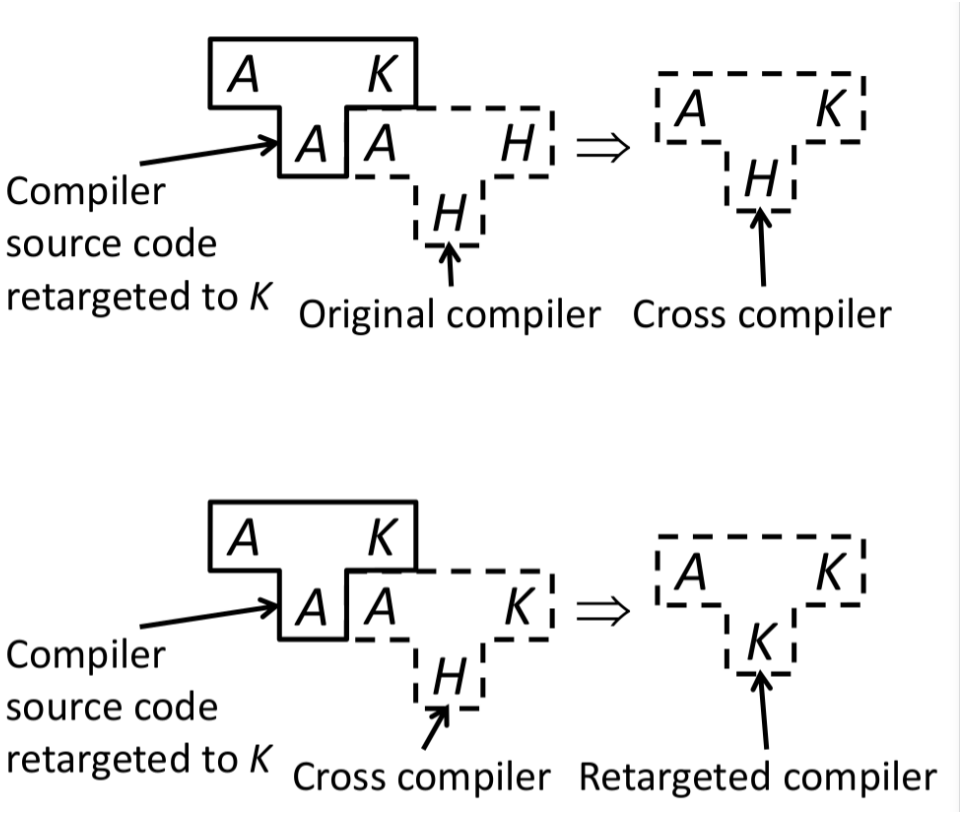

Figure 3: Porting a compiler

Combining T-Diagrams

Combining T-Diagrams is super easy and straight forward. Just replace the language (or letter) to the left of the T-Diagram with the one on the right. A few examples are shown in Figure 4

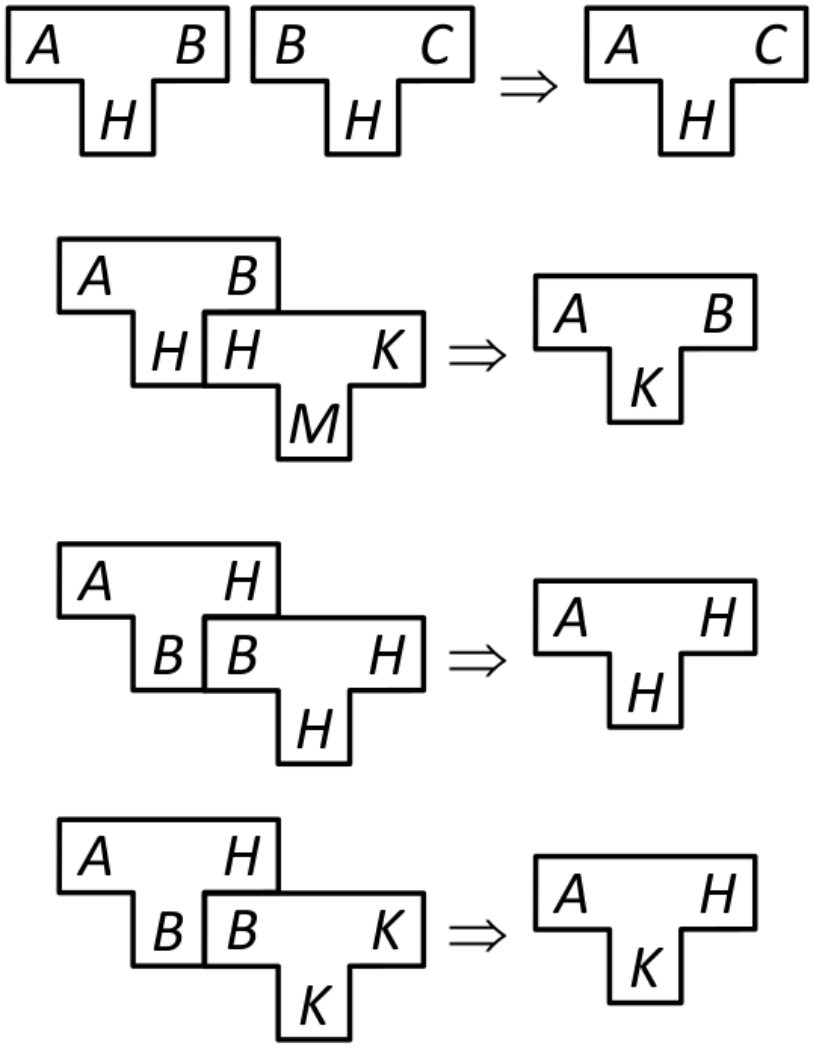

Figure 4: Combining T-Diagrams

2 Lexical Analysis (Scanning)

During scanning we want to split a stream of characters into a stream of tokens. The main goals are separating keywords, variables, constants, operators, etc. We do this by

creating regular expressions (regexes) for every type of token we want. These regexes are then converted to finite automata, which we can simulate nicely. We also want this phase of the compiler to be as efficient as possible, in order to save time for the more complex phases.

2.1 Tokens

Tokens in general are defined as an enumerated type, because you almost never define new keywords.

> If you are using a language where you can define keywords dynamically, seek help.

Common token types are keywords (if, else), special symbols (+,-,*), identifiers (for variables) and numbers (int, float, ...). These tokens can have one or more attributes, e.g. the string matched for the token (*lexeme*), the numerical value for numbers, the operator for special symbols, ... These attributes, as well as the type, are stored in a token record, which is just a simple container object or struct or whatever your language supports. The scanner is usually controlled by the parser, which just calls it again and again, each time receiving a single token and processing it (adding it to the token table, matching grammar rules, ...).

As most of us are in computer science because we are both lazy and smart (being less dumb than the average is sufficient), we want to do as little work as possible, and because some people actually program for fun, we don't have to. All we have to do is specify the different program tokens with regexes, and let an automatic tool build the lexical analyzer for us.

2.2 Regular Expressions

I will just skim regexes here because honestly, if you don't know them by now you have been cheating on all your CL exams, so why are you even reading this? All you need to start is an alphabet Σ, and the rest works as defined by this very simple grammar:

$$E = a \in (E \cup \{\epsilon\}) \mid E|E \mid EE \mid E^* \mid (E)$$

So regexes can do:

- letters
- or
- concatenation
- repitition
- subexpressions

as defined in this order. The notation for regexes is quite inconsistent, but this is the bare minimum, as defined in theoretical CS. Most tools that use regexes can do some syntactic sugar, namely $+$ for one or more repititions, . for any character, $[a-z]$ for any character (works with numbers as well), $\sim$ for set negation ($\hat{}$ for character classes) and $?$ for optional subexpressions. With this you can create most regexes quite comfortably.

Ambiguity

Some strings may be matched by several regular expressions. Usually the longest match is used (e.g. "forever" is not a keyword, even though it starts with "for") but sometimes even that fails. If there is doubt, most scanners will just use the rule that comes first, but for some more granular decisions token delimiters can be needed (e.g. line end or whitespace). However, the problem with these is that they should not be consumed but instead returned to the input stream. This is called a lookahead character. Usually one lookahead character is enough, but sometimes more is required.

2.3 Deterministic Finite Automata (DFA)

Seriously, you should know this. For an actual implementation, you just use a switch (or a series of ifs if you have to) for the incoming character and then go to the next state accordingly. You can also do the same thing with a transition table, and code less, but transition tables can get very large. Then you have to decide whether you want to implement sparse matrices (or use them if you can) or write a whole bunch of stupid switches.

2.4 Constructing DFAs from Regexes

This works just as it did in FLAT (or FML). You create an NFA from the regexes (usually by combining NFAs for the subexpressions), and then create a DFA from that. There's really not much to it, but if you're still unsure, just look at the FLAT slides, or ask you friends.

3 Parsing

We apparently didn't cover parsing in the lecture (honestly I wasn't there all too often so I don't know). Accordingly, this chapter, as well as the sub-chapters top-down parsers and bottom-up parsers aren't part of the exam. As I did write some stuff on top-down and bottom-up parsing before I learned this, I still have some information on them. Including the whole sections was not sensible, but I did leave in the introductions to the subsections. Anyway, here it goes:

Parsing is the process of converting the token stream into a syntax tree. The syntax tree contains all syntactic blocks of the program (if, while, for, functions, ...), and in its structure contains their relations. It is constructed according to rules defined in a context free grammar (I think context-sensitive grammars would also be possible, but the lecture slides only talk about context free grammars). You should remember how CFGs work from FLAT, so I won't discuss them here (they were skipped in the lecture anyway).

The finished syntax tree is one of the most important artifacts of the compilation process. After it is annotated with additional information (types, consant values, ...) it fully represents the program and its structure, and already allows some optimizations to take place.

The two main categories of parsing strategies are top-down and bottom-up.

3.1 Top Down Parsing

Top-down parsing starts with the first symbol of a token stream and follows the steps of a leftmost derivation of the grammar. These parsers traverse the parse tree of the program in pre-order form from the root to the leaves. As the next choice of derivation can sometimes be ambiguous, multiple methods of disambiguation exist. One is using *lookahead tokens*. By looking ahead in the token stream far enough to find a point where the rules become disambiguous, the parser can always find the correct derivation. This method causes some memory overhead, as the lookahead tokens have to be saved somewhere.

The alternative is using *backtracking*, and just deriving according to any possible rule, and backtracking if the choice was incorrect. This method potentially causes additional computation time due to the backtracking. This computational overhead makes them too slow to be used for practical parsers.

3.2 Bottom Up Parsing

If you read the previous chapter, you know that choosing the right derivation step is hard for top down parsing. This is the major strength of bottom-up parsers, because they can just take incoming tokens until a rule is fulfilled. They are named and differentiated by their derivation policy and and the number of lookahead tokens, but in general they all follow the same procedure. Their two main functions are *shift*, which moves a token (or terminal) to the top of the stack, and *reduce*, which replaces a string α on the stack with a nonterminal A (according to a rule $A \rightarrow \alpha$).

4 Attribute Grammars & Semantic Analysis

Attribute grammars are an extension to context free grammars. They allow the inclusion of additional semantic information, like types and values, which are very useful for

a compiler. Attribute grammars are expressed through *semantic rules* (also sometimes called *attribute equations*). Semantic analysis is usually much less formalized than syntactical analysis and still has to be handwritten everytime. Semantic information is also very important for code optimization, which makes semantic analysis even more complicated and important.

The syntax of attribute grammars is $X.a$, where a is an attribute associated with X. It might be a data type, a value, a memory location or pretty much anything. We differentiate between *static attributes* which can be evaluated at compile time (like most types, and object code of a procedure or function) and *dynamic attributes* which need to be evaluated at runtime (values of nonconstants and memory locations for dynamically allocated structures). The time at which an attribute can be fixed is called the *binding time* of that attribute.

An attribute equation or semantic rule is of the form

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_n, a_1, \ldots, X_n.a_k) \quad (1)$$

where all $X_i \in N \cup T$ (the union of nonterminals and terminals) are reachable by derivation rules from the start symbol, and $a_1, \ldots, a_k$ are attributes. In a less formal way this means the value of an attribute is a function of all other attributes. For a concrete instance the arguments of f are often only a subset of all attributes. Semantic errors arise and must be indicated during the evaluation of f. It is also important to note that the number of attribute equations for a semantic rule is not limited. (However, as the form of f is not really specified, one could argue that instead of using multiple attribute equations one could also use a more complex function).

For each grammar rule we can create a *dependency graph*, giving us the order in which attributes need to be evaluated. We can get the structure of this graph from the arguments of the individual f. Depending on the dependency directions, we have different types of attributes: *synthesized* and *inherited*.

4.1 Synthesized Attributes & S-attributed Grammars

An attribute a is synthesized synthesized if all dependencies point from the children to the parents in the parse tree. That means that all synthesized attributes can be evaluated in one bottom-up postorder traversal (children first, then the parents, a standard recursive call) of the parse tree. An S-attributed grammar is an attribute grammar where every attribute is synthesized.

4.2 Inherited Attributes

Inherited attributes are propagated downwards in the parse tree. They can be inherited from parents to children or between siblings. In order to fix all inherited attributes, a preorder (only parent to children) or a combined preorder/inorder traversal (parents to children and between siblings) is required. The order in which children are evaluated is important for performance.

4.3 Dependency Graph Construction

Dependency graphs can be constructed at compile-time from the parse tree, however doing this is complex and increases compile time. The dependency graph must be a directed acyclic graph (DAG).

Alternatively, the dependency graph can be based on an attribute evaluation order fixed by the compiler writer.

4.4 L-attributed Grammars

Several synthesized and inherited attributes can be calculated and propagatade in a single syntax tree traversal if:

- Synthesized attributes depend on inherited attributes and other synthesized attributes
- Inherited attributes do not depend on any synthesized attributes

Inherited attributes that depend on synthesized attributes require additional traversals.

For L-attributed grammars the attributes can already be calculated during parsing. A grammar is L-attributed iff:

$$\forall i, j : X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k) \tag{2}$$

that means that all attributes can be calculated if the previous nonterminals had all their attributes fixed. This allows the parser to fix all attributes in a left-right traversal for an L-attributed grammar. Any S-attributed grammar is L-attributed.

5 Symbol Tables

The symbol table is the most important inherited attribute of the compilation process, and the second most important artifact after the syntax tree. As it is used many times during the compilation the underlying data structure of the symbol table is very important. Its general appearance as a data structure is that of a dictionary (key-value pairs and stuff) and it needs to know three fundamental operations: *insert*, *delete* and *lookup*. Possible candidates for the data structure are simple linear lists, which have constant insert time but linear lookup and delete time, tree structures (in any n-ary form, b-trees, etc...), which have really nice lookup and insert times but complex deletes, and hash tables, which are just perfect for dictionaries if you are not super limited in space. Hash tables give us (almost) constant performance for all three operations.

5.1 Hash Tables

For those who don't remember the structure of a hashtable I will repeat it very quickly. All you do is allocate an array of constant size, ideally larger than the amount of data required by some magical factor (you should be able to find formulas and lecture material on it). Then all you need is a way of distributing your indices (the keys of the dictionary) evenly across the entire range. We achieve this by what we call a hash function. Now, this hash function does not have to be a cryptograhic hash, and it doesn't even have to be chaotic. All it has to do is provide an even distribution of the indices over the range of the array, with as little collisions as possible. For this reason, the perfect hash function for our data is domain specific, but we don't know that function beforehand. So, we just take some way of converting arbitrary keys to integers which gives us not too many collisions while being reasonably fast (and in compilers reasonably means as fast as possible). For example you could use the Horner schema to generate an integer from the identifier string. After you have the hash (of which you take the modulo with your index range in order to avoid having to look for segfaults for hours), you save your data at that index. This gives you a data structure with constant insert, delet, and lookup time, as all you have to do is calculate the hash to get the index. Amazing right?

Now, please put your throbbing erections away for a second, because we still have one problem to deal with: what to do if we get a collision. There are two fairly simple solutions to this problem. The first is not saving the data at the index directly, but instead saving a linked list there and putting our data in there. This is called **separate chaining**. Alternatively we can just look for an empty space in our hashtable somewhere else, which gives us less memory overhead, but increases the average distance of indices and hashes. This is known as **open addressing**.

5.2 Declarations and Scope

Now it's time to figure out what to put in our symbol table. For constant declarations we can put the value in our symbol table. Sometimes there is also a separate constant table for this, but its structure is very similar to the symbol table. For type declarations we add the alias and possbily the memory layout of objects in the symbol table (the lecture slides only mention the alias, the rest is just my common sense). For procedure and function declarations we can store the parameter and return value types. And finally, in some languages you can have implicit declaration by use (just like the C/C++ compiler warning you of an implicit function declaration if you get parameters wrong), which also need to be added to the symbol table. As you can probably imagine, these different value types for the symbol table might require multiple actual tables in the background, but we usually only consider one abstract table for our purposes.

Scope and Lifetime

Two very important but usually implicit declaration parameters that need to be stored in the symobl table are scope and lifetime. The scope tells us in which context it is legal to use a variable, and this is usually solved by having different symbol tables for different scopes (e.g. for different functions and a global symbol table). If a variable is referenced it is first searched in the symbol table of the samllest surrounding scope, and this search is then gradually exetended outward, until the variable is either found, or until it is declared nonexistant (the bahaviour in this case depends on the language).

The lifetime of a variable tells us how long the memory for it has to be allocated, and when it can be freed. This is not very important for variables that live on the stack, as that memory is freed automatically as the program leaves the function the stack frame belonged to, but for heap variables this is something that has to be kept in mind. The memory of stack variables needs to be freed as long as that variable is no longer valid. This is handled differently in different languages. Some have a garbage collector (e.g. Java), while others require the user to do this or face the consequences of memory leaks. Global variables have a lifetime that is independent of functions, and live as long as the program is running.

> Almost all combinations of scope and lifetime are possible. A variable can have function scope but global lifetime (see static function variables in C).

Special Points

For many modern languages, runtime environments manage the memory, freeing the user from doing this. This has its benefits but brings its own problems, at it menas less control for the user.

One small thing to note is also that *extern* declarations in C do not allocate memory anywhere. The program will simply fail if no memory location is known for the declared variable or function.

5.3 Block Structured Languages

Most modern programming languages are organized in blocks. A block is any area that can contain declarations (e.g. any area in brackets for C). In these languages, every block is a new level of scope, and blocks can also be nested. Variables declared in a block are valid in this block's scope, and all nested block's scope. Also, they are always added to the most closely nested block's symbol table (the one containing the declaration).

For languages that require declare before use, the symbol table can be built during parsing. For others, it requires a separate pass. Symbol tables of nested scopes can contain a reference to the directly containing scope to decrease lookup time. The

symbol table can also be released upon leaving the scope, which removes the need for delete operations. This can sometimes prevent reusing uninitialized values (however the memory of the new table might still contain garbage), and reduces the time required upon leaving the scope. Using linked symbol tables also allows redaclaration of variables at a different scope, which is usually a desired property.

Same-Level Declarations

Declaring the same variable multiple times at the same level is illegal. However, for many script languages that have no strict typing there is no syntactic difference between declaration and assignment. This forces them to allow "redeclaration" of varibables, although it is really only an assignment that changes the type of the variable.

The way that multiple declarations are processed is also different between languages. Some languages use **sequential declaration**, which means they process one declaration after the other, adding variables to the symbol table as they go. Another variant is **collateral declaration** which means that all declared variables for the current block are searched, processed and then added to the symbol table together.

For **recursive declarations** the prototype has to be processed and added before the function body. Some languages require a forward declaration of the prototype. This is especially prevalent for mutually recursive functions (recursive functions that call each other).

5.4 Types and Type Checking

Type checking ensures that the program satisfies the type rules of the language. But before we can check types we need a definition of what a type is. A data type is a definition of an allowed set of values and some parameters (**integer** $\cup \{+, *\}$).

Declarations also either specifiy a type explicitly or implicitly. An explicit type declaration should be clear, and implicit declarations usually use some kind of keyword like *var*, *auto*, or you use a scripting language that doesn't really declare types anyway. Implicit types need to be inferred from the content, and if they can't be, the compiler should throw an error.

There usually are predefined types for languages, and most languages give the programmer the ability to add user types. When defining user types, we differentiate between simple types (like enums, or subrange types), and complex datatypes (structs, classes, ...). Declaring complex data types requires some type of *type constructor* (which should just be the constructor itself).

Recursive Data Structures

Recursive data structures are a special case for symbol tables, as they need to be added to the symbol table before they can be completely processed. As an example, think of a simple implementation of a tree structure. The tree is built up of nodes, and a node

needs to reference its children. Because we can't create a new type for every child, the children must be of the same type as the parent, and therefore this type needs to be used before it is finished. As we can't copy the cild into the parent by value, because that would give us data structures of variable size, we need to either store them by a reference or a pointer (which are essentially the same thing).

Type Equivalence

We often need to check whether two expressions have the same type, in order to see if an assignment is legal or not. For this we have three methods:

- **Structural equivalence**, which checks whether two types have the same memory layout, and syntax tree. This can lead to some quite funky shenanigans, due to reinterpretation of memory, but can be used for cases where no type names are available.
- **Name equivalence**, which checks if both types have the same declared name and simple type (I think pointers count too). This is easier to evaluate, but requires type names to be added to the symbol table. Also, the compiler must generate a unique internal name for every declaration.
- **Declaration equivalence**, which equates every type name to *base type name*, which is either a predefined type or a type expression. This requires the symbol table to support an operation which gets the base type for a given type.

> Declaration equivalence only creates type aliases, and is just fucked up. C uses declaration7;57M7;57m equivalence for simple types, and structural equivalence for stucts and unions.

Using this type equivalence, we can create a type checker.

5.5 Polymorphism

Polymorphism, as opposed to *Monomorphism* allows variables to have multiple types at once. The benefits and shapes of polymorphism are not part of this lecture. Two parts of it that are relevant for this lecture are *overloading* and *templates*.

Overloading

Overloading is the use of the same operator for multiple operations (e.g. the operator $+$ is used for integer and float addition). Some languages allow programmers to overload user functions. This requires the symbol table to include types as parameters for the lookup for function declarations.

Type Conversion and Coercion

Type rules can be extended to allow the conversion between data types. Most languages supply this capability for simple types, but some also allow the programmer to create new conversion rules, especially for user defined types.

Type coercion is when the compiler or runtime environment supplies this mechanism implicitly (as opposed to e.g. casting in C). Languages that support user defined types and polymorphism usually follow the subtype principle, where every subtype can be coerced to a supertype (as it is a more specific variant of the supertype). This principle is the main point of contact that most programmers have with polymorphism.

Templates

Templates (also sometimes called generics, depending on the language), are *type parameters*, which allow the definition of functions where the parameter types aren't fixed in the declaration. However, they usually can be restricted to subtypes of a common supertype, which requires the type checker to perform elaborate pattern matching.

5.6 Intermediate Representeations

Intermediate representations (IR) are used in compilers in order to make some operations easier. They reduce the number of actual compilers needed to be written for multi-language or multi-platform compilers. Additionally, a nice IR makes code optimization much easier than optimizing the source or target code.

There exist many IRs, and they are grouped in high-level, low-level, and those in between. Some compilers also use more than one IR (for different purposes).

Three-address Code

This is a low level IR that is very close to (platform independent) Assembler. You might think that doing anything in Assembler can't be easier than doing it differently, but for a platform independent IR it is quite nice. Three address-code is also somehow much less tedious than Assembler, which makes you wonder why we don't use it instead of Assembler in the first place. The answer to this is: because the different Assembler dialects were there first. Figure 5 shows an annotated example of three-address code.

Three-address code also follows a very computational logic-y design, in that it is designed like a binary tree. The general approach is that it's easy to generate code for an expression if you can generate code for its subexpressions. However, this code is not necessarily optimal (or even fast), but we can mitigate this by optimizing the IR code.

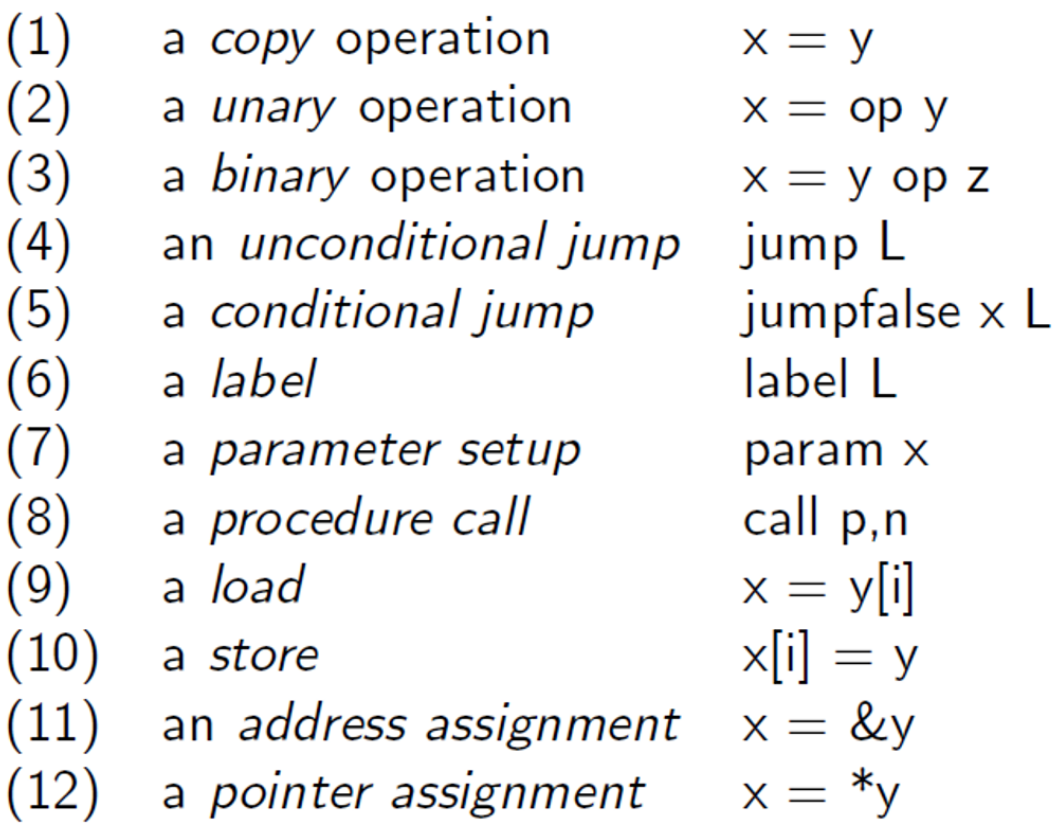

Figure 5: Three-address Code

Three-address code is to Assembler what Python is to Java (sort of). It's less of a hassle, but doing things naively is way slower. This is why we optimize our IR code.

Three-address code is generally implemented in one of three ways:

- **Quadruples:** Here you store the two arguments, the operator and the target. This means that the compiler can shuffle operations around easily, but that the targets need to be entered into symbol tables. The representation also requires a bit more space. Figure 6 shows some example code.
- **Triples:** Here we only store the arguments and the operators. The targets are given by the index of the current operation. This makes shuffling a bit harder for the compiler, but saves space, and doesn't require a symbol table. Figure 7 shows the same example as before.
- **Indirect Triples:** Here the execution order is stored separately from the actual operations, just like a database table with a foreign key. This makes shuffling

easier while still not needing a symbol table. However, the data structure complexity and memory requirement is larger than for straight triples. Figure 8 shows an example.

```
t1 = a > b
jumpfalse t1 L1
max = a
jump L2
label L1
max = b
label L2
```

Operation	Argument 1	Argument 2	Result
>	a	b	t1
jumpfalse	t1	-	L1
assign	a	-	max
jump	-	-	L2
label	-	-	L1
assign	b	-	max
label	-	-	L2

Figure 6: A three address code implementation using quadruples

5.7 Basic Blocks

A basic block (BB) is a sequence of intermediate statements which are always executed together. This requires a BB to not contain any branching statements (jumps), except for the last statement. BBs are used for analysis and optimizations that are sensitive to control flow, as they are the control flow can only jump between BBs.

A BB starts with a leader, which is one of the following:

- the first statement in the program
- the target statement of a jump
- the statement following a jump

A BB then goes from each leader up to (but not including) the next leader. Figure 9 shows an example.

5.8 Beyond Three-address Code

There are quite a few more intermediate representations. One example is the IR used in the LLVM compiler, which sits somewhere between high and low level. It is very similar

```
t1 = a > b
jumpfalse t1 L1
max = a
jump L2
label L1
max = b
label L2
```

	Operation	Argument 1	Argument 2
(0)	>	a	b
(1)	jumpfalse	(0)	(4)
(2)	assign	a	max
(3)	jump	-	(5)
(4)	assign	b	max
(5)	...	...	...
(6)	...	...	...

Figure 7: A three address code implementation using triples

to Assembler in some aspects, but also already allows for the definition of functions, has types, and is all around a pretty nifty thing. I included an example in Figure 10.

Implementing IR

Low level IR is fairly straightforward to implement (represent in memory, not generate target code from), as it is close to machine language and consists mostly of simple mechanisms anyway. More high level IRs also usually contain constructs that are a bit more tricky to implement, like functions and the like. For this you convert the AST into a logical control flow tree, which is physically implemented using node sharing. This makes the representation conservative on memory, but slightly trickier to implement. But you should be fine, you have are in the Master's Programme after all ;)

6 Optimization Theory

Optimization is probably the most complex phase of the compiler, and the one with the most ongoing research. It is meant to improve the performance of the code in one (or a combination) of the non-functional parameters, like execution time, memory usage etc. (the functional parameter being the correctness of the result).

```
t1 = a > b
jumpfalse t1 L1
max = a
jump L2
label L1
max = b
label L2
```

	Statement
(0)	(85)
(1)	(86)
(2)	(87)
(3)	(88)
(4)	(89)
(5)	...
(6)	...

	Operation	Argument 1	Argument 2
(85)	>	a	b
(86)	jumpfalse	(85)	(89)
(87)	assign	a	max
(88)	jump	-	(90)
(89)	assign	b	max
(90)	...	...	...
(91)	...	...	...

Figure 8: A three address code implementation using indirect triples

6.1 Optimization Classification

Optimizations can be split into two major categories: *machine dependent* and *machine independent*. The border between the two is sometimes not clear cut, as machine independent techniques can have special cases based on the individual platform. The main trends in architecture are machine dependent, like instruction level parallelism (ILP) and memory access or pipeline optimization. There are however some machine independent optimizations, which are applicable to all platfroms, like eliminating redundant work, or using less expensive operations through arithmetic transformation.

Optimization can also be split into categories based on the level at which they are performed: *source level*, *high/mid/low-level IR level*, and Assembler/machine code level.

6.2 Optimization Scope

Different optimization techniques operate on different scopes. The simplest scope is *local*, which means the technique operates on a single basic block. The next larger scope is *regional*, meaning multiple blocks but less than a procedure. Unintuitively named, *global* or *intraprocedural* techniques operate on entire procedures. Finally, *interprocedural* or *whole-program* techniques do operate on two or more procedures, the entire program.

Figure 9: An example of basic blocks

6.3 Analysis and Transformation

An optimization pass is actually split into two phases: *analysis* and *transformation*. The analysis finds places and ways to apply an optimization, the transformation then actually changes the code. Local techniques can interleave analysis and transformation steps, due to the defined order of statement execution in basic blocks. Larger regions require the entire analysis to finish before a transformation can be made. Also, the transformation may invalidate previously performed analyes.

6.4 Qualities of an Optimization

The three main qualities of an optimization are *safety*, *profitability* and *opportunity*. We want our optimizations to preserve the result of the computation, actually speed up the computation, and be efficient in candidate location and application.

Safety

Safety is the most important aspect of an optimization. If the "meaning" of the code is not preserved, the compiler is worthless. In theory observational equivalence means the following: *Two expressions M and N are* **observationally equivalent** *iff in any context C where both M and N are closed (all their variables are fixed), evaluating $C[M]$ and $C[N]$ either produces identical results or neither terminates.* Above statement should be fairly clear if you think about it for a bit, but if not, just keep in mind that M and N must produce the same result whenever both are closed (they have no free variables).

As this statement is too broad for practical use, we reduce this to observational equivalence in their *actual program context*. Optimization gets easier the more context is given, and a lack of context leads to very general (ie. slow) code.

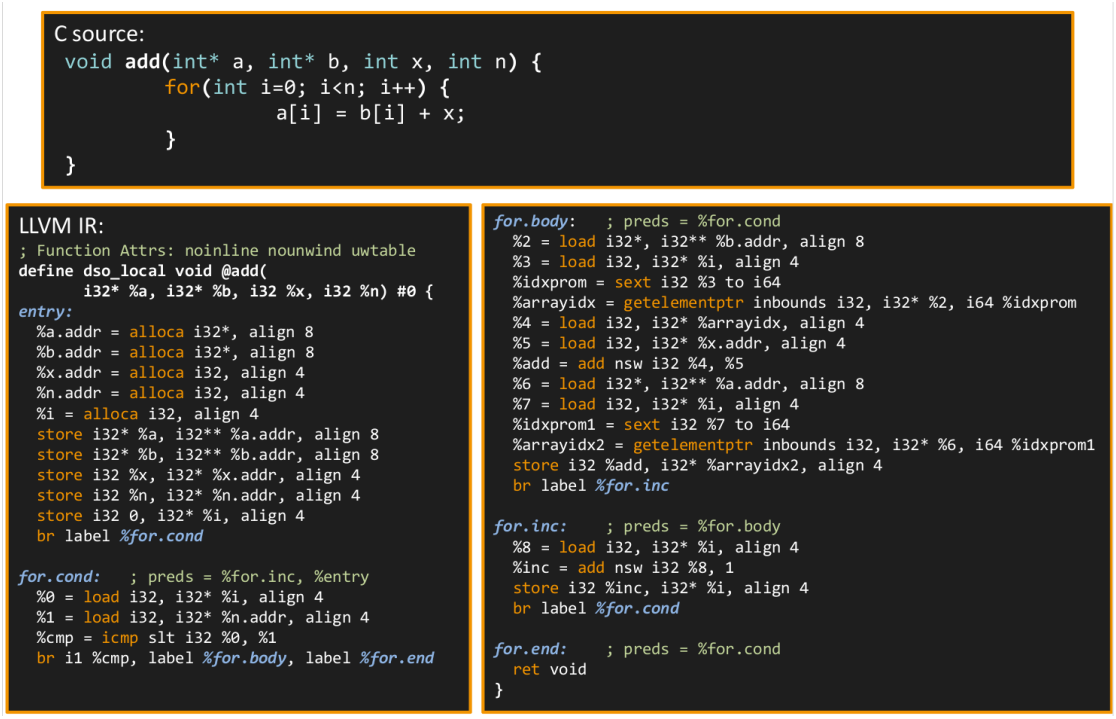

Figure 10: An example of LLVM IR code

Profitability

The compiler only needs to apply optimizations when it actually helps the non-functional parameters. As the desired combination of non-functional parameter performances can be complex, so can deciding on the profitability of an optimization. In some cases, profitability can be proven (e.g. constant folding), sometimes we have a good guess (pulling loop invariants out of the loop), and sometimes we can't tell easily (function inlining).

The compiler oftenn uses heuristicts to reach conclusions on the question of profitability.

Opportunity

Before it can perform an optimization, the compiler first needs to find the places in the code where the transformations can be applied. Ideally, we would want the compiler to check every possible location, but this is not feasible with desired compile times. How to specify and locate optimization opportunities efficiently is a big issue. Some analysis forms do look at every single operation, but in a very fast and efficient manner, while others perform more in-depth analysis only on parts of the code.

6.5 Control Flow Graph

In order for control flow analysis we need to understand the control flow graph. The nodes in the control flow graph are basic blocks, and the edges are branches. The different paths from the first to the last node are then the different execution paths of the covered part of the program. Figure 11 shows an example.

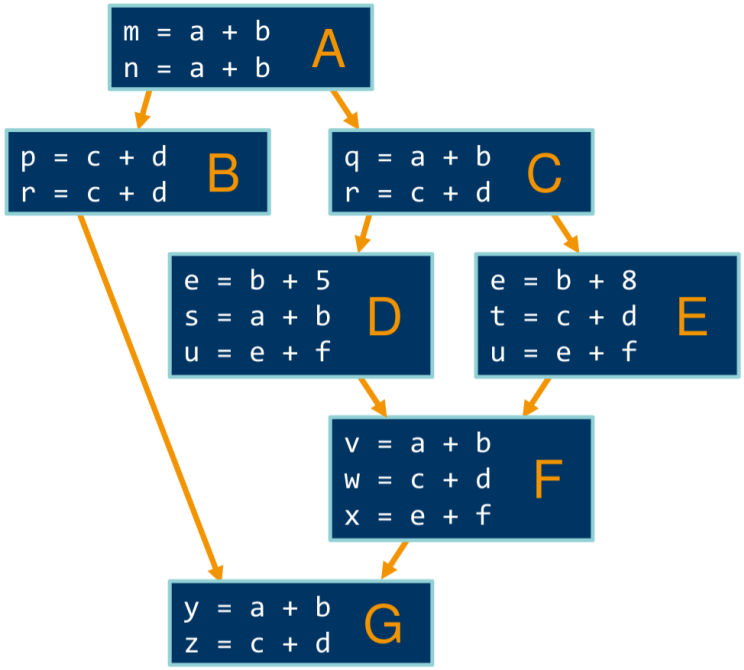

Figure 11: An example control flow graph

Control flow graphs also give rise to the definition of *Extended Basic Blocks* (EBBs):

- An EBB is a tree of BBs $b_1, b_2, \ldots, b_n$
- b_1 can have any number of predecessors
- All other b_i have a single unique predecessor (but possibly multiple exits)
- The EBB is only entered at the root

In Figure 11 {A,B,C,D,E}, {F} and {G} are EBBs.

An EBB contains 1 or more paths. $b_1, b_2, \ldots, b_n$ is a path iff:

- For all edges (a, b_i) in the CFG $a = b_{i-1}$
- b_i has 1 predecessor: b_{i-1}

Figure 12 shows the longest path in the example CFG.

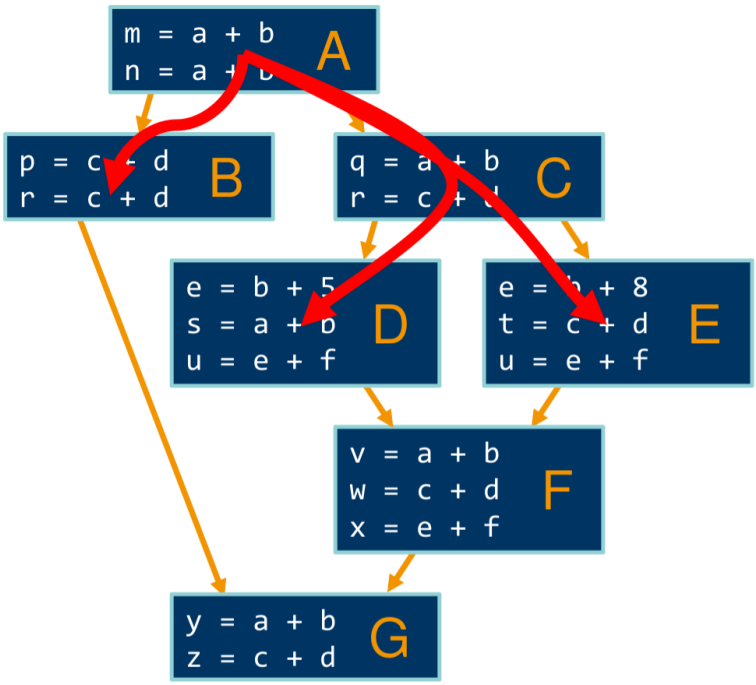

Figure 12: The longest path in the example CFG

7 Optimization Techniques

Here I will cover the different optimization techniques discussed in the lecture.

7.1 Redundant Expression Elimination

The easiset way to optimize code is by eliminating redundant expressions. Redundant expressions are defined as follows: *An expression x (op) y is redundant iff it has already been evaluated along every execution path from the procedure's entry and the operands have not been redefined since.*

This means that if an expression is redundant we can remove the current evaluation and replace it with a reference. In order to do so, we first need to prove that the expression is really redundant, and then rewrite the code to eliminate the redundant evaluation.

Local Value Numbering

The key idea behind proving the fact that an expression is really redundant is assigning an identifying *value number* $V(n)$ to each expression. $V(x+y) = V(j)$ iff $x+y$ and j always have the same value.

An implementation of this can be very simple. For the basic block you start with an empty hash table. Then you process expression by expression, calculating the hash $\langle op, V(arg_1), V(arg_2)\rangle$ for every expression e. If the hash is already contained in the hash table, you replace e with a reference (the value for the hash key), otherwise add the hash and calculate the result. If arg_1 and arg_2 are constants, evaluate and replace e with the immediate. Figure 13 shows an example, and the largest potential issue of local value numbering. This issue can be avoided by renaming, as shown in Figure 14. As Figure 14 states, this issue is avoided by the immutability of SSA variables.

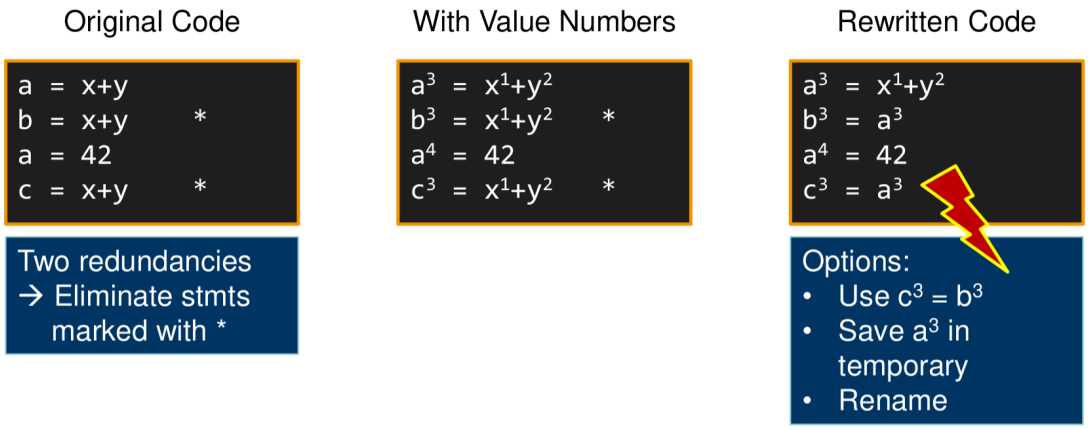

Figure 13: An example of local value numbering without renaming

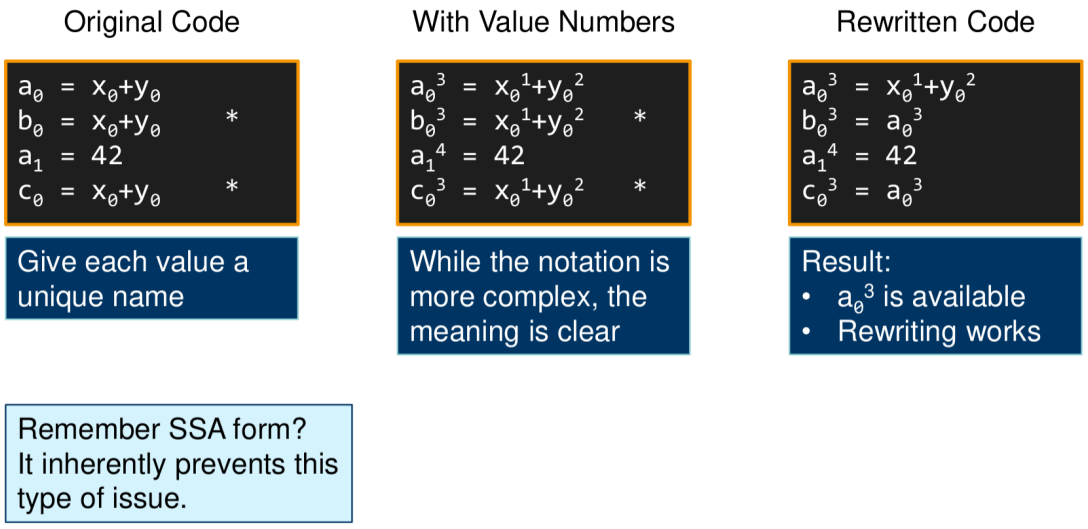

Figure 14: An example of local value numbering with renaming

Local value numbering (and value numbering in general) can be extended to also fold constants by including an information bit that records when a value is constant.

If it is constant the expression can be evaluated at compile time and replaced with a load immediate or an immediate operand. There is no stronger local algorithm.

Algebraic identities can also be handled by this, but potentially many different identities may need to be checked. For this a tree can be used to increase lookup efficiency. If an identity is found, the result can be replaced by the input value number.

Safety of Local Value Numbering is always given, because the hash table starts empty and is filled as expressions are processed. If the value number for an expression is found in the table, it has been computed (at least once) before and can be referenced. Changing values are somewhat problematic, but as long as the target temporary for the original expression e is still valid, all expressions e' can reference it without a problem. This holds true for basic blocks, because if any statement executes, they all execute, in a predetermined order. For areas larger than basic blocks this analysis becomes a bit more complex.

Profitability of Local Value Numbering is given if reuse is cheaper than re-computation. Reuse is only cheaper if it does not cause a memory page spill or copy (which is expensive), but in practice this is often assumed to be true. Local constant folding is always profitable, because loading an immediate uses a register, just like recomputation, but loading an immediate operand saves even that. The profitability of algebraic identities is given if we can eliminate an entire operation. If not, it depends on the architecture and the speed of different operations.

Opportunities for Local Value Numbering are found by linearly scanning each basic block in execution order (ie. no backtracking). Each operation needs to be considered as an opportunity for expression elimination, constant folding/propagation and algebraic identity transformation. The cost for this can be reduced by multiple techniques. Using a hashtable for the value numbers reduces the complexity of the lookup to $\mathcal{O}(1)$ per operand and operator, and algebraic identity lookups can be sped up by using trees for identities.

Superlocal Value Numbering

Local value numbering misses many opportunities for eliminating redundant expressions. Figure 15 shows the missed elimination opportunities in the example CFG from Section 6.5.

Superlocal value numbering works by applying the same algorithm as local value numbering to every path in the EBB. The hash tables are then not emptied for every basic block, but reused for the entire path. However, this method still misses quite some opportunities for elimination.

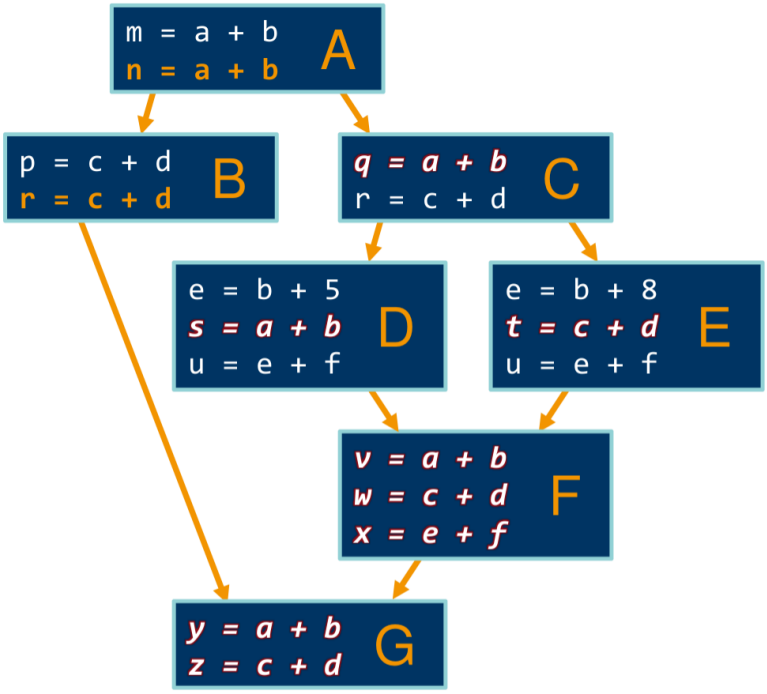

Figure 15: The missed elimination opportunities in the example CFG from Section 6.5