Construction of Compilers

Alexander Schlögl

June 20, 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.

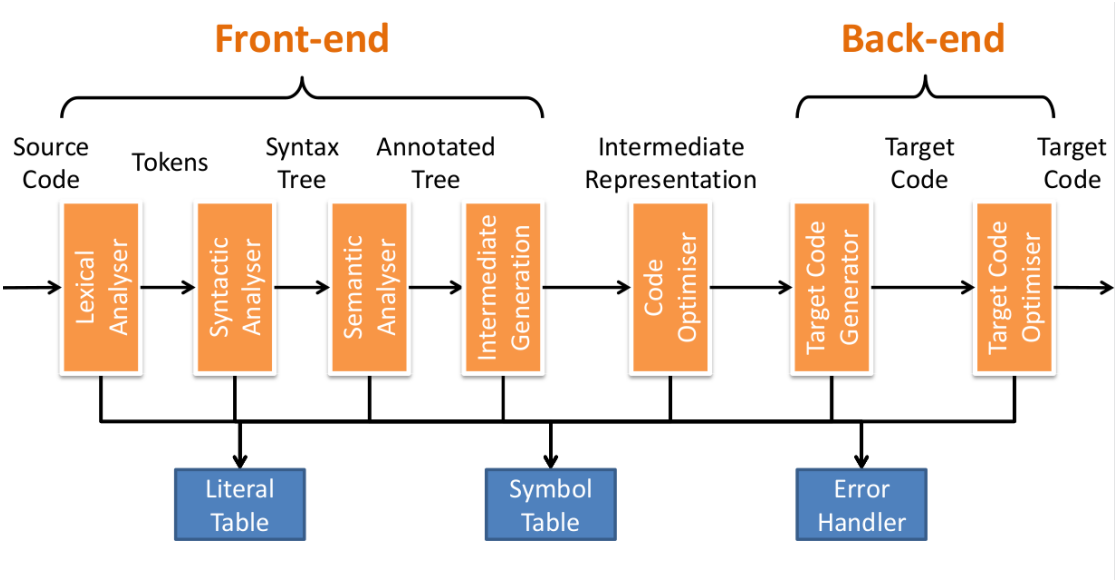


Figure 1: The phases of a compiler

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

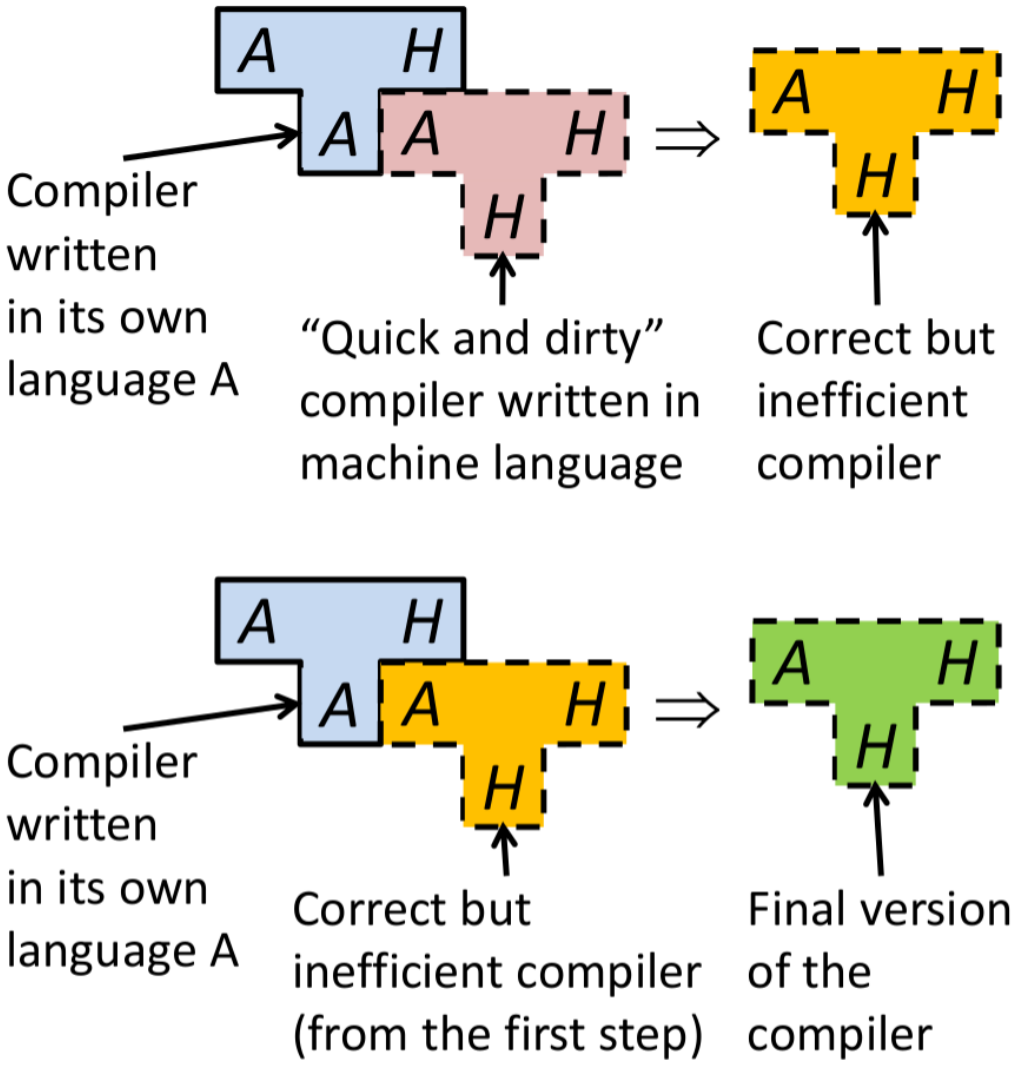


Figure 2: The bootstrapping process in less text and more images

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

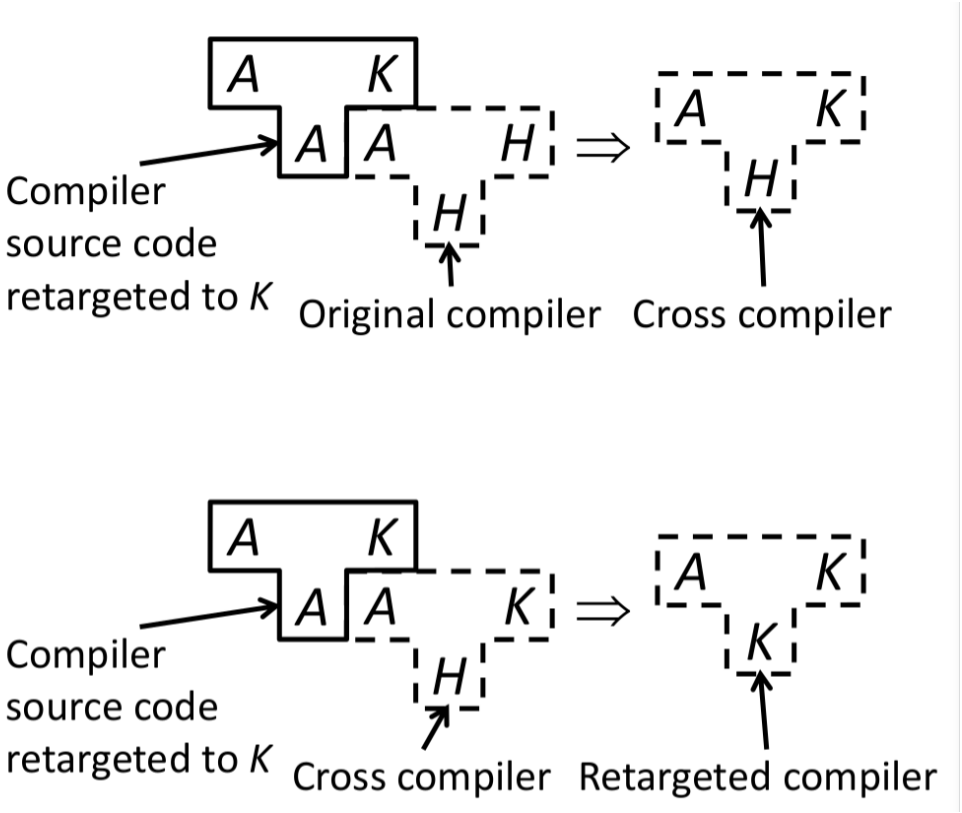


Figure 3: Porting a compiler

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

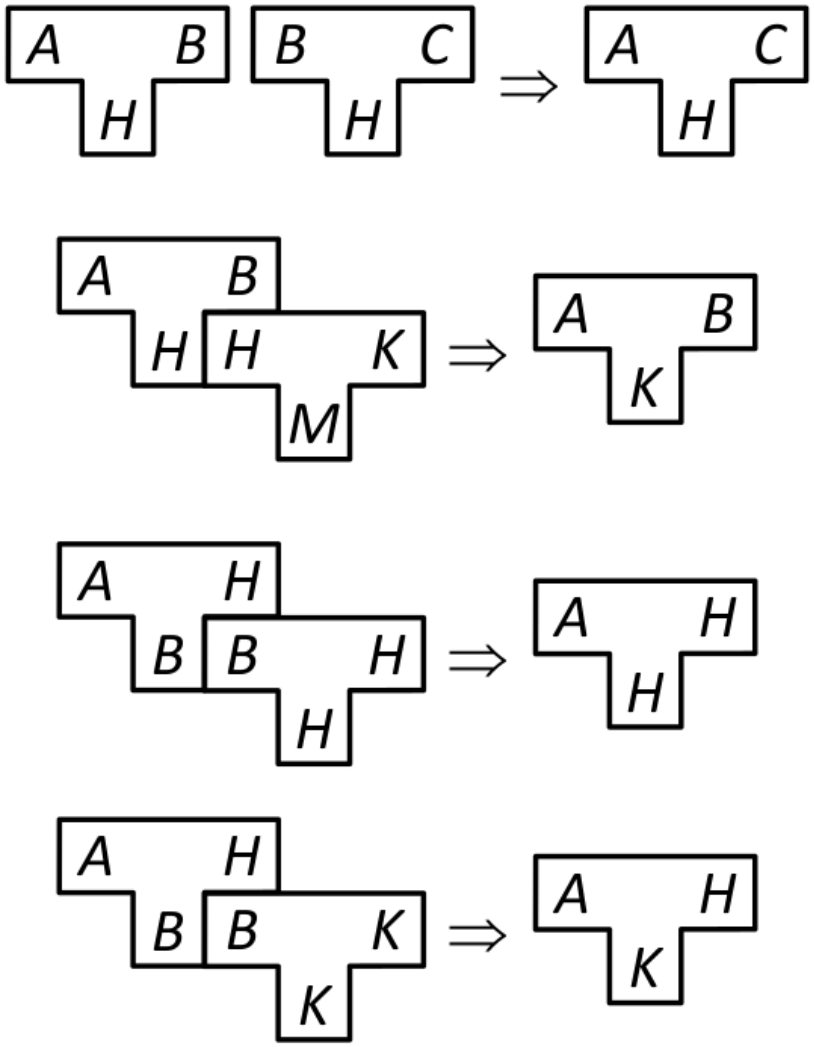


Figure 4: Combining T-Diagrams

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

Almost all of the parsing stuff from the lecture was covered in FLAT, which is a prerequisite for this course, so you should know that. I will only cover the parts not covered in FLAT, and maybe include some refreshers needed for that. The rest will only be included briefly, so you know what to look for.

3.1 Context Free Grammars (CFGs)

These are covered extensively in FLAT. The only thing might be the Backus-Naur-Form (BNF): $S \to A|b|Cd$. Derivations, parse-trees, derivation strategies are all things you should know. You can also probably figure out how to implement this (efficiently) in C.

3.2 Ambiguity

Ambiguity for CFGs is when you can derive a word with multiple different parse trees. We don't want ambiguity in our language, which can be achieved by left factoring, not part of the exam. Associativity of operators removes ambiguity, and for non-associative operators we need to prohibit chains of more than one operator. The dangling else is a classic ambiguity problem, and we need rules to take care of this (a classic solution is *most closely nested*). There is also the term *inessential ambiguity*, which is ambiguity that the semantics don't depend on. This kind of ambiguity can just be removed.

3.3 Extended BNF (EBNF)

EBNF allows us to more comfortably express the things we want to express. They contain syntactic sugar like shown in Figure 5, which is equivalent to $term\ (addop\ term)^*$. They also contain optional constructs, expressed by brackets [ ].

$$exp \to exp\ addop\ term \mid term$$

- ENBF left associative form

$$exp \to term\ \{\ addop\ term\ \}$$

- EBNF right associative form

$$exp \to \{\ term\ addop\ \}\ term$$

Figure 5: EBNF form for star notation

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 Representations

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 6 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

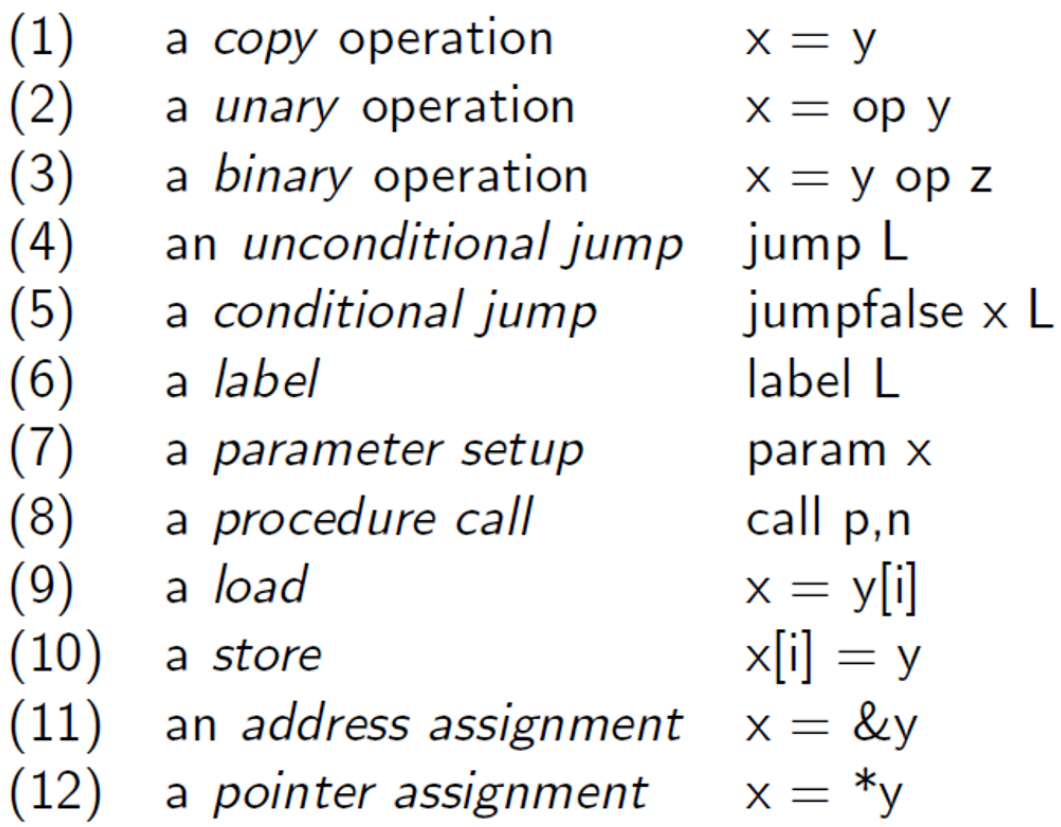


Figure 6: Three-address Code

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.

> 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 7 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 8

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 9 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 7: 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 10 shows an example.

```
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 8: A three address code implementation using triples

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 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 11.

Static Single Assignment Form

Static Single Assignment Form (shortened to SSA) is a property of intermediate representations. In this form, every name is defined by exactly one operation or expression, and every operand referes to exactly one definition. To reconcile principles when branches meet, we add subscripts for uniqueness and insert ϕ-functions at merge points. The ϕ-functions choose one of its operands based on the taken execution path. Figure 12 shows what this means. In the actual output code, the ϕ-function is simply the reuse of the same register, which is written to by only one taken path.

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

```
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 9: A three address code implementation using indirect triples

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 ;)

5.9 Call Graphs and The Stack

We all know how to call a function (I hope). But some of us may not know what needs to be done when a function is called. We need to allocate space somewhere to store parameters and return values (that are not stored directly in registers), store a return address, and then jump to the actual function.

To represent the calling relationship between functions we use a call graph. The call graph is simply a graph that shows which functions call which other functions. It can be either static (determined at compile time), or dynamic (generated at runtime and always a tree). Call graphs can also have varying degrees of context sensitivity (accounting for calling context or not). Figure 13 shows a nice visualization of a call graph.

In order to manage function calls we need *The Stack*. *The Stack* stores information that cannot (for various reasons) be stored in registers. Maybe the information is too large, maybe we need *The Stack* in order to do some magic, it doesn't matter. *The Stack* is there for us.

It starts at some point in memory and grows downwards (towards lower addresses). All we need to interact with *The Stack* are two pointers: a stack pointer and a frame pointer. The first points to the current *end* of *The Stack*, while the latter points to the beginning of the current stack frame (the end of *The Stack* after this function call

Figure 10: An example of basic blocks

ends). When a new function call is entered, the old frame pointer is stored (so we can restore the state), the old stack pointer is stored in the frame pointer, and *The Stack* is extended downwards (according to the memory we need), while updating our stack pointer accordingly. **??** shows a graphical representation of *The Stack*.

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).

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

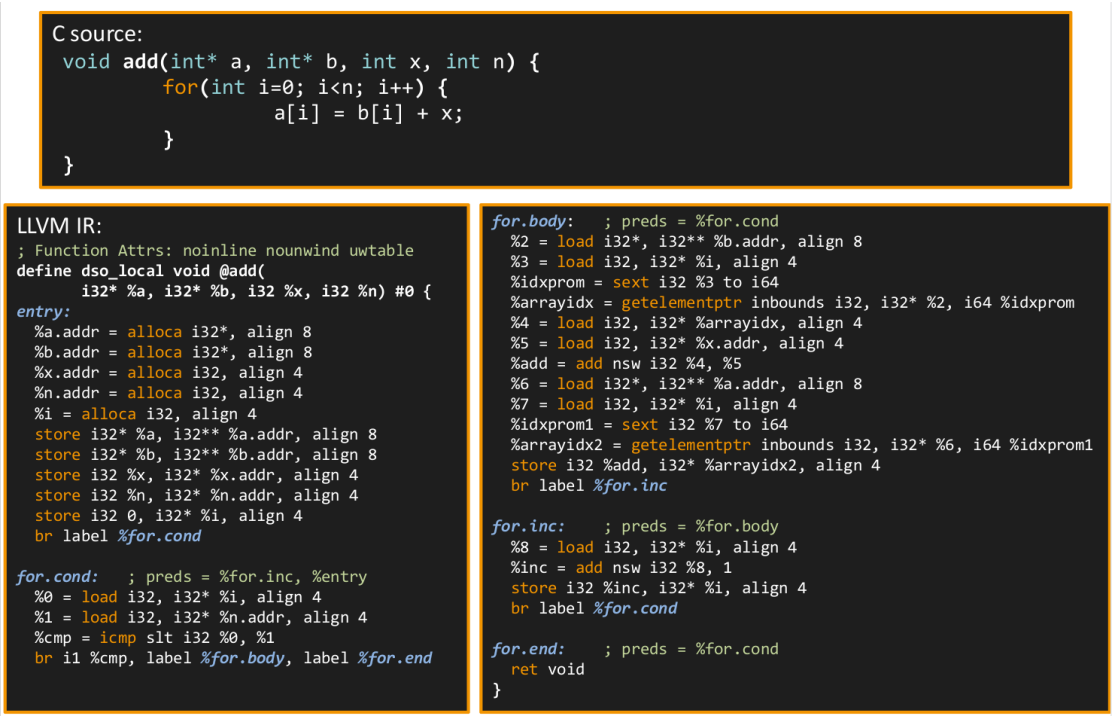


Figure 11: An example of LLVM IR code

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.

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.

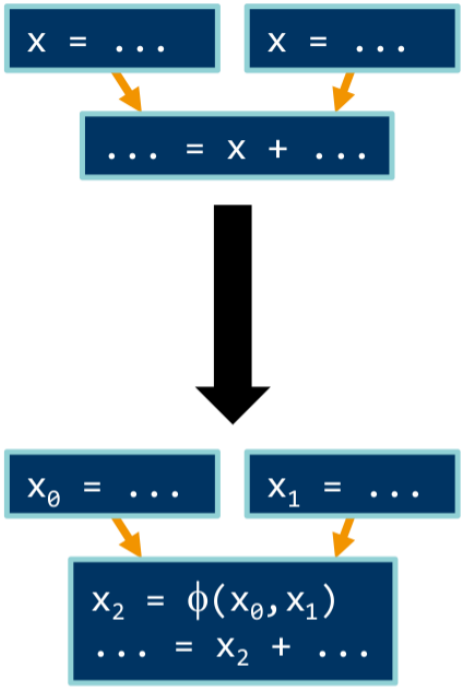


Figure 12: An example reconciliation in SSA using a ϕ-function

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.

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

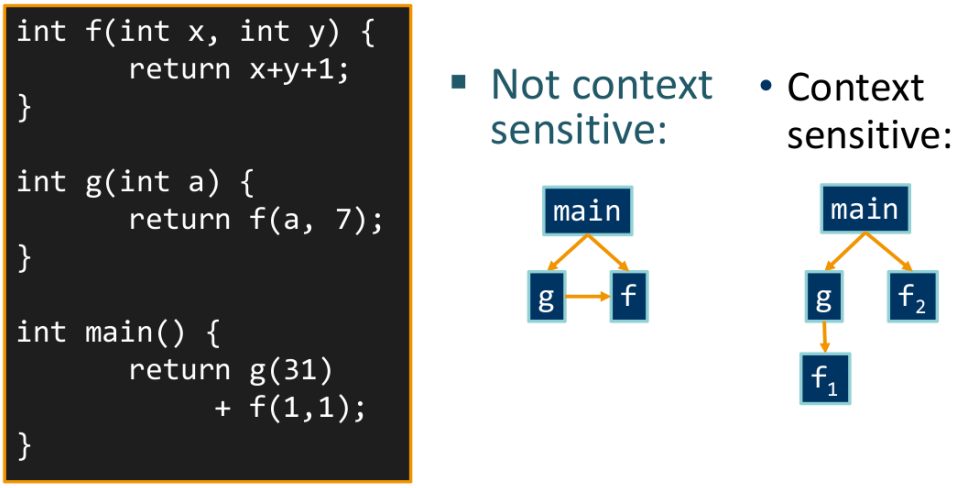


Figure 13: A context sensitive and a context insensitive call graph

(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 15 shows an example.

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)

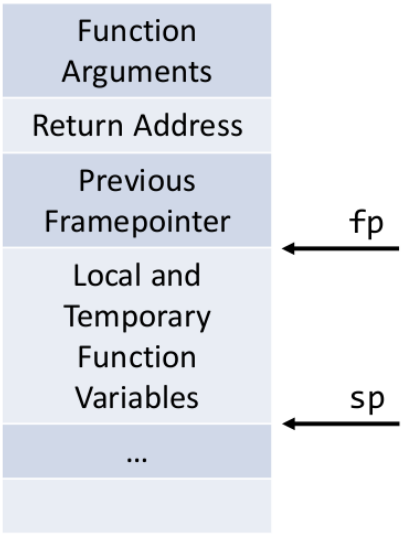


Figure 14: *The Stack*

- The EBB is only entered at the root

In Figure 15 {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 16 shows the longest path in the example CFG.

6.6 Dominators

In a flow graph, x dominates y iff every path from the entry of the control-flow to the node y includes x. By definition x dominates x. We call the set of dominators for a node DOM and DOM(x) always contains at least one element (x).

For any node x there must be a $y \neq x$ in DOM(x) closest to x, unless $x = n_0$ (x is the first node in the flow graph). We call y the *immediate dominator* of x, and the notation is $y =$IDOM(x). The *Dominator Tree* is then drawn by connecting every node to its immediate dominator.

Calculating Dominators

A node n dominates m iff it is on every path from n_0 to m. We can calculate DOM by iteratively appling the following equation:

$$\text{DOM}(n) = \{n\} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p)) \tag{3}$$

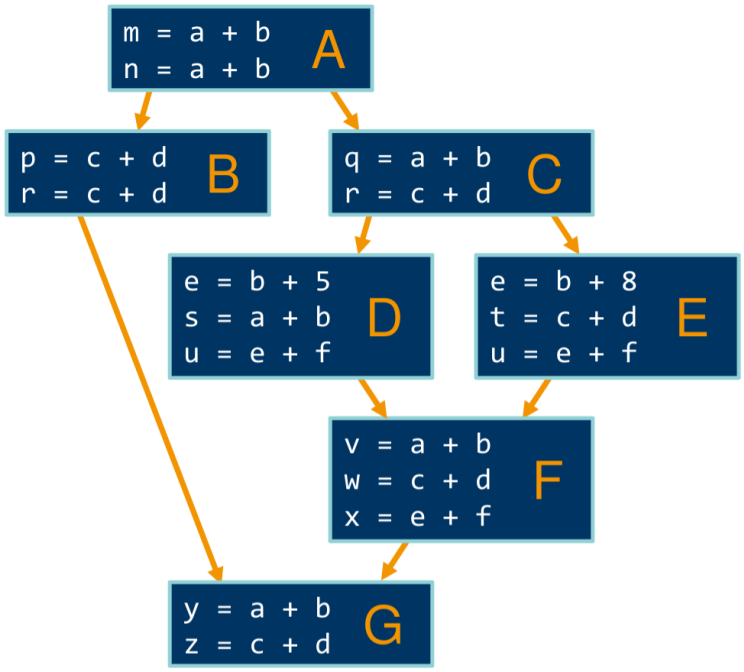


Figure 15: An example control flow graph

with DOM$(n_0) = n_0$. By just applying this equation to all nodes until DOM set changes we will be guaranteed to get a result. Figure 17 shows an example of CFG and dominator tree, with DOM and IDOM for every node.

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.

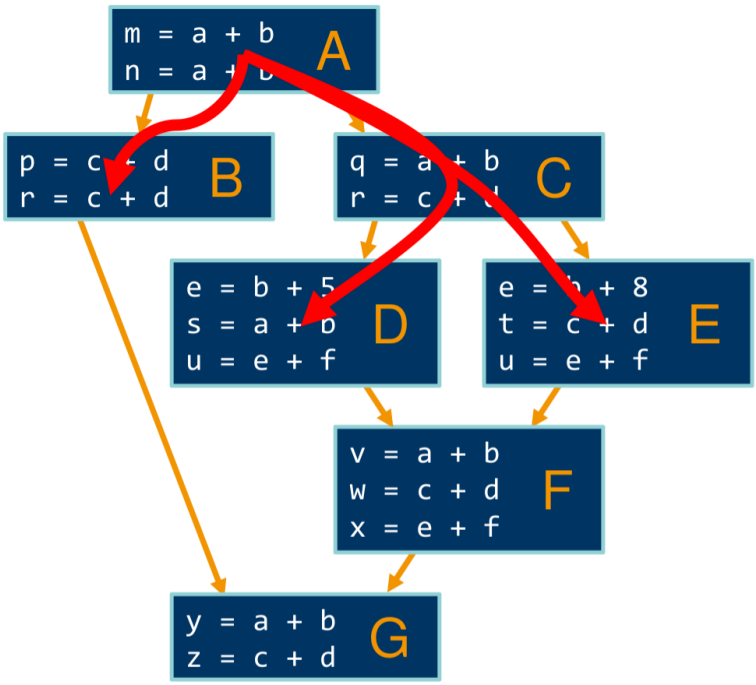


Figure 16: The longest path in the example CFG

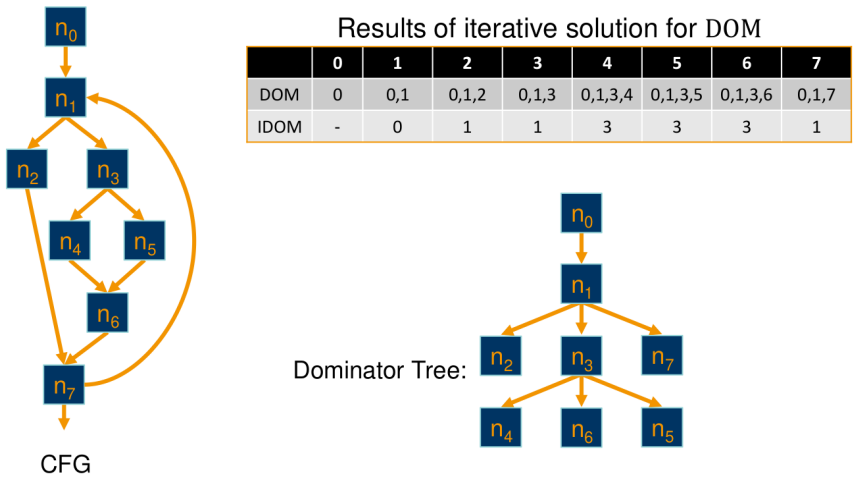
Local Value Numbering (LVN)

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 18 shows an example, and the largest potential issue of LVN. This issue can be avoided by renaming, as shown in Figure 19. As Figure 19 states, this issue is avoided by the immutability of SSA variables.

LVN (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.



	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDOM	-	0	1	1	3	3	3	1

Figure 17: A CFG with DOM, IDOM and dominator tree

Safety of LVN 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 LVN 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 LVN 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.

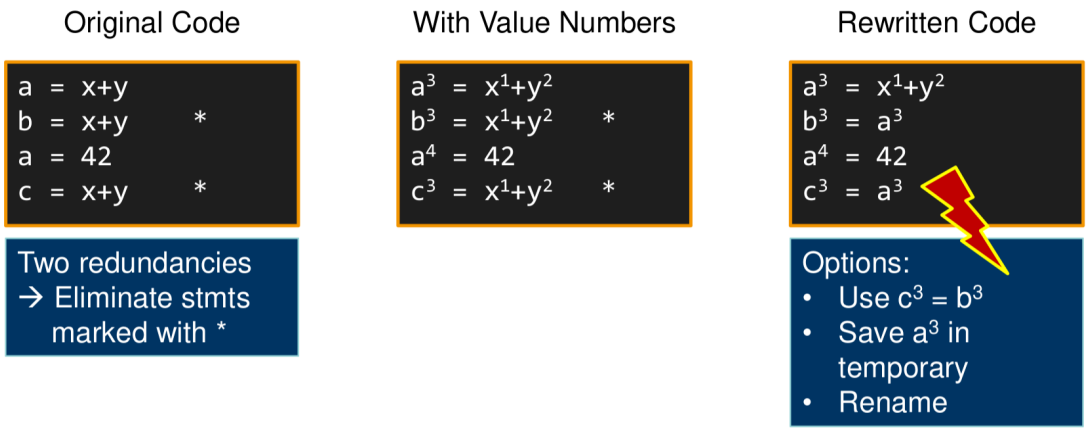


Figure 18: An example of LVN without renaming

Superlocal Value Numbering (SVN)

LVN misses many opportunities for eliminating redundant expressions. Figure 20 shows the missed elimination opportunities in the example CFG from Section 6.5.

SVN works by applying the same algorithm as LVN 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.

Dominator Value Numbering (DVNT)

We want to decide which facts are true for the current basic block, but that information depends on the taken execution path. The dominators however give us information on things that **always** happen before the current block.

So, in order to make use of even more opportunities, we can apply SVN, retaining the use of scoped hash tables in the path, and start each node with the table of its IDOM. Unfortunately, still no values flow back along edges (around loops), leading to quite a few missed opportunities. Constant folding and algebraic identities can be applied as before.

DVNT should only be applied in combination with SSA form IR code. This is due to the fact that changes to the operands in blocks between the IDOM and the current block can make this technique have incorrect results. In SSA form however, variables changed in multiple blocks are run through a ϕ-function. This ϕ-function causes the resulting variable to have a completely new value number, distinct from the value numbers of the variable in all previous blocks. The assignment of a new value number is by definition and is meant to avoid problems such as this, which are caused by uncertainty over the state of the variable.

DVNT finds more redundancy than SVN, but still misses some opportunities and does not remove loop carried constant expressions. The additionally eliminated expres-

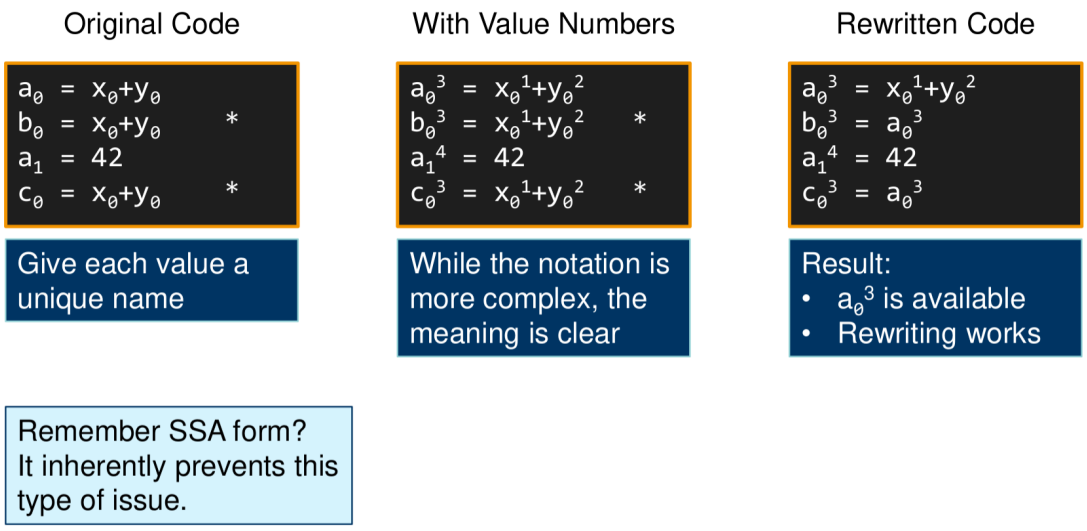


Figure 19: An example of LVN with renaming

sions are shown in Figure 21.

7.2 Data Flow Analysis

Data flow analysis is a collection of techniques for reasoning about the *run-time flow* of values *at compile time*. These techniques always operate on a graph. Problems are trivial on a single block, as the graph only has one node. Interprocedural problems use the call graph of the current function. Global problems use the control-flow graph of the program. Data flow problems are then formulated as simultaneous set equations, which can be solved easily by iterative algorithms. Note that I said easily, not quickly.

Data flow analysis determine a property of the program, and we usually are interested in a *meet over all paths* solution:

- What is true on every path from the entry node?
- Can this event happen on any path from the entry node?

The iterative solvers start with an approximate solution, which will converge to the exact solution in a finite number of iterations if equation is a lattice.

Semi-lattice

A semi lattice is a set L and a meet operator $\wedge$ with $\forall a, b, c \in L$:

$$a \wedge a = a \tag{4}$$
$$a \wedge b = b \wedge a \tag{5}$$
$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{6}$$

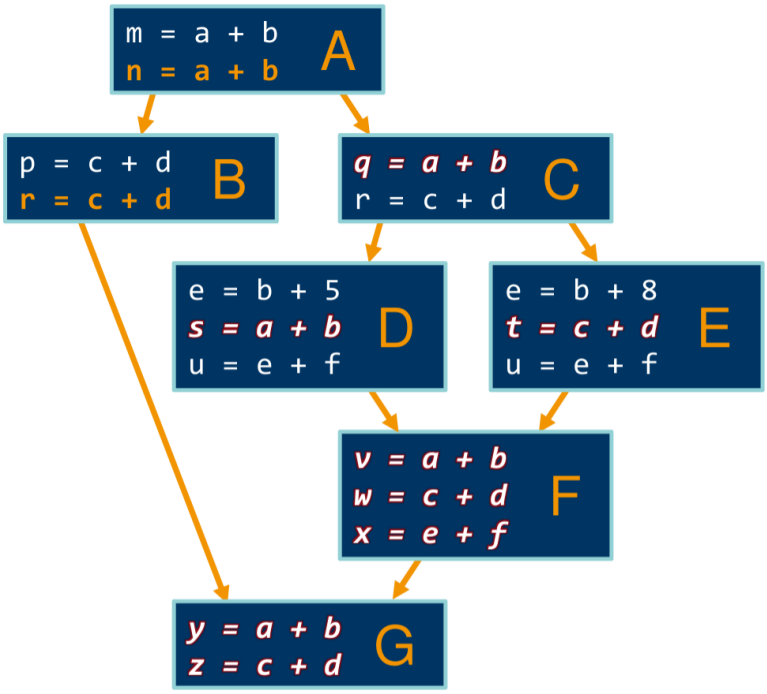


Figure 20: The missed elimination opportunities in the example CFG from Section 6.5

$\wedge$ imposes an order $a \geq b \leftrightarrow a \wedge b = b$ and also has a defined bottom element: $\bot \wedge a = \bot \rightarrow (a \geq \bot)$ So, a lattice is just a set with an operator that has idempotence, is commutative, associative, defines an order and has a bottom element (e.g. set union with the largest set as bottom element).

Lattices aren't really used for any technique covered in the lecture. However, I wanted to include the definition because knowing this might come in handy.

7.3 Computing Live Information

A value v is live at p iff there exists a path from p to some use of v along which v is not redefined. This means that if the value of v is read later on, without being rewritten before that, we need to keep the value of v.

In data flow we use the sets $LIVEOUT$ and $LIVEIN$ for this:

$$LIVEOUT(b) = \cup_{s \in succ(b)} LIVEIN(s) \tag{7}$$

$$LIVEIN(b) = UEVAR(b) \cup (LIVEOUT(b) \setminus KILLED(b)) \tag{8}$$

$$LIVEOUT(n_{|N|-1}) = \emptyset \tag{9}$$

where $UEVAR(b)$ is the set of variables used in b before being defined in b (the variables that must exist before b is executed), and $KILLED(b)$ is the set of variables defined in b. The lecture slides use $NOTKILLED$ instead of $KILLED$ which is

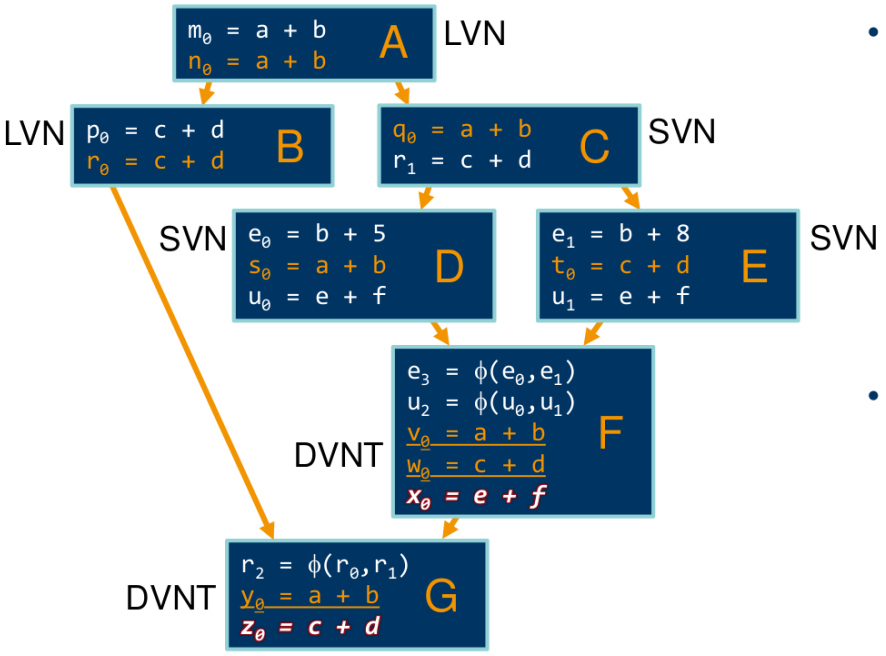


Figure 21: The expressions eliminated by DVNT, the results of ϕ-functions have a different value numbers than the operands

just the inverse, but that only changes the operation between $LIVEOUT(b)$ and $NOTKILLED(b)$ from set difference to an intersection.

A very simple algorithm to compute this is to update $LIVEIN$ and $LIVEOUT$ repeatedly, until the sets for no block have changed. This works, but is highly inefficient. The order of traversal also has a large impact on convergence speed, but finding the optimal order is NP-complete.

Worklist Algorithm

Figure 22 shows the pseudocode for the algorithm, which should be easy enough to understand. If W does not contain duplicate (use your standard set implementation) this converges, as the number of possible additions to the sets (and therefore to the worklist) is finite. The equations for the sets are monotone (they are lattices, see above), and $UEVAR$ and $KILLED$ are costants for b. Also $LIVEIN$ and $LIVEOUT$ are finite sets.

Using Live Information for Redundancy Elimination

Now that we know which variables and values are live, we can eliminate unneeded stores. As soon as a variable is not live anymore, we can reuse that register. To do

```
W = { all blocks }
while(W ≠ ∅)
        let b ∈ W
        W = W \ {b}
        recompute LIVEOUT(b)
        t = LIVEIN(b)
        recompute LIVEIN(b)
        if LIVEIN(b) ≠ t then
                W = W ∪ pred(b)
```

Figure 22: The pseudocode for the worklist algorithm

this, we first solve $LIVEIN$ and $LIVEOUT$ for the blocks, and then compute local liveness incrementally from there. If the target of a store is not in the current $LIVE$ set, we can delete that store. If all stores are eliminated, we can delete the reserved space for it.

This elimination is safe because if $x \notin LIVE(s)$ at the point of some store s, its value is not used without being written first. The safety of this technique relies on the correctness of $LIVE$. By searching the blocks linearly, for store operations, and computing $LIVE$ at every point s of a store, and eliminating the store if possible, we can speed up the execution. This assumes that not doing the store costs less than doing it (which is almost always true).

7.4 Global Redundancy Elimination (GRE)

By calculating $AVAIL(b) = \cap_{p \in pred(b)}(DEEXPR(p) \cup (AVAIL(p) \setminus KILLED(p)))$, with $DEEXPR(b)$ being the subexpressions defined in b and $KILLED(b)$ being the set of expressions killed in b. An expression is killed if one of its operands is changed. Computing this requires making some changes to the table that stores the expression references, in order to speed up our lookup. $DEEXPR$ and $KILLED$ can be calculated locally (again, the lecture uses $NOTKILLED$, but I think this is more concise). We initialize $AVAIL(b) = \emptyset$ and solve forwards (instead of backwards for $LIVEOUT$). Figure 23 contains an example, including an exercise task.

Applying GRE saves us from doing any redundant work, as shown in Figure 24.

	A	B	C	D	E	F	G
pred	-	A	A	C	C	D,E	B,F
DEEXPR	a+b	c+d	a+b, c+d	b+5, a+b, e+f	a+8, c+d, e+f	a+b, c+d, e+f	a+b, c+d
KILLED	-	-	-	e+f	e+f	-	-

- $\text{AVAIL(F)} = (\text{DEEXPR(D)} \cup (\text{AVAIL(D)} \cap \text{NOTKILLED(D)}))$
  $\cap (\text{DEEXPR(E)} \cup (\text{AVAIL(E)} \cap \text{NOTKILLED(E)}))$
  $= \{ \text{b+5, a+b, e+f} \} \cup (\{ \text{a+b, c+d} \} \cap \text{E}\backslash\{ \text{e+f} \} \})$
  $\cap \{ \text{a+8, c+d, e+f} \} \cup (\{ \text{a+b, c+d} \} \cap \text{E}\backslash\{ \text{e+f} \} \})$
  $= \{\text{a+b, c+d, e+f}\}$
- AVAIL(G): Calculate this on your own

Figure 23: An example of $AVAIL$ calculation

8 Code Generation

Now that we have (fairly) optimal IR code, we can start generating machine code for the target architecture. Obviously we want to generate the most efficient assembler code possible. To this end, in true divide-and-conquer fashion, we split the code generation into three phases: Code Generation & Instruction Selection, Instruction Schedulig, and Register Allocation. In general, the problem of optimal code generation is NP-complete, but in practice we can find some good solutions based on heuristics.

We also want to keep the generated code as portable as possible. While this is hard due to the inherently machine dependent nature of target code, we can help ourselves, as we will see later on.

In most RISC architectures we can do the same thing in multiple different ways. As an example, registers can be copied using some different methods (with *i2i* being the binary copy operator):

- $i2i\ r1 \rightarrow r2$
- $addI\ r1, 0 \rightarrow r2$
- $lshiftI\ r1, 0 \rightarrow r2$

If different operators are assigned to different units in hardware, the selection of operators can have a huge impact on performance. Due to this, simply walking through the code and translating it line-by-line (as is done for generating IR code from the AST) is not the way to go. An example of this is shown in Figure 25, with *A* being code generated line by line, and *B* being intelligently generated target code.

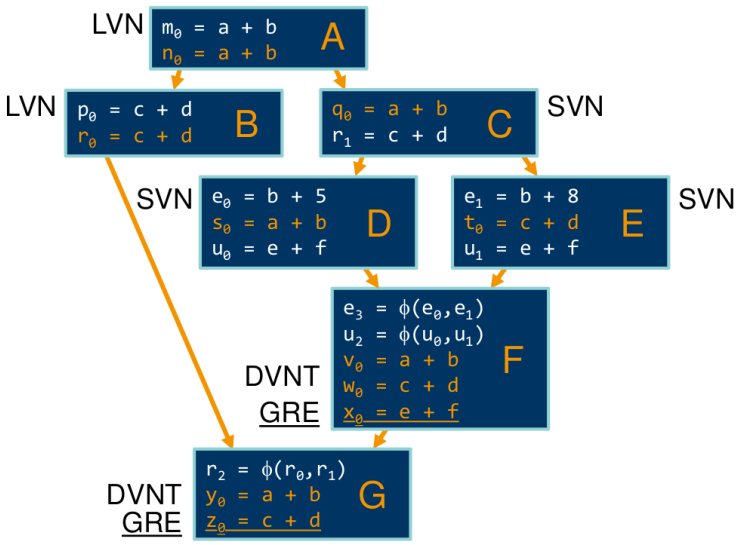


Figure 24: The result after applying GRE

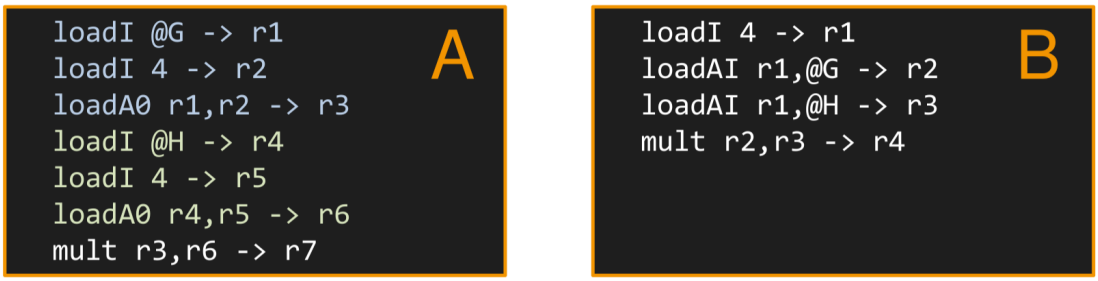


Figure 25: A comparison of naively and intelligently generated target code

8.1 Cost Based Code Generation

One way to generate target code based on the cost goes as follows: We add information on storage location (register or memory) to the AST, and then *tile* the tree using the combination of operators and storage locations (the operator information is already implicitly stored in the AST). Tiling in this case means combining nodes in the AST into target code, with all nodes in a tile being combined into a *single* statement. The tiling is done based on *ambiguous* rewrite rules, which are dependent on the target instruction set and architecture. The rewrite rules are ambiguous to allow for multiple tilings, based on cost. s the output of one tiling is the input of the other (if they are adjacent in the AST), they need to agree on storage locations. As an example, I included the entire example slides in Figure 26 and Figure 27. Figure 26 shows the AST and the rewrite rules, while Figure 27 shows the resulting tiling and target code. You can solve the question in Figure 27 yourself as an exercise. The main approach is

repeatedly applying the rewrite rules. This causes the code generation to turn into a term rewrite problem, for which other people have well thought out algorithms. The cost of matching rewrite rules to tiles can be reduced by using lookup tables.

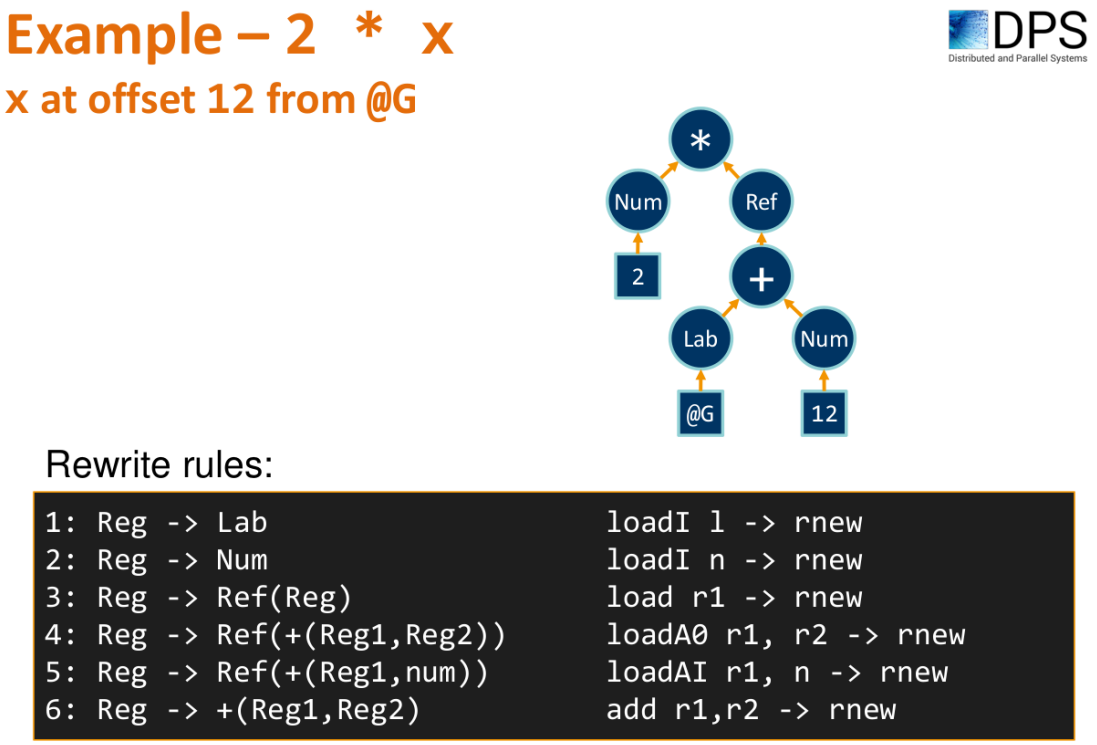


Figure 26: The AST and rewrite rules

Peephole Optimization

As target code generation (and optimization) complexity grows superlinear with the number of analyzed lines, this is often done using sliding windows (just like the view through a peephole). This technique works very well with linear IR and gives good practical performance. Its performance is dependent on the window size, with the optimal window size not really being computable (but that didn't bother anyone for Neural Nets either). It also sometimes has difficulty handling control flow, as the IR control structures may be further apart then the sliding window can see.

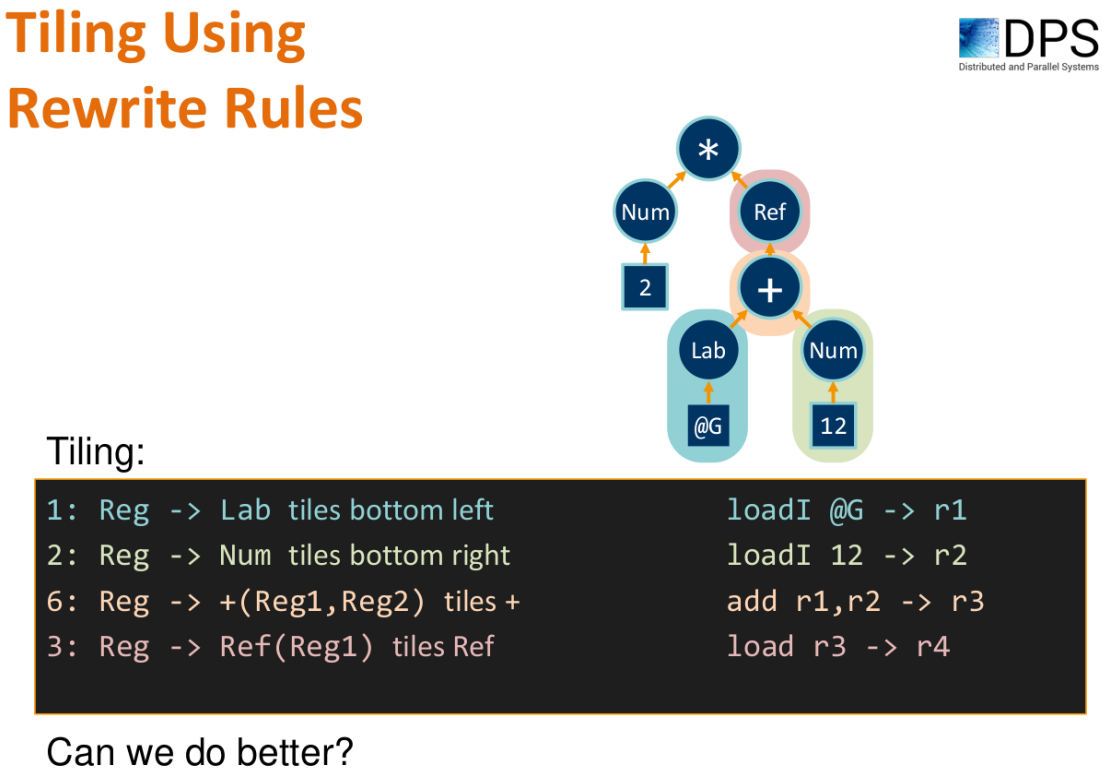


Figure 27: The resulting tiling

Superoptimization

Superoptimization is a technique that some theoretical computer scientists came up with. They decided to use theorem provers with a set of axioms for equivalent instructions to find and prove an optimal set of generated instructions. This is exponentially complex, as all combinations have to be searched (the search space can be reduced, but its still very large).

8.2 Special Code Generation Targets

In addition to utilizing ILP possibilities for RISC architectures, we sometimes are confronted with special architectures that require consideration in the compilation process.

Embedded Code Generation

For embedded architectures the code must be adapted to the individual platform to be efficient. The number of registers, RAM speed etc. are very variable for this kind of target. Here our goal is minimizing hardware-specific changes rquired in the compiler for new hardware, whil still generating high-quality code. This also applies to very

distinct and specialized hardware (e.g. heterogeneous instruction sets, coprocessors, ...).

Multimedia Code Generation

Standard Code Generation is completely blind to parallelism. Generating shorter code might severly restrict ILP in the architecture.

Multimedia instructions (instructions executed on a highly parallel device like a GPU) are often Single Instruction Multiple Data (SIMD). This needs special consideration and is often handled by special techniques or even special compilers. Automatic parallelization requires high-quality analysis, dependence information, knowledge of data structures and the details of individual hardware. With its complexity comes a huge benefit, making automatic parallelization a huge research topic in compilers.

9 Instruction Scheduling

With the increasing number of independent functional units (FUs) in modern processors, the order of instructions has a large influence on performance. We can hide the latency of instructions by scheduling other instructions in their shadow, thus keeping the pipeline filled.

Additionally, modern architectures can execute multiple instructions in parallel, if their operands are available. The two main flavors of modern ILP are Superscalar architectures and VLIW. In its essence, superscalar architectures have different FUs for processing parts of an instruction (load, ALU, store, ...). This allows different parts of multiple instructions being executed at the same time (what we call a pipeline), just like starting work on a new instruction every cycle. VLIW (very long instruction word) architectures have explicit insructions for processing multiple operations at the same times. Both ILP families are dependent on the scheduling of operations, with VLIW needing explicit combination of instructions into a VLIW.

The number of FUs and the number of registers dictates the way the instructions need to be scheduled for optimal ILP usage. For instruction scheduling we treat the number of registers as infinite, and deal with register allocation in a later step (this isn't optimal, but it's good enough and faster). Figure 28 shows the difference good instruction scheduling can make. In this example we have 1 FU, so we can run a new instruction each cycle iff the operands are ready. Our delays are: load/store 3 cycles, mult 2 cycles, others 1 cycle.

9.1 List Scheduling

To determine a good schedule for local instructions (inside a BB) we first build a dependence graph. For this graph we connect each instruction to the instructions whose results it depends on, with the edge weights being the instruction delays. Figure 29

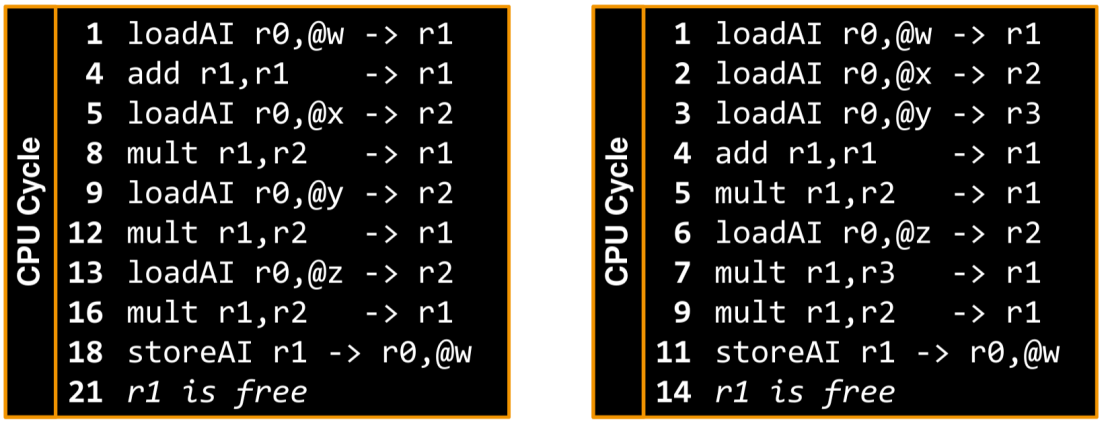


Figure 28: The difference in needed CPU cycles for different instruction orderings

shows an example with the corresponding code. We ignore anti dependencies (registers needing to be available) and just assume an unlimited number of registers.

Determining the best schedule from this graph is an NP complete problem, but we have a good heuristic approach. The algorithm for it is shown in Figure 30. We can define the priority function as we need, making this algorithm very versatile. (For those interested in efficient algorithms, this is a nice application of a heap).

When using the algorithm from Figure 30 (using the remaining critical path length as priority) on the example shown in Figure 29, we get the result shown in Figure 31

We can use a lot of different metrics as a priority for list scheduling, like longest latency, last use of value, number of descendants, ... We can also combine multiple metrics into a custom priority function.

Forward- and Backward-Scheduling

We can start scheduling from the leaves (as shown in Figure 30), or from the root. These different approaches might result in a difference in cycles or registers, but both should be pretty good, depending on the used metrics.

Superlocal Instruction Scheduling

It is possible to shift instructions between BBs. This can increase ILP usage, but requires compensation code. Moving instructions between BBs can be worth it, especially for control paths that are more often travelled than others. This is called *Trace Scheduling* and requires recompilation of the program after it has been run (more on stuff like this in slidedeck chapter 11, which is not covered by this summary).

```
a loadAI r0,@w -> r1
b add r1,r1    -> r1
c loadAI r0,@x -> r2
d mult r1,r2   -> r1
e loadAI r0,@y -> r2
f mult r1,r2   -> r1
g loadAI r0,@z -> r2
h mult r1,r2   -> r1
i storeAI r1   -> r0,@w
```

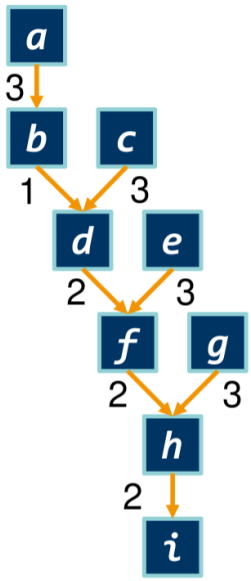


Figure 29: Example code and the dependencygraph for it

9.2 Loop Scheduling

As loops usually dominate the run time, optimizing their latency (and speed) is very crucial for program performance. Loop Unrolling can improve instruction scheduling, but increases register and cache pressure. For VLIW hardware we can also use a special technique called *Software Pipelining* to increase loop performance. Figure 32 shows the example code that will be used for this section.

In software pipelining the instructions inside the loop are arranged in such a way that multiple instructions *from different iterations* can be executed on the same cycle. This has the same effect as HW pipelining and branch prediction in superscalar architectures. For rearrangement we have to ensure the dependencies between the iterations are still fulfilled. Figure 33 shows inter-iteration dependencies for the example code.

To achieve this, the loop code is split into a prologue (executed once before the target code loop), the kernel (which is the target code loop) and an epilogue (executed once at the end), as shown in Figure 34. After this split, the target code is equivalent to the source code. We then apply *modulo scheduling* on the kernel, splitting the operations over the cycles of the longest instruction shadow. Figure 35 shows a more optimal scheduling for the loop in the example.

Now we only need to know how to apply modulo scheduling. To do this, we first need to calculate the *initiation interval* (ii), which is bounded by the number of function units and recurrence distance (longest delay of an operation with dependents). A smaller ii means a smaller loop body, and therefore a faster loop.

In the example we have 2 integer operations and 1 unit, giving us a minimal ii of 2.

```
cycle = 0
ready = leaves of dependence graph G
active = empty
while( ready ∪ active != empty )
   if available then take instruction i from ready
                          (based on priority)
   add instruction i to active
   for each instruction in active
      if completed remove from active
      for each successor of instruction
         if successors operand ready then
            add to ready
```

Figure 30: The list scheduling algorithm in pseudocode

Also, the recurrences on c have a delay of 1, giving us a minimal ii of 1. This means a combined ii of 2.

We then apply modulo scheduling using our ii of 2 as the modulo, scheduling instructions from the kernel to the corresponding units in cycle 0, then cycle 1. If we have additional instructions, we try to schedule them in cycle 0 again. If we're not able to (because we have run out of units, or because of dependencies) we increase the ii and try again. We repeat this until we can find a scheduling that works. After we found a scheduling for the kernel, we add the prologue and epilogue code, adding register copies if we need to. And then we're done, that wasn't too hard now, was it?

10 Register Allocation

Now that we have the target code, we only need to find a way to fit all operations into the registers we have. In order to reduce the number of registers we need to keep, we can use the liveness analysis covered earlier to find dead variables. The basic idea is, if a value is not needed reuse the register, and if there are no more available registers, spill memory. Spilling memory is very expensive, as accessing RAM is *so much more expensive* than using registers. Finding the best register allocation is an NP complete problem, so we use heuristics to reduce complexity. We also assume that code generation and instruction scheduling are both already finished.

```
1  a loadAI r0,@w -> r1
2  c loadAI r0,@x -> r2
3  e loadAI r0,@y -> r3
4  b add r1,r1    -> r1
5  d mult r1,r2   -> r1
6  g loadAI r0,@z -> r2
7  f mult r1,r3   -> r1
9  h mult r1,r2   -> r1
11 i storeAI r1 -> r0,@w
```

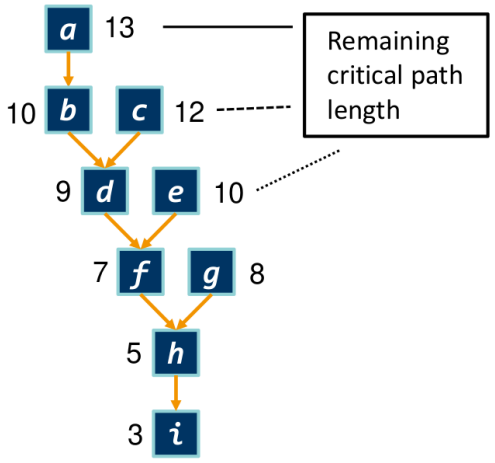


Figure 31: The result when using critical path length as priority

10.1 Local Allocation

We first focus on basic blocks and map virtual registers to physical registers. The *top-down* approach computes a priority of virtual registers with the most important ones being allocated first. The ones that don't fit are just spilled. This technique has poor performance as virtual registers blocka physical register for the entire scope.

The *bottom-up* approcah iterates over the instructions in a block, allocating on demand. It frees a register if the containing value is no longer needed (liveness), and uses the distance to next use as a spill metric.

> When spilling a value, it is better to spill a *clean* value (one that was not changed after loading), rather than a *dirty* one (one that was changed). This saves us a store operation, as we can just load the pristine value later on. However, it can be better to spill dirty values, based on usage. This is another NP-complete problem.

10.2 Global Register Allocation

As local allocation does not consider the reuse of values across multible BBs, it leads to suboptimal results. To handle register usage across BBs, we assign registers to *live ranges* instead of variables or values. A live range is the range of instructions between the definition and last use of a variable. We then perform live variable dataflow analysis to track live variables across blocks, and only allocate registers for live variables.

As this can be used just as well for local allocation, global register allocation makes no difference between local and global. An *interference graph* is constructed from live

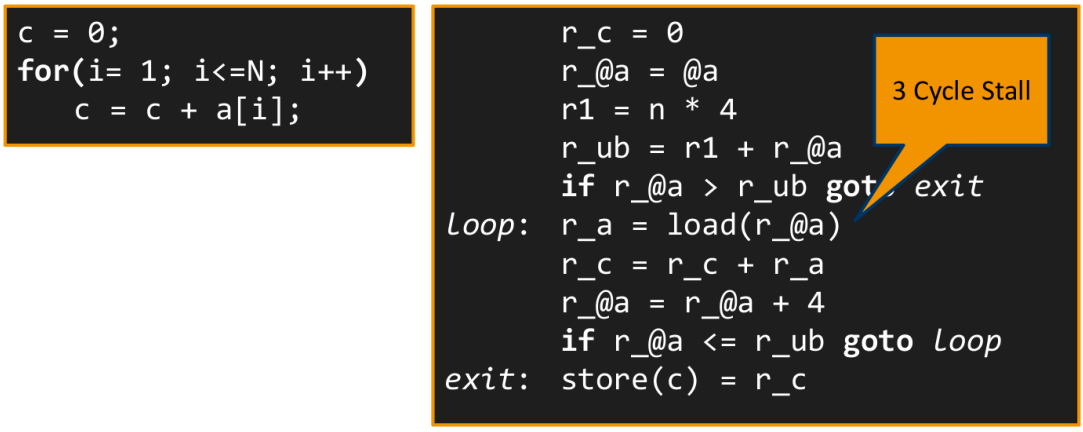


Figure 32: The example code for this section

variables, with neighbors being variables live at the same time (intersecting live ranges with different values). We then try to color this graph in k different colors (with k being the number of available registers), with no neighbors having the same color. If the graph needs more than k colors, we need to spill. After finishing, we can directly map the colors to physical registers.

As graph coloring is another NP-complete problem, we need heuristics to "solve" this efficiently.

Graph Coloring

We start with an important observation: *Any node n that has less than k neighbors $|n| < k$ can always be colored*

We then get a very simple algorithm:

1. Pick any node $|n| < k$ and put it on *The Stack*
2. Remove that node and its edges, reducing the degree of neighbors
3. If there are no nodes $|n| < k$, spill any node and continue
4. After all nodes are on *The Stack*, pop them and color them one by one (considering neighbor's colors)

Figure 36 shows an example of a colored interference graph. Note that there are cases where this algorithm does not find the best coloring (e.g. a diamond shaped interference graph is 2 colorable, but this algorithm does not find that solution). It is better than trying all combinations however.

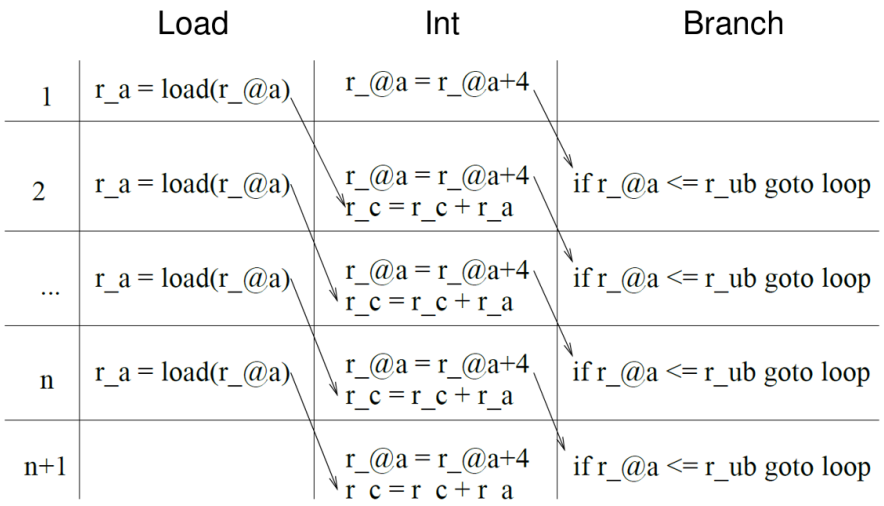


Figure 33: Dependencies between iterations for the example

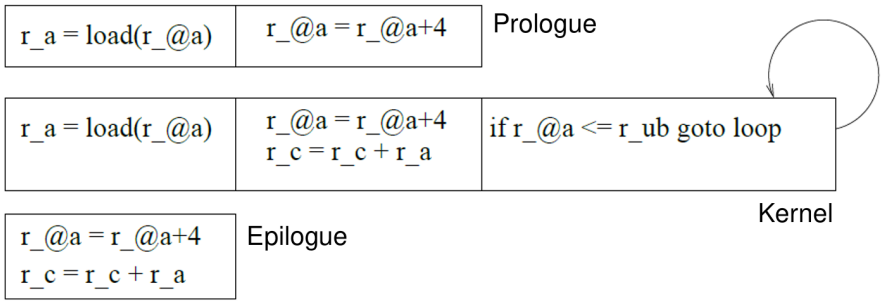


Figure 34: The loop code split into prologue, kernel, and epilogue

Spill Candidates

When spilling values we want to minimize the spill cost. The spill cost is the number of loads and stores needed for the spill. Also, the higher the degree of the node in the inteference graph, the more likely its spill is to benefit us.

In cases where there is a load and store being made to the same location without any operations depending on it (the value not being used), the spill cost is negative. Spilling that value actually saves a load.

For cases where spilling does not decrease the live range of a variable, the spill cost is infinite.

	Load Unit	Integer Unit	Branch Unit
	nop	r_@a = @a	nop
	nop	r1 = n * 4	nop
	nop	r_ub = r1 + r_@a	nop
	r_a= load(r_@a)	rc = 0	nop
	nop	r_@a = r_@a + 4	if r_@a >r_ub goto exit
Loop:	r_a =load(r_@a)	r_@a = r_@a + 4	if r_@a >r_ub goto exit
	nop	r_c = r_c + r_a	nop
exit	nop	nop	nop
	nop	r_c = r_c +ra	nop

Figure 35: A more optimal scheduling for the example code

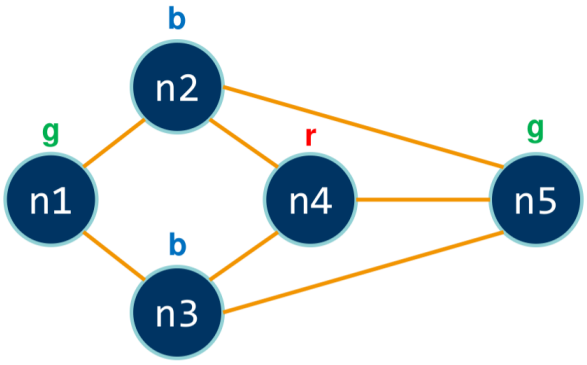


Figure 36: A colored interference graph

Alternatives to Spilling

Instead of spilling entire live ranges, we can also just spill the variable in areas of high register demand, called partial live ranges.

We can also split live ranges with artificially induced stores and loads, splitting nodes in the interference graph and thus reducing their degree. If the splitting is smart, the spill can occur in "cheap" regions, leading to less stalling and a potentially higher usage of the pipeline.

Finally, we can coalesce registers if they don't interfere, and are connected by a copy. This just means that instead of two nodes in the interference graph connected by a copy we now have one.

10.3 Coalescing

This is easiest shown on an example. In Figure 37 Ra, Rb and Rc actually have the same value. Ra and Rb are combined into a single register, and we could do the same thing with Rc.

Figure 37: An example of register coalescing

Coalescing *generally* reduces the degree of the interference nodes, and is therefore often applied before coloring the interference graph. However, sometimes coalescing can *increase* the degree of nodes, but this problem can be handled by checking if it actually helps before applying coalescing (called *conservative coalescing*). This leads to iterative processing, which is just conservative coalescing and coloring applied in turn.

Alright, that was it. You're probably as glad to be done as I was when writing this. Now all that's left to say is good luck in the exam, I hope this summary helped.