



# Genetic Programming to Generate Better Compilers

Brad Alexander and Michael Gratton

# Outline

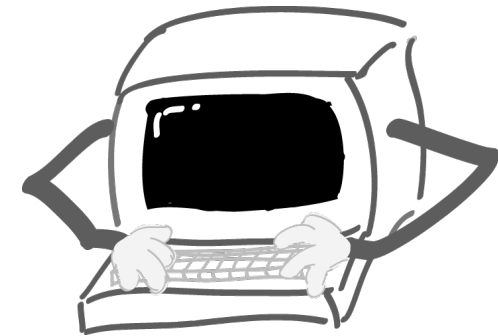
- Background and Context
- Current Approaches
- Experimental Aim
- Design Choices
- Experimental Setup
- Experimental Results
- Conclusions/Future Work

Alexander/Gratton



# Background

- This talk is about making computers write programs for themselves.
  - Using Genetic Programming (GP)
- This is not new
  - GP is nearly 20 years old.
- The new part is what we do with GP
  - We get the computer to program a tricky part of a compiler.
  - This task is usually challenging for humans.
- What is a compiler?



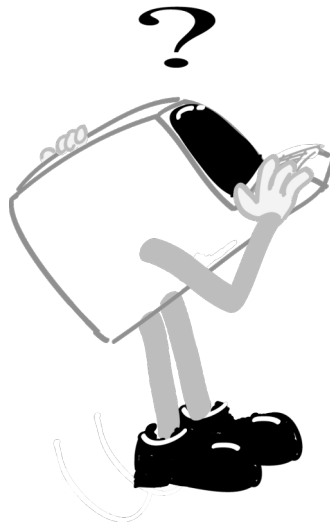
# Background: Programs

- Computing is the art of creating software to do new things.
- Software is generally expressed as a program e.g.

```
#include <stdio.h>
int main(){
    printf("Hello World");
}
```

# Background: Machine code

- Unfortunately computers can't directly understand our programs.
- They only understand machine code.
  - Sequences of instructions expressed as ones and zeros.

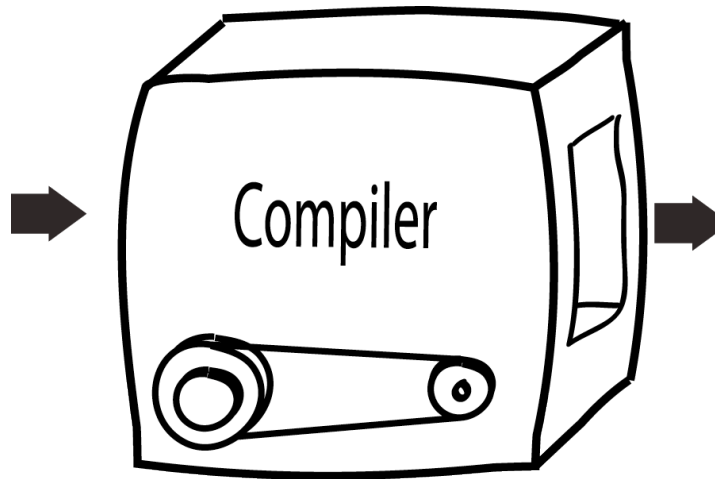


```
#include <stdio.h>
int main(){
    printf("Hello World");
}
```

# Background: Compilers

- Compilers are programs that translate our programs into machine code that a computer can understand.

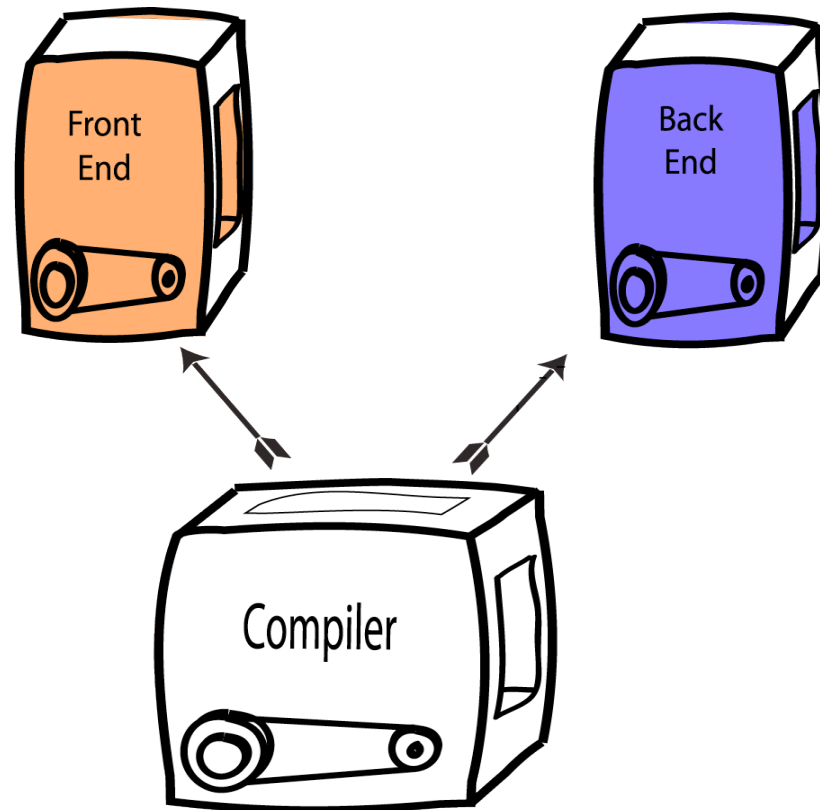
```
#include <stdio.h>
int main(){
    printf("Hello World");
}
```



```
010110100101
111000110101
010101010110
010100101010
```

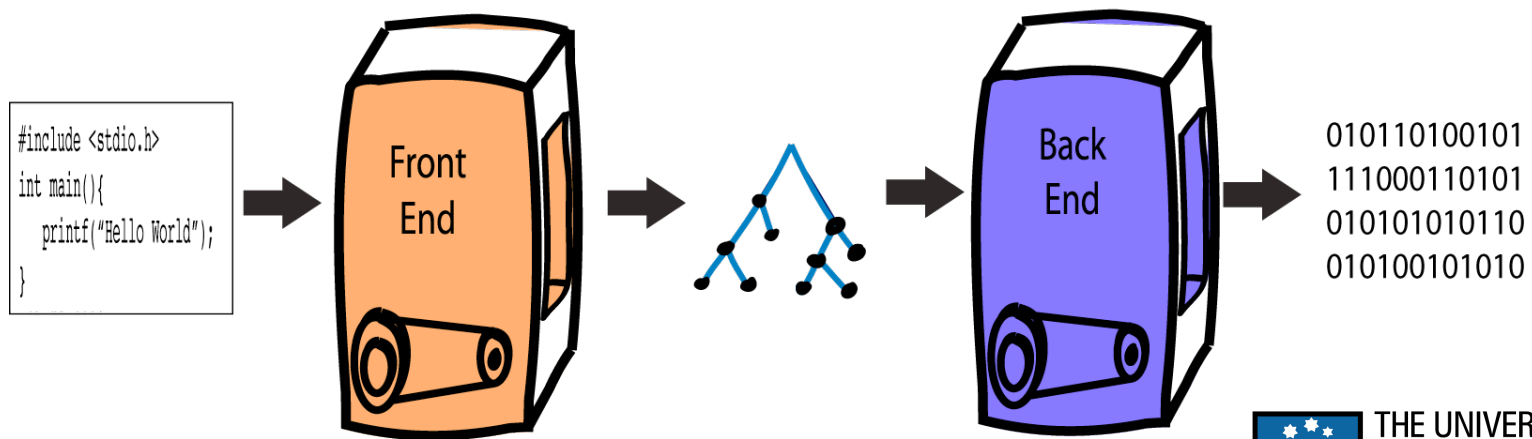
# Anatomy of a compiler

- A basic compiler contains two main parts.
  - A Front End and a Back End



# Anatomy of a Compiler

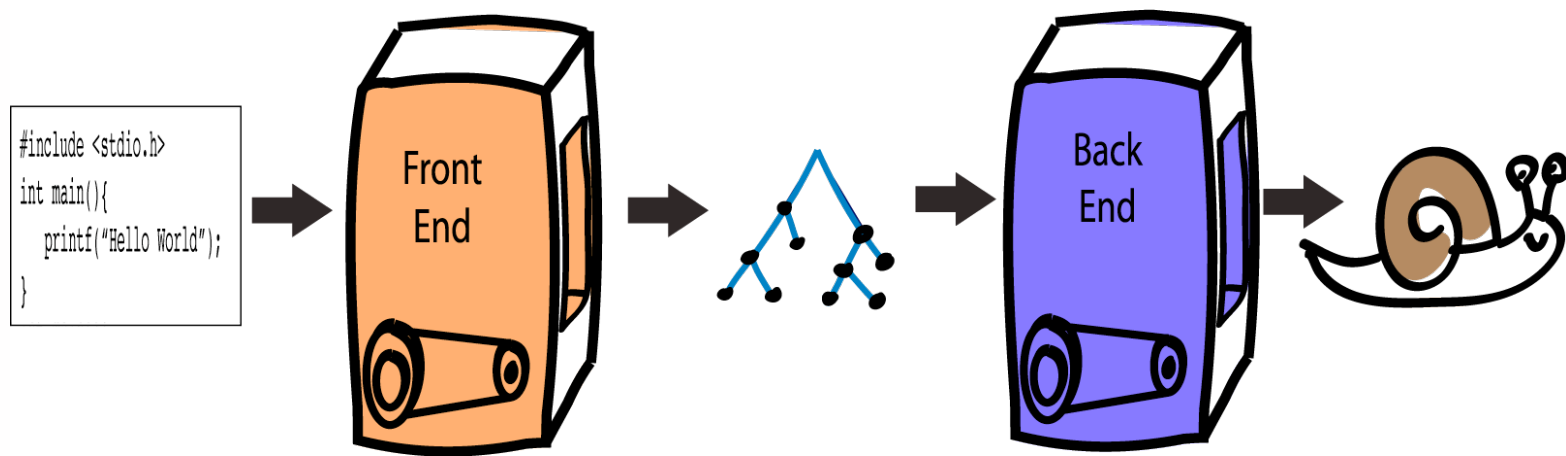
- The front end takes program code and converts it to an intermediate code.
- The back end takes intermediate code and converts it to machine code.





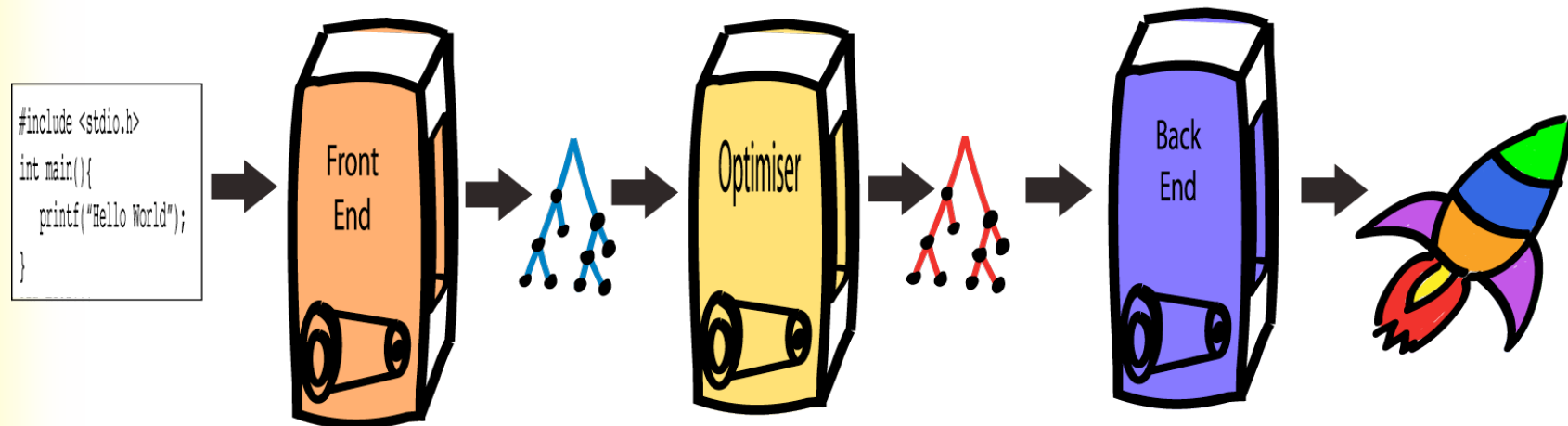
# Anatomy of a Compiler

- Unfortunately, for certain applications compilers consisting only of a front and back end will produce slow code.



# Background: The Optimiser

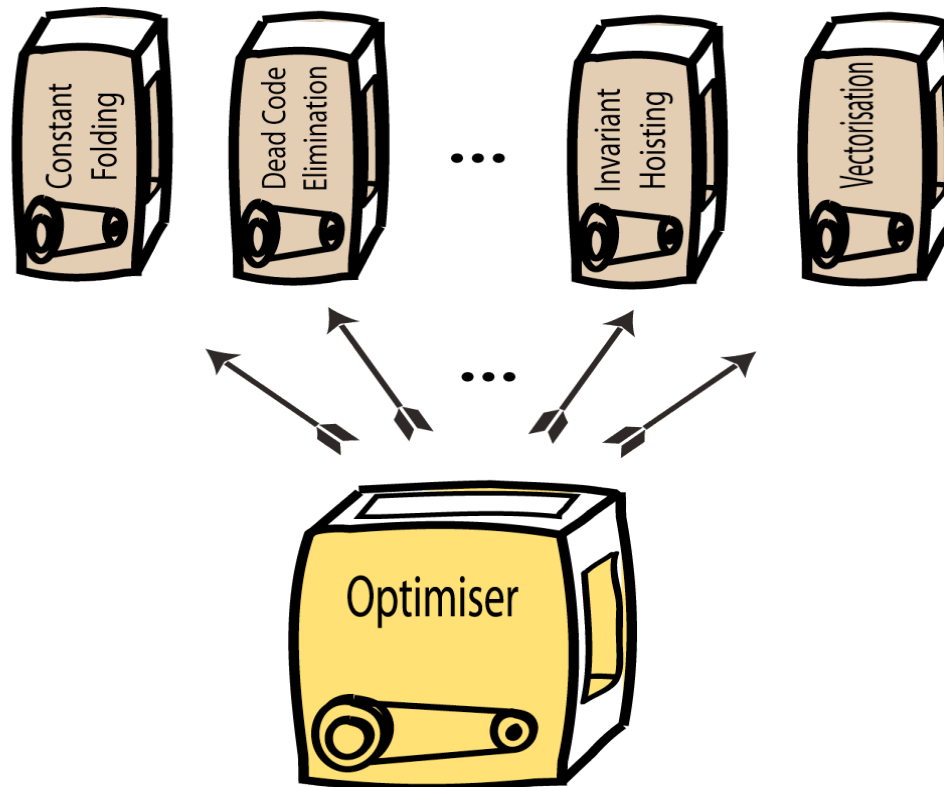
- An Optimiser can be used to transform intermediate code to make it more efficient:



Alexander/Gratton

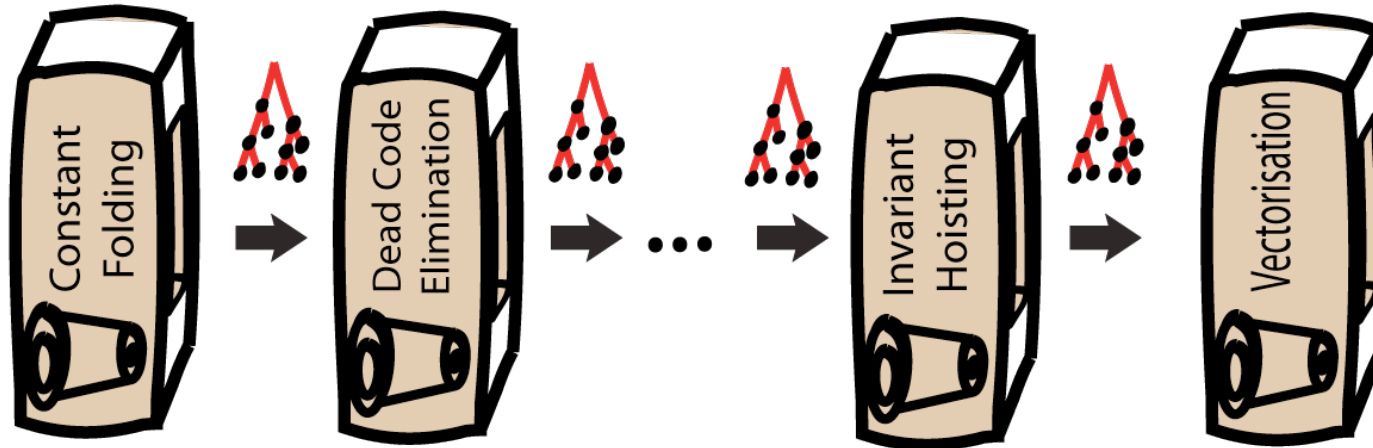
# Background: Optimiser Internals

- Optimisers are not monolithic
  - Instead, they often consist of 20 or more self contained optimisation phases.



## Background: Optimiser Internals(2)

- Intermediate code is pushed through these phases one after the other.

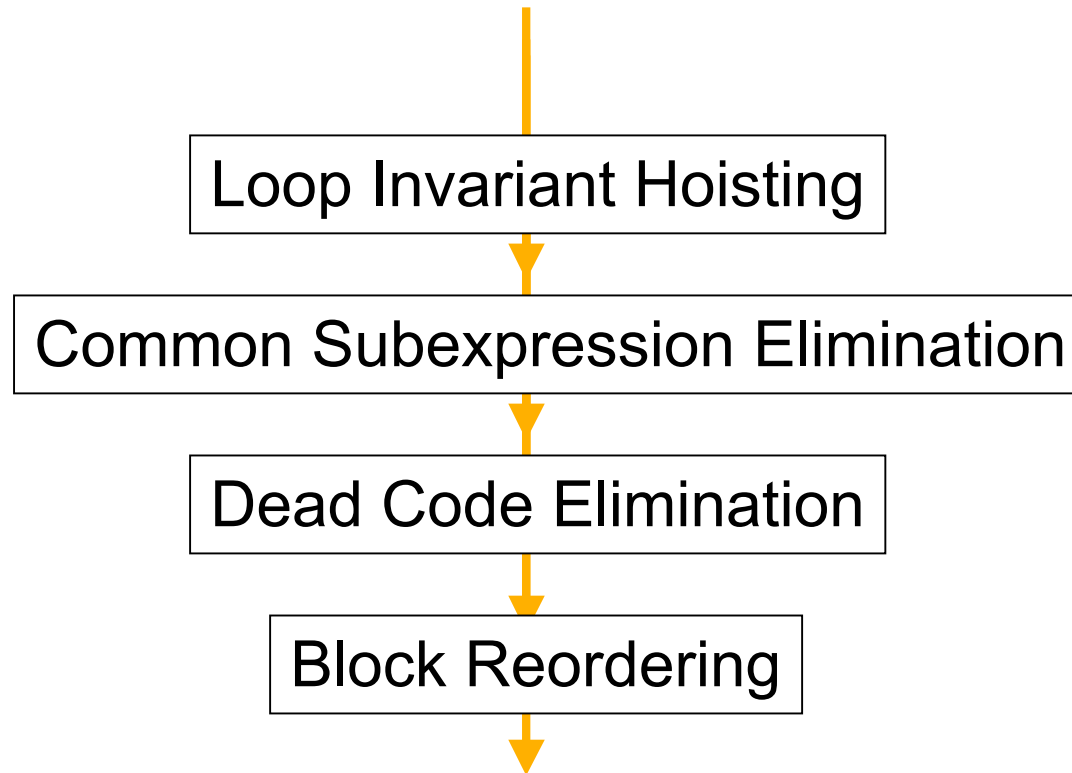


# Optimising the Optimiser

- How do we know that our optimiser's sequence of phases is the best for our applications?
  - We don't...
- So why not automatically adapt the optimiser to the set of programs we use?
- Problem:
  - The design space is huge and chaotic.
  - however, can search this space using heuristic methods.

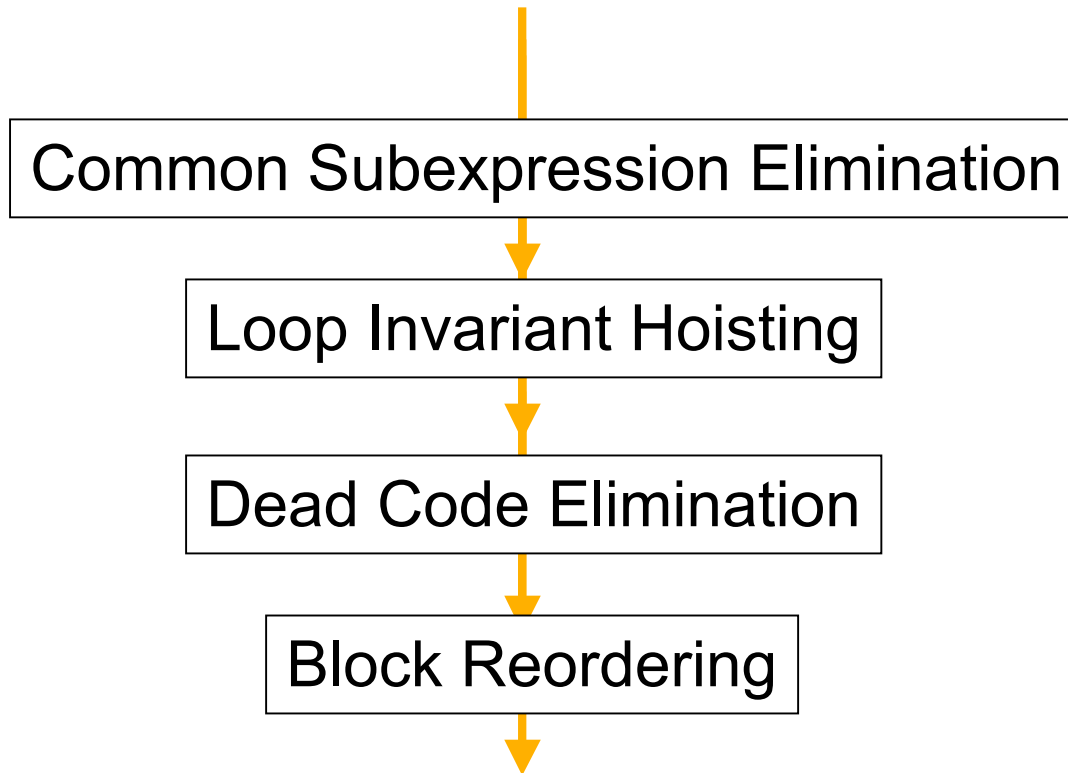
# Current Approaches

- Phase sequencing



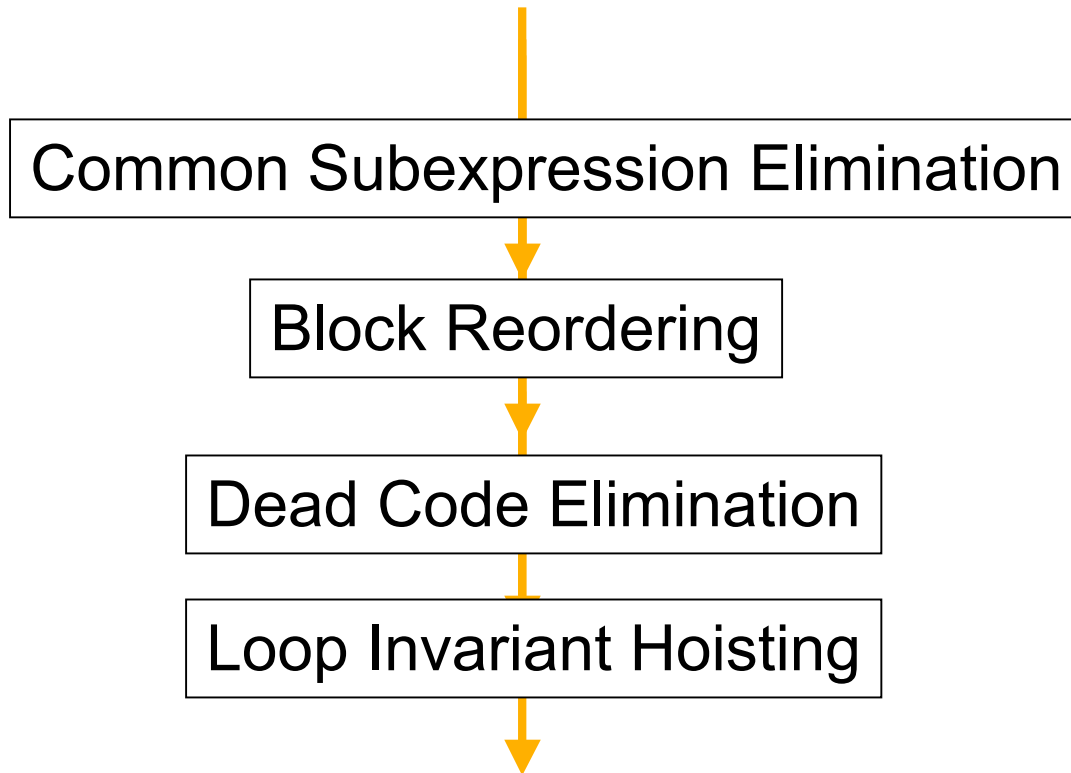
# Current Approaches

- Phase sequencing



# Current Approaches

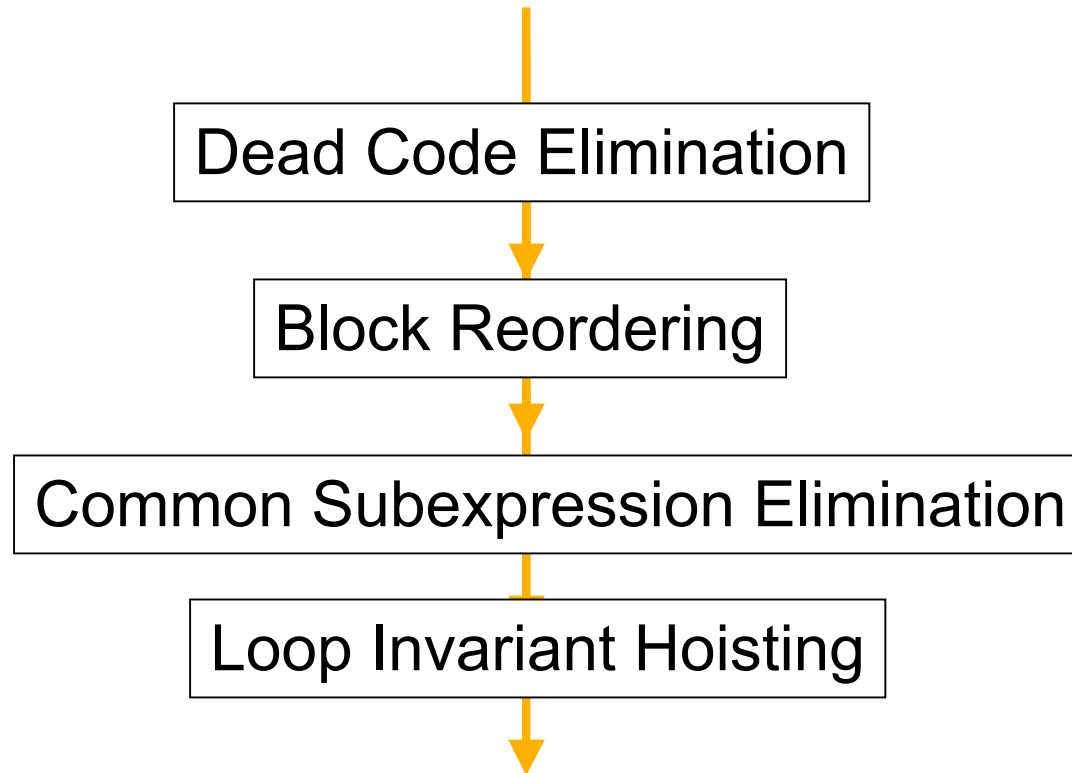
- Phase sequencing





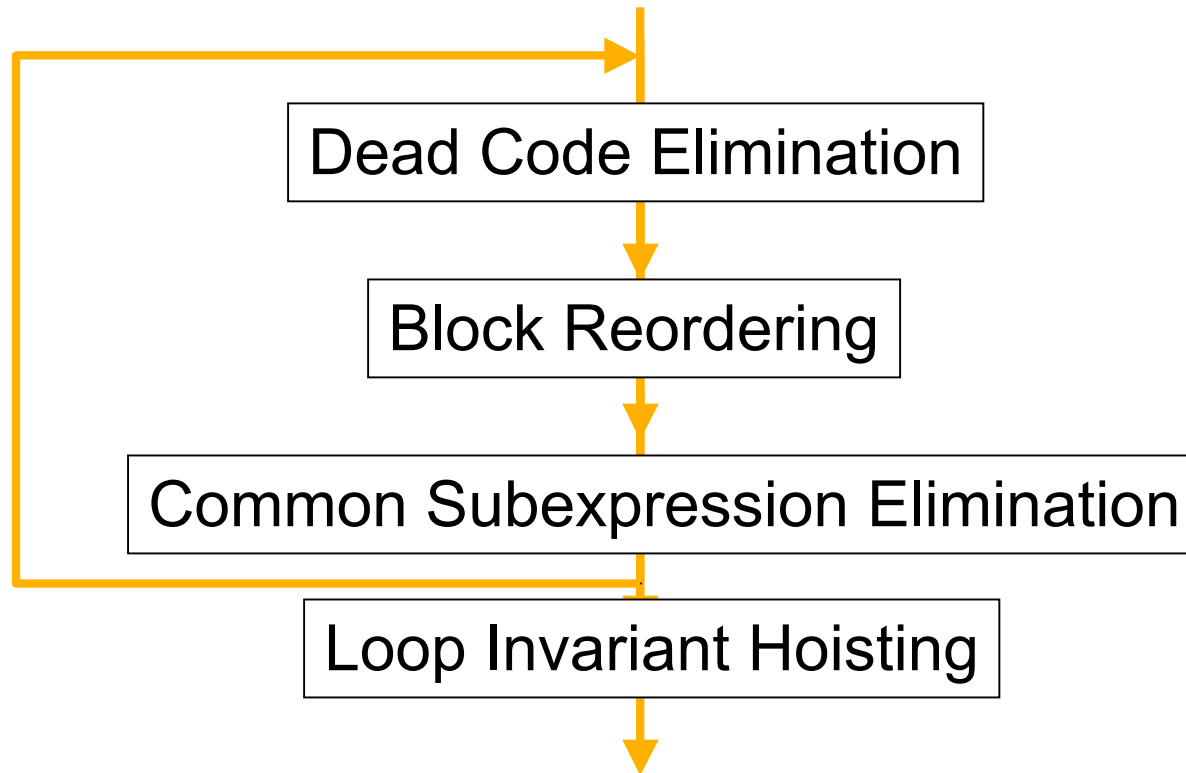
# Current Approaches

- Phase sequencing



# Current Approaches

- Phase sequencing



# Current Approaches

- Parameter Tuning

loop tiling/unrolling phase

Loop unroll factor:

3

Loop tiling factor:

2

# Current Approaches

- Parameter Tuning

loop tiling/unrolling phase

Loop unroll factor:

4

Loop tiling factor:

3

# Current Approaches

- Evolution of Control Code

Register Allocation



# Current Approaches

- Evolution of Control Code

```
if( reg_size > &  
    spill_cost ...)
```

Register Allocation

# Current Approaches

- Evolution of Control Code

Register Allocation

```
if( reg_size > &  
    spill_cost ...)
```

# The Story So Far

- Thus far we have shown the following:
  - Programs need compilers.
  - Compilers are made of **front-ends**, **back-ends** and **optimisers**.
  - Optimisers help compilers produce fast machine-code.
  - Optimisers have many stages.
  - There have been successful experiments in using computers to automatically:
    - Reorder optimisation phases.
    - Control optimisation phases using parameters



# What's Missing?

- All current work assumes that optimisation phases are pre-existing and atomic or parametric.
- Currently no work on the construction of these phases from smaller building blocks.
- Our Research Question:
  - Can we use Genetic Programming (GP) to build a non trivial optimisation phase?
- From our experiments the answer is a clear yes!
  - But as in all GP exercises, careful design is required.

# Experimental Design Choices

- In designing an experiment we need to make the following design choices:
  - Application Domain
  - Intermediate Language for candidate optimisers to transform.
  - Raw ingredients to build the candidate optimisation phases out of.
  - Choosing a GP Framework
  - Choosing an Evaluation Function.
- I cover each of these in turn.

# Choose an Application Domain

- The optimisation phase we used is part of a compiler mapping a simple language, Adl, to programmable hardware.
- The optimisation phase we want to build is responsible for reducing the amount of data flowing through intermediate code.
  - Less data == Less wires!!
- This is a parallel high-performance application so we have more to gain from optimisation than we would in most application domains.
  - But optimisers for parallel programs harder to build.

# Choose an Intermediate Language

- The language used to express intermediate code is important.
  - We chose Point-Free-Form (functional programming without variables)
- Three advantages:
  1. Point-Free-Form is easy to transform -
    - makes it easier for our compiler to make progress.
  2. Point-Free-Form has explicit flows of data between operations.
    - We can see what we're optimising.
  3. Point-Free-Form is naturally parallel.
    - Easy to map to a parallel machine.

# Choose Ingredients

- As much as we'd like, we can't just say to our computer:  
    “Build Me an Optimisation Phase!”
  - GP wouldn't know where to start!
- We need to provide a carefully selected set of ingredients.
- For this application our ingredients are:
  1. Small sets of rewrite rules for changing point-free code.
  2. Strategies for applying these rules to different parts of point-free programs.
- We borrowed these ingredients from a hand-written optimiser.
- Both are expressed in Stratego
  - An amazing language for writing optimisers and other program transformers using rewriting.

# Ingredients: Choosing Rewriting

There is no great writing, only great rewriting.  
(Louis Brandeis)

- Rewriting is the transformation of system through a series of small local changes.
- Rewriting is a great basis for program transformation.
  - If your rewrite rules are correct then you cannot use them to produce an incorrect program.
  - This gives you a lot of freedom to experiment with how you apply these rules without worrying about breaking the user's program.
- In our experiments we keep the rewriting rules fixed
  - The GP algorithm experiments with how these rules are applied.
- The how is important - rewriting systems can be hard to control..

# An Aside: The Importance of Rewriting

- The most beautiful and important things we know are rewriting systems or are products of rewriting systems:
  - Mathematics
  - Fractals
  - Nuclear Physics
  - Chemistry
  - Life....
- Rewriting systems are often chaotic and hard to control but..
- Rewriting works!



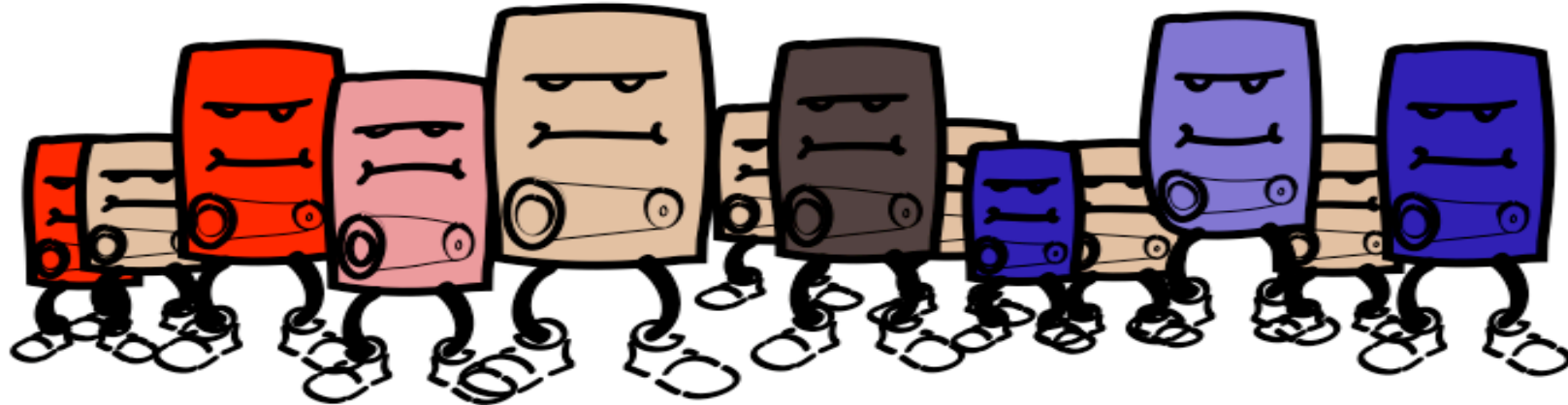
# Choosing a GP Framework

- The genetic programming framework is responsible for:
  - Generating an initial population of candidate optimisers (individuals).
  - and then, over many generations:
    - Applying the evaluation function to each new individual.
    - Selecting individuals that did well enough.
    - Applying genetic operators to some surviving individuals
  - Deciding when to stop.
- We chose Grammatical Evolution, using LibGE as a framework because it generates, mostly, viable individuals, which makes it work faster.



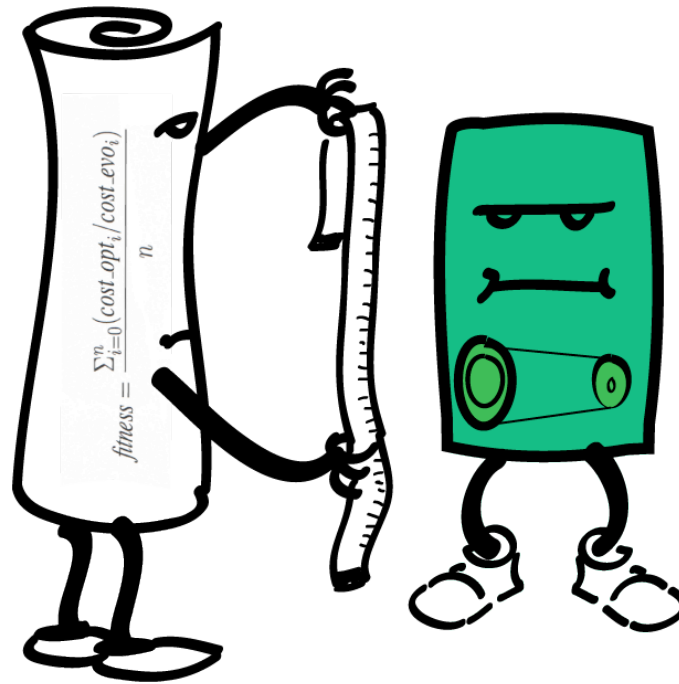
# GP Framework (in pictures)

- Generate an initial population:
  - We used between 100 and 300 individuals.
  - These individuals are not very good to start with
  - But our population will get better over many generations.



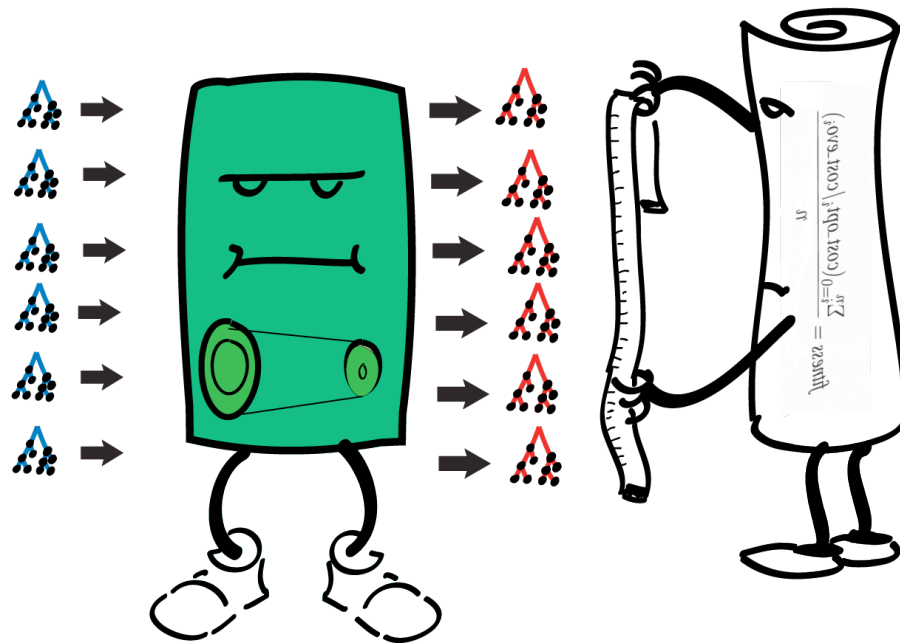
# GP Framework: Evaluation(1)

- Applying an evaluation function to measure the fitness of individuals in each generation.
  - The fitter ones are given a better chance of surviving.



## GP Framework: Evaluation(2)

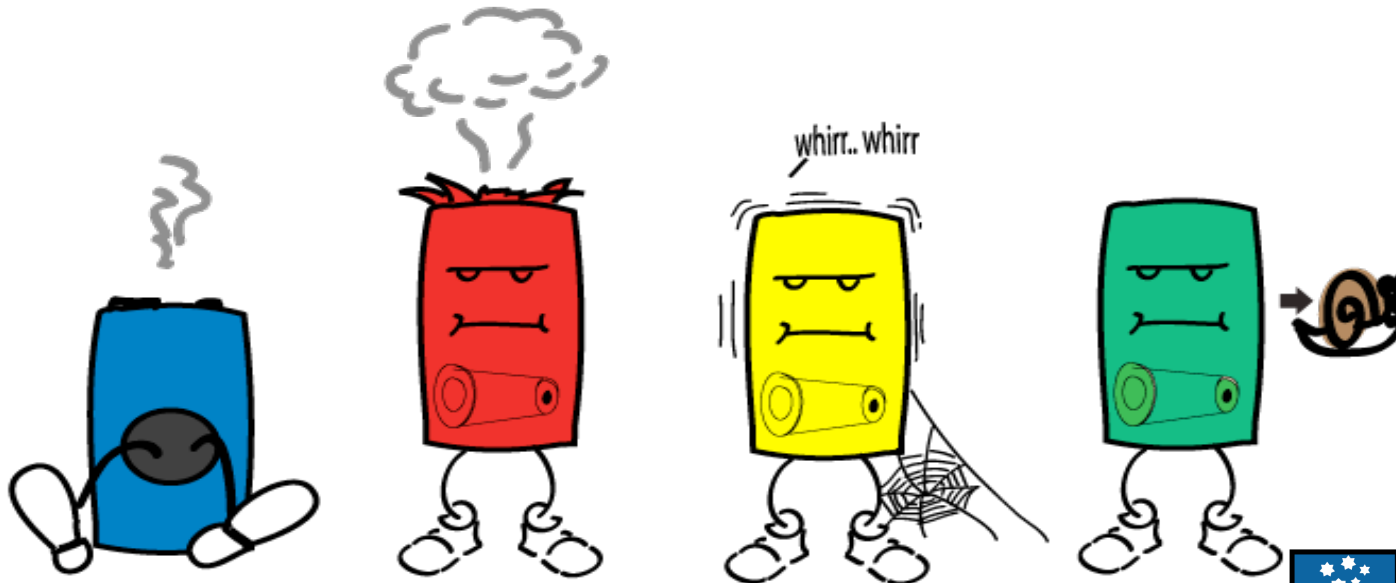
- We feed it a small number of benchmark intermediate codes and measure the average performance of these.
  - We used the results from the hand-coded optimiser as a basis for comparison.



Alexander/Gratton

# GP Framework: Evaluation(3)

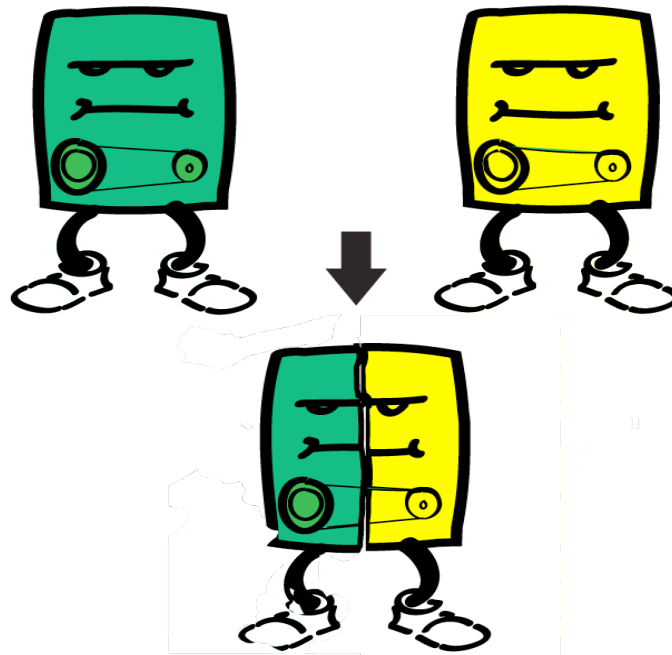
- Special treatment is needed when individuals fail.
  - Four failure modes:
    - DOA (fail to compile individual), Optimiser blows stack, Optimiser takes too long, A benchmark takes too long.
    - These are all detected and minimum fitness is assigned.



Alexander/Gratton

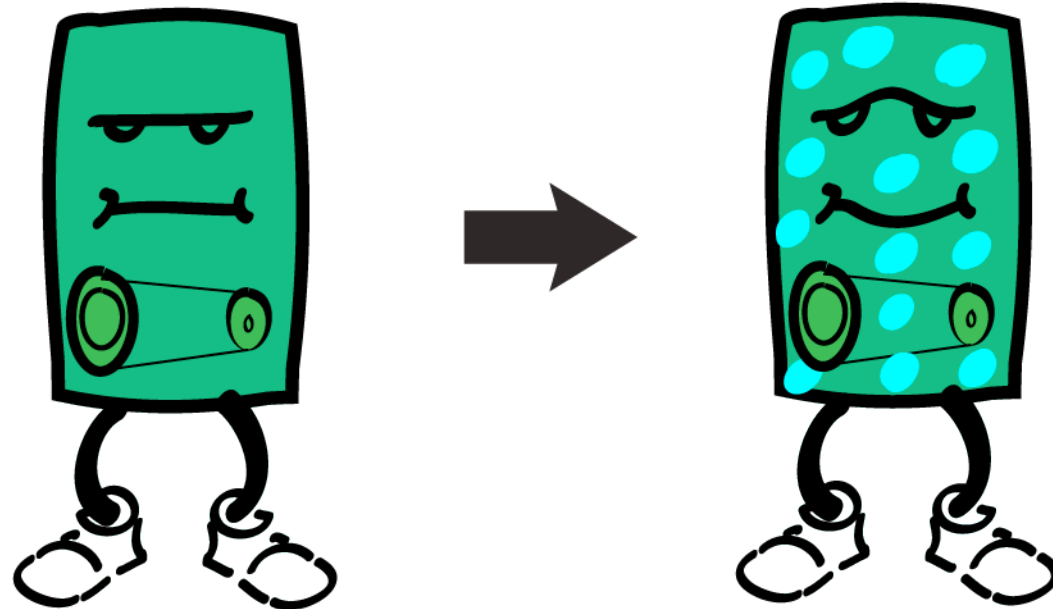
# GP Framework: Genetic Operators: Crossover

- Fitter individuals are randomly selected for breeding using crossover.
  - Crossover mixes the genes of individuals in the hope that good traits will combine.



# GP Framework: Genetic Operators: Mutation

- Randomly selected individuals will be mutated in each generation.
  - Not all mutations are beneficial!!



# GP Framework: Deciding When to Stop

- The framework stops after a certain amount of time or when a certain number of generations have been reached.
  - We stopped at times between 20 and 60 hours of runtime on a single 2.5GHz Intel processor (50 to 80 generations).
  - After stopping the fittest individuals can be selected for use.



Alexander/Gratton

# Experimental Setup

- All grammar elements pre-compiled into stratego libraries for faster running.
- Several runs conducted to tune fitness function.
- Final two runs:
  - Population approximately 250 individuals
  - Run for 80 generations and 63 generations respectively.
  - LibGE settings: Max tree depth 15. Read of genome can wrap-around twice.
  - Mostly default LibGA settings (for GE): Roulette wheel selection, 90% probability of crossover, 1% mutation probability, 1% replacement ratio and elitism switched on.



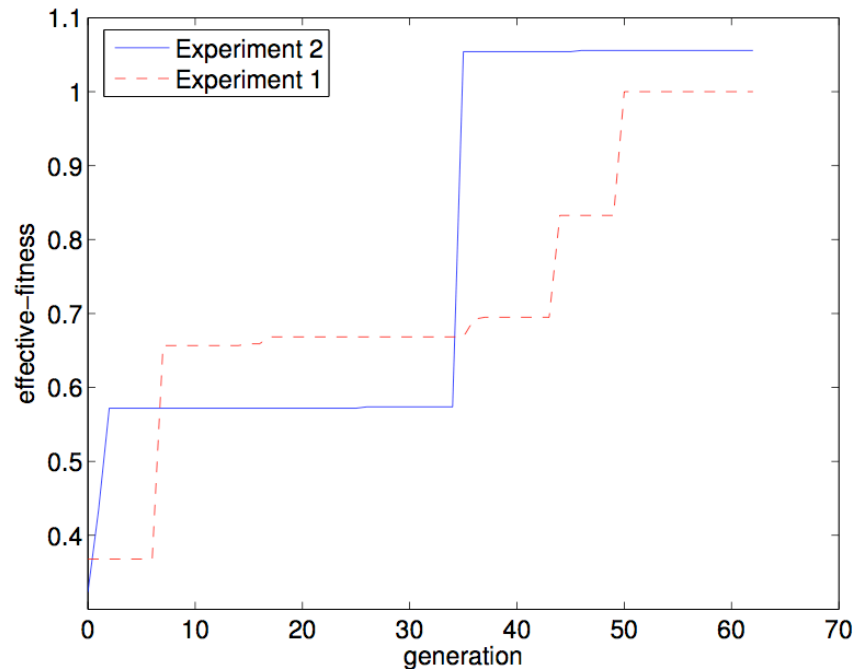
# Experimental Setup (2)

- Hand Coded Benchmark:

```
repeatUntilCycle(  
  bottomup(  
    repeatUntilCycle(  
      innermost(LeftAssociate)  
      ;innermost(pushDownComp)  
      ;innermost(LeftAssociate)  
      ;innermost(simp)  
      ;innermost(LeftAssociate)  
      ;innermost(pushDownMap)  
      ;innermost(LeftAssociate)  
      ;innermost(simp)))  
    bottomup(  
      repeatUntilCycle(  
        innermost(LeftAssociate)  
        ;innermost(pushDownAlltup)  
        ;innermost(LeftAssociate)  
        ;innermost(alltupSimp)  
        ;innermost(LeftAssociate)  
        ;innermost(convertAndRemoveIds))))
```

# Experimental Results (1)

- Both runs evolved individuals at least as good as the handwritten DMO's on the benchmarks.



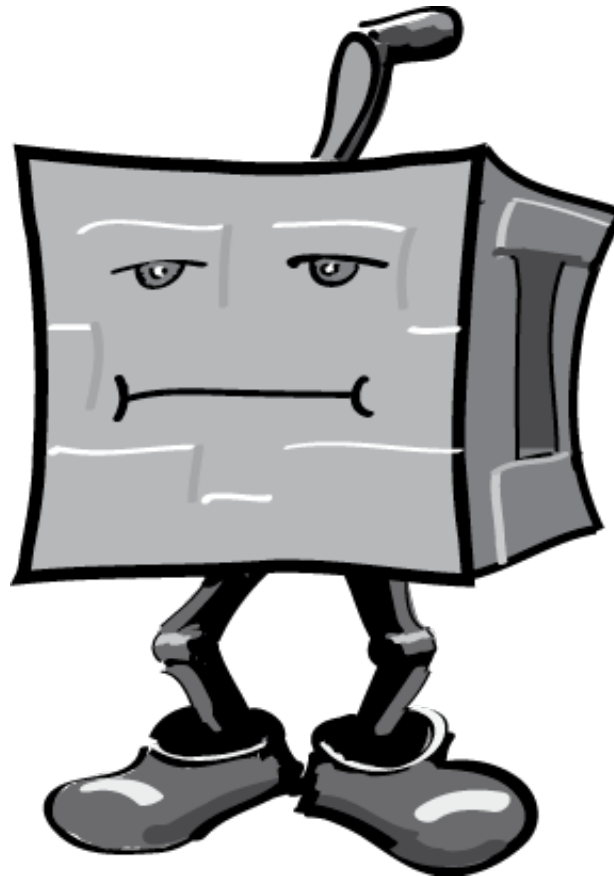
# Experimental Results (2)

- Robustness
  - Take the fittest individuals and expose them to thirty benchmarks and measure their performances.
  - Most did not generalise well but the fittest did slightly better than hand coded optimiser.
- Benchmark Choice
  - Need at least one that makes even mediocre individuals look good.
- Correctness
  - 500 fittest individuals collected and tested.
  - None produced semantic errors.
- Code Size
  - Best individuals very large with much redundancy.

# Conclusions and Future Work

- **Evolving a non-trivial optimisation phase is feasible**
  - Good results for effectiveness, robustness and correctness.
- **Future work includes:**
  - Pushing evolutionary process down to individual rules
  - Controlling code-size and efficiency.
  - Extending work to rewriting systems in other languages.

# Questions?



Alexander/Gratton

# Experimental Application

- Evolution of a phase of a compiler mapping a functional language (Adl) to a hardware definition language (Bluespec).
- The target phase is the Data Movement Optimiser (DMO) that reduces data flowing through a functional intermediate form (point-free code).
- There is an extant hand-written DMO that:
  - was non-trivial to construct.
  - can be used as a source of building blocks.
  - can be used as a benchmark
- The DMO is written in Stratego, a term-rewriting language consisting of rewrite **rules** and **strategies** for their application.

# Ingredients

- Three ingredients in any GP exercise:
  1. The grammar for building individuals consisting of:
    - terminals
    - non terminals
  2. The evolutionary framework.
  3. The evaluation function
- We look at these in turn.

# The Language Grammar (1)

- All individuals are expressed in Stratego
- **Terminals**
  - Consist of simple rewrite rules e.g.
    - CompIntoMap:  $f^* \circ g^* \rightarrow (f \circ g)^*$
    - MapIntoComp:  $(f \circ g)^* \rightarrow f^* \circ g^*$
    - RemoveId:  $\text{id} \circ f \rightarrow f$
  - grouped together using the left choice (<+) operator e.g.
    - CompIntoMap <+ RemoveId
  - **Semantics**: try applying CompIntoMap to current node and, if that fails, try applying RemoveId.
- We use the same terminals as the handwritten DMO



## The Language Grammar (2)

- Actual terminals include:

<code>pushDownMap</code>	(vectorise)
<code>pushDownComp</code>	(fuse loops)
<code>simp</code>	(apply simplifying rules)
<code>leftAssociate</code>	(left associate binary composition)

- In most contexts, the order of rules within a group is of minor consequence
  - If they can be applied they eventually will be applied.
- These terminals have little impact without strategies to apply them.

## The Language Grammar (3)

- **Non-terminals are strategies for rule application.**
  - These take strategies or rule-groups as parameters and apply the them to the target AST in some order.
- **Examples include:**
  - bottomup(s)** : apply s to the current sub-tree bottomup
  - innermost(s)** : apply s to the current sub-tree bottomup until it can no longer be applied (fixpoint strategy)
  - s ; t** : apply s to current sub-tree followed by t
  - repeatUntilCycle(s)** : apply s to the current sub-tree until a result seen before in this invocation is detected.
- **Example:**
  - bottomup(leftAssociate;innermost(simp))**

# The Evolutionary Framework

- We used LibGE in our experiments.
  - A popular framework for developing GE applications.
- LibGE (based on LibGA) takes:
  - A grammar definition and,
  - A evaluation function
  - Some parameter settingsand handles:
  - Population initialisation, application of the evaluation function to individuals, application of genetic operators, collection of statistics and, genotype to phenotype mapping.
- The mapping works by using 8-bit numbers in the genotype string to select productions in the language grammar.

# Evaluation Function(1)

- Fitness is calculated by running evolved optimisers against up to six benchmark programs and their data against a dynamic cost-model.
  - Benchmarks needed to be carefully chosen to require multiple strategies and have a gradual gradient of difficulty.
- Fitness calculated relative to cost of hand-coded DMO on each benchmark  $i$  ( $cost_{opt_i}$ ):

$$fitness = \frac{\sum_{i=0}^n (cost_{opt_i} / cost_{evo_i})}{n}$$

- Average fitness evaluation takes 5 seconds. Zero fitness for timeout or stack-overflow error.