Compilers:
Machine-Independent Optimisation

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

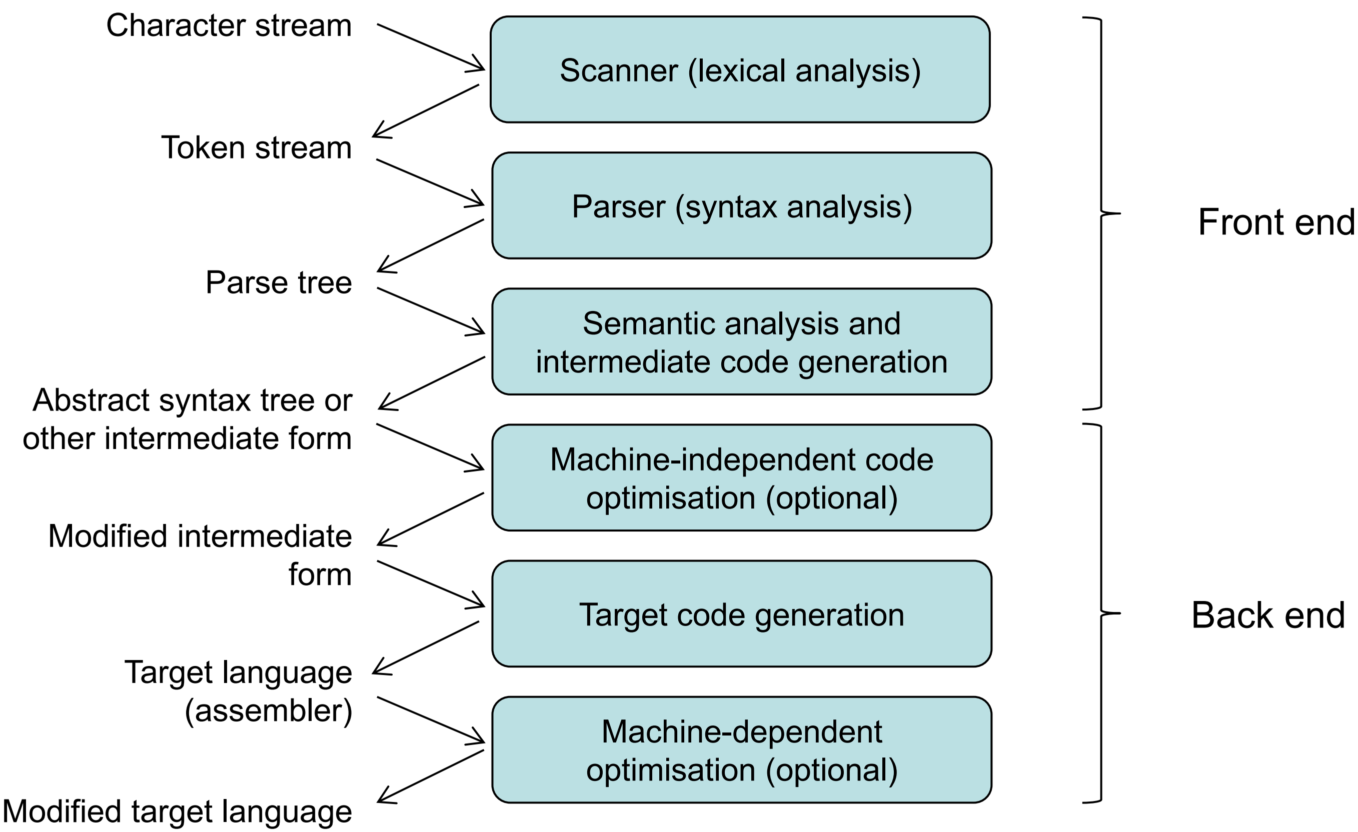
Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- **Machine-independent optimisation**
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

Objectives

- Define optimisation
- Discuss need for optimisation
- Define machine independent optimisation
- Discuss low-level intermediate representations
- Describe five types of scalar optimisation
- Give examples of transformations to optimise code

Phases of Compilation



Optimisation

- Optimisation is the process of transforming a program to make it run more efficiently
- There are different measures upon which one can optimise:
  - running time
  - processor usage
  - memory usage

Why Optimise?

- Program goes through several transformations already in the compilation process
  - These might introduce inefficiencies into even a highly efficient algorithm
  - Low level features that the programmer is not aware
  - Might not have been an efficient program to begin with

Machine Independent Optimisation

- Identify inefficiencies in the intermediate representation and transform to a more efficient form
- Ignores any features of the hardware upon which the code will be run
  - These are considered during a later phase of compilation

Intermediate Representations (again!)

- IRs thus far have preserved the structure of the high-level language
- Syntax trees more common for syntax/semantic analysis
- Useful to introduce a lower level representation, closer to machine code
  - Simplifies final transition to machine code
  - Facilities identification of efficiencies and implementation of transformations

Three Address Code

- *Three Address Code* (TAC) is the name of a family of low-level intermediate representations
- General form:

  x ← y *op* z

  with one operator (*op*) and (at most) three operands
- Example:

  z ← x – 2 * y

  becomes

  t ← 2 * y

  z ← x - t

Example TAC

```
...
for (i = 0; i < 10; i++) {
     a = i*i;
}
...
```

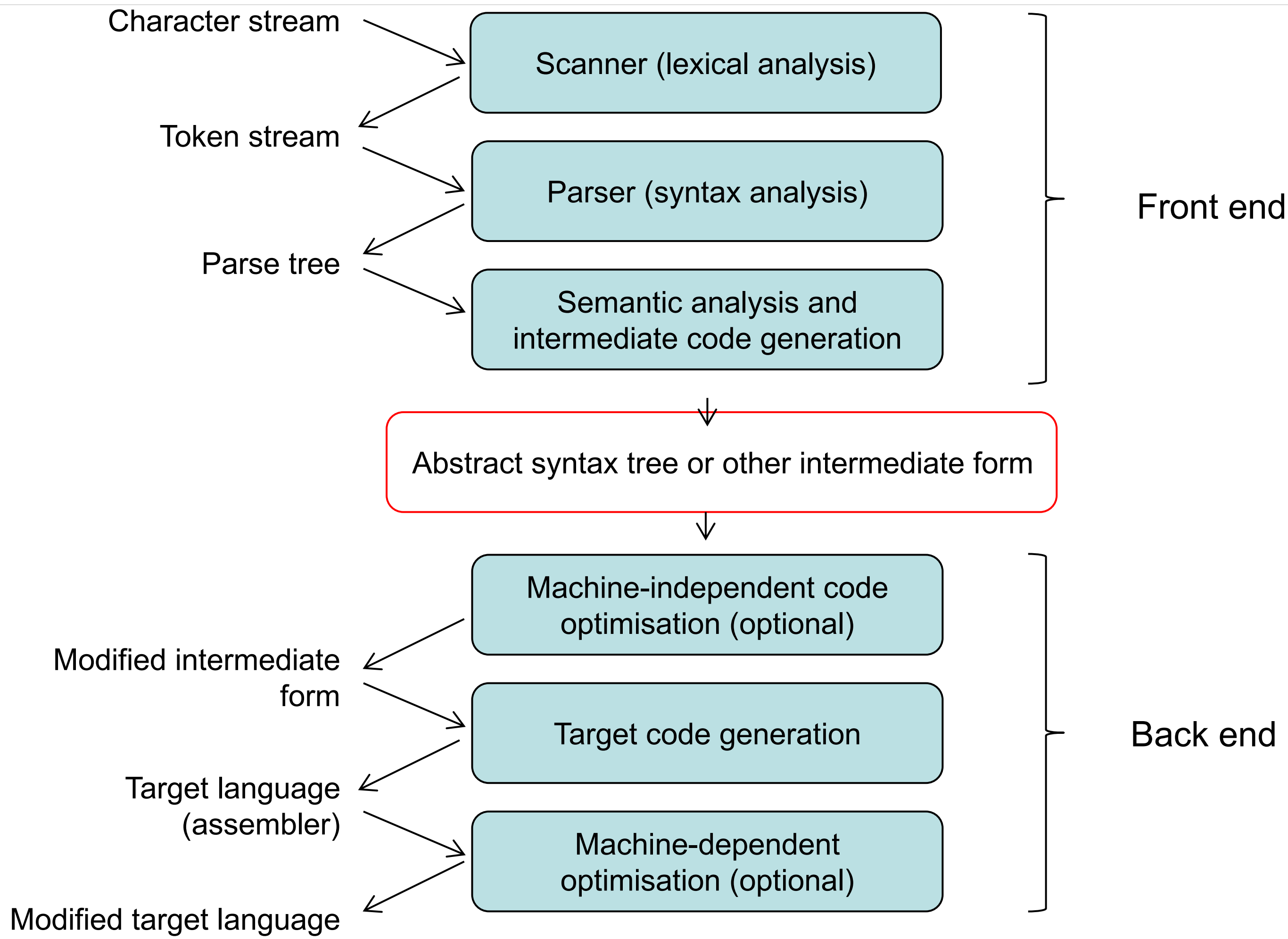
⇒

```
     t1 ← 0                 ; i = 0
L1:  if t1 >= 10 goto L2    ; i < 10
     t2 ← t1 * t1           ; a = i*i
     t1 ← t1 + 1            ; i++
     goto L1                ; repeat
L2:                         ; end of loop
```

TAC (continued)

- Advantages:
  - Resembles machine code
  - Compact form
- Many variations
  - Quadruples (naïve, simple approach)

    (op, dest, arg1, arg2) – arg2 is optional
  - Triples (similar, but dest is implicit from row index)
  - Static single assignment (SSA) (every reference to a name replaced by a unique name)


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Front end
Abstract syntax tree or other intermediate form
Machine-independent code optimisation (optional)
Modified intermediate form
Target code generation
Target language (assembler)
Machine-dependent optimisation (optional)
Modified target language
Back end

Scalar Optimisations

Optimisation along a single thread of control

- Dead code elimination
- Code motion
- Strength reduction
- Common subexpression sharing
- Transformation to enable other transformations

Multiple Passes

- Compiler has already performed multiple passes of the input program
- Typically each transformation by the optimiser is at least one more pass
  - Slows down compilation but improves final executable
- There are thousands of algorithms available to perform the different types of transformations
  - Compiler writer must choose which ones to apply, and in which order

Dead Code Elimination

- This involves identifying code that has no impact on the program outcomes.
  - Most simple case is variables which are declared but never used.
  - Another common case is in unreachable statements, such as code after a return statement, or a condition that always evaluates to false

Dead Code Examples

```
int main() {
        int a = 5, b = 2, c;
        boolean debug = false;

        int count = get_input();
        if (debug)
                print (b+count);
        return (a*count);
}
```

```
int main() {
        int a = 5;


        int count = get_input();


        return (a*count);
}
```

Code Motion

- Involves detecting repeated calculations of same value within a loop and moving it to a single calculation outside the loop

```
for (i = 0; i < a.length(); i++) {
        // do something
}
```

```
int size = a.length();
for (i = 0; i < size; i++) {
        // do something
}
```

```
for (int i = 0; i < n; i++) {
        x = y + z;
        a[i] = 6 * i + x * x;
}
```

```
x = y + z;
tmp = x * x;
for (int i = 0; i < n; i++) {
        a[i] = 6 * i + tmp;
}
```

Strength Reduction

- Involves identification of costly operators that can be replaced with less costly operators
- Typically moving from multiplication to shift or addition operators

```
for (i = 0; i < n; i++)
        print (val*i);
```

```
tmp = 0;
for (i = 0; i < n; i++) {
        print (tmp);
        tmp += val;
}
```

```
for (i = 0; i < n; i++)
        print (val*i);
```

Iteration	i	val	val*i	print
		1		
1	0	1	0	0
2	1	1	1	1
3	2	1	2	2
4	3	1	3	3

```
tmp = 0;
for (i = 0; i < n; i++) {
        print (tmp);
        tmp += val;
}
```

Iteration	i	val	print	tmp
		1		0
1	0	1	0	1
2	1	1	1	2
3	2	1	2	3
4	3	1	3	4

Strength Reduction

$\boxed{y = b / 8}$

$\boxed{y = b >> 3}$

Binary	Decimal
00010000	16
00000010	2

Common Subexpression Sharing

```
x = 2 * i + m;
y = 2 * i + s;
```

```
tmp = 2 * i;
x = tmp + m;
y = tmp + s;
```

Other Transformations

- Transformations themselves can sometimes introduce redundancies and other types of inefficiencies
- Equally, some transformations can facilitate further transformations
  - sometimes a pass will not itself produce an optimisation, but will facilitate one at the next pass

Method inlining

```
public int addOne (int val) {
   int z = 0;
   return val + 1;
}
```

```
public void main () {

   int x = addOne(3);

}
```

```
public void main () {

   int z = 0;
   int x = 3 + 1;

}
```

Dead-code elimination

```
public void main () {

   int x = 3 + 1;

}
```

Further Machine Independent Optimisation

- There are literally thousands of algorithms that have been developed for scalar optimisation
  - Big question is which are worthwhile
- You are not expected to know the algorithms, but demonstrate a knowledge of the types of optimisation that have been presented today
  - Identify code that could benefit from optimisation
  - Show how it could be transformed

Summary

- Optimisation is…
- Discuss need for optimisation
- Define machine independent optimisation
- Discuss low-level intermediate representations
- Describe five types of scalar optimisation
- Identify code that can be optimised, and demonstrate how

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- ~~Machine-independent optimisation~~
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review