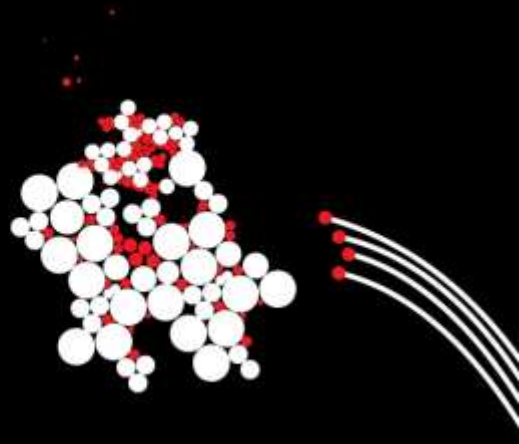
UNIVERSITY OF TWENTE.

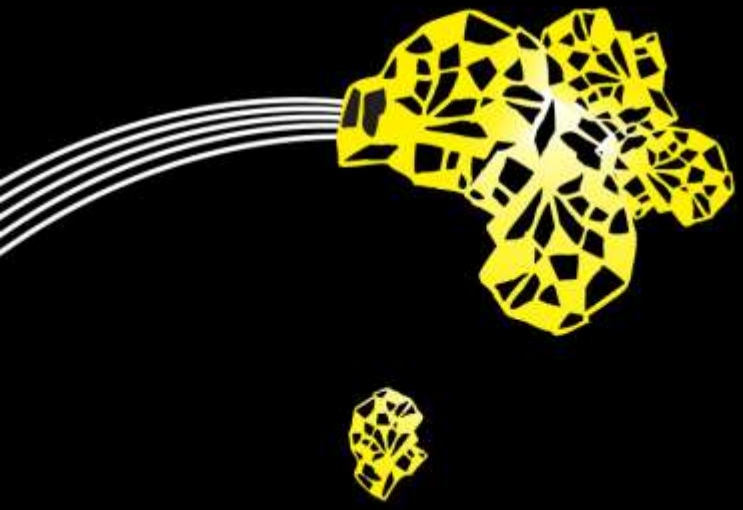


COMPILER CONSTRUCTION:

CC 6-3: OPTIMISATIONS & WRAP-UP

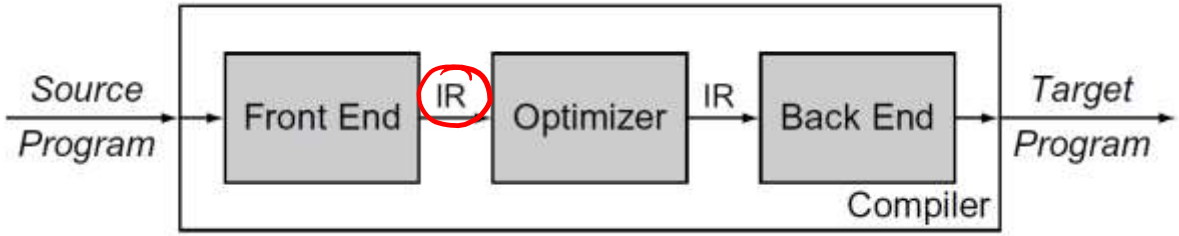
4 JUNE 2019

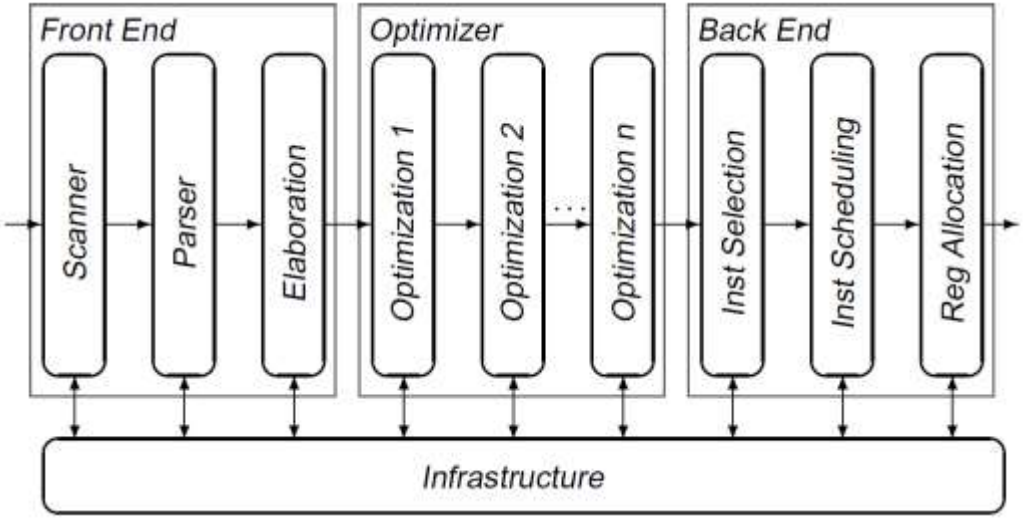
MODULE 8: PROGRAMMING PARADIGMS



THREE-PHASE COMPILERS

- We can play around with the intermediate representation
  - Optimize: rewrite so program gets faster/smaller/less power





TYPICAL OPTIMISATIONS: EC CHAPTERS 8-10

- Local optimisation: within basic blocks
  - *Value numbering:* removing code redundancy
  - *Tree balancing:* taking advantage of instruction-level parallelism in CPU

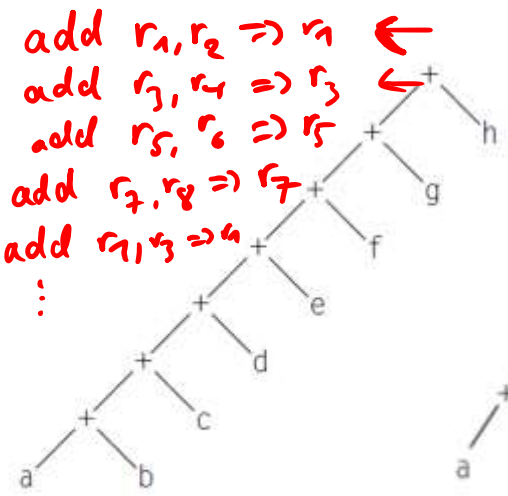
$a_3 \leftarrow b_1 + c_2$
$b_5 \leftarrow a_3 - d_4$
$c_6 \leftarrow b_5 + c_2$
$d \leftarrow a_3 - d_4$

Original Block

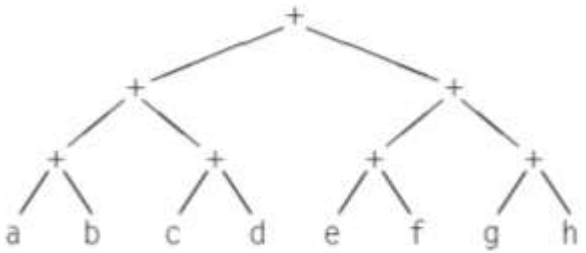
```
a ← b + c
b ← a - d
c ← b + c
d ← b
```

Rewritten Block

a + b + c + d + e + f + g + h



(a) Left-Associative Tree



(b) Balanced Tree

TYPICAL OPTIMISATIONS: EC CHAPTERS 8-10

- Regional optimisation: over extended basic blocks
  - *Super-local value numbering*
  - *Loop unrolling*

```
      do 60 j = 1, n2
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
50       continue
60    continue
```

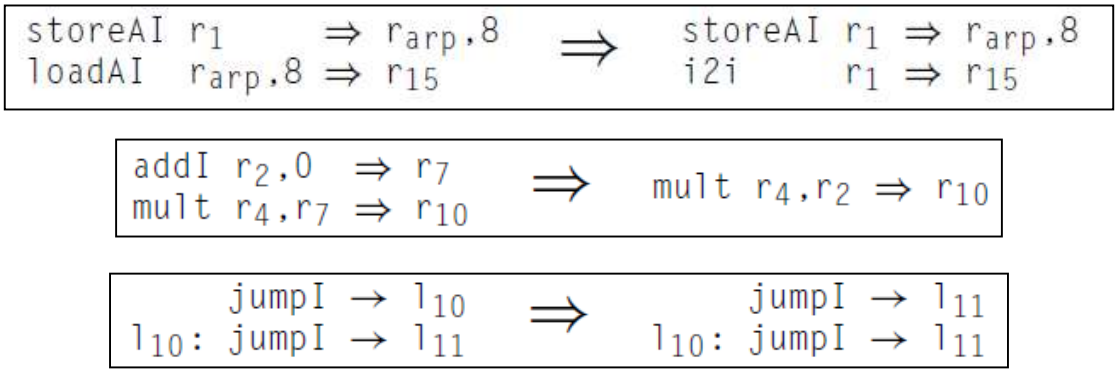
```
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
          do 50 i = 1, n1
           y(i) = ((((((((((((((( (y(i))
     $          + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $          + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $          + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $          + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $          + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $          + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $          + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $          + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
50        continue
60    continue
```

TYPICAL OPTIMISATIONS: EC CHAPTERS 8-10

- Local optimisation: within basic blocks
  - *Value numbering:* removing code redundancy
  - *Tree balancing:* taking advantage of instruction-level parallelism in CPU
- Regional optimisation: over extended basic blocks
  - *Super-local value numbering*
  - *Loop unrolling*
- Global (intraprocedural) optimisation: within single procedure
  - *Code placement:* keep related and frequently executed code together
  - Moving computation outside loops, eliminating dead code
- Interprocedural optimisation:
  - *Inlining:* Avoiding method calls by moving code from callee to caller
  - *Procedure placement*
- Analysis techniques
  - Control flow and data flow analysis
  - Static single assignment form

INSTRUCTION SELECTION: EC CHAPTER 11

- From intermediate representation to actual target machine instructions
- For linear IRs, this overlaps with construction of IR (elaboration phase)
  - Instruction selection is easy if target machine is similar to IR
  - Non-trivial if the architecture is (very) different
  - E.g., from stack-based IR to register-based CPU or vice versa
- *Peephole optimisation*: find local improvements



- Why are these optimisations not found earlier?
  - Different scope & level of granularity

INSTRUCTION SCHEDULING: EC CHAPTER 12

- Reorders machine code
  - Purpose: speed-up
  - Avoiding *interlocks*: artificial delays to wait for a value to become available
  - Taking advantage of instruction-level parallelism of target machine
- Again: local and regional scheduling
  - Local: *list scheduling*
  - Regional: *trace scheduling*
- Why are these optimisations not found earlier?
  - Requires awareness of characteristics of target machine
  - Target machine not known before this phase

REGISTER ALLOCATION + ASSIGNMENT: EC CHAPTER 13

- So far, we have assumed unlimited name space for registers
  - Every real machine has limited set of registers
  - Some registers are dedicated to special values/operations

- Register allocation
  - Maps unlimited name space to set of actual register names
    - No more than *m* registers active at any given moment
    - Both for register-to-register and for memory-to-memory
  - Decides when to *spill* values, i.e. save to memory

NP-complete (in program size)

- Register assignment
  - Maps actual register names to physical registers

Polynomial (in program size)

FROM THE INTRO: OUR PROMISE

- How do you make a computer understand human-readable text?
  - Correctly parse and execute 2 + x * 3 and (2 + x) * 3 and 2 * x + 3

```
grammar Expr;

prog : stat+ ;

stat : expr LINE
     | ID '=' expr LINE
     | LINE ;

expr : expr ( '*' | '/' ) expr
     | expr ( '+' | '-' ) expr
     | INT
     | ID
     | '(' expr ')' ;

ID   : [a-zA-Z]+ ;  // match identifiers
INT  : [0-9]+ ;     // match integers
LINE : '\r'? '\n' ;  // line ending = separator
WS   : [ \t]+ -> skip ;  // toss out whitespace
```

Ten weeks from now, you'll be able to
- write this grammar
- generate executing code

Source: *The Definitive Antlr4 Reference, Terence Parr*

THE REMAINDER OF THE COURSE – CC PART

- Two more CC lab sessions:
  - Today, after the lecture
  - Next week Wednesday (June 12), all afternoon
- Block 5 sign-off
  - Hard deadline: tomorrow (no lab session – so do it today!)
- Block 6 sign-off
  - Hard deadline: June 12 (see above!)
- Second take-home exam
  - Now on Canvas
  - Remember: to be done individually
  - Deadline: Monday, June 17
- Tomorrow
  - Q&A for second take-home exam
  - Excursion: CC in Formal Methods and Tools
- The Final Project