A compiler

High level (algorithm level) statements

Source code → Front-end → IR → Middle-end → IR → Back-end → Assembly generation

- Register allocation
- Assembly generation

Machine code

The language needs to help humans to write (efficient and robust) code.

The language needs to be easy to be analyzed and transformed.

The language needs to be easy to execute efficiently.

Explicit, simple, and architecture-independent instructions

Only a few registers, explicit instructions with constraints (e.g., lea)
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
From L2 to IR going through L3

Explicit, simple, and architecture-independent instructions designed for code analysis and transformation

Explicit semantic (e.g., add)

add, br, load, store (no lea)

no registers, no calling convention

Small piece of computation
**L2 language**

- Explicit entry point
- Explicit calling convention
- Complex per-instruction semantic
- Registers and variables

```plaintext
(@go (@go 0
   rdi <- 5
   mem rsp-8 <- :myF_ret
   call @myF 1
   :myF_ret
   %myRes <- rax
   %myRes @ %myRes %myRes 4
   return )
(@myF 1
   rax <- rdi
   return
   ) )
```

**L3 language**

- Pre-defined entry point
- Hidden calling convention
- Simple per-instruction semantic
- Variables only

```plaintext
define @main (){
   %myRes <- call @myF(5)
   %v1 <- %myRes * 4
   %myRes <- %myRes + %v1
   return
}
define @myF (%p1){
   %p2 <- %p1 + 1
   return %p2
}
```
p ::= (lf)
f ::= (INi)
i ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
    w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
    w <- t cmp t | cjump t cmp t label | label | goto label |
    return | call u N | call print 1 | call input 0 | call allocate 2 | call tensor-error F |
    w ++ | w -- | w @ w w E
w ::= a | rax
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx | var
s ::= t | label | l
t ::= x | N
u ::= w | l
x ::= w | rsp
aop ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
E ::= 1 | 2 | 4 | 8
F ::= 1 | 3 | 4
M ::= multiplicative of 8 constant (e.g., 0, 8, 16)
N ::= (+|-)? [1-9][0-9]* | 0
l ::= @name
label ::= :name
name ::= sequence of chars matching [a-zA-Z_] [a-zA-Z_0-9]*
var ::= %name
Explicit signature

No CISC instructions

No calling convention

The scope of labels is the function!
L3 program examples

define @main (){  
  %myRes <- call @myF(5)  
  %v1 <- %myRes * 4  
  %v2 <- %myRes + %v1  
  return %v2  
}
define @myF (%p1){  
  %l1 <- %p1 + 1  
  return %l1  
}
define @main (){  
  %v1 <- 1  
  %v2 <- 2  
  %v3 <- %v1 >= %v2  
  return %v3  
}
define @myEqual (%p1, %p2){  
  %v3 <- %p1 = %p2  
  br %v3 :myLabelTrue  
  return 0  
  :myLabelTrue  
  return 1  
}
define @main (){  
  %ret <- call @myEqual(3,5)  
  return %ret  
}
Final notes on L3

As for L2:

• Values are encoded following the same rules of L1

• Same rules for memory heap allocation

• Same undefined behaviors
Now that you know the L3 language

1. Rewrite your sorting L2 program using L3 and

2. Write a new L3 program to perform matrix multiplication
   (Example of input file = MM.L3.in on canvas)
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
The L3 compiler (L3c)

- To build L3c: translate an L3 program to an equivalent L2
- We need to encode the calling convention API -> ABI
- We need to select which L2 instructions to use for the L3 ones Instruction selection
L3 parser

• Significantly simpler than the L2 parser
• Pay attention to the L3 grammar

\[
\begin{align*}
\text{i} & ::= \ldots \\
& \quad \text{call callee ( args ) | var} \leftarrow \text{call callee ( args )} \\
\text{callee} & ::= u \mid \text{print} \mid \text{allocate} \mid \text{tensor-error} \\
u & ::= \text{var} \mid l \\
\text{args} & ::= t \mid t , t^* \\
t & ::= \text{var} \mid N \\
\end{align*}
\]

• Same rule for all call instructions
Parsing an L3 program

define @main (){ %myRes <- call @myF(5) call @myF(5) return }
define @main (){ %myA <- call allocate(3, 1) call allocate(3, 1) return }
Entry point

define @main(){
    ...
}

Your work

(@main
 (@main
  0
  ...
 )
 ...
 )
Making the calling convention explicit: caller

define @main(){
  %v1 <- call @myF(3)
  ...
}

Your work

(@main
  @main
  0
  mem rsp -8 <- :myF_ret
  rdi <- 3
  call @myF 1
  :myF_ret
  %v1 <- rax
  ...
)
)
Making the calling convention explicit: callee

define @myF (%p1) {
    return %p1
}

Your work

(@myF
  1
  %p1 <- rdi
  rax <- %p1
  return
)
Stack arguments, registers, and variables

- L3c is responsible to allocate space on the stack for >6 arguments
- L3c can generate L2 code with registers and variables
- L2c already performs a good register allocation
- Good engineering: don’t replicate functionality
  - L3c should not perform register allocation
  - L3c should use variables always with the only exceptions of implementing the calling convention
Labels

• The L3 compiler needs to translate L3 instruction labels to L2 instruction labels
  • No need to change function names
  • L3 labels: the scope is the function
    • 2 labels with the exact name in 2 different function are possible
  • L2 labels: the scope is the program
    • 2 labels with the exact name are not possible

• A possible mapping from L3 labels to L2 ones:
  1. Find the longest label for the whole L3 program: LL
  2. Append “_global_” to it: LLG
  3. For every L3 label :LABELNAME of a function F, generate an L2 label by increasing a global counter and appending it to LLG

You can design your own translation scheme (it must be correct)
Label example

define @main ( ){
    :begin
    ...
    :end
    ...
}

• LL is ":begin"
• LLG is ":begin_global_"

:begin_global0

:begin_global1
Outline

• L3

• Translating L3 to L2: calling convention and labels

• Translating L3 to L2: instruction selection
A compiler

Middle-end

IR

Back-end

Instruction selection
Register allocation
Assembly generation

Machine code
Instruction selection

The process of selecting the lower-level instructions (assembly instructions) to use to translate a higher-level representation (e.g., L3)

Instruction selection is intra-procedural
Naive instruction selection for L3

```c
define @myF (%p1, %p2){
    %v1 <- %p1 * 4
    ...
}
```

Translate L3 instructions one by one and independently with the surrounding ones

```c
(@myF
  2
  %p1 <- rdi
  %p2 <- rsi
  %v1 <- %p1
  %v1 *= 4
  ...
)
```
Naive translation of an L3 function: problem

\[
\text{define } @\text{myF (}%p1, %p2)\{ \\
\quad %v1 <- %p1 * 4 \\
\quad %v2 <- %v1 + %p2 \\
\quad \ldots \\
\}
\]

Translate L3 instructions one by one and independently with the surrounding ones

Instruction selection depends on the context!

\[
(@\text{myF} \\
\quad 2 \ 0 \\
\quad %p1 <- \text{rdi} \\
\quad %p2 <- \text{rsi} \\
\quad %v1 <- %p1 \\
\quad %v1 *= 4 \\
\quad %v2 <- %v1 \\
\quad %v2 += %p2 \\
\quad \ldots \\
\}
\]

\[
(@\text{myF} \\
\quad 2 \ 0 \\
\quad %p1 <- \text{rdi} \\
\quad %p2 <- \text{rsi} \\
\quad %v2 @ %p2 %p1 4 \\
\quad \ldots \\
\}
\]

Is there a better translation?
Instruction selection: it isn’t that easy

```plaintext
define @myF (%p1, %p2) {
    %v1 <- %p1 * 3
    %v2 <- %v1 + %p2
    ...
}
```

Your work

```plaintext
(@myF
  2 0
  %p1 <- rdi
  %p2 <- rsi;
  %v2 @ %p2 %p1 3
  ...
)
```

Instruction selection must satisfy all constraints of the target language!
Instruction selection: context

• Instruction selection depends on the context

• Context for this class:
  sequence of instructions that does not include
  • a label instruction or
  • a call instruction

• The sequence must end when a branch or a return is encountered
  (the branch or return are part of the context)

\[
\begin{align*}
%V3 & \leftarrow %v2 + %v1 \\
%V4 & \leftarrow %v3 * 4 \\
\text{:a\_label} & \\
%V5 & \leftarrow %V4 * 2 \\
br & \text{:another\_label}
\end{align*}
\]
Instruction selection step 1: identify contexts

Inst = F.entryPoint()
C = new Context()
While (Inst != nullptr){
    if (Inst is not Label or a call) C.add(Inst)
    if (Inst is Label, Branch, Call, Return) {
        C = new Context()
    }
    Inst = F.nextInst(Inst)
}
Delete empty contexts

:myLabel
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :otherLabel
Instruction selection step 2: tree generation

We need to generate the tree representation of the instructions of a context, for every context:

• Generate a separate tree for every instruction
• The order of the trees define the order of translation/code generation (e.g., the first L2 instructions generated translate the first tree)

\[
\begin{align*}
\%v1 & \leftarrow \%p1 \times 4 \\
\%v2 & \leftarrow \%v1 + \%p2
\end{align*}
\]

![Diagram of tree structure with operations and variables]
Instruction selection step 3: merging trees

- We perform instruction selection per tree
  - A target instruction (e.g., @ in L2) cannot cover nodes that belong to different trees
  - The bigger is the tree, the more optimal the instruction selection can be
- We aim to make trees as big as possible
  - We have generated the smallest trees (one per instruction)
  - Now we need to merge them as much as possible
  - Quality – complexity tradeoff: this class targets what is reasonable for one week of work

\[
\begin{align*}
\%v1 & \leftarrow \%p1 \times 4 \\
\%v2 & \leftarrow \%v1 + \%p2 \\
\text{Ideal selection: } & \%v2 \leftarrow \%p2 \times \%p1 \times 4
\end{align*}
\]

We cannot obtain the ideal selection because the target instruction (@) would cover nodes of different trees.
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

When is it safe to merge trees?

\[
\begin{align*}
%v2 & \leftarrow %p1 \times 4 \\
%v2 & \leftarrow %v1 + %p2
\end{align*}
\]
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context
   
   Let $T_1$, $T_2$ be two trees that belong to the same context
   
   I. $T_1$ uses a variable $\%v$ defined by $T_2$
   
   II. What else?

    \[
    \begin{align*}
    \%v_1 &< - \%p_1 \times 4 \\
    \%v_2 &< - \%v_1 + \%p_2 \\
    \%
    \end{align*}
    \]
Instruction selection step 3: merging trees

\[
\begin{align*}
%v1 & \leftarrow %p1 \times 4 \\
%v2 & \leftarrow %v1 + %p2 \\
\text{br} & : \text{MYL} \\
\vdots & : \text{MYL} \\
%v3 & \leftarrow %v1 + 1
\end{align*}
\]

Is it correct?
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $\%v$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so

A. $\%v$ is dead after the instruction related to $T_1$ or $\%v$ is only used by $T_1$
Instruction selection step 3: merging trees

%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :MYL
...
:MYL
%v3 <- %v1 + 1
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $%v$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so

   A. $%v$ is dead after the instruction attached to $T_1$ or $%v$ is only used by $T_1$
   B. What else?

\[
\begin{align*}
%v_1 & \leftarrow %p_1 \times 4 \\
%v_2 & \leftarrow %v_1 + %p_2
\end{align*}
\]
Instruction selection step 3: merging trees

%v1 ← %p1 * 4
%v3 ← %v1 + 1
%v2 ← %v1 + %p2

Is it correct?
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $\%v$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so

A. $\%v$ is dead after the instruction attached to $T_1$ or $\%v$ is only used by $T_1$

B. No instruction that depends on $T_2$ between $T_2$ and $T_1$

Including $T_2$ in this range
• Dependences exist between instructions when they both access a variable or memory location and one of them is a write

• For variables the condition B of the previous slide becomes the following
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $%v$ defined by $T_2$
II. Merge $T_2$ into $T_1$ only when it is safe to do so
   A. $%v$ is dead after the instruction attached to $T_1$ or $%v$ is only used by $T_1$
   B. No instruction that depends on $T_2$ between $T_2$ and $T_1$

Including $T_2$ in this range

Should we merge?

\[
\begin{align*}
%v_1 & \leftarrow %p_1 \times 4 & T_2 \\
%v_2 & \leftarrow %v_1 + %p_2 & T_1
\end{align*}
\]

%v1

%v2

* +

%p1 4 %v1 %p2

T2

T1
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $\%v$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so
   A. $\%v$ is dead after the instruction attached to $T_1$ or $\%v$ is only used by $T_1$
   B. No other uses of $\%v$ between $T_2$ and $T_1$ and
      Including $T_2$ in this range
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $\%v$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so

A. $\%v$ is dead after the instruction attached to $T_1$ or $\%v$ is only used by $T_1$

B. No other uses of $\%v$ between $T_2$ and $T_1$ and $\%v_1 < -\%v_1 * 4$ $\%v_2 < -\%v_1 + \%v_1$

Including $T_2$ in this range

Should we merge?
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context.

2. Merge trees (as much as possible) that belong to the same context.

Let $T_1$, $T_2$ be two trees that belong to the same context.

I. $T_1$ uses a variable $%v$ defined by $T_2$.

II. Merge $T_2$ into $T_1$ only when it is safe to do so.

   A. $%v$ is dead after the instruction attached to $T_1$ or $%v$ is only used by $T_1$.

   B. No other uses of $%v$ between $T_2$ and $T_1$ and What else?

\[
%v_1 \leftarrow %p_1 \times 4
\]

\[
%v_2 \leftarrow %v_1 + %p_2
\]

Should we merge?
Instruction selection step 3: merging trees

%v1 <- %p1 * 4
%p1 <- %p1 + 1
%v2 <- %v1 + %p2

br :MYL

Is it correct?
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context

2. Merge trees (as much as possible) that belong to the same context

Let $T_1$, $T_2$ be two trees that belong to the same context

I. $T_1$ uses a variable $\%V$ defined by $T_2$

II. Merge $T_2$ into $T_1$ only when it is safe to do so

A. $\%V$ is dead after the instruction represented by $T_1$ or $\%V$ is only used by $T_1$

B. No other uses of $\%V$ between $T_2$ and $T_1$ and no definitions of variables used by $T_2$ between $T_2$ and $T_1$
• The previous condition excludes the possibility to have instructions between T2 and T1 that depends on T2

• **Dependence definition:**
  two generic instructions depend on each other if they both access a variable or memory location and one of them is a write

• If T2 accesses a memory location, then condition B becomes the following
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context
   
   Let $T_1, T_2$ be two trees that belong to the same context
   
   I. $T_1$ uses a variable $V$ defined by $T_2$
   
   II. Merge $T_2$ into $T_1$ only when it is safe to do so
       
       A. $V$ is dead after the instruction attached to $T_1$ or $V$ is only used by $T_1$
       
       B. No memory instruction between $T_2$ and $T_1$

   Including $T_2$ in this range

Should we merge?
Instruction selection step 4: tiling trees

• Tile = instruction of the target language (e.g., L2) = pattern
• Instruction selectors use pattern-matching on trees with tiles
  • Use a tree-based code representation
  • Each target instruction defines a tile (pattern) that can be used to cover the tree
  • Used tiles (patterns) = selected target instructions to generate

\[
\%v1 \leftarrow %p1 \times 4 \\
\%v2 \leftarrow %v1 + %p2 \\
\text{var} *= N
\]
From L3 instructions to L2 instructions

1. Translate L3 instructions of a context into a list of trees
   • Order needs to be preserved
2. Merge as many trees as possible
3. For each tree (in order):
   A. **Tiling**: cover the tree with L2 tiles
   B. **Code generation**: from the bottom to the top of the tree:
      i. Get the next tile
      ii. Append L2 instructions generated by the current tile
Example: tiles and tiling

? *= ?

VAR3

* 

VAR1

VAR2

VAR3 *= VAR2

VAR2 <- VAR1

VAR2 <- VAR1

%v1 <- %p1

%v3 <- %v2 * %v1

%v3

*

%v2

%v3

*

%v1

<=

%v1

<=

%p1

%v3 *= %v1

%p1
Specialized tiles

VAR3 <- VAR1
VAR3 *= VAR2

var1 *= var2

%v1 <- %v1*%v2
%v3 <- %v1 * 5

%v3 *= 5
Large tiles

\[ ? \times = ? \]

\[
\begin{array}{c}
\text{VAR1} \\
+ \\
\text{VAR2}
\end{array}
\]

\[
\begin{array}{c}
\text{VAR3} \\
\div \\
\text{VAR1}
\end{array}
\]

\[
\text{VAR3} \leftarrow \text{VAR1} \\
\text{VAR3} \mathbf{+=} \text{VAR2}
\]

\[
\begin{array}{c}
\text{VAR1} \\
+ \\
\text{VAR2}
\end{array}
\]

\[
\begin{array}{c}
\text{VAR1} \\
+ \\
\text{VAR3}
\end{array}
\]

\[
\text{VAR3} \leftarrow \text{VAR1} \\
\text{VAR3} \mathbf{+=} \text{VAR2}
\]

\[
\begin{array}{c}
\text{VAR2} \\
+ \\
\text{VAR1}
\end{array}
\]

\[
\begin{array}{c}
\text{VAR1} \\
+ \\
\text{VAR1}
\end{array}
\]

\[
\text{var1} \mathbf{+=} \text{var2}
\]

\[
\text{var2} \leftarrow \text{var1} \\
\text{var1} \mathbf{\ll= 2}
\]
Tiles and tiling

- Tiles capture compiler’s understanding of the target instruction set

- In general, for any given tree, many tilings are possible
  - Each resulting in a different instruction sequence

- We ensure pattern coverage by covering, at a minimum, all atomic L3 trees
The instruction selection problem

• Many solutions to cover a tree are possible

• How to pick tiles that cover our tree with minimum execution time?

• Need a good selection of tiles
  • Small tiles to make sure we can tile every tree
  • Large tiles for efficiency
Quality of a tile in CC

• Instruction selection should prefer high-quality tiles

• The quality of a tile $t$ is related to the latency of the instructions generated by $t$

• In this class, we use the number of instructions as proxy to the latency

• Hence, if two tiles cover the same sub-tree, then we choose the one that has less instructions
  • Each tile reports the number of instructions generated by it
Tiles in CC

• Tiles need to be designed such that a large tile $t$ has $\leq$ instructions than a possible set of small tiles that cover the same sub-tree

• Hence, we prefer larger tiles: fewer instructions

```
VAR3 <- VAR1
VAR3 += VAR2
var2 <- var1
var1 <<= 2
```
Quality of a solution of the tiling problem

• Tiling problem: choose a set of tiles to cover a tree

• Quality of a tiling solution: the cumulative execution time of all instructions generated to cover a tree

• In instruction selection, we estimate the total execution time as the sum of costs of all tiles
  • In this class: the cost of a tile is the number of instructions of it
  • Hence, in in this class: the quality of a tiling solution is the total number of instructions generated
Example of tiling cost for L3

? *= ?

VAR3

VAR1

VAR2

? <- ?

VAR2 <- VAR1

VAR1

VAR3 <= VAR1

VAR3 *= VAR2

%v1 <- %p1

%v3 <- %v2 * %v1

%v3

%v2

%v1

%p1

%v1 <- %p1

%v3 <- %v2

%v3 *= %v1

Total cost: 3
Other examples of L2 tiles

\[
\text{VAR} \quad \text{VAR} \\
\downarrow \quad + \\
\text{VAR} \quad \text{CONST}
\]

\[\text{Cost:1}\]

\[
\text{VAR} \quad \text{VAR} \\
\downarrow \quad \downarrow \\
\text{VAR} \quad \text{VAR} \\
\downarrow \quad \ast \\
\text{VAR} \quad \text{CONST 1, 2, 4}
\]

\[\text{Cost:1}\]
Global vs. local optimal solution

- We want the “lowest cost” tiling
  - Take into account cost/delay of each instruction (i.e., timing model)

- **Optimum** tiling:
  lowest-cost tiling

- **Locally Optimal** tiling:
  no two adjacent tiles can be combined into one tile of lower cost
Locally optimal tilings

• A simple greedy algorithm works extremely well in practice: **Maximal munch**

• Choose the largest pattern with lowest cost, i.e., the “maximal munch”

• Algorithm:
  • Start at root
  • Use “biggest” match (in # of nodes)
    • This is the munch
    • Use cost to break ties
  • Recursively apply maximal much at each subtree of this munch
Maximal munch example

\[
%v2 \gets %v1 + 8 \\
%v3 \gets %v3 * 2 \\
%v3 \gets %v3 * 4 \\
%v4 \gets \text{load } %v2 \\
%v5 \gets %v4 + %v3
\]
Maximal munch example

%v2 <- %v1 + 8

%v3 <- %v3 * 2

%v3 <- %v3 * 4

%v4 <- load %v2

%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
→%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2

⇀ %v5 <- %v4 + %v3
Maximal munch example

\%v2 \leftarrow \%v1 + 8
\%v3 \leftarrow \%v3 \times 2
\%v4 \leftarrow \text{load} \%v2
\%v5 \leftarrow \%v4 + \%v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3

VAR3 <- VAR1
VAR3 *= VAR2

Cost: 2
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

\[
\begin{align*}
\%v2 &\gets \%v1 + 8 \\
\%v3 &\gets \%v3 \times 2 \\
\%v3 &\gets \%v3 \times 4 \\
\%v4 &\gets \text{load } \%v2 \\
\%v5 &\gets \%v4 + \%v3
\end{align*}
\]

Cost: 4

Biggest munch!

\[\text{load } \%v2\]

\[\%v3 \leftarrow \text{mem } \%v1 \text{ CONST}\]

\[\%v4 \leftarrow \text{load } \%v2\]

\[\%v5 \leftarrow \%v4 + \%v3\]
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

\[
\begin{align*}
%v2 &= \%v1 + 8 \\
%v3 &= \%v3 \times 2 \\
%v3 &= \%v3 \times 4 \\
%v4 &= \text{load } %v2 \\
%v5 &= %v4 + %v3
\end{align*}
\]
Maximal munch example

\[
\begin{align*}
%v2 & \leftarrow %v1 + 8 \\
%v3 & \leftarrow %v3 \times 2 \\
%v3 & \leftarrow %v3 \times 4 \\
%v4 & \leftarrow \text{load} \ %v2 \\
%v5 & \leftarrow %v4 + %v3
\end{align*}
\]
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3

%v3 <<= 8
%v4 <- mem %v1 8
Maximal munch example

%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3

%v3 <<= 8
%v4 <- mem %v1 8
Maximal munch example

%v2 ← %v1 + 8
%v3 ← %v3 * 2
%v3 ← %v3 * 4
%v4 ← load %v2
%v5 ← %v4 + %v3

%v3 <<= 8
%v4 ← mem %v1 8
%v5 ← %v4
%v5 += %v3
Maximal munch example

\[
\begin{align*}
%v2 & \leftarrow %v1 + 8 \\
%v3 & \leftarrow %v3 \times 2 \\
%v3 & \leftarrow %v3 \times 4 \\
%v4 & \leftarrow \text{load} \ %v2 \\
%v5 & \leftarrow %v4 + %v3 \\
%v3 & \leftarrow= 8 \\
%v4 & \leftarrow \text{mem} \ %v1 \ 8 \\
%v5 & \leftarrow %v4 \\
%v5 & \leftarrow= %v3
\end{align*}
\]
Maximal munch

• Maximal munch does not necessarily produce the optimum selection of instructions

• But:
  • it is easy to implement
  • it tends to work “well” for current instruction-set architectures
... but if we want the optimum?
Instruction selection complexity

• Finding the optimum for tree: P

• Finding the optimum for DAG: NP
  • Countless number of heuristics proposed (including the one described in this class)
  • Dynamic programming

• Most (all) of programs we run are DAGs
Homework #3: the L3 compiler

For every L3 function $f$

1. Label globalization
2. Instruction selection
3. Excluding *only* Instruction selection step 3: merging trees
4. API $\rightarrow$ ABI

L2 function
Always have faith in your ability

Success will come your way eventually

Best of luck!