Simone Campanoni
simonec@eecs.northwestern.edu
A compiler

Source code -> Front-end -> IR

- High level (algorithm level) statements

IR -> Middle-end

- Explicit, simple, and architecture-independent instructions
- The language needs to be easy to be analyzed and transformed

IR -> Back-end

- Only a few registers, explicit instructions with constraints (e.g., lea)
- Register allocation
- Assembly generation

Back-end -> Machine code

- The language needs to help humans to write (efficient and robust) code
- The language needs to be easy to execute efficiently
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

L3 language

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

(define :main (){
  %myRes <- call :myF(5)
  %v1 <- %myRes * 4
  %myRes <- %myRes + %v1
  return %myRes
}
define :myF (%p1) {
  %p2 <- %p1 + 1
  return %p2
}
p ::= (label f+)
f ::= (label N i+)
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 allocate 2 | call array-error 2 |
    w ++ | w -- | w @ w w E
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx | var
s ::= t | label
t ::= x | N
u ::= w | label
x ::= w | rsp
aop ::= += | -= | *= | &:=
sop ::= <<= | >>=
cmp ::= < | <= | =
E ::= 1 | 2 | 4 | 8
M ::= N times 8
N ::= (+|-)? [1-9][0-9]*
var ::= %name
label ::= sequence of chars matching :[a-zA-Z][a-zA-Z]0-9]*
Explicit signature

\[ p ::= f^+ \]
\[ f ::= \text{define} \ \text{label} \ ( \vars \ ) \ \{ \ i^+ \} \]
\[ i ::= \text{var} \leftarrow s \ | \ \text{var} \leftarrow t \ \text{op} \ t \ | \ \text{var} \leftarrow t \ \text{cmp} \ t \ | \]
\[ \quad \text{var} \leftarrow \text{load} \ \text{var} \ | \ \text{store} \ \text{var} \leftarrow s \ | \]
\[ \quad \text{return} \ | \ \text{return} \ t \ | \ \text{label} \ | \ \text{br} \ \text{label} \ | \ \text{br} \ \text{var} \ \text{label} \ | \]
\[ \quad \text{call} \ \text{callee} \ ( \ \text{args} \ ) \ | \ \text{var} \leftarrow \text{call} \ \text{callee} \ ( \ \text{args} \ ) \]
\[ \text{callee} ::= u \ | \ \text{print} \ | \ \text{allocate} \ | \ \text{array-error} \]
\[ \vars ::= \ | \ \text{var} \ | \ \text{var} \ (, \ \text{var})^* \]
\[ \text{args} ::= \ | \ t \ | \ t \ (, \ t)^* \]
\[ s ::= t \ | \ \text{label} \]
\[ t ::= \text{var} \ | \ N \]
\[ u ::= \text{var} \ | \ \text{label} \]
\[ \text{op} ::= + \ | \ - \ | \ * \ | \ & \ | \ << \ | \ >> \]
\[ \text{cmp} ::= < \ | \ <= \ | \ = \ | \ >= \ | \ > \]
\[ N ::= (+|-)? [1-9][0-9]^* \]
\[ \text{label} ::= :\text{name} \]
\[ \text{var} ::= \%\text{name} \]
\[ \text{name} ::= \text{sequence} \text{of} \text{chars} \text{matching} \ [a-zA-Z_][a-zA-Z_0-9]^* \]
define :main (){  
  %myRes <- call :myF(5)  
  %v1 <- %myRes * 4  
  %v2 <- %myRes + %v1  
  return %v2  
}
define :myF (%p1){  
  %l1 <- %p1 + 1  
  return %l1  
}
define :myEqual (%p1, %p2){  
  %v3 <- %p1 = %p2  
  br %v3 :myLabelTrue  
  return 0  
  :myLabelTrue  
  return 1  
}
define :main (){  
  %ret <- call :myEqual(3,5)  
  return %ret  
}
Now that you know the L3 language

Rewrite your L2 programs in L3 and

write a new L3 program with more than 30 instructions
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

\[
\begin{align*}
\text{i} & \ ::= \ ... \\
& \quad \text{call callee ( args ) } | \text{ var } <\text{ call callee ( args )} \\
\text{callee} & \ ::= \ u \ | \ \text{print} \ | \ \text{allocate} \ | \ \text{array-error} \\
\text{u} & \ ::= \ \text{var} \ | \ \text{label} \\
\text{args} & \ ::= \ t \ | \ t (, t)^* \\
\text{t} & \ ::= \ \text{var} \ | \ N
\end{align*}
\]

• 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

```plaintext
define :main()
    %v1 <- call :myF(3)
    ...
}
```

Your work:

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

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

(: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 labels to L2 labels
  • 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”, generate an L2 label by appending “LABELNAME” to LLG
Label example

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

- LL is "begin"
- LLG is "begin_global_"

\[\begin{align*}
\text{begin} & \quad \text{begin}_\text{global}_\text{begin} \\
\text{begin} & \quad \text{begin}_\text{global}_\text{end}
\end{align*}\]
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

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

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

Define:
```l3
define :myF (%p1, %p2){
    %v1 <- %p1 * 4
    %v2 <- %v1 + %p2
    ...
}
```

Translate L3 instructions one by one and independently with the surrounding ones:
```l3
(:myF 2 0
 %p1 <- rdi
 %p2 <- rsi
 %v1 <- %p1
 %v1 *= 4
 %v2 <- %v1
 %v2 += %p2
 ...
)
```

Instruction selection depends on the context!

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

Define: `myF (%p1, %p2) {`  

```
%v1 <- %p1 * 3
%v2 <- %v1 + %p2
...
```

End

Your work:

```
(: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

%V3 <- %v2 + %v1

%V4 <- %v3 * 4

:a_label

%V5 <- %V4 * 2

br :another_label

In CC: a context is almost a basic block (we will learn the concept of basic block next week)
Instruction selection step 1: identify contexts

```c
Inst = F.entryPoint()
C = new Context()
While (Inst != nullptr){
    if (Inst is not Label) C.add(Inst)
    if (Inst is Label or Inst is Branch) {
        C = new Context()
    }
    Inst = F.nextInst(Inst)
}
Delete empty contexts
```

:myLabel

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

We need to generate the tree representation of the instructions

- Generate a tree for every instruction (including labels)
- A tree for a label contains just one node
- The order of the trees define the order of translation (the first L2 instructions generated translate the first tree)

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

Code generation order
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

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

Ideal selection: \( %v2 @ %p2 %p1 4 \)

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?
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 T1, T2 be two trees that belong to the same context
   
   I. T1 uses a variable %V defined by T2
   II. What else?

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

\[ \%v1 \leftarrow \%p1 \times 4 \]
\[ \%v2 \leftarrow \%v1 + \%p2 \]
\[ \text{br :MYL} \]
\[ \text{... :MYL} \]
\[ \%v3 \leftarrow \%v1 + 1 \]

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 T1, T2 be two trees that belong to the same context

I. T1 uses a variable %V defined by T2
II. Merge T2 into T1 only when it is safe to do so
   A. %V is dead after the instruction attached to T1 or %V is only used by T1

\[
\begin{align*}
%v1 & \leftarrow %p1 \times 4 \\
%v2 & \leftarrow %v1 + %p2
\end{align*}
\]
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 T1, T2 be two trees that belong to the same context

I. T1 uses a variable %V defined by T2

II. Merge T2 into T1 only when it is safe to do so
   A. %V is dead after the instruction attached to T1 or %V is only used by T1
   B. What else?

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

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

\[
\begin{align*}
%v1 & \leftarrow %p1 \times 4 \\
%v3 & \leftarrow %v1 + 1 \\
%v2 & \leftarrow %v1 + %p2 \\
\text{merge} & \\
%v2 & \leftarrow %p2 \%p1 \ 4 \\
\text{Is it correct?}
\end{align*}
\]
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 T1, T2 be two trees that belong to the same context.

I. T1 uses a variable %V defined by T2.
II. Merge T2 into T1 only when it is safe to do so.
   - A. %V is dead after the instruction attached to T1 or %V is only used by T1.
   - B. No other uses of %V between T2 and T1.
   - C. What else?

\[
\begin{align*}
%v1 & \leftarrow %p1 \times 4 \\
%v2 & \leftarrow %v1 + %p2
\end{align*}
\]
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 attached to $T_1$ or $%V$ is only used by $T_1$

B. No other uses of $%V$ between $T_2$ and $T_1$

C. No definitions of variables used by $T_2$ between $T_2$ and $T_1$

\[
%v1 \leftarrow %p1 \times 4
\]

\[
%v2 \leftarrow %v1 + %p2
\]
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

\[
\%v1 \leftarrow \%p1 \times 4 \\
\%v2 \leftarrow \%v1 + \%p2
\]

```
var *= N
    *  VAR? N?
```

```
\%v2 += \%v1 + \%p2
```

```
\%p2
```

```
\%p1
    4
```

```
\%v1
```

```
+ 
```

```
\%v2
```
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
VAR1 = VAR1
VAR2 = VAR1

%v1 = %p1
%v3 = %v2 %v1
%v1 = %p1
%v3 = %v2
%v3 = %v1
Specialized tiles

\[ ? \times = ? \]

\[ VAR3 \times = VAR1 \times = VAR2 \]

\[ VAR3 \times = VAR1 \times = VAR2 \]

\[ \%v1 \times = \%v2 \]

\[ \%v1 \times = \%v2 \]

\[ \%v3 \times = 5 \]

\[ \%v3 \times = 5 \]
Large tiles

? * = ?

VAR1 + VAR2

VAR3

VAR1

VAR2

VAR1

VAR1

VAR2

VAR3

VAR1

VAR1

VAR1

VAR1

VAR2

VAR1

VAR1

VAR3

VAR1

VAR1

VAR1

VAR1

VAR2

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1

VAR1
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
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 CC: the cost of a tile is the number of instructions of it
  • Hence, in CC: the quality of a tiling solution is the total number of instructions generated
Example of tiling cost for L3

Cost: 2

VAR1

VAR2

Cost: 1

VAR1

%v1 <- %p1
%v3 <- %v2 * %v1

%v2

%v3 <- %v1

%v3 *= %v1

VAR3 <- VAR1
VAR3 *= VAR2

VAR2 <- VAR1

%v1 <- %p1
%v3 <- %v2
%v3 *= %v1

Total cost: 3
Quality of a tile in state-of-the-art compilers

• Instruction selection should prefer high-quality tiles

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

• Instructions $\neq$ cycles: different instructions may take different cycles to execute
  • Each tile $t$ reports the number of clock cycles that are required to execute the generated instructions of $t$
Timing model

• Idea: associate cost with each tile (proportional to # cycles to execute)
  • Caveat: cost is fictional on modern architectures
• Estimate of total execution time is sum of costs of all tiles

Total cost: 5
Example of L2 tiles

\[
\begin{align*}
\text{VAR} + \text{CONST} \quad \text{Cost: 1} \\
\text{VAR} + \text{VAR} \\
\text{VAR} + \text{VAR} \quad \text{Cost: 1}
\end{align*}
\]
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

Total cost: 4

%v2 <- %v1 + 8
%v3 <- %v2
%v3 <- 5 + %v3
%v4 <- load %v2

v4 <- mem v1 8
v3 <- v1
v3 += 8
v3 += 5
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?
Finding optimum tiling

• **Goal**: find minimum total cost tiling of tree

• **Algorithm**:
  • For every node, find minimum total cost tiling of that node and sub-tree

• **Lemma**:
  • Once minimum cost tiling of all children of a node is known,
  • We can find minimum cost tiling of the node by trying out all possible tiles matching the node

• **Therefore**: start from leaves, work upward to top node
Optimum selection

• To achieve optimum instruction selection: Dynamic programming

• In contrast to maximal munch, the trees are matched bottom-up

• But
  • Significantly more complex to implement
  • More time and memory consuming than maximal munch
Dynamic programming

• First pass: tiling
  • Working bottom up
  • Given the optimum tilings of all subtrees, generate optimum tiling of the current tree
    • Consider all tiles for the root of the current tree
    • Sum cost of best subtree tiles and each tile
    • Choose tile with minimum total cost

• Second pass: code generation
  • Generates the code using the obtained tiles
Value of instruction selection

- The simpler the target ISA is, the less important obtaining the optimum is
  - Reduced Instruction Set Computing (RISC)

- The more complex the target ISA is, the bigger is the gap between the solution found by a simple (e.g., maximal munch) instruction selection and the optimum one (e.g., dynamic programming)
  - Complex Instruction Set Computing (CISC)
Instruction selection complexity

• Finding the optimum for tree: P

• Finding the optimum for DAG: NP
  • Countless number of heuristics proposed

• Most (all) of programs we run are DAGs
Homework #4: the L3 compiler

For every L3 function $f$

1. Label globalization
2. Instruction selection
3. API -> ABI

L2 function
Homework #4 naïve implementation

• You can implement the L3 compiler without instruction selection but you will not get the related point (this is to avoid being stuck in this homework and move to the next one)
  • Don’t forget that you get an extra point if you will participate to the final competition

• Naïve and correct implementation of L3c:
  • Map each L3 instruction into L2 instructions at design time
  • Use this map to translate L3 instructions one by one without considering their context