Compilers

- Compilers translate a source language (e.g., C++) to a destination language (e.g., x86_64)
  - We use them every day
  - If you understand their internals, you better understand (and take advantage of) the tools you rely on
  - Are you interested in computer architectures? Their inputs is the outputs of a compiler
The L1 source language

- L1 is going to be the input of your first compiler L1c
- The output of L1c is an executable ELF binary that can run on Linux-based and Intel-based systems

```
L1 program → L1c → a.out
```
Outline

• L1 language

• Value encoding

• Calling convention

• Heap
From now on, we need to use the mindset of “we want to become L1 developers” rather than “we want to build a compiler for L1” (this will come later)
The L1 source language

• Similar to a subset of x86_64, but with some abstractions
  
  movq $1, %rax  
  rax <- 1

• L1 only has integer values and memory addresses
  (no floating point values)

• L1 has only
  compare, call, arithmetic, branch, and memory instructions
L1 label

label ::= sequence of chars matching :[a-zA-Z_] [a-zA-Z_0-9]*

:go ← This is a label

:3go ← This is not a label
L1 program

\[ p ::= (\text{label } f^+) \]
\[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ (:go \quad \text{The entry point of this L1 program is the function } :go \]
\[ \quad f1 \quad \text{One of these functions must be } :go \]
\[ f2 \]
\[ ) \]
L1 function

\[ p \ ::= \ (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f \ ::= \ (\text{label } N N \ i^+) \]

\[ N ::= (\text{+}|\text{-})? \ [1-9][0-9]^* \]

\(:=\text{go } 4 \ 2 \ i1 \ i2 \)
L1 instruction: return

\[ p ::= (\text{label } f^+) \]
\[ \text{label} ::= \text{sequence of chars matching: } [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (\text{label } N \ N \ i^+) \]
\[ i ::= \text{return} \]
L1 example

(:go
 (:go
  0 0
  return
 )
)

This is a complete and correct L1 program
L1 instruction: assignment

\[ p ::= (\text{label } f^+) \]
\[ \text{label ::= sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (\text{label } N N i^+) \]
\[ i ::= \ldots | w \leftarrow s \]

\[ w ::= a | \text{rax} | \text{rbx} | \text{rbp} | r10 | r11 | r12 | r13 | r14 | r15 \]
\[ a ::= \text{rdi} | \text{rsi} | \text{rdx} | \text{rcx} | r8 | r9 \]
\[ s ::= x | N | \text{label} \]
\[ x ::= w | \text{rsp} \]
The execution goes top->down, instruction after instruction.

Undefined behavior:
if the instruction at the bottom of the function is executed and the semantics is to execute the next one, then the behavior is undefined.
L1 example

(:go
 (:go
  0 0
  rdi <- 5
  rax <- rdi
 )
 )

• The execution goes top->down, instruction after instruction

• Undefined behavior:
  if the instruction at the bottom of the function is executed and the semantics is to execute the next one, then the behavior is undefined
L1 instruction: assignment

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching} : [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= ... | w \gets s \]

\[ w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15 \]

\[ a ::= rdi | rsi | rdx | rcx | r8 | r9 \]

\[ s ::= x | N | \text{label} \]

\[ x ::= w | rsp \]
L1 instruction: load

\[ p ::= (\text{label } f^+) \]

label ::= \textit{sequence of chars matching} : [a-zA-Z_] [a-zA-Z0-9]*

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= ... \mid w \leftarrow \text{mem } x \ M \]

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]

\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid \text{r8} \mid \text{r9} \]

\[ s ::= x \mid N \mid \text{label} \]

\[ x ::= w \mid \text{rsp} \]

\[ M ::= \textit{multiplicative of 8 constant (e.g., 0, 8, 16)} \]
L1 example

(:go
 (:go
  0 0
  rdi <- 5
  rbx <- mem rdi 8
  return
 )
)
)
L1 instruction: load

\[ p \ ::= (\text{label } f^+) \]

\textit{label} ::= \textit{sequence of chars matching} :\texttt{[a-zA-Z_][a-zA-Z_0-9]*}

\[ f \ ::= (\text{label } N N \ i^+) \]

\[ i \ ::= \ldots \mid w <\text{- mem} \ x \ M \]

\[ w \ ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid r10 \mid r11 \mid r12 \mid r13 \mid r14 \mid r15 \]

\[ a \ ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid r8 \mid r9 \]

\[ s \ ::= x \mid N \mid \text{label} \]

\[ x \ ::= w \mid \text{rsp} \]

\[ M \ ::= \text{multiplicative of 8 constant (e.g., 0, 8, 16)} \]
L1 instruction: store

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= ... \mid w \leftarrow \text{mem } x M \mid \text{mem } x M \leftarrow s \]

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid r10 \mid r11 \mid r12 \mid r13 \mid r14 \mid r15 \]

\[ a ::= rdi \mid rsi \mid rdx \mid rcx \mid r8 \mid r9 \]

\[ s ::= x \mid N \mid \text{label} \]

\[ x ::= w \mid \text{rsp} \]

\[ M ::= \text{multiplicative of 8 constant (e.g., } 0, 8, 16) \]
L1 instruction: arithmetic operations

\[\begin{align*}
\text{p} &::= (\text{label } f^+) \\
\text{label} &::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \\
\text{f} &::= (\text{label } N N i^+) \\
i &::= \ldots | w \text{ aop } t \\
\text{aop} &::= += | -= | *= | &= \\
t &::= x | N
\end{align*}\]
L1 example

(:go
  (:go
    0 0
    rdi <- 5
    rdi += 2
    return
  )
)
)
L1 instruction: arithmetic operations

\[ p ::= (\text{label } f^+) \]
\[ \text{label ::= sequence of chars matching :}[a-zA-Z_][a-zA-Z0-9]^* \]
\[ f ::= (\text{label } N N \text{ i}^+) \]
\[ i ::= \ldots | w \text{ aop } t \]
\[ \text{aop ::= } += | -= | *= | &= \]
\[ t ::= x | N \]

**Integer overflow is undefined behavior**
L1 instruction: shifting

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \text{ N } i^+) \]

\[ i ::= \ldots \mid w \text{ aop t } \mid w \text{ sop rcx} \]

\[ \text{sop} ::= \lll \mid \ggg \]
L1 instruction: shifting

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching } :[a-zA-Z_][a-zA-Z0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= \ldots \mid \text{w aop t} \mid \text{w sop rcx} \mid \text{w sop N} \]

\[ \text{sop} ::= \lll \mid \rrr \]
L1 instruction: memory arithmetic operations

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]* \]

\[ f ::= (\text{label N N i}^+) \]

\[ i ::= ... \]

\[ | \text{mem x M += t} \]
\[ | \text{mem x M -= t} \]
\[ | w += \text{mem x M} \]
\[ | w -= \text{mem x M} \]

Notice you cannot have both operands in memory
L1 instruction: comparison

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \ N \ i^+) \]

\[ i ::= ... \]

\[ | \ w \gets t \ \text{cmp} \ t \]

\[ \text{cmp ::= < | <= | =} \]

Notice there is neither > nor >=
L1 example

(:go
 (:go
  0 0
  rax <- 5
  rdi <- rax <= 3
  return
 )
)
)
L1 instruction: comparison

\[ p ::= (\text{label } f^+) \]

\[ \text{label} ::= \text{sequence of chars matching }: [a-zA-Z_][a-zA-Z0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= \ldots \]

\[ \mid w \leftarrow t \text{ cmp } t \]

\[ \text{cmp} ::= < | \leq | = \]
L1 instruction: conditional jump

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N N \ i^+) \]

\[ i ::= \ldots \]

\[ | \ w <- t \ \text{cmp } t \]

\[ | \ \text{cjump } t \ \text{cmp } t \ \text{label} \quad \text{Fall-through semantic} \]

\[ \text{cmp ::= } < | <= | = \]
L1 example

`:go
`:go
  0 0
  rax <- 5
  :true
  rdi <- rax <= 3
  cjump rdi = 1 :true
  return
)
L1 instruction: label and jump

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= \ldots \]

\[ \ldots \]

\[ = \]

\[ \begin{align*}
& w \triangleq t \text{ cmp } t \\
& \text{cj} \text{ump } t \text{ cmp } t \text{ label} \\
& \text{label} \\
& \text{goto } \text{ label}
\end{align*} \]
L1 example

(:go
  (:go
    0 0
    rax <- 5
    rax += 2
    cjump rax <= 3 :END
    rax += 4
    goto :END
  :END
  :END
  return
))
L1 instruction: label and jump

\[ p ::= (\text{label } f^+) \]
\[ \text{label ::= sequence of chars matching :}[a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (\text{label N N i}^+) \]
\[ i ::= ... \]
\[ \mid w \leftarrow t \text{ cmp } t \]
\[ \mid \text{cjum} p t \text{ cmp } t \text{ label} \]
\[ \mid \text{label} \]
\[ \mid \text{goto label} \]

The scope of labels is the program
L1 another example

(:F1
 (:F1
  0 0
  :L1
  return
 )
 (:F2
  0 0
  :L1
  return
 )
)
L1 instruction: call

\[ p ::= (\text{label } f^+) \]

\text{label} ::= \textit{sequence of chars matching} : [a-zA-Z_][a-zA-Z0-9]^*

\[ f ::= (\text{label } N N i^+) \]

\[ i ::= \ldots \]

\[ | \text{call } u \ N \]

Number of arguments of the called function (a.k.a. callee)

\[ u ::= w \mid \text{label} \]
L1 example

Why do we have redundant information in L1?
To simplify the L1 compiler (your work)

They must match

Number of parameters of the function

---

(:go
 (:go
   0 0
   call :myF2 0
   return
 )
 (:myF2
   0 0
   return
 )
)
L1 instruction: call

\[
\begin{align*}
p & := (\text{label } f^+) \\
\text{label} & := \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \\
f & := (\text{label } N \ N \ i^+) \\
i & := \ldots \\
| & \text{ call } u \ N \\
| & \text{ call } \text{print } 1
\end{align*}
\]
L1 example

(:go
 (:go
   4 2
   rdi <- 5
   call print 1
   return
 )
 )

The calling convention will be explained soon
L1 instruction: call

\[ p ::= (\text{label } f^+) \]

\[ \text{label ::= sequence of chars matching } :[a-zA-Z_][a-zA-Z_0-9]^* \]

\[ f ::= (\text{label } N \text{ } N \text{ } i^+) \]

\[ i ::= \ldots \]

\[ | \text{ call } u \text{ } N \]

\[ | \text{ call print } 1 \]

\[ | \text{ call input } 0 \]

\[ | \text{ call allocate } 2 \]

\[ | \text{ call tensor-error } F \]

\[ F ::= 1 \mid 3 \mid 4 \]
L1 instruction: misc

\[
p ::= (\text{label } f^+) \\
\text{label} ::= \text{sequence of chars matching }:[a-zA-Z_][a-zA-Z_0-9]^* \\
f ::= (\text{label } N N i^+) \\
i ::= \ldots \\
| \ w++ \\
| \ w-- \\
| \ w @ w w E \\
\]

\[
E ::= 1 | 2 | 4 | 8 \\
\]

\[
\text{rax} \ @ \ rdi \ rsi \ 4 \\
\text{Set rax to rdi} + (rsi \ * \ 4) \\
\]
\[ p ::= (\text{label } f^*) \]

\[ f ::= (\text{label } N N i^*) \]

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x M \mid \text{mem } x M \leftarrow s \mid \]
\[ w \text{ aop } t \mid w \text{ sop } sx \mid w \text{ sop } N \mid \text{mem } x M \leftarrow t \mid \text{mem } x M = t \mid w = \text{mem } x M \mid w = \text{mem } x M \mid \]
\[ w \leftarrow t \text{ cmp } t \mid \text{cjump } t \text{ cmp } t \text{ label} \mid \text{label} \mid \text{goto label} \mid \]
\[ \text{return} \mid \text{call } u N \mid \text{call print } 1 \mid \text{call input } 0 \mid \text{call allocate } 2 \mid \text{call tensor-error } F \mid \]
\[ w \leftarrow+ \mid w \leftarrow- \mid w \leftarrow w \leftarrow E \]

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]

\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{sx} \mid \text{r8} \mid \text{r9} \]

\[ sx ::= \text{rcx} \]

\[ s ::= t \mid \text{label} \]

\[ t ::= x \mid N \]

\[ u ::= w \mid \text{label} \]

\[ x ::= w \mid \text{rsp} \]

\[ \text{aop} ::= +\mid -\mid *\mid \&\mid = \]

\[ \text{sop} ::= \ll\mid \gg\]

\[ \text{cmp} ::= <\mid \le\mid = \]

\[ E ::= 1 \mid 2 \mid 4 \mid 8 \]

\[ F ::= 1 \mid 3 \mid 4 \]

\[ M ::= \text{multiplicative of 8 constant (e.g., 0, 8, 16)} \]

\[ N ::= (+|-)\? [1-9][0-9]* \]

\[ \text{label} ::= \text{sequence of chars matching }[:a-zA-Z_][a-zA-Z_0-9]* \]
Outline

• L1 language

• Value encoding

• Calling convention

• Heap
High level vs. low level languages

C language

```c
printf("5");
```

You expect the output

5

Back-end languages

```c
rdi <- 5
call print 1
```

You expect the output

?  

It depends on the encoding scheme designed for correctness
Value encoding in L1

• A value is either an 8 byte integer value or a memory address
• We would like to differentiate between the two
  • Safer programming environment
  • Problem: how to do it?
  • For example:
    mem rdi 8 <- rax
    is the value in rdi a memory address?
• This class solution: using the least significant bit to specify it
  0: it is a memory address
  1: it is an integer value
• Values in L1 are all encoded
High level vs. low level language L1

C language
printf("5");  You expect the output 5

L1 language
rdi <- 5  You expect the output 2
call print 1

00000101  00000010
Decoding an encoded value

• $x \& 1 = 0$
  $x$ is a memory address

• $x \& 1 = 1$
  $x >> 1$ is a 63 bit two’s complement integer

• Values (integer or addresses) must be encoded for runtime APIs
  • print
  • input (it returns the encoded value of the one read)
  • allocate
  • tensor-error
L1 example

(:go
 (:go
  0 0
  rdi <- 5
  call print 1
  return
 )
)

• print writes to the terminal the integer value encoded in rdi if rdi contains a number

• What is going to be the output?
  2
Outline

• L1 language

• Value encoding

• Calling convention

• Heap
Calling convention

• How many arguments a given function has?
  call :myF 2

• Where are the arguments stored?

• Who (caller vs. callee) is responsible for what?

• Where is the return value stored?
  rax
The stack in L1

- A call instruction that invokes F allocates new memory on top of the stack needed by F and to pass its arguments
- A return instruction executed in F frees that space
The stack in L1: function frame convention

So before calling a function, we need to store the return address on top of the stack.
Storing the return address

Two type of calls:

• Calls to L1 functions
  • L1 code is responsible
to store the return address on top of the stack

• Calls to runtime
  • L1 code is not responsible
to store the return address on top of the stack
Whoever generates L1 code (developer, compiler that targets L1) is responsible
  • to define the return label just after the call
  • to store that label on top of the stack
Function call example (2)

```
(:myF
  0 0
  call print 1
  return
)
```

- The call itself writes the return address on top of the stack
- There is no need to define the label after the call
What about function parameters?

• The convention used in the L1 language is that the first 6 parameters of the callee are passed using registers

• The other parameters are passed using the function frame of the callee stored on the stack
• What about the previous value of r10?
• We want to write our function without knowing the registers used/needed by every possible caller
  • Is it possible to know them all?
• Who is responsible to save the previous value?
  • Are we (the callee)?
  • Are the callers?
• We need to establish a convention
Registers

Arguments

- rdi
- rsi
- rdx
- rcx
- r8
- r9

Result

- rax

Caller save

- r10
- r11
- r8
- r9
- rax
- rcx
- rdi
- rdx
- rsi

Callee save

- r12
- r13
- r14
- r15
- rbp
- rbx

First argument
Caller save registers (e.g., r10)

(:myF
  0 1
  r10 <- 5
  mem rsp 0 <- r10
  mem rsp -8 <- :myF2_ret
  call :myF2 0
  :myF2_ret
  r10 <- mem rsp 0
  rdi <- r10
  call print 1
  return
)

(:myF2
  0 0
  r10 <- 3
  return
)

What is the output?

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers
Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers. R10 is not used after the call. Hence, we don’t need to save it.
Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers as well as callee-save registers.
Callee save registers (e.g., r12)

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers as well as callee-save registers.
The stack in L1

Stack space used to store values needed by the related function

Locals are used as function variables
The stack in L1

High address

Low address

Ret addr
Arg 7
Arg N
Locals

Bottom
Top
Stack frame: <= 6 arguments, no locals

(:go
...
 rdi <- 5
mem rsp -8 <- :f_ret
call :f 1
-> :f_ret
...
)
(:f
 1 0
-> return
)
Stack frame: > 6 arguments, no locals

(:go
  rdi<-1
  rsi<-3
  rdx<-5
  rcx<-7
  r8<-9
  r9<-11
  ...
  call :f 7
  ...
)

First argument
Stack frame: > 6 arguments, no locals

\[
\begin{align*}
\text{(:go} & \\
& \quad \text{//passing the first 6 arguments} \\
& \quad \text{mem rsp -8 } \leftarrow :f\_\text{ret} \\
& \quad \text{mem rsp -16 } \leftarrow 11 \\
& \quad \text{call :f 7} \\
& \text{)} \\
\text{(:f} & \\
& \quad 7 \ 0 \\
& \text{rdi } \leftarrow \text{mem rsp 0} \\
& \text{call print 1} \\
& \text{return})
\end{align*}
\]
Stack frame: <= 6 arguments, 1 local

(:go
   ...
   mem rsp -8 <- :f_ret
   call :f 1
   :f_ret
      ...
   )
)
(:f
   11
   return
   )
L1 program example

(:go
  (:go
    0 0
    rdi <- 5
    rsi <- 3
    call :myF 2
    return
  )
)

(:myF
  2 0
  call print 1
  rdi <- rsi
  call print 1
  return
)

What is the output?

Is there a bug? Where?

mem rsp -8 <- :f_ret

2
1
Stack frame: > 6 arguments, > 0 locals

`:go

... //passing the first 6 arguments
mem rsp -8 <- :f_ret
mem rsp -16 <- 11
call :f 7

:f_ret

):

(:f

7 1

→ rdi <- mem rsp 0
call print 1 What does it print?
return)
Stack pointer

• \texttt{rsp} (the stack pointer) is never modified directly

• \texttt{call} and \texttt{return} instructions modify \texttt{rsp} to do their jobs
  (see the grammar)
Outline

• L1 language

• Value encoding

• Calling convention

• Heap
Heap memory in L1

• Arrays are allocated in the heap

• No explicit deallocation
  • A garbage collector is assumed

• APIs
  • allocate:
    allocate an array of a given number of 64-bit integer elements
  • tensor-error:
    write to stdout an error message and abort the execution
Heap memory in L1

• allocate
  • Argument 1: number of array elements to allocate
  • Argument 2: 64-bit integer value used to initialize all array elements
  • Return: pointer to the array allocated and initialized

Not encoded
Example of L1 program using heap memory

(:go
  (:go
    0 0
    rdi <- 5
    rsi <- 7
    call allocate 2
    return
  )
)
Example of L1 program using heap memory

(:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  rdi <- mem rax 8
  call print 1
  return
)

What is the output?
3

rax + 8

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Example of L1 program using heap memory

(:go
0 0
rdi <- 5
rsi <- 7
call allocate 2
rdi <- mem rax 16
call print 1
return
)

What is the output?

3

rax + 16

2
7
7
Example of L1 program using heap memory

(:go
  0 0
  rdi <- 5
  rsi <- 7
  call allocate 2
  rdi <- mem rax 0
  call print 1
  return
)

What is the output?
Segmentation fault

How can we fix this L1 program?
Printing an array

• The API `print` writes to stdout the whole array if its pointer is passed as argument.

```assembly
rdi <- 5
rsi <- 7
call allocate 2
rdi <- rax
call print 1
{s:2, 3, 3}
```
Tensors: array of arrays

```go
(:go 0 0
 rdi <- 5
 rsi <- 7
 call allocate 2
 rdi <- 7
 rsi <- rax
 call allocate 2
 rdi <- rax
 call print 1
 return
)
```

Allocate an array of 2 integer values
Allocate an array of 3 pointers and initialize them to point to the previously allocated array

The output: `{s:3, {s:2, 3, 3}, {s:2, 3, 3}, {s:2, 3, 3}}`
Error messaging in L1

tensor-error 1

• Goal: report to the program’s developer a tensor access error and abort the execution

• Type of error: a heap object has been accessed without allocating it first

• Arguments:
  • First: line number of the program’s file where the tensor access error has occurred at run-time
Example of L1 code that uses tensor-error

(:myF 1 0
  cjump rdi = 0 :ERROR
  call print 1
  :ERROR
  rdi <- 5
  call tensor-error 1
  No need for a return instruction
  )

If this instruction executes, then no other instructions will execute
tensor-error 3

• Goal: report to the program’s developer a tensor access error and abort the execution

• Type of error: out-of-bound array access

• Arguments:
  • First: line number of the program’s file where the access error has occurred at run-time
  • Second: length of the array that has been accessed incorrectly
  • Third: index of the array used to access the array incorrectly
Error messaging in L1

tensor-error 4
• Goal: report to the program’s developer a tensor access error and abort the execution
• Type of error: out-of-bound tensor access
• Arguments:
  • First: line number of the program’s file where the access error has occurred at run-time
  • Second: dimension of the out-of-bound tensor access
  • Third: length of the dimension of the tensor accessed incorrectly
  • Forth: index used in the dimension that has generated the run-time error
Final notes

• The calling convention must be ALWAYS preserved

• An L1 program with undefined behavior is an incorrect L1 program

• You can write comments in L1
  • A comment starts with “//” and it comments until the end of the line
  • Example
    // This is a comment
    rdi <- 5

    // this is another comment
Tests

• Write 3 non trivial (more than 30 instructions) L1 programs
  • The name of the files must end with .L1
  • For example: myTest1.L1

• Deadline: 2 days from today at midnight

• Tests and pairs (only for L1 tests)
  • Anyone in a pair can submit the 3 tests
  • Tests are considered to be of both ones of the related pair