Simone Campanoni
simone.campanoni@northwestern.edu
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
Compilers

Character stream (Source code)

Front-end

IR

Middle-end

IR

Back-end

0101010101 (Machine code)
Compilers

Character stream
(Source code)

Front-end

Middle-end

Back-end

0101010101
(Machine code)
Compilers

Character stream
(Source code)

Front-end

IR

Middle-end

IR

Back-end

0101010101
(Machine code)
Compilers

Machine code generation, Assembler, Linker

Instruction selection

Register allocation

Data and control linearization

IRc

L3c

L2c

L1c

IR

L3

L2

L1

0101010101 (Machine code)
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
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
Correct programs that can be written using a language are specified using a grammar and some formal specification for its semantics.

A program is a sequence of characters. A grammar specifies the set of sequences of characters that are allowed.

Let’s have a quick introduction to a trivial grammar and then we’ll look at the L1 grammar.
Trivial example of a grammar

Let’s assume we want a grammar that allows only the next sequences of characters:

• a
• 0
• 10
• 20
• 30

This is the definition of a program

This is a rule of the grammar

p is defined as

p
Trivial example of a grammar

Let’s assume we want a grammar that allows only the next sequences of characters:

• a
• 0
• 10
• 20
• 30

The exact character “a”

or another symbol that will require another rule for its definition

The exact character “0”

Any number from 1 included to 3 included
• a
• 0
• 10
• 20
• 30
• c
• +c

\[ p ::= a \mid b \]

\[ b ::= 0 \mid [1-3]0 \mid +? c \]
\begin{itemize}
\item a
\item 0
\item 10
\item 20
\item 30
\item c
\item +c
\item -c
\end{itemize}

\begin{verbatim}
p ::= a \mid b
b ::= 0 | [1-3]0 \mid (+|-)? c
\end{verbatim}
\[p ::= a \mid b\]
\[b ::= 0 \mid [1-3]0 \mid (+-)? c \mid ([a-z] | [A-Z])\]
\text{p ::= a | b}

\text{b ::= 0 | [1-3]0 | (+|-)? c | [a-zA-Z]}

• a
• 0
• 10
• 20
• 30
• c
• +c
• -c
• Z
• ...
• :Aaaaaabbdfsdgfdssdfdsgfs
• :A
• :ZFRDFGDFdfsfsdfsdf

\[p ::= a \mid b\]
\[b ::= 0 \mid [1-3]0 \mid (+-)? c \mid :[a-zA-Z]^+\]
• :Aaaaabbdfsdgfdsdfdsgfs
• :A
• :ZFRDFGDFdfsdfsdfsdf
• :
  
  p ::= a | b
  
  b ::= 0 | [1-3]0 | (+-) c | [:a-zA-Z]*
Now we are ready to look at the L1 grammar
L1 name

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

go  This is a name

3go  This is not a name
L1 label

label ::= :name
name ::= 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

\[ p ::= (l f^+) \]
\[ l ::= @name \]
\[ name ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z0-9]^* \]

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

(}
L1 function

\[ p ::= (l \ f^+) \]
\[ l ::= @\text{name} \]
\[ \text{name} ::= \text{sequence of chars matching } [a-zA-Z_][a-zA-Z_0-9]^* \]
\[ f ::= (l \ N \ N \ i^+) \]
\[ N ::= (+|-)?[1-9][0-9]^* | 0 \]

(\@\text{go} 4 2
  i1 \quad \text{We now need to look at}
  \text{the possible instructions that we can include}
  \text{in an L1 function}
  i2 )
L1 instruction: return

\[ f ::= (\text{I} \text{N} \text{N} \ i^+) \]
\[ i ::= \text{return} \]
This is a complete and correct L1 program
L1 instruction: assignment

\[ f ::= (l N N i^+) \]
\[ i ::= \ldots \mathbin{|} w \Leftarrow s \]

\[ w ::= a \mathbin{|} rax \mathbin{|} rbx \mathbin{|} rbp \mathbin{|} r10 \mathbin{|} r11 \mathbin{|} r12 \mathbin{|} r13 \mathbin{|} r14 \mathbin{|} r15 \]
\[ a ::= rdi \mathbin{|} rsi \mathbin{|} rdx \mathbin{|} rcx \mathbin{|} r8 \mathbin{|} r9 \]
\[ s ::= x \mathbin{|} N \mathbin{|} label \]
\[ x ::= w \mathbin{|} rsp \]
L1 example

(@go
 (@go
  0 0
  rdi <- 5
  rax <- rdi
  return
 )
 )

• 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

\[ f ::= (l \ N \ N \ i^+) \]
\[ i ::= \ldots \mid w \leftarrow s \]

\textit{When s is a label, then it must be an existing function name}

\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]
\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid \text{r8} \mid \text{r9} \]
\[ s ::= x \mid \text{N} \mid \text{label} \]
\[ x ::= w \mid \text{rsp} \]
L1 instruction: load

\[ f ::= (l \ N \ N \ i^+) \]
\[ i ::= \ldots \mid w \gets \text{mem \ x \ M} \]
\[ w ::= a \mid \text{rax} \mid \text{rbx} \mid \text{rbp} \mid \text{r10} \mid \text{r11} \mid \text{r12} \mid \text{r13} \mid \text{r14} \mid \text{r15} \]
\[ a ::= \text{rdi} \mid \text{rsi} \mid \text{rdx} \mid \text{rcx} \mid \text{r8} \mid \text{r9} \]
\[ s ::= x \mid \text{N} \mid \text{label} \]
\[ x ::= w \mid \text{rsp} \]
\[ M ::= \text{multiplicative of 8 constant (e.g., 0, 8, 16)} \]
L1 example

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

\[ f ::= (l \ N \ N \ i^+) \]
\[ i ::= \ldots \ | \ w <- \text{mem} \ x \ M \]

\[ w ::= a \ | \ rax \ | \ rbx \ | \ rbp \ | \ r10 \ | \ r11 \ | \ r12 \ | \ r13 \ | \ r14 \ | \ r15 \]
\[ a ::= \text{rdi} \ | \ rsi \ | \ rdx \ | \ rcx \ | \ r8 \ | \ r9 \]
\[ s ::= x \ | \ N \ | \ \text{label} \]
\[ x ::= w \ | \ \text{rsp} \]
\[ M ::= \text{multiplicative of 8 constant (e.g., 0, 8, 16)} \]
L1 instruction: store

\[ f ::= (l \ N \ N \ i^+) \]

\[ i ::= \ldots \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \]

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

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

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

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

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

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

\[ i ::= \ldots | w \text{ } \text{aop} \text{ } t \]

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

\[ t ::= x | N \]
L1 example

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

\[ f ::= (l \mathord N N i^+) \]

\[ i ::= \ldots \mid w \ aop \ t \]

\[ aop ::= +\mid -\mid \ast\mid \&= \]

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

*Integer overflow is undefined behavior*
L1 instruction: shifting

\[ f ::= (l\ N\ N\ i) \]

\[ i ::= \ldots\ |\ w\ aop\ t\ |\ w\ sop\ rcx \]

\[ sop ::= \ll\ |\ \gg\gg \]

\[ rdi \ll rcx \]
L1 instruction: shifting

\[ f ::= (l \; N \; N \; i^+) \]
\[ i ::= \ldots \mid w \; aop \; t \mid w \; sop \; rcx \mid w \; sop \; N \]
\[ sop ::= \lll \mid \ggg \]

\[ \text{rdi} \lll \text{rcx} \]
\[ \text{rdi} \lll 3 \]
L1 instruction: memory arithmetic operations

\[
f ::= (l \ N \ N \ i^+) \\\n
i ::= ... \\\n    | mem x M += t \\\n    | mem x M -= t \\\n    | w += mem x M \\\n    | w -= mem x M
\]

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

\[ f ::= (l \ N \ N i^+) \]
\[ i ::= … \]
\[ \mid w \leftarrow t \ cmp \ t \]

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

Notice there is neither 
\[ > \]
\[ \text{nor} \]
\[ >= \]
L1 example

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

\[ f ::= (| N N i^+) \]

\[ i ::= \ldots \]
\[ w \leftarrow t \text{ cmp } t \]

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

\[ f ::= (l \, N \, N \, i^+) \]

\[ i ::= \ldots \]

\[ \mid w <- t \, \text{cmp} \, t \]

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

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

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

\[ f ::= (l \ N \ N \ i^+) \]

\[ i ::= \ldots \]

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

\[ | \ \text{cjump} \ t \ \text{cmp} \ t \ \text{label} \]

\[ | \ \text{label} \]

\[ | \ \text{goto \ label} \]
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

\[ f ::= (l \ N \ N \ i^+) \]
\[ i ::= \ldots \]
  \[ | \ \text{w} \leftarrow \text{t} \ \text{cmp} \ \text{t} \]
  \[ | \ \text{cjump} \ \text{t} \ \text{cmp} \ \text{t} \ \text{label} \]
  \[ | \ \text{label} \]
  \[ | \ \text{goto} \ \text{label} \]

The scope of labels is the program
L1 another example

(@F1
 (@F1
   0 0
   :L1
   return
 )
 (@F2
   0 0
   :L1
   return
 )
)
L1 instruction: label and jump

\[ f ::= (l \ N \ N \ i^+) \]

\[ i ::= ... \]

\[ | \ w <- t \ cmp \ t \]

\[ | \ cjump \ t \ cmp \ t \ label \]

\[ | \ label \]

\[ | \ goto \ label \]

The scope of labels is the program but you cannot jump to another function using \texttt{cjump} or \texttt{goto}
L1 another example

(@F1
 (@F1
  0 0
  goto :L1
  return
 )
 (@F2
  0 0
  :L1
  return
 )
 )

This is an incorrect L1 program
L1 instruction: call

\[ f ::= (l \ N \ N \ i^+) \]
\[ i ::= \ldots \]
\[ | \text{call } u \ N \]

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

\[ u ::= w \mid l \]

Name of a function

Register that holds the reference (name) of the function to call
L1 example

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

Number of parameters of the function

They must match
L1 instruction: call

\[ f ::= (l\ N\ N\ i^+) \]

\[ i ::= \ldots \]

  | call u N

  | call print 1
L1 example

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

The calling convention will be explained soon
L1 instruction: call

\[ f ::= (l \, N \, N \, i^+) \]

\[ i ::= \ldots \]

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

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

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

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

\[ \quad | \text{call } \text{tuple-error } 3 \]

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

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

\[ f ::= (l N N i^+) \]

\[ i ::= \ldots \]
  \[ w++ \]
  \[ w-- \]
  \[ w @ w w E \]

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

\[ \text{Set rax to rdi + (rsi} \ast 4) \]
\[ p ::= (I f^*) \]
\[ f ::= (I N N i^*) \]
\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x M \mid \text{mem } x M \leftarrow s \mid w \text{ aop } t \mid w \text{ sop } sx \mid w \text{ sop } N \mid \text{mem } x M := t \mid \text{mem } x M := t \mid w := \text{mem } x M \mid w := \text{mem } x M \mid w \leftarrow t \text{ cmp } t \mid \text{cjump } t \text{ cmp } t \text{ label } \mid \text{label } \mid \text{goto } \text{ label } \mid \text{return } \mid \text{call } u N \mid \text{call } \text{print } 1 \mid \text{call } \text{input } 0 \mid \text{call } \text{allocate } 2 \mid \text{call } \text{tuple-error } 3 \mid \text{call } \text{tensor-error } F \mid w ++ \mid w -- \mid w @ w w E \]
\[ w ::= a \mid \text{rax } \mid \text{rbx } \mid \text{rbp } \mid r10 \mid r11 \mid r12 \mid r13 \mid r14 \mid r15 \]
\[ a ::= \text{rdi } \mid \text{rsi } \mid \text{rdx } \mid sx \mid r8 \mid r9 \]
\[ sx ::= \text{rcx} \]
\[ s ::= t \mid \text{label } \mid l \]
\[ t ::= x \mid N \]
\[ u ::= w \mid l \]
\[ x ::= w \mid \text{rsp} \]
\[ \text{aop} ::= += \mid -= \mid **= \mid &= \]
\[ \text{sop} ::= <<= \mid >>= \]
\[ \text{cmp} ::= < \mid <= \mid = \]
\[ E ::= 1 \mid 2 \mid 4 \mid 8 \]
\[ F ::= 1 \mid 3 \mid 4 \]
\[ M ::= \text{multiplicative of 8 constant (e.g., 0, 8, 16)} \]
\[ N ::= (+|-)\? [1-9][0-9]* \mid 0 \]
\[ l ::= @\text{name} \]
\[ \text{label} ::= :\text{name} \]
\[ \text{name} ::= \text{sequence of chars matching } [a-zA-Z][a-zA-Z0-9]* \]
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

```assembly
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
call print 1
```

You expect the output

2

00000101

00000010
Decoding an encoded value

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

• $x \& 1 = 1$
  $x \gg 1$ 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
  • tuple-error and 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
  • The rest of the calling convention is the same with calls to L1 functions
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
Registers

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

- Result
  - rax

First argument

```plaintext
(@go
  0 1
  r10 <- 5
  ...
)
```

- 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)

Whoever generates L1 code (developer, compiler that targets L1) is responsible to properly save caller-save registers.
Caller save registers (e.g., r10)

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.
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.
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
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
  ...
)

Arguments
  rdi
  rsi
  rdx
  rcx
  r8
  r9

First argument
Bottom

rsp
Stack frame: > 6 arguments, no locals

(@go
  ... //passing the first 6 arguments
  mem rsp -8 <- :f_ret
  mem rsp -16 <- 11
  call @f 7
    :f_ret
  )
(@f
  7 0
  rdi <- mem rsp 0
  call print 1
  return)
Stack frame: <= 6 arguments, 1 local

(@go
  ...
  mem rsp -8 <- :f_ret
  call @f 1
  :f_ret
  ...
)
(@f
  11
  return
)
Stack frame: <= 6 arguments, 2 locals

(@go
  ...
  mem rsp -8 <- :f_ret
  call @f 1
    :f_ret
      ...
  )
  (@f
    12
    return
  )
L1 program example

```
(@go
 (@go
  0 0
  rdi <- 5
  rsi <- 3
  mem rsp -8 <- :f_ret
  call @myF 2
  :f_ret
  return
 )
)
```

Is there a bug? Where?

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

What is the output?

```
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)

Bottom

Ret addr
1011
?????
Stack pointer

• \texttt{rsp} (the stack pointer) is never modified directly by L1 code

• \texttt{call} and \texttt{return} instructions implicitly 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 and tuple-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
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

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>7</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

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

```plaintext
(@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 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)  
If this instruction executes, then no other instructions will execute
No need for a return instruction
Error messaging in L1

tensor-error 3

• Goal: report to the program’s developer an array 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
Error messaging in L1

tuple-error 3

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

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

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

• Every line of an L1 program is a comment, an instruction, or an empty line
  rdi <- 5 // this is incorrect
Tests

• Write an L1 program that takes as input a sequence of numbers and print them in descending order (an example of an input file is available on canvas)
  • Example of input file:
    - Number of elements to sort
    - Elements to sort
    - Your L1 code
    - 4 2 2
  - The name of the L1 program file must end with .L1
  - For example: myTest1.L1
• Deadline: 2 days from today (see Canvas for the exact deadline)
• Tests and pairs
  • Submit one L1 program per pair
Always have faith in your ability

Success will come your way eventually

Best of luck!