Instruction Sets and Code Generation

Eelco Visser

TU Delft

CS4200 | Compiler Construction | November 26, 2020

This Lecture

Operational Semantics

– of ChocoPy

Machine Architecture

– components of a (virtual) machine

RISC-V Instruction Set

– instructions, registers, conventions

Code Generation by Term Transformation

– from source AST to target AST

Compilation Schemas

– how do source language constructs map to

Operational Semantics

Operational Semantics

6 Operational semantics

This section contains the formal operational semantics for the ChocoPy language.

The operational semantics define how every definition, statement, or expression in a ChocoPy program should be evaluated in a given context.

Literals

$$\frac{}{G, E, S \vdash \texttt{None} : \mathit{None}, S, \_} \quad \text{[NONE]}$$

$$\frac{}{G, E, S \vdash \texttt{False} : \mathit{bool}(\mathit{false}), S, \_} \quad \text{[BOOL-FALSE]}$$

$$\frac{}{G, E, S \vdash \texttt{True} : \mathit{bool}(\mathit{true}), S, \_} \quad \text{[BOOL-TRUE]}$$

$$\frac{i \text{ is an integer literal}}{G, E, S \vdash i : \mathit{int}(i), S, \_} \quad \text{[INT]}$$

$$\frac{\begin{array}{l} s \text{ is a string literal} \\ n \text{ is the length of the string } s \end{array}}{G, E, S \vdash s : \mathit{str}(n, s), S, \_} \quad \text{[STR]}$$

Expression Statement

$$\frac{G, E, S \vdash e : v, S', \_}{G, E, S \vdash e : \_, S', \_} \quad \text{[EXPR-STMT]}$$

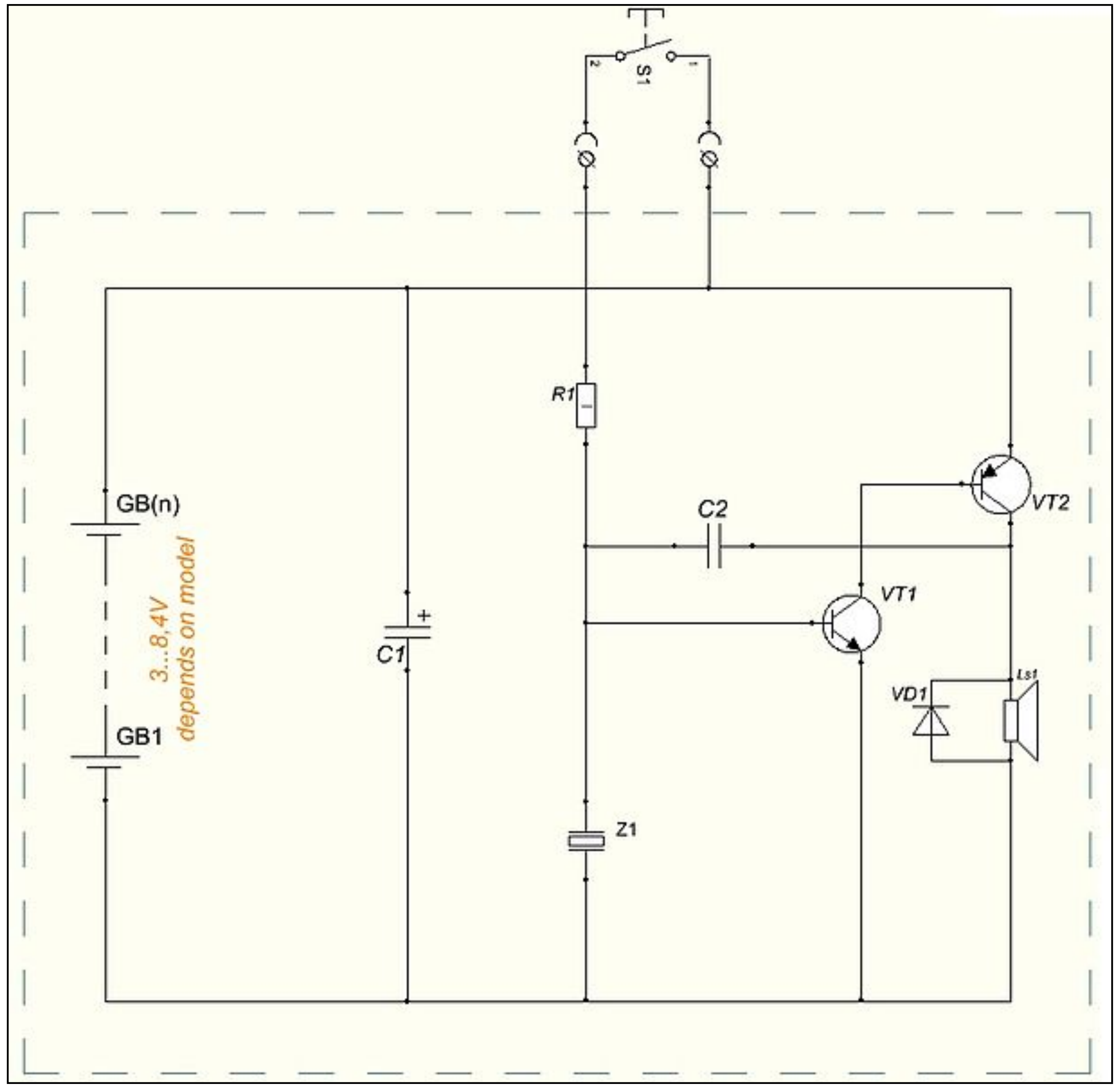
Arithmetic Expressions

$$\frac{\begin{array}{l} G, E, S \vdash e : int(i_1), S_1, \_ \\ v = int(-i_1) \end{array}}{G, E, S \vdash \texttt{-}\ e : v, S_1, \_} \quad [\text{NEGATE}]$$

$$\frac{\begin{array}{l} G, E, S \vdash e_1 : int(i_1), S_1, \_ \\ G, E, S_1 \vdash e_2 : int(i_2), S_2, \_ \\ op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{//}, \texttt{\%}\} \\ op \in \{\texttt{//}, \texttt{\%}\} \Rightarrow i_2 \neq 0 \\ v = int(i_1\ op\ i_2) \end{array}}{G, E, S \vdash e_1\ op\ e_2 : v, S_2, \_} \quad [\text{ARITH}]$$

Machine Architecture

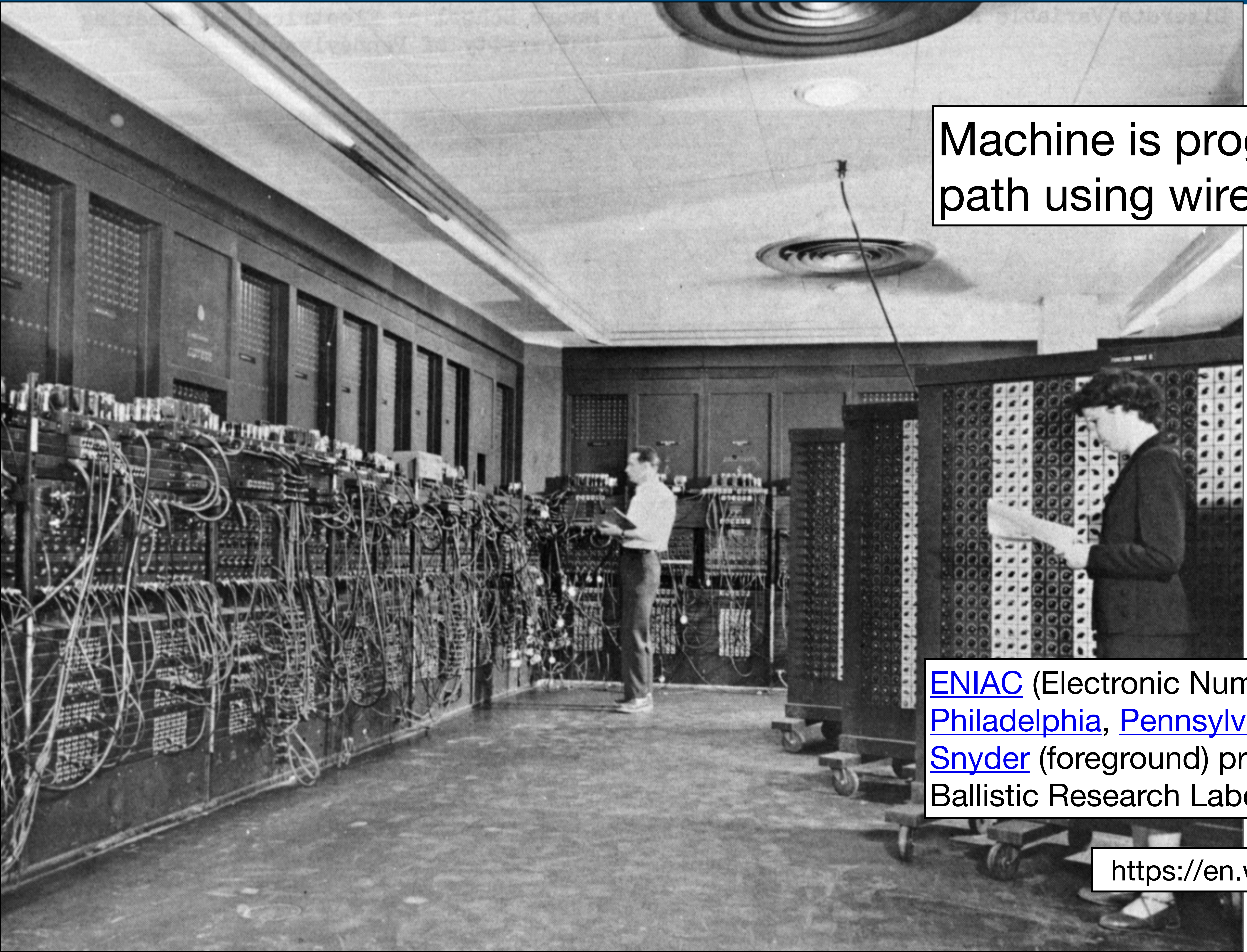
Hard-Wired Programs



Fixed to perform one computation from input to output

https://commons.wikimedia.org/wiki/File:Wiring_diagram_of_battery-powered_doorbell.JPG

Programmable Machines



Machine is programmed by creating data path using wires

ENIAC (Electronic Numerical Integrator And Computer) in Philadelphia, Pennsylvania. Glen Beck (background) and Betty Snyder (foreground) program the ENIAC in building 328 at the Ballistic Research Laboratory (BRL).

https://en.wikipedia.org/wiki/ENIAC#/media/File:Eniac.jpg

Stored-Program Computer (Von Neumann Architecture)

Central Processing Unit

- Processor registers
- Arithmetic logic unit

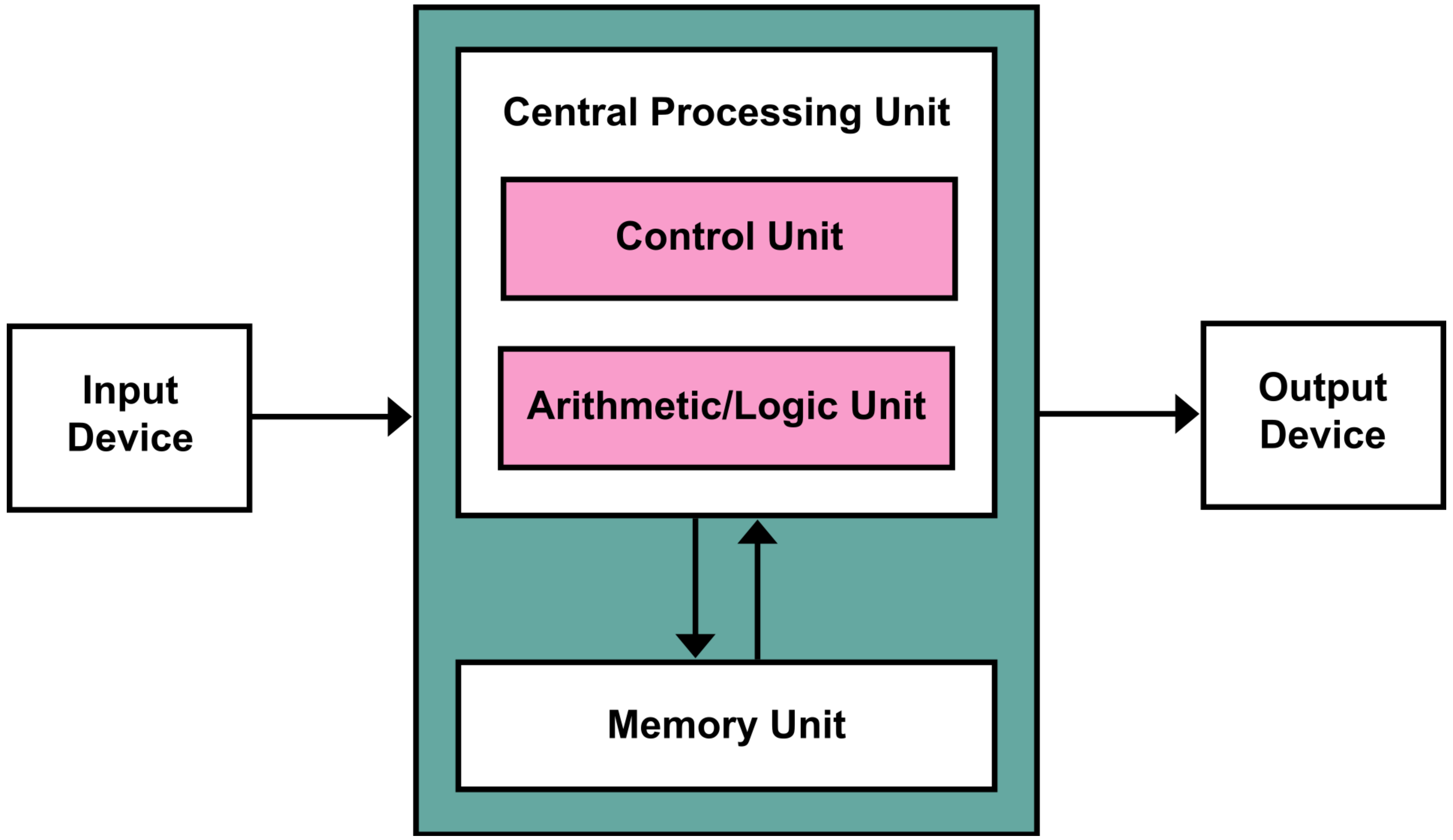
Main Memory

- Stores data and instructions

External Storage

- Persistent storage of data

Input/Output



https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg

State

Machine state

- data stored in memory
- memory hierarchy: registers, RAM, disk, network, …

Imperative program

- computation is series of changes to memory
- basic operations on memory (increment register)
- controlling such operations (jump, return address, …)
- control represented by state (program counter, stack, …)

Example: x86 Assembler

`mov AX [1]`	read memory
`mov CX AX`	
`L: dec CX`	
`mul CX`	calculation
`cmp CX 1`	
`ja L`	jump
`mov [2] AX`	write memory

Example: Java Bytecode

```
.method static public m(I)I

        iload 1
        ifne else                jump
        iconst_1
        ireturn

else:   iload 1                  read memory
        dup
        iconst_1
        isub                     calculation
        invokestatic Math/m(I)I
        imul
        ireturn
```

Memory & Control Abstractions

Memory abstractions

- variables: abstract over data storage
- expressions: combine data into new data
- assignment: abstract over storage operations

Control-flow abstractions

- structured control-flow: abstract over unstructured jumps
- ‘go to statement considered harmful’ Edgser Dijkstra, 1968

Example: C

```
int f = 1
int x = 5
int s = f + x

while (x > 1) {
   f = x * f ;
   x = x - 1
}
```

variable

expression

control flow

assignment

Procedural Abstraction

Control-flow abstraction

- Procedure: named unit of computation
- Procedure call: jump to unit of computation and return

Memory abstraction

- Formal parameter: the name of the parameter
- Actual parameter: value that is passed to procedure
- Local variable: temporary memory

Recursion

- Procedure may (indirectly) call itself
- Consequence?

RISC-V Instruction Set

Concrete Syntax

```
.globl main
main:
  lui a0, 8192                      # Initialize heap size (in multiples of 4KB)
  add s11, s11, a0                  # Save heap size
  jal heap.init                     # Call heap.init routine
  mv gp, a0                         # Initialize heap pointer
  mv s10, gp                        # Set beginning of heap
  add s11, s10, s11                 # Set end of heap (= start of heap + heap size)
  mv ra, zero                       # No normal return from main program.
  mv fp, zero                       # No preceding frame.
  mv fp, zero                       # Top saved FP is 0.
  mv ra, zero                       # No function return from top level.
  addi sp, sp, -@..main.size        # Reserve space for stack frame.
  sw ra, @..main.size-4(sp)         # return address
  sw fp, @..main.size-8(sp)         # control link
  addi fp, sp, @..main.size         # New fp is at old SP.
  jal initchars                     # Initialize one-character strings.
  li a0, 1                          # Load boolean literal: true
  beqz a0, label_1                  # Operator and: short-circuit left operand
  li a0, 0                          # Load boolean literal: false
  seqz a0, a0                       # Logical not
label_1:                            # Done evaluating operator: and
  .equiv @..main.size, 16
label_0:                            # End of program
  li a0, 10                         # Code for ecall: exit
  ecall
```

Syntax Definition

```
// RV32I - Base
// Math
Instruction.Add  = <add  <ID>, <ID>, <ID>>      {case-insensitive}
Instruction.Addi = <addi <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.And  = <and  <ID>, <ID>, <ID>>      {case-insensitive}
Instruction.Andi = <andi <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Or   = <or   <ID>, <ID>, <ID>>      {case-insensitive}
Instruction.Ori  = <ori  <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Xor  = <xor  <ID>, <ID>, <ID>>      {case-insensitive}
Instruction.Xori = <xori <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Sub  = <sub  <ID>, <ID>, <ID>>      {case-insensitive}

// Branches
Instruction.Beq  = <beq <ID>, <ID>, <IntOrID>>  {case-insensitive}
Instruction.Bne  = <bne <ID>, <ID>, <IntOrID>>  {case-insensitive}
Instruction.Blt  = <blt <ID>, <ID>, <IntOrID>>  {case-insensitive}
Instruction.Bge  = <bge <ID>, <ID>, <IntOrID>>  {case-insensitive}
Instruction.Bltu = <bltu <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Bgeu = <bgeu <ID>, <ID>, <IntOrID>> {case-insensitive}

// Misc.
Instruction.Ecall = <ecall>
Instruction.Lui   = <lui   <ID>, <IntOrID>> {case-insensitive}
Instruction.Auipc = <auipc <ID>, <IntOrID>> {case-insensitive}

// Jumps
Instruction.Jal  = <jal  <ID>, <IntOrID>>       {case-insensitive}
Instruction.Jalr = <jalr <ID>, <ID>, <IntOrID>> {case-insensitive}
```

RV32IM.sdf3

Abstract Syntax Signature

```
module signatures/RV32IM-sig

imports signatures/Common-sig

signature
  sorts Start Line Label Statement Pseudodirective Instruction IntOrID
  constructors
    Program                 : List(Line) → Start
                            : Statement → Line
                            : Label → Line
    Label                   : ID → Label
                            : INT → IntOrID
                            : ID → IntOrID
                            : Pseudodirective → Statement
                            : Instruction → Statement
    PSData                  : Pseudodirective
    PSText                  : Pseudodirective
    PSString                : STRING → Pseudodirective
    PSAsciiz                : STRING → Pseudodirective
    PSWord                  : List(IntOrID) → Pseudodirective
    PSSpace                 : INT → Pseudodirective
    …
    Add                     : ID * ID * ID → Instruction
    Addi                    : ID * ID * IntOrID → Instruction
    And                     : ID * ID * ID → Instruction
    Andi                    : ID * ID * IntOrID → Instruction
    Or                      : ID * ID * ID → Instruction
    Ori                     : ID * ID * IntOrID → Instruction
    …
```

RV32IM-sig.str

RISC-V Assembly Programmer's Manual

Load Immediate

The following example shows the `li` pseudo instruction which is used to load immediate values:

```
        .equ    CONSTANT, 0xdeadbeef

        li      a0, CONSTANT
```

Which, for RV32I, generates the following assembler output, as seen by `objdump`:

```
00000000 <.text>:
   0:   deadc537                lui     a0,0xdeadc
   4:   eef50513                addi    a0,a0,-273 # deadbeef <CONSTANT+0x0>
```

https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md

RISC-V Assembly Programmer's Manual

Load Address

The following example shows the `la` pseudo instruction which is used to load symbol addresses:

```
        la      a0, msg + 1
```

Which generates the following assembler output and relocations for non-PIC as seen by `objdump`:

```
0000000000000000 <.text>:
   0:   00000517                auipc   a0,0x0
                        0: R_RISCV_PCREL_HI20   msg+0x1
   4:   00050513                mv      a0,a0
                        4: R_RISCV_PCREL_LO12_I .L0
```

And generates the following assembler output and relocations for PIC as seen by `objdump`:

```
0000000000000000 <.text>:
   0:   00000517                auipc   a0,0x0
                        0: R_RISCV_GOT_HI20     msg+0x1
   4:   00053503                ld      a0,0(a0) # 0 <.text>
                        4: R_RISCV_PCREL_LO12_I .L0
```

https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md

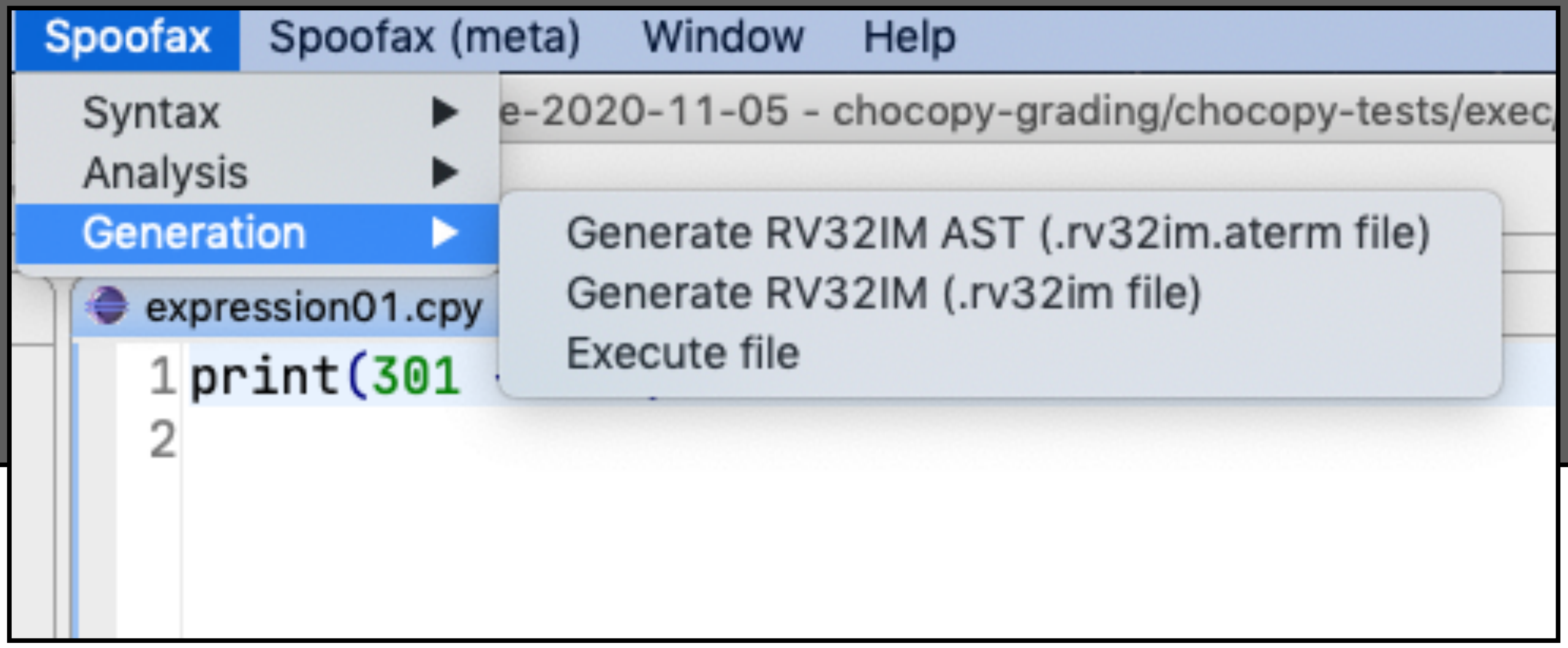
From Concrete Syntax to Abstract Syntax

```
.text

li a0, 1
li a1, 15
ecall

li a0, 10
ecall
```

```
Program(
  [ PSText()
  , Li("a0", "1")
  , Li("a1", "15")
  , Ecall()
  , Li("a0", "10")
  , Ecall()
  ]
)
```

Code Generation by Term Transformation

Compilation Menu




```
module Generation

menus
  menu: "Generation" (openeditor)
    action: "Generate RV32IM AST (.rv32im.aterm file)" = editor-generate-rv32im-ast
    action: "Generate RV32IM (.rv32im file)"           = editor-generate-rv32im
    action: "Execute file"                             = editor-execute

language
  provider : lib/venus164-0.2.5.jar
  provider : lib/kotlin-stdlib-1.4.10.jar
```

Invoking the Compiler

```
strategies

  editor-generate-rv32im-ast:
    (_, _, ast, path, _) → (filename, result)
    with
      filename := <guarantee-extension(|$[rv32im.aterm])> path;
      result   := <chocopy-to-rv32im> ast

  editor-generate-rv32im:
    (_, _, ast, path, _) → (filename, result)
    with
      filename := <guarantee-extension(|$[rv32im])> path;
      result   := <chocopy-to-rv32im; pp-RV32IM-string> ast

  editor-execute:
    (_, _, ast, path, _) → (filename, result)
    with
      filename := <guarantee-extension(|$[result.txt])> path;
      result   := <execute-program> ast
```

The Compiler Pipeline

```
strategies

  execute-program =
    chocopy-to-rv32im
    ; pp-RV32IM-string
    ; execute-riscv
    ; process-output

  chocopy-to-rv32im =
    program-to-rv32im

rules

  program-to-rv32im:
    ast@Program(definitions, statements) → Program(list)
    where list := […] // your code here
```

Translating Additions

Example: Right-Associative Addition

```
301 + (202 + (234 + (53 + (2342 + 51))))
```

```
li r, i // load immediate
```

```
add r1, r2, r3 // r1 := r2 + r23
```

```
Program(
  [ PSText()
  , Li("a1", "301")
  , Li("t0", "202")
  , Li("t1", "234")
  , Li("t2", "53")
  , Li("t3", "2342")
  , Addi("t3", "t3", "51")
  , Add("t2", "t2", "t3")
  , Add("t1", "t1", "t2")
  , Add("t0", "t0", "t1")
  , Add("a1", "a1", "t0")
  , Li("a0", "1")
  , Ecall()
  ]
)
```

```
.text
li a1, 301
li t0, 202
li t1, 234
li t2, 53
li t3, 2342
addi t3, t3, 51
add  t2, t2, t3
add  t1, t1, t2
add  t0, t0, t1
add  a1, a1, t0
li a0, 1
ecall
```

```
3183
```

Example: Left Associative Addition ⇒ Add with Immediate

```
301 + 202 + 234 + 53 + 2342 + 51
```

```
addi r1, r2, i
```

```
Program(
  [ PSText()
  , Li("a1", "301")
  , Addi("a1", "a1", "202")
  , Addi("a1", "a1", "234")
  , Addi("a1", "a1", "53")
  , Li("t0", "2342")
  , Add("a1", "a1", "t0")
  , Addi("a1", "a1", "51")
  , Li("a0", "1")
  , Ecall()
  ]
```

```
.text
li a1, 301
addi a1, a1, 202
addi a1, a1, 234
addi a1, a1, 53
li t0, 2342
add  a1, a1, t0
addi a1, a1, 51
li a0, 1
ecall
```

```
3183
```

Compiling an Expression Statement

```
rules

  program-to-rv32im:
    ast@Program(definitions, [stat]) → Program(instrs2)
    where
      a := <stx-get-ast-analysis>
      ; <stat-to-instrs(|"a1", <registers>)> stat ⇒ instrs1
      ; instrs2 := <concat> [
          [PSText()]
        , instrs1
        , [Li("a0", "1"),
           Ecall()]
        ]

  registers = !["t0", "t1", "t2", "t3", "t4", "t5", "t6"]

  stat-to-instrs(|r, regs) :
    Exp(e) → instrs
    where <exp-to-instrs(|r, regs)> e ⇒ instrs
```

program to instructions

available registers

statement to instructions

Expression to Instructions | Compiling Integer Literal

expression to instructions

catch failure

```
rules

  exp-to-instrs(|r, regs) =
    exp-to-instrs-(|r, regs)
    <+ (debug(!"exp-to-instr: "); fail)

  exp-to-instrs-(|r, regs) :
    Int(i) -> [Li(r, i)]
```

result in register r

available temporary registers regs

load integer literal

return list of instructions

Compiling Addition with Integer Literal

```
rules

  exp-to-instrs-(|r, regs) :
    add@Add(e, Int(i)) → <concat> [
        instrs
      , [Addi(r, r, i)]
    ]
    where
      <gtS>(i, "-2049"); <ltS>(i, "2048")
      ; <stx-get-ast-analysis> add ⇒ a
      ; <get-type(|a)> add ⇒ INT()
      ; <exp-to-instrs(|r, regs)> e ⇒ instrs
```

result in register r

addi: addition with integer literal

check range of literal

check type of expression

recursively translate e

Compiling Addition: General Case

```
rules

  exp-to-instrs-(|r, regs) :
    add@Add(e1, e2) → <concat> [
        instrs1
      , instrs2
      , [Add(r, r, r2)]
    ]
    where
      <stx-get-ast-analysis> add ⇒ a
      ; <get-type(|a)> add ⇒ INT()
      ; <exp-to-instrs(|r, regs)> e1 ⇒ instrs1
      ; <spill> regs ⇒ (r2, regs')
      ; <exp-to-instrs(|r2, regs')> e2 ⇒ instrs2
```

result in register r

available registers regs

check type of expression

recursively translate e1

take fresh register

recursively translate e2

Alternative Approaches

Problem: limited number of temporary registers

Push all results on stack

Use infinitely many temporary registers
+ register allocation

Compilation Schemas

Abstract From Implementation Details

$$|[\ i\ ]|_{r,regs} \Rightarrow \texttt{li r, } i$$

$$|[\ e + i\ ]|_{r,regs} \Rightarrow |[\ e\ ]|_{r,regs}\ \texttt{addi r, r, } i$$

$$|[\ e1 + e2\ ]|_{r1,r2,regs} \Rightarrow |[\ e1\ ]|_{r1,r2,regs}\ |[\ e2\ ]|_{r2,regs}\ \texttt{add r1, r1, r2}$$