v.8.0.0.11

4 Book

> ~3 Chapters still being rewritten

4.1 A Compiler Begins with a Language

To me, the study of compilers is the study of languages. As a mathematical object, a compiler is a transformation between languages, transforming a source language into a target language—but that is for another time. As a program, a compiler is a function that takes any program in a *source language* and produces some program that "behaves the same" in a *target language*.

Before we can write our first compiler, we need to choose languages for the compiler. We need a source language and target language. Often, we choose a pre-existing target language. Picking a target language is an interesting point in the design of any compiler. We pick a target language shortly. The source language, by contrast, is usually designed, often to achieve a balance between aesthetic and pragmatic goals. The rest of this course walks you through the design of a high-level language with functional and imperative features that can be compiled efficiently.

After we have picked our languages, we must write programs in the languages. In fact, we must do more—we must write programs *that write programs*, *i.e.,* we must write meta-programs. We will write programs that validate, analyze, and transform programs. We will write templates for programs, whole classes of programs. That is the nature of a compiler.

But we do not want to jump straight into writing meta-programs. That would be the sin of *premature abstraction*, attempting to solve the general problem before we understand how to solve a specific instance of the problem. We must learn to walk before we run. We must write programs before we write meta-programs.

4.1.1 Picking the target language

Our first language is x86-64 plus system calls, which we call *x64* for short. We'll start with a subset of this language and expand it slightly throughout the course. I elaborate on this language and what system calls are shortly.

This is an unusual choice for a target language in a contemporary compiler. Contemporary compilers often choose higher-level languages, such as C, LLVM, or even JavaScript as target language.

There are good reasons to a choose higher-level language as a target language. Often, a higher-level language provides more convenient abstractions, making generating programs simpler. The language may already have an efficient implementation, making writing optimizations less important. This frees the compiler writer to focus on the aesthetic goals of the new source language. Perhaps they want to focus on ruling out more errors statically, or providing convenient syntax for certain programming patterns they find common. These are common goals in source language and domain-specific language design.

However, choosing such languages as a target has a cost. These targets come with their own implementation, their own compilers and run-time systems. If we want to use them, we commit ourselves to using their toolchains as well. This may complicate our own development if it is difficult to setup those toolchains. Choosing a language also fixes an abstraction boundary—we cannot choose to work at a level of abstraction lower than our target language. This may prevent us from implementing certain functionality or optimizations.

For this course, we choose x64 for two main reasons. First, for educational goals, we want to write transformations and optimizations at some of the lowest abstraction layers we can. Second, for pragmatic goals, it is significantly easier on the students (many) to setup the toolchain for x64, at a minor cost to the course staff (few) who must maintain some x64 support code. The second reason is a common rationale in compiler design—simplifying something for the many users at the expense of the few compiler writers is usually a good trade-off.

4.1.2 Programming in x64

Having chosen our target language, we must now understand the abstractions it provides. As far as we are concerned, the target language was not designed—it was discovered

> Take a computer architecture course if you want to learn how it was designed.

It is the language of wild CPU found by geologists in cave somewhere, probably in the Amazon. It is primordial. It is not sensible to ask questions like "why"—x64 *is*. And, because we want to talk to the CPU, we must deal with it.

While we have choosen x64 as our target language, x64 itself does not run on most hardware. Instead, it is the source language yet another compiler, which compiles x64 to a binary machine code that I call bin64 for now. In this course, we use the `nasm` assembler to compile x64 into bin64. The compiled programs can run natively on most hardware, but you may need an interpreter depending on your machine.

> Usually, interpreters for machine code are called "virtual machines".

To get started with x64, let's consider the factorial function `fact`. In Racket, following the design recipe, we would simply write:

Examples:

```
; Natural Number -> Natural Number
; Takes a natural number n and returns the factorial of n.
; (fact 0) -> 1
; (fact 1) -> 1
; (fact 5) -> 120
> (define (fact n)
    (if (zero? n)
        1
        (* n (fact (sub1 n)))))
> (module+ test
    (require rackunit)
    (check-equal? (fact 0) 1)
    (check-equal? (fact 1) 1)
    (check-equal? (fact 5) 120))
> (fact 5)
120
```

We run this and the computer prints `120`.

We can implement the equivalent computation in x64 as follows.

> x64 actually supports at least two syntaxes. We use Intel assembler syntax, as it maps more closely to familiar imperative language syntax.

```
global start   ; Declare starting block for linker.

section .text                   ; Start of the "code"

;; Compute (fact 5)
start:
  mov r8, 5

;; Compute the factorial of the value in r8.
;; Expects  r8 : Positive int64
;; Produces r9 : Positive int64, the result of the factorial of r8.
fact:
  mov r9, 1                     ; Initialize accumulator to (fact 0).

;; Computes the factorial of r8 with accumulator r9
fact_acc:
  cmp r8, 0                     ; compare r8 and 0, i.e., (zero? r8)
  je fact_done                  ; jump if last comparison was equal
  imul r9, r8                   ; (set! r9 (* r9 r8))
  dec r8                        ; (set! r8 (sub1 r8))
  jmp fact_acc                  ; (fact r8)

fact_done:
;; Done. The result is in r9.

section .data ; The data section. This is where we declare initialized memory locations.

dummy: db 0
```

```
> nasm -f elf64 -o plain-fact-x64.o plain-fact-x64.s
> ld -e start -o plain-fact-x64.exe plain-fact-x64.o
> ./plain-fact-x64.exe
```

In this example, we compute the result, which is stored in `r9`. Then, the machine happily tries to execute the next instruction in memory. There is no next instruction, so it executes whatever happens to be in that memory location. The program hopefully crashes with a `SEGFAULT` or `SIGBUS` error, but this depends on the operating system (*OS*).

This fact that the behavior depends on the OS is our first clue that this example is not implemented in *just* x86-64, the language of the CPU. The meaning of the program, and thus the actual language in which we were programming, depends critically on the OS.

Each OS makes different assumptions about and enforce different restrictions on the above example. For example, the linker for each OS assumes a different entry label for where to start executing (although you can usually override this default). The entry label is assumed to be "start" on macOS (at least, on some versions), "_start" on Linux, and "Start" on Windows (depending on which Windows linker you use). On macOS, the `.data` section, which is used to declare initialized memory locations, cannot be empty, and the linker rejects the program if it is. In the above example, we don't actually use the heap, so we declare a dummy value to ensure compatibility with macOS. Most operating systems require that you explicitly "exit" the process, by calling a special, operating-specific "exit" procedure at the end of the process. Each OS defines a different set of these procedures, which we can think of as the "standard library" for x64. This example is compatible with macOS, Linux, and Windows, as long as you specify "start" as the entry label, and don't mind exiting with a bang.

Because of this critical dependence on the operating system, the target language in this course cannot *really* be "plain" x64—x64 is a family of programming languages, indexed by an operating system. In this class, we never program the raw CPU—we program the operating system. The CPU together with the operating system implements a different programming language than the CPU by itself. From a programming languages perspective, the operating system (*e.g.,* Linux) is the run-time system for the OS-flavoured x64 programming language (*e.g.,* x64-linux).

In x64, the result of the computation is not implicitly returned to the user via the REPL like it is in Racket. Instead, we have to explicitly say what the result is, and how to communicate it to the user. To do this, we need to use *system calls*, the procedures predefined by the operating system that define x64 interacts with the operating system.

After compiling our x64 program with `nasm`, we are left with program in the first target language—a machine code binary, which is just about the raw language of the CPU. In fact, it is machine code in a binary format described by the OS, which contains x64 instructions arranged according to the OS specification, with additional boilerplate (often installed by the linker) to ensure correct interoperation with the OS. I call this target language *bin64*, and like x64, it is a family of languages. As we rely on `nasm` to generate bin64, we do not study this target language in any detail, any normally forget about it's existence except when directing `nasm` to generate the right language.

> Take an operating systems course if you want to know more about how to program the "raw" CPU to implement the "lowest" level of programming language.

> Of course, a chip designer would really be implementing the "lowest" level language by implementing the interpreter in raw silicon.

> Or, well, I guess the electrical engineer, who builds the transistors that the chip designer uses, would be implementing the "lowest" language, which implements an interpreter for binary gates in the language of raw, analogue electrical signals.

> But what about the chemist who ... It's languages all the way down!

4.1.3 Flavours of x64

Now that we know x64 is actually a family of languages, let us study the different flavours of x64.

In this course, the main difference between the flavours of x64 are in the system calls. System calls are x64 primitives provided by the OS. Once we start using system calls, code becomes OS-specific. One of the first things a compiler writer will do is abstract away from system calls. But to abstract away from them, we need to understand how they work in various flavours of x64.

Our earlier example x64 program was limited becaues we did not know how to communicate. In the rest of this section, we walk through how to use basic system calls to make a complete program that can exit properly and communicate its result. We will introduce further system calls throughout this course.

4.1.3.1 x64-linux

In this course, *x64-linux* refers to the Linux-flavoured x64. This is x64, using Linux system calls, compiled to elf64. This version of x64 has few quirky restrictions and will be our "default", the point of comparison for other flavours of x64.

There are multiple system calls we could use to return a value to the system in Linux. For example, we could modify the program to return a value as an exit code through the `exit` system call. In Linux, each process must call the `exit` system call to terminate properly. `exit` takes an *exit code*, a number between 0 and 255, which is passed to the parent process after the child process exits. Usually, the exit code indicates whether the process terminated normally (0) or not (non-zero). But we could just as well use it to return the value of factorial.

```
  global start

  section .text

  start:
    mov r8, 5

  fact:
    mov r9, 1

  fact_acc:
    cmp r8, 0
    je fact_done
    imul r9, r8
    dec r8
    jmp fact_acc

  fact_done:
  exit:
    mov     rax, 60         ; I'm about to call the OS sys_exit function
    mov     rdi, r9         ; The exit code is the value from r9.
    syscall                 ; Perform the system call operation
```

```
> nasm -f elf64 -o exit-fact-x64-linux.o exit-fact-x64-linux.s
> ld -e start -o exit-fact-x64-linux.exe exit-fact-x64-linux.o
> ./exit-fact-x64-linux.exe
> echo $?
> 120
```

Now, when `fact` completes, we call the `exit` system call. This requires we set the register `rax` to the value `60` (in decimal), which corresponds to the exit system call in Linux. This system call expects one argument, the exit code, passed in register `rdi`.

In general, the Linux system call interface uses the following calling convention. The system call number is set in register `rax`. The arguments are set according to the System V AMD64 ABI. The first six integer or pointer arguments are passed in register (in-order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. Remaining arguments are passed on the stack, although we will not use any system calls with more than six arguments. We also won't be dealing with floating point arguments, which are passed in separate registers. For more details, Wikipedia is not a bad place to start: https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI. But you could also go to the source: https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

To compile this on Linux, we run `nasm -f elf64 exit-fact-x64-linux.s -o exit-fact-x64-linux.o`. This will generate a binary in the elf64 target language. The *elf64* is the Linux-flavoured bin64, a binary format used on Linux. It includes the x64 instructions and additional structure that allows the Linux operating system to control the binary.

Next, we link the file by running `ld -e start -o exit-fact-x64-linux.exe exit-fact-x64-linux.o`, which essentially connects the binary to the operating system and any other external libraries, and specifies the entry point where execution should begin (the `start` label). Now we can execute the file `exit-fact-x64-linux.exe` on the command line in the usual way, *e.g.,* set it to executable using `chmod +x exit-fact-x64-linux.exe` then execute with `./exit-fact-x64-linux.exe`. We can get the result of the exit code using `echo $?` or `echo $status`, depending on your shell.

Most programming languages communicate their result by printing. Thankfully, the OS does provide a print system call. We could write the program below to print the answer instead of using the exit status, to get a somewhat friendly user interface.

```
  global start

  section .text

  start:
    mov r8, 5

  fact:
    mov r9, 1

  fact_acc:
    cmp r8, 0
    je fact_done
    imul r9, r8
    dec r8
    jmp fact_acc

  fact_done:
    mov r8, msg  ; Move the address of `msg` into `r8`.
    mov [r8], r9 ; move r9 into the 0th index of the address pointed to by r8.

  print_msg:
    mov     rax, 1      ; Say, "I'm about to call the OS sys_write function"
    mov     rdi, 1      ; And I want it to write to the standard output port
    mov     rsi, msg    ; The message to print is stored in msg
    mov     rdx, len    ; Length is len.
    syscall

  exit:
    mov     rax, 60
    mov     rdi, 0                ; The exit code is 0; normal.
    syscall

  section .data

  len:   equ   1         ; The constant 1, representing the length of the message.
  msg:   times len dw 0  ; A `len` length, in bytes, buffer, initialized to be full of 0.
```

```
> nasm -f elf64 -o fact-x64-linux.o fact-x64-linux.s
> ld -e start -o fact-x64-linux.exe fact-x64-linux.o
> ./fact-x64-linux.exe
x
```

To use the `print` system call, we need to create a memory buffer to store the message we wish to print. We can statically allocate a buffer in the data section by creating a label, `msg:`, and using the `times` keyword to allocate `len` bytes (indicated by `dw`) of space, and initializing each to `0`. Then we call the print system call, number `1`, specifying the standard output port as the file to print to by setting `rdi` to `1`, specifying `msg` as the address to print from in `rsi`, and the length of the buffer in `rdx`. When we run the program, it prints the expected result—`x`.

Question: Why is `x` the expected result instead of `120`?

You can find complete lists of the Linux system calls online:

- https://syscalls64.paolostivanin.com/
- https://filippo.io/linux-syscall-table/

4.1.3.2 x64-macos

macOS is very similar to Linux, and we can easily adapt the above examples to macOS. On macOS, there are 4 system call tables, computed by adding the system call number to a specific prefix. The BSD system call table, the most generic and Linux-like, uses the hex prefix `#x2000000`. The exit system call is system call 1 in the BSD table, so we use the system call number `#x2000001`. The example "exit-fact-x64-linux.s" from above is rewritten for macOS below.

```
  global start

  section .text

  start:
    mov r8, 5

  fact:
    mov r9, 1

  fact_acc:
    cmp r8, 0
    je fact_done
    imul r9, r8
    dec r8
    jmp fact_acc

  fact_done:
  exit:
    mov     rax, 0x2000001       ; I'm about to call the OS sys_exit function.
                                 ; Specifying the value in hex for readability,
                                 ; but could convert it to decimal for a simpler
                                 ; assembler.
    mov     rdi, r9              ; The exit code is the value from r9.
    syscall

  section .data

  dummy: db 0
```

```
> nasm -f macho64 -o exit-fact-x64-macos.o exit-fact-x64-macos.s
> ld -macosx_version_min 10.6 -e start -o exit-fact-x64-macos.exe exit-fact-x64-macos.o
> ./exit-fact-x64-macos.exe
> echo $?
> 120
```

To compile this file, we run `nasm -f macho64 exit-fact-x64-macos.s exit-fact-x64-macos.o`. `macho64` is the binary formatted used by 64-bit macOS. To link, we run `ld -macosx_version_min 10.6 -e start -o exit-fact-x64-macos.exe exit-fact-x64-macos.o`. We need to specify a minimum macOS version number of 10.6, otherwise the linker will ignore the custom entry label and fail to link. We can execute the file `exit-fact-x64-macos.exe` on the command line in the usual way, and can get the result of the exit code using `echo $?` or `echo $status`, depending on your shell.

macOS also has a similar write system call: number 4 in the BSD table. The file `fact-x64-linux.s` is ported to macOS below.

```
  global start

  section .text

  start:
    mov r8, 5

  fact:
    mov r9, 1

  fact_acc:
    cmp r8, 0
    je fact_done
    imul r9, r8
    dec r8
    jmp fact_acc

  fact_done:
```

```
    mov r8, msg  ; Move the address of `msg` into `r8`.
    mov [r8], r9 ; move r9 into the 0th index of the address pointed to by r8.

  print_msg:
    mov     rax, 0x2000004    ; Say, "I'm about to call the OS sys_write function"
    mov     rdi, 1            ; And I want it to write to the standard output port
    mov     rsi, msg          ; The message to print is stored in msg
    mov     rdx, len          ; Length is len.
    syscall

  exit:
    mov     rax, 0x2000001
    mov     rdi, 0                ; The exit code is 0; normal.
    syscall

  section .data

  len:   equ   1         ; The constant 1, representing the length of the message.
  msg:   times len dw 0  ; A `len` length, in bytes, buffer, initialized to be full of 0.
```

```
> nasm -f macho64 -o fact-x64-macos.o fact-x64-macos.s
> ld -macosx_version_min 10.6 -e start -o fact-x64-macos.exe fact-x64-macos.o
> ./fact-x64-macos.exe
x
```

4.1.4 x64-windows

Windows does not allow user processes to perform system calls. Instead, we have to call kernel functions that provide similar functionality. The `ExitProcess` kernel function provides the same functionality as the exit system call on Linux and macOS. However, making a function call is more complex than making a system call. We have to declare external functions for the linker, and ensure we link with the library that includes that function.

```
  global Start   ; GoLink expects "Start"
    extern ExitProcess ; Declare that ExitProcess is an external symbol to be resolved by the linker.

  section .text

  Start:
    mov r8, 5

  fact:
    mov r9, 1

  fact_acc:
    cmp r8, 0
    je fact_done
    imul r9, r8
    dec r8
    jmp fact_acc

  fact_done:
  exit:
    mov rcx, r8                  ; Windows 64-bit calling convention uses rcx as first argument
    call ExitProcess             ; Windows doesn't expose system calls to user processes; call ExitProcess function to exit.
```

```
> nasm -f win64 -o exit-fact-x64-windows.o exit-fact-x64-windows.s
> golink /entry Start /fo exit-fact-x64-windows.exe exit-fact-x64-windows.o kernel32.dll
> ./exit-fact-x64-windows.exe
> echo $?
> 120
```

Windows also doesn't ship with a linker. GoLink is a small, freely available linker. After downloading `nasm` and `GoLink`, you can compile using `nasm -f win64 exit-fact-x64-windows.s -o exit-fact-x64-windows.o` and link using `golink /entry Start /fo exit-fact-x64-windows.exe exit-fact-x64-windows.o kernel32.dll`.

4.2 The First Compiler: Abstracting Boilerplate

Once we have picked a target language, a language from which to start building, we have fixed some set of abstractions. From this starting point, we ask a question:

> What's wrong with this language?

Our goal in designing and implementing a language is to systematically design and build new layers of abstractions. These abstractions are meant to solve some problem, such as software development being error-prone, software design being complex, or software produced in the language being unsafe, unportable, or verbose. Ideally, we solve these problems without introducing some cost, such as a high learning curve, or introducing some performance cost. To solve these problems, we must first concretely identify one, and then design a layer of abstraction to address it.

The first limitation in x64 we identify is boilerplate. Writing programs in x64 requires the programmer to insert repetitive boilerplate, such as the declaration of the initial label, and some code to exit the program and report the result to the user. This boilerplate prevents the user from focusing on the program and requires them to copy and paste the same snippets of code into their programs, an error-prone process if that snippet ever needs to change.

To address this, we design a new source language that is free of this boilerplate, and design a compiler to transform the source language into the target language by introducing the boilerplate.

4.2.1 Designing an abstraction

Our goal is to introduce the abstraction of *instruction sequences*: lists of instructions that represent the code of an x64 program. Instruction sequences separate the code from the boilerplate. As a result, we get a notion of program composition, allowing us to focus on the program, and decompose a program into separate pieces that we can easily stitch together. Supposing *p_1* and *p_2* are both instruction sequences, then there exists `(p-append p_1 p_2)` (for some definition of `p-append`) which first executes the instructions in *p_1* and then executes the

instructions in *p_2*.

Since these are not valid x64 programs on their own, we need to define the meaning of instruction sequences to make clear whether we are compiling them correctly. We define their meaning by describing an interpreter for instruction sequences: executing the instruction sequence begins at the first instruction in the sequence, and ends with the last instruction. We describe how to execute each instruction once we fix a specific set of instructions. The final result of the instruction sequence is the value of some designated register after the last instruction is executed.

Below, we select the subset of x64 instructions we plan to suppport.

- `mov` *`triv`*`, `*`triv`*

  This string represents the move instruction, which moves a value from one location to another, or moves a value into a locaction.

  x64 imposes further restrictions on `mov` (remember—x64 *is*, we must not ask why). We can only move a value into a register, or a value in one register to another register.

- `add` *`triv`*`, `*`triv`*

  This string represents the add instruction, which intuitively adds two values.

  In fact, we cannot add values in x64: we can add a value (*i.e.,* an integer), to a register `add` *`reg`*`, `*`integer`*, or add the values of two registers, `add` *`reg_1`*`, `*`reg_2`*.

  Furthermore, when using an integer directly in the add statement, it must be a 32-bit integer, in the range `-2^31 <= i <= 2^31 - 1`. Yes, that's right, 32-bit not 64-bit; we do not ask "why" of x64. For example, `add rax, 2147483647` is valid, but `add rax, 2147483648` is not. Instead, we would first need to move `2147483648` into a register to add it to `rax`. For example:

  ```
  mov rbx, 2147483648
  add rax, rbx
  ```

- `imul` *`triv`*`, `*`triv`*

  This string represents the multiply instruction, which intuitively multiplies two values.

  Multiplication is further restricted by x64. Again, we cannot multiply

values directly. We can only multiply the value of a register by a 32-bit integer `imul` *`reg_1`*`, `*`int32`*, or the value in a register by the value in a register `imul` *`reg_1`*`, `*`reg_2`*.

Below is an example of a valid sequence of instructions in our subset of x64.

```
mov rax, 170679
mov rdi, rax
add rdi, rdi
mov rsp, rdi
imul rsp, rsp
mov rbx, 8991
```

Note that this does not correspond to a x64 *program*, as it is missing much of the structure: the starting label, the section declarations, etc. The goal of our first compiler is to capture this notion of instruction sequences, and insert the necessary boilerplate to generate a valid x64 program.

We represent x64 instruction sequences as Racket strings, with each instruction separated by newline characters. For example, the following x64 instruction sequence `mov rax, 42` corresponds to the Racket string `"  mov rax, 42"`.

Note that the representation of x64 instruction sequences is whitespace sensitive. For example, the following program corresponds to the Racket string `"  mov rbx, 2147483648\\n  add rax, rbx"`.

```
mov rbx, 2147483648
add rax, rbx
```

4.2.2 Defining a Source Language

Above we defined instruction sequences in terms of the concrete syntax of x64 and strings, but this syntax is not convenient for a compiler to manipulate. Concrete syntax often contains irrelevant details that are useful for human programmers, but irrelevant to a machine. For example, strings are also difficult to work with, as they lack structure for conveniently accessing substructures, and whitespace-sensitivity makes means there is multiple ways to represent identical programs.

Therefore, we define an abstract syntax our new language, *Paren-x64 v1*, with its new instruction sequence abstraction. We first present a simple definition, then gradually refine the definition to encode more constraints.

```
       p ::= (begin s ...)

       s ::= (set! triv triv)
           | (set! triv (binop triv triv))

     triv ::= reg
           | integer

   binop ::= +
           | *

     reg ::= rsp
           | rbp
           | rax
           | rbx
           | rcx
           | rdx
           | rsi
           | rdi
           | r8
           | r9
           | r10
           | r11
           | r12
           | r13
           | r14
           | r15

integer ::= integer?
```

Every Paren-x64 v1 program, *p*, begins with `begin`, followed by a sequence of statements (instructions). The non-terminal *p* defines the instruction sequence abstraction. We can define the composition operation as follows:

```
(define (p-append p1 p2)
  (match `(,p1 ,p2)
    [`((begin ,s1 ...) (begin ,s2 ...))
     `(begin ,@s1 ,@s2)]))
```

Each instruction *s* corresponds to one of the x64 instructions described earlier. For example, the x64 instruction `mov` *reg*, *integer* corresponds to the Paren-x64 v1 instruction `(set!` *reg integer*`)`. Our abstract syntax makes more clear that the arithmetic operations are actually a combination of an arithmetic operation and an operation that changes state. The instruction `add` *reg*, *integer* is represented `(set!` *reg* `(+` *reg integer*`))`. Note that this duplicates the register,

and the two occurences must be the same in order to be translated to a valid x64.

Some of the restrictions from x64 are not apparent in the definition of the grammar. To simplify precisely defining languages as grammars without too much additional English specification, we create two conventions.

- Any time we suffix a non-terminal by an underscore and a number, such as *reg_1*, this is a reference to a particular instance of a non-terminal, and it is restricted to be identical to any other instance of the same non-terminal with the same underscore suffix within the same expression. So `(set! reg_1 (+ reg_1 integer))` represents an `add` instruction in Paren-x64 v1, and both occurrences of *reg_1* must be the same register. However, two instances of non-terminals with different suffixes, like *reg_1* and *reg_2*, are not necessarily different—they are only not guaranteed to be the same.
- Any non-terminal that is not defined as syntax, such as *int64*, may be defined by a Racket predicate, such as `int64?`. Such definitions will link to the documentation defining the predicate.

Using these two conventions, we define the final grammar of Paren-x64 v1 with all the above restrictions below.

```
p ::= (begin s ...)

s ::= (set! reg int64)
    | (set! reg reg)
    | (set! reg_1 (binop reg_1 int32))
    | (set! reg_1 (binop reg_1 reg))

reg ::= rsp
      | rbp
      | rax
      | rbx
      | rcx
      | rdx
      | rsi
      | rdi
      | r8
      | r9
      | r10
      | r11
      | r12
      | r13
```

```
        |  r14
        |  r15

  binop ::= *
        |  +

  int64 ::= int64?

  int32 ::= int32?
```

The predicates `int64?` and `int32?` are defined by the support library `cpsc411/compiler-lib`, and return `#t` if and only if given an integer in the range for a 64-bit or 32-bit, respectively, signed two's complement integer.

Since Paren-x64 v1 is an imperative language and does not return values like Racket does, we must decide how to interpret a Paren-x64 v1 program as a value. As discussed above, we can do this by imposing some convention on what the value of the program is. We choose to interpret the value of an Paren-x64 v1 program as the value in `rax` when the program is finished executing (has no instructions in the instruction sequence left to execute). This choice is completely up to us, but this choice is designed to continue to work well as we build up our compiler.

We also require that any register is initialized before it is accessed. We inherit this restriction from x64. In x64, the value of an uninitialized register is undefined, that is, accessing an uninitialized register results in *undefined behaviour*, behavior that has no specified behavior defined by the language specification.

Undefined behaviour is common in low-level languages that lack a strong enough enforcement mechanism for checking assumptions. Eliminating undefined behavior by adding static or dynamic checks in the source language improves the ability of programmers to predict behaviour of all programs in your language. However, it is not always practical to achieve. It may be too difficult to statically check an assumption and still allow all the programs you want to allow, or too expensive to check a property dynamically. In these cases, we are forced to make assumptions that we cannot enforce, injecting undefined behaviour into our compiler.

In this book, we make it a non-negotiable goal: source languages must never have undefined behaviour. If they might, we hobble them until we can eliminate the undefined behaviour.

4.2.3 Enforcing Assumptions

One of the jobs of the front-end of a compiler is to enforce assumptions the rest of the compiler makes. Enforcement mechanisms take various forms, such as parsers, type checkers, linters, and static analyses.

We're going to design a function `check-paren-x64` to validate Paren-x64 v1 programs. It is, in essence, a parser. It reads an arbitrary value, expected to represent a Paren-x64 v1 program, and returns a valid program in the language Paren-x64 v1, or raises an error. However, it is a trivial parser. Usually we think of parsers as transforming from one representation to another, but this parser does not transform the input if it is valid. It does not deal with transforming strings into abstract syntax. Instead, it expects its input to be abstract syntax already, and either returns it if the abstract syntax is already valid, or raises an error. This means our parsers, like the rest of our compiler, are simple tree automata, rather than complex string automata.

You could view `check-paren-x64` as a type checker. In this view, it checks for a single type: *The-Paren-x64-Type*, which every valid instruction has, and which has quite simple typing rules. `check-paren-x64` checks that the input program is following the typing disciplines of the language (which aren't very restrictive).

Writing validators, such as parsers or type checkers, for intermediate language programs produced by your compiler is a powerful debugging technique. By designing them in the same way as `check-paren-x64`, so that they return the input if it's valid, you can easily add them as passes in your compiler and detect when an early pass produces an ill-typed program. This is a form of *property-based testing*, and will catch many more bugs than unit testing alone.

To ensure no undefined behaviour, our validator should check the following.

- The input conforms to the grammmar of Paren-x64 v1, including restrictions regarding the valid range of integers and when the same register must appear in two places.
- No register is referred to before is it initialized. We assume that no register is initialized at the beginning of a program.
- The register `rax` is initialized before the end of the program.

We split this into two separate functions to enable separation of concerns. We always want our syntax to be valid, but because Paren-

x64 v1 will serve as a target language, we may not always want to enforce that registers are provabably initialized. For example, the language may evolve to the point where checking this will be undecidable, and source languages will be responsible to enforcing the guarantee isntead. By separating the two checks, we'll be able to reuse code in this eventuality.

First, we check that the syntax is valid.

(**check-paren-x64-syntax** *x*) → paren-x64-v1? procedure
  x : any/c

Takes an arbitrary value and either returns it, if it is valid Paren-x64 v1 syntax, or raises an error with a descriptive error message.

Examples:

```
> (check-paren-x64
    `(begin (set! rax ,(min-int 64))))
'(begin (set! rax -9223372036854775808))
> (check-paren-x64
    `(begin (set! rax ,(- (min-int 64) 1))))
check-paren-x64: Invalid statement; expected one of:
  Expected a triv, got -9223372036854775809
  something of the form `(set! ,reg_1 (,binop ,reg_2
,int32)), but got '(set! rax -9223372036854775809)
  something of the form `(set! ,reg_1 (,binop ,reg_2
,reg_3)), but got '(set! rax -9223372036854775809)
> (check-paren-x64-syntax
    '(begin
       (set! r17 170679)))
check-paren-x64: Invalid statement; expected one of:
  Expected a register, got r17
  something of the form `(set! ,reg_1 (,binop ,reg_2
,int32)), but got '(set! r17 170679)
  something of the form `(set! ,reg_1 (,binop ,reg_2
,reg_3)), but got '(set! r17 170679)
> (check-paren-x64-syntax
    '(begin
       (set! rax 170679)
       (set! rdi rax)
       (set! rdi (+ rdi rdi))
       (set! rsp rdi)
       (set! rsp (* rsp rsp))
```

```
      (set! rbx 8991)))
'(begin
   (set! rax 170679)
   (set! rdi rax)
   (set! rdi (+ rdi rdi))
   (set! rsp rdi)
   (set! rsp (* rsp rsp))
   (set! rbx 8991))
```

Then, we check register initialization. Note that this procedure can assume its input is well-formed Paren-x64 v1 syntax, and only concern itself with register initialization.

(**check-paren-x64-init** *x*) → paren-x64-v1? procedure
x : paren-x64-v1?

Takes valid Paren-x64 v1 syntax, and returns a valid Paren-x64 v1 program or raises an error with a descriptive error message.

Examples:

```
> (check-paren-x64-init
   '(set! (+ rax rdi) 42))
check-paren-x64-init: contract violation
  expected: paren-x64-v1?
  given: '(set! (+ rax rdi) 42)
  in: the 1st argument of
      the check-paren-x64-init method in
      (class/c
       (check-paren-x64 (->m any/c paren-x64-v1?))
       (check-paren-x64-syntax
        (->m any/c paren-x64-v1?))
       (check-paren-x64-init
        (->m paren-x64-v1? paren-x64-v1?))
       (generate-x64 (->m paren-x64-v1? string?))
       (interp-paren-x64
        (->m paren-x64-v1? int64?))
       (wrap-x64-run-time (->m string? string?))
       (wrap-x64-boilerplate
        (->m string? string?)))
  contract from: (definition a1-compile)
  contract on: a1-compile
  blaming: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1
-solution.rkt
```

```
   (assuming the contract is correct)
  at: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1-solu
tion.rkt:268.17
> (check-paren-x64-init
   '(begin
      (set! (+ rax rdi) 42)))
check-paren-x64-init: contract violation
  expected: paren-x64-v1?
  given: '(begin (set! (+ rax rdi) 42))
  in: the 1st argument of
      the check-paren-x64-init method in
      (class/c
       (check-paren-x64 (->m any/c paren-x64-v1?))
       (check-paren-x64-syntax
        (->m any/c paren-x64-v1?))
       (check-paren-x64-init
        (->m paren-x64-v1? paren-x64-v1?))
       (generate-x64 (->m paren-x64-v1? string?))
       (interp-paren-x64
        (->m paren-x64-v1? int64?))
       (wrap-x64-run-time (->m string? string?))
       (wrap-x64-boilerplate
        (->m string? string?)))
  contract from: (definition a1-compile)
  contract on: a1-compile
  blaming: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1
-solution.rkt
   (assuming the contract is correct)
  at: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1-solu
tion.rkt:268.17
> (check-paren-x64-init
   '(begin
      (set! rax (+ rax 42))))
check-paren-x64-init: Invalid statement; expected one of:
  Expected a triv, got (+ rax 42)
  Cannot involve the initialized register rax in a binop
  Expected a register, got 42
> (check-paren-x64-init
   '(begin
      (set! rax (+ rdi 42))))
check-paren-x64-init: contract violation
  expected: paren-x64-v1?
  given: '(begin (set! rax (+ rdi 42)))
  in: the 1st argument of
      the check-paren-x64-init method in
      (class/c
```

```
        (check-paren-x64 (->m any/c paren-x64-v1?))
        (check-paren-x64-syntax
         (->m any/c paren-x64-v1?))
        (check-paren-x64-init
         (->m paren-x64-v1? paren-x64-v1?))
        (generate-x64 (->m paren-x64-v1? string?))
        (interp-paren-x64
         (->m paren-x64-v1? int64?))
        (wrap-x64-run-time (->m string? string?))
        (wrap-x64-boilerplate
         (->m string? string?)))
  contract from: (definition a1-compile)
  contract on: a1-compile
  blaming: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1
-solution.rkt
   (assuming the contract is correct)
  at: <pkgs>/cpsc411-reference-lib/cpsc411/reference/a1-solu
tion.rkt:268.17
> (check-paren-x64-init
   '(begin
      (set! rax 170679)
      (set! rdi rax)
      (set! rdi (+ rdi rdi))
      (set! rsp rdi)
      (set! rsp (* rsp rsp))
      (set! rbx 8991)))
'(begin
   (set! rax 170679)
   (set! rdi rax)
   (set! rdi (+ rdi rdi))
   (set! rsp rdi)
   (set! rsp (* rsp rsp))
   (set! rbx 8991))
```

For convenience, we define a single validator that validates all properties of the language as `check-paren-x64`.

(**check-paren-x64** *x*) → paren-x64-v1? procedure

x : any/c

Takes an arbitrary value and either returns it, if it is valid Paren-x64 v1 program, or raises an error with a descriptive error message.

4.2.4 Understanding Meaning

When we create a new language, we want to ensure we understand its meaning separate from how it is compiled. This is for two reasons. First, optimizations depend on when various programs in a language are equivalent. We need to understand the language in order to understand when programs are equivalent. Second, we cannot know whether or not the compiler is correct if we do not know the meaning of programs before they are compiled. Unit tests will help us debug, but when we know the meaning of *all* programs in the language, we can say whether that meaning is preserved through compilation.

We can define the meaning of a language by writing an interpreter. To design an interpreter for Paren-x64 v1 is straightforward. We implement a register machine: a recursive function over instruction sequences that interprets each instructions by modifying a dictionary mapping registers mapped to values. When there are no instructions left, the interpreter returns the value of the designated register `rax`.

In the interpreter, we assume the input is a valid Paren-x64 v1 program. In a user interface, we would validate all input, for example by using `check-paren-x64`, but in the implementation of the interpreter, we keep the two concerns separate. Instead, the interpreter is free to assume all integers are in the right range, and arithmetic instructions correctly refer to the same register in both operand positions. For example, in the instruction `(set! reg_1 (+ reg_2 integer))`, we assume *`reg_1`* and *`reg_2`* are identical, and *`integer`* is a 32-bit integer, since otherwise the input would not have been a valid Paren-x64 v1 program. In fact, it would be bad style to check these again, since this mixes concerns and duplicates code.

`(`**`interp-paren-x64`** *`x`*`)` → `int64?` procedure

`x` : `paren-x64-v1?`

Interprets the Paren-x64 v1 program, returning the final value as an exit code in the range 0–255.

Examples:

```
> (interp-paren-x64
   '(begin
      (set! rax 0)
      (set! rax (+ rax 42))))
42
```

```
> (interp-paren-x64
   '(begin
      (set! rax 170679)
      (set! rdi rax)
      (set! rdi (+ rdi rdi))
      (set! rsp rdi)
      (set! rsp (* rsp rsp))
      (set! rbx 8991)))
183
```

To properly implement arithmetic operations, you need to handle two's complement arithmetic, which overflows on large positive numbers and underflows on small negative numbers. You may want to use `x64-add` and `x64-mul` from `cpsc411/compiler-lib`.

Once we have the meaning of programs defined, we can define what it means for a compiler to be correct.

A compiler for Paren-x64 v1 is correct if:

- the meaning (as defined by the interpreter) of a program *p* is the value *integer_1*
- we compile *p* and execute it as a x64 program and get the value *integer_2*
- the values *integer_1* and *integer_2* are *equivalent*. In general, we have to define equivalence for each pair of source and target languages. In this case, the interpreter and the compiler should return *the same* value.

4.2.5 Compiling

Finally, we get to compiling. We have designed our new abstraction, made it precise in the form of a language, enforced our assumptions, and understood its meaning

The job of our compiler is to translate one level of abstraction into another. We currently have a three levels of abstraction: (1) Paren-x64 v1, the abstract syntax representation of x64 instruction sequences, (2) the string representation of x64 instruction sequences, and (3), x64 programs. The structure of our compiler is determined partially by these levels of abstractions, both the source layer and the target layer. There are two pieces that need to be added to a Paren-x64 v1 program to make it a complete x64 program: (1) boilerplate, such as the

declaration of the starting label and (2) the run-time system, which implements our convention for the meaning of instruction sequences.

We design our compiler as a series of compiler passes along these natural lines in order to separate concerns as much as possible. First, we translate from Paren-x64 v1 into the string representation of instruction sequences. Then, we introduce the run-time system. Finally, we wrap the whole thing in boilerplate. This order is not arbitrary; it is dictated by our abstract layers. Implementing the run-time system requires instructions outside the subset used by Paren-x64 v1, so we must reach a lower level of abstraction to implement it. However, it can be implemented as an instruction sequence, and we want to take advantage of instruction sequence composition if we can. We won't be able to do that after introducing boilerplate.

`(generate-x64 p) → string?` procedure

`p : paren-x64-v1?`

Compiles a Paren-x64 v1 program into a x64 instruction sequence represented as a string.

Examples:

```
> (generate-x64
   '(begin
      (set! rax 0)
      (set! rax (+ rax 42))))
"mov rax, 0\nadd rax, 42\n"
> (require racket/pretty)
> (pretty-display
   (generate-x64
    '(begin
       (set! rax 0)
       (set! rax (+ rax 42)))))
mov rax, 0
add rax, 42
> (pretty-display
   (generate-x64
    '(begin
       (set! rax 170679)
       (set! rdi rax)
       (set! rdi (+ rdi rdi))
       (set! rsp rdi)
       (set! rsp (* rsp rsp))
       (set! rbx 8991))))
```

```
mov rax, 170679
mov rdi, rax
add rdi, rdi
mov rsp, rdi
imul rsp, rsp
mov rbx, 8991
```

4.2.5.1 Implementing a Run-time System

The abstractions provided by the operating system for running x64 are not the same as the convention we just created for Paren-x64 v1. The operating system does not know it should "return" the result of `rax`, whatever "return" means. To implement this convention, we need to write some x64 code (ideally, an instruction sequence) that will take any instruction sequence implementing the Paren-x64 v1 convention and communicate the result to the operating system. This code is a very simple run-time system.

Our choice of run-time system depends on the abstractions provided by the language, the machine, the operating system, and the user interface we desire. Some languages print the final result. Some languages discard the final result, relying on the user program to print the result, or modify the filesystem, or modify the state of the machine (or the world) in some other non-temporary way.

Our language does not provide the user with any way to observe the state of the machine, so our run-time system must do the job of communicating the return value to the user.

We could print the result, but as we saw in the factorial example above, the operating system's definition of "print" does not match our intuition. When trying to print "120", we get the character "x". This would make for a very confusing user interface, or a very complicated run-time system.

Instead, we opt for a very simple run-time system: communicate via the operating system exit code. This exit code is a number between 0 and 255 given to the exit system call, and is easily accessible in shells via the variable `$?` (or `$status` in some shells). In Racket, we can access the exit code of a subprocess using `system/exit-code`. This limits how much our programs can communicate; we will lift that restriction in later versions of our compiler.

Our run-time system is an x64 instruction sequence which expects to

be composed after another instruction sequence. The run-time system assumes that the first instruction sequence must initialize `rax`. The run-time system then calls the `exit` system call with the value of `rax` passed as the exit code. `cpsc411/compiler-lib` provides some definitions, such as `sys-exit`, that are helpful for this.

For formatting strings in Racket, you may want to investigate `format`, `~a`, and `at-exp`.

`(wrap-x64-run-time x) → string?` procedure

x : `string?`

Installs the Paren-x64 v1 run-time system. The input is the same as the output for `generate-x64`: a string representing an x64 instruction sequence. The run-time system is composed with the input as a second instruction sequence.

4.2.5.2 Wrapping it all up

Finally, we implement a simple pass to wrap the instruction sequence with the x64 boilerplate described in A Compiler Begins with a Language.

`(wrap-x64-boilerplate x) → string?` procedure

x : `string?`

Takes an x64 instruction sequence and wraps it with the necessary boilerplate to return a complete x64 program in Intel syntax.

After that, we have a complete compiler. The compiler is easily defined by composing all the individual passes.

```
(define (paren-x64-v1-compiler x)
  (wrap-x64-boilerplate (wrap-x64-run-time (generate-x64 x))))

(define paren-x64-v1-compiler^
  (compose wrap-x64-boilerplate wrap-x64-run-time generate-x64))
```

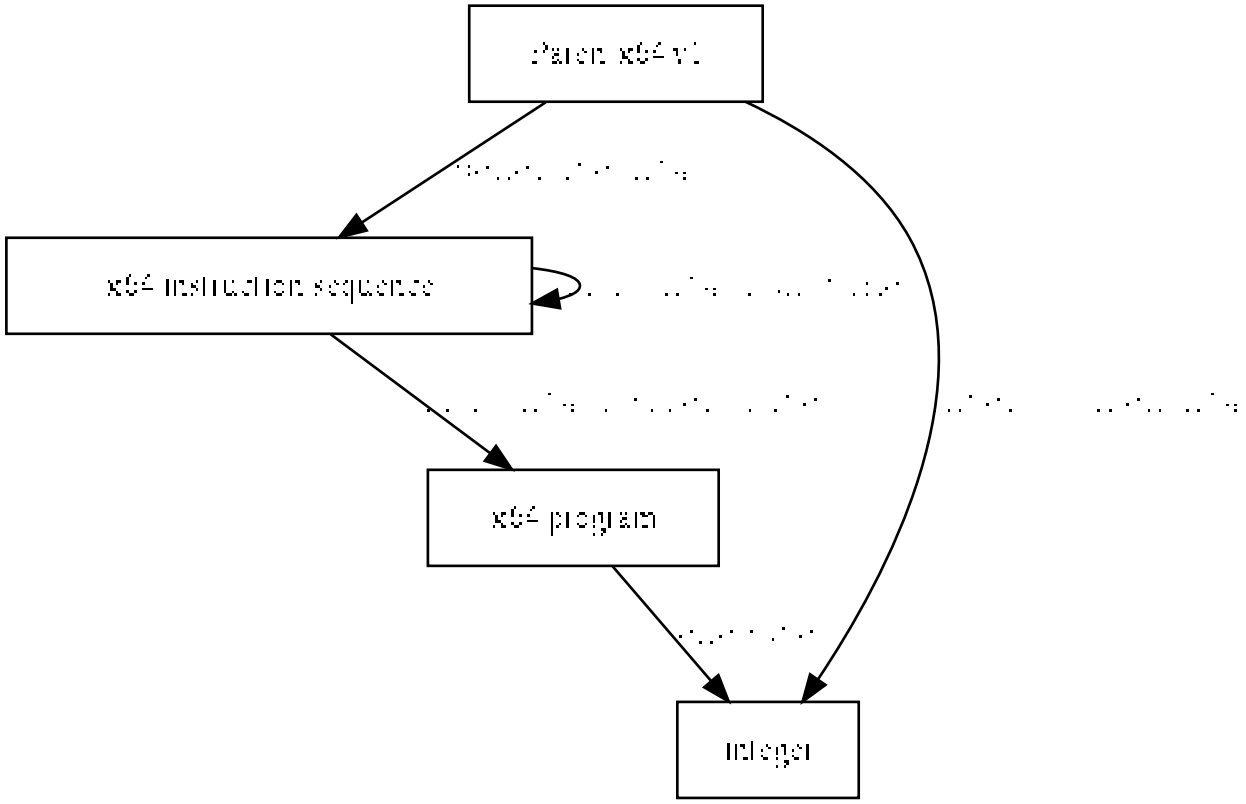
Examples:

```
> (interp-paren-x64
   '(begin
```

```
      (set! rax 170679)
      (set! rdi rax)
      (set! rdi (+ rdi rdi))
      (set! rsp rdi)
      (set! rsp (* rsp rsp))
      (set! rbx 8991)))
183
> (current-pass-list
   (list
    check-paren-x64
    generate-x64
    wrap-x64-run-time
    wrap-x64-boilerplate))
> (execute
   '(begin
      (set! rax 170679)
      (set! rdi rax)
      (set! rdi (+ rdi rdi))
      (set! rsp rdi)
      (set! rsp (* rsp rsp))
      (set! rbx 8991)))
#<eof>
```

The support library `cpsc411/compiler-lib` provides a few abstractions for deriving the compiler from a list of passes. See `current-pass-list`, `compile`, and `execute`.

4.2.6 Appendix: Compiler Overview



4.2.7 Appendix: Language Definitions

(**paren-x64-v1?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-v1` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v1 : grammar?

```
p ::= (begin s ...)

s ::= (set! reg int64)
    | (set! reg reg)
    | (set! reg_1 (binop reg_1 int32))
    | (set! reg_1 (binop reg_1 reg))

reg ::= rsp
      | rbp
      | rax
      | rbx
      | rcx
      | rdx
```

```
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r10
        |  r11
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

int64 ::= int64?

int32 ::= int32?
```

4.3 Abstracting Machine Details

4.3.1 Preface: What's wrong with our language?

In the last chapter, we designed our first language, Paren-x64 v1, and wrote our first compiler to implement it. This language introduces the barest, although still very useful, abstraction—freedom from boilerplate. Paren-x64 v1 abstracts away from the boilerplate of x64, and details of exactly how to pass a value to the operating system.

While Paren-x64 v1 is an improvement of x64, it has a significant limitation for writing programs that we will address this week. The language requires the programmer to remember many machine details—in particular, which of the small number of registers are in use—while programming. Human memory is much less reliable than computer memory, so we should design languages that make the computer remember more and free the human to remember less. This will prevent the human from causing run-time errors when they inevitable make a mistake and overwrite a register that was still in use.

4.3.2 Introduction to Designing a Source language

When designing a new language, I often start by writing some programs until I spot a pattern I dislike.

The pattern in Paren-x64 v1 is that all computations act on a small set of 16 physical locations.

This limits the way we can write computations. We must, manually, order and collapse sub-computations to keep the number of locations that are in use small. We must keep track of which locations are still in use before we move a value into a location, or we will overwrite part of our computation.

Furthermore, the instructions we're given are idiosyncratic—they only work with certain operands, such as requiring some integer literals to be 32-bit or 64-bit, depending on which instruction we're using. The programmer is forced to consider the size of their data before choosing

how to proceed with a computation.

These limitations make programming cumbersome and error-prone.

Instead, we should free the programmers (ourselves), eliminating cumbersomeness and removing error-prone programming patterns.

We should free the programmer to invent new locations at will if it helps their programming, and not worry about irrelevant machine-specific restrictions on instructions.

We design a new language, Asm-lang v2, to abstract away from these two machine-specific concerns.

```
      p ::= (module info tail)

   info ::= info?

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

The language no longer knows about registers, but instead presents an abstract location to the programmer. There are two assembly instructions—move a value to an abstract location, or perform a binary operation on an abstract location. These now act uniformly on their arguments. The left-hand-side is always the destination location, and the right-hand-side is an arbitrary trivial value.

4.3.3 Exposing Memory in Paren-x64

We first need to expose x64 features to access memory locations. In particular, we expose displacement mode operands for memory locations. The *displacement mode operand* is a new operand that can

appear in some location positions as the argument or an instruction. This allows accessing memory locations using pointer arithmetic. It is written as `QWORD [reg - int32]` or `QWORD [reg + int32]` in x64, where `reg` is a register holding some memory address and the `int32` is an offset number of bytes from that address to access, as a 32-bit integer. The keyword `QWORD`, which is an unintuitive spelling of "8 bytes", indicates that this operand is accessing 64 bits at a time.

> "Word" normally means the unit of addressing memory—64bits in our case. Unfortunately, in the past, the word size was different. In order to avoid backwards incompatibilty changes, tools that use `WORD` as a keyword, like `nasm`, didn't want to change it's meaning. Instead, the keyword `WORD` means 16-bits, not the word size, and prefixes give us multiple of that notion of `WORD`. So `QWORD` is `4` `WORD`s, or 64 bits, which is the word size on x64.

For example, if `rbp` holds a memory address, we can move the value `42` to that memory address using the instruction `mov QWORD [rbp - 0], 42`. We can move the value from memory into the register `rax` using the instruction `mov rax, QWORD [rbp - 0]`.

Our offsets will be multiples of 8. The offset is a number of bytes, and since we are dealing primarily with 64-bit values, we will increment pointers in values of 8. For example, the following snippet of code moves two values into memory, then pulls them out and adds them.

```
mov QWORD [rbp - 0], 21
mov QWORD [rbp - 8], 21
mov rax, QWORD [rbp - 8]
mov rbx, QWORD [rbp - 0]
add rax, rbx
```

These accesses grow *downwards*, subtracting from the base pointer rather than adding, following common conventions about how stack memory is used. This is an arbitrary choice, but we choose to follow the convention.

The new version of *Paren-x64 v2* is below.

$$
\begin{aligned}
p &::= \texttt{(begin } s \texttt{ ...)} \\
\\
s &::= \texttt{(set! } addr\ int32\texttt{)}^{+} \\
&\mid \texttt{(set! } addr\ reg\texttt{)}^{+} \\
&\mid \texttt{(set! } reg\ loc^{+}\ int64^{-}\texttt{)} \\
&\mid \texttt{(set! } reg\ triv^{+}\ reg^{-}\texttt{)}
\end{aligned}
$$

```
                | (set! reg_1 (binop reg_1 int32))
                | (set! reg_1 (binop reg_1 loc+ reg-))

        triv+ ::= reg
                | int64

         loc+ ::= reg
                | addr

          reg ::= rsp
                | rbp
                | rax
                | rbx
                | rcx
                | rdx
                | rsi
                | rdi
                | r8
                | r9
                | r10
                | r11
                | r12
                | r13
                | r14
                | r15

        addr+ ::= (fbp - dispoffset)

         fbp+ ::= frame-base-pointer-register?

        binop ::= *
                | +

        int64 ::= int64?

        int32 ::= int32?

  dispoffset+ ::= dispoffset?
```

We add the new nonterminal *addr* to the language, and add *addr* as a production to *loc*. The *addr* nonterminal represents a displacement mode operand to an instruction. We abstract the language over the *base frame pointer*, the pointer to the start of the current stack frame, which is stored in the parameter `current-frame-base-pointer-register` and is `rbp` by default. An *addr* may only be used with the `current-frame-base-pointer-register` as its first operand. The offset of each *addr* is restricted to be an integer that is divisible by 8, the

number of bytes in a machine word in x64. This ensures all memory accesses are machine-word aligned, meaning we leave space for all bytes in the word between each access. Note that the offset is *negative*; we access the stack backwards, following the x64 "stack grows down" convention.

All languages in our compiler assume that the uses of `current-frame-base-pointer-register` obey the *stack discipline*, defined below; all other uses are undefined behavior. Setting its value directly is forbidden. Pointer arithmetic, such as `(set! rbp (+ rbp opand))`, is allowed only when the *opand* is a `dispoffset?`. Incrementing the pointer beyond its initial value given by the run-time system is forbidden. We do not try to enforce these statically, since it may be impossible to do so in general. Instead, they must be undefined behavior.

> *Design digression:*
>
> The language is parameterized by the `current-frame-base-pointer-register`. Parameterizing the language this way lets us avoid committing to particular register choices, making the language inherently more machine agnostic. This is helpful in designing a compiler with multiple machine backends. A real compiler would want to support many machines, not just x64, and parameterizing the language makes this simpler. We could imagine retargeting a new machine by changing the value of `current-frame-base-pointer-register`, among other parameters. If our language definitions were sufficiently parameterized, few if any compiler passes would need to differ between target machines. This language is not suffiently abstract yet, but using parameterized languages in this way is a common tool we will use.

The `current-frame-base-pointer-register` is initialized by the run-time system. Implementing this will require a new run-time system to initialize the stack. From now on, the run-time system is provided by the `cpsc411/compiler-lib` library.

The run-time system provides a default stack of size 8 megabytes. This should be enough for now, but if it's not, you can use the `parameter?` `current-stack-size` to increase it.

The new run-time system also prints the value of `rax` to the screen, instead of returning it via an exit code, and does the work of converting numbers to ASCII strings. Now when you run your binary, you'll get a more familiar output. The `execute` function will parse this output into a Racket number. If you're interested in how this is done, you can read the definition of `x86-64-runtime`.

(**generate-x64** *p*) → (and/c string? x64-instructions?) procedure

p : paren-x64-v2?

Compile the Paren-x64 v2 program into a valid sequence of x64 instructions, represented as a string.

Examples:

```
> (require racket/pretty)
> (pretty-display
   (generate-x64 '(begin (set! rax 42))))
mov rax, 42
> (pretty-display
   (generate-x64 '(begin (set! rax 42) (set! rax (+ rax 0)))))
mov rax, 42
add rax, 0
> (pretty-display
   (generate-x64
   '(begin
      (set! (rbp - 0) 0)
      (set! (rbp - 8) 42)
      (set! rax (rbp - 0))
      (set! rax (+ rax (rbp - 8))))))
mov QWORD [rbp - 0], 0
mov QWORD [rbp - 8], 42
mov rax, QWORD [rbp - 0]
add rax, QWORD [rbp - 8]
> (pretty-display
   (generate-x64
   '(begin
      (set! rax 0)
      (set! rbx 0)
      (set! r9 42)
      (set! rax (+ rax r9)))))
mov rax, 0
mov rbx, 0
mov r9, 42
add rax, r9
> (current-pass-list
   (list
    check-paren-x64
    generate-x64
    wrap-x64-run-time
    wrap-x64-boilerplate))
> (execute '(begin (set! rax 42)))
```

```
42
> (execute '(begin (set! rax 42) (set! rax (+ rax 0))))
42
> (execute
   '(begin
      (set! (rbp - 0) 0)
      (set! (rbp - 8) 42)
      (set! rax (rbp - 0))
      (set! rax (+ rax (rbp - 8)))))
42
> (execute
   '(begin
      (set! rax 0)
      (set! rbx 0)
      (set! r9 42)
      (set! rax (+ rax r9))))
42
```

4.3.4 Abstracting the Machine

Now that we have effectively unlimited physical locations on the machine, we want to begin abstracting away from machine details.

The first thing we do is to abstract away from the displacement mode operand, introducing an abstract notion of *frame variable*, a variable that is located in a particular slot on the frame. This lets the programmer stop worrying about exactly what the displacement mode offset contraints are.

We define *Paren-x64-fvars v2* below.

```
      p ::= (begin s ...)

      s ::= (set! fvar+ addr- int32)
          | (set! fvar+ addr- reg)
          | (set! reg loc)
          | (set! reg triv)
          | (set! reg_1 (binop reg_1 int32))
          | (set! reg_1 (binop reg_1 loc))

   triv ::= reg
          | int64

    loc ::= reg
          | fvar+
```

```
              |  addr⁻

        reg ::= rsp
              |  rbp
              |  rax
              |  rbx
              |  rcx
              |  rdx
              |  rsi
              |  rdi
              |  r8
              |  r9
              |  r10
              |  r11
              |  r12
              |  r13
              |  r14
              |  r15

      addr⁻ ::= (fbp - dispoffset)

       fbp⁻ ::= frame-base-pointer-register?

      binop ::= *
              |  +

      int64 ::= int64?

      int32 ::= int32?

     fvar⁺ ::= fvar?

dispoffset⁻ ::= dispoffset?
```

We replace the notation of an address with that of an *fvar*, which represents a unique location on the frame, relative to the current value of `current-frame-base-pointer-register`. These are written as the symbol `fv` followed by a number indicating the slot on the frame. For example *fv1* is the frame variables indicating the first slot on the frame. Frame variables are distinct from labels, alocs, and rlocs. The number represents the index into the frame for the current function.

In Paren-x64-fvars v2, it is still undefined behaviour to violate stack discipline when using `current-frame-base-pointer-register`.

(**implement-fvars** *p*) → Paren-x64-v2.p procedure

p : Paren-x64-fvars-v2.p

Compile the Paren-x64-fvars v2 to Paren-x64 v2 by reifying *fvar*s into displacement mode operands. The pass should use `current-frame-base-pointer-register`.

This is a useful step toward an abstract assembly language, but we're still forced to remember odd restrictions on which kind of physical location each instruction takes as an operand. Our instructions in Paren x64-v2 are restricted by x64. For example, binary operations must use a register as their first operand.

We can create a new language which allows the user to ignore the differences between physical locations, enabling *any* instruction to work on any kind of physical locations. This way, the language is responsible for managing these annoying details instead of the programmer.

We do this by defining *Para-asm v2*, a kind of less finiky assembly language. We can think of this language as parameterized by the set of locations, hence the name.

```
        p ::= (begin effect+ s⁻ ... (halt triv)+)

  effect+ ::= (set! loc triv)
            | (set! loc_1 (binop loc_1 triv))

       s⁻ ::= (set! addr int32)
            | (set! addr reg)
            | (set! reg loc)
            | (set! reg triv)
            | (set! reg_1 (binop reg_1 int32))
            | (set! reg_1 (binop reg_1 loc))

     triv ::= reg⁻
            | int64
            | loc+

      loc ::= reg
            | fvar+
            | addr⁻

      reg ::= rsp
            | rbp
            | rax
```

```
                | rbx
                | rcx
                | rdx
                | rsi
                | rdi
                | r8
                | r9
                | r10⁻
                | r11⁻
                | r12
                | r13
                | r14
                | r15

        addr⁻ ::= (fbp - dispoffset)

         fbp⁻ ::= frame-base-pointer-register?

        binop ::= *
                | +

        int64 ::= int64?

        fvar⁺ ::= fvar?

       int32⁻ ::= int32?

  dispoffset⁻ ::= dispoffset?
```

The main difference is in the *s* non-terminal. Now, instructions can use an arbitrary kind of *loc* as their operands. The semantics are otherwise unchanged. `(set! loc triv)` moves the value of *triv* into *loc*, and `(set! loc_1 (+ loc_1 triv))` adds the values of *loc_1* and *triv*, storing the result in *loc_1*. Note that the two occurrences of *loc_1* in a binary operations are still required to be identical. We're not trying to lift all of the restrictions from x64 yet; we're doing *one thing well*.

We also add the *halt* instruction, which moves the the value of its operand into the `current-return-value-register` (`rax` by default). Adding this abstraction frees the user from remembering which register is used for this purpose. This instruction is valid only as the final instruction executed in a program, and it must be present.

Notice that two registers, `r10`, and `r11`, have been removed from *reg*. Para-asm v2 assumes control of these registers, forbidding the programmer from using them directly. Instead, the language

implementation will make use of this when compiling to Paren-x64 v2. These auxiliary registers are defined by the parameter `current-auxiliary-registers`.

Note that we do not have to restrict `rax`, despite the language making use of it.

Question: Why not?

> *Design digression:*
>
> If we were trying to support multiple backends, we might parameterize the language further by the set of unrestricted registers. This set would describe the registers that can be used in an arbitrary way by the program, and would exclude the set of `current-auxiliary-registers`. Each restricted register would have some invariants associated with it, similar to the stack discipline invariant for the `current-frame-base-pointer-register`.

`(patch-instructions p)` → `paren-x64-fvars-v2?` procedure
`p` : `para-asm-lang-v2?`

Compiles the Para-asm v2 to Paren-x64-fvars v2 by patching instructions that have no x64 analogue into a sequence of instructions. The implementation should use auxiliary registers from `current-patch-instructions-registers` when generating instruction sequences, and `current-return-value-register` for compiling *`halt`*.

Examples:

```
> (patch-instructions '(begin (set! rbx 42) (halt rbx)))
'(begin (set! rbx 42) (set! rax rbx))
> (patch-instructions
   '(begin
      (set! fv0 0)
      (set! fv1 42)
      (set! fv0 fv1)
      (halt fv0)))
'(begin
   (set! fv0 0)
   (set! fv1 42)
   (set! r10 fv1)
   (set! fv0 r10)
   (set! rax fv0))
```

```
> (patch-instructions
   '(begin
      (set! rbx 0)
      (set! rcx 0)
      (set! r9 42)
      (set! rbx rcx)
      (set! rbx (+ rbx r9))
      (halt rbx)))
'(begin
   (set! rbx 0)
   (set! rcx 0)
   (set! r9 42)
   (set! rbx rcx)
   (set! rbx (+ rbx r9))
   (set! rax rbx))
```

4.3.5 Nesting Instruction Sequences

One of the most restrictive limitations in our language is that expressions cannot be nested. Each program must be a linear sequence of instructions.

We can easily lift this restriction by designing a language that supports nesting, and compiling it using our well-known operations for composing instruction sequences. This simplifies the job of later passes that are now free to nest instructions if it's helpful.

Below, we design *nested-asm-lang-v2*.

```
     p+ ::= tail

  tail+ ::= (halt triv)+
          | p−
          | (begin effect ... tail+ (halt triv)−)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 triv))
          | (begin effect ... effect)+

   triv ::= int64
          | loc

    loc ::= reg
          | fvar

    reg ::= rsp
```

```
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

int64 ::= int64?

 fvar ::= fvar?
```

We add a *tail* production which loosely corresponds to the top-level program from Para-asm v2. However, these can be nested, before eventually ending in a *halt* instruction. The *tail* production represents the "tail", or last, computation in the program.

We also enable nesting in effect position. This essentially allows to copy and paste some instruction sequence into the middle of a program.

(**flatten-begins** *p*) → para-asm-lang-v2 procedure
  p : nested-asm-lang-v2

Flatten all nested `begin` expressions.

4.3.6 Abstracting Physical Locations

We are still required to think about *physical locations*—the registers and frame variables of x64. We don't usually care which location a value is stored in, so long as it is stored *somewhere*.

We can introduce an abstraction to capture this idea. We define an *abstract location* to be a globally unique name for some physical location.

We define *Asm-lang v2* below. Asm-lang v2 is an imperative, assembly-like language.

```
      p ::= (module info tail)+
          | tail-

   info+ ::= info?

    tail ::= (halt triv)
          | (begin effect ... tail)

  effect ::= (set! aloc+ loc- triv)
          | (set! aloc_1+ loc_1- (binop aloc_1+ loc_1- triv))
          | (begin effect ... effect)

    triv ::= int64
          | aloc+
          | loc-

    loc- ::= reg
          | fvar

    reg- ::= rsp
          | rbp
          | rax
          | rbx
          | rcx
          | rdx
          | rsi
          | rdi
          | r8
          | r9
          | r12
          | r13
          | r14
          | r15

   binop ::= *
          | +

   aloc+ ::= aloc?

   int64 ::= int64?

   fvar- ::= fvar?
```

In Asm-lang v2, we generalize instructions to work over abstract

location, *aloc*. An *aloc* is a symbol that is of the form `<name>.<number>`; this is captured by the `aloc?` predicate. Note that this overlaps with the definition of `fvar?`, but we will never be able to compare the two, so we don't feel a need to distinguish them. We assume all *aloc*s are globally unique, and any reference to the same *aloc* is to the same location—these are not the names yet.

Implementing Asm-lang v2 is a multi-step process. We gather up all the abstract locations, then assign them to physical locations.

For each step, we create an *administrative language*—an intermediate language whose semantics does not differ at all from its parent language, but whose syntax is potentially decorated with additional data that simplifies the next step of the compiler. We represent this additional data in an *info field*, an annotation in the program that serves only to store additional information for compilation. This additional information is often the result of some program analysis or preprocessing step that informs the next compiler pass. In the parent language, Asm-lang v2 in this case, the info field is unrestricted—it could contain anything at all. In fact, we can view the parent language as a family of languages, each differing in its info field.

We represent the info field as an association list of keys to proper list whose first element is the value of the key, such as `((key value))`. The module `cpsc411/info-lib` provides utilities for working with this representation. In general, we will only give a partial specification—it may contain arbitrary other key-value pairs, indicated by the `any/c` pattern in the definition. This lax specification is useful for debugging: you may leave residual *info* from earlier languages, or include your own debugging *info*. The *info* field is also unordered, so it should be accessed through the `cpsc411/info-lib` interface.

> *Design digression:*
>
> In a production compiler, we would probably not represent these administrative languages at all, but instead store the contents of the info field "on the side", as a separate data structure. This would prevent us from deconstructing and reconstructing the syntax tree when modifying or accessing the info field.
>
> However, directly representing the info field as part of the language has some advantages. It becomes simple to compose passes under a single interface. The program representation captures *all* of its invariants, and we do not need to consider an external data structure, making debugging, printing, and reading programs simpler.
>
> Furthermore, our representation of the info field is not particularly

> efficient. The proper list representation uses strictly more memory than necessary, and does not support random access. We could improve it by using improper lists, such as `((key . value))`, or perhaps a a hash table. However, this representation is simpler to read and write in program text.

First, we analyze the Asm-lang v2 to discover which abstract locations are in use. This is a straight-forward analysis, traversing the program and collecting any abstract location we see into a set.

We define the administrative language *Asm-lang v2/locals* (pronounced "Asm lang v2 with locals") to capture this information.

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...))))+
          | info?-

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

The only difference is in the info field, which now contains a *locals set*, a `set?` of all abstract locations used in the associated *`tail`*. In this language, there is an invariant that any abstract location that is used must appear in the locals set.

`(`**`uncover-locals`** *`p`*`)` → `asm-lang-v2/locals?` `procedure`
  `p` : `asm-lang-v2?`

Compiles Asm-lang v2 to Asm-lang v2/locals, analysing which abstract locations are used in the program and decorating the program with the set of variables in an info field.

Examples:

```
> (uncover-locals
   '(module ()
     (begin
       (set! x.1 0)
       (halt x.1))))
'(module ((locals (x.1))) (begin (set! x.1 0) (halt x.1)))
> (uncover-locals
   '(module ()
     (begin
       (set! x.1 0)
       (set! y.1 x.1)
       (set! y.1 (+ y.1 x.1))
       (halt y.1))))
'(module
   ((locals (x.1 y.1)))
   (begin (set! x.1 0) (set! y.1 x.1) (set! y.1 (+ y.1 x.1)) (halt y.1)))
```

Next, we assign homes for all the abstract locations. For now, our strategy is trivial: we assign each abstract location a fresh frame variable.

We capture this information in a new info field, described in *Asm-lang-v2/assignments*.

```
      p ::= (module info tail)

   info ::= (#:from-
             contract (info/c (locals (aloc ...)) (assignment
             ((aloc loc) ...))+))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

   loc+ ::= reg
          | fvar

   reg+ ::= rsp
          | rbp
          | rax
          | rbx
```

```
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

 aloc ::= aloc?

int64 ::= int64?

fvar+ ::= fvar?
```

The `assignment` info field records a mapping from each abstract location to some physical location. The language is more general than our implementation strategy; we allow an abstract location to be assigned to any valid physical location, including registers, to permit future optimizations or hand-written code.

(assign-fvars *p***)** → `asm-lang-v2/assignments?` procedure

p : `asm-lang-v2/locals?`

Compiles Asm-lang v2/locals to Asm-lang v2/assignments, by assigning each abstract location from the `locals` info field to a fresh frame variable.

Finally, we simply replace all the abstract locations with the physical location assigned by `assign-fvars`.

(replace-locations *p***)** → `nested-asm-lang-v2?` procedure

p : `asm-lang-v2/assignments?`

Compiles Asm-lang v2/assignments to Nested-asm-lang v2, replaced each abstract location with its assigned physical location from the `assignment` info field.

Examples:

```
> (replace-locations
    '(module ((locals (x.1)) (assignment ((x.1 rax))))
      (begin
        (set! x.1 0)
        (halt x.1))))
'(begin (set! rax 0) (halt rax))
> (replace-locations
    '(module ((locals (x.1 y.1 w.1))
              (assignment ((x.1 rax) (y.1 rbx) (w.1 r9))))
      (begin
        (set! x.1 0)
        (set! y.1 x.1)
        (set! w.1 (+ w.1 y.1))
        (halt w.1))))
'(begin (set! rax 0) (set! rbx rax) (set! r9 (+ r9 rbx)) (halt r9))
```

As these three passes implement a single true language, we wrap them up as a single pass.

(assign-homes *p*) → nested-asm-lang-v2? procedure

p : asm-lang-v2?

Compiles Asm-lang v2 to Nested-asm-lang v2, replacing each abstract location with a physical location.

4.3.7 Appendix: Overview

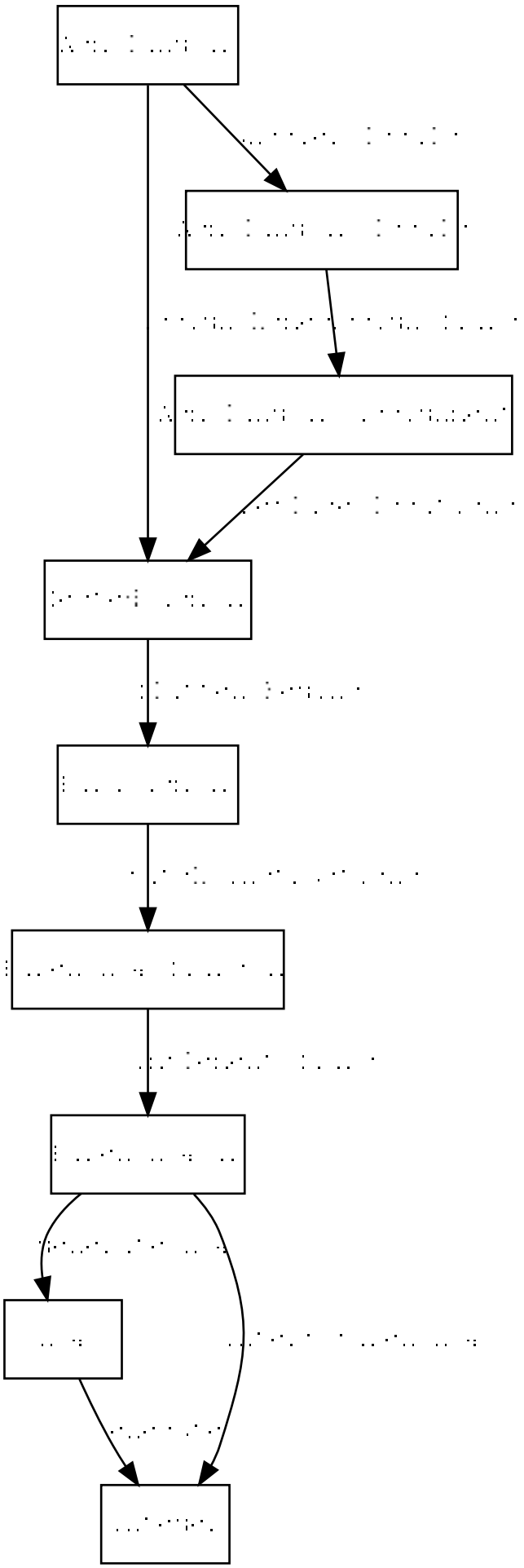


Figure 1: Overview of Compiler Version 2

4.3.8 Appendix: Language Definitions

`(asm-lang-v2? a) → boolean?` procedure

`a : any/c`

Decides whether `a` is a valid program in the `asm-lang-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2 : grammar?

```
      p ::= (module info tail)

   info ::= info?

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

`(asm-lang-v2/locals? a) → boolean?` procedure

`a : any/c`

Decides whether `a` is a valid program in the `asm-lang-v2/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/locals : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...))))
```

```
  tail ::= (halt triv)
         | (begin effect ... tail)

effect ::= (set! aloc triv)
         | (set! aloc_1 (binop aloc_1 triv))
         | (begin effect ... effect)

  triv ::= int64
         | aloc

 binop ::= *
         | +

  aloc ::= aloc?

 int64 ::= int64?
```

(**asm-lang-v2/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2/assignments` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/assignments : grammar?

```
    p ::= (module info tail)

 info ::= (#:from-
           contract (info/c (locals (aloc ...)) (assignment ((aloc loc) ...))))

  tail ::= (halt triv)
         | (begin effect ... tail)

effect ::= (set! aloc triv)
         | (set! aloc_1 (binop aloc_1 triv))
         | (begin effect ... effect)

  triv ::= int64
         | aloc

   loc ::= reg
         | fvar
```

```
 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r12
       | r13
       | r14
       | r15

binop ::= *
       | +

 aloc ::= aloc?

int64 ::= int64?

 fvar ::= fvar?
```

(**nested-asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the nested-asm-lang-v2 grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v2 : grammar?

```
    p ::= tail

 tail ::= (halt triv)
       | (begin effect ... tail)

effect ::= (set! loc triv)
       | (set! loc_1 (binop loc_1 triv))
       | (begin effect ... effect)

 triv ::= int64
       | loc
```

```
   loc ::= reg
         | fvar

   reg ::= rsp
         | rbp
         | rax
         | rbx
         | rcx
         | rdx
         | rsi
         | rdi
         | r8
         | r9
         | r12
         | r13
         | r14
         | r15

 binop ::= *
         | +

 int64 ::= int64?

  fvar ::= fvar?
```

(**para-asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the para-asm-lang-v2 grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v2 : grammar?

```
     p ::= (begin effect ... (halt triv))

effect ::= (set! loc triv)
         | (set! loc_1 (binop loc_1 triv))

  triv ::= int64
         | loc

   loc ::= reg
```

```
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 int64 ::= int64?

  fvar ::= fvar?
```

(**paren-x64-fvars-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-fvars-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-fvars-v2 : grammar?

```
    p ::= (begin s ...)

    s ::= (set! fvar int32)
       |  (set! fvar reg)
       |  (set! reg loc)
       |  (set! reg triv)
       |  (set! reg_1 (binop reg_1 int32))
       |  (set! reg_1 (binop reg_1 loc))

 triv ::= reg
```

```
        | int64

  loc ::= reg
        | fvar

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r10
        | r11
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +

int64 ::= int64?

int32 ::= int32?

 fvar ::= fvar?
```

(paren-x64-v2? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v2 : grammar?

```
p ::= (begin s ...)

s ::= (set! addr int32)
    | (set! addr reg)
    | (set! reg loc)
```

```
          |  (set! reg triv)
          |  (set! reg_1 (binop reg_1 int32))
          |  (set! reg_1 (binop reg_1 loc))

     triv ::= reg
          |  int64

      loc ::= reg
          |  addr

      reg ::= rsp
          |  rbp
          |  rax
          |  rbx
          |  rcx
          |  rdx
          |  rsi
          |  rdi
          |  r8
          |  r9
          |  r10
          |  r11
          |  r12
          |  r13
          |  r14
          |  r15

     addr ::= (fbp - dispoffset)

      fbp ::= frame-base-pointer-register?

    binop ::= *
          |  +

    int64 ::= int64?

    int32 ::= int32?

dispoffset ::= dispoffset?
```

4.4 Imperative Abstractions

4.4.1 Preface: What's wrong with our language?

In Asm-lang v2, which abstracted away from machine-specific details such as what physical locations exist and machine-specific restrictions on instructions. This is often a goal of compiler implementation—we want a more portable language to easily retarget to different machines.

However, the source language is still not something we want to program. In order to program, we, as Asm-lang v2 programmers, must manually keep track of where a value is located in order to perform computation on it. We must move values into particular locations before computing. For example, when doing arithmetic, we cannot simply write `(+ 2 2)`, *i.e.,* "Add the values 2 and 2". Instead, we must write "first move 2 into a location, then move 2 into a different location, now add the contents of the two locations".

We want to move towards a *value-oriented* language, *i.e.,* a language where operations consume and produce values directly, and away from *imperative language* that manipulates some underlying machine state that the programmer must always keep in mind and manually manipulate. This would free the programmer from keeping the state of the machine in mind at all times and manually manipulating it to perform computation.

To this end we design a new source language, Values-lang v3. Implementing this language is the goal for this chapter. The language is designed primarily to address the above limitations.

```
    p ::= (module tail)

 tail ::= value
        | (let ([x value] ...) tail)

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)

 triv ::= int64
        | x
```

```
    x ::= name?

binop ::= *
        |  +

int64 ::= int64?
```

We can see the language has changed significantly. The binary operations now act uniformly on *trivs*, which represent expressions that directly compute to values, rather than *alocs*, which represent locations on the machine where values are stored. Intermediate operations can be named using `let`, which binds an arbitrary number of independent computations to names.

A *name*, or *lexical identifier*, is a placeholder that is indistinguishable from the value of the expression it names. Like abstract locations, names can be created at will. Unlike abstract locations, names obey lexical binding, shadowing the same name if an existing name is reused but without overwriting the old value. For example, the following example uses the same names multiple times, but this doesn't overwrite any other use of the name.

(interp-values-lang *p*) → int64? procedure
p : values-lang-v3?

Interprets the Values-lang v3 program *p* as a value. For all *p*, the value of `(interp-values-lang p)` should equal to `(execute p)`.

Example:

```
> (interp-values-lang
   '(module
      (let ([x (let ([y 1]
                     [x 2])
                 (+ y x))])
        (let ([y (let ([x 3]) x)])
          (+ x y)))))
6
```

This gives the programmer the ability to name sub-computations in a non-imperative way. When we need some sub-computation, we are able to make up a brand new name without fear that we will erroneously overwrite some existing value; at worst, we locally shadow it. We also do not manually move values into locations, but rather

name them and let the language sort out where values live.

Some of the names for non-terminals are counter-intuitive. For example, *triv* for trivial values makes sense, but why is the expression `(+ 2 2)` a *value*? In this language, we use the name of a non-terminal to indicate in which *context* it is valid, not to merely describe the non-terminal's productions. `(+ 2 2)` is valid in any context that expects a value, since it computes to a value and is indistinguishable from a value by the operations of our language. One context that expects a value is the right-hand side of a `let` binding. Only a *value* or a `let` whose body is a *tail* is valid in tail context.

In Values-lang v3, the `let` expression implements a particular kind of lexcial binding. The same lexical identifier can be shadowed by nested `let` expressions, and nested `let` expressions can refer to the bindings of prior `let` expressions. However, in the *same* `let` expression, the binding are considered *paralell*—they do not shadow, and cannot refer to each other. This means we can freely reorder the bindings in a single `let` statement. This can be useful for optimization. However, it means we cannot allow duplicate bindings or allow reference to bindings in the same `let` statement. Otherwise, the ability to reorder could introduce undefined behaviour.

To guard against this undefined behaviour, we introduce a validator.

We must also check for reference to unbound variables, since these may be compiled into reference to uninitialized memory, and resulting in undefined behaviour.

`(check-values-lang p) → values-lang-v3?` procedure

`p : any/c`

Takes an arbitrary value and either returns it, if it is a valid Values-lang v3 program, or raises an error with a descriptive error message.

Examples:

```
> (check-values-lang
    '(module
       (let ([x 5]
             [y 6])
         x)))
'(module (let ((x 5) (y 6)) x))
```

```
> (check-values-lang
   '(module
      (let ([x 5]
            [y 6])
        (let ([y x])
          y))))
'(module (let ((x 5) (y 6)) (let ((y x)) y)))
> (check-values-lang
   '(module
      (let ([x 5]
            [y x])
        y)))
check-values-lang: Invalid triv; expected one of:
  something of the form int, but got 'x
  a bound name, got x in environment ()
> (check-values-lang
   '(module
      (let ([x 5]
            [x 6])
        x)))
check-values-lang: Invalid triv; expected one of:
  something of the form int, but got '(let ((x 5) (x 6)) x)
  a bound name, got (let ((x 5) (x 6)) x) in environment ()
> (check-values-lang
   '(module (let () 5)))
'(module (let () 5))
> (check-values-lang
    '(module (let () x)))
check-values-lang: Invalid triv; expected one of:
  something of the form int, but got 'x
  a bound name, got x in environment ()
```

4.4.2 Abstracting from Assembly

With Asm-lang v2, we abstracted all machine details—there are no more registers, no more odd machine-specific restrictions, just a location-oriented abstract assembly language.

The language is a still an extremely imperative, non-compositional language. We first add the ability to more easily compose imperative expression, before moving to composing non-imperative expressions representing values.

But the assembly language is still an (*extremely*) imperative machine language. The operations it provides are: (1) move a value to a

location (2) perform a binary operation on a location. This requires the programmer to always remember the values of abstract locations and manipulate this underlying state to program.

We design *Imp-cmf-lang v3*, an imperative language that allows expressing operations on values directly, and composing them by sequencing these computation through abstract location. This removes some amount of imperativity from the language. The programmer can reason about each primitive operation using only values, and needs only to think about the state of the machine when composing operations.

> Imp-cmf-lang v3 is in a variant of *a-normal form* (*ANF*), a syntactic form that restricts all operations to trivial values, and forbids nesting. It is roughly equivalence to other compiler intermdiate forms, such as static-single assignment.

```
    p ::= (module info⁻ tail)

 info⁻ ::= info?

  tail ::= value⁺
         | (halt triv)⁻
         | (begin effect ... tail)

value⁺ ::= triv
         | (binop triv triv)

effect ::= (set! aloc value⁺ triv⁻)
         | (set! aloc_1 (binop aloc_1 triv))⁻
         | (begin effect ... effect)

  triv ::= int64
         | aloc

 binop ::= *
         | +

  aloc ::= aloc?

 int64 ::= int64?
```

We add the *value* non-terminal to represent operations that produce values at run time. Now, `set!` simply assigns any value, or operation that produces a value, to an abstract location. At the top-level, a

program is a *tail*, which represents the last computation executed in a program. It is either an operation that produces a value, or a sequence of such operations composed by storing the result of the *value* in an intermediate abstract locations. The Imp-cmf-lang v3 program implicitly halts with the final value of the *tail*.

To implement this language, we compile each operation to a sequence of Asm-lang v2 instructions. This involves select inserting an explicit `halt` in the final value of the *tail*, and selecting instruction sequences to implement primitive operations, and possibly introducing auxiliary abstract locations.

(**select-instructions** *p*) → asm-lang-v2 procedure
p : imp-cmf-lang-v3

Compiles Imp-cmf-lang v3 to Asm-lang v2, selecting appropriate sequences of abstract assembly instructions to implement the operations of the source language.

Examples:

```
> (select-instructions '(module (+ 2 2)))
'(module () (begin (set! tmp.1 2) (set! tmp.1 (+ tmp.1 2)) (halt tmp.1)))
> (select-instructions
   '(module
      (begin (set! x.1 5) x.1)))
'(module () (begin (set! x.1 5) (halt x.1)))
> (select-instructions
   '(module
      (begin
        (set! x.1 (+ 2 2))
        x.1)))
'(module () (begin (set! x.1 2) (set! x.1 (+ x.1 2)) (halt x.1)))
> (select-instructions
   '(module
      (begin
        (set! x.1 2)
        (set! x.2 2)
        (+ x.1 x.2))))
'(module
   ()
   (begin
     (set! x.1 2)
     (set! x.2 2)
     (set! tmp.2 x.1)
```

```
      (set! tmp.2 (+ tmp.2 x.2))
      (halt tmp.2)))
```

Imp-cmf-lang v3 forces us to carefully linearize all effects at the top-level. This is an annoying detail that the language could manage instead.

Below, we design *Imp-mf-lang v3*, which allows nesting effects in most contexts.

```
      p ::= (module tail)

   tail ::= value
          | (begin effect ... tail)

  value ::= triv
          | (binop triv triv)
          | (begin effect ... value)+

 effect ::= (set! aloc value)
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

Design digression:

Imp-mf-lang v3 is in *monadic form* (*MF*), a syntactic form that allows composing operations that operate on values and have no side-effect (such as changing the value of an abstract location), but requires explicit sequencing any effectful operations. This allows additional nesting compared to ANF. Monadic form is often used in high-level functional languages to support reasoning about when side-effecting operations happen, and is also a useful compiler intermediate form for the same reason.

ANF *almost* corresponds to a *canonical form* for MF, a syntactic form in which equal programs (for some notion of equality) have the same representation. The form is canonical in the sense that there is one right way to represent every program. All ANF programs are also in MF, but not all MF programs are in ANF.

Writing transformations and optimizations over canonical forms is often

easier since we do not have to manually consider two equal programs as they have the same representation. Unfortunately, transformations over canonical forms are often tricky because the a transformation may not preserve canonicity. In the case of of MF and ANF, it is often easier to write the same optimization over MF, since MF frees the optimization from attempting to unnest operations. On the other hand, selecting instructions over ANF is simpler since assembly features no nesting, and ANF guarantees that no nesting exists.

Strictly speaking, our Imp-cmf-lang v3 is a variant of ANF that is not truely canonical, since we can nest *tail*s in two different ways: `(begin (begin (set! x.1 5)) x.1)` and `(begin (set! x.1 5) x.1)` are the same program. However, this little bit of non-canonicity simplifies the work of the compiler in some cases, and only complicates a single pass in the short-term—once we add new features, it will not complicate anything, since we will be forced to deal with the complication anyway.

This design choice is an example of where the compiler design, like most software design, benefits from *iteration*. While the bottom-up approach of building layers on abstraction is often beneficial, some design decisions may not be obvious until the software evolves.

`(canonicalize-bind p)` → `imp-cmf-lang-v3?` procedure
`p` : `imp-mf-lang-v3?`

Compiles Imp-mf-lang v3 to Imp-cmf-lang v3, pushing `set!` under `begin` so that the right-hand-side of each `set!` is simple value-producing operation. This canonicalizes Imp-mf-lang v3 with respect to the equations

`(set! aloc (begin effect_1 ... value))` = `(begin effect_1 ... (set! aloc value))`

4.4.3 Be, not Do

Now we want to abstract further, away from locations and focus on values and operations on values. This allows us to express computation as something that represents a value, not a sequence of operations to compute a value. That is, to declare what *is*, not instruct what *to do*. This is a move toward *declarative* programming.

Values-lang v3 is the final source language for this week. It abstracts away from operations on locations and instead has operations on values. It also include a feature for binding values to names for reuse. The value of a Values-lang v3 program is the final value of an

expression.

But dealing with names is hard, so we make a simplifying assumption. We assume that actually, someone has already resolved all names into abstract locations. This gives us the language *Values-unique-lang v3*, defined below.

```
     p ::= (module tail)

  tail ::= value
         | (let ([aloc value] ...) tail)

 value ::= triv
         | (binop triv triv)
         | (let ([aloc value] ...) value)

  triv ::= int64
         | aloc

 binop ::= *
         | +

  aloc ::= aloc?

 int64 ::= int64?
```

This language is similar to Imp-mf-lang v3, but uses `let` to compose operations rather than sequential, imperative `set!` instructions. These two operations have different semantics. All of the bindings in a single `let` are assumed to be independent of each other, and can happen in any order. They should not be thought of as sequential, but simply declarations that some bindings to some values exist. For example, the following two Values-unique-lang v3 programs are equivalent:

```
(let ([x.1 5] [y.2 6]) (+ x.1 y.2)) = (let ([y.2 6] [x.1 5]) (+ x.1 y.2))
```

However, the apparently similar Imp-mf-lang v3 programs are not, since `set!`s are required to happen in-order:

```
(begin (set! x.1 5) (set! y.2 6) (+ x.1 y.2)) ≠ (begin (set! y.2 6) (set! x.1 5) (+ x.1 y.2))
```

This means we can implement optimizations over Values-unique-lang v3 that are not possible in Imp-mf-lang v3, although no such optimizations are apparent yet.

(**optimize-let-bindings** *p*) → Values-unique-lang-v3.p procedure
p : Values-unique-lang-v3.p

Optimizes `let` bindings by reordering them to minimize or maximize some metric.

To implement Values-unique-lang v3, we must sequentalize `let`.

(**sequentialize-let** *p*) → imp-mf-lang-v3.p procedure
p : Values-unique-lang-v3.p

Compiles Values-unique-lang v3 to Imp-mf-lang v3 by picking a particular order to implement `let` expressions using `set!`.

Finally, we must discharge our assumption that all names are unique. Below we define *Values-lang v3*, a value-oriented language with simple binary expressions and lexical binding.

```
    p ::= (module tail)

 tail ::= value
        | (let ([x+ aloc- value] ...) tail)

value ::= triv
        | (binop triv triv)
        | (let ([x+ aloc- value] ...) value)

 triv ::= int64
        | x+
        | aloc-

   x+ ::= name?

binop ::= *
        | +

aloc- ::= aloc?

int64 ::= int64?
```

x is a short-hand nonterminal for names, which are arbitrary symbols. To ensure transform names into abstract location, it suffices to append a unique number to each.

To implement this language, we must resolve all lexical binding and replace names by abstract locations

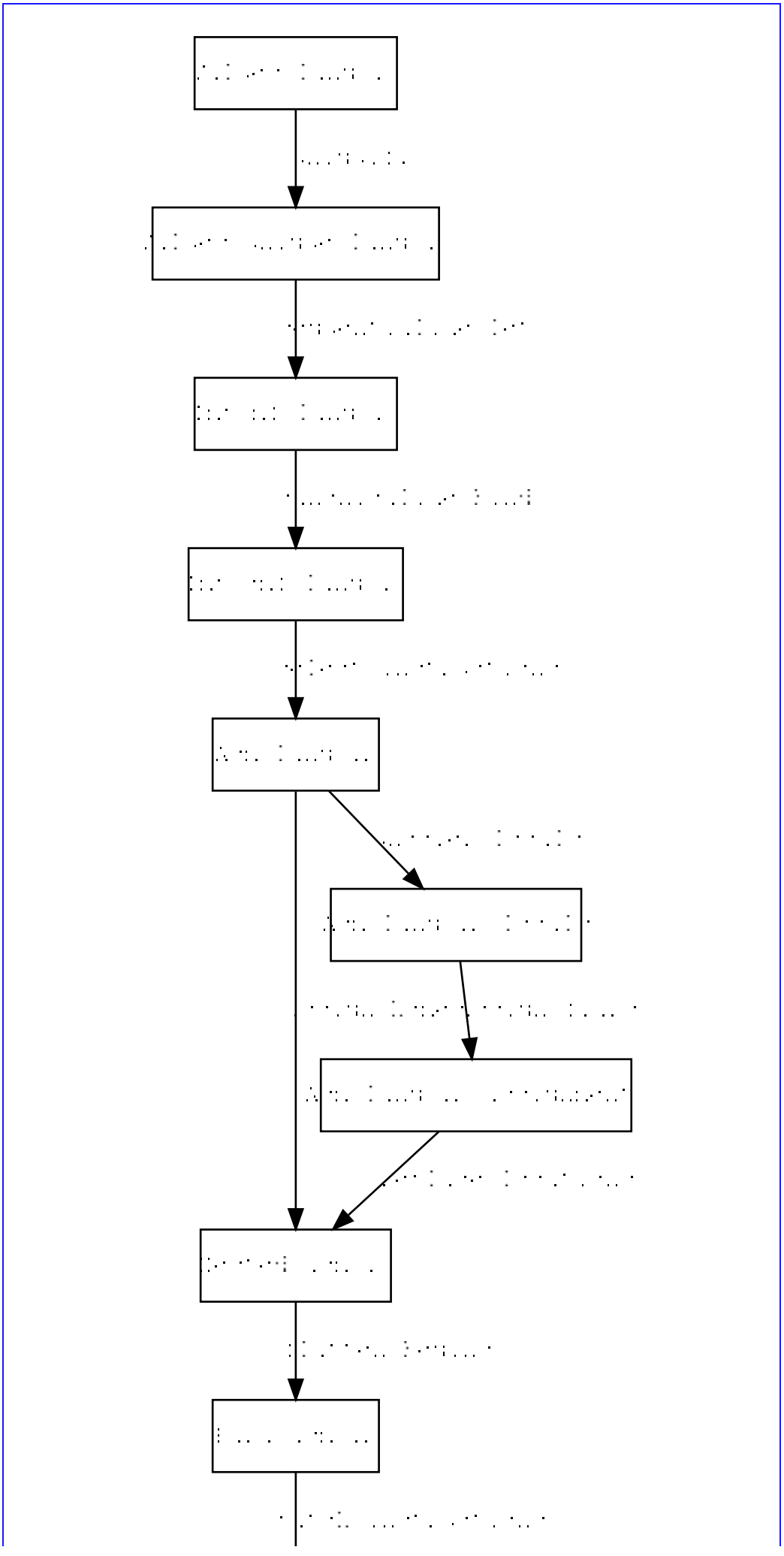
(**uniquify** *p*) → Values-unique-lang-v3.p procedure
  p : Values-lang-v3.p

Compiles Values-lang v3 to Values-unique-lang v3 by resolving all lexical identifiers to abstract locations.

Examples:

```
> (uniquify '(module (+ 2 2)))
'(module (+ 2 2))
> (uniquify
   '(module
      (let ([x 5])
        x)))
'(module (let ((x.3 5)) x.3))
> (uniquify
   '(module
      (let ([x (+ 2 2)])
        x)))
'(module (let ((x.4 (+ 2 2))) x.4))
> (uniquify
   '(module
      (let ([x 2])
        (let ([y 2])
          (+ x y)))))
'(module (let ((x.5 2)) (let ((y.6 2)) (+ x.5 y.6))))
> (uniquify
   '(module
      (let ([x 2])
        (let ([x 2])
          (+ x x)))))
'(module (let ((x.7 2)) (let ((x.8 2)) (+ x.8 x.8))))
```

4.4.4 Appendix: Overview



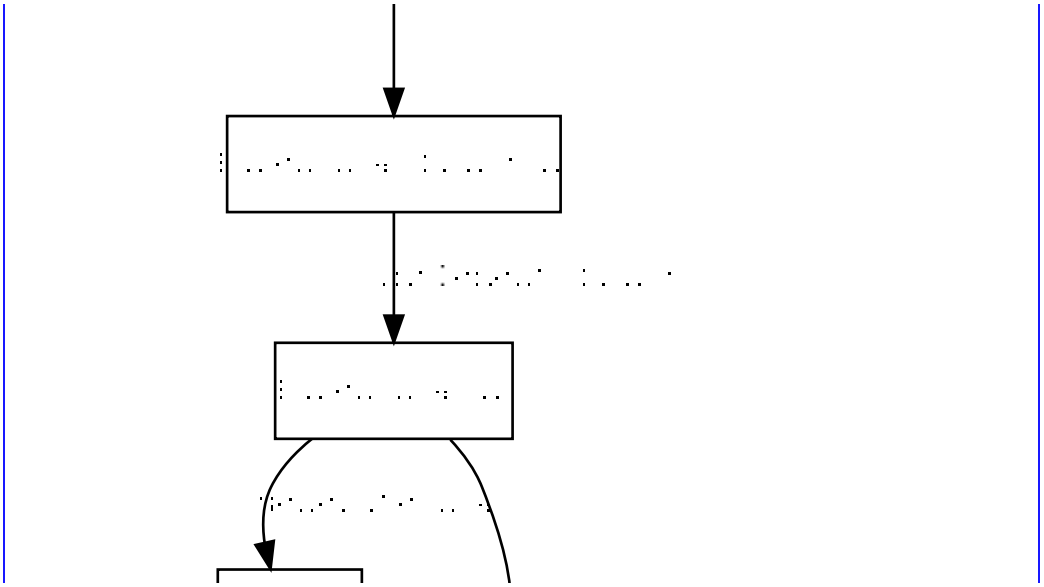


Figure 2: Overview of Compiler Version 3

4.4.5 Appendix: Language Definitions

(**values-lang-v3?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-lang-v3` grammar. The first non-terminal in the grammar defines valid programs.

values-lang-v3 : grammar?

```
    p ::= (module tail)

 tail ::= value
        | (let ([x value] ...) tail)

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)

 triv ::= int64
        | x
```

```
    x ::= name?

binop ::= *
        |  +

int64 ::= int64?
```

(**values-unique-lang-v3?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-unique-lang-v3` grammar. The first non-terminal in the grammar defines valid programs.

---

values-unique-lang-v3 : grammar?

```
    p ::= (module tail)

 tail ::= value
        |  (let ([aloc value] ...) tail)

value ::= triv
        |  (binop triv triv)
        |  (let ([aloc value] ...) value)

 triv ::= int64
        |  aloc

binop ::= *
        |  +

 aloc ::= aloc?

int64 ::= int64?
```

(**imp-mf-lang-v3?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v3` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v3 : grammar?

```
      p ::= (module tail)

   tail ::= value
          | (begin effect ... tail)

  value ::= triv
          | (binop triv triv)
          | (begin effect ... value)

 effect ::= (set! aloc value)
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(imp-cmf-lang-v3? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v3` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v3 : grammar?

```
      p ::= (module tail)

   tail ::= value
          | (begin effect ... tail)

  value ::= triv
          | (binop triv triv)

 effect ::= (set! aloc value)
          | (begin effect ... effect)

   triv ::= int64
          | aloc
```

```
  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(**asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2 : grammar?

```
      p ::= (module info tail)

   info ::= info?

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(**asm-lang-v2/locals?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2/locals` grammar. The first non-terminal in the grammar defines valid programs.

asm-lang-v2/locals : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...))))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(**asm-lang-v2/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-lang-v2/assignments grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/assignments : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-
             contract (info/c (locals (aloc ...)) (assignment ((aloc loc) ...))))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
```

```
        | aloc

   loc ::= reg
        | fvar

   reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

 binop ::= *
        | +

  aloc ::= aloc?

 int64 ::= int64?

  fvar ::= fvar?
```

(**nested-asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v2 : grammar?

```
     p ::= tail

  tail ::= (halt triv)
        | (begin effect ... tail)

effect ::= (set! loc triv)
```

```
        |  (set! loc_1 (binop loc_1 triv))
        |  (begin effect ... effect)

  triv ::= int64
        |  loc

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 int64 ::= int64?

  fvar ::= fvar?
```

(**para-asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the para-asm-lang-v2 grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v2 : grammar?

```
      p ::= (begin effect ... (halt triv))

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 triv))
```

```
triv ::= int64
       | loc

 loc ::= reg
       | fvar

 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r12
       | r13
       | r14
       | r15

binop ::= *
       | +

int64 ::= int64?

 fvar ::= fvar?
```

(**paren-x64-fvars-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-fvars-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-fvars-v2 : grammar?

```
  p ::= (begin s ...)

  s ::= (set! fvar int32)
      | (set! fvar reg)
      | (set! reg loc)
      | (set! reg triv)
```

```
        |  (set! reg_1 (binop reg_1 int32))
        |  (set! reg_1 (binop reg_1 loc))

  triv ::= reg
        |  int64

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r10
        |  r11
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 int64 ::= int64?

 int32 ::= int32?

  fvar ::= fvar?
```

(**paren-x64-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the paren-x64-v2 grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v2 : grammar?

```
p ::= (begin s ...)
```

```
          s ::= (set! addr int32)
              | (set! addr reg)
              | (set! reg loc)
              | (set! reg triv)
              | (set! reg_1 (binop reg_1 int32))
              | (set! reg_1 (binop reg_1 loc))

       triv ::= reg
              | int64

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
              | r9
              | r10
              | r11
              | r12
              | r13
              | r14
              | r15

       addr ::= (fbp - dispoffset)

        fbp ::= frame-base-pointer-register?

      binop ::= *
              | +

      int64 ::= int64?

      int32 ::= int32?

 dispoffset ::= dispoffset?
```

4.5 Register Allocation

4.5.1 Preface: What's wrong with our language?

With Asm-lang v2, we introduced abstract locations to free the programmer from thinking about physical locations. Unfortunately, our implementation strategy has a severe limitation. While it's simple and works, it's extremely slow! Each and every variable assignment or reference accesses memory. While memory accesses have improved a lot compared to old computers due to caching, accessing memory are still orders of magnitude slower than accessing a register when our variable is not in the cache (and, in general, it won't be in cache). Our compiler will have better performance if we help the machine out by using registers as much as possible.

Assigning variables to registers automatically is a non-trivial task. There's a reason we started out by compiling to frame variables. There are infinitely many frame variables, but only 16 registers. To assign an abstract location a new frame variable is trivial—just pick a new one, there's always new one. It's a simple, systematic algorithm. To assign an abstract location to a register, however, we need to pick a register that isn't currently in use. Since there's only 16 registers, we very quickly run out of registers... unless we're clever.

In general, being clever should be a last resort.

4.5.2 Optimizing Location Assignment

Conceptually, register allocation is a simple idea.

- Undead analysis: figure out which abstract locations might still be needed after each instruction.
- Conflict analysis: figure out which abstract locations cannot be assigned to the same register.
- Register allocation: assign each abstract locations to a register that is different from any conflicting abstract locations.
- Spilling: if we fail to find a register for an abstract location, put it in a frame variable.

Our Asm-lang v2 compiler is already a trivial implementation of this algorithm. It assumes every abstract location might be needed forever, and is in conflict with every other abstract location. It doesn't bother trying, and thus fails, to

find a register for every abstract location, and so spills everything to the frame. But we want to optimize this process.

In general, we will never do a perfect job, due to Rice's Theorem. We want to try to do a better job. Well, not better—we can never decide what is "better" due to Rice's Theorem. But we'd like to produce code that runs faster, and are willing to trade a more complex and slower compiler for it.

> Rice's Theorem is a huge buzz kill. It tells us, formally, that we cannot decide anything "interesting" about any Turing-complete program. Formally, "interesting" is defined as any property that is not either true for every program, or false for every program. "Decide" means write a program that returns either true or false on all programs. The halting program is an instance of Rice's Theorem. Rice's Theorem tells us we will *never* be able to do a perfect job—not when optimizing, not when analyzing, not when asking whether two programs are equivalent, not when deciding which program is "better". We will only be able to make trade-offs.

Since register allocation is a modification of our existing abstract location assignment strategy, we already have a source and target language in mind. We start from an existing language Asm-lang v2/locals, which is reproduced below:

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...))))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

This language, the output of `uncover-locals`, records which abstract locations are referenced in the program and thus need to be assigned physical locations. The register allocator takes over from here to perform that assignment.

And we use Asm-lang v2/assignments, reproduced below.

```
      p ::= (module info tail)
```

```
  info ::= (#:from-
            contract (info/c (locals (aloc ...)) (assignment ((aloc loc) ...))))

  tail ::= (halt triv)
         | (begin effect ... tail)

effect ::= (set! aloc triv)
         | (set! aloc_1 (binop aloc_1 triv))
         | (begin effect ... effect)

  triv ::= int64
         | aloc

   loc ::= reg
         | fvar

   reg ::= rsp
         | rbp
         | rax
         | rbx
         | rcx
         | rdx
         | rsi
         | rdi
         | r8
         | r9
         | r12
         | r13
         | r14
         | r15

 binop ::= *
         | +

  aloc ::= aloc?

 int64 ::= int64?

  fvar ::= fvar?
```

After this, our existing pass `replace-locations` should handle replacing abstract locations by the assigned physical locations.

An optimization should be seen as a drop-in replacement for some existing functionality, but one that performs "better" (for some definition of "better"). We will replace `assign-homes` with `assign-homes-opt`.

(assign-homes-opt *p*) → `nested-asm-lang-v2?` procedure

p : `asm-lang-v2?`

Compiles Asm-lang v2 to Nested-asm-lang v2, replacing each abstract

location with a physical location. This version performs graph-colouring register allocation.

4.5.3 Undeadness Analysis

We begin the register allocation by figuring out which locations might still be needed after each instruction. This process is often called liveness analysis, but we refer to it as undeadness analysis for reasons we explain shortly.

A variable (either abstract location or physical location) with a *particular* value that *definitely* will be used is considered *live*, and any variable (with a particular value) that *definitely* won't be used is considered *dead*. Recall that due to Rice's Theorem, we know it's generally impossible to decide whether a variable is dead or alive. This means that when writing an analysis, we must assume partial knowledge. The result is that we ignore liveness entirely.

> Asm-lang v2/locals is a simple enough language that we can tell whether a variable is dead or alive. Later, when we add new instructions, we will modify the undead analysis and find variables that aren't necessarily live or dead, and must be assumed to be undead. This is also necessary if we want to handle linking, or separate compilation.

We cannot in general decide the value of a variable at a given instruction. Instead, we focus on analyzing each *assignment* to a variable, which changes the value of the variable.

We assume that any variable that gets used, or might get used, might *not* be dead, *i.e.,* we assume it is *undead*, and consider a variable dead only when we have conclusive proof—like witnessing an instruction driving a new value through its heart, er, storing a new value in its location.

We collect the undead variables into sets. The *undead-out set* of an instruction is a set of variables that are undead after executing that instruction.

> Most compilers call these live-out or live-after sets. This suggests that variables are definitely alive, and that the analysis is computing sets of variables that are definitely alive, neither of which is true. Undead is not exactly the same as not-definitely-dead, except in horror movies and video games, but it's more suggestive of reality for compilers.

To determine whether a variable is in the undead-out set for an instruction, we analyze the program by looping over the instruction sequence backwards, starting with the last statement. The loop takes an instruction and its undead-out set. We analyze each instruction with its undead-out set and compute an

> We don't *need* to analyze the program backwards, but it's faster.

undead-in set for the instruction, which is the same as the undead-out set of the preceding instruction. That is, the undead-in set for an instruction `s_i` is the undead-out set for the instruction `s_{i-1}` in the program.

Each iteration of the loop performs the following analysis on a particular instruction. We start by assuming the undead-in set is the same as the undead-out set, then update it depending on what happens in the instruction. If a variable is *used* in the instruction, it *ought to be* live—we don't actually know, since that instruction's result itself might not be used, but the variable is at least acting like its live—and is added to the undead-in set. If a variable is *assigned*, *i.e.,* its value is overwritten, in the instruction, it is *definitely* dead at that point, and we remove it from the undead-in set.

To start the loop, this algorithm requires a default undead-out set for the last instruction; the default undead-out set for Asm-lang v2/locals is empty. In general, the default set may not be empty, because we may assume that some values are live after the program. For example, in Paren-x64 v1, we assume `rax` is live out.

This algorithm creates the undead-out sets for each instruction so that later passes can associate each instruction with its undead-out set. There are many ways to associate the undead-out sets with instructions. A simple way is to create a data structure that maps each set to an instruction.

Since our programs are trees of instructions, we represent the undead-out sets for each instruction as a tree of undead-out sets. We define the data undead-set tree to mirror the structure of Asm-lang v2 programs. An *undead-set tree* is either:

- an undead-out set `(aloc ...)`, corresponding to a single instruction such as `(halt triv)`
- or a list of undead-set trees, corresponding to the undead-set trees `(undead-set-tree?_1 ... undead-set-tree?_2)` corresponding to a `begin` statement `(begin effect_1 ... effect_2)` The first element of the list represents undead-set tree for the first `effect`, the second element represents the undead-set tree for the second `effect`, and so on.

An undead-set tree `(undead-set-tree? ...)` together with a list of instructions `(effect ...)` can be traversed together using the template for two trees simultaneously. This is similar to traversing two lists simultaneously:

```
(define (fn-for-s-and-undead-outs ss undead-outs)
  (match (cons ss undead-outs)
    [(cons '() '())
     (... case-for-empty ...)]
    [(cons `(,s ,rest-ss ...) `(,undead-out ,rest-undead-outs ...))
     (... (fn-for-s-and-undead-out s undead-out)
          (fn-for-ss-and-undead-outs rest-ss rest-undead-outs))]))
```

You'll need to design the template for traversing *tail* and *effect* trees simultaneously with an undead-set tree yourself.

To describe the output of the analysis, we define a new administrative language. We collect the undead-set tree a new *info* field. Below, we define *Asm-lang v2/undead*. The only change compared to Asm-lang v2/locals is in the *info* field, so we typeset the difference in the *info* field.

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-
            out undead-set-tree?)+))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(undead-analysis *p***)** → asm-lang-v2/undead? procedure
p : asm-lang-v2/locals?

Performs undeadness analysis, decorating the program with undead-set tree. Only the info field of the program is modified.

Examples:

```
> (undead-analysis
   '(module ((locals (x.1)))
      (begin
        (set! x.1 42)
        (halt x.1))))
'(module
   ((locals (x.1)) (undead-out ((x.1) ())))
   (begin (set! x.1 42) (halt x.1)))
> (undead-analysis
   '(module ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1)))
      (begin
        (set! v.1 1)
        (set! w.2 46)
```

```
        (set! x.3 v.1)
        (set! p.1 7)
        (set! x.3 (+ x.3 p.1))
        (set! y.4 x.3)
        (set! p.1 4)
        (set! y.4 (+ y.4 p.1))
        (set! z.5 x.3)
        (set! z.5 (+ z.5 w.2))
        (set! t.6 y.4)
        (set! p.1 -1)
        (set! t.6 (* t.6 p.1))
        (set! z.5 (+ z.5 t.6))
        (halt z.5))))
'(module
   ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1))
    (undead-out
     ((v.1)
      (v.1 w.2)
      (x.3 w.2)
      (p.1 x.3 w.2)
      (x.3 w.2)
      (y.4 x.3 w.2)
      (p.1 y.4 x.3 w.2)
      (x.3 w.2 y.4)
      (w.2 z.5 y.4)
      (y.4 z.5)
      (t.6 z.5)
      (p.1 t.6 z.5)
      (t.6 z.5)
      (z.5)
      ())))
   (begin
     (set! v.1 1)
     (set! w.2 46)
     (set! x.3 v.1)
     (set! p.1 7)
     (set! x.3 (+ x.3 p.1))
     (set! y.4 x.3)
     (set! p.1 4)
     (set! y.4 (+ y.4 p.1))
     (set! z.5 x.3)
     (set! z.5 (+ z.5 w.2))
     (set! t.6 y.4)
     (set! p.1 -1)
     (set! t.6 (* t.6 p.1))
     (set! z.5 (+ z.5 t.6))
     (halt z.5)))
```

An important corner case to consider is what happens when unused variables appear in a program.

Examples:

```
> (undead-analysis
   '(module ((locals (x.1 y.1)))
      (begin
        (set! y.1 42)
        (set! x.1 5)
        (halt x.1))))
'(module
   ((locals (x.1 y.1)) (undead-out (() (x.1) ())))
   (begin (set! y.1 42) (set! x.1 5) (halt x.1)))
> (undead-analysis
   '(module ((locals (x.1 y.1)))
      (begin
        (set! x.1 5)
        (set! y.1 42)
        (halt x.1))))
'(module
   ((locals (x.1 y.1)) (undead-out ((x.1) (x.1) ())))
   (begin (set! x.1 5) (set! y.1 42) (halt x.1)))
```

In a realistic compiler, unused variables should be removed be an optimization.

4.5.4 Conflict Analysis

To assign abstract locations to physical locations efficiently, we need to know when any two variables are *in conflict*, *i.e.,* cannot be assigned to the same physical location because different values might be in those variables at the same time.

We start by defining what a conflict is precisely.

True Definition of Conflict: Any variable that gets a new value during an instruction is *in conflict* with every variable that (1) has a different value at the same time and (2) will still be used after that instruction.

Unfortunately, due to Rice's Theorem, we cannot decide either property. We cannot figure out the value of every variable before run time; if we could, compiling would not be necessary. We also do not know which variables are live, only which are undead. Therefore, we can only approximate conflicts.

To approximate conflicts, we ignore values, and once more focus on assignments to variables. An assignment to the variable means the it *might* take on a new value that *might* be different from the value of any variable which *might* be live at that point. We have already approximated liveness via undead-out sets, so what remains is to approximate when a variable takes on a new value. Below, we describe how to approximate conflcits and slightly refine these criteria.

We represent conflicts in a data structure called a *conflict graph*. Interpreted as an undirected graph, the variables are represented as nodes (also known as vertexes), and conflicts between variables are represented as an edge from the variable to each of the variables in the associated set of conflicts. If there is an edge between any two nodes, then they are in conflict. Interpreted as a dictionary, the conflict graph maps each variable to a set of variables with which it is in conflict.

We create a conflict graph from the undead-out sets as follows. Any variable that is *defined* during an instruction is in conflict with every variable (except itself) in the undead-out set associated with that instruction. For example, the variable `x.1` is defined in `(set! x.1 (+ x.1 x.2))`, while the variables `x.2` and `x.1` are referenced. No variable can conflict with itself, since it always has the same value as itself. We approximate the values of variables by assuming each *definition* assigns a new unique value to the variable, and by assuming each variable in the undead-out set also has a unique value. This approximation tells us that `x.1` cannot be assigned the same physical location as any other variable in the undead-out set. If `x.2` is undead-out at this instruction, and we try to put `x.1` in the same physical location as `x.2`, then the value of `x.2` would be overwritten by the value of `(+ x.1 x.2)`.

We can reduce the number of conflicts, and thus possibly fit more variables into registers, by observing that one instruction does tell us that two values will be the same. A move instruction, such as `(set! x.1 x.2)`, is an instruction that simply defines the value of one variable to be that of another.

Approximation of Conflict

- Any variable *defined* during a non-move instruction is in conflict with every variable (except itself) in the undead-out set associated with the instruction.
- Any variable *defined* during a move instruction is in conflict with every variable in the undead-out set associated with the instruction, except itself and the variable referenced in the move.

To implement conflict analysis, we design the language *Asm-lang v2/conflicts* to capture the conflict graph. We typeset the difference compared to Asm-lang v2/undead.

```
info ::= (#:from-contract (info/c (locals (aloc ...)) (conflicts
         ((aloc (aloc ...)) ...))+ (undead-out undead-set-
         tree?)−))
```

The `info` field is extended with a conflict graph, represented as an association list from a variable to undead-out sets.

As in Asm-lang v2/undead, the `info` field also contains a declaration of the abstract locations that may be used in the program, and (as usual) possibly other non-required but useful information.

To implement conflict analysis, we simultaneously traverse a program with its undead-set tree, and analyze each instruction according to the approxiate conflict definition above. We start with a graph that initially contains a node for every abstract location in the `locals` set, and extend the graph with conflicts as we discover them.

`(conflict-analysis p) → asm-lang-v2/conflicts?` `procedure`
`p : asm-lang-v2/undead?`

Decorates a program with its conflict graph.

Examples:

```
> (conflict-analysis
   '(module ((locals (x.1))
             (undead-out ((x.1) ())))
      (begin
        (set! x.1 42)
        (halt x.1))))
'(module
   ((locals (x.1)) (conflicts ((x.1 ()))))
   (begin (set! x.1 42) (halt x.1)))
> (conflict-analysis
   '(module ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1))
             (undead-out
              ((v.1)
               (v.1 w.2)
               (w.2 x.3)
               (p.1 w.2 x.3)
               (w.2 x.3)
               (y.4 w.2 x.3)
               (p.1 y.4 w.2 x.3)
               (y.4 w.2 x.3)
               (z.5 y.4 w.2)
               (z.5 y.4)
               (t.6 z.5)
               (t.6 z.5 p.1)
               (t.6 z.5)
               (z.5)
               ())))
      (begin
        (set! v.1 1)
        (set! w.2 46)
        (set! x.3 v.1)
        (set! p.1 7)
        (set! x.3 (+ x.3 p.1))
        (set! y.4 x.3)
        (set! p.1 4)
        (set! y.4 (+ y.4 p.1))
```

```
        (set! z.5 x.3)
        (set! z.5 (+ z.5 w.2))
        (set! t.6 y.4)
        (set! p.1 -1)
        (set! t.6 (* t.6 p.1))
        (set! z.5 (+ z.5 t.6))
        (halt z.5))))
'(module
   ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1))
    (conflicts
     ((p.1 (z.5 t.6 y.4 x.3 w.2))
      (t.6 (p.1 z.5))
      (z.5 (p.1 t.6 w.2 y.4))
      (y.4 (z.5 x.3 p.1 w.2))
      (x.3 (y.4 p.1 w.2))
      (w.2 (z.5 y.4 p.1 x.3 v.1))
      (v.1 (w.2)))))
   (begin
     (set! v.1 1)
     (set! w.2 46)
     (set! x.3 v.1)
     (set! p.1 7)
     (set! x.3 (+ x.3 p.1))
     (set! y.4 x.3)
     (set! p.1 4)
     (set! y.4 (+ y.4 p.1))
     (set! z.5 x.3)
     (set! z.5 (+ z.5 w.2))
     (set! t.6 y.4)
     (set! p.1 -1)
     (set! t.6 (* t.6 p.1))
     (set! z.5 (+ z.5 t.6))
     (halt z.5)))
```

4.5.5 Register Allocation

Register allocation, as in the step that actually assigns abstract locations to physical locations, takes the set of abstract locations to assign homes, the conflict graph, and some set of assignable registers, and tries to assign the most abstract locations to registers. As usual, Rice's Theorem tells us we'll never be able to decide the maximal number of variables we can fit in registers. We'll have to approximate.

We'll use graph-colouring register allocation, an algorithm that is quadratic (and slows compile time down quite a lot), but usually assigns more abstract locations to registers than other faster algorithms.

> It's worth noting that since this core pass is quadratic, compile time is

> dominated by this single pass. One might be tempted to try to fuse many of our small compiler passes to save compile time, but most of our compile passes are linear, and have essentially no effect on compile time compared to graph-colouring register allocation.

The core algorithm has a straight-forward recursive description. We recur over the set of `locals` and produce an *assignment*, *i.e.,* a dictionary mapping abstract locations to physical locations.

- If the set of abstract locations is empty, return the empty assignment.
- Otherwise, choose a low-degree abstract location from the input set of abstract locations, if one exists. Otherwise, pick an arbitrary abstract location from the set.

  A *low-degree* abstract location is one with fewer than `k` conflicts, for some for pre-defined `k`. We pick `k` to be the number of registers in the set of assignable registers.
- Recur with the chosen abstract location removed from the input set and the conflict graph. The recursive call should return an assignment for all the remaining abstract locations.
- Attempt to select a register for the chosen abstract location. You cannot select registers to which conflicting abstract locations were assigned by the recursive call. This attempt succeeds if a low-degree abstract location was chosen, and *might* fail otherwise (but it depends on which registers got allocated in the recursive call).
  - If you succeed in selecting a register, then add the assignment for the chosen abstract location to the result of the recursive call.
  - Otherwise, we cannot assign the choosen abstract location to a register. Instead, we *spill it*, *i.e.,* we assign it a fresh frame variable.

> This algorithm is due to R. Kent Dybvig, itself an adaptation of the optimistic register allocation described in "Improvements to graph coloring register allocation" (ACM TOPLAS 6:3, 1994) by Preston Briggs, et al.

We can simplify the implementation of this algorithm by separting it into two parts: first, sort all abstract locations in degree order, then assign each register in sorted order.

To describe the output of the register allocator, we reuse Asm-lang v2/assignments. Below, we typeset the changes compared to Asm-lang v2/conflicts. Note only the *`info`* field changes.

```
info ::= (#:from-
          contract (info/c (locals (aloc ...)) (assignment+ conflicts- ((aloc loc+ (aloc
```

```
      ...)⁻) ...))))
```

(**assign-registers** *p*) → asm-lang-v2/assignments? procedure
  p : asm-lang-v2/conflicts

Performs graph-colouring register allocation. The pass attempts to fit each of the abstract location declared in the locals set into a register, and if one cannot be found, assigns it a frame variable instead.

Examples:

```
> (assign-registers
   '(module ((locals (x.1))
             (conflicts ((x.1 ()))))
     (begin
       (set! x.1 42)
       (halt x.1))))
'(module
   ((locals (x.1)) (conflicts ((x.1 ()))) (assignment ((x.1 r15))))
   (begin (set! x.1 42) (halt x.1)))
> (parameterize ([current-assignable-registers '(r9)])
    (assign-registers
     '(module ((locals (x.1))
               (conflicts ((x.1 ()))))
       (begin
         (set! x.1 42)
         (halt x.1)))))
'(module
   ((locals (x.1)) (conflicts ((x.1 ()))) (assignment ((x.1 r9))))
   (begin (set! x.1 42) (halt x.1)))
> (parameterize ([current-assignable-registers '()])
    (assign-registers
     '(module ((locals (x.1))
               (conflicts ((x.1 ()))))
       (begin
         (set! x.1 42)
         (halt x.1)))))
'(module
   ((locals (x.1)) (conflicts ((x.1 ()))) (assignment ((x.1 fv0))))
   (begin (set! x.1 42) (halt x.1)))
> (assign-registers
   '(module ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1))
             (conflicts
              ((x.3 (z.5 p.1 y.4 v.1 w.2))
               (w.2 (z.5 p.1 y.4 v.1 x.3))
               (v.1 (w.2 x.3))
               (y.4 (t.6 z.5 p.1 w.2 x.3))
               (p.1 (t.6 z.5 y.4 w.2 x.3))
               (z.5 (t.6 p.1 y.4 w.2 x.3))
```

```
                (t.6 (z.5 p.1 y.4)))))
      (begin
        (set! v.1 1)
        (set! w.2 46)
        (set! x.3 v.1)
        (set! p.1 7)
        (set! x.3 (+ x.3 p.1))
        (set! y.4 x.3)
        (set! p.1 4)
        (set! y.4 (+ y.4 p.1))
        (set! z.5 x.3)
        (set! z.5 (+ z.5 w.2))
        (set! t.6 y.4)
        (set! p.1 -1)
        (set! t.6 (* t.6 p.1))
        (set! z.5 (+ z.5 t.6))
        (halt z.5))))
'(module
   ((locals (v.1 w.2 x.3 y.4 z.5 t.6 p.1))
    (conflicts
     ((x.3 (z.5 p.1 y.4 v.1 w.2))
      (w.2 (z.5 p.1 y.4 v.1 x.3))
      (v.1 (w.2 x.3))
      (y.4 (t.6 z.5 p.1 w.2 x.3))
      (p.1 (t.6 z.5 y.4 w.2 x.3))
      (z.5 (t.6 p.1 y.4 w.2 x.3))
      (t.6 (z.5 p.1 y.4))))
    (assignment
     ((p.1 r15) (z.5 r14) (y.4 r13) (x.3 r9) (w.2 r8) (t.6 r9) (v.1 r15))))
   (begin
     (set! v.1 1)
     (set! w.2 46)
     (set! x.3 v.1)
     (set! p.1 7)
     (set! x.3 (+ x.3 p.1))
     (set! y.4 x.3)
     (set! p.1 4)
     (set! y.4 (+ y.4 p.1))
     (set! z.5 x.3)
     (set! z.5 (+ z.5 w.2))
     (set! t.6 y.4)
     (set! p.1 -1)
     (set! t.6 (* t.6 p.1))
     (set! z.5 (+ z.5 t.6))
     (halt z.5)))
```

4.5.6 Appendix: Compiler Overview

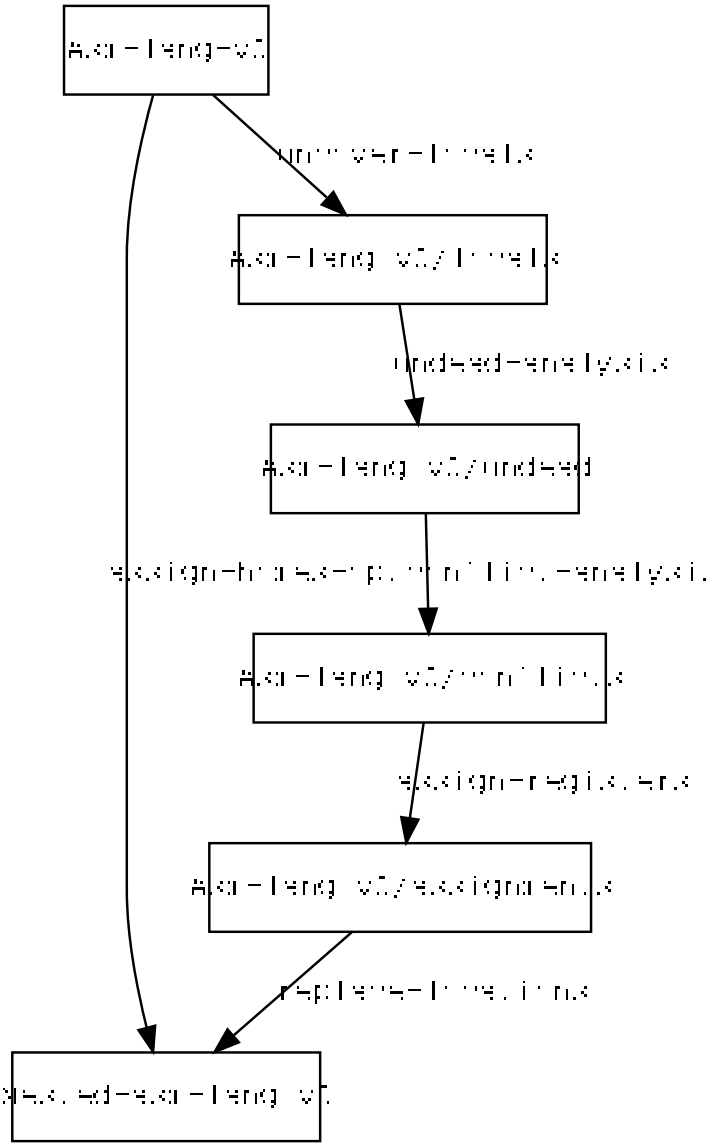


Figure 3: Overview of Register Allocation Optimization

4.5.7 Appendix: Language Definitions

(**asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-lang-v2 grammar. The first non-terminal in the grammar defines valid programs.

asm-lang-v2 : grammar?

p ::= (module *info* *tail*)

info ::= info?

tail ::= (halt *triv*)

```
        |  (begin effect ... tail)

 effect ::= (set! aloc triv)
        |  (set! aloc_1 (binop aloc_1 triv))
        |  (begin effect ... effect)

   triv ::= int64
        |  aloc

  binop ::= *
        |  +

   aloc ::= aloc?

  int64 ::= int64?
```

(asm-lang-v2/locals? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/locals : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...))))

   tail ::= (halt triv)
        |  (begin effect ... tail)

 effect ::= (set! aloc triv)
        |  (set! aloc_1 (binop aloc_1 triv))
        |  (begin effect ... effect)

   triv ::= int64
        |  aloc

  binop ::= *
        |  +

   aloc ::= aloc?

  int64 ::= int64?
```

(asm-lang-v2/undead? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2/undead` grammar.
The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/undead : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-
            out undead-set-tree?)))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
          | aloc

  binop ::= *
          | +

   aloc ::= aloc?

  int64 ::= int64?
```

(**asm-lang-v2/conflicts?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-lang-v2/conflicts` grammar.
The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/conflicts : grammar?

```
      p ::= (module info tail)

   info ::= (#:from-
            contract (info/c (locals (aloc ...)) (conflicts ((aloc (aloc ...)) ...))))

   tail ::= (halt triv)
          | (begin effect ... tail)

 effect ::= (set! aloc triv)
          | (set! aloc_1 (binop aloc_1 triv))
          | (begin effect ... effect)

   triv ::= int64
```

```
        | aloc

  binop ::= *
        | +

   aloc ::= aloc?

  int64 ::= int64?
```

(**asm-lang-v2/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-lang-v2/assignments grammar. The first non-terminal in the grammar defines valid programs.

---

asm-lang-v2/assignments : grammar?

```
     p ::= (module info tail)

  info ::= (#:from-
            contract (info/c (locals (aloc ...)) (assignment ((aloc loc) ...))))

  tail ::= (halt triv)
        | (begin effect ... tail)

effect ::= (set! aloc triv)
        | (set! aloc_1 (binop aloc_1 triv))
        | (begin effect ... effect)

  triv ::= int64
        | aloc

   loc ::= reg
        | fvar

   reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
```

```
        |  r15

  binop ::= *
        |  +

   aloc ::= aloc?

  int64 ::= int64?

   fvar ::= fvar?
```

(**nested-asm-lang-v2?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v2` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v2 : grammar?

```
      p ::= tail

   tail ::= (halt triv)
        |  (begin effect ... tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 triv))
        |  (begin effect ... effect)

   triv ::= int64
        |  loc

    loc ::= reg
        |  fvar

    reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15
```

```
binop ::= *
        |  +

int64 ::= int64?

 fvar ::= fvar?
```

4.6 Structured Control Flow

4.6.1 Preface: What's wrong with our language?

In the last chapter, we designed the language Values-lang v3. This language is an improvement over x64, but has a significant limitation: we can only express simple, straight-line arithmetic computations. We'll never be able to write any interesting programs!

In this chapter, we will expose a machine feature, control flow instructions in the form of labels and `jmp` instructions, and systematically abstract these into a structured control-flow primitive: `if` expressions. Control flow is complex, and adding it requires changes to nearly every pass and every intermediate language.

The overview of this version of the compiler is given in Figure 4.

4.6.2 Designing a source language with structured control flow

As usual, we'll start with our goal and then design the compiler bottom-up. We want to extend Values-lang v3 with a structured control-flow feature: a form of `if` expression.

Our goal is Values-lang v4, duplicated below.

```
      p ::= (module tail)

pred+ ::= (relop triv triv)
        | (true)
        | (false)
        | (not pred)
        | (let ([x value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([x value] ...) tail)
        | (if pred tail tail)+

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)
        | (if pred value value)+
```

```
   triv ::= int64
          | x

      x ::= name?

  binop ::= *
          | +

relop+ ::= <
          | <=
          | =
          | >=
          | >
          | !=

 int64 ::= int64?
```

Note that an `if` expression, `(if pred e e)`, is limited: it cannot branch on an arbitrary expression. It is restricted so that a comparison between two values must appear in the predicate position. This is a necessary artifact we'll inherit from x64. Even by the end of this chapter, we will not yet have enough support from the low-level language to add values that represent the result of a comparison, *i.e.,* we won't have booleans.

> *Design digression:*
>
> This is a design choice. With the abstraction we have so far, we could add pseudo-boolean expressions by giving an arbitrary interpretation to any values in the predicate position. For example, we could interpret `0` as true and `1` as false, or vice versa. Unfortunately, this would expose our source language to undefined behavior—what happens when an `if` expression branches on any other value? We could remedy this slightly by making any non-zero integer true, for example. However, it also means we become unable to distinguish booleans from integers, leading to type confusion when reading and writing data. If a programmer sees the number `-5` printed or in a program, is it a boolean or a number?
>
> There are many ways to solve the problem, but the most robust way is to add proper booleans as a separate primitive data type. That is a task separate from adding control-flow, so we deal with it later, and instead implement a limited form of structured control-flow.

4.6.3 Exposing Control-Flow Primitives

When we want to add a new feature to the source language, we must always ask if there's some existing abstraction in the target language that we can use. So far, Paren-x64 v2 exposes only intructions to move data between locations and perform simple arithmetic computation on locations. This is insufficiently expressive, so we must reach even lower and expose new primitives from the machine.

Thankfully, the machine does expose a primitive. x64 exposes labels, written `l:` before some instruction, and jumps, written `jmp trg` where `trg` is either a label or a register containing a label. Labels can be chained in x64, as we've seen in prior assignments. For example, the following assigns two labels to the same instruction:

```
L1:
L2:
  mov rax, 42
```

We'll begin the next version of our compiler by designing a new *Paren-x64 v4* to expose the additional features of x64 necessary to implement control flow abstractions. This extends the previous Paren-x64 v2 with comparison operations, labels, and conditional and unconditional jump operations.

```
          p ::= (begin s ...)

          s ::= (set! addr int32)
              | (set! addr trg+ reg-)
              | (set! reg loc)
              | (set! reg triv)
              | (set! reg_1 (binop reg_1 int32))
              | (set! reg_1 (binop reg_1 loc))
              | (with-label label s)+
              | (jump trg)+
              | (compare reg opand)+
              | (jump-if relop label)+

        trg+ ::= reg
              | label

       triv ::= trg+
              | reg-
              | int64

      opand+ ::= int64
              | reg

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
```

```
              |  r9
              |  r10
              |  r11
              |  r12
              |  r13
              |  r14
              |  r15

       addr ::= (fbp - dispoffset)

        fbp ::= frame-base-pointer-register?

      binop ::= *
              |  +

     relop+ ::= <
              |  <=
              |  =
              |  >=
              |  >
              |  !=

      int64 ::= int64?

      int32 ::= int32?

 dispoffset ::= dispoffset?

     label+ ::= label?
```

Labels are too complex to define by grammar; instead, they're defined by the `label?` predicate in `cpsc411/compiler-lib`.

In Paren-x64 v4, we model labels with the `(with-label label s)` instruction, which defines a label *`label`* at the instruction *`s`* in the instruction sequence. This corresponds to the x64 string `label:\n s`. Note that they can be nested, allowing the same behavior as chaining labels in x64. For convenience, we assume all labels are symbols of the form *`L.<name>.<number>`*, and are globally unique.

Note that `(with-label label s)` does *not* behave like a `define` in Racket or in Values-lang v3. The instruction *`s`* gets executed after the previous instruction in the sequence, even if the previous instruction was not a jump. `with-label` additionally names the instruction so that we can jump to it later, the same as a label in x64.

The new comparison instruction `(compare reg opand)` corresponds to the x64 instruction `cmp reg, opand`. This instruction compares *`reg`* to *`opand`* and sets some flags in the machine describing their relation, such as whether *`reg`* is less than *`opand`*, or whether they are equal. The flags are used by the next conditional jump instruction.

The conditional jump instructions in x64, in the same order as the definition of *relop*, are: `jl label`, `jle label`, `je label`, `jge label`, `jg label`, and `jne label` and each corresponds to "jump to *trg* if the comparison flag is set to ___". For example, the instruction `je label` jumps to *label* if the comparison flag "equal" is set.

In Paren-x64 v4, we abstract the various conditional jump instructions into a single instruction with multiple flags. The instruction `je l` corresponds to `(jump-if = l)`. `jl l` jumps to *l* if comparison flag "less than" is set, and corresponds to `(jump-if < l)`. The rest of the instructions follow this pattern.

We make an additional simplifying restriction in Paren-x64 v4 compared to x64. We assume that a `compare` instruction is always followed immediately by a `jump-if`, and similarly, that any `jump-if` is immediately preceeded by a `compare`. This is necessay since other instructions in x64, such as binary operations, can affect comparison flags. However, we do not want to try to reason about how the flags are affected by arbitrary comparison, and our compiler will always generate a compare-and-then-conditional-jump sequence of instructions.

To implement Paren-x64 v4, we define the procedure `generate-x64`, which simply converts each instruction to its x64 string form.

```
(generate-x64 p) → (and/c string? x64-instructions?)   procedure
  p : paren-x64-v4?
```

Compile the Paren-x64 v4 program into a valid sequence of x64 instructions, represented as a string.

4.6.3.1 The semantics of labels and jumps as linking

Labels and jumps are a small change to the language syntactically, but have a large effect on the semantics. We can see this by writing an interpreter for the language with labels and jumps and comparing it to an interpreter for a language without them.

We can no longer write the Paren-x64 v4 interpreter in one simple loop over the instructions. Instead, we need some way to resolve labels. That way, when running the interpreter, we can easily jump to any expression at any time—a possibility the language now allows. This process of resolving labels is called *linking*.

We can view the process of linking as yet another compiler, and thus a language design problem. We design a simple linker for Paren-x64 v4 to give you a rough idea of how the operating system's linker works. We use a low-level linking implementation that is similar to the operating system's linker: we first resolve all labels to their address in memory (in our case, their index in

the instruction sequence) and then implement jumps by simply setting a program counter to the instruction's address.

To do this, we design a new language *Paren-x64-rt v4*, which represents the *r*un-*t*ime language used by the interpreter after linking.

```
   p ::= (begin s ...)

   s ::= (set! addr int32)
       | (set! addr trg)
       | (set! reg loc)
       | (set! reg triv)
       | (set! reg_1 (binop reg_1 int32))
       | (set! reg_1 (binop reg_1 loc))
       | (with-label label s)⁻
       | (jump trg)
       | (compare reg opand)
       | (jump-if relop pc-addr⁺ label⁻)

 trg ::= reg
       | pc-addr⁺
       | label⁻

triv ::= trg
       | int64

opand ::= int64
       | reg

 loc ::= reg
       | addr

 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r10
       | r11
       | r12
       | r13
       | r14
       | r15

addr ::= (fbp - dispoffset)
```

```
        fbp ::= frame-base-pointer-register?

      binop ::= *
              |  +

      relop ::= <
              |  <=
              |  =
              |  >=
              |  >
              |  !=

    int32+  ::= int32?

      int64 ::= int64?

  pc-addr+  ::= natural-number/c

    int32−  ::= int32?

 dispoffset ::= dispoffset?

    label−  ::= label?
```

```
(define pc-addr? natural-number/c)
```

We remove the instruction `(with-label label s)` and turn all label values into *pc-addr*, a representation of an address recognized by the interpreter. This encodes the idea that the linker, the compiler from Paren-x64 v4 to Paren-x64-rt v4, resolves labels into addresses. In our case, since programs are represented as lists of instructions, a *pc-addr* is a natural number representing the position of the instruction in the list.

To implement Paren-x64-rt v4 and thus perform linking, we define the procedure `link-paren-x64`.

(link-paren-x64 *p*) → paren-x64-rt-v4? procedure
p : paren-x64-v4?

Compiles Paren-x64 v4 to Paren-x64-rt v4 by resolving all labels to their position in the instruction sequence.

Now we can give a semantics to Paren-x64-rt v4, and thus give a semantics to Paren-x64 v4 and understand the meaning of labels and jumps. We define an interpreter `interp-paren-x64` for Paren-x64 v4 by first linking using `link-paren-x64`, and then running an interpreter for Paren-x64-rt v4. The main loop of the Paren-x64-rt v4 interpreter uses a program counter to keep track of its position in an instruction sequence. To interpret a jump, we simply change the program counter to the number indicated in the jump.

(**interp-paren-x64** *p*) → int64? procedure

p : paren-x64-v4?

Interpret the Paren-x64 v4 program *p* as a value, returning the exit code for *p*.

(**interp-loop** *code memory pc*) → int64? procedure

code : (listof paren-x64-rt-v4.s)

memory : dict?

pc : natural-number/c

The main loop of the interpreter for Paren-x64-rt v4. *code* does not change. *memory* is a dict? mapping physical locations (as symbol?s) to their values (int64?). *pc* is the program counter, indicating the current instruction being executed as an index into the list *code*.

4.6.3.2 Finding the next abstraction boundary

Having exposed x64 features to our compiler internal languages, we now need to find the right boundary at which to abstract away from the low-level representation of control-flow—labels and jumps—and introduce a more structured form of control flow, *if*.

Our next two languages in the pipeline, bottom-up, are Paren-x64-fvars v4 and Para-asm-lang v4. Both of these languages abstract machine-specific details about physical locations. This doesn't seem very related to control-flow, so we simply want to propagate our new primitives up through these layers of abstraction. We expose the new instructions while abstracting away from the machine constraints about which instructions work on which physical locations. Now jumps can target arbitrary locations, and compare can compare arbitrary locations. We can also move labels into locations.

Below we typeset *Paren-x64-fvars v4* with differences compared to Paren-x64-fvars v2.

```
p ::= (begin s ...)

s ::= (set! fvar int32)
    | (set! fvar trg+ reg-)
    | (set! reg loc)
    | (set! reg triv)
    | (set! reg_1 (binop reg_1 int32))
    | (set! reg_1 (binop reg_1 loc))
    | (with-label label s)+
    | (jump trg)+
    | (compare reg opand)+
```

```
        |  (jump-if relop label)+

  trg+ ::= reg
        |  label

  triv ::= trg+
        |  reg-
        |  int64

opand+ ::= int64
        |  reg

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r10
        |  r11
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

relop+ ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int32+ ::= int32?

 int64 ::= int64?

int32- ::= int32?

  fvar ::= fvar?

label+ ::= label?
```

Nothing important changes in Paren-x64-fvars v4. We simply add the new control-flow primitives.

`(implement-fvars p)` → `paren-x64-v4?` procedure

`p` : `paren-x64-fvars-v4?`

Compile the Paren-x64-fvars v4 to Paren-x64 v4 by reifying *fvar*s into displacement mode operands. The pass should use `current-frame-base-pointer`.

Next we typeset *Para-asm-lang v4* compared to Para-asm-lang v2.

```
      p ::= (begin s+ effect- ... (halt triv)-)

     s+ ::= (halt opand)+
          | effect-
          | (set! loc triv)
          | (set! loc_1 (binop loc_1 opand+ triv-))
          | (jump trg)+
          | (with-label label s)+
          | (compare loc opand)+
          | (jump-if relop trg)+

   triv+ ::= opand
          | label

  opand+ ::= triv-
          | int64
          | loc

    trg+ ::= label
          | loc

     loc ::= reg
          | fvar

     reg ::= rsp
          | rbp
          | rax
          | rbx
          | rcx
          | rdx
          | rsi
          | rdi
          | r8
          | r9
          | r12
          | r13
```

```
          |  r14
          |  r15

   binop ::= *
          |  +

relop+ ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

   int64 ::= int64?

   aloc+ ::= aloc?

    fvar ::= fvar?

label+ ::= label?
```

While `halt` is still an instruction, we assume that there is exactly one *dynamic* halt and that it is the final instruction executed in the program. We cannot restrict the syntax to require this, since we now support jumps. Jumps mean our syntax does not give us a clear indication of which instruction is executed last. It might be the case that `halt` is the second instruction in the instruction sequence, but is always executed last because of the control flow of the program. It could also be that there are multiple `halt` instructions syntactically, but only one will ever be executed due to conditional jumps.

This also means compiling `halt` is slightly more complicated. We must ensure that `halt` is the last instruction executed. The run-time system provides a special label, the symbol `'done`, which is expected to be executed at the end of the program. Straightline code will fall through to this label, but it can be jumped to instead.

To implement Para-asm-lang v4, we extend `patch-instructions`. The implementation is essentially similar to the definition from Imperative Abstractions.

The tricky operation to support is `(jump-if relop loc)`, since x64 gives us only `(jump-if relop label)`. We can do this by generating a 2-instruction sequence and negating the *relop*.

`(`**`patch-instructions`** *`p`*`)` → `paren-x64-fvars-v4?` procedure

`p` : `para-asm-lang-v4?`

Compiles Para-asm-lang v4 to Paren-x64-fvars v4 by patching each instruction that has no x64 analogue into a sequence of instructions

using auxiliary register from `current-patch-instructions-registers`.

4.6.4 New Abstractions: Blocks and Predicates

Working our way up the pipeline, the next language from the previous version of our compiler is the Asm-lang v2 family of languages. Recall that this family of languages includes several administrative languages, including Asm-lang v2/assignments. This is the output language of the register allocator.

So here we must stop and ask: is Asm-lang v2 the right place to start from when abstracting away from labels and jumps?

For that, we need to think about what happens in Asm-lang v2. The register allocation and related analyses all happen in Asm-lang v2. If we continue to propagate the primitives up, then the register allocator will be forced to deal with labels and jumps. If we abstract away, then we can design an abstraction that might work better with the register allocator and the analyses.

When control can jump to any instruction at any time, giving semantics to programs, such as writing an interpreter or an analysis, is very difficult. At any label, we must make assumptions about the state of the machine, since the state is not affected only by the sequence of instructions that came before the label in the instruction sequence, but potentially arbitrary instructions that were executed prior to a jumping to the label. This is why we introduced a linking pass before the interpreter. We do not want to have to deal with linking during register allocation.

We therefore want to abstract away from labels and jumps before register allocation.

> *Design digression:*
>
> There are advantages to making the register allocator more aware of labels and jumps. We could write a more complex analysis that tries to resolve labels and jumps, essentially resolving linking and then doing the analysis over the linked program. This would give the register allocator more accurate information about control flow, allowing it to do a better job of minimizing register conflicts, but at the expense of a more complex and slower analysis.

Question: Thinking ahead, what is the problem with analyzing jump instructions? Which parts of the register allocator must change to handle them: conflict analysis, undead analysis, register allocation, or some combination of the three?

To simplify reasoning about programs with control flow, we can organize code into *basic blocks*, labeled blocks where control can only enter the beginning of the block and must exit at the end of the block. This gives us more structure on which to hang assumptions, and can make more assumptions about code when

writing analyses. In particular, we will be able to annotate which registers are undead on entry to and on exit from a block, so our analysis does not have to resolve labels and jumps.

We need to develop this basic block abstraction before we get to the register allocator, so we introduce it next, as an abstraction of Para-asm-lang v4.

We design *Block-asm-lang v4*, a basic-block-structured abstract assembly language in which sequences of statements are organized into basic blocks, and code can jump between blocks. Labels are no longer instructions that can happen anywhere; instead, each block is labeled. Jumps cannot appear just anywhere; instead, they happen only at the end of a block.

```
    p+ ::= (module b ... b)

    b+ ::= (define label tail)

  tail+ ::= (halt opand)+
          | (jump trg)+
          | p-
          | (begin s ... tail+)
          | (if (relop loc opand) (jump trg) (jump trg))+

     s ::= (halt opand)-
         | (set! loc triv)
         | (set! loc_1 (binop loc_1 opand))
         | (jump trg)-
         | (with-label label s)-
         | (compare loc opand)-
         | (jump-if relop trg)-

  triv ::= opand
         | label

 opand ::= int64
         | loc

   trg ::= label
         | loc

   loc ::= reg
         | fvar

   reg ::= rsp
         | rbp
         | rax
         | rbx
         | rcx
         | rdx
         | rsi
```

```
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

In Block-asm-lang v4, a program is a non-empty sequence of labeled blocks. We consider the first block in the sequence to be the start of the program. A *tail* represents a self-contained block of statements. Jumps can only appear at the end of blocks, and jumps only enter the beginning of blocks.

The basic block abstraction essentially forces us to add an `if` statement. We want to ensure jumps happen only at the end of a block, but how could that be if we only have separate `jump-if` instructions as in Para-asm-lang v4? At the very least, we would need to support a block that ends in three instruction sequences: `compare`, followed by a `jump-if`, followed by a `jump`. This is the low-level implementation of an `if` statement. Rather than trying to recognize a three-instruction sequence, we simply abstract the sequence into a single instruction: `(if (`*`cmp loc opand`*`) (jump `*`trg`*`) (jump `*`trg`*`))`. This buys us simplicity in analyzing basic blocks.

The *halt* instruction should only be executed at the end of the final block; it cannot stop control flow, but only indicates that if the program has ended, the *opand* is the final value. Again, we cannot enforce this syntactically due to jumps. Instead, we require that *halt* appears at the end of a block, and assume only one *halt* instruction is ever executed during execution.

To implement Block-asm-lang v4, we simply flatten blocks, moving the *label* from the *define* to the first instruction in the block using *with-label*.

(**flatten-program** *p*) → para-asm-lang-v4? procedure
  p : block-asm-lang-v4?

Compile Block-asm-lang v4 to Para-asm-lang v4 by flattening basic blocks into labeled instructions.

4.6.4.1 Designing A Language for Optimization

When introducing a new statement or expression, we should ask ourselves: what equations do we want to be true of this expression? For example, should we be able to rewrite

```
(if (< 0 1) (jump trg_1) (jump trg_2))
```

to `(jump trg_1)`?

This would be ideal, as it optimizes away the predicate test. What about this: are the following two programs equivalent?

```
(if (< 0 1) (jump trg_1) (jump trg_2))
```

```
(if (>= 0 1) (jump trg_2) (jump trg_1))
```

This... should be true, but it's less obvious why we might do this. But perhaps there are cases where `>=` is faster than `<`, or perhaps for some reason we would like *`(jump trg_1)`* to be the final jump because another optimization would be able to inline that jump.

While it would be straightforward to write an analysis to support these transformations, we could do better by recognizing a pattern and introducing an abstraction. In the first case, what we *really* want to do is transform any expression where the predicate is *obviously true*—we'll write this as `(true)`. Then we could write a simple optimization to transform *`(if (true) (jump trg_1) (jump trg_2))`* into `(jump trg_1)`. Similarly, if we had a predicate that was obviously false, written `(false)`, we could rewrite `(if (false) (jump trg_1) (jump trg_2))` into *`(jump trg_2)`*. If we had a language with a *predicate* abstraction, we could separate the *analysis* of which comparisons are obvious from the *optimization* that rewrites `if` statements with obvious predicates.

We therefore introduce the language *Block-pred-lang v4*. It introduces *`pred`* position. A *`pred`* is *not* a boolean; we can easily compile all *`pred`*s into either a simple *`(relop loc opand)`* or eliminate them entirely. They exist as a way to express the output of some analysis over predicates and enable us to easily rewrite `if` statements.

```
p ::= (module b ... b)
```

```
b ::= (define label tail)
```

```
pred+ ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)

 tail ::= (halt opand)
        | (jump trg)
        | (begin s ... tail)
        | (if pred+ (relop loc opand)- (jump trg) (jump trg))

    s ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))

 triv ::= opand
        | label

opand ::= int64
        | loc

  trg ::= label
        | loc

  loc ::= reg
        | fvar

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

We elide *reg*, *binop*, and *relop* from the grammar above for brevity.

The *pred* position allows *relop*s as before, but also obviously true and false predicates, and predicate negation. This abstraction gives some later pass the ability to optimize *(> 1 0)* to `(true)`.

Obvious predicates, like `(true)` and `(false)` simply compile by transforming the `if` statement into either the first or second branch. The negation predicate, `(not pred)`, swaps the branches and continues compiling `(if pred (jump trg_2) (jump trg_2))`. We leave the *relop* predicate alone.

We implement Block-pred-lang v4 with a simple compiler, `resolve-predicates`.

`(resolve-predicates p)` → `block-asm-lang-v4?` procedure

p : `block-pred-lang-v4?`

Compile the Block-pred-lang v4 to Block-asm-lang v4 by manipulating the branches of *if* statements to resolve branches.

Note that this pass is not an optimization. Optimization passes are intra-language. However, its existence allows us to implement an optimization pass by transforming predicates in the predicate language. We delay writing this optimization for one more language, as an additional abstraction will help us unlock further optimizations.

4.6.4.2 Abstracting Away Jumps

We have already introduced the basic-block abstraction to simplify the structure of jumps for register allocation, but can simplify further. Since our source language, Values-lang v4, doesn't use jumps at all and exposes only an `if` expression, there is no point (yet) to exposing jumps further up the pipeline. If we abstract away from them here and now, then no part of the register allocator will need to deal with jumps.

To abstract away from jumps, we need to design a feature that is sufficient to express `if` *expressions* in terms of `if` *statements* without jumps. The key difference between the two can be seen clearly in the pseudo-grammar-diff below:

```
e ::= (if pred (jump trg_1)+ (jump trg_2)+ e_1- e_2-)
```

In `if` expressions, like other expressions, we support arbitrarily nested sub-expressions in the branches. In `if` statements, we restrict the branches.

To compile `if` expressions to `if` statements, we must generate new basic blocks with fresh labels from nested branches, and transform the branches into jumps. Phrased top-down, the question is if we should do that before or after register allocation? Phrased bottom-up, should we expose jumps through the register allocation languages or not?

As discussed earlier when describing the semantics of labels and jumps, jumps are difficult to give semantics to. We want to avoid analyzing them if we can. We therefore choose to *not* expose jumps any further up the pipeline.

We introduce *Nested-asm-lang v4*, which allows nesting `begin` and `if` expressions that would otherwise need to be expressed with labeled blocks and jumps. This means we could have an *if* statement of the form *(if pred (begin s ... (halt loc)) (begin s ... (halt loc)))*. The nesting structure allows all higher compiler passes to ignore jumps. The language roughly corresponds to an imperative programming language without loops, but one assembly-like feature still remains: physical locations.

```
p ::= (module tail+ b- ...- b-)

b- ::= (define label tail)
```

```
   pred ::= (relop loc triv+ opand−)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)+
          | (if pred pred pred)+

   tail ::= (halt triv+ opand−)
          | (jump trg)−
          | (begin effect+ s− ... tail)
          | (if pred tail+ tail+ (jump trg)− (jump trg)−)

effect+ ::= s−
          | (set! loc triv)
          | (set! loc_1 (binop loc_1 triv+ opand−))
          | (begin effect ... effect)+
          | (if pred effect effect)+

  triv− ::= opand
          | label

 triv+ ::= opand−
          | int64
          | loc

   trg− ::= label
          | loc

    loc ::= reg
          | fvar

    reg ::= rsp
          | rbp
          | rax
          | rbx
          | rcx
          | rdx
          | rsi
          | rdi
          | r8
          | r9
          | r12
          | r13
          | r14
          | r15

  binop ::= *
          | +
```

```
relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label¯ ::= label?
```

Note that Nested-asm-lang v4 enables much of the same nesting we find in monadic form. We skipped over a-normal form. Unnesting `if` requires jumps, unless we want to duplicate code;for efficiency and simplicity, it is beneficial to maintain monadic form until this very low level in the compiler.

To implement Nested-asm-lang v4, we define the procedure `expose-basic-blocks`. The strategy for writing this is slightly complex. Each helper for processing a nonterminal may need to introduce new basic blocks, and transforming a nested `if` expression requires knowning the target of each branch.

The transformation for predicates should transform predicates and generate an `if` statement whose branches are jumps. When processing a *pred*, we need two additional inputs, a "true" and a "false" label, used to generate the output `if` instruction. For a base predicate, such as `(true)` or `(relop aloc triv)`, you can generate an `if` statement. When you find an `if` in predicate position, you'll need to generate two new basic blocks, and rearrange the current true and false labels.

The transformer for effects should take care to unnest `begin` statments. This is not really related to exposing basic blocks, but it is trivial to deal with using the right abstraction, and so does not warrant a separate compiler pass.

`(expose-basic-blocks p)` → `block-pred-lang-v4?` procedure
`p` : `nested-asm-lang-v4?`

Compile the Nested-asm-lang v4 to Block-pred-lang v4, eliminating all nested expressions by generating fresh basic blocks and jumps.

We can now express various optimizations in Nested-asm-lang v4. For example, we can express the following rewrites:

Source⇒Target
`(begin (set! reg 1) (> reg 0))⇒(begin (set! reg 1) (true))`

```
       (begin (set! reg 1) (< reg 0))⇒(begin (set! reg 1) (false))
(begin (set! reg opand_1) (< (max-int ⇒(begin (set! reg opand_1)
                                 64)))  (true))
     (begin (set! reg opand_1) (= reg  ⇒(begin (set! reg opand_1)
                               opand_1))  (true))
     (begin (set! reg int64_1) (= reg  ⇒(false)
                               int64_2))
```

The language doesn't allow us to express relational opreations directly on *opand*s, so we have to be a little more clever to record the possible values of abstract locations, and detect `(> loc 0)`, when `loc` is surely greater than 0.

More generally, we might define an *abstract interpreter*. This interpreter would run during compile-time, and thus over possibly incomplete programs. This means it has to define some abstract notion of the value of a statement. In the worst case, such an abstract value will represent "any run-time value", meaning that we don't have enough static information to predict the result. However, we might be able to evaluate a predicate to determine that in `(begin (set! fv0 5) (> fv0 5))`, `fv0` is surely 5, and that `(not (true))` is surely `(false)` in the abstract interpreter, and if so, this justifies optimizations.

Note that when rewriting predicates, we must be careful to preserve any effects, since we can't (locally) know whether they're necessary or not.

Question: Can you think of any predicates that require using nested `if` statements?

`(optimize-predicates p) → nested-asm-lang-v4?` procedure

`p : nested-asm-lang-v4?`

Optimize Nested-asm-lang v4 programs by analyzing and simplifying predicates.

4.6.5 Register Allocation

Next, we design *Asm-pred-lang v4*, an imperative language that supports some nested structured control-flow. Like Asm-lang v2, this language is a family of administrative languages, each differing only in its info fields.

```
      p ::= (module info+ tail)

  info+ ::= info?

   pred ::= (relop aloc+ loc- triv)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
```

```
        |  (if pred pred pred)

  tail ::= (halt triv)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! aloc+ loc- triv)
        |  (set! aloc_1+ loc_1- (binop aloc_1+ loc_1- triv))
        |  (begin effect ... effect)
        |  (if pred effect effect)

  triv ::= int64
        |  aloc+
        |  loc-

  loc- ::= reg
        |  fvar

  reg- ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 fvar- ::= fvar?
```

The big difference is that physical locations have changed to abstract locations. Recall that this is the big abstraction register allocation buys us, so it ought to

be the only big change.

As before, we treat the register allocator as a single compiler from Asm-pred-lang v4 to Nested-asm-lang v4.

`(assign-homes-opt p) → nested-asm-lang-v4?` procedure
`p : asm-pred-lang-v4?`

Compiles Asm-pred-lang v4 to Nested-asm-lang v4 by replacing all abstract locations with physical locations.

Recall that in our register allocator, we are not designing layers of abstraction like most of our compiler. We are following an existing design: undead analysis, conflict analysis, graph colouring register allocation. We therefore walk through the register allocator in this order.

4.6.5.1 Uncovering Locals

First, we extend `uncover-locals` to analyze `if`. We design the administrative language *Asm-pred-lang v4/locals* below. As with other administrative languages, the only change is the *info* field for the module. It now contains a locals set, describing all variables used in the module.

```
info ::= ((locals (aloc ...)) any ...)+
       | info?-
```

`(uncover-locals p) → asm-pred-lang-v4/locals?` procedure
`p : asm-pred-lang-v4?`

Compiles Asm-pred-lang v4 to Asm-pred-lang v4/locals, analysing which abstract locations are used in the module and decorating the module with the set of variables in an *info* field.

4.6.5.2 Undead Analysis

Now our undead analysis must change to follow the branches of `if` statements. *Asm-pred-lang v4/undead* defines the output of `undead-analysis`.

```
info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out
         undead-set-tree?)))+
       | ((locals (aloc ...)) any ...)-
```

The key to describing the analysis is designing a representation of undead-out sets that can representing the new structure of our statements. Now, statements can branch at `if`. We update the definition of `undead-set-tree?` to

include this case:

```
(define (undead-set? x)
  (and (list? x)
       (andmap aloc? x)
       (= (set-count (list->set x)) (length x))))

(define (undead-set-tree? ust)
  (match ust
    [(? undead-set?) #t]
    [(list (? undead-set?) (? undead-set-tree?) (? undead-set-tree?)) #t]
    [`(,(? undead-set-tree?) ... (? undead-set-tree?)) #t]
    [else #f]))
```

We design a new data structure call the *Undead-set-tree* below.

```
Undead-set is (listof aloc)
interp. a set of undead alocs at a particular instruction

Undead-set-tree is one of:
- Undead-set
- (list Undead-set Undead-set-tree Undead-set-tree)
- (listof Undead-set-tree)
WARNING: datatype is non-canonical since Undead-set-tree can be an
         Undead-set, so second and third case can overlap.
         An Undead-set-tree is meant to be traversed simultaneously with an
         Undead-block-lang/tail, so this ambiguity is not a problem.
interp. a tree of Undead-sets.  The structure of the tree mirrors the
  structure of a Asm-pred-lang. There are three kinds of sub-trees:
(1) an instruction node is simply an undead set;
(2) an if node has an undead-set for the condition and two branch sub-trees.
(3) a begin node is a list of undead set trees, culminating in a sub-tree;
```

For example, consider the following Undead-set-tree.

```
((x.1)(x.1y.2)((y.2b.3)(b.3)(b.3c.4)((c.4)()((c.4)()))))
```

This corresponds to the following *tail*.

```
`(module
   ((locals (x.1 y.2 b.3 c.4)))
   (begin
     (set! x.1 5)
     (set! y.2 x.1)
     (begin
       (set! b.3 x.1)
       (set! b.3 (+ b.3 y.2))
       (set! c.4 b.3)
       (if (= c.4 b.3)
           (halt c.4)
           (begin
             (set! x.1 c.4)
             (halt c.4))))))
```

(**undead-analysis** *p*) → asm-pred-lang-v4/undead? procedure
  p : asm-pred-lang-v4/locals?

Performs undeadness analysis, decorating the program with Undead-set-tree. Only the info field of the program is modified.

4.6.5.3 Conflict Analysis

Next we need to compute the conflict graph.

Below, we design *Asm-pred-lang v4/conflicts* below with structured control-flow.

```
info ::= (#:from-contract (info/c (locals (aloc ...)) (conflicts
         ((aloc (aloc ...)) ...))+ (undead-out undead-set-
         tree?)−))
```

The `conflict-analysis` does not change significantly. We simply extend the algorithm to support the new statements. Note that new statements only reference but never define an abstract location.

(**conflict-analysis** *p*) → asm-pred-lang-v4/conflicts? procedure
  p : asm-pred-lang-v4/undead?

Decorates a program with its conflict graph.

4.6.5.4 Assign Registers

Finally, we design the register allocator as the administrative language *Asm-pred-lang v4/assignments* (pronounced "Asm-pred-lang v4, with assignments").

```
info ::= (#:from-
         contract (info/c (locals (aloc ...)) (assignment+ conflicts− ((aloc loc+ (aloc
         ...)−) ...))))
```

The allocator should run the same algorithm as before. Since the allocator doesn't traverse programs, it shouldn't need any changes.

(**assign-registers** *p*) → asm-pred-lang-v4/assignments? procedure
  p : asm-pred-lang-v4/conflicts?

Performs graph-colouring register allocation, compiling asm-pred-lang v4/conflicts to Asm-pred-lang v4/assignments by decorating programs with their register assignments.

4.6.5.5 Replace Locations

Finally, we actually replace abstract locations with physical locations. In the process, we're free to discard the info from the analyses.

```
info ::= info?+
       | (#:from-contract (info/c (locals (aloc ...)) (assignment
         ((aloc loc) ...))))⁻
```

(**replace-locations** *p*) → nested-asm-lang-v4? procedure

p : asm-pred-lang-v4/assignments?

Compiles Asm-pred-lang v4/assignments to Nested-asm-lang v4 by replacing all abstract location with physical locations using the assignments described in the *assignment* info field.

4.6.6 Imperative Abstractions

Finally, we have all the abstractions in place to abstract away from imperative statements.

First we abstract to an imperative language from our abstract assembly language. We design *Imp-cmf-lang v4*, a pseudo-ANF restricted imperative language.

```
      p ::= (module info⁻ tail)

  info⁻ ::= info?

   pred ::= (relop triv+ aloc⁻ triv)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= value+
          | (halt triv)⁻
          | (begin effect ... tail)
          | (if pred tail tail)

 value+ ::= triv
          | (binop triv triv)

 effect ::= (set! aloc value+ triv⁻)
          | (set! aloc_1 (binop aloc_1 triv))⁻
          | (begin effect ... effect)
```

```
        |  (if pred effect effect)

 triv ::= int64
        |  aloc

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?
```

It is mostly a straightforward extension of Imp-anf-lang v3 to include the `if` and *pred*. However, note that it breaks ANF by allowing nested *effect*s in *pred* position. This means the language is even more non-canonical. The following two programs are equal: `(if (begin (set! x.1 5) (= x.1 5)) (halt x.1) (halt 6))` `(begin (set! x.1 5) (if (= x.1 5) (halt x.1) (halt 6)))`

(select-instructions *p*) → asm-pred-lang-v4? procedure

p : imp-cmf-lang-v4?

Compiles Imp-cmf-lang v4 to Asm-pred-lang v4, selecting appropriate sequences of abstract assembly instructions to implement the operations of the source language.

Similarly, we easily extend Imp-mf-lang v3 to *Imp-mf-lang v4*, defined below.

```
   p ::= (module tail)

pred ::= (relop triv triv)
       |  (true)
       |  (false)
       |  (not pred)
       |  (begin effect ... pred)
       |  (if pred pred pred)

tail ::= value
       |  (begin effect ... tail)
       |  (if pred tail tail)

value ::= triv
       |  (binop triv triv)
       |  (if pred value value)+
       |  (begin effect ... value)+
```

```
effect ::= (set! aloc value)
       |  (if pred effect effect)+
       |  (begin effect ... effect)
       |  (if pred effect effect)−

 triv ::= int64
       |  aloc

binop ::= *
       |  +

relop ::= <
       |  <=
       |  =
       |  >=
       |  >
       |  !=

int64 ::= int64?

 aloc ::= aloc?
```

(**canonicalize-bind** *p*) → imp-cmf-lang-v4? procedure
p : imp-mf-lang-v4?

Compiles Imp-mf-lang v4 to Imp-cmf-lang v4, pushing `set!` under `begin` and `if` so that the right-hand-side of each `set!` is a simple value-producing operation.

This canonicalizes Imp-mf-lang v4 with respect to the equations

```
(set! aloc (begin effect_1 ... value))  = (begin effect_1 ... (set! aloc value))
(set! aloc (if pred value_1 value_2))   = (if pred (set! aloc value_1) (set! aloc value_2))
```

We describe *Values-unique-lang v4* below.

```
   p ::= (module tail)

pred ::= (relop triv triv)
       |  (true)
       |  (false)
       |  (not pred)
       |  (let ([aloc value] ...) pred)+
       |  (begin effect ... pred)−
       |  (if pred pred pred)

tail ::= value
       |  (let ([aloc value] ...) tail)+
```

```
          |  (begin effect ... tail)⁻
          |  (if pred tail tail)

    value ::= triv
          |  (binop triv triv)
          |  (let ([aloc value] ...) value)⁺
          |  (if pred value value)
          |  (begin effect ... value)⁻

  effect⁻ ::= (set! aloc value)
          |  (if pred effect effect)
          |  (begin effect ... effect)

     triv ::= int64
          |  aloc

    binop ::= *
          |  +

    relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

    int64 ::= int64?

     aloc ::= aloc?
```

In Values-unique-lang v4, we extend expressions with an *if* expression that takes a predicate. The predicate form, or sub-language, is still not a true boolean datatype. They cannot be bound to abstract locations or returned as values. We have to restrict the predicate position this way since we have no explicit run-time representation of the value that a comparison operation produces, *i.e.,* we don't have booleans.

(**sequentialize-let** *p*) → imp-mf-lang-v4? procedure

p : values-unique-lang-v4?

Compiles Values-unique-lang v4 to Imp-mf-lang v4 by picking a particular order to implement `let` expressions using `set!`.

Finally, we abstract away from abstract locations and introduce lexical identifiers.

We defined *Values-lang v4* below.

```
      p ::= (module tail)

   pred ::= (relop triv triv)
```

```
      | (true)
      | (false)
      | (not pred)
      | (let ([x+ aloc- value] ...) pred)
      | (if pred pred pred)

 tail ::= value
      | (let ([x+ aloc- value] ...) tail)
      | (if pred tail tail)

value ::= triv
      | (binop triv triv)
      | (let ([x+ aloc- value] ...) value)
      | (if pred value value)

 triv ::= int64
      | x+
      | aloc-

   x+ ::= name?

binop ::= *
      | +

relop ::= <
      | <=
      | =
      | >=
      | >
      | !=

int64 ::= int64?

aloc- ::= aloc?
```

(**uniquify** *p*) → values-unique-lang-v4? procedure

p : values-lang-v4?

Compiles Values-lang v4 to Values-unique-lang v4 by resolving lexical identifiers into unique abstract locations.

4.6.7 Polishing Version 4

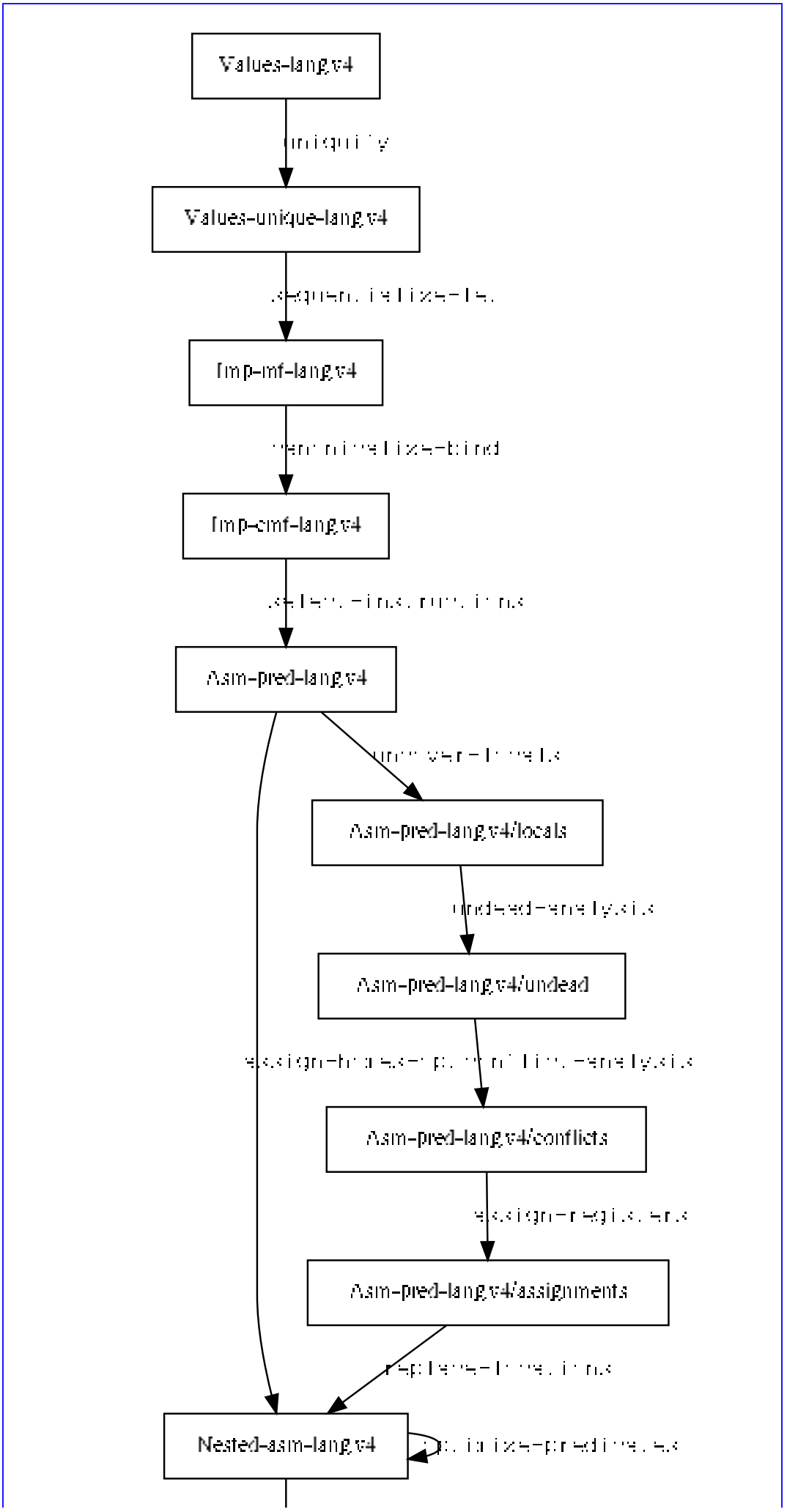
(**interp-values-lang** *p*) → int64? procedure

p : values-lang-v4?

Interpret the Values-lang v4 program *p* as a value. For all *p*, the value

of `(interp-values-lang p)` should equal to `(execute p)`.

4.6.8 Appendix: Overview


Values-lang.v4
uniquify
Values-unique-lang.v4
sequentialize-lang
Imp-mf-lang.v4
uncurry-bind
Imp-cmf-lang.v4
select-instructions
Asm-pred-lang.v4
uncover-locals
Asm-pred-lang.v4/locals
undead-analysis
Asm-pred-lang.v4/undead
assign-homes-conflict-analysis
Asm-pred-lang.v4/conflicts
assign-registers
Asm-pred-lang.v4/assignments
replace-locations
Nested-asm-lang.v4
finalize-predicates

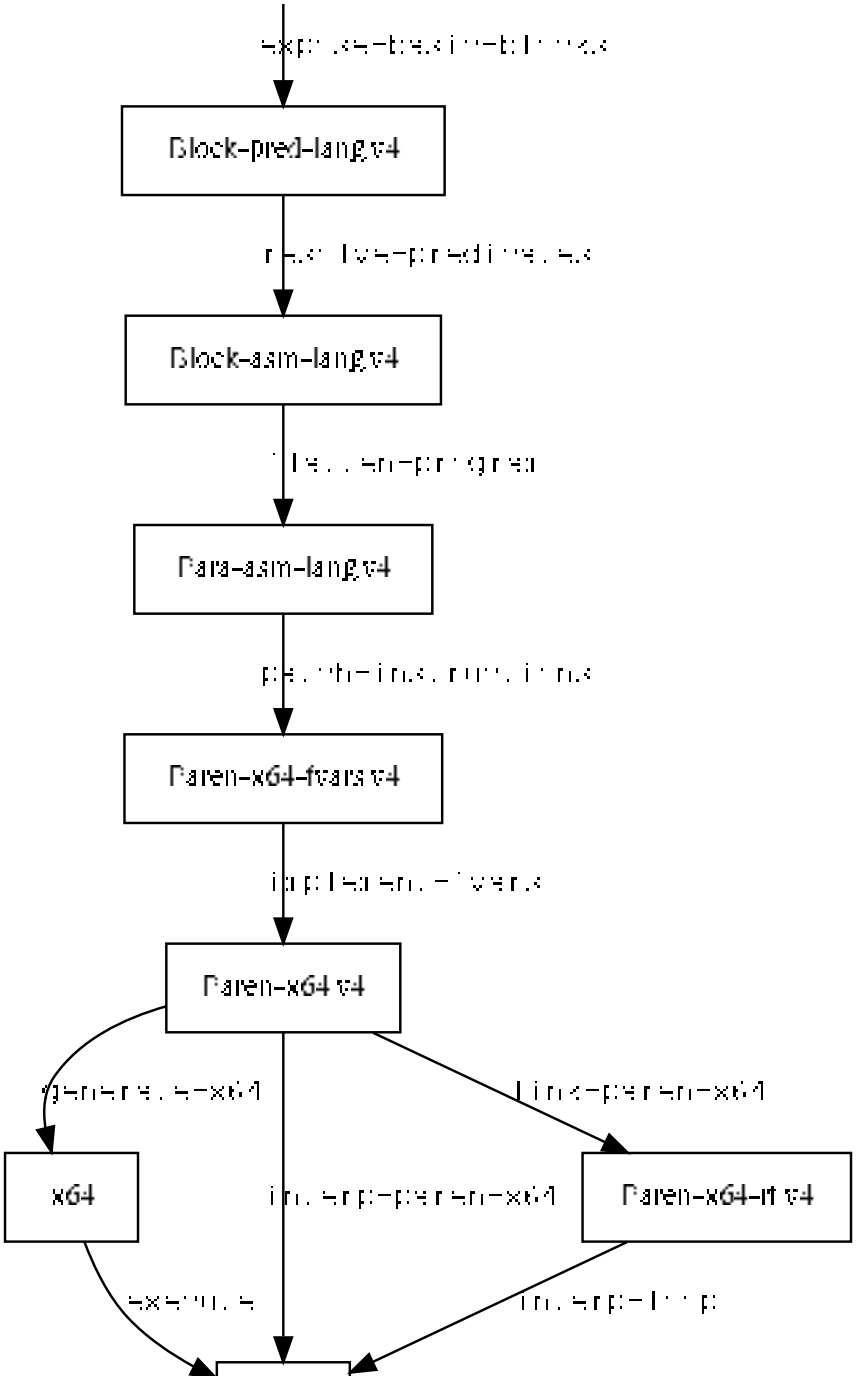


Figure 4: Overview of Compiler Version 4

4.6.9 Appendix: Languages

(values-lang-v4? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

values-lang-v4 : grammar?

```
    p ::= (module tail)

 pred ::= (relop triv triv)
        | (true)
        | (false)
        | (not pred)
        | (let ([x value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([x value] ...) tail)
        | (if pred tail tail)

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)
        | (if pred value value)

 triv ::= int64
        | x

    x ::= name?

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?
```

(**values-unique-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-unique-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

values-unique-lang-v4 : grammar?

```
    p ::= (module tail)

 pred ::= (relop triv triv)
        | (true)
        | (false)
```

```
        |  (not pred)
        |  (let ([aloc value] ...) pred)
        |  (if pred pred pred)

 tail ::= value
        |  (let ([aloc value] ...) tail)
        |  (if pred tail tail)

value ::= triv
        |  (binop triv triv)
        |  (let ([aloc value] ...) value)
        |  (if pred value value)

 triv ::= int64
        |  aloc

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?
```

(**imp-mf-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v4 : grammar?

```
    p ::= (module tail)

 pred ::= (relop triv triv)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

 tail ::= value
```

```
        |  (begin effect ... tail)
        |  (if pred tail tail)

  value ::= triv
        |  (binop triv triv)
        |  (if pred value value)
        |  (begin effect ... value)

 effect ::= (set! aloc value)
        |  (if pred effect effect)
        |  (begin effect ... effect)

   triv ::= int64
        |  aloc

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?
```

(**imp-cmf-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v4 : grammar?

```
      p ::= (module tail)

   pred ::= (relop triv triv)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= value
        |  (begin effect ... tail)
```

```
        |  (if pred tail tail)

  value ::= triv
        |  (binop triv triv)

 effect ::= (set! aloc value)
        |  (begin effect ... effect)
        |  (if pred effect effect)

   triv ::= int64
        |  aloc

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?
```

(asm-pred-lang-v4? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v4 : grammar?

```
      p ::= (module info tail)

   info ::= info?

   pred ::= (relop aloc triv)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (halt triv)
        |  (begin effect ... tail)
        |  (if pred tail tail)
```

```
effect ::= (set! aloc triv)
         | (set! aloc_1 (binop aloc_1 triv))
         | (begin effect ... effect)
         | (if pred effect effect)

  triv ::= int64
         | aloc

 binop ::= *
         | +

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?
```

(asm-pred-lang-v4/locals? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v4/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v4/locals : grammar?

```
    p ::= (module info tail)

 info ::= ((locals (aloc ...)) any ...)

 pred ::= (relop aloc triv)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt triv)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! aloc triv)
         | (set! aloc_1 (binop aloc_1 triv))
```

```
        |  (begin effect ... effect)
        |  (if pred effect effect)

  triv ::= int64
        |  aloc

 binop ::= *
        |  +

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?
```

(**asm-pred-lang-v4/undead?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v4/undead grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v4/undead : grammar?

```
     p ::= (module info tail)

  info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-
           out undead-set-tree?)))

  pred ::= (relop aloc triv)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt triv)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! aloc triv)
        |  (set! aloc_1 (binop aloc_1 triv))
        |  (begin effect ... effect)
        |  (if pred effect effect)
```

```
 triv ::= int64
        | aloc

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?
```

(**asm-pred-lang-v4/conflicts?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v4/conflicts` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v4/conflicts : grammar?

```
    p ::= (module info tail)

 info ::= (#:from-
           contract (info/c (locals (aloc ...)) (conflicts ((aloc (aloc ...)) ...))))

 pred ::= (relop aloc triv)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt triv)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! aloc triv)
        | (set! aloc_1 (binop aloc_1 triv))
        | (begin effect ... effect)
        | (if pred effect effect)

 triv ::= int64
```

```
        |  aloc

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?
```

(**asm-pred-lang-v4/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v4/assignments grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v4/assignments : grammar?

```
     p ::= (module info tail)

  info ::= (#:from-
            contract (info/c (locals (aloc ...)) (assignment ((aloc loc) ...))))

  pred ::= (relop aloc triv)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt triv)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! aloc triv)
        |  (set! aloc_1 (binop aloc_1 triv))
        |  (begin effect ... effect)
        |  (if pred effect effect)

  triv ::= int64
        |  aloc
```

```
 rloc ::= reg
        | fvar

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?
```

(nested-asm-lang-v4? *a***)** → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v4 : grammar?

```
    p ::= (module tail)

 pred ::= (relop loc triv)
        | (true)
        | (false)
```

```
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt triv)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 triv))
        |  (begin effect ... effect)
        |  (if pred effect effect)

  triv ::= int64
        |  loc

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

  fvar ::= fvar?
```

(**block-pred-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-pred-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

block-pred-lang-v4 : grammar?

```
   p ::= (module b ... b)

   b ::= (define label tail)

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)

tail ::= (halt opand)
       | (jump trg)
       | (begin s ... tail)
       | (if pred (jump trg) (jump trg))

   s ::= (set! loc triv)
       | (set! loc_1 (binop loc_1 opand))

triv ::= opand
       | label

opand ::= int64
       | loc

 trg ::= label
       | loc

 loc ::= reg
       | fvar

 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r12
```

```
        |  r13
        |  r14
        |  r15

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

   fvar ::= fvar?

  label ::= label?
```

(**block-asm-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-asm-lang-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

block-asm-lang-v4 : grammar?

```
      p ::= (module b ... b)

      b ::= (define label tail)

   tail ::= (halt opand)
        |  (jump trg)
        |  (begin s ... tail)
        |  (if (relop loc opand) (jump trg) (jump trg))

      s ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))

   triv ::= opand
        |  label

  opand ::= int64
        |  loc

    trg ::= label
```

```
        |  loc

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

  fvar ::= fvar?

 label ::= label?
```

(**para-asm-lang-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the para-asm-lang-v4 grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v4 : grammar?

```
    p ::= (begin s ...)
```

```
    s ::= (halt opand)
        |  (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (jump trg)
        |  (with-label label s)
        |  (compare loc opand)
        |  (jump-if relop trg)

 triv ::= opand
        |  label

opand ::= int64
        |  loc

  trg ::= label
        |  loc

  loc ::= reg
        |  fvar

  reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?
```

```
 fvar ::= fvar?

label ::= label?
```

(**paren-x64-fvars-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-fvars-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-fvars-v4 : grammar?

```
    p ::= (begin s ...)

    s ::= (set! fvar int32)
        | (set! fvar trg)
        | (set! reg loc)
        | (set! reg triv)
        | (set! reg_1 (binop reg_1 int32))
        | (set! reg_1 (binop reg_1 loc))
        | (with-label label s)
        | (jump trg)
        | (compare reg opand)
        | (jump-if relop label)

  trg ::= reg
        | label

 triv ::= trg
        | int64

opand ::= int64
        | reg

  loc ::= reg
        | fvar

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
```

```
        |  r10
        |  r11
        |  r12
        |  r13
        |  r14
        |  r15

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int32 ::= int32?

  int64 ::= int64?

   fvar ::= fvar?

  label ::= label?
```

(**paren-x64-rt-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the paren-x64-rt-v4 grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-rt-v4 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! addr int32)
             |  (set! addr trg)
             |  (set! reg loc)
             |  (set! reg triv)
             |  (set! reg_1 (binop reg_1 int32))
             |  (set! reg_1 (binop reg_1 loc))
             |  (jump trg)
             |  (compare reg opand)
             |  (jump-if relop pc-addr)

        trg ::= reg
             |  pc-addr
```

```
 triv ::= trg
        | int64

opand ::= int64
        | reg

  loc ::= reg
        | addr

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r10
        | r11
        | r12
        | r13
        | r14
        | r15

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int32 ::= int32?

int64 ::= int64?

pc-addr ::= natural-number/c

dispoffset ::= dispoffset?
```

(**paren-x64-v4?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-v4` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v4 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! addr int32)
              | (set! addr trg)
              | (set! reg loc)
              | (set! reg triv)
              | (set! reg_1 (binop reg_1 int32))
              | (set! reg_1 (binop reg_1 loc))
              | (with-label label s)
              | (jump trg)
              | (compare reg opand)
              | (jump-if relop label)

        trg ::= reg
              | label

       triv ::= trg
              | int64

      opand ::= int64
              | reg

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
              | r9
              | r10
              | r11
              | r12
              | r13
              | r14
```

```
          |  r15

   addr ::= (fbp - dispoffset)

    fbp ::= frame-base-pointer-register?

  binop ::= *
          |  +

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int64 ::= int64?

  int32 ::= int32?

dispoffset ::= dispoffset?

  label ::= label?
```

4.7 Procedural Abstraction: Call

4.7.1 Preface: What's wrong with our language?

In the last chapter, we designed the language Values-lang v4 with support for structured control-flow operations. This enables our programs to make run-time decisions that influence what code is executed. However, the language is missing a key feature necessary for practical programming: the ability to reuse code.

In this chapter, we introduce a common method of reusing code: procedural abstraction. We introduce a limited form of procedure that is essentially similar to the basic blocks from the last chapter. This feature is a thin layer of abstraction over the `jmp` instruction that we can *safely* expose in the source language. Our low-level language already exposes `jmp`, so we have all the machinery we need from our low-level languages. However, we need to solve a critical problem: designing a calling convention.

> Procedures are sometimes called functions. This is often a misnomer, since "function" evokes a purely mathematical construct specifying input and output pairs that does not necessarily give rise to an algorithm, while a procedure is code that executes algorithmically, and may rely on machine state in addition to its declared inputs and outputs.

Normally, procedures are thought about as composed of two features: call and return. However, these are two separate abstractions. In this chapter, we introduce only the call abstraction. Procedures can be called, but never return. We limit where such as call can happen so we do not have to answer inconvenient questions.

4.7.2 Designing a source language with call

Below, we define *Values-lang v5* by extending Values-lang-v4 with a *tail calls*—a procedure *call* in `tail` position.

```
p ::= (module (define x (lambda (x ...)
```

```
          tail))+ ...+ tail)

 pred ::= (relop triv triv)
        | (true)
        | (false)
        | (not pred)
        | (let ([x value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([x value] ...) tail)
        | (if pred tail tail)
        | (call x triv ...)+

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)
        | (if pred value value)

 triv ::= int64
        | x

    x ::= name?

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?
```

A program now begins with a set of declared *procedures*, named blocks of code that are parameterized over some data indicated by the list of names `(x ...)`, called the *parameters*. Each of these procedures can be used by the *call* abstraction `(call name value ...)`, which takes the name of a procedure and a list of values, called the *arguments*, with which to instantiate the parameters. At run-time, the call unconditionally transfers control to the code declared in the definition of the procedure *name*.

Notice this call is restricted to *tail* context. If we allowed a call in value context, for example, then we could have a call on the right-hand side of a `let`. This would transfer control with a *tail* still remaining to be executed. Defining what this means in a sensible way is difficult, and requires introducing some abstraction to transfer control back from a

procedure to the middle of some existing computation. Recall that a `value` in `tail` context is implicitly the final answer, and is compiled to a `halt` instruction. In terms of the lower-level languages, we can understand a tail call as jumping until the program reaches a `halt` instruction.

We already have the machinery to compile procedure definitions and calls. Procedure definitions are transformed into basic blocks, and calls into, essentially, `jmp` instructions.

The only question is how to pass arguments. The call instruction needs to know in which locations to store the arguments, and the called procedure needs to know from which locations to read its parameters. The problem is deciding how to ensure the locations end up the same.

4.7.3 Calling Conventions Introduction

Introducing a calling convention solves the problem mentioned above.

The *calling convention* gives us a pattern to set up a call to *any* procedure. We fix a set of *physical locations*. Both the caller and the callee agree that those locations are the only thing they need to know about. Every call will first set those locations and pass control to the procedure. Every procedure will read from those locations on entry, and move the arguments into its own abstract locations. This way, no procedure needs to know about another's abstract locations, and we maintain the per-block regsiter allocator we've used until now.

> *Design digression:*
>
> Strictly speaking, we don't need to use physical locations for our calling convention. What we need is a set of global, shared locations, whose names are unique across the entire program, and any program that we might link with. The register allocator needs to assign all uses of these shared locations to the same physical locations.
>
> A simple implementation of these global shared location is to use physical locations. Physical locations are automatically globally consistent and unique, and automatically known by all programs we might link with. If we allow those physical locations to be registers, we can generate very fast procedure calls by keeping memory out of the picture as much as possible. Unfortunately, using them requires that we expose physical locations through every layer of abstraction up to the point where we implement the calling convention. This makes all our abstractions only partial abstractions, and injects undefined behavior back into our intermediate languages.
>
> We could also use the stack to implement the calling convention. This is simpler, as we can keep registers abstract and need to expose memory high in

> the compiler pipeline anyway, but slower since every procedure call must now access memory.
>
> We could try to create global abstract locations. However, then our register allocator will only work over whole programs; we won't be able to support separate compilation, without implementing a separate, lower-level calling convention. This would create unnecessary indirection and be less efficient.
>
> We could try to introduce abstract call-setup and return-from-call instructions, and leave it to the register allocator to implement these in terms of physical locations. This complicates the register allocator, an already complicated pass. It also mixes concerns: assigning abstract locations to physical locations, and generating the instructions to implement call and return. At some level, the register allocator will have to perform the transformation we're already suggesting: first setup physical locations for the call, create conflicts between those physical locations and abstract locations, then assign registers. At very least, we need to do this before conflict analysis, so we might as well do it before all the analyses. This leads us right back to the original design: expose physical locations through the register allocator, high in the compiler pipeline.

Our calling convention is based on the x64 System V ABI. We deviate from it slightly and develop a similar, but simplified, calling convention. The calling convention is defined by parameters in `cpsc411/compiler-lib`.

Our calling convention passes the first `n` arguments as registers, using the set of registers defined in the parameter `current-parameter-registers`. The default value is `'(rdi rsi rdx rcx r8 r9)`, which is defined by the x64 System V ABI to be where the first 6 arguments of any procedure are stored. To deal with an arbitrary number of arguments, we may need more than the `n` registers we have available for parameters. For the rest, we use fresh frame variables.

Since tail calls never return, we do not need to worry about what is on the frame before a call. We can simply overwrite all existing frame variables. This means recursive tail calls have the same performance characteristic as loops: they use a constant amount of stack space, compile directly to jumps, and only need registers as long as they use fewer than 6 arguments.

4.7.3.1 Designing the Calling Convention Translation

To design our calling convention translation, we start by looking at how we want to translate terms into abstraction we already know about. We then redesign our intermediate languages to support our translation.

> *Design digression:*

> We must be careful to design a translation that eliminates some abstraction layer, or we risk developing a translation that simply kicks the real problem down to the next language. A good heuristic here is to avoid designing a translation that introduces any new abstractions. If we need new abstractions, we should look at the problem bottom up—designing the proposed abstraction as an abstraction of some features expressible in the target language.

Intuitively, we know how to compile calls. We want to instantiate the parameters of the procedure, moving arguments to the shared locations, then jump to the label of the procedure. Concretely, we want to perform the following transformations.

- When transforming a procedure `` `(lambda (,x_0 ... ,x_n-1 ,x_n ... ,x_n+k-1) ,body) ``, we generate:

  ```
  `(begin
     (set! ,x_0 ,r_0)
     ...
     (set! ,x_n-1 ,r_n-1)
     (set! ,x_n ,fv_0)
     ...
     (set! ,x_n+k-1 ,fv_k-1)
     ,body)
  ```

  where:

  - `fv_0 ... fv_k-1` are the first `k` frame variable.
  - `r_0 ... r_n-1` are the `n` physical locations from `current-parameter-registers`.

  The order of the `set!`s is not important for correctness, but can influence optimizations. We want limit the live ranges of registers to help the register allocator make better use of registers, so we should generate accesses to registers first to limit their live ranges.

- When transforming a tail call, `` `(call ,v ,v_0 ... ,v_n-1 ,v_n ... ,v_n+k-1) ``, we generate:

  ```
  `(begin
     (set! ,fv_0 ,v_n) ...
     (set! ,fv_k-1 ,v_n+k-1) ...
     (set! ,r_0 ,v_0) ...
     (set! ,r_n-1 ,v_n-1) ...
     (jump ,v ,fbp ,r_0 ... ,r_n-1 ,fv_0 ... ,fv_k-1))
  ```

  where:

  - `fbp` is the physical location storing the frame base pointer, `current-`

`frame-base-pointer-register`.

- all other meta-variables are the same as in the case for transforming *`lambda`*.

Again, the order of the `set!`s is not important for correctness, but can enable better use of registers and optimizations. This time, we move values into the registers last, to keep there lives as short as possible.

Here, we decorate the `jump` instruction with its undead-out set. Going top-down, it is not obvious why we would do this.

We know we want to compile a call to a series of assignments and a jump. But, since we're going top-down, we need to ask: will the lower-level language be able to implement jump? We must look bottom-up at how we would expose jump from the lower level languages.

We will need to expose `jump` from Block-pred-lang v5, all the way up to whatever this transformation targets. This means at least exposing `jump` through the regsiter allocator. In `undead-analysis`, we will need to decide what set of locations is undead after a `jump`. In general, if we want to support separate compilation, we cannot answer that question. To design this `jump` abstraction in a way we will be able to implement requires explicitly annotating it with its undead-out set.

Now that we know the translation we would like to perform, we need to figure out where to fit the new translation in our compiler pipeline. To do this, we need to ask a few questions.

First: which target languages support the features needed in the target of our translation? If we look at the pipeline from the last chapter, Appendix: Overview, we know the translation must come at least after `sequentialize-let`, since the target language will need to use imperative features introduced in that pass. Introducing calling conventions also generates `begin` statement in tail position, and generates `set!` expressions with a combination of values and locations for operands. This last feature tells us we should place the new pass before `select-instructions`, since we want the unrestricted `set!` feature `select-instructions` provides.

Second: which translations might be disrupted by introducing new features in their source language? If we add the new translation before `canonicalize-bind`, then we won't need to extend `canonicalize-bind` to support the `call` construct, but we will need to make sure `canonicalize-bind` handles the `jump` construct. This might be a problem, if `jump` appears in effect context, but note that since `call` appears in tail context, so must the `jump` that `call` compiles to. This suggests we could place the new pass

either right before or right after `canonicalize-bind`.

Third, we try to future proof our design decisions: will the answer to any of the above questions change if we add new features in the future? This is hard to predict; the future is vast so the search space is large. We can limit our search by focusing on limitations in features we're now adding. We just added `call` in tail context. Are we likely to lift this restriction and allow calls in other contexts later? Yes, that seems likely; most languages allow calls in non-tail context. If we allowed calls in effect context, *e.g.,* then our translation would need to generate a `begin` in effect context, and not just tail context. This tells us we want to support nested `begin` in the target language of our new pass. Since we changed the target of `canonicalize-bind` to allow nested `begin`, we can still place our new pass before or after `canonicalize-bind`.

Finally (and it's important to consider last), we consider minor matters of performance. Since the new pass will introduce many new instructions, it will increase the size of code each pass must transform, and thus slow down the compiler. If we place the new pass after `canonicalize-bind`, then 1 fewer pass has to run on the larger code.

We conclude the new pass should go just after `canonicalize-bind`. Since we're proceeding top-down, we start by extending the first few passes down to `canonicalize-bind`.

4.7.4 Extending front-end with support for call

We start by formally defining *Values-lang v5*, the new source language. We typeset the difference compared to Values-lang v4.

```
      p ::= (module (define x (lambda (x ...)
              tail))+ ...+ tail)

   pred ::= (relop triv triv)
          | (true)
          | (false)
          | (not pred)
          | (let ([x value] ...) pred)
          | (if pred pred pred)

   tail ::= value
          | (let ([x value] ...) tail)
          | (if pred tail tail)
          | (call x triv ...)+

  value ::= triv
```

```
        |  (binop triv triv)
        |  (let ([x value] ...) value)
        |  (if pred value value)

 triv ::= int64
        |  x

    x ::= name?

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?
```

Modules now define a set of procedures at the top-level before the initial `tail`. The program begins executing the `tail` following the set of definitions.

The defined procedures can be used anywhere in the module. We support mutually recursive calls, so procedures defined later in the module can be referenced by definitions earlier in the module. For example, the following is a valid Values-lang v5 program:

```
(module
  (define odd?
    (lambda (x)
      (if (= x 0)
          0
          (let ([y (+ x -1)])
            (call even? y)))))
  (define even?
    (lambda (x)
      (if (= x 0)
          1
          (let ([y (+ x -1)])
            (call odd? y)))))
  (call even? 5))
```

We continue to require that the source program is well bound: all lexical identifiers are defined before they are used. We also restrict procedure to not bind the same identifier twice. We could allow this and define a shadowing order, but this would always introduce a dead variables and is

probably a mistake in the source language.

We have not introduced a method for dynamically checking that a procedure is used correctly yet. To avoid exposing undefined behaviour, we make an additional restriction to calls. Calls must call a statically known procedure with exactly the right number of arguments. Otherwise, our desired transformation will leave some physical locations uninitialized, resulting in undefined behaviour.

We now have two data types exposed in the source language: integers, and procedures. This introduces more opporutnities for undefined behaviour. If we try to compare a procedure to an integer, we will generate code with undefined behaviour. We therefore must check the source program does not use labels in this way.

To validate these assumptions, we implement `check-values-lang`.

`check-values-lang` must necessarily rule out many well-defined programs, since without modifying the source language, we cannot tell the types of all procedure parameters. We design a conservative approximation that is sound, *i.e.,* it never accepts a program with undefined behaviour, but incomplete, *i.e.,* some well-defined programs are rejected. This is a normal trade off in compilers, and yet another consequence of Rice's Theorem.

We use the following heuristics to implement `check-values-lang`:

- A procedure's name can only appear in application position of a `call`, or bound in the right-hand side of a `let`.
- The parameters to a procedure are assumed to be integers.
- A call `(call x triv ...)` is only well typed if `x` is bound to a procedure with `n` parameters and there are exactly `n` arguments in `call`.
- A binary operation `(binop triv_1 triv_2)` is only well typed, and has type integer, if both *`triv_1`* and *`triv_2`* have type integer.
- A relational operation `(relop triv_1 triv_2)` is only well typed if both *`triv_1`* and *`triv_2`* have type integer.
- An `if` expression `(if pred tail_1 tail_2)` is only well typed if *`pred`* is a well-typed predicate.
- Every procedure must return an *`int64`*.

Finally, we have one restriction imposed by the run-time system: the final

result of the program must be an *int64*.

(**check-values-lang** *p*) → Values-lang-v5? procedure
  p : Values-lang-v5?

Validates that the Values-lang v5 is syntactically well-formed, well bound and well typed: all procedure calls pass the correct number of arguments, and all *binop* and *relop* are never used with labels. You may want to separet this into two problems: first checking syntax, then type checking.

Next we need to resolve lexical identifiers. This is slightly complicated by introducing procedures. We want to compile procedures to labeled blocks and jumps, so we need to compile their names to labels rather than abstract locations.

First, we design *Values-unique-lang v5*. We typeset the differences compared to Values-lang v5.

```
      p ::= (module (define label+ x- (lambda (aloc+ x- ...) tail)) ... tail)

   pred ::= (relop opand+ opand+ triv- triv-)
          | (true)
          | (false)
          | (not pred)
          | (let ([aloc+ x- value] ...) pred)
          | (if pred pred pred)

   tail ::= value
          | (let ([aloc+ x- value] ...) tail)
          | (if pred tail tail)
          | (call x- triv opand+ ...)

  value ::= triv
          | (binop opand+ opand+ triv- triv-)
          | (let ([aloc+ x- value] ...) value)
          | (if pred value value)

opand+ ::= int64
          | aloc

   triv ::= opand+
          | label+
          | int64-
          | x-
```

```
   x⁻ ::= name?

 binop ::= *
         | +

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

 aloc⁺ ::= aloc?

 label⁺ ::= label?
```

As usual, we change local lexical identifiers to abstract locations. However, we also change top-level lexical identifiers into *labels*. *labels* are *trivs*, although we assume they are only used according to the typing rules imposed by `check-values-lang`.

`(`**uniquify** *p*`)` → `Values-unique-lang-v5?` procedure

p : `Values-lang-v5?`

Compiles Values-lang v5 to Values-unique-lang v5 by resolving top-level lexical identifiers into unique labels, and all other lexical identifiers into unique abstract locations.

Next we design *Proc-imp-mf-lang v5*, an imperative language in monadic form with procedures. We typeset the differences compared to Values-unique-lang v5.

```
    p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)⁺
        | (let ([aloc value] ...) pred)⁻
        | (if pred pred pred)

 tail ::= value
        | (call triv opand ...)⁺
        | (begin effect ... tail)⁺
```

```
        | (let ([aloc value] ...) tail)⁻
        | (if pred tail tail)
        | (call triv opand ...)⁻

  value ::= triv
        | (binop opand opand)
        | (begin effect ... value)⁺
        | (let ([aloc value] ...) value)⁻
        | (if pred value value)

effect⁺ ::= (set! aloc value)
        | (begin effect ... effect)
        | (if pred effect effect)

  opand ::= int64
        | aloc

   triv ::= opand
        | label

  binop ::= *
        | +

  relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?
```

There are no interesting changes. We simply propagate the new procedure forms form down one more level of abstraction.

(**sequentialize-let** *p*) → `proc-imp-mf-lang-v5?procedure`
  p : `values-unique-lang-v5?`

Compiles Values-unique-lang v5 to Proc-imp-mf-lang v5 by picking a particular order to implement `let` expressions using `set!`.

4.7.5 Implementing Calling Conventions

Now we can design *Imp-mf-lang v5*. The design follows the needs of the translation we designed in Designing the Calling Convention Translation.

```
      p ::= (module (define label tail+ (lambda (aloc ...)
              tail)-) ... tail)

   pred ::= (relop opand opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= value
          | (jump trg loc ...)+
          | (call triv opand ...)-
          | (begin effect ... tail)
          | (if pred tail tail)

  value ::= triv
          | (binop opand opand)
          | (begin effect ... value)
          | (if pred value value)

 effect ::= (set! loc+ aloc- value)
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | loc+
          | aloc-

   triv ::= opand
          | label

   loc+ ::= rloc
          | aloc

   trg+ ::= label
          | loc

  binop ::= *
          | +

  relop ::= <
          | <=
```

```
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

rloc+ ::= register?
        |  fvar?
```

We remove the `call` form and replace it by the `jump` form. As described in Designing the Calling Convention Translation, all calls are compiled to a sequence of `set!`s moving the arguments followed by a `jump`, and all procedure definitions are compiled to a block that assigns the parameters, as directed by the calling convention.

Note that we now require physical locations in the target language, so we must gradually expose physical locations up to this language from the rest of the compiler.

(impose-calling-conventions *p*) → imp-mf-lang-v5-p?
p : proc-imp-mf-lang-v5

Compiles Proc-imp-mf-lang v5 to Imp-mf-lang v5 by imposing calling conventions on all calls and procedure definitions. The parameter registers are defined by the list `current-parameter-registers`.

Finally, we design *Imp-cmf-lang v5*.

```
   p ::= (module (define label tail) ... tail)

pred ::= (relop opand opand)
       |  (true)
       |  (false)
       |  (not pred)
       |  (begin effect ... pred)
       |  (if pred pred pred)

tail ::= value
       |  (jump trg loc ...)
       |  (begin effect ... tail)
       |  (if pred tail tail)
```

```
 value ::= triv
         | (binop opand opand)
         | (begin effect ... value)⁻
         | (if pred value value)⁻

effect ::= (set! loc value)
         | (begin effect ... effect)
         | (if pred effect effect)

 opand ::= int64
         | loc

  triv ::= opand
         | label

   loc ::= rloc
         | aloc

   trg ::= label
         | loc

 binop ::= *
         | +

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
         | fvar?
```

There are no interesting changes.

(**canonicalize-bind** *p*) → imp-cmf-lang-v5.p procedure
p : imp-mf-lang-v5.p

Compiles Imp-mf-lang v5 to Imp-cmf-lang v5, pushing `set!` under `begin` so that the right-hand-side of each `set!` is simple value-producing operation.

This canonicalizes Imp-mf-lang v5 with respect to the equations:

```
(set! aloc (begin effect_1 ... value))  = (begin effect_1 ... (set! aloc value))
(set! aloc (if pred value_1 value_2))   = (if pred (set! aloc value_1) (set! aloc value_2))
```

4.7.6 Exposing Jumps

Implementing our calling convention requires exposing jumps quite high in the compiler pipeline, in Imp-mf-lang v5, while in the previous chapter we hid jumps behind an abstraction boundary in Block-pred-lang v4. Thankfully, it is not very difficult to propagate jumps up the compiler pipeline. The main challenge is in adjusting the register allocator, but we have already done the design work to simplify that.

First, we design *Asm-pred-lang v5*, the target of our `select-instructions` pass. To see how to extend `select-instructions`, we should view the difference, we typeset the differences compared to Asm-pred-lang v4.

```
      p ::= (module info (define label info tail)+ ...+ tail)

   info ::= info?

   pred ::= (relop loc+ opand+ aloc- triv-)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (halt opand+ triv-)
          | (jump trg loc ...)+
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc+ aloc- triv)
          | (set! loc_1+ aloc_1- (binop loc_1+ opand+ aloc_1- triv-))
          | (begin effect ... effect)
          | (if pred effect effect)

opand+ ::= int64
          | loc

 triv+ ::= opand
          | label
```

```
loc+ ::= rloc+
       | triv⁻
       | int64⁻
       | aloc

trg+ ::= label
       | loc

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

aloc ::= aloc?

label+ ::= label?

rloc+ ::= register?
        | fvar?
```

The main difference is in the addition of the `jump` instruction. Note that the "arguments" to the jump are not part of meaning of the instruction; they are just metadata used later by the compiler.

`(`**`select-instructions`** *`p`*`)` → `asm-pred-lang-v5.p` `procedure`
`p` : `imp-cmf-lang-v5.p`

Compiles Imp-cmf-lang v5 to Asm-pred-lang v5, selecting appropriate sequences of abstract assembly instructions to implement the operations of the source language.

4.7.6.1 Extending Register Allocation

First, we extend `uncover-locals` to analyze jumps. We design the administrative language *Asm-pred-lang v5/locals* (Asm-pred-lang v5 with locals) below. Note that the only difference is in the specification of the *`info`* field.

```
p ::= (module info (define label info tail) ... tail)
```

```
info ::= (#:from-contract (info/c (locals (aloc ...))))+
       | info?-
```

Note that because the source language now includes blocks, we need to perform the local analysis over each block. Each block gets its own *info* field, with its own locals set. The locals set for the initial tail of the module is stored in the module's info field.

(**uncover-locals** *p*) → asm-pred-lang-v5/locals.p? procedure
p : asm-pred-lang-v5.p?

Compiles Asm-pred-lang v5 to Asm-pred-lang v5/locals, analysing which abstract locations are used in each block, and each block and the module with the set of variables in an `info?` fields.

Now our undead algorithm must change to analyze jumps. *Asm-pred-lang v5/undead* defines the output of `undead-analysis`.

```
info ::= (#:from-
          contract (info/c (locals (aloc ...)) (undead-out
          undead-set-tree/rloc?)+))

tail ::= (halt opand)
       | (jump trg loc ...)
       | (begin effect ... tail)
       | (if pred tail tail)
```

When analyzing a `jump` statement, we need to compute its undead-out set.

In general, this is difficult. In general, we may not know the destination of the jump, so we would either have to conservatively approximate and say "anything could be live, *i.e.,* everything is undead", or analyze the control flow of the program, following jumps and analyzing the destination.

Thankfully, none of that is necessary. Because jumps in our language only come from procedure calls, and our calling convention translation decorated the jump with the locations used by the procedure call, our undead analysis is trivial. The undead-out set of a jump statement `(jump triv_1 triv_2 ...)` is the set `triv_2 ...`.

This requires no changes to the undead-set tree.

Again, we also need to modify the analysis slightly to perform local

analysis on each block, and store undead-set trees in the info field for the corresponding block.

`(undead-analysis p)` → `Asm-pred-lang-v5/undead?` procedure
  `p` : `Asm-pred-lang-v5/locals?`

Performs undead analysis, compiling Asm-pred-lang v5/locals to Asm-pred-lang v5/undead by decorating programs with their undead-set trees.

Next we need to compute the conflict graph.

Below, we design *Asm-pred-lang v5/conflicts* below with structured control-flow.

```
info ::= (#:from-
          contract (info/c (locals (aloc ...)) (conflicts
          ((loc (loc ...)) ...))+ (undead-out undead-set-
          tree/rloc?)−))
```

The `conflict-analysis` does not change significantly. We simply extend the algorithm to support jump statements. Note that `jump` only references but never defines an abstract location.

Again, the analysis should perform local analysis on each block separately.

`(conflict-analysis p)` → `Asm-pred-lang-v5/conflicts?` procedure
  `p` : `Asm-pred-lang-v5/undead?`

Performs conflict analysis, compiling Asm-pred-lang v5/undead to Asm-pred-lang v5/conflicts by decorating programs with their conflict graph.

The graph colouring register alloctor does not need major changes. Below we define *Asm-pred-lang v5/assignments*, which only changes in the `info` field as usual.

```
info ::= (#:from-
          contract (info/c (locals (aloc ...)) (assignment+ conflicts− ((aloc
          rloc)+ (loc (loc ...))− ...))))
```

The allocator should run the same algorithm as before, but this time, on each block separately.

(**assign-registers** *p*) → Asm-pred-lang-v5/assignments? procedure
  p : Asm-pred-lang-v5/conflicts?

Performs graph-coloring register allocation, compiling Asm-pred-lang v5/conflicts to Asm-pred-lang v5/assignments by decorating programs with their register assignments.

Finally, we actually replace abstract locations with physical locations.

```
    p ::= (module info⁻ (define label info⁻ tail) ... tail)

 info⁻ ::= (#:from-
            contract (info/c (locals (aloc ...)) (assignment ((aloc rloc) ...))))

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt opand)
        | (jump trg loc⁻ ...⁻)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)

opand ::= int64
        | loc

 triv ::= opand
        | label

 loc⁻ ::= rloc
        | aloc

  trg ::= label
        | loc

 loc⁺ ::= reg
        | fvar

 reg⁺ ::= rsp
        | rbp
```

```
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

fvar+ ::= fvar?

label ::= label?

rloc- ::= register?
        |  fvar?
```

We need to extend the implementation to traverse each block, and support jump statements. In the process, we also discard the undead annotations on the jump instruction.

(**replace-locations** *p*) → nested-asm-lang-v5? procedure
  p : asm-pred-lang-v5/assignments?

Replaces all abstract location with physical locations using the assignment described in the *assignment* info field.

4.7.6.2 Exposing Basic Blocks

The last update need to make is to `expose-basic-blocks`. We design the source, *Nested-asm-lang v5* below, typeset compared to Nested-asm-lang v4

```
       p ::= (module (define label tail)+ ...+ tail)

    pred ::= (relop loc opand+ triv−)
           | (true)
           | (false)
           | (not pred)
           | (begin effect ... pred)
           | (if pred pred pred)

    tail ::= (halt opand+ triv−)
           | (jump trg)+
           | (begin effect ... tail)
           | (if pred tail tail)

  effect ::= (set! loc triv)
           | (set! loc_1 (binop loc_1 opand+ triv−))
           | (begin effect ... effect)
           | (if pred effect effect)

 opand+ ::= triv−
           | int64
           | loc

  triv+ ::= opand
           | label

   trg+ ::= label
           | loc

     loc ::= reg
           | fvar

     reg ::= rsp
           | rbp
           | rax
           | rbx
           | rcx
           | rdx
           | rsi
           | rdi
           | r8
           | r9
           | r12
```

```
        |  r13
        |  r14
        |  r15

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

   fvar ::= fvar?

 label+ ::= label?
```

The main difference is the inclusion of `jump` expressions and block definitions. These do not complicate the process of exposing basic blocks much. We simply need to traverse each block, exposing new blocks in the process.

Note that we again need to impose the convention that execution begins with the first basic block, and move the initial *`tail`* into an explicit block.

The target language is *Block-pred-lang v5*, typeset compared to Block-pred-lang v4 below.

```
     p ::= (module b ... b)

     b ::= (define label tail)

  pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)

  tail ::= (halt opand)
        |  (jump trg)
        |  (begin s ... tail)
        |  (if pred (jump trg) (jump trg))

     s ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
```

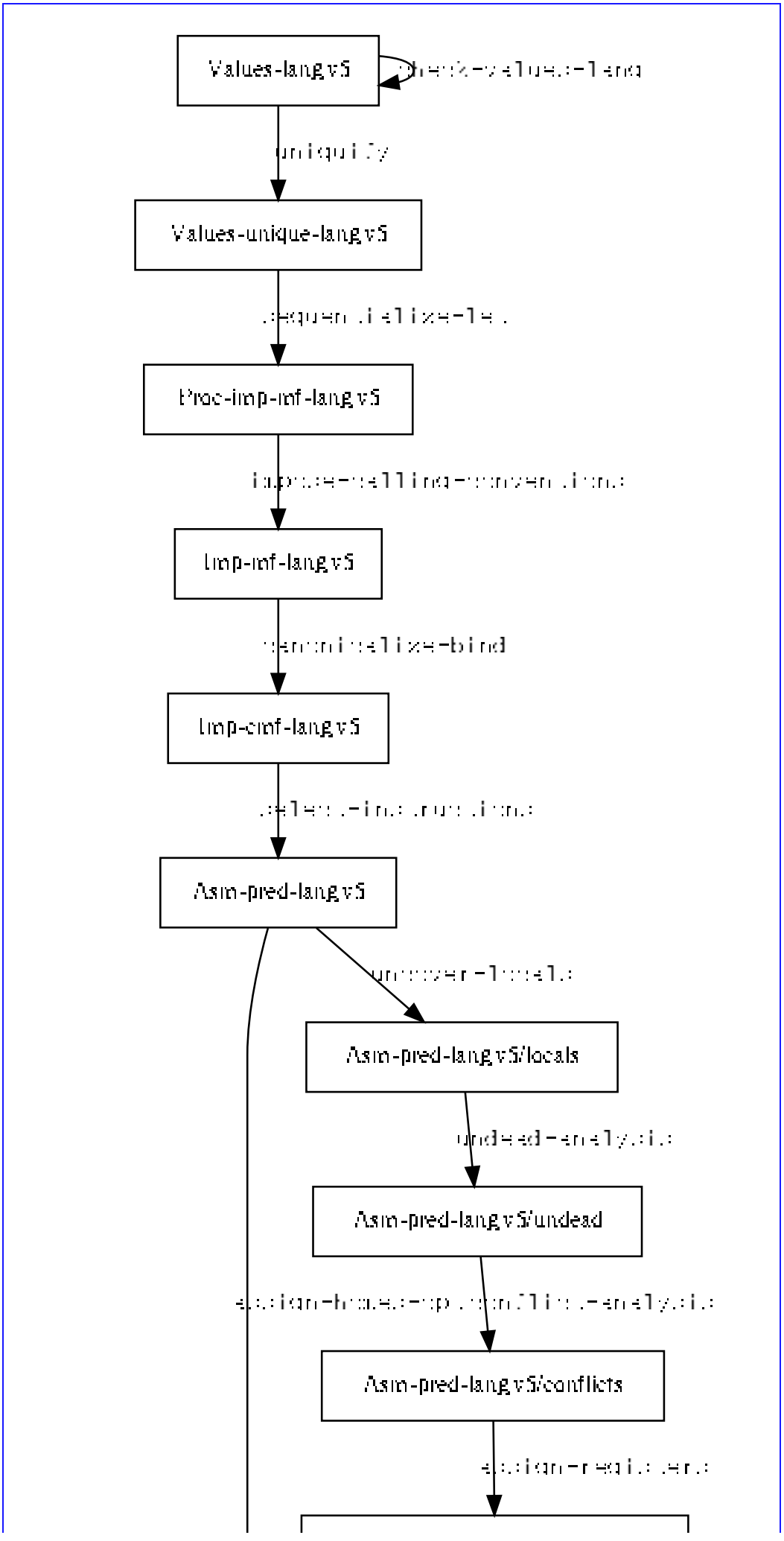
```
opand+ ::= int64
         | loc

  triv ::= opand
         | label

opand- ::= int64
         | loc

   trg ::= label
         | loc

   loc ::= reg
         | fvar

   reg ::= rsp
         | rbp
         | rax
         | rbx
         | rcx
         | rdx
         | rsi
         | rdi
         | r8
         | r9
         | r12
         | r13
         | r14
         | r15

 binop ::= *
         | +

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?

  fvar ::= fvar?

 label ::= label?
```

(**expose-basic-blocks** *p*) → block-pred-lang-v5 procedure
  p : nested-asm-lang-v5?

Compile the Nested-asm-lang v5 to Block-pred-lang v5, eliminating all nested expressions by generate fresh basic blocks and jumps.

Nothing else in the compiler needs to change.

4.7.7 Appendix: Overview


Values-lang.v5
uniquify
Values-unique-lang.v5
Proc-imp-inf-lang.v5
Imp-inf-lang.v5
Asm-pred-lang.v5
Asm-pred-lang.v5/locals
Asm-pred-lang.v5/undead
Asm-pred-lang.v5/conflicts

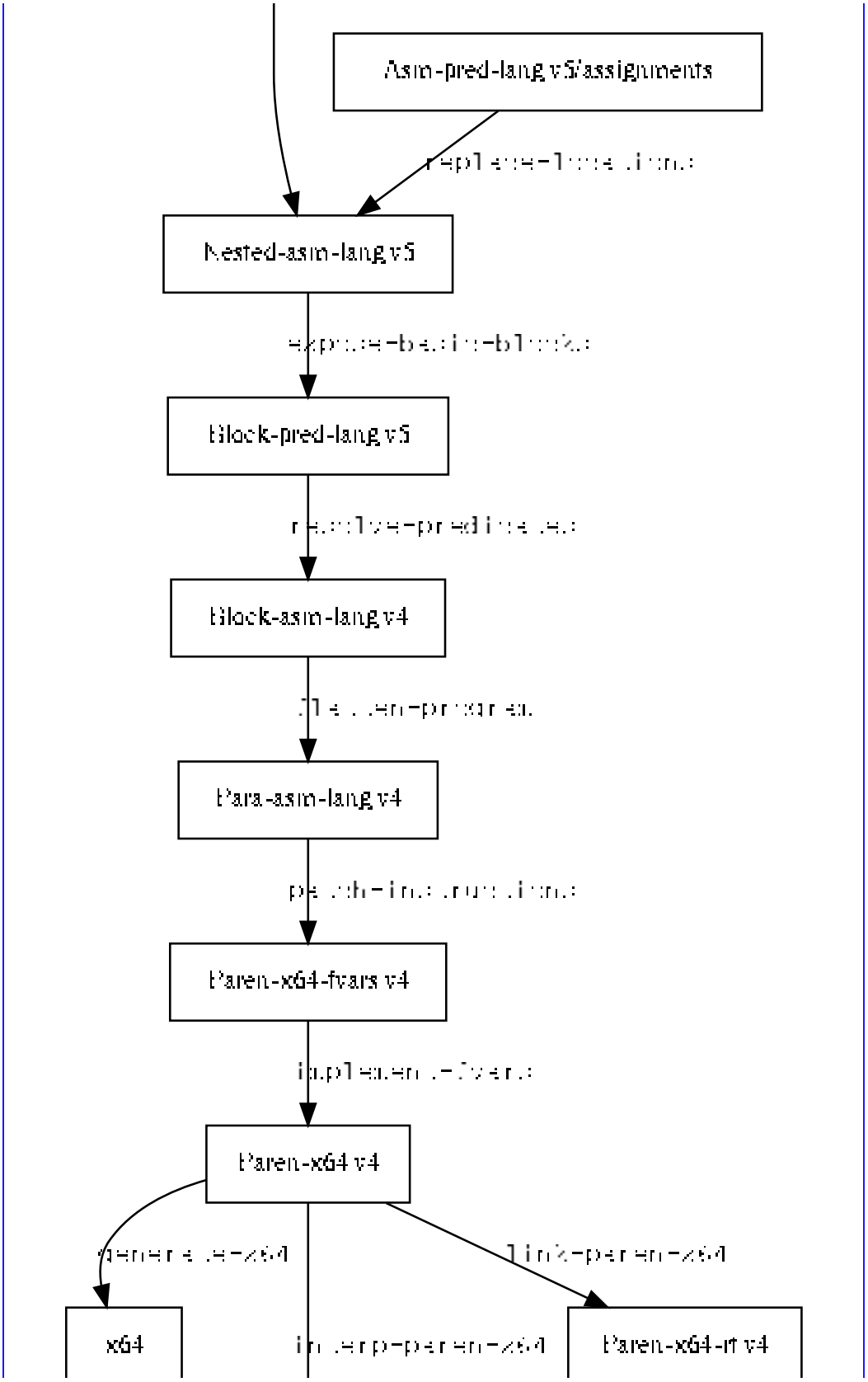


Figure 5: Overview of Compiler Version 5

4.7.8 Appendix: Languages

(**values-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

values-lang-v5 : grammar?

```
    p ::= (module (define x (lambda (x ...) tail)) ... tail)

 pred ::= (relop triv triv)
        | (true)
        | (false)
        | (not pred)
        | (let ([x value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([x value] ...) tail)
        | (if pred tail tail)
        | (call x triv ...)

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)
        | (if pred value value)

 triv ::= int64
        | x

    x ::= name?

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?
```

(**values-unique-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-unique-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

---

values-unique-lang-v5 : grammar?

```
    p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([aloc value] ...) tail)
        | (if pred tail tail)
        | (call triv opand ...)

value ::= triv
        | (binop opand opand)
        | (let ([aloc value] ...) value)
        | (if pred value value)

opand ::= int64
        | aloc

 triv ::= opand
        | label

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?
```

```
label ::= label?
```

(proc-imp-mf-lang-v5? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `proc-imp-mf-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

---

proc-imp-mf-lang-v5 : grammar?

```
      p ::= (module (define label (lambda (aloc ...) tail)) ...
                    tail)

   pred ::= (relop opand opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= value
          | (call triv opand ...)
          | (begin effect ... tail)
          | (if pred tail tail)

  value ::= triv
          | (binop opand opand)
          | (begin effect ... value)
          | (if pred value value)

 effect ::= (set! aloc value)
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | aloc

   triv ::= opand
          | label

  binop ::= *
          | +

  relop ::= <
          | <=
```

```
          |  =
          |  >=
          |  >
          |  !=

   int64 ::= int64?

    aloc ::= aloc?

   label ::= label?
```

(**imp-mf-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v5 : grammar?

```
       p ::= (module (define label tail) ... tail)

    pred ::= (relop opand opand)
          |  (true)
          |  (false)
          |  (not pred)
          |  (begin effect ... pred)
          |  (if pred pred pred)

    tail ::= value
          |  (jump trg loc ...)
          |  (begin effect ... tail)
          |  (if pred tail tail)

   value ::= triv
          |  (binop opand opand)
          |  (begin effect ... value)
          |  (if pred value value)

  effect ::= (set! loc value)
          |  (begin effect ... effect)
          |  (if pred effect effect)

   opand ::= int64
          |  loc
```

```
 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**imp-cmf-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v5 : grammar?

```
   p ::= (module (define label tail) ... tail)

pred ::= (relop opand opand)
       | (true)
       | (false)
       | (not pred)
       | (begin effect ... pred)
       | (if pred pred pred)

tail ::= value
```

```
        |  (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 value ::= triv
        |  (binop opand opand)

effect ::= (set! loc value)
        |  (begin effect ... effect)
        |  (if pred effect effect)

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   loc ::= rloc
        |  aloc

   trg ::= label
        |  loc

 binop ::= *
        |  +

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

asm-pred-lang-v5 : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= info?

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (halt opand)
          | (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  binop ::= *
          | +

  relop ::= <
          | <=
          | =
          | >=
          | >
          | !=

  int64 ::= int64?
```

```
 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v5/locals?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v5/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v5/locals : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (locals (aloc ...))))

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt opand)
        | (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
```

```
        |  loc

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v5/undead?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v5/undead` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v5/undead : grammar?

```
     p ::= (module info (define label info tail) ... tail)

  info ::= (#:from-
            contract (info/c (locals (aloc ...)) (undead-
            out undead-set-tree/rloc?)))

  pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt opand)
        |  (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)
```

```
effect ::= (set! loc triv)
         | (set! loc_1 (binop loc_1 opand))
         | (begin effect ... effect)
         | (if pred effect effect)

opand ::= int64
        | loc

triv ::= opand
       | label

loc ::= rloc
      | aloc

trg ::= label
      | loc

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

aloc ::= aloc?

label ::= label?

rloc ::= register?
       | fvar?
```

(asm-pred-lang-v5/conflicts? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v5/conflicts` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v5/conflicts : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-
           contract (info/c (locals (aloc ...)) (conflicts ((loc (loc ...)) ...))))

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt opand)
        | (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?
```

```
  label ::= label?

   rloc ::= register?
          | fvar?
```

(asm-pred-lang-v5/assignments? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v5/assignments` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v5/assignments : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-
             contract (info/c (locals (aloc ...)) (assignment ((aloc rloc) ...))))

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (halt opand)
          | (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
```

```
        |  loc

  binop ::= *
        |  +

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**nested-asm-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v5 : grammar?

```
     p ::= (module (define label tail) ... tail)

  pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt opand)
        |  (jump trg)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
```

```
       |  (begin effect ... effect)
       |  (if pred effect effect)

opand ::= int64
       |  loc

 triv ::= opand
       |  label

  trg ::= label
       |  loc

  loc ::= reg
       |  fvar

  reg ::= rsp
       |  rbp
       |  rax
       |  rbx
       |  rcx
       |  rdx
       |  rsi
       |  rdi
       |  r8
       |  r9
       |  r12
       |  r13
       |  r14
       |  r15

binop ::= *
       |  +

relop ::= <
       |  <=
       |  =
       |  >=
       |  >
       |  !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

(**block-pred-lang-v5?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-pred-lang-v5` grammar. The first non-terminal in the grammar defines valid programs.

block-pred-lang-v5 : grammar?

```
    p ::= (module b ... b)

    b ::= (define label tail)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)

 tail ::= (halt opand)
        | (jump trg)
        | (begin s ... tail)
        | (if pred (jump trg) (jump trg))

    s ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))

opand ::= int64
        | loc

 triv ::= opand
        | label

  trg ::= label
        | loc

  loc ::= reg
        | fvar

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
```

```
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

binop ::= *
        |  +

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

4.8 Procedural Abstraction: Return

4.8.1 Preface: What's wrong with our language?

In Values-lang v5, we added a limited form of procedure which supported only the call abstraction. This necessarily limited in which context procedures could be called, restricting them to *tail* context. This is unfortunate, since we may want to use procedural abstraction to abstract over side effects, and use such procedures in *effect* context, or to compute values as part of some computation to be used in *value* context. We can emulate these behaviours by manually writing our code in continuation-passing style (CPS) (sometimes known as "callback hell"), but we don't want to be sent to The Hague, so we will not design a language that forces programmers use it.

To allow calls in arbitrary context, we need the *return* abstraction, which allows the language to essentially jump back from a procedure call into the middle of a computation. Thankfully, we already have the abstractions required to implement this: labels, jumps, and a calling convention. All we need to do is slightly generalized the calling convention to introduce a new label at the return point of a procedure call, store that label somewhere, and arrange for the caller to jump back to that label.

> *Design digression:*
>
> We might think we could pre-process Values-lang v6 to lift all non-tail calls into new procedure and rewrite the program to have only tail calls. This would correspond to a CPS transformation and would be difficult to optimize correctly, particularly at this level of abstraction. Each non-tail call would introduce an entire procedure call setup to the "rest" of the body, as a continuation. Any parameters live across the non-tail call would need to be packaged and explicitly passed as arguments to the continuation. By creating a return point abstraction, later in the compiler when we have access to lower-level abstraction (basic blocks), we'll instead be able to generate a single jump back to this return point, instead of a whole procedure call.

4.8.2 Designing a source language with return

Below, we define *Values-lang v6* by extending Values-lang-v5 with a return, and with calls in arbitrary contexts.

```
     p ::= (module (define x (lambda (x ...) tail)) ... tail)

  pred ::= (relop triv triv)
         |  (true)
```

```
        |  (false)
        |  (not pred)
        |  (let ([x value] ...) pred)
        |  (if pred pred pred)

 tail ::= value
        |  (let ([x value] ...) tail)
        |  (if pred tail tail)
        |  (call x triv ...)

value ::= triv
        |  (binop triv triv)
        |  (let ([x value] ...) value)
        |  (if pred value value)
        |  (call x triv ...)+

 triv ::= int64
        |  x

    x ::= name?

binop ::= *
        |  +
        |  -+

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?
```

The only syntactic change is the addition of `(call triv triv ...)` in *value* context. The implementation of this requires a semantic change to call, to ensure it returns. Otherwise, an expression such as `(let ([x (call f 5)]) (+ 1 x))` would return the value *x*, and not `(+ 1 x)` as intended.

Note that *tail* and *value* context now coincide. We do not collapse them yet, as imposing a distinction will improve our compiler, by allowing us to transform tail calls (which need not return) separately from *non-tail calls*, *i.e.,* calls in any context other than *tail* context. This will let us maintain the performance characteristics of tail calls, namely that they use a constant amount of stack space, and do not need to jump after their computation is complete.

We also add subtraction as a *binop*. This week, we start to need subtraction a lot more and it's tiring to encode it when x64 supports it. This requires almost no changes to the compiler if it parameterized by the set of *binop*s.

4.8.3 Extending our Calling Convention

In Procedural Abstraction: Call, we designed a calling convention, but only aimed to support calls without return. We need to modify it in three ways to support return, and non-tail calls.

First, we modify how we transform each procedure. When generating code for a procedure, we cannot know (Rice's Theorem) whether it will need to return or not, *i.e.,* whether it will be called via a tail call or non-tail call. We therefore design our calling convention to enable any procedure to return.

We modify our calling convention, designating a register `current-return-address-register` in which to pass the *return address*, the label to which the procedure will jump after it is finish executing. On entry to any procedure, we load the `current-return-address-register` into a fresh abstract location, which we call *`aloc_tmp-ra`*. This is necessary since the `current-return-address-register` needs to be available immediately for any new calls, but we will not need the return address until the end of the procedure.

For convenience, we slightly generalize from procedures and define entry points that load the `current-return-address-register`. An *entry point* is the top-level *`tail`* expression that begins execution of code. This happens either as the body of a procedure, or as the initial *`tail`* in a module `(module tail)`.

> *Design digression:*
>
> This generalization lets us treat all returns, including returning the final value to the run-time system, uniformly, and allows us to eliminate the *`halt`* instruction. This isn't necessary; we could continue to support *`halt`* and treat the module-level entry point separate from procedures. However, there is no benefit to doing so, and it clutters the compiler with special cases. The only benefit of keeping *`halt`* would be saving a single jump instruction, but this has almost no cost, will probably be predicted by a CPU's branch predictor, and could easily be optimized away by a small pass nearly anywhere in the compiler pipeline.

By default, we use `r15` as the `current-return-address-register`.

Second, we need to modify non-tail calls to create the return address and update the `current-return-address-register` prior to jumping to the procedure. Since we do not have access to labels Imp-mf-lang v5, we will need to introduce an abstraction for creating a return address in our new intermediate language. After returning, the code at the return address will read from a designated register and continue the rest of the computation. We reuse the `current-return-value-register` as this designated register.

Finally, we need to explicitly return a value in *`tail`* position. Previously, the final value in *`tail`* position was also the final value of the program. This value was implicitly returned to the run-time system. However, now, a value in *`tail`* position may either be returned to the run-time system, or may be returned to some other computation from a non-tail call. We transform a value in *`tail`*

position by moving it into the `current-return-value-register`, and jumping to the return address stored in `aloc_tmp-ra`.

- When transforming an entry point *entry*, we generate:

  ```
  `(begin
    (set! ,tmp-ra ,(current-return-address-register))
    ,entry)
  ```

  where:

  - `tmp-ra` is a fresh abstract location used to store the return address for this entry point.

- When transforming a procedure `` `(lambda (,x_0 ... ,x_n-1 ,x_n ... ,x_n+k-1) ,body) ``, we generate:

  ```
  `(begin
     (set! ,x_0 ,r_0)
     ...
     (set! ,x_n-1 ,r_n-1)
     (set! ,x_n ,fv_0)
     ...
     (set! ,x_n+k-1 ,fv_k-1)
     ,body)
  ```

  where:

  - `fv_0 ... fv_k-1` are the first `k` frame variables.
  - `r_0 ... r_n-1` are the `n` physical locations from `current-parameter-registers`.

  The order of the `set!`s is not important for correctness, but can influence optimizations. We want to limit the live ranges of registers to help the register allocator make better use of registers, so we should generate accesses to registers first to limit their live ranges.

- When transforming a tail call, `` `(call ,v ,v_0 ... ,v_n-1 ,v_n ... ,v_n+k-1) ``, we generate:

  ```
  `(begin
     (set! ,fv_0 ,v_n) ...
     (set! ,fv_k-1 ,v_n+k-1) ...
     (set! ,r_0 ,v_0) ...
     (set! ,r_n-1 ,v_n-1) ...
     (set! ,ra ,tmp-ra)
     (jump ,v ,fbp ,ra ,r_0 ... ,r_n-1 ,fv_0 ... ,fv_k-1))
  ```

  where:

  - `fbp` is the physical location storing the frame base pointer, `current-`

`frame-base-pointer-register`.

- all other meta-variables are the same as in the case for transforming *`lambda`*.
- `tmp-ra` is the same fresh variable generated on entry to the current entry point.
- `ra` is the physical location storing the return address, `current-return-address-register`.

- When transforming a base *`value`* (either a *`triv`* or (*`binop opand opand`*)) *`value`* in *`tail`* position we generate:

```
`(begin
   (set! ,rv ,value)
   (jump ,tmp-ra ,fbp ,rv))
```

where:

- `rv` is the physical location used to store the return value, `current-return-value-register`.
- `fbp` is the physical location storing the frame base pointer, `current-frame-base-pointer-register`.
- `tmp-ra` is the designated abstract location created for the current entry point.

This explicitly returns the value to the current return address.

- When transforming a call in *non-tail position*, *i.e.,* a non-tail call `` `(call ,v ,v_0 ... ,v_n-1 ,v_n ... ,v_n+k-1) ``, we generate the following.

```
`(return-point ,rp-label
   (begin
     (set! ,nfv_0 ,v_n) ...
     (set! ,nfv_k-1 ,v_n+k-1) ...
     (set! ,r_0 ,v_0) ...
     (set! ,r_n-1 ,v_n-1) ...
     (set! ,ra ,rp-label)
     (jump ,v ,fbp ,ra ,r_0 ... ,r_n-1 ,nfv_0 ... ,nfv_k-1)))
```

where:

- `rp-label` is a fresh label.

  Intuitively, we need some way to introduce a label at *this instruction*, so when the call is complete we can return to the instruction after the call. This `return-point` will be a new instruction in the target language that introduces a fresh label `rp-label`.

Note that our non-tail calls can appear in value position, for example, as the right-hand side of a `(set! aloc value)` instruction. Recall that after the non-tail call, the return value of the procedure will be stored in `(current-return-value-register)`. When normalizing the `set!` instructions `canonicalize-bind`, we can translate this instruction as:

```
`(begin
   ,translation-of-non-tail-call
   (set! ,aloc ,rv))
```

- `nfv_0 ... nfv_k-1` should be the first `k` *frame variables*, *i.e.,* locations on the *callee's* frame.

However, we can't make these frame variables yet. These variables are assigned in the caller, and the callee may not have the same frame base as the caller. Consider the following expression with a non-tail call using frame variables.

```
`(begin
   (set! fv0 1)
   (set! rax 42)
   (return-point L.rp.1
     (set! fv0 rax)
     (set! r15 L.rp.1)
     (jump L.label.1 rbp r15 fv0))
   (set! r9 fv0)
   (set! rax (+ rax r9)))
```

For this example, suppose we've exhausted our `(current-parameter-registers)`, which we simulate by making it the empty set.

The frame location `'fv0` is live across the call; we use it after the call returns. However, we need to store `'rax` in the callee's `'fv0`. If we try to use frame variables directly, we overwrite the original value of `'fv0`. We first need to figure out how large the caller's frame is, and then move `'rax` to the index *after* the last frame location used by the caller.

Prior to undead analysis, we don't know how much we need to increment the base frame pointer to save variables live after this non-tail call. This makes figuring the exact index for these new frame variables non-trivial. We need to know the base of the callee's frame, while we only know the base of our frame, *i.e.,* the caller's frame.

But to do undead analysis, we really need both of those occurrences of `'fv0` to be unique. We've assumed all locations are uniquely identified by name, but the return point changes the interpretation of frame variables, so two occurrences of the same frame variable are not necessarily the same. We want to avoid introducing such an ambiguous location.

Instead, we'll generate something like this:

```
`(begin
   (set! fv0 1)
   (set! rax 42)
   (return-point L.rp.1
     (set! nfv.0 rax)
     (set! r15 L.rp.1)
     (jump L.label.1 rbp r15 nfv.0))
   (set! r9 fv0)
   (set! rax (+ rax r9)))
```

where *nfv.0* must be assigned to frame location 0 in the callee's frame. For now, we record these *new-frame variables* in an info field, and we'll figure out how to assign frames later.

Note that this is only a problem in a non-tail call. In a tail call, we can reuse the caller's frame as the callee's frame, since we never return.

4.8.4 Extending front-end with support for non-tail calls

As we don't require exposing any new low-level primitives, we modify our compiler proceeding top-down instead of bottom-up.

The main required changes are in the calling convention, so we begin by extending all the passes between the source language and `impose-calling-conventions` to support non-tail calls.

We first update `check-values-lang` to allow non-tail calls. The heuristics remain the same as in `v5:check-values-lang`.

(check-values-lang *p***)** → `values-lang-v6?` procedure

p : `values-lang-v6?`

Validates that the Values-lang v6 is well bound and well typed: all procedure calls pass the correct number of arguments, and all *binop* and *relop* are never used with labels.

Next, we extend `uniquify`. First, of course, we design the updated *Values-unique-lang v6*. We typeset the differences with respect to Values-unique-lang v5.

```
    p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc value] ...) pred)
        | (if pred pred pred)
```

```
 tail ::= value
        | (let ([aloc value] ...) tail)
        | (if pred tail tail)
        | (call triv opand ...)

value ::= triv
        | (binop opand opand)
        | (let ([aloc value] ...) value)
        | (if pred value value)
        | (call triv opand ...)+

opand ::= int64
        | aloc

 triv ::= opand
        | label

binop ::= *
        | +
        | -+

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

This requires no changes specific to non-tail calls, so the changes compared to `v5:uniquify` are trivial.

(**uniquify** *p*) → values-unique-lang-v6? procedure

p : values-lang-v6?

Compiles Values-lang v6 to Values-unique-lang v6 by resolving top-level lexical identifiers into unique labels, and all other lexical identifiers into unique abstract locations.

Finally, we expose non-tail calls through `sequentialize-let`. Below we define *Proc-imp-mf-lang v6*, where we transform lexical binding into sequential imperative assignments.

v5 Diff (excerpts) Full

```
p ::= (module (define label (lambda (aloc ...) entry+ tail-)) ... entry+ tail-)
```

```
entry+ ::= tail

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= value
        | (call triv opand ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
        | (binop opand opand)
        | (begin effect ... value)
        | (if pred value value)
        | (call triv opand ...)+

effect ::= (set! aloc value)
        | (begin effect ... effect)
        | (if pred effect effect)

binop ::= *
        | +
        | -+
```

Note that this language contains a definition *entry* designating the top-level tail used as the entry point for each procedure and for the module as a whole. There is no syntactic distinction, but making a semantic distinction will simplify our implementation of the calling convention to support return.

(sequentialize-let *p*) → proc-imp-mf-lang-v6? procedure
p : values-unique-lang-v6?

Compiles Values-unique-lang v6 to Proc-imp-mf-lang v6 by picking a particular order to implement `let` expressions using `set!`.

4.8.5 Extending Calling Convention

Next we design *Imp-mf-lang v6*, the target language of the calling convention translation. Below, we typeset the differences compared to Imp-mf-lang v5.

```
p ::= (module info+ (define label info+ tail) ... tail)
```

```
 info+ ::= (#:from-contract (info/c (new-frames (frame ...))))

frame+ ::= (aloc ...)

  pred ::= (relop opand opand)
         | (true)
         | (false)
         | (not pred)
         | (begin effect ... pred)
         | (if pred pred pred)

  tail ::= value-
         | (jump trg loc ...)
         | (begin effect ... tail)
         | (if pred tail tail)

 value ::= triv
         | (binop opand opand)
         | (begin effect ... value)
         | (if pred value value)
         | (return-point label tail)+

effect ::= (set! loc value)
         | (begin effect ... effect)
         | (if pred effect effect)

 opand ::= int64
         | loc

  triv ::= opand
         | label

   loc ::= rloc
         | aloc

   trg ::= label
         | loc

 binop ::= *
         | +
         | -+

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?
```

```
label ::= label?

 rloc ::= register?
        | fvar?
```

We now allow `(jump trg opand ...)` in *value* position. This corresponds to the addition of non-tail calls to the source language.

We also add the `return-point` form to effect context. This instruction introducing a new, non-top-level *label* in the middle of an instruction sequence, which is expected to be exclusively used by the calling convention to implement return. By introducing this new abstraction, we are required to implement this abstraction lower in the compiler pipeline.

We further assume that a `return-point` cannot appear inside another `return-point`, *i.e.,* there are no nested `return-points`. Our compiler can never generate this code, and there is no reason to support.

The implicit return value, *value* in *tail* position, is no longer valid. Instead, the run-time system will set the first return address, and the final result is returned to the run-time system using the calling conventions. The run-time system initializes the `current-return-address-register` to be the address of the exit procedure.

To implement return, we modify every entry point to store the `current-return-address-register` as described by our calling convention. Then we explicitly return the base expressions in *value* context.

To implement *fvar*s later, we require that `current-frame-base-pointer-register` is assigned only by incrementing or decrementing it by an integer literal. Other uses `current-frame-base-pointer-register` are *invalid programs*. Later passes will assume this in order to compute frame variable locations.

The *info* field records all the new frames created in the block, and will be used later to push new frames on to the stack, and assign new-frame variables to frame locations. There should be one frame for each non-tail call, even if that frame is empty. The new-frame variables should be in order. Each new-frame variable must only appear in one list in the `new-frames` field. Recall that *aloc*s are unique, and the `new-frames` field represents newly defined *aloc*s. It would not make sense for the same *aloc* to appear in two frames.

`(impose-calling-conventions p)` → `imp-mf-lang-v6?` `procedure`
  p : `proc-imp-mf-lang-v6?`

Compiles Proc-imp-mf-lang v6 to Imp-mf-lang v6 by imposing calling conventions on all calls (both tail and non-tail calls), and entry points. The registers used to passing parameters are defined by `current-`

`parameter-registers`, and the registers used for returning are defined by `current-return-address-register` and `current-return-value-register`.

After implementing the calling conventions, we have two abstractions that we need to implement.

First, we must implement frames, or more specifically, a stack of frames (also known as a stack). Non-tail calls cannot reuse their frame since some abstract locations may be live (will be undead) after the call. In general, there will not be enough registers to keep them all around, so we store them on the frame. But if the caller starts writing to the frame, it would overwrite live values. So we need to install a new frame for the caller before executing the non-tail call. We've already collected the new frame variables, and we need to modify the register allocator to determine the size and allocate new frames. This requires explicitly manipulating the frame base pointer, which also changes how frame variables are implemented.

Second, we need to implement `return-points`. These will be compiled to raw labels, so we essentially preserve them until a low-level language with access to raw labels.

4.8.6 Implementing A Stack of Frames

Our calling convention passes the first `n` arguments as registers, using the set of registers defined in the parameter `current-parameter-registers`. To deal with an arbitrary number of arguments, we may need more than the `n` registers we have available for parameters. For the rest, we use fresh frame locations.

Unfortunately, the callee will not know the exact offset into the frame that we, the caller, might be using. We therefore need to agree a priori on which frame locations are used across the call. Since there might be arbitrary frame locations currently in use (for example, because the register allocator will be putting some abstract locations on the frame), there is no particular index that it's safe to start from.

The solution is to introduce a stack of frames. Each function assumes that, prior to being called, it was given a *brand-new frame* from which it can start indexing at 0. The callee knows how many arguments it expected to receive, and can start counting from 0. The caller knows how many frame locations it has used, and can increment the frame base pointer beyond its own locations to a safe starting index. After the function call returns, the caller can decrement the frame base pointer by the same amount, restoring its own frame. These increment/decrement operations can be interpreted as "pushing" and "popping" a new frame on and off the *stack of frames* (which we will usually shorten to just *stack*).

This usage of stack differs from what x64 provides. We now require the stack to

be frame-aligned, and any accesses outside the current frame boundaries results in undefined behaviour.

To compute the size of each caller's frame, we need to know how many variables might be live across a call, so this must wait until after `undead-analysis`. To do a good job assigning these to frame locations, we also want to wait until after `conflict-analysis`, so we can try to assign non-conflicting variables to the same frame location.

4.8.6.1 Updating intermediate passes

Before allocating frames, there are a few passes we must update to pass through our new abstractions.

First, we extend `canonicalize-bind`. We define *Imp-cmf-lang v6* below. We typeset the differences compared to Imp-cmf-lang v5.

```
      p ::= (module info+ (define label info+ tail) ... tail)

  info+ ::= (#:from-contract (info/c (new-frames (frame ...))))

 frame+ ::= (aloc ...)

   pred ::= (relop opand opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= value⁻
          | (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

  value ::= triv
          | (binop opand opand)

 effect ::= (set! loc value)
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)+

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
```

```
        |  aloc

  trg ::= label
        |  loc

binop ::= *
        |  +
        |  -+

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        |  fvar?
```

We simply extend Imp-cmf-lang v5 with our new abstractions, including non-tail calls and return points. We also require the *new-frames* declaration in the *info* field.

`(canonicalize-bind p) → imp-cmf-lang-v6?` procedure

p : `imp-mf-lang-v6?`

Compiles Imp-mf-lang v6 to Imp-cmf-lang v6, pushing `set!` under `begin` so that the right-hand-side of each `set!` is base value-producing operation.

This canonicalizes Imp-mf-lang v6 with respect to the equations:

```
(set! aloc (begin
effect_1 ...          =(begin effect_1 ... (set! aloc value))
value))
(set! aloc (if pred   =(if pred (set! aloc value_1) (set! aloc
value_1 value_2))     value_2))
(set! aloc (return-   =(begin (return-point label tail) (set!
point label tail))    aloc (unquote (current-return-value-
                      register))))
```

Next we impose some machine restrictions on our language with `select-instructions`. We define *Asm-pred-lang v6* below, with changes typeset with respect to Asm-pred-lang v5.

```
  p ::= (module info (define label info tail) ... tail)
```

```
  info ::= (#:from-contract (info/c (new-frames (frame ...))))+
         | info?-

frame+ ::= (aloc ...)

  pred ::= (relop loc opand)
         | (true)
         | (false)
         | (not pred)
         | (begin effect ... pred)
         | (if pred pred pred)

  tail ::= (halt opand)-
         | (jump trg loc ...)
         | (begin effect ... tail)
         | (if pred tail tail)

effect ::= (set! loc triv)
         | (set! loc_1 (binop loc_1 opand))
         | (begin effect ... effect)
         | (if pred effect effect)
         | (return-point label tail)+

 opand ::= int64
         | loc

  triv ::= opand
         | label

   loc ::= rloc
         | aloc

   trg ::= label
         | loc

 binop ::= *
         | +
         | -+

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?
```

```
rloc ::= register?
       | fvar?
```

There are no new restrictions for `return-point`. We simply extend the pass to support *jumps* in effect context.

(**select-instructions** *p*) → asm-pred-lang-v6? procedure

p : imp-cmf-lang-v6?

Compiles Imp-cmf-lang v6 to Asm-pred-lang v6, selecting appropriate sequences of abstract assembly instructions to implement the operations of the source language.

4.8.6.2 Analyzing Return Points and Non-tail Calls

We first extend `uncover-locals` to find locals in non-tail calls and return points. Below we define *Asm-pred-lang-v6/locals*. We typeset changes compared to Asm-pred-lang v5/locals.

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (new-frames (frame
         ...))+ (locals (aloc ...))))

frame+ ::= (aloc ...)

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)
       | (begin effect ... pred)
       | (if pred pred pred)

tail ::= (halt opand)−
       | (jump trg loc ...)
       | (begin effect ... tail)
       | (if pred tail tail)

effect ::= (set! loc triv)
       | (set! loc_1 (binop loc_1 opand))
       | (begin effect ... effect)
       | (if pred effect effect)
       | (return-point label tail)+

opand ::= int64
       | loc

triv ::= opand
       | label
```

```
  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -+

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

Updating this analysis is not complicated. We simply add a case to handle `jump` in tail position, and to traverse return points. Remember that the "arguments" to `jump` are only used for later analyses and not consider locals for this analysis.

(uncover-locals *p*) → asm-pred-lang-v6/locals? procedure

p : asm-pred-lang-v6?

Compiles Asm-pred-lang v6 to Asm-pred-lang v6/locals, analysing which abstract locations are used in each block, and each block and the module with the set of variables in an `info?` fields.

The new-frame variables should be listed in the locals set for the enclosing block.

Next we extend `undead-analysis`. We design *Asm-pred-lang-v6/undead* below, typeset with respect to Asm-pred-lang v5/undead.

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (new-frames (frame
           ...))+ (locals (aloc ...)) (call-undead (loc
           ...))+ (undead-out undead-set-tree/rloc?)))

frame+ ::= (aloc ...)
```

```
 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (halt opand)⁻
        | (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)⁺

 opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

 binop ::= *
        | +
        | -⁺

 relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

 int64 ::= int64?

 aloc ::= aloc?

 label ::= label?

 rloc ::= register?
        | fvar?
```

We add two new *info* fields: `undead-out` and `call-undead`. The `undead-out` field will continue to store the undead-set tree, which we must update to track

`return-points`. The `call-undead` is the set of all locations that are live after *any* non-tail call in a block.

The `call-undead` field stores *every* abstract location or frame variable that is in the undead-out set of a return point. These must be allocated separately from other variables, so we store them separately.

First, we update the definition of `undead-set-tree?` to handle the `return-point` instruction, which includes a nested *`tail`*.

```
Undead-set-tree is one of:
- Undead-set
- (list Undead-set Undead-set-tree)
- (list Undead-set Undead-set-tree Undead-set-tree)
- (listof Undead-set-tree)

WARNING: datatype is non-canonical since Undead-set-tree can be an
         Undead-set, so third and fourth case, and the second and fourth case can
         overlap.
         An Undead-set-tree is meant to be traversed simultaneously with an
         Asm-pred-lang tail, so this ambiguity is not a problem.

interp. a tree of Undead-sets.
The structure of the tree mirrors the structure of a Asm-pred-lang tail.
There are three kinds of sub-trees:

(1) a set! node is simply an undead set;
(2) a return-point node is a list whose first element is the undead-set
    representing the undead-out, of the return-point (the locations undead after
    the call), and whose second element is the Undead-set-tree of the nested
    tail.
(3) an if node has an Undead-set for the predicate and two sub-trees for the
    branches.
(4) a begin node is a list of Undead-sets-trees, each corresponding to an
    instruction in a begin tail.
```

```
(define (undead-set-tree? ust)
  (match ust
    ; for an instruction
    [(? undead-set?) #t]
    ; for a return point
    [(list (? undead-set?) (? undead-set-tree?))]
    ; for an if
    [(list (? undead-set?) (? undead-set-tree?) (? undead-set-tree?)) #t]
    ; for a begin
    [`(,(? undead-set-tree?) ...) #t]
    [else #f]))
```

Analyzing non-tail jumps is no different from other jumps; we reuse the "arguments" annotated on the jump as the undead-out set, and discard the annotation.

Analyzing `return-point` requires making explicit a fact from our calling convention. After returning, we expect `current-return-value-register` to be

live. We model this as treating a `return-point` as assigning the `current-return-value-register`.

(**undead-analysis** *p*) → asm-pred-lang-v6/undead? procedure
p : asm-pred-lang-v6/locals?

Performs undead analysis, compiling Asm-pred-lang v6/locals to Asm-pred-lang v6/undead by decorating programs with their undead-set trees.

Examples:

```
> (pretty-display
   ((compose
     undead-analysis
     uncover-locals
     select-instructions
     canonicalize-bind
     impose-calling-conventions
     sequentialize-let)
    '(module
       (define L.swap.1
         (lambda (x.1 y.2)
           (if (< y.2 x.1)
               x.1
               (let ([z.3 (call L.swap.1 y.2 x.1)])
                 z.3))))
       (call L.swap.1 1 2))))
(module
  ((new-frames ())
   (locals (tmp-ra.2))
   (call-undead ())
   (undead-out
    ((tmp-ra.2 rbp)
     (tmp-ra.2 rsi rbp)
     (tmp-ra.2 rsi rdi rbp)
     (rsi rdi r15 rbp)
     (rsi rdi r15 rbp))))
  (define L.swap.1
    ((new-frames (()))
     (locals (z.3 tmp-ra.1 x.1 y.2))
     (undead-out
      ((rdi rsi tmp-ra.1 rbp)
       (rsi x.1 tmp-ra.1 rbp)
       (y.2 x.1 tmp-ra.1 rbp)
       ((y.2 x.1 tmp-ra.1 rbp)
        ((tmp-ra.1 rax rbp) (rax rbp))
        (((rax tmp-ra.1 rbp)
          ((y.2 rsi rbp) (rsi rdi rbp) (rsi rdi r15 rbp) (rsi rdi r15 rbp)))
```

```
          (z.3 tmp-ra.1 rbp)
          (tmp-ra.1 rax rbp)
          (rax rbp)))))
     (call-undead (tmp-ra.1)))
    (begin
      (set! tmp-ra.1 r15)
      (set! x.1 rdi)
      (set! y.2 rsi)
      (if (< y.2 x.1)
        (begin (set! rax x.1) (jump tmp-ra.1 rbp rax))
        (begin
          (return-point L.rp.1
            (begin
              (set! rsi x.1)
              (set! rdi y.2)
              (set! r15 L.rp.1)
              (jump L.swap.1 rbp r15 rdi rsi)))
          (set! z.3 rax)
          (set! rax z.3)
          (jump tmp-ra.1 rbp rax)))))
  (begin
    (set! tmp-ra.2 r15)
    (set! rsi 2)
    (set! rdi 1)
    (set! r15 tmp-ra.2)
    (jump L.swap.1 rbp r15 rdi rsi)))
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       undead-analysis
       uncover-locals
       select-instructions
       canonicalize-bind
       impose-calling-conventions
       sequentialize-let)
      '(module
         (define L.swap.1
           (lambda (x.1 y.2)
             (if (< y.2 x.1)
                 x.1
                 (let ([z.3 (call L.swap.1 y.2 x.1)])
                   z.3))))
         (call L.swap.1 1 2)))))
(module
  ((new-frames ())
   (locals (tmp-ra.6))
   (call-undead ())
   (undead-out
    ((tmp-ra.6 rbp)
     (tmp-ra.6 fv1 rbp)
```

```
    (tmp-ra.6 fv1 fv0 rbp)
    (fv1 fv0 r15 rbp)
    (fv1 fv0 r15 rbp))))
  (define L.swap.1
    ((new-frames ((nfv.4 nfv.5)))
     (locals (nfv.4 nfv.5 z.3 tmp-ra.3 x.1 y.2))
     (undead-out
      ((fv0 fv1 tmp-ra.3 rbp)
       (fv1 x.1 tmp-ra.3 rbp)
       (y.2 x.1 tmp-ra.3 rbp)
       ((y.2 x.1 tmp-ra.3 rbp)
        ((tmp-ra.3 rax rbp) (rax rbp))
        (((rax tmp-ra.3 rbp)
          ((y.2 nfv.5 rbp)
           (nfv.5 nfv.4 rbp)
           (nfv.5 nfv.4 r15 rbp)
           (nfv.5 nfv.4 r15 rbp)))
         (z.3 tmp-ra.3 rbp)
         (tmp-ra.3 rax rbp)
         (rax rbp)))))
     (call-undead (tmp-ra.3)))
    (begin
      (set! tmp-ra.3 r15)
      (set! x.1 fv0)
      (set! y.2 fv1)
      (if (< y.2 x.1)
        (begin (set! rax x.1) (jump tmp-ra.3 rbp rax))
        (begin
          (return-point L.rp.2
            (begin
              (set! nfv.5 x.1)
              (set! nfv.4 y.2)
              (set! r15 L.rp.2)
              (jump L.swap.1 rbp r15 nfv.4 nfv.5)))
          (set! z.3 rax)
          (set! rax z.3)
          (jump tmp-ra.3 rbp rax)))))
  (begin
    (set! tmp-ra.6 r15)
    (set! fv1 2)
    (set! fv0 1)
    (set! r15 tmp-ra.6)
    (jump L.swap.1 rbp r15 fv0 fv1)))
```

The following example shows the output on an intermediate representation of non-tail recursive factorial, compiled with `current-parameter-registers` set to `'()` to force the compiler to generate frame variables and test edge cases.

Example:

```
> (pretty-display
```

```
    (undead-analysis
     '(module
        ((new-frames ()) (locals (ra.12)))
        (define L.fact.4
          ((new-frames ((nfv.16)))
           (locals (ra.13 x.9 tmp.14 tmp.15 new-n.10 nfv.16 factn-1.11 tmp.17)))
          (begin
            (set! x.9 fv0)
            (set! ra.13 r15)
            (if (= x.9 0)
                (begin (set! rax 1) (jump ra.13 rbp rax))
                (begin
                  (set! tmp.14 -1)
                  (set! tmp.15 x.9)
                  (set! tmp.15 (+ tmp.15 tmp.14))
                  (set! new-n.10 tmp.15)
                  (return-point
                      L.rp.6
                    (begin
                      (set! nfv.16 new-n.10)
                      (set! r15 L.rp.6)
                      (jump L.fact.4 rbp r15 nfv.16)))
                  (set! factn-1.11 rax)
                  (set! tmp.17 x.9)
                  (set! tmp.17 (* tmp.17 factn-1.11))
                  (set! rax tmp.17)
                  (jump ra.13 rbp rax)))))
        (begin
          (set! ra.12 r15)
          (set! fv0 5)
          (set! r15 ra.12)
          (jump L.fact.4 rbp r15 fv0)))))
(module
  ((new-frames ())
   (locals (ra.12))
   (call-undead ())
   (undead-out ((ra.12 rbp) (ra.12 fv0 rbp) (fv0 r15 rbp) (fv0 r15 rbp))))
  (define L.fact.4
    ((new-frames ((nfv.16)))
     (locals (ra.13 x.9 tmp.14 tmp.15 new-n.10 nfv.16 factn-1.11 tmp.17))
     (undead-out
      ((r15 x.9 rbp)
       (x.9 ra.13 rbp)
       ((x.9 ra.13 rbp)
        ((ra.13 rax rbp) (rax rbp))
        ((tmp.14 x.9 ra.13 rbp)
         (tmp.14 tmp.15 x.9 ra.13 rbp)
         (tmp.15 x.9 ra.13 rbp)
         (new-n.10 x.9 ra.13 rbp)
         ((rax x.9 ra.13 rbp) ((nfv.16 rbp) (nfv.16 r15 rbp) (nfv.16 r15 rbp)))
```

```
          (x.9 factn-1.11 ra.13 rbp)
          (factn-1.11 tmp.17 ra.13 rbp)
          (tmp.17 ra.13 rbp)
          (ra.13 rax rbp)
          (rax rbp)))))
      (call-undead (x.9 ra.13)))
    (begin
      (set! x.9 fv0)
      (set! ra.13 r15)
      (if (= x.9 0)
        (begin (set! rax 1) (jump ra.13 rbp rax))
        (begin
          (set! tmp.14 -1)
          (set! tmp.15 x.9)
          (set! tmp.15 (+ tmp.15 tmp.14))
          (set! new-n.10 tmp.15)
          (return-point L.rp.6
            (begin
              (set! nfv.16 new-n.10)
              (set! r15 L.rp.6)
              (jump L.fact.4 rbp r15 nfv.16)))
          (set! factn-1.11 rax)
          (set! tmp.17 x.9)
          (set! tmp.17 (* tmp.17 factn-1.11))
          (set! rax tmp.17)
          (jump ra.13 rbp rax)))))
  (begin
    (set! ra.12 r15)
    (set! fv0 5)
    (set! r15 ra.12)
    (jump L.fact.4 rbp r15 fv0)))
```

Next we update the `conflict-analysis`. Below, we define *Asm-pred-lang v6/conflicts*, typeset with differences compared to Asm-pred-lang v5/conflicts.

```
       p ::= (module info (define label info tail) ... tail)

    info ::= (#:from-contract (info/c (new-frames (frame
             ...))+ (locals (aloc ...)) (call-undead (loc
             ...))+ (undead-out undead-set-
             tree/rloc?)+ (conflicts ((loc (loc ...)) ...))))

  frame+ ::= (aloc ...)

    pred ::= (relop loc opand)
           | (true)
           | (false)
           | (not pred)
           | (begin effect ... pred)
           | (if pred pred pred)
```

```
 tail ::= (halt opand)⁻
        | (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)⁺

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -⁺

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

We need to assign the new-frame variables to frame locations. However, we also reuse frame locations when possible, to minimize the size of frame and thus memory usage.

This is straightforward to solve. We run `conflict-analysis`, but also collect conflicts between abstract locations and physical locations.

Recall that `current-return-value-register` is assigned by a non-tail call. Also note that `current-frame-base-pointer-register` and `current-return-value-`

`register` are likely to end up in conflict with everything, even though we have removed them from the `current-assignable-registers` set.

The interpretation of the conflict graph will be somewhat more difficult than in prior versions. It might contain conflicts between physical locations, which will never matter since we don't try to assign physical locations.

> If our frame allocation was more clever, we would need to adjust the conflict analysis to make all caller saved registers in conflict with a non-tail call. However, we instead assign all call-undead variables to the frame, so we don't need to do very much for non-tail calls.

```
(conflict-analysis p) → asm-pred-lang-v6/conflicts?   procedure
  p : asm-pred-lang-v6/undead?
```

Performs conflict analysis, compiling Asm-pred-lang v6/undead to Asm-pred-lang v6/conflicts by decorating programs with their conflict graph.

4.8.6.3 Frame Allocation

The size of a frame `n` (in slots) for a given non-tail call is one more than the maximum of:

- the number of locations in the undead-out set for the non-tail call, or
- the index of the largest frame location in the undead-out set for the non-tail call.

The frame for the call must save all location live across the call, since the caller might overwrite any register. Since we're allowing physical locations in our source language, we could have a frame variable live after a call, and must preserve up to that index.

We can model this as follows, although our implementation will be simpler as we discuss. Prior to the call, we push all locations live across the all onto the frame, then increment the base frame pointer. After the call, we decrement the base frame pointer, restoring the caller's frame, and load all locations live after the call from the frame.

> In practice, calling conventions distinguish between two sets of registers: callee-saved and caller-saved, to allow some registers to be live across a call. We ignore this for simplicity and assume all registers are caller-saved.

Intuitively, we want to transform `` `(return-point ,rp ,tail) `` into:

```
`(begin
   (set! ,nfv_0 ,x_0)
   ...
   (set! ,nfv_n-1 ,x_n-1)

   (set! ,fbp (+ ,fbp ,nb))
   (return-point ,rp ,tail)
   (set! ,fbp (- ,fbp ,nb))

   (set! ,x_0 ,nfv_0)
   ...
   (set! ,x_n-1 ,nfv_n-1))
```

where:

- `nb` is the number of bytes required to save `n` slots on the frame, *i.e.,* `(* n (current-word-size-bytes))`.
- `fbp` is the value of the parameter `current-frame-base-pointer-register`.
- `x_0`, ... `x_n` are the locations in the undead-out set for the non-tail call.
- `nfv_0`, ... `nfv_n-1` are `n` free frame variables.

Unfortunately, we can't implement this transformation as it. We want to avoid producing new `set!`s. First, they will invalidate the undead analysis we've just performed. Second, some or all the moves might be unnecessary. We don't know whether those variables `'x_0 ... 'x_n-1` need to be in registers, so it would be potentially more efficient to assign them to frame locations in the first place and leave some other pass to move them into registers when necessary.

Instead, we perform this transformation in two steps. First, a new pass `assign-call-undead-variables` assigns each location that is live across a call to frame locations, instead of producing new moves. This produces a partial assignment of abstract locations to frame variables, which the register allocator will work from. Second, a new pass, `allocate-frames` does the work of updating the frame base pointer, effectively allocating a frame for each non-tail call.

We define *Asm-pred-lang-v6/pre-framed* below, typeset with changes with respect to Asm-pred-lang-v6/conflicts.

```
info ::= (#:from-contract (info/c (new-
          frames (frame ...)) (locals (aloc ...)) (call-
          undead (loc ...)) (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment
          ((aloc loc) ...))+))
```

The core of `assign-call-undead-variables` is similar to `assign-registers`. The core algorithm is a straight-forward recursion over the `call-undead` set, and produces an `assignment`.

- If the input set is empty, return the default `assignment`.
- Choose a variable, `x`, from the input set of variables.
- Recur with `x` removed from the input set and the conflict graph. The recursive call should return an assignment for all the remaining variables.
- Select a compatible frame variable for `x`.

  A variable `x` is compatible with a frame location *`fvar_i`* if it is not directly in conflict with *`fvar_i`*, and it is not in conflict with a variable `y` that has been assigned to *`fvar_i`*.

  An easy way to find a compatible frame variable is to find the set of frame variables to which `x` cannot be assigned. Then, starting from *`fv0`*, assign the first frame variable that is not in the incompatible set.

  Finally, add the assignment for the `x` to the result of the recursive call.

The default *`assignment`* for this pass is the empty *`assignment`*, since nothing has been assigned yet.

Since many frame variables will be assigned prior to register allocation, you will need to modify `assign-registers` to use a similar algorithm for spilling, instead of a naive algorithm that starts spilling at frame location 0.

(**assign-call-undead-variables** *p*) → asm-pred-lang-v6/pre-framed?
p : asm-pred-lang-v6/conflicts?

Compiles Asm-pred-lang-v6/conflicts to Asm-pred-lang-v6/pre-framed by pre-assigning all variables in the *`call-undead`* sets to to frame locations.

Example:

```
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       assign-call-undead-variables
       conflict-analysis
       undead-analysis
       uncover-locals
       select-instructions
       canonicalize-bind
       impose-calling-conventions
       sequentialize-let)
      '(module
         (define L.swap.1
           (lambda (x.1 y.2)
             (if (< y.2 x.1)
```

```
                x.1
                (let ([z.3 (call L.swap.1 y.2 x.1)])
                  z.3))))
          (call L.swap.1 1 2)))))
(module
  ((new-frames ())
   (locals (tmp-ra.10))
   (call-undead ())
   (undead-out
    ((tmp-ra.10 rbp)
     (tmp-ra.10 fv1 rbp)
     (tmp-ra.10 fv1 fv0 rbp)
     (fv1 fv0 r15 rbp)
     (fv1 fv0 r15 rbp)))
   (conflicts
    ((tmp-ra.10 (fv0 fv1 rbp))
     (rbp (r15 fv0 fv1 tmp-ra.10))
     (fv1 (r15 fv0 rbp tmp-ra.10))
     (fv0 (r15 rbp fv1 tmp-ra.10))
     (r15 (rbp fv0 fv1))))
   (assignment ()))
  (define L.swap.1
    ((new-frames ((nfv.8 nfv.9)))
     (locals (y.2 x.1 z.3 nfv.9 nfv.8))
     (undead-out
      ((fv0 fv1 tmp-ra.7 rbp)
       (fv1 x.1 tmp-ra.7 rbp)
       (y.2 x.1 tmp-ra.7 rbp)
       ((y.2 x.1 tmp-ra.7 rbp)
        ((tmp-ra.7 rax rbp) (rax rbp))
        (((rax tmp-ra.7 rbp)
          ((y.2 nfv.9 rbp)
           (nfv.9 nfv.8 rbp)
           (nfv.9 nfv.8 r15 rbp)
           (nfv.9 nfv.8 r15 rbp)))
         (z.3 tmp-ra.7 rbp)
         (tmp-ra.7 rax rbp)
         (rax rbp)))))
     (call-undead (tmp-ra.7))
     (conflicts
      ((y.2 (rbp tmp-ra.7 x.1 nfv.9))
       (x.1 (y.2 rbp tmp-ra.7 fv1))
       (tmp-ra.7 (y.2 x.1 rbp fv1 fv0 rax z.3))
       (z.3 (rbp tmp-ra.7))
       (nfv.9 (r15 nfv.8 rbp y.2))
       (nfv.8 (r15 rbp nfv.9))
       (rbp (y.2 x.1 tmp-ra.7 rax z.3 r15 nfv.8 nfv.9))
       (r15 (rbp nfv.8 nfv.9))
       (rax (rbp tmp-ra.7))
       (fv0 (tmp-ra.7))
```

```
        (fv1 (x.1 tmp-ra.7))))
     (assignment ((tmp-ra.7 fv2))))
    (begin
      (set! tmp-ra.7 r15)
      (set! x.1 fv0)
      (set! y.2 fv1)
      (if (< y.2 x.1)
        (begin (set! rax x.1) (jump tmp-ra.7 rbp rax))
        (begin
          (return-point L.rp.3
            (begin
              (set! nfv.9 x.1)
              (set! nfv.8 y.2)
              (set! r15 L.rp.3)
              (jump L.swap.1 rbp r15 nfv.8 nfv.9)))
          (set! z.3 rax)
          (set! rax z.3)
          (jump tmp-ra.7 rbp rax)))))
  (begin
    (set! tmp-ra.10 r15)
    (set! fv1 2)
    (set! fv0 1)
    (set! r15 tmp-ra.10)
    (jump L.swap.1 rbp r15 fv0 fv1)))
```

Now we can allocate frames for each non-tail call. For each block, we compute the size of the frames for *all* non-tail call, and adjust each. The size `n` is one plus the maximum of the index of all frame variables in the `call-undead` set for the enclosing block.

To adjust the callee's frame, we transform `` `(return-point ,rp ,tail) `` into

```
`(begin
   (set! ,fbp (- ,fbp ,nb))
   (return-point ,rp ,tail)
   (set! ,fbp (+ ,fbp ,nb)))
```

where:

- `nb` is the number of bytes required to save `n` slots on the frame, *i.e.,* `(* n (current-word-size-bytes))`.
- `fbp` is `(current-frame-base-pointer-register)`.

Recall that the stack grows downward, so we *subtract* `nb` bytes from the current frame base pointer to allocate a frame of with `n` slots.

> We could allocate a different sized frame for each call, but this would require associating `call-undead` sets with each return point, and complicate the assignment of new-frame variables.

We also assign each of the new-frame variables from the *new-frame* lists. In order, each new-frame variable is assigned to a frame variable starting with `(make-fvar n)`. These assignments are added to the `assignment` field the enclosing block.

The output is *Asm-pred-lang-v6/framed*, which only changes in its info fields compared to Asm-pred-lang-v6/pre-framed.

```
info ::= (#:from-contract (info/c (new-frames (frame ...))⁻ (locals (aloc ...)) (call-
          undead (loc ...))⁻ (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))
```

The only differences are in the info field. The call-undead sets and new-frame fields are removed.

We remove the assigned variables from the locals set, to allow later passes to assume the locals set are all unassigned.

(allocate-frames *p*) → asm-pred-lang-v6/framed? procedure
p : asm-pred-lang-v6/pre-framed?

Compiles Asm-pred-lang-v6/pre-framed to Asm-pred-lang-v6/framed by allocating frames for each non-tail call, and assigning all new-frame variables to frame variables in the new frame.

Example:

```
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       allocate-frames
       assign-call-undead-variables
       conflict-analysis
       undead-analysis
       uncover-locals
       select-instructions
       canonicalize-bind
       impose-calling-conventions
       sequentialize-let)
      '(module
         (define L.swap.1
           (lambda (x.1 y.2)
             (if (< y.2 x.1)
                 x.1
                 (let ([z.3 (call L.swap.1 y.2 x.1)])
                   z.3))))
         (call L.swap.1 1 2)))))
(module
  ((locals (tmp-ra.14))
   (undead-out
```

```
   ((tmp-ra.14 rbp)
    (tmp-ra.14 fv1 rbp)
    (tmp-ra.14 fv1 fv0 rbp)
    (fv1 fv0 r15 rbp)
    (fv1 fv0 r15 rbp)))
  (conflicts
   ((tmp-ra.14 (fv0 fv1 rbp))
    (rbp (r15 fv0 fv1 tmp-ra.14))
    (fv1 (r15 fv0 rbp tmp-ra.14))
    (fv0 (r15 rbp fv1 tmp-ra.14))
    (r15 (rbp fv0 fv1))))
  (assignment ()))
 (define L.swap.1
   ((locals (z.3 x.1 y.2))
    (undead-out
     ((fv0 fv1 tmp-ra.11 rbp)
      (fv1 x.1 tmp-ra.11 rbp)
      (y.2 x.1 tmp-ra.11 rbp)
      ((y.2 x.1 tmp-ra.11 rbp)
       ((tmp-ra.11 rax rbp) (rax rbp))
       (((rax tmp-ra.11 rbp)
         ((y.2 nfv.13 rbp)
          (nfv.13 nfv.12 rbp)
          (nfv.13 nfv.12 r15 rbp)
          (nfv.13 nfv.12 r15 rbp)))
        (z.3 tmp-ra.11 rbp)
        (tmp-ra.11 rax rbp)
        (rax rbp)))))
    (conflicts
     ((y.2 (rbp tmp-ra.11 x.1 nfv.13))
      (x.1 (y.2 rbp tmp-ra.11 fv1))
      (tmp-ra.11 (y.2 x.1 rbp fv1 fv0 rax z.3))
      (z.3 (rbp tmp-ra.11))
      (nfv.13 (r15 nfv.12 rbp y.2))
      (nfv.12 (r15 rbp nfv.13))
      (rbp (y.2 x.1 tmp-ra.11 rax z.3 r15 nfv.12 nfv.13))
      (r15 (rbp nfv.12 nfv.13))
      (rax (rbp tmp-ra.11))
      (fv0 (tmp-ra.11))
      (fv1 (x.1 tmp-ra.11))))
    (assignment ((tmp-ra.11 fv2) (nfv.12 fv3) (nfv.13 fv4))))
   (begin
     (set! tmp-ra.11 r15)
     (set! x.1 fv0)
     (set! y.2 fv1)
     (if (< y.2 x.1)
       (begin (set! rax x.1) (jump tmp-ra.11 rbp rax))
       (begin
         (begin
           (set! rbp (- rbp 24))
```

```
            (return-point L.rp.4
              (begin
                (set! nfv.13 x.1)
                (set! nfv.12 y.2)
                (set! r15 L.rp.4)
                (jump L.swap.1 rbp r15 nfv.12 nfv.13)))
            (set! rbp (+ rbp 24)))
          (set! z.3 rax)
          (set! rax z.3)
          (jump tmp-ra.11 rbp rax)))))
  (begin
    (set! tmp-ra.14 r15)
    (set! fv1 2)
    (set! fv0 1)
    (set! r15 tmp-ra.14)
    (jump L.swap.1 rbp r15 fv0 fv1)))
```

Example:

```
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       conflict-analysis
       undead-analysis
       uncover-locals
       select-instructions
       canonicalize-bind
       impose-calling-conventions
       sequentialize-let)
      '(module
         (define L.swap.1
           (lambda (x.1 y.2)
             (if (< y.2 x.1)
                 x.1
                 (let ([z.3 (call L.swap.1 y.2 x.1)])
                   z.3))))
         (call L.swap.1 1 2)))))
(module
  ((new-frames ())
   (locals (tmp-ra.18))
   (call-undead ())
   (undead-out
    ((tmp-ra.18 rbp)
     (tmp-ra.18 fv1 rbp)
     (tmp-ra.18 fv1 fv0 rbp)
     (fv1 fv0 r15 rbp)
     (fv1 fv0 r15 rbp)))
   (conflicts
    ((tmp-ra.18 (fv0 fv1 rbp))
     (rbp (r15 fv0 fv1 tmp-ra.18))
     (fv1 (r15 fv0 rbp tmp-ra.18))
```

```
   (fv0 (r15 rbp fv1 tmp-ra.18))
   (r15 (rbp fv0 fv1)))))
(define L.swap.1
  ((new-frames ((nfv.16 nfv.17)))
   (locals (nfv.16 nfv.17 z.3 tmp-ra.15 x.1 y.2))
   (undead-out
    ((fv0 fv1 tmp-ra.15 rbp)
     (fv1 x.1 tmp-ra.15 rbp)
     (y.2 x.1 tmp-ra.15 rbp)
     ((y.2 x.1 tmp-ra.15 rbp)
      ((tmp-ra.15 rax rbp) (rax rbp))
      (((rax tmp-ra.15 rbp)
        ((y.2 nfv.17 rbp)
         (nfv.17 nfv.16 rbp)
         (nfv.17 nfv.16 r15 rbp)
         (nfv.17 nfv.16 r15 rbp)))
       (z.3 tmp-ra.15 rbp)
       (tmp-ra.15 rax rbp)
       (rax rbp)))))
   (call-undead (tmp-ra.15))
   (conflicts
    ((y.2 (rbp tmp-ra.15 x.1 nfv.17))
     (x.1 (y.2 rbp tmp-ra.15 fv1))
     (tmp-ra.15 (y.2 x.1 rbp fv1 fv0 rax z.3))
     (z.3 (rbp tmp-ra.15))
     (nfv.17 (r15 nfv.16 rbp y.2))
     (nfv.16 (r15 rbp nfv.17))
     (rbp (y.2 x.1 tmp-ra.15 rax z.3 r15 nfv.16 nfv.17))
     (r15 (rbp nfv.16 nfv.17))
     (rax (rbp tmp-ra.15))
     (fv0 (tmp-ra.15))
     (fv1 (x.1 tmp-ra.15)))))
  (begin
    (set! tmp-ra.15 r15)
    (set! x.1 fv0)
    (set! y.2 fv1)
    (if (< y.2 x.1)
      (begin (set! rax x.1) (jump tmp-ra.15 rbp rax))
      (begin
        (return-point L.rp.5
          (begin
            (set! nfv.17 x.1)
            (set! nfv.16 y.2)
            (set! r15 L.rp.5)
            (jump L.swap.1 rbp r15 nfv.16 nfv.17)))
        (set! z.3 rax)
        (set! rax z.3)
        (jump tmp-ra.15 rbp rax)))))
(begin
  (set! tmp-ra.18 r15)
```

```
    (set! fv1 2)
    (set! fv0 1)
    (set! r15 tmp-ra.18)
    (jump L.swap.1 rbp r15 fv0 fv1)))
```

Because the frame allocator sits in the middle of our register allocation pipeline, the optimized allocator `assign-homes-opt` is no longer a drop-in replacement for the naive `assign-homes`. We therefore remove `assign-homes-opt` and inline the passes in the `current-pass-list`.

4.8.6.4 Adjusting the Register Allocator

Frames are now implemented and all new-frame variables and variables live across a call are assigned to frame locations. We need to adjust our register allocator so that it does not try to spill variables into frame variables that are already taken.

To do this, we essentially remove spilling from `assign-registers`. Instead, in the output, the `locals` set should include only spilled locations. A separate pass (which looks suspiciously like `assign-call-undead-variables`) handles spilling.

Asm-pred-lang-v6/spilled is defined below, only changed by allowing assignments to arbitrary *rlocs*. We typeset the differences with respect to Asm-pred-lang-v6/framed.

```
info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
         tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))
```

(assign-registers *p*) → asm-pred-lang-v5/spilled? procedure
p : asm-pred-lang-v6/framed?

Performs graph-colouring register allocation, compiling Asm-pred-lang v6/framed to Asm-pred-lang v6/spilled by decorating programs with their register assignments.

The final change to the register allocator is to assign spilled variables to frame locations.

The new language, *Asm-pred-lang-v6/assignments*, is the familiar output of our register allocation process, which has all abstract locations assigned to physical locations.

```
info ::= (#:from-contract (info/c (locals (aloc ...))⁻ (undead-out
         undead-set-tree/rloc?)⁻ (conflicts ((loc (loc ...))
         ...))⁻ (assignment ((aloc loc) ...))))
```

After assigning frame variables, we can discard all the assorted info fields and

keep only the `assignment`.

`(assign-frame-variables p)` → `asm-pred-lang-v6/assignments?` procedure

`p` : `asm-pred-lang-v6/spilled?`

Compiles Asm-pred-lang-v6/spilled to Asm-pred-lang-v6/assignments by allocating all abstract locations in the locals set to free frame locations.

Finally, we actually replace abstract locations with physical locations. Below we define *Nested-asm-lang-fvars v6*, typeset with differences compared to Nested-asm-lang v5.

v5 Diff (excerpts) vs Source Diff (excerpts) Full

```
      p ::= (module (define label tail) ... tail)

   tail ::= (halt opand)⁻
          | (jump trg)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)⁺

  binop ::= *
          | +
          | -⁺
```

We need to update the pass to handle `return-points`.

`(replace-locations p)` → `nested-asm-lang-fvars-v6?` procedure

`p` : `asm-pred-lang-v6/assignments?`

Compiles Asm-pred-lang v6/assignments to Nested-asm-lang-fvars v6 by replacing all abstract location with physical locations using the assignment described in the `assignment` info field.

Example:

```
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       replace-locations
       assign-frame-variables
```

```
        assign-registers
        allocate-frames
        assign-call-undead-variables
        conflict-analysis
        undead-analysis
        uncover-locals
        select-instructions
        canonicalize-bind
        impose-calling-conventions
        sequentialize-let)
       '(module
          (define L.swap.1
            (lambda (x.1 y.2)
              (if (< y.2 x.1)
                  x.1
                  (let ([z.3 (call L.swap.1 y.2 x.1)])
                    z.3))))
          (call L.swap.1 1 2)))))
(module
  (define L.swap.1
    (begin
      (set! fv2 r15)
      (set! r14 fv0)
      (set! r15 fv1)
      (if (< r15 r14)
        (begin (set! rax r14) (jump fv2))
        (begin
          (begin
            (set! rbp (- rbp 24))
            (return-point L.rp.6
              (begin
                (set! fv4 r14)
                (set! fv3 r15)
                (set! r15 L.rp.6)
                (jump L.swap.1)))
            (set! rbp (+ rbp 24)))
          (set! r15 rax)
          (set! rax r15)
          (jump fv2)))))
  (begin
    (set! r15 r15)
    (set! fv1 2)
    (set! fv0 1)
    (set! r15 r15)
    (jump L.swap.1)))
```

4.8.7 Adjusting Frame Variables

We changed the invariants on *fbp*, the `current-frame-base-pointer-`

`register`, when added the stack of frames. We now allow it to be incremented and decremented by an integer literal. This affects how we implement frame variables.

Previously, the frame variables *fv1* represented the address `(fbp - 8)` in all contexts. However, after now the compilation is non-trivial, as it must be aware of increments and decrements to the *fbp*.

Consider the example snippet

```
`(begin
   (set! rbp (- rbp 8))
   (return-point L.rp.8
     (begin
       (set! rdi fv3)
       (jump L.f.1)))
   (set! rbp (+ rbp 8)))
```

In this example, the frame variable *fv3* is being passed to the procedure *L.f.1* in a non-tail call. *fv3* does not refer to 3rd frame variable on callee's, but the 3rd frame variable on the caller's frame. Since the frame is allocated prior to the return point, we need to fix-up this index by translating frame variables relative to frame allocations introduced around return points.

To do this, we change `implement-fvars` to be aware of the current *fbp* offset. The simplest way to do this is to relocate `implement-fvars` in the compiler pipeline, to before `expose-basic-blocks`. This allows the compiler to make use of the nesting structure of the program while tracking changes to *fbp*.

To update `implement-fvars`, we need to keep an accumulator of the current offset from the base of the frame. On entry to a block, frame variables start indexing from the base of the frame, so the offset is 0. So, *fv3* corresponds to `(fbp - 24)` (`(- (* 3 (current-word-size-bytes)) 0)`). After "pushing" or allocating a frame, such as `(set! fbp (- fbp 8))`, *fv3* corresponds to `(fbp - 16)` (`(+ (* 3 (current-word-size-bytes)) -8)`). After "popping" or deallocating a frame, such as `(set! fbp (+ fbp 8))` *fv3* corresponds to `(fbp - 24)` (`(+ (* 3 (current-word-size-bytes)) 0)`) again.

Recall that *fbp* is only incremented or decremented by integer literal values, like those generated by `allocate-frames`. Other assignments to *fbp* are invalid programs. This means we don't have to consider complicated data flows into *fbp*.

The source language for `implement-fvars`, *Nested-asm-lang-fvars v6*, is defined below typeset with respect to *Nested-asm-lang v5*.

```
      p ::= (module (define label tail) ... tail)

   pred ::= (relop loc opand)
          | (true)
          | (false)
```

```
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (halt opand)⁻
        |  (jump trg)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)⁺

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   trg ::= label
        |  loc

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +
        |  -⁺

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
```

```
        |  !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

The language does not change much, only adding a new *binop*.

The target language simply changes *fvar*s to *addr*s. We define *Nested-asm-lang-v6* below.

```
           p ::= (module (define label tail) ... tail)

        pred ::= (relop loc opand)
               |  (true)
               |  (false)
               |  (not pred)
               |  (begin effect ... pred)
               |  (if pred pred pred)

        tail ::= (jump trg)
               |  (begin effect ... tail)
               |  (if pred tail tail)

      effect ::= (set! loc triv)
               |  (set! loc_1 (binop loc_1 opand))
               |  (begin effect ... effect)
               |  (if pred effect effect)
               |  (return-point label tail)

      triv+ ::= opand
               |  label

       opand ::= int64
               |  loc

       triv- ::= opand
               |  label

         trg ::= label
               |  loc

         loc ::= reg
               |  addr+
               |  fvar-

         reg ::= rsp
               |  rbp
               |  rax
```

```
              |  rbx
              |  rcx
              |  rdx
              |  rsi
              |  rdi
              |  r8
              |  r9
              |  r12
              |  r13
              |  r14
              |  r15

       binop ::= *
              |  +
              |  -

       relop ::= <
              |  <=
              |  =
              |  >=
              |  >
              |  !=

       int64 ::= int64?

        aloc ::= aloc?

      addr+  ::= (fbp - dispoffset)

       fbp+  ::= frame-base-pointer-register?

dispoffset+  ::= dispoffset?

       fvar-  ::= fvar?

       label ::= label?
```

All languages following this pass need to be updated to use *addr*s instead of *fvars*. This should not affect most passes.

(**implement-fvars** *p*) → nested-asm-lang-v6? procedure
p : nested-asm-lang-fvars-v6?

Reifies *fvar*s into displacement mode operands.

4.8.8 Implementing Return Points

Finally, to accommodate non-tail calls, we introduced a new abstraction: return points. We must now implement this abstraction.

To implement return points, we need to compile all the instructions following the return points into labelled blocks, since that is our low-level implementation of labels. We lift all the instructions following the return point in to a new block, and merge the tail implementing the call into the `begin` of the caller. Essentially, we transform:

```
`(begin
   ,ss1 ...
   ,(return-point ,rp ,tail)
   ,ss2 ...)
```

into:

```
`(define ,rp (begin ,ss2 ...))
`(begin
   ,ss1 ...
   ,tail)
```

This transformation is part of the `expose-basic-blocks` pass, which lifts many inline blocks into top-level explicitly labelled blocks, and should now do the same for return points.

The target language of the transformation is *Block-pred-lang v6*, defined below as a change over Block-pred-lang v5.

```
          p ::= (module b ... b)

          b ::= (define label tail)

       pred ::= (relop loc opand)
              | (true)
              | (false)
              | (not pred)

       tail ::= (halt opand)⁻
              | (jump trg)
              | (begin s ... tail)
              | (if pred (jump trg) (jump trg))

          s ::= (set! loc triv)
              | (set! loc_1 (binop loc_1 opand))

      triv⁺ ::= opand
              | label

      opand ::= int64
              | loc

       triv⁻ ::= opand
              | label

        trg ::= label
```

```
          |  loc

    loc ::= reg
          |  addr+
          |  fvar−

    reg ::= rsp
          |  rbp
          |  rax
          |  rbx
          |  rcx
          |  rdx
          |  rsi
          |  rdi
          |  r8
          |  r9
          |  r12
          |  r13
          |  r14
          |  r15

  binop ::= *
          |  +
          |  -+

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int64 ::= int64?

   aloc ::= aloc?

 addr+ ::= (fbp - dispoffset)

  fbp+ ::= frame-base-pointer-register?

dispoffset+ ::= dispoffset?

  fvar− ::= fvar?

  label ::= label?
```

There are no major differences, since we are compiling a new abstraction into an old one. Note that only - has been added to the *binops*.

(**expose-basic-blocks** *p*) → block-pred-lang-v6? procedure

p : nested-asm-lang-v6?

Compile the Nested-asm-lang v6 to Block-pred-lang v6, eliminating all nested expressions by generate fresh basic blocks and jumps.

Example:

```
> (parameterize ([current-parameter-registers '()])
    (pretty-display
     ((compose
       expose-basic-blocks
       implement-fvars
       replace-locations
       assign-frame-variables
       assign-registers
       allocate-frames
       assign-call-undead-variables
       conflict-analysis
       undead-analysis
       uncover-locals
       select-instructions
       canonicalize-bind
       impose-calling-conventions
       sequentialize-let)
      '(module
         (define L.swap.1
           (lambda (x.1 y.2)
             (if (< y.2 x.1)
                 x.1
                 (let ([z.3 (call L.swap.1 y.2 x.1)])
                   z.3))))
         (call L.swap.1 1 2)))))
(module
  (define L.__main.8
    (begin
      (set! r15 r15)
      (set! (rbp - 8) 2)
      (set! (rbp - 0) 1)
      (set! r15 r15)
      (jump L.swap.1)))
  (define L.swap.1
    (begin
      (set! (rbp - 16) r15)
      (set! r14 (rbp - 0))
      (set! r15 (rbp - 8))
      (if (< r15 r14) (jump L.__nested.9) (jump L.__nested.10))))
  (define L.rp.7
    (begin
      (set! rbp (+ rbp 24))
      (set! r15 rax)
```

```
      (set! rax r15)
      (jump (rbp - 16))))
  (define L.__nested.9 (begin (set! rax r14) (jump (rbp - 16))))
  (define L.__nested.10
    (begin
      (set! rbp (- rbp 24))
      (set! (rbp - 8) r14)
      (set! (rbp - 0) r15)
      (set! r15 L.rp.7)
      (jump L.swap.1))))
```

4.8.9 Final Passes

The only two passes that should require changes are `patch-instructions` and `generate-x64`.

`patch-instructions` should be updated to work over *addr*s instead of *fvars*s. This can be done by changing a few predicates.

(patch-instructions *p***)** → `paren-x64-v6?` procedure

p : `para-asm-lang-v6?`

Compile the Para-asm-lang v6 to Paren-x64 v6 by patching instructions that have no x64 analogue into to a sequence of instructions and an auxiliary register from `current-patch-instructions-registers`.

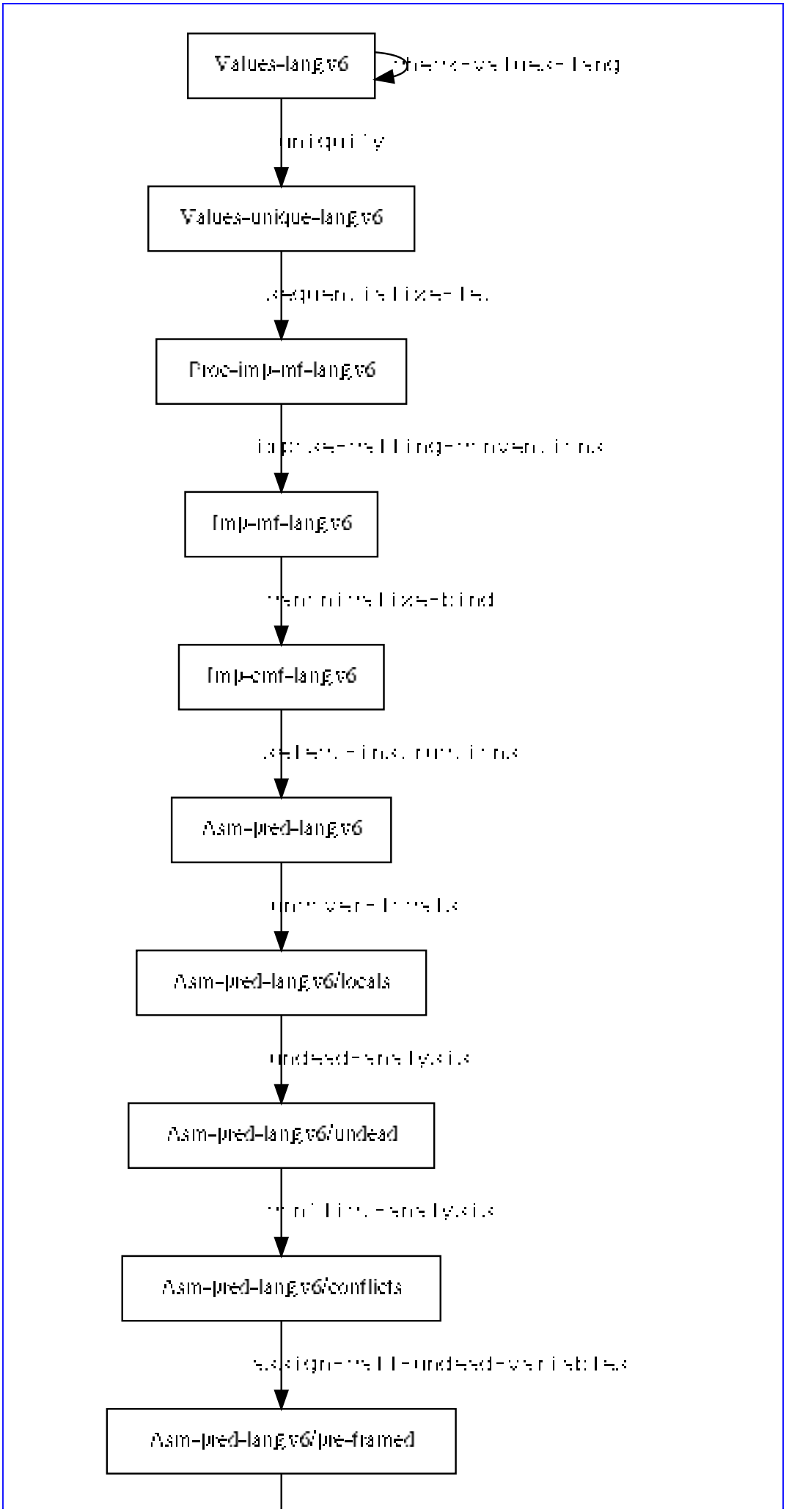
`generate-x64` needs to be updated to generate the new *binop*. Ideally, there is a separate helper for generating *binop*s, so this is only a minimal change.

(generate-x64 *p***)** → `(and/c string? x64-instructions?)` procedure

p : `paren-x64-v6?`

Compile the Paren-x64 v6 program into a valid sequence of x64 instructions, represented as a string.

4.8.10 Appendix: Overview


Values-lang.v6
Values-unique-lang.v6
Proc-imp-inf-lang.v6
Imp-inf-lang.v6
Imp-cmf-lang.v6
Asm-pred-lang.v6
Asm-pred-lang.v6/locals
Asm-pred-lang.v6/undead
Asm-pred-lang.v6/conflicts
Asm-pred-lang.v6/pre-framed

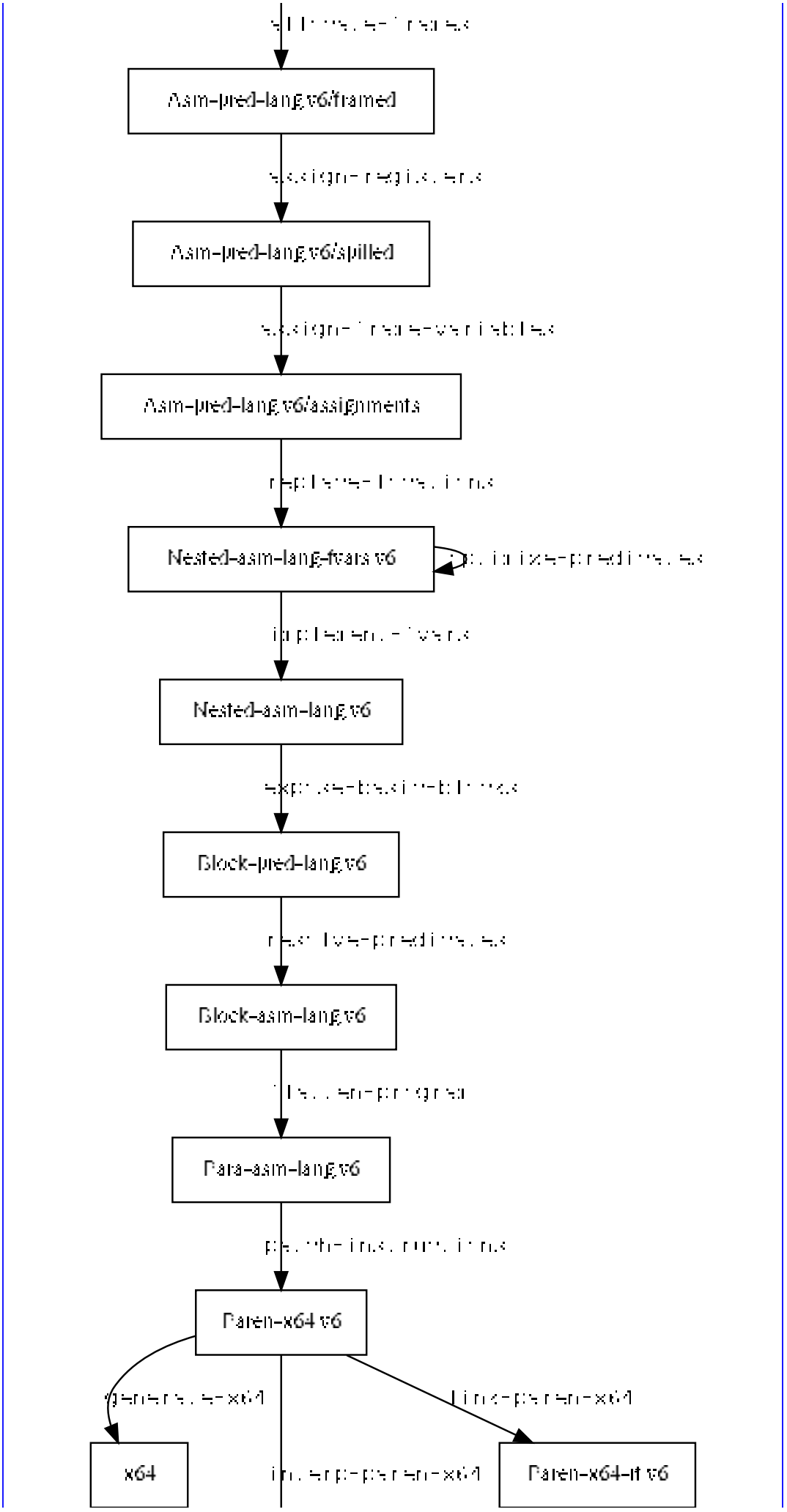

Asm-pred-lang.v6/framed
Asm-pred-lang.v6/spilled
Asm-pred-lang.v6/assignments
Nested-asm-lang-fvars.v6
Nested-asm-lang.v6
Block-pred-lang.v6
Block-asm-lang.v6
Para-asm-lang.v6
Paren-x64.v6
x64
Paren-x64-rt.v6

Figure 6: Overview of Compiler Version 6

4.8.11 Appendix: Languages

(**values-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

values-lang-v6 : grammar?

```
    p ::= (module (define x (lambda (x ...) tail)) ... tail)

 pred ::= (relop triv triv)
        | (true)
        | (false)
        | (not pred)
        | (let ([x value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([x value] ...) tail)
        | (if pred tail tail)
        | (call x triv ...)

value ::= triv
        | (binop triv triv)
        | (let ([x value] ...) value)
        | (if pred value value)
        | (call x triv ...)

 triv ::= int64
        | x

    x ::= name?

binop ::= *
        | +
        | -

relop ::= <
```

```
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?
```

(**values-unique-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-unique-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

values-unique-lang-v6 : grammar?

```
     p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

  pred ::= (relop opand opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (let ([aloc value] ...) pred)
        |  (if pred pred pred)

  tail ::= value
        |  (let ([aloc value] ...) tail)
        |  (if pred tail tail)
        |  (call triv opand ...)

 value ::= triv
        |  (binop opand opand)
        |  (let ([aloc value] ...) value)
        |  (if pred value value)
        |  (call triv opand ...)

 opand ::= int64
        |  aloc

  triv ::= opand
        |  label

 binop ::= *
        |  +
        |  -

 relop ::= <
        |  <=
```

```
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?
```

(**proc-imp-mf-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `proc-imp-mf-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

proc-imp-mf-lang-v6 : grammar?

```
      p ::= (module (define label (lambda (aloc ...) entry)) ...
              entry)

  entry ::= tail

   pred ::= (relop opand opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= value
        |  (call triv opand ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

  value ::= triv
        |  (binop opand opand)
        |  (begin effect ... value)
        |  (if pred value value)
        |  (call triv opand ...)

 effect ::= (set! aloc value)
        |  (begin effect ... effect)
        |  (if pred effect effect)

  opand ::= int64
        |  aloc
```

```
 triv ::= opand
        | label

binop ::= *
        | +
        | -

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

(**imp-mf-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v6 : grammar?

```
   p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
        | (binop opand opand)
        | (begin effect ... value)
```

```
        | (if pred value value)
        | (return-point label tail)

 effect ::= (set! loc value)
        | (begin effect ... effect)
        | (if pred effect effect)

  opand ::= int64
        | loc

   triv ::= opand
        | label

    loc ::= rloc
        | aloc

    trg ::= label
        | loc

  binop ::= *
        | +
        | -

  relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        | fvar?
```

(imp-cmf-lang-v6? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v6 : grammar?

```
      p ::= (module info (define label info tail) ... tail)
```

```
 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
        | (binop opand opand)

effect ::= (set! loc value)
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

```
  rloc ::= register?
         | fvar?
```

(asm-pred-lang-v6? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6 : grammar?

```
     p ::= (module info (define label info tail) ... tail)

  info ::= (#:from-contract (info/c (new-frames (frame ...))))

 frame ::= (aloc ...)

  pred ::= (relop loc opand)
         | (true)
         | (false)
         | (not pred)
         | (begin effect ... pred)
         | (if pred pred pred)

  tail ::= (jump trg loc ...)
         | (begin effect ... tail)
         | (if pred tail tail)

effect ::= (set! loc triv)
         | (set! loc_1 (binop loc_1 opand))
         | (begin effect ... effect)
         | (if pred effect effect)
         | (return-point label tail)

 opand ::= int64
         | loc

  triv ::= opand
         | label

   loc ::= rloc
         | aloc

   trg ::= label
         | loc

 binop ::= *
         | +
```

```
        |  -

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v6/locals?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v6/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/locals : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-
            frames (frame ...)) (locals (aloc ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
```

```
        |  (return-point label tail)

  opand ::= int64
        |  loc

   triv ::= opand
        |  label

    loc ::= rloc
        |  aloc

    trg ::= label
        |  loc

  binop ::= *
        |  +
        |  -

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v6/undead?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v6/undead grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/undead : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-
            frames (frame ...)) (locals (aloc ...)) (call-
            undead (loc ...)) (undead-out undead-set-tree/rloc?)))

  frame ::= (aloc ...)
```

```
 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v6/conflicts?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v6/conflicts grammar. The first non-terminal in the grammar defines valid programs.

asm-pred-lang-v6/conflicts : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-
             frames (frame ...)) (locals (aloc ...)) (call-
             undead (loc ...)) (undead-out undead-set-
             tree/rloc?) (conflicts ((loc (loc ...)) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  binop ::= *
          | +
          | -
```

```
relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v6/pre-framed?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v6/pre-framed grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/pre-framed : grammar?

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (new-frames (frame ...)) (locals (aloc ...)) (call-
          undead (loc ...)) (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

frame ::= (aloc ...)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)
```

```
opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v6/framed?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v6/framed grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/framed : grammar?

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
         tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

pred ::= (relop loc opand)
       | (true)
       | (false)
```

```
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   loc ::= rloc
        |  aloc

   trg ::= label
        |  loc

 binop ::= *
        |  +
        |  -

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v6/spilled?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v6/spilled` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/spilled : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
            tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  binop ::= *
          | +
          | -

  relop ::= <
          | <=
          | =
          | >=
          | >
          | !=
```

```
  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
          | fvar?
```

(**asm-pred-lang-v6/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v6/assignments grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v6/assignments : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-
             contract (info/c (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
```

```
        |  aloc

   trg ::= label
        |  loc

 binop ::= *
        |  +
        |  -

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
        |  fvar?
```

(**nested-asm-lang-fvars-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-fvars-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-fvars-v6 : grammar?

```
    p ::= (module (define label tail) ... tail)

 pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

 tail ::= (jump trg)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
```

```
       |  (begin effect ... effect)
       |  (if pred effect effect)
       |  (return-point label tail)

opand ::= int64
       |  loc

 triv ::= opand
       |  label

  trg ::= label
       |  loc

  loc ::= reg
       |  fvar

  reg ::= rsp
       |  rbp
       |  rax
       |  rbx
       |  rcx
       |  rdx
       |  rsi
       |  rdi
       |  r8
       |  r9
       |  r12
       |  r13
       |  r14
       |  r15

binop ::= *
       |  +
       |  -

relop ::= <
       |  <=
       |  =
       |  >=
       |  >
       |  !=

int64 ::= int64?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

(**nested-asm-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

nested-asm-lang-v6 : grammar?

```
          p ::= (module (define label tail) ... tail)

       pred ::= (relop loc opand)
              | (true)
              | (false)
              | (not pred)
              | (begin effect ... pred)
              | (if pred pred pred)

       tail ::= (jump trg)
              | (begin effect ... tail)
              | (if pred tail tail)

     effect ::= (set! loc triv)
              | (set! loc_1 (binop loc_1 opand))
              | (begin effect ... effect)
              | (if pred effect effect)
              | (return-point label tail)

       triv ::= opand
              | label

      opand ::= int64
              | loc

        trg ::= label
              | loc

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
```

```
          |  r9
          |  r12
          |  r13
          |  r14
          |  r15

    binop ::= *
          |  +
          |  -

    relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

    int64 ::= int64?

     aloc ::= aloc?

     addr ::= (fbp - dispoffset)

      fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

    label ::= label?
```

(**block-pred-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-pred-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

block-pred-lang-v6 : grammar?

```
       p ::= (module b ... b)

       b ::= (define label tail)

    pred ::= (relop loc opand)
          |  (true)
          |  (false)
          |  (not pred)

    tail ::= (jump trg)
          |  (begin s ... tail)
          |  (if pred (jump trg) (jump trg))
```

```
    s ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))

 triv ::= opand
        | label

opand ::= int64
        | loc

  trg ::= label
        | loc

  loc ::= reg
        | addr

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +
        | -

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?
```

```
dispoffset ::= dispoffset?

     label ::= label?
```

(block-asm-lang-v6? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-asm-lang-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

block-asm-lang-v6 : grammar?

```
        p ::= (module b ... b)

        b ::= (define label tail)

     tail ::= (jump trg)
            | (begin s ... tail)
            | (if (relop loc opand) (jump trg) (jump trg))

        s ::= (set! loc triv)
            | (set! loc_1 (binop loc_1 opand))

     triv ::= opand
            | label

    opand ::= int64
            | loc

      trg ::= label
            | loc

      loc ::= reg
            | addr

      reg ::= rsp
            | rbp
            | rax
            | rbx
            | rcx
            | rdx
            | rsi
            | rdi
            | r8
            | r9
            | r12
            | r13
            | r14
```

```
          | r15

    binop ::= *
          | +
          | -

    relop ::= <
          | <=
          | =
          | >=
          | >
          | !=

    int64 ::= int64?

     aloc ::= aloc?

     addr ::= (fbp - dispoffset)

      fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

    label ::= label?
```

(**para-asm-lang-v6?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the para-asm-lang-v6 grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v6 : grammar?

```
        p ::= (begin s ...)

        s ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (jump trg)
          | (with-label label s)
          | (compare loc opand)
          | (jump-if relop trg)

     triv ::= opand
          | label

    opand ::= int64
          | loc

      trg ::= label
```

```
           |  loc

     loc ::= reg
           |  addr

     reg ::= rsp
           |  rbp
           |  rax
           |  rbx
           |  rcx
           |  rdx
           |  rsi
           |  rdi
           |  r8
           |  r9
           |  r12
           |  r13
           |  r14
           |  r15

   binop ::= +
           |  *
           |  -

   relop ::= <
           |  <=
           |  =
           |  >=
           |  >
           |  !=

   int64 ::= int64?

    aloc ::= aloc?

    addr ::= (fbp - dispoffset)

     fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

   label ::= label?
```

(paren-x64-v6? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the paren-x64-v6 grammar. The first non-terminal in the grammar defines valid programs.

paren-x64-v6 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! addr int32)
              | (set! addr trg)
              | (set! reg loc)
              | (set! reg triv)
              | (set! reg_1 (binop reg_1 int32))
              | (set! reg_1 (binop reg_1 loc))
              | (with-label label s)
              | (jump trg)
              | (compare reg opand)
              | (jump-if relop label?)

        trg ::= reg
              | label

       triv ::= trg
              | int64

      opand ::= int64
              | reg

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
              | r9
              | r10
              | r11
              | r12
              | r13
              | r14
              | r15

       addr ::= (fbp - dispoffset)

        fbp ::= frame-base-pointer-register?

      binop ::= *
```

```
          |  +
          |  -

    relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

    int32 ::= int32?

    int64 ::= int64?

    label ::= label?

dispoffset ::= dispoffset?
```

(paren-x64-rt-v6? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-rt-v6` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-rt-v6 : grammar?

```
       p ::= (begin s ...)

       s ::= (set! loc triv)
          |  (set! reg fvar)
          |  (set! reg_1 (binop reg_1 int32))
          |  (set! reg_1 (binop reg_1 loc))
          |  (jump trg)
          |  (compare reg opand)
          |  (jump-if relop pc-addr)

     trg ::= reg
          |  pc-addr

    triv ::= trg
          |  int64

   opand ::= int64
          |  reg

     loc ::= reg
          |  addr

     reg ::= rsp
```

```
          |  rbp
          |  rax
          |  rbx
          |  rcx
          |  rdx
          |  rsi
          |  rdi
          |  r8
          |  r9
          |  r10
          |  r11
          |  r12
          |  r13
          |  r14
          |  r15

   addr ::= (fbp - dispoffset)

    fbp ::= frame-base-pointer-register?

  binop ::= *
          |  +
          |  -

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int32 ::= int32?

  int64 ::= int64?

  label ::= label?

dispoffset ::= dispoffset?
```

4.9 Data types: Immediates

4.9.1 Preface: What's wrong with Values-lang v6

Values-lang v6 gained the ability to express non-tail calls, an important step forward in writing real programs. However, our functions are still limited to work on machine integers. A realistic language would allow us to express programs over more interesting data types than mere machine integers.

Unfortunately, once we add data types, we have a problem distinguishing between any two data types. Everything is eventually represented as 64-bit integers in x64. We need some way to distinguish a collection of 64-bit as an integer, from a boolean, from the empty list, from a pointer to a procedure. Otherwise, we will have serious problems: undefined behaviours, unsafe casts, or memory safety errors.

This is not only a problem for ruling out unsafe behaviour in the source language. A static type system could take care to prevent the user from calling an integer as a procedure, for example. However, the language itself may need to distinguish different kinds of data at run-time. For example, a garbage collector may need to traverse data structures, but not immediate data like integers; a pretty printer may want to print different data differently.

To enable the language to distinguish different kinds of data, we can steal a few of our 64-bits to represent a data type tag. This limits the range of machine integers, but allows us to us to distinguish data dynamically, enabling safety and abstractions that must differentiate data dynamically.

There's a second limitation in Values-lang v6 that we lift this week. So far, we do not have algebraic expressions—we have limited ability to nest expressions, and must manually bind many intermediate computations to names. This is particularly tedious since we know how to write a compiler, which ought to do the tedious work of binding expressions to temporary names. After we have proper data types with dynamic checking, we can more easily nest expressions, particularly in the predicate position of an `if`.

> *Design digression:*
>
> Normally, we take care to design each new abstraction independently from each other, when we can. However, data types and algebraic expressions are difficult to separate without sacrificing some design goal. With only algebraic expressions and no data types, it becomes difficult to enforce safety without adding a more sophisticated type system with type annotations. With only data types and no algebraic expressions, implementing the compiler passes is more tedious because implementing pointer manipulation operations to implement data type tags benefits

> from algebraic expressions.

Our goal in this assignment is to implement the following language, Exprs-lang v7.

```
                p ::= (module b ... e)

                b ::= (define x (lambda (x ...) e))

                e ::= v
                    | (call e e ...)
                    | (let ([x e] ...) e)
                    | (if e e e)

                x ::= name?
                    | prim-f

                v ::= x
                    | fixnum
                    | #t
                    | #f
                    | empty
                    | (void)
                    | (error uint8)
                    | ascii-char-literal

          prim-f ::= binop
                    | unop

           binop ::= *
                    | +
                    | -
                    | eq?
                    | <
                    | <=
                    | >
                    | >=

            unop ::= fixnum?
                    | boolean?
                    | empty?
                    | void?
                    | ascii-char?
                    | error?
                    | not

          fixnum ::= int61?

           uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

We add a bunch of new values in Exprs-lang v7, including booleans, the empty list, the void object, (printable) ASCII character literals, and an error value. Exprs-lang v7 programs are allowed to return any of these values.

> We restrict ASCII characters to the printable ones, so we don't have to figure out how to print non-printable characters. The run-time system will work with some non-printable characters, but the results will not be converted to Racket properly.

Data type require new support from the run-time system. The new run-time system, `cpsc411/ptr-run-time`, supports printing the new values. `execute` takes a new optional second parameter, which can be used to change your view of the result. By default, you get back a Racket value. You can pass `nasm-run/print-string` to get back the string representation of the result, which will be handy to view the result of `(void)` correctly. You can pass `nasm-run/exit-code` to get the exit code, which is helpful for viewing the result of `(error` *`uint8`*`)`, which sets the exit code to *`uint8`*.

We also add new primitive operations, primarily predicates on our new data types. The interpretation of several operations will also change to add dynamic type checking. This will prevent those operations from being a source of undefined behaviour.

With proper booleans, we can finally allow an arbitrary value in the predicate position of an `if` expression in the surface language.

It is not clear how to support new data types going top-down, so we begin this assignment at the bottom.

4.9.2 Introduction to Tagged Object Representation

A common approach to efficiently represent word-sized data types is called *object tagging*.

> *Design digression:*
>
> Some of our clever tag choices, and some terminology, is borrowed from R. Kent Dybvig. You can learn more about it's use in Chez Scheme from this talk by his former PhD student Andy Keep: https://www.youtube.com/watch?v=BcC3KScZ-yA.

Each data type in our language will now be represented by a *ptr* (pronounced like *footer*). A ptr is a machine word whose `n` least-significant bits represent the *primary tag*, and whose upper `(- (* 8 (current-word-size-bytes)) n)` bits represent the data.

In our system, we have 64-bit words and will use the 3 least-significant bits for primary tags. With 3 bits, we can represent 8 primary data types in a ptr. We

want to reserve these for the most common and most performance critical data that can fit in a ptr, but we have additional constraints. This will limit the size of data we can represent, but gives us additional data types—probably a good trade off.

If we want more than 8 data types, which most languages support, we must reserve one tag for "everything else" (a pointer to memory). The structure in memory has infinite space in which to store additional tags. This seems to leave us only 7 possible data types.

Some data types require fewer than 64-bits, and we can exploit this fact to gain a few extra data types. For example, booleans really only require one bit. The empty list and the void object only require a tag, since they do not contain any data. ASCII characters only require 8 bits. We can make all of these share a single primary tag, and steal some of the unnecessary high-order bits for a *secondary tag*. This gives us 4 data types for the price of only 1 primary tag—what a bargain!

This leaves us 6 primary tags. One obvious choice is to use one for fixed sized integers; these will now be 61-bit integers. Integer operations are fast, and we don't want to slow them down by making them go through heap allocation and pointer indirection. If we wanted to support graphics and scientific operations, we would reserve one for floating-point numbers too.

We reserve the remaining primary tags for *tagged pointers*, pointers to common heap-allocated structures. Storing the tag on the pointer instead of in memory avoids the expense of an additional memory access to check a tag and the additional memory overhead of storing the tag in memory. We'll address the implementation of heap-allocated structured data in a future chapter.

Here is the default set of tags we will use in this assignment, given in base 2.

- `#b000`, *fixnums*, fixed-sized integers
- `#b001`, *unused*
- `#b010`, *unused*
- `#b011`, *unused*
- `#b100`, *unused*
- `#b101`, *unused*
- `#b110`, non-fixnum immediates (booleans, etc)
- `#b111`, *unused*

> Racket supports binary literals, and automatically interprets them as two's complement integers. If you type `#b111`, you get back `7`, and the two values are `equal?`. This will be helpful when writing this assignment and writing

> tests.

For the non-fixnum immediates, we use the following secondary tags. Note that the 3 least-significant bits of secondary tags are the shared primary tag.

- `#b00000110`, for `#f`
- `#b00001110`, for `#t`
- `#b00010110`, for `empty`
- `#b00011110`, for `(void)`
- `#b00101110`, for an ASCII character
- `#b00111110`, for the error value

> For annoying technical reasons, our representation of the empty list differs slightly from Racket. We use `empty`, while Racket uses `'()`.

The integer 7 is `#b111` in base 2, and would be represented, in base 2, as the ptr `#b111000`. The three low-order bits are the tag `#b000`, and the high-order bits are the data `#b111` (implicitly padded with 0 to 64-bits). To perform arithmetic on this representation, we can use simply right-shift the ptr by the size of a primary tag to get a two's complement integer:

Example:

```
> (arithmetic-shift 56 -3)
7
```

> All these notes are typeset using base-2 literals, but Racket cannot distinguish them from integers, so they may get accidentally rendered in base 10.

A handy fact about the choice of tag for fixnums is that any number `n` is represented as `(* 8 n)`. This fact allows for some optimizations on fixnum operations, and will make porting examples and tests easier.

Similarly, the ASCII character 7 would be as a ptr, in base 2, `#b011011100101110`.

> See https://en.wikipedia.org/wiki/ASCII#Printable_characters for the representation of ASCII characters.

For character operations, we must right-shift by the size of a secondary tag (8). The character `#b0110111` is the ASCII representation of 7. Combined with its

tag, this is `#b011011100101110`, which reads in Racket as `14126`.

Example:

```
> (integer->char (arithmetic-shift 14126 -8))
#\7
```

We can implement tag checks using bitwise masking. For example, to check whether a ptr is a fixnum, we mask the ptr with tag `#b111` (`7`) and compare the result to the primary tag for fixnums, `#b000` (`0`). For example, we can ask whether the above two examples are fixnums (and whether the latter is a character) as follows:

Examples:

```
> (eq? (bitwise-and 56 7) 0)
#t
> (eq? (bitwise-and 14126 7) 0)
#f
> (eq? (bitwise-and 14126 255) 46)
#t
```

Our representation of the number 7 is a fixnum, while the representation of the character 7 is not.

The representation of booleans is a bit tricky. We don't quite introduce a new tag for booleans, and then add a single bit in the payload. Instead, we essentially represent `#t` and `#f` as separate data types, whose tags are `#b110` (`6`) and `#b1110` (`14`). It makes masking a boolean slightly more complex. We use the mask `#b11110111` (`247`), to succeed regardless of whether the fourth bit is set.

Examples:

```
> (eq? (bitwise-and 14 247) 6)
#t
> (eq? (bitwise-and 6 247) 6)
#t
```

The benefit is that any immediate that is not `#f` can be interpreted as `#t` using *`bitwise-xor`*, instead of converting to a boolean and using an arithmetic shift to compute the value of the boolean. Only `#f` is 0 when "xor"'d with the non-fixnum immediate tag, `#b110`.

Examples:

```
> (eq? (arithmetic-shift 6 -3) 0)
#t
> (eq? (arithmetic-shift 14 -3) 0)
#f
> (eq? (bitwise-xor 6 6) 0)
#t
```

```
> (eq? (bitwise-xor 14 6) 0)
#f
> (eq? (bitwise-xor 56 6) 0)
#f
> (eq? (bitwise-and 14 7) 0)
#f
> (eq? (bitwise-xor 14126 6) 0)
#f
```

The representation of `error` is inefficient. We only require 8 bits of data for the error code, so 24 bits are wasted. But we're not adding more data types with secondary tags, so it is not worth over-engineering. Besides, a better error data type would use at least a string payload, which requires allocating space on the heap anyway.

The particular choice of tags is not important for correctness, although clever choices like above can help us implement common operations more efficiently. We therefore should introduce abstractions to keep the compiler abstract with respect to particular tag choices, so we can change our minds later if a better representation occurs to us.

To implement ptrs, we need bitwise operations so that we can mask off the tags bits and operate on the data. This means we need to expose bitwise operations from x64.

4.9.3 Exposing Bitwise Operations in Paren-x64

We expose the following x64 instructions this week.

- `sar` *`loc`*`,` *`int`*

  Perform arithmetic right-shift by *`int`* on *`loc`*.

  This instruction requires that its second operand be an integer literal between 0 and 63, inclusive, *i.e.,* `0 <= int <= 63`. We will assume this constraint in the intermediate languages, and never expose this operation in Exprs-lang v7.

- `and` *`loc`*`,` *`op`*

  Compute the bitwise "and" of *`loc`* and *`op`*, storing the result in *`loc`*. Like with other binary operations, when *`op`* is an integer, it must be an *`int32`*, and when *`op`* is an *`addr`*, *`loc`* cannot also be an *`addr`*.

- `or` *`loc`*`,` *`op`*

  Compute the bitwise "inclusive or" of *`loc`* and *`op`*, storing the result in *`loc`*. Like with other binary operations, when *`op`* is an integer, it must be an *`int32`*, and when *`op`* is an *`addr`*, *`loc`* cannot also be an *`addr`*.

- `xor` *`loc`*, *`op`*

  Compute the bitwise "exclusive or" of *`loc`* and *`op`*, storing the result in *`loc`*. Like with other binary operations, when *`op`* is an integer, it must be an *`int32`*, and when *`op`* is an *`addr`*, *`loc`* cannot also be an *`addr`*.

First, we add each of these operations as a *`binop`* to *Paren-x64 v7* below. The differences are typeset with respect to Paren-x64 v6.

```
      p ::= (begin s ...)

      s ::= (set! addr int32)
          | (set! addr trg)
          | (set! reg loc)
          | (set! reg triv)
          | (set! reg_1 (binop reg_1 int32))
          | (set! reg_1 (binop reg_1 loc))
          | (with-label label?+ label- s)
          | (jump trg)
          | (compare reg opand)
          | (jump-if relop label+ label?-)

    trg ::= reg
          | label

   triv ::= trg
          | int64

  opand ::= int64
          | reg

    loc ::= reg
          | addr

    reg ::= rsp
          | rbp
          | rax
          | rbx
          | rcx
          | rdx
          | rsi
          | rdi
          | r8
          | r9
          | r10
          | r11
          | r12
          | r13
          | r14
          | r15

   addr ::= (fbp - dispoffset)
```

```
  fbp ::= frame-base-pointer-register?

binop ::= *
        | +
        | -
        | bitwise-and+
        | bitwise-ior+
        | bitwise-xor+
        | arithmetic-shift-right+

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int32 ::= int32?

int64 ::= int64?

label ::= label?

dispoffset ::= dispoffset?
```

(**generate-x64** *p*) → (and/c string? x64-instructions?) procedure

p : paren-x64-v7?

Compile the Paren-x64 v7 program into a valid sequence of x64 instructions, represented as a string.

4.9.4 Exposing Bitwise Operations

Specifying the tagged representation happens very close to our source language. We need to expose these operations all the way up to our prior source language, Values-unique-lang-v6. Below, we design the new *Values-bits-lang v7*, typeset with differences compared to Values-unique-lang-v6.

```
  p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

pred ::= (relop opand opand)
       | (true)
       | (false)
       | (not pred)
       | (let ([aloc value] ...) pred)
       | (if pred pred pred)

tail ::= value
```

```
        |  (let ([aloc value] ...) tail)
        |  (if pred tail tail)
        |  (call triv opand ...)

value ::= triv
        |  (binop opand opand)
        |  (let ([aloc value] ...) value)
        |  (if pred value value)
        |  (call triv opand ...)

opand ::= int64
        |  aloc

 triv ::= opand
        |  label

binop ::= *
        |  +
        |  -
        |  bitwise-and+
        |  bitwise-ior+
        |  bitwise-xor+
        |  arithmetic-shift-right+

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

The only data type in Values-bits-lang v7 is BITS, 64 for them, interpreted as an integer sometimes.

We assume that each *binop* is well-typed; they shouldn't be used with labels as arguments, the and calls to `arithmetic-shift-right` follow the restrictions required by x64.

The new operations do not have large effects on the language designs or compiler passes between `sequentialize-let` and `generate-x64`, so these details are left unspecified in this chapter. You will need to redesign the intermediate languages with the new operations.

4.9.5 Algebraic Expressions

As we saw in Introduction to Tagged Object Representation, many of the operations we want to perform on ptrs are easily expressed as algebraic expressions. The expression *(fixnum? 7)* is expressed as `(eq? (bitwise-and 7 #b111) #b000)`. Representing this in Values-bits-lang v7, which does not have algebraic expressions, is unnecessarily complicated. We would have to introduce auxiliary variables, manually pass them around the compiler, and generate code with additional let-bindings. This is particularly silly; we know how to write compilers, so the compiler should do this for us, and allow us to write the code we want to write.

We therefore design *Exprs-bits-lang v7*, a language that has only bits and bitwise operations, but that allows algebraic expressions in most positions. The predicate position of `if` expressions is still restricted, since we cannot introduce algebraic `if` expressions without booleans.

```
      p ::= (module b ... e)

      b ::= (define label (lambda (aloc ...) e))

   pred ::= (relop e e)
          | (true)
          | (false)
          | (not pred)
          | (let ([aloc e] ...) pred)
          | (if pred pred pred)

      e ::= v
          | (binop e e)
          | (call e e ...)
          | (let ([aloc e] ...) e)
          | (if pred e e)

      v ::= label
          | aloc
          | int64

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
          | arithmetic-shift-right

   aloc ::= aloc?

  label ::= label?

  relop ::= <
          | <=
          | =
          | >=
```

```
        |  >
        |  !=

int64 ::= int64?
```

Looking at this grammar, it may not be obvious how to transform this into Values-bits-lang v7. We can see the structure of the pass more clearly if we expand the grammar slightly. Below, we define the expanded grammar *Exprs-bit-lang v7/context*. We typeset the differences compared to Values-bits-lang v7

```
      p ::= (module b+ (define label (lambda (aloc ...)
            tail))- ... tail)

     b+ ::= (define label (lambda (aloc ...) tail))

   pred ::= (relop value+ value+ opand- opand-)
          |  (true)
          |  (false)
          |  (not pred)
          |  (let ([aloc value] ...) pred)
          |  (if pred pred pred)

   tail ::= value
          |  (let ([aloc value] ...) tail)
          |  (if pred tail tail)
          |  (call triv opand ...)-

  value ::= triv
          |  (binop value+ value+ opand- opand-)
          |  (call value value ...)+
          |  (let ([aloc value] ...) value)
          |  (if pred value value)
          |  (call triv opand ...)-

opand- ::= int64
          |  aloc

   triv ::= opand-
          |  label
          |  aloc+
          |  int64+

  binop ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right
```

```
 aloc+ ::= aloc?

label+ ::= label?

 relop ::= <
         |  <=
         |  =
         |  >=
         |  >
         |  !=

 int64 ::= int64?

 aloc- ::= aloc?

label- ::= label?
```

Now we can see the the main difference, semantically, is additional nesting in the value context. Calls and binary operations can now take nested *value* expressions rather than *triv*s. While this means we can collapse the syntax, separating the syntax semantically helps us see the true difference and see that the essence of the transformation is let-binding intermediate results to make all operands trivial.

To transform this into Values-bits-lang v7, we need to perform the moandic form translation. In essence, we recursively translate any nested expression that is not a *value* into one by let-binding all intermediate computations. For example, we would transform a call `` `(call ,e_1 ,e_2) `` into something like the following:

```
`(let ([,x_1 ,(normalize e_1)])
   (let ([,x_2 ,(normalize e_2)])
     (call ,x_1 ,x_2)))
```

If you've written any examples or tests in Values-lang v6, you've probably done this tranformation by hand many times.

> *Design digression:*
>
> We can do a better job if we following the template for the output language while processing the input language. In this way, we design functions for each non-terminal in the output language. These functions describe in what context the current term is being processed. For example, when processing `(relop value_1 value_2)`, we want to process *value_1* in *aloc* context, since that's the restriction in the target language, but can process *value_2* in *triv* context to avoid creating an unnecessary auxiliary binding.

For historical reasons, we call this pass `remove-complex-opera*`.

`(remove-complex-opera* p)` → `values-bits-lang-v7?` procedure

```
p : exprs-bits-lang-v7?
```

Performs the monadic form transformation, unnesting all non-trivial operators and operands to *binop*s, `calls`, and *relop*s, making data flow explicit and and simple to implement imperatively.

4.9.6 Specifying Data Type Representation

4.9.6.1 specify-representation

Next we design *Exprs-unsafe-data-lang v7*. We replace bits with proper data types, and lift the restriction on `if` expressions, which are now properly algebraic.

```
                p ::= (module b ... e)

                b ::= (define label (lambda (aloc ...) e))

             pred ::= e+
                    | (relop e e)−
                    | (true)
                    | (false)
                    | (not pred)
                    | (let ([aloc e] ...) pred)
                    | (if pred pred pred)

                e ::= v
                    | (primop+ binop− e ...+ e−)
                    | (call e e ...)
                    | (let ([aloc e] ...) e)
                    | (if pred e e)

               v+ ::= label
                    | aloc
                    | fixnum
                    | #t
                    | #f
                    | empty
                    | (void)
                    | (error uint8)
                    | ascii-char-literal

          primop+ ::= binop
                    | unop

           binop+ ::= unsafe-fx*
                    | unsafe-fx+
```

```
                  |  unsafe-fx-
                  |  eq?
                  |  unsafe-fx<
                  |  unsafe-fx<=
                  |  unsafe-fx>
                  |  unsafe-fx>=

        unop+ ::= fixnum?
                  |  boolean?
                  |  empty?
                  |  void?
                  |  ascii-char?
                  |  error?
                  |  not

           v- ::= label
                  |  aloc
                  |  int64

        binop- ::= *
                  |  +
                  |  -
                  |  bitwise-and
                  |  bitwise-ior
                  |  bitwise-xor
                  |  arithmetic-shift-right

         aloc ::= aloc?

        label ::= label?

       fixnum+ ::= int61?

        uint8+ ::= uint8?

ascii-char-literal+ ::= ascii-char-literal?

        relop- ::= <
                  |  <=
                  |  =
                  |  >=
                  |  >
                  |  !=

        int64- ::= int64?
```

We assume all operations are well-typed in this language, and implement dynamic checks later. To make this clear, we prefix all the operators that require a dynamic check with *unsafe-*. `call` is still unsafe, since we do not know how to tag functions yet.

First, we translate each value literal to ptrs.

For booleans, empty, and void, this is trivial. We simply emit their ptr representation; you can find some parameters for this defined in `cpsc411/compiler-lib`.

For data types with a payload (fixnum and ASCII characters) we need to do some work to merge the payload data with the tag. The general strategy is to first left shift the data by the number of bits in the tag, then perform an inclusive or with the tag.

```
(bitwise-ior (arithmetic-shift 7 3) #b000)
(bitwise-ior (arithmetic-shift (char->integer #\x) 8) #b00101110)
```

Note that because the fixnum tag is all 0s, we can omit the bitwise or.

```
(arithmetic-shift 7 3)
```

Remember not to use magic numbers in your compiler, and instead use appropriate parameters so we can change tags and masks easily later.

The complicated cases are for operations on numbers, but even these are mostly unchanged due to some handy algebraic facts. Recall that every fixnum `n` is represented by a ptr whose value is `(* 8 n)`. For `+` and `-`, this means we don't need to do anything at all, since `8x + 8y = 8(x + y)`, and similarly `8x - 8y = 8(x - y)`. Similarly, `<`, `<=`, `>`, `>=`, and *`eq?`* all work unchanged on ptrs. However, these are boolean operations in Exprs-data-lang v7, so their implementation must return a boolean ptr.

Only `*` poses a problem, since `8x * 8y = 64(x * y)`. However, we do not need to adjust both arguments: we observe that `8x * y = 8(x * y)`, and similarly `x * 8y = 8(x * y)`. We only need to shift one operand before performing `*` to get the correct result as a ptr. If either argument is constant, we can perform the shift at compile time, completely eliminating the additional overhead. Otherwise, we translate `(* e_1 e_2)` to (roughly) `(* e_1 (arithmetic-shift-right e_2 3))`.

Next, we translate `if`, which should be translated from an operation on booleans to an operation on *`pred`*s. We'll do some work later transforming booleans into predicates, for optimization, but for now we just consider how to implement booleans correctly. Racket and Scheme are falsey languages—any thing that is not `#f` is considered true. We can implement this naively: simply compare to the ptr for `#f`. Recall from earlier that our representation allows us to treat anything that is not false as true by a simple `bitwise-xor` and comparison to 0, but we might want to leave that for a more general optimization.

When translating the booleans *`unops`* and *`binops`* *`binops`* on ptrs, we need to produce something that the translation of `if` can consume. `if` is expecting a boolean value, so each *`unop`* should be translated to an expression that returns a

boolean. As we saw earlier, type predicates are implemented by masking the ptr using `bitwise-and`, and comparing the result to the tag using `=`. But the target language `=` is a relop, not a boolean operation, so we translate `(fixnum? e)` to `(if (= (bitwise-and e 7) 0) #t #f)`. Our representation of booleans supports optimizing this, as described earlier, but we should leave that optimization for a separate pass.

`(specify-representation p)` → `exprs-bits-lang-v7?` procedure
`p` : `exprs-unsafe-data-lang-v7?`

Compiles immediate data and primitive operations into their implementations as ptrs and primitive bitwise operations on ptrs.

Next we design *Exprs-unique-lang v7*, which exposes a uniform safe interface to our language with immediate data. It exposes dynamically checked versions of each unsafe operation, hides the predicate sub-language from the user, and exposes all primitive operations as functions which should be used with `call` in the source language.

To implement this language, we essentially "link" the definitions of each procedure wrapper for each primitive operation and replace the reserved `prim-f` names for the functions with the appropriate fresh labels. Since our compiler as not provided any means of linking separately compiled modules, we implement this by adding new definitions to the module. Each safe function should raise a different error code depending on which operation was attempted, and which argument was not well-typed. Be sure to document your error codes.

```
    p ::= (module b ... e)

    b ::= (define label (lambda (aloc ...) e))

pred⁻ ::= e
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc e] ...) pred)
        | (if pred pred pred)

    e ::= v
        | (primop e ...)⁻
        | (call e e ...)
        | (let ([aloc e] ...) e)
        | (if e⁺ pred⁻ e e)

    v ::= label
        | aloc
        | prim-f⁺
        | fixnum
```

```
                | #t
                | #f
                | empty
                | (void)
                | (error uint8)
                | ascii-char-literal

      prim-f+ ::= primop-
                | binop
                | unop

       binop+ ::= *
                | +
                | -
                | <
                | eq?
                | <=
                | >
                | >=

       binop- ::= unsafe-fx*
                | unsafe-fx+
                | unsafe-fx-
                | eq?
                | unsafe-fx<
                | unsafe-fx<=
                | unsafe-fx>
                | unsafe-fx>=

         unop ::= fixnum?
                | boolean?
                | empty?
                | void?
                | ascii-char?
                | error?
                | not

         aloc ::= aloc?

        label ::= label?

       fixnum ::= int61?

        uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

In Exprs-unique-lang v7, most ill-typed expressions are valid programs. For example, `(+ #t (eq? 5 (void)))` is a valid program. The only invalid programs are those that attempt to `apply` a non-function, or use a label in any position except the first operand of `apply`; a limitation we will solve in the

coming chapters.

```
(implement-safe-primops p) → exprs-unsafe-data-lang-v7? procedure
  p : exprs-unique-lang-v7?
```

Implement safe primitive operations by inserting procedure definitions for each primitive operation which perform dynamic tag checking, to ensure type safety.

4.9.6.2 uniquify

Last but not least, we update uniquify. The source language, *Exprs-lang v7*, is defined below.

```
p ::= (module b ... e)

b ::= (define x+ label- (lambda (x+ aloc- ...) e))

e ::= v
    | (call e e ...)
    | (let ([x+ aloc- e] ...) e)
    | (if e e e)

x+ ::= name?
     | prim-f

v ::= x+
    | label-
    | aloc-
    | prim-f-
    | fixnum
    | #t
    | #f
    | empty
    | (void)
    | (error uint8)
    | ascii-char-literal

prim-f ::= binop
         | unop

binop ::= *
        | +
        | -
        | eq?+
        | <
        | eq?-
        | <=
        | >
```

```
                |  >=

          unop ::= fixnum?
                |  boolean?
                |  empty?
                |  void?
                |  ascii-char?
                |  error?
                |  not

         aloc¯ ::= aloc?

        label¯ ::= label?

       fixnum ::= int61?

        uint8 ::= uint8?

  ascii-char- ::= ascii-char-literal?
      literal
```

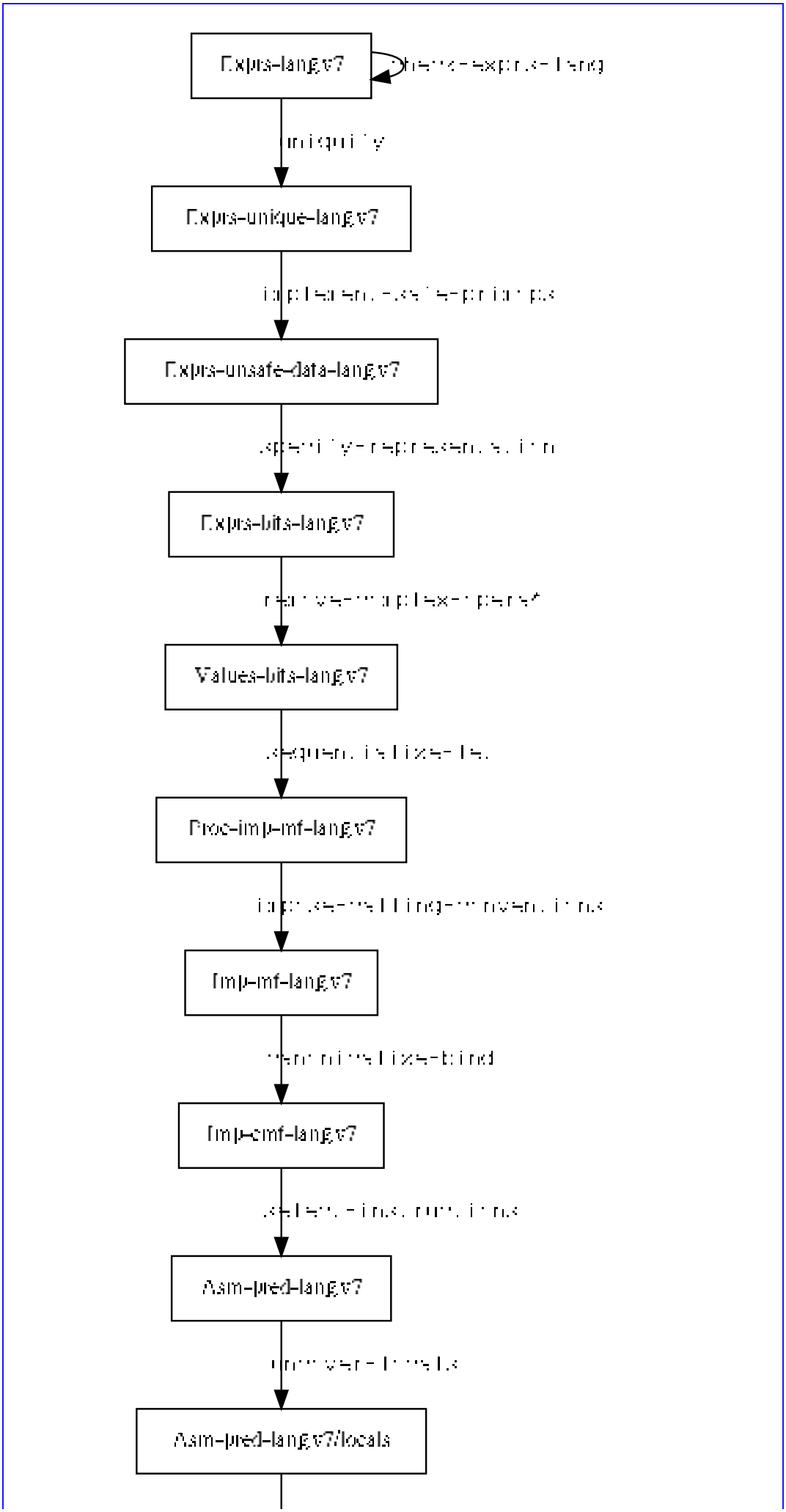
You are allowed to shadow `"prim-f"`s.

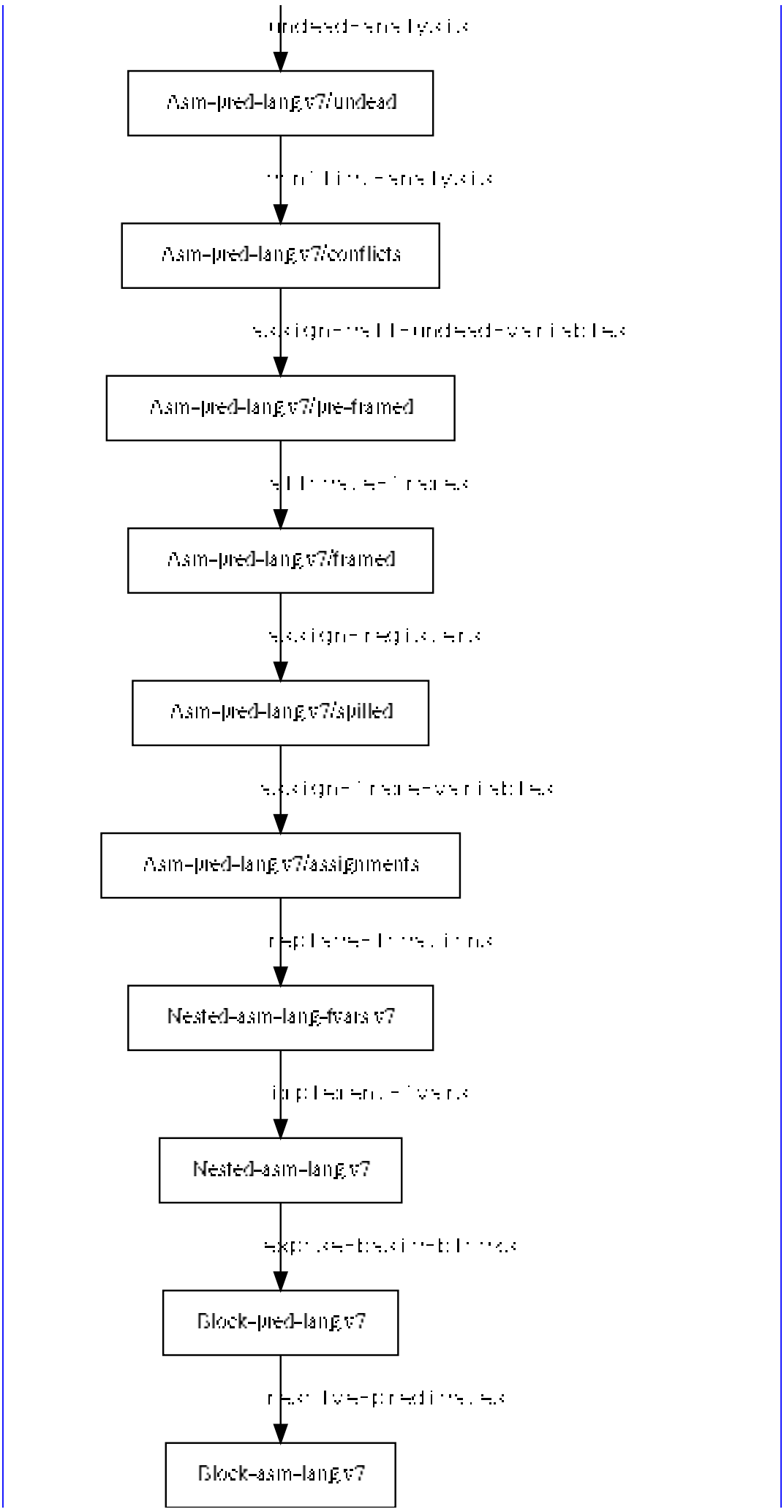
(uniquify *p*) → exprs-unique-lang-v7 procedure

p : exprs-lang-v7

Resolves top-level lexical identifiers into unique labels, and all other lexical identifiers into unique abstract locations.

4.9.7 Appendix: Overview


Exprs-lang.v7
Unique-exprs-lang
uniquify
Exprs-unique-lang.v7
implicate-unsafe-primops
Exprs-unsafe-data-lang.v7
explicate-prepare-unsafe
Exprs-bits-lang.v7
reduce-to-simple-exprs
Values-bits-lang.v7
sequentialize
Proc-imp-mf-lang.v7
implement-allocation-runtime
Imp-mf-lang.v7
uncover-live
Imp-cmf-lang.v7
select-instructions
Asm-pred-lang.v7
uncover-locals
Asm-pred-lang.v7/locals


undead-analysis
Asm-pred-lang.v7/undead
conflicts-analysis
Asm-pred-lang.v7/conflicts
assign-call-undead-variables
Asm-pred-lang.v7/pre-framed
call-save-frames
Asm-pred-lang.v7/framed
assign-register
Asm-pred-lang.v7/spilled
assign-frame-variables
Asm-pred-lang.v7/assignments
replace-locations
Nested-asm-lang-fvars.v7
implement-fvars
Nested-asm-lang.v7
expose-basic-blocks
Block-pred-lang.v7
resolve-predicates
Block-asm-lang.v7

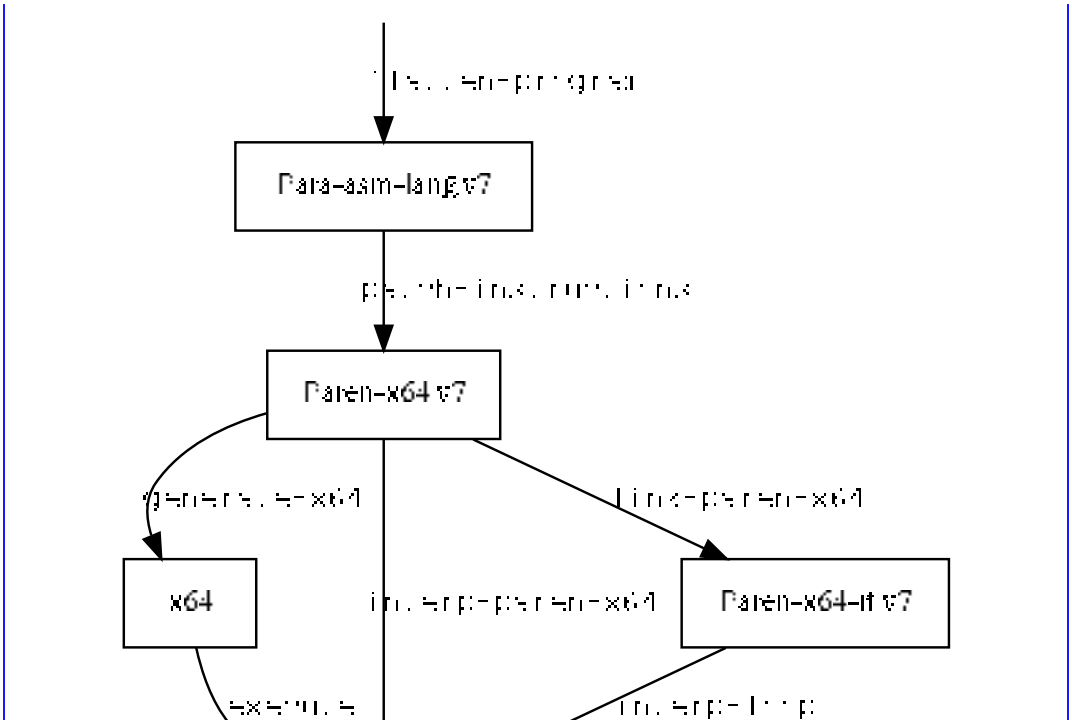


Figure 7: Overview of Compiler Version 7

4.9.8 Appendix: Languages

(**exprs-lang-v7?** *a*) → boolean? procedure
a : any/c

Decides whether *a* is a valid program in the `exprs-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

exprs-lang-v7 : grammar?

```
p ::= (module b ... e)

b ::= (define x (lambda (x ...) e))

e ::= v
    | (call e e ...)
    | (let ([x e] ...) e)
    | (if e e e)

x ::= name?
    | prim-f

v ::= x
    | fixnum
    | #t
```

```
                  |  #f
                  |  empty
                  |  (void)
                  |  (error uint8)
                  |  ascii-char-literal

            prim-f ::= binop
                  |  unop

             binop ::= *
                  |  +
                  |  -
                  |  eq?
                  |  <
                  |  <=
                  |  >
                  |  >=

              unop ::= fixnum?
                  |  boolean?
                  |  empty?
                  |  void?
                  |  ascii-char?
                  |  error?
                  |  not

            fixnum ::= int61?

             uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(exprs-unique-lang-v7? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unique-lang-v7 grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-unique-lang-v7 : grammar?

```
p ::= (module b ... e)

b ::= (define label (lambda (aloc ...) e))

e ::= v
   |  (call e e ...)
   |  (let ([aloc e] ...) e)
   |  (if e e e)
```

```
                v ::= label
                    | aloc
                    | prim-f
                    | fixnum
                    | #t
                    | #f
                    | empty
                    | (void)
                    | (error uint8)
                    | ascii-char-literal

           prim-f ::= binop
                    | unop

            binop ::= *
                    | +
                    | -
                    | <
                    | eq?
                    | <=
                    | >
                    | >=

             unop ::= fixnum?
                    | boolean?
                    | empty?
                    | void?
                    | ascii-char?
                    | error?
                    | not

             aloc ::= aloc?

            label ::= label?

           fixnum ::= int61?

            uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(**exprs-unsafe-data-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unsafe-data-lang-v7 grammar. The first non-terminal in the grammar defines valid programs.

exprs-unsafe-data-lang-v7 : grammar?

```
   p ::= (module b ... e)

   b ::= (define label (lambda (aloc ...) e))

pred ::= e
       | (true)
       | (false)
       | (not pred)
       | (let ([aloc e] ...) pred)
       | (if pred pred pred)

   e ::= v
       | (primop e ...)
       | (call e e ...)
       | (let ([aloc e] ...) e)
       | (if pred e e)

   v ::= label
       | aloc
       | fixnum
       | #t
       | #f
       | empty
       | (void)
       | (error uint8)
       | ascii-char-literal

primop ::= binop
       | unop

 binop ::= unsafe-fx*
       | unsafe-fx+
       | unsafe-fx-
       | eq?
       | unsafe-fx<
       | unsafe-fx<=
       | unsafe-fx>
       | unsafe-fx>=

 unop ::= fixnum?
       | boolean?
       | empty?
       | void?
       | ascii-char?
       | error?
       | not
```

```
    aloc ::= aloc?

   label ::= label?

  fixnum ::= int61?

   uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(**exprs-bits-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `exprs-bits-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-bits-lang-v7 : grammar?

```
    p ::= (module b ... e)

    b ::= (define label (lambda (aloc ...) e))

 pred ::= (relop e e)
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc e] ...) pred)
        | (if pred pred pred)

    e ::= v
        | (binop e e)
        | (call e e ...)
        | (let ([aloc e] ...) e)
        | (if pred e e)

    v ::= label
        | aloc
        | int64

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

 aloc ::= aloc?
```

```
label ::= label?

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?
```

(**exprs-bits-lang-v7/contexts?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-bits-lang-v7/contexts grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-bits-lang-v7/contexts : grammar?

```
    p ::= (module b ... tail)

    b ::= (define label (lambda (aloc ...) tail))

 pred ::= (relop value value)
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc value] ...) pred)
        | (if pred pred pred)

 tail ::= value
        | (let ([aloc value] ...) tail)
        | (if pred tail tail)

value ::= triv
        | (binop value value)
        | (call value value ...)
        | (let ([aloc value] ...) value)
        | (if pred value value)

 triv ::= label
        | aloc
        | int64

binop ::= *
        | +
        | -
        | bitwise-and
```

```
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

 aloc ::= aloc?

label ::= label?

relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

int64 ::= int64?
```

(values-bits-lang-v7? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-bits-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

values-bits-lang-v7 : grammar?

```
    p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

 pred ::= (relop opand opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (let ([aloc value] ...) pred)
        |  (if pred pred pred)

 tail ::= value
        |  (let ([aloc value] ...) tail)
        |  (if pred tail tail)
        |  (call triv opand ...)

value ::= triv
        |  (binop opand opand)
        |  (let ([aloc value] ...) value)
        |  (if pred value value)
        |  (call triv opand ...)

opand ::= int64
        |  aloc
```

```
 triv ::= opand
        | label

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

(**proc-imp-mf-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `proc-imp-mf-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

proc-imp-mf-lang-v7 : grammar?

```
    p ::= (module (define label (lambda (aloc ...) entry)) ...
            entry)

 entry ::= tail

  pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

  tail ::= value
        | (call triv opand ...)
        | (begin effect ... tail)
        | (if pred tail tail)
```

```
  value ::= triv
          | (binop opand opand)
          | (begin effect ... value)
          | (if pred value value)
          | (call triv opand ...)

 effect ::= (set! aloc value)
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | aloc

   triv ::= opand
          | label

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
          | arithmetic-shift-right

  relop ::= <
          | <=
          | =
          | >=
          | >
          | !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?
```

(**imp-mf-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v7 : grammar?

```
    p ::= (module info (define label info tail) ... tail)
```

```
 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
        | (binop opand opand)
        | (begin effect ... value)
        | (if pred value value)
        | (return-point label tail)

effect ::= (set! loc value)
        | (begin effect ... effect)
        | (if pred effect effect)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
```

```
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**imp-cmf-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v7 : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...))))

  frame ::= (aloc ...)

   pred ::= (relop opand opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

  value ::= triv
        |  (binop opand opand)

 effect ::= (set! loc value)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  opand ::= int64
        |  loc

   triv ::= opand
        |  label
```

```
  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(asm-pred-lang-v7? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7 : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
```

```
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   loc ::= rloc
        |  aloc

   trg ::= label
        |  loc

 binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v7/locals?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v7/locals grammar. The first non-terminal in the grammar defines valid programs.

asm-pred-lang-v7/locals : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-
             frames (frame ...)) (locals (aloc ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
```

```
        |  bitwise-xor
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(asm-pred-lang-v7/undead? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v7/undead grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/undead : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-
            frames (frame ...)) (locals (aloc ...)) (call-
            undead (loc ...)) (undead-out undead-set-tree/rloc?)))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
```

```
        |  (if pred effect effect)
        |  (return-point label tail)

  opand ::= int64
        |  loc

   triv ::= opand
        |  label

    loc ::= rloc
        |  aloc

    trg ::= label
        |  loc

  binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v7/conflicts?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v7/conflicts grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/conflicts : grammar?

```
    p ::= (module info (define label info tail) ... tail)
```

```
 info ::= (#:from-contract (info/c (new-
          frames (frame ...)) (locals (aloc ...)) (call-
          undead (loc ...)) (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...))))

frame ::= (aloc ...)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)

opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=
```

```
  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
          | fvar?
```

(**asm-pred-lang-v7/pre-framed?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v7/pre-framed grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/pre-framed : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...)) (locals (aloc ...)) (call-
            undead (loc ...)) (undead-out undead-set-
            tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label
```

```
  loc ::= rloc
        | aloc

  trg ::= label
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v7/framed?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v7/framed grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/framed : grammar?

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)
       | (begin effect ... pred)
```

```
        |  (if pred pred pred)

  tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   loc ::= rloc
        |  aloc

   trg ::= label
        |  loc

 binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v7/spilled?** *a*) → boolean? procedure

```
a : any/c
```

Decides whether *a* is a valid program in the `asm-pred-lang-v7/spilled` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/spilled : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
             tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
```

```
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(asm-pred-lang-v7/assignments? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v7/assignments` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v7/assignments : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-
             contract (info/c (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
```

```
        |  (return-point label tail)

  opand ::= int64
        |  loc

   triv ::= opand
        |  label

    loc ::= rloc
        |  aloc

    trg ::= label
        |  loc

  binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(nested-asm-lang-fvars-v7? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-fvars-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-fvars-v7 : grammar?

```
      p ::= (module (define label tail) ... tail)

   pred ::= (relop loc opand)
```

```
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (jump trg)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  triv ::= opand
        |  label

 opand ::= int64
        |  loc

   trg ::= label
        |  loc

   loc ::= reg
        |  fvar

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
```

```
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

   aloc ::= aloc?

   fvar ::= fvar?

  label ::= label?
```

(**nested-asm-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v7 : grammar?

```
          p ::= (module (define label tail) ... tail)

       pred ::= (relop loc opand)
            |  (true)
            |  (false)
            |  (not pred)
            |  (begin effect ... pred)
            |  (if pred pred pred)

       tail ::= (jump trg)
            |  (begin effect ... tail)
            |  (if pred tail tail)

     effect ::= (set! loc triv)
            |  (set! loc_1 (binop loc_1 opand))
            |  (begin effect ... effect)
            |  (if pred effect effect)
            |  (return-point label tail)

       triv ::= opand
            |  label

      opand ::= int64
```

```
          |  loc

    trg ::= label
          |  loc

    loc ::= reg
          |  addr

    reg ::= rsp
          |  rbp
          |  rax
          |  rbx
          |  rcx
          |  rdx
          |  rsi
          |  rdi
          |  r8
          |  r9
          |  r12
          |  r13
          |  r14
          |  r15

  binop ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int64 ::= int64?

   aloc ::= aloc?

   addr ::= (fbp - dispoffset)

    fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

  label ::= label?
```

(**block-pred-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-pred-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

block-pred-lang-v7 : grammar?

```
   p ::= (module b ... b)

   b ::= (define label tail)

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)

tail ::= (jump trg)
       | (begin s ... tail)
       | (if pred (jump trg) (jump trg))

   s ::= (set! loc triv)
       | (set! loc_1 (binop loc_1 opand))

triv ::= opand
       | label

opand ::= int64
       | loc

 trg ::= label
       | loc

 loc ::= reg
       | addr

 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r12
       | r13
```

```
          |  r14
          |  r15

    binop ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right

    relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

    int64 ::= int64?

     aloc ::= aloc?

     addr ::= (fbp - dispoffset)

      fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

    label ::= label?
```

(**block-asm-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-asm-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

block-asm-lang-v7 : grammar?

```
       p ::= (module b ... b)

       b ::= (define label tail)

    tail ::= (jump trg)
          |  (begin s ... tail)
          |  (if (relop loc opand) (jump trg) (jump trg))

       s ::= (set! loc triv)
          |  (set! loc_1 (binop loc_1 opand))
```

```
 triv ::= opand
        | label

opand ::= int64
        | loc

  trg ::= label
        | loc

  loc ::= reg
        | addr

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?
```

```
dispoffset ::= dispoffset?

     label ::= label?
```

(**para-asm-lang-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `para-asm-lang-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v7 : grammar?

```
         p ::= (begin s ...)

         s ::= (set! loc triv)
            |  (set! loc_1 (binop loc_1 opand))
            |  (with-label label s)
            |  (jump trg)
            |  (compare loc opand)
            |  (jump-if relop trg)

       trg ::= label
            |  loc

      triv ::= opand
            |  label

     opand ::= int64
            |  loc

       loc ::= reg
            |  addr

       reg ::= rsp
            |  rbp
            |  rax
            |  rbx
            |  rcx
            |  rdx
            |  rsi
            |  rdi
            |  r8
            |  r9
            |  r12
            |  r13
            |  r14
            |  r15
```

```
 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

label ::= label?

dispoffset ::= dispoffset?
```

(**paren-x64-v7?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v7 : grammar?

```
   p ::= (begin s ...)

   s ::= (set! addr int32)
       | (set! addr trg)
       | (set! reg loc)
       | (set! reg triv)
       | (set! reg_1 (binop reg_1 int32))
       | (set! reg_1 (binop reg_1 loc))
       | (with-label label? s)
       | (jump trg)
       | (compare reg opand)
       | (jump-if relop label)

 trg ::= reg
       | label
```

```
 triv ::= trg
        | int64

opand ::= int64
        | reg

  loc ::= reg
        | addr

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r10
        | r11
        | r12
        | r13
        | r14
        | r15

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int32 ::= int32?

int64 ::= int64?
```

```
      label ::= label?

  dispoffset ::= dispoffset?
```

(paren-x64-rt-v7? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-rt-v7` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-rt-v7 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! loc triv)
             |  (set! reg fvar)
             |  (set! reg_1 (binop reg_1 int32))
             |  (set! reg_1 (binop reg_1 loc))
             |  (jump trg)
             |  (compare reg opand)
             |  (jump-if relop pc-addr)

        trg ::= reg
             |  pc-addr

       triv ::= trg
             |  int64

      opand ::= int64
             |  reg

        loc ::= reg
             |  addr

        reg ::= rsp
             |  rbp
             |  rax
             |  rbx
             |  rcx
             |  rdx
             |  rsi
             |  rdi
             |  r8
             |  r9
             |  r10
             |  r11
             |  r12
             |  r13
```

```
          |  r14
          |  r15

   addr ::= (fbp - dispoffset)

    fbp ::= frame-base-pointer-register?

  binop ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int32 ::= int32?

  int64 ::= int64?

  label ::= label?

dispoffset ::= dispoffset?
```

4.10 Data types: Structured Data and Heap Allocation

4.10.1 Preface: What's wrong with Exprs-Lang v7

Exprs-lang v7 gained proper data types and algebraic expressions, which is a huge step forward in expressivity and high-level reasoning. However, it still does not allow us to express structured data. Real languages require structured data—such as strings, vectors, and linked lists—to express interesting programs over data larger than a single word. Functional languages use procedures, a data structure, to provide functions as first-class values.

To express data larger than a single word, we need support from the low-level languages to get access to locations larger than a single word in size. All our locations so far, registers and frame locations, are only a single word in size. We need access to *heap pointers*, memory locations whose size can be arbitrary.

Our strategy is to add three forms that `specify-representation` use to create new data structures for its surface language. These forms are:

- `(alloc e)` allocates a number of bytes specified by `e` and returns a pointer to the base address of those bytes.
- `(mref e_base e_index)` dereferences the pointer at `e_base` with the offset specified by `e_index`. Thinking in terms of pointer arithmetic, this dereferences `(+ e_base e_index)`. The value of `(+ e_base e_index)` should always be word-aligned, *i.e.,* a multiple of 8, and point to a an initialized heap allocated value.
- `(mset! e_base e_index e)` stores the value of `e` in the address `(+ e_base e_index)`, *i.e.,* in the address given by pointer at `e_base` with the offset specified by `e_index`. The value of `(+ e_base e_index)` should always be word-aligned, *i.e.,* a multiple of 8.

To implement these new memory operations, or *mops* (pronounced *em ops*), we need to expose additional features from x64.

4.10.2 Exposing Heap Pointers in the Back-end

4.10.2.1 generate-x64

We start by exposing a generalized `addr` form in *Paren-x64 v8* below, which will allow us to access arbitrary memory locations.

v7 Diff (excerpt) v8 Full

```
p ::= (begin s ...)
```

```
    s ::= (set! addr int32)
        | (set! addr trg)
        | (set! reg loc)
        | (set! reg triv)
        | (set! reg_1 (binop reg_1 int32))
        | (set! reg_1 (binop reg_1 loc))
        | (with-label label? s)
        | (jump trg)
        | (compare reg opand)
        | (jump-if relop label)

  trg ::= reg
        | label

 triv ::= trg
        | int64

opand ::= int64
        | reg

  loc ::= reg
        | addr

 addr ::= (fbp - dispoffset)
        | (reg + int32)+
        | (reg + reg)+
```

The language contains a new *addr* representing the x64 index-mode operand `(reg + reg)`. This supports accessing a memory location by the index stored in another register. For example, in x64, we represent loading the *n*th element of an array into *r10* using `mov r10 [r11 + r12]`, where the base of the array is stored at `r11` and the value of *n* is stored in `r12`.

The index-mode operand is not restricted to use a particular register, unlike the displacement-mode operand from Paren-x64 v7. We will use this feature to store pointers to structured data, and the register allocator will move those pointers into whichever registers it chooses.

We also allow a generalized form of displacement-mode operand. We can access the value pointed to by a base register *reg* at offset *int32*. by `(reg + int32)`. This allows optimizing heap accesses when a constant offset is known, which is often the case for some data structures. The index is not restricted to be a multiple of 8, but it should be the case in our compiler that the value of the base plus the value of the offset is a multiple of 8.

All languages with direct access to registers, including Paren-x64 v8, are now parameterized by a new register, `current-heap-base-pointer-register` (abbreviated *hbp*). The run-time system initializes this register to point to the base of the heap. Allocation is implemented by copying the current value of this pointer, and incrementing it by the number of bytes we wish to allocate. The pointer must only be incremented by word-size multiples of bytes. Any other access to this register is now undefined behavior, similar to accesses to *fbp* that do not obey the stack of frames

discipline.

> *Design digression:*
>
> A real language implementation might abstract access to the `mmap` system call for allocation, and implement a strategy (such as garbage collection) to deallocate memory that is no longer used. Garbage collection is tricky to implement and requires too much time for this course, so we use a different strategy. Our implementation of allocation is trivial, and does not support de-allocation. We rely on the operating system to clean up memory after our process exits.
>
> For a quick introduction to garbage collection, see this short video https://twitter.com/TartanLlama/status/1296413612907663361?s=20.

`(generate-x64 s) → string?` procedure

`s : paren-x64-v8`

Compile the Paren-x64 v8 program into a valid sequence of x64 instructions, represented as a string.

4.10.2.2 Em-Ops: Abstracting Memory Operations

Like when we implemented *fvar*s to support working with frame locations, we implement primitive memory operations, mops , to simplify working with heap addresses. We should do this before `patch-instructions` to avoid complicating the already complex logic for rewriting *set!* instructions.

Below, we define *Paren-x64-mops v8*, with differences compared to Paren-x64-v8.

v8 Diff (excerpts) Full

```
p ::= (begin s ...)

s ::= (set! addr int32)
    | (set! addr trg)
    | (set! reg loc)
    | (set! reg triv)
    | (set! reg_1 (binop reg_1 int32))
    | (set! reg_1 (binop reg_1 loc))
    | (set! reg_1 (mref reg_2 index))+
    | (mset! reg_1 index triv)+
    | (with-label label+ label?− s)
    | (jump trg)
    | (compare reg opand)
    | (jump-if relop label)

addr ::= (fbp - dispoffset)
```

We add two new instructions that directly map to operations on heap addresses, either as index- or displacement-mode operands.

> We could encode the restricted form of *addr*, used for frame variables, as an `mref`,

> but the rest of our compiler already knows about *addr*s, and it represents a semantically different concept, so we leave it alone.

`(implement-mops p)` → `paren-x64-v8?` procedure

p : `paren-x64-mops-v8?`

Compiles mops to instructions on pointers with index- and displacement-mode operands.

Next we design *Para-asm-lang v8*. Below, we give a definition.

v7 Diff (excerpts) source/target Diff Full

```
           p ::= (begin s ...)

           s ::= (set! loc triv)
              |  (set! loc_1 (binop loc_1 opand))
              |  (set! loc_1 (mref loc_2 index))+
              |  (mset! loc_1 index triv)+
              |  (with-label label s)
              |  (jump trg)
              |  (compare loc opand)
              |  (jump-if relop trg)

        triv ::= opand
              |  label

         loc ::= reg
              |  addr

      index+ ::= int64
              |  loc

        addr ::= (fbp - dispoffset)

         fbp ::= frame-base-pointer-register?

      int32+ ::= int32?

  dispoffset ::= dispoffset?
```

By introducing mops, we implicitly restrict how heap addresses appear in the language and simplify the job of `patch-instructions`. The mops implicitly restrict heap addresses to being part of a move instruction, so we do not have to patch binary operation instructions despite apparently adding a new form of physical location. By making them separate forms, we only need to patch the new instructions, and leave old code untouched.

In `patch-instructions`, we also lift the restriction on *index*, so *int64*s can appear as an *index*. This makes *index* and *opand* coincide, syntactically, but they are conceptually different so we maintain separate non-terminal definitions.

(**patch-instructions** *s*) → paren-x64-mops-v8 procedure
s : para-asm-lang-v8

Patches instructions that have no x64 analogue into to a sequence of instructions and an auxiliary register from `current-patch-instructions-registers`.

4.10.2.3 Exposing mops up the pipeline

The new mops require minor changes to most of the pipeline up to Asm-alloc-lang v8, where we will use them to implement data structures.

Exercise 9: Redesign and extend the implementation of

- `flatten-program`, should require no changes
- `resolve-predicates`, should require no changes
- `expose-basic-blocks`, should require no changes
- `implement-fvars`, should require minor changes to support mops. Note that we assume the *fbp* is not modified by mops.
- `optimize-predicates`, could require minor changes.
- `replace-locations`, should require minor changes to support mops.
- `assign-frame-variables`, should require no changes.
- `assign-registers`, should require no changes.
- `allocate-frames`, should require no changes.
- `assign-call-undead-variables`, should require no changes.
- `conflict-analysis`, should require minor changes. Note that mops do not *assign* any registers or frame variables.
- `undead-analysis`, should require minor changes. Note that mops do not *assign* any registers or frame variables.
- `uncover-locals`, should require minor changes.

4.10.2.4 Implementing Allocation

Before we introduce structured data, we implement the `alloc` instruction to allow programs to allocate a bunch of bytes and not worry about the details of the allocation pointer. We want to do this *after* the passes that analyze physical locations, since then we do not have to update those passes to know that `alloc` introduces a reference to a register. However, we want to do this *before* we abstract away from all machine details so we do not need to expose registers beyond Asm-alloc-lang v8.

We choose to insert this pass between `uncover-locals` and `select-instructions`.

Below, we design *Asm-alloc-lang v8*, the source language for this pass. We typeset the differences compared to Asm-pred-lang v7.

v7 Diff (excerpts) Full

```
       p ::= (module info (define label info tail) ... tail)

    info ::= (#:from-contract (info/c (new-frames (frame ...))))

   frame ::= (aloc ...)

    pred ::= (relop loc opand)
           | (true)
           | (false)
           | (not pred)
           | (begin effect ... pred)
           | (if pred pred pred)

    tail ::= (jump trg loc ...)
           | (begin effect ... tail)
           | (if pred tail tail)

  effect ::= (set! loc triv)
           | (set! loc_1 (binop loc_1 opand))
           | (set! loc_1 (mref loc_2 index))+
           | (set! loc (alloc index))+
           | (mset! loc index triv)+
           | (begin effect ... effect)
           | (if pred effect effect)
           | (return-point label tail)

  index+ ::= int64
           | loc

  int32+ ::= int32?
```

We expose the earlier mops, and add a new one, `(set! loc (alloc index))`. This is the low-level form of our allocation operation, which we will abstract into an expression in Exprs-lang v8. In `(set! loc (alloc index))`, the *index* is restricted to be an *int32* if it is an integer literal, for the same reasons as the restriction on *binop*s. It must also be a multiple of a word size. This requirement is partly from the operating system's `mmap` (which will usually ignore us if we violate the restriction and give us page-aligned memory anyway), but mostly to ensure every pointer we get has `#b000` as its final three bits so we can tag all pointers.

We design the target language, *Asm-pred-lang v8*, below. This language removes the `(alloc index)` form and is the highest-level language parameterized by `current-heap-base-pointer-register`. This language, and all languages between it and x64, assumes all accesses to *hbp* obey the restrictions described in Paren-x64 v8.

Source/Target Diff (excerpts) v7 Diff (excerpts) Full

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (set! loc_1 (mref loc_2 index))
          | (set! loc (alloc index))
          | (mset! loc index triv)
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  index ::= int64
          | loc
```

Intuitively, we will transform each `` `(set! ,loc (alloc ,index)) `` into

```
`(begin
   (set! ,loc ,hbp)
   (set! ,hbp (+ ,hbp ,index)))
```

(expose-allocation-pointer *p*) → asm-pred-lang-v8? procedure

p : asm-alloc-lang-v8?

Implements the allocation primitive in terms of pointer arithmetic on the current-heap-base-pointer-register.

4.10.2.5 Abstracting Mops

Before we implement structured data, we expose our mops through a few layers of abstractions. Below we design *Imp-cmf-lang v8* with support for mops. We typeset the differences compared to Imp-cmf-lang v7.

v7 Diff (excerpts) Source/Target Diff Full

```
      p ::= (module info (define label info tail) ... tail)

   pred ::= (relop opand opand)
```

```
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
        | (binop opand opand)
        | (mref loc opand)+
        | (alloc opand)+

effect ::= (set! loc value)
        | (mset! loc opand triv)+
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)
```

We add value forms of `mref` and `alloc`. We require these forms are used in a well-typed way. This is a simple extension.

(select-instructions *p*) → asm-alloc-lang-v8? procedure
p : imp-cmf-lang-v8?

Selects appropriate sequences of abstract assembly instructions to implement the operations of the source language.

Next we design *Imp-mf-lang v8* with support for mops.

v7 Diff (excerpts) Full

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

value ::= triv
```

```
          | (binop opand opand)
          | (mref loc opand)+
          | (alloc opand)+
          | (begin effect ... value)
          | (if pred value value)
          | (return-point label tail)

  effect ::= (set! loc value)
          | (mset! loc opand value)+
          | (begin effect ... effect)
          | (if pred effect effect)
```

We introduce the *value* context, but notice that the index position for an `mset!` instruction is still restricted.

Question: Why can't (or shouldn't) we allow the index position to also be a *value*?

`(canonicalize-bind p)` → `imp-cmf-lang-v8?` procedure

p : `imp-mf-lang-v8?`

Pushes `set!` and `mset!` under `begin` and `if` so that the right-hand-side of each is simple value-producing operand.

This canonicalizes Imp-mf-lang v8 with respect to the equations

```
(set! loc (begin effect_1 ... value))   =(begin effect_1 ... (set! loc value))
(set! loc (if pred value_1 value_2))    =(if pred (set! loc value_1) (set! loc value_2))
(mset! loc opand (begin effect_1 ... value))   =(begin effect_1 ... (mset! loc opand value))
(mset! loc opand (if pred value_1 value_2))    =(if pred (mset! loc opand value_1) (mset! loc opand value_2))
```

The `impose-calling-conventions` pass requires only minor changes.

For `sequentialize-let`, we need to design an *effect* context in order to expose `mset!`. We design *Values-bits-lang v8* to include a few imperative features.

v7 Diff (excerpts) Source/Target Diff (Excerpts) Full

```
        p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

     pred ::= (relop opand opand)
            | (true)
            | (false)
            | (not pred)
            | (let ([aloc value] ...) pred)
            | (if pred pred pred)
            | (begin effect ... pred)+

     tail ::= value
            | (let ([aloc value] ...) tail)
```

```
          | (if pred tail tail)
          | (call triv opand ...)
          | (begin effect ... tail)+

   value ::= triv
          | (binop opand opand)
          | (mref aloc opand)+
          | (alloc opand)+
          | (let ([aloc value] ...) value)
          | (if pred value value)
          | (call triv opand ...)
          | (begin effect ... value)+

effect+ ::= (mset! aloc opand value)
          | (let ([aloc value] ...) effect)
          | (begin effect ... effect)
```

We add an *effect* context to support `mset!`, and a *begin* expression for convenience. Previously, all expressions in the Values-lang languages were *pure*—they evaluated and produced the same value regardless of in which order expressions were evaluated. We could freely reorder expressions, as long as we respected scope. Now, however, `mset!` modifies memory during its execution. It not safe to reorder expressions after an `mset!`. Furthermore, `mset!` does not return a useful value.

To deal with this, we introduce a contextual distinction in the language. We add the nonterminal *effect* to represent an impure computation. A *effect* represents an expression that does not have a value, and is executed only for its effect. We can use *effect* in certain expression contexts using *begin*. If we're already in an impure context, that is, in a *effect*, then we can freely nest other *effect*s.

This contextual distinction is similar to the one we introduce to distinguish tail calls from non-tail calls.

Despite the conceptually complex change to the language, the transformation is still straightforward.

> *Design digression:*
>
> Now that effects can appear on the right-hand side of a `let` expression, it MAY not longer be safe to reorder them. This is a design choice: we could make it clear to the programmer that `let` does not guarantee a particular order of evaluation for its bindings, but then effects on the right-hand side lead to undefined behaviour. Or, we could impose a particular order, such as left-to-right, forbidding an optimization. A middle ground is to impose such an order only if any effects are detected in the right-hand side of a `let` (or rather, if we can guarantee no effects are present, because Rice still does not let us know for sure).

(sequentialize-let *s*) → proc-imp-mf-lang-v8? — procedure

s : values-bits-lang-v8?

Picks a particular order to implement `let` expressions using `set!`.

Finally, we enable arbitrary nesting in value position. We design *Exprs-bits-lang v8/contexts* below.

v7 Diff (excerpts) Full

```
        p ::= (module b ... tail)

        b ::= (define label (lambda (aloc ...) tail))

     pred ::= (relop value value)
            | (true)
            | (false)
            | (not pred)
            | (let ([aloc value] ...) pred)
            | (if pred pred pred)
            | (begin effect ... pred)+

     tail ::= value
            | (let ([aloc value] ...) tail)
            | (if pred tail tail)
            | (begin effect ... tail)+

    value ::= triv
            | (binop value value)
            | (mref value value)+
            | (alloc value)+
            | (call value value ...)
            | (let ([aloc value] ...-) value)
            | (if pred value value)
            | (begin effect ... value)+

  effect+ ::= (mset! value value value)
            | (begin effect ... effect)
```

Supporting *effect* context requires paying attention to order when designing `remove-complex-opera*`, but does not significantly complicate anything.

(**remove-complex-opera*** *p*) → values-bits-lang-v8? procedure
p : exprs-bigs-lang-v8/contexts?

Performs the monadic form transformation, unnesting all non-trivial operators and operands, making data flow explicit and and simple to implement imperatively.

4.10.2.6 Implementing Structured Data

Now we have all the abstractions necessary to implement structured data.

We design a new *Exprs-unsafe-data-lang v8* below. The language is large, as we include several new structured data types and their primitives.

v7 Diff (excerpts) Full

```
        p ::= (module b ... e)

        b ::= (define label (lambda (aloc ...) e))

     pred ::= e
            | (true)
            | (false)
            | (not pred)
            | (let ([aloc e] ...) pred)
            | (if pred pred pred)

        e ::= v
            | (primop e ...)
            | (call e e ...)
            | (let ([aloc e] ...) e)
            | (if pred e e)
            | (begin effect ... e)+

  effect+ ::= (primop e ...)
            | (begin effect ... effect)

  primop+ ::= unsafe-fx*
            | unsafe-fx+
            | unsafe-fx-
            | eq?
            | unsafe-fx<
            | unsafe-fx<=
            | unsafe-fx>
            | unsafe-fx>=
            | fixnum?
            | boolean?
            | empty?
            | void?
            | ascii-char?
            | error?
            | not
            | pair?
            | vector?
            | cons
            | unsafe-car
            | unsafe-cdr
            | unsafe-make-vector
            | unsafe-vector-length
            | unsafe-vector-set!
            | unsafe-vector-ref

  primop- ::= binop
            | unop
```

```
binop⁻ ::= unsafe-fx*
         | unsafe-fx+
         | unsafe-fx-
         | eq?
         | unsafe-fx<
         | unsafe-fx<=
         | unsafe-fx>
         | unsafe-fx>=

 unop⁻ ::= fixnum?
         | boolean?
         | empty?
         | void?
         | ascii-char?
         | error?
         | not
```

We add new primops for each of our new data types. Since the number of primitive operations is growing, we simplify the syntax to only give *primops*, rather than distinguishing *unops*, *binops*, and so on, so we can easily group like primops with like.

We also add impure computations, since vectors are a mutable data structure. Only effectful *primops*, those ending with `!`, are allowed in effect context.

We add two heap-allocated data types, described below:

- *Pairs* are constructed using `(cons e_1 e_2)`. The predicate `pair?` should return `#t` when passed any value constructed this way, and #f for any other value—`(eq? (pair? (cons e_1 e_2)) #t)`. `(unsafe-car e)` returns the value of the first element of the pair, and `(unsafe-cdr e)` returns the value of the second element. That is, `(eq? (unsafe-car (cons e_1 e_2)) e_1)` and `(eq? (unsafe-cdr (cons e_1 e_2)) e_2)`.

- *Vectors* are essentially arrays that know their length. They are constructed using `(unsafe-make-vector e)`; the constructor takes the length of the vector as the argument. The predicate `vector?` should return `#t` for any value constructed this way, and #f for any other value—`(eq? (vector? (unsafe-make-vector e)) #t)`. `(unsafe-vector-ref e_1 e_2)` returns the value at index `e_2` in the vector `e_1`. `(unsafe-vector-set! e_1 e_2 e_3)` mutates the index `e_2` in the vector `e_1`, setting its value to the value of `e_3`.

  `(unsafe-vector-set! e_1 e_2 e_3)` is only allowed in impure computation context.

As we're adding new data types, we need new tags. Here is our updated list of tags: Here is the set of tags we will use in this assignment, given in base 2.

- `#b000`, *fixnums*, fixed-sized integers

- `#b001`, pairs

- `#b010`, *unused*

- `#b011`, vectors
- `#b100`, *unused*
- `#b101`, *unused*
- `#b110`, non-fixnum immediates (booleans, etc)
- `#b111`, *unused*

We follow the same pattern as Data types: Immediates to implement the new predicates.

To implement constructors, we need to compile to *`alloc`*. `cons` allocates two words of space, storing its first argument in the first word and the second element in the second word, producing `` `(alloc ,(current-pair-size)) ``. The ptr we get back needs to be tagged, so we produce `` `(bitwise-ior (alloc ,(current-pair-size)) ,(current-pair-tag)) ``. `unsafe-make-vector` allocates one word for the length, and then one word for every element of the vector. That is, it should allocate `n+1` words for a vector of length `n`.

After allocating, we initialize the data structures using `mset!`. For example, we would transform `` `(cons ,e_1 ,e_2) ``.

```
`(let ([x.1 (bitwise-ior (alloc 16) ,(current-pair-tag))])
   (begin
     (mset! (bitwise-xor x.1 ,(current-pair-tag)) 0 ,e_1)
     (mset! (bitwise-xor x.1 ,(current-pair-tag)) 8 ,e_2)
     x.1))
```

Since the length of the vector is dynamically determined, we do not initialize each field when implementing its constructor. Instead, we expose an unsafe constructor for vectors, and leave it to a safer language to initialize the vector.

We can optimize memory operations to avoid masking the pointer by taking advantage of pointer arithmetic. For example, `(bitwise-ior (alloc 16) 1)` is the same as `(+ (alloc 16) 1)`. We can therefore adjust the index by -1 to access the base of the pointer, instead of masking the pointer. Performing this optimization for pairs, we would instead transform `` `(cons ,e_1 ,e_2) `` into

```
`(let ([x.1 (+ (alloc 16) 1)])
   (begin
     (mset! x.1 -1 ,e_1)
     (mset! x.1 7 ,e_2)
     x.1))
```

The same optimization holds for vectors with different constants.

(specify-representation *p***)** → exprs-bits-lang-v8/contexts? procedure
p : exprs-unsafe-data-lang-v8?

Compiles data types and primitive operations into their implementations as ptrs and primitive bitwise operations on ptrs.

4.10.2.7 New Safe Primops

All the accessors for the new data types can result in undefined behaviour if used on the wrong ptr. Similarly, vector reference can access undefined values if the vector is constructed but never initialized.

Below we define *Exprs-unique-lang v8*.

v7 Diff (excerpts) Full

```
p ::= (module b ... e)

b ::= (define label (lambda (aloc ...) e))

e ::= v
    | (call e e ...)
    | (let ([aloc e] ...) e)
    | (if e e e)

v ::= label
    | aloc
    | prim-f
    | fixnum
    | #t
    | #f
    | empty
    | (void)
    | (error uint8)
    | ascii-char-literal

prim-f+ ::= *
          | +
          | -
          | <
          | <=
          | >
          | >=
          | eq?
          | fixnum?
          | boolean?
          | empty?
          | void?
          | ascii-char?
          | error?
          | not
          | pair?
          | vector?
          | cons
          | car
          | cdr
          | make-vector
          | vector-length
```

```
          |  vector-set!
          |  vector-ref

  prim-f¯ ::= binop
          |  unop

   binop¯ ::= *
          |  +
          |  -
          |  <
          |  eq?
          |  <=
          |  >
          |  >=

    unop¯ ::= fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
```

Note that in this language, we remove `begin`. The user must manually call impure functions and bind the result. The result of an effectful function could be `void`, or an `error`. It would be unwise, although technically safe, to simple discard errors.

To implement this safe language, we wrap all accessors to perform dynamic tag checking before using the unsafe operations. We also wrap `unsafe-make-vector` to initialize all elements to `0`.

Writing a compiler for the following specification language may simplify this task:

```
; Symbol x Symbol x (List-of Parameter-Types)
; The first symbol is the name of a function in the source language.
; The second is either the name of a primop or a label in the target language implementing the
; behaviour safely, assuming well-typed parameters.
; The third is list of predicates, one for each argument to the source
; function, to check the parameters with. `any?` is specially recognized to
; not be checked.
(define prim-f-specs
  `((* unsafe-fx* (fixnum? fixnum?))
    (+ unsafe-fx+ (fixnum? fixnum?))
    (- unsafe-fx- (fixnum? fixnum?))
    (< unsafe-fx< (fixnum? fixnum?))
    (<= unsafe-fx<= (fixnum? fixnum?))
    (> unsafe-fx> (fixnum? fixnum?))
    (>= unsafe-fx>= (fixnum? fixnum?))

    (make-vector ,make-init-vector-label (fixnum?))
    (vector-length unsafe-vector-length (vector?))
    (vector-set! ,unsafe-vector-set!-label (vector? fixnum? any?))
```

```
    (vector-ref ,unsafe-vector-ref-label (vector? fixnum?))

    (car unsafe-car (pair?))
    (cdr unsafe-cdr (pair?))

    ,@(map (lambda (x) `(,x ,x (any?)))
           '(fixnum? boolean? empty? void? ascii-char? error? pair?
                     vector? not))
    ,@(map (lambda (x) `(,x ,x (any? any?)))
           '(cons eq?))))
```

All impure computations, those that end in `!`, should only return `(void)` or an `error`.

(implement-safe-primops *p*) → exprs-unsafe-data-lang-v8 procedure

p : exprs-unique-lang-v8

Implement safe primitive operations by inserting procedure definitions for each primitive operation which perform dynamic tag checking, to ensure type and memory safety.

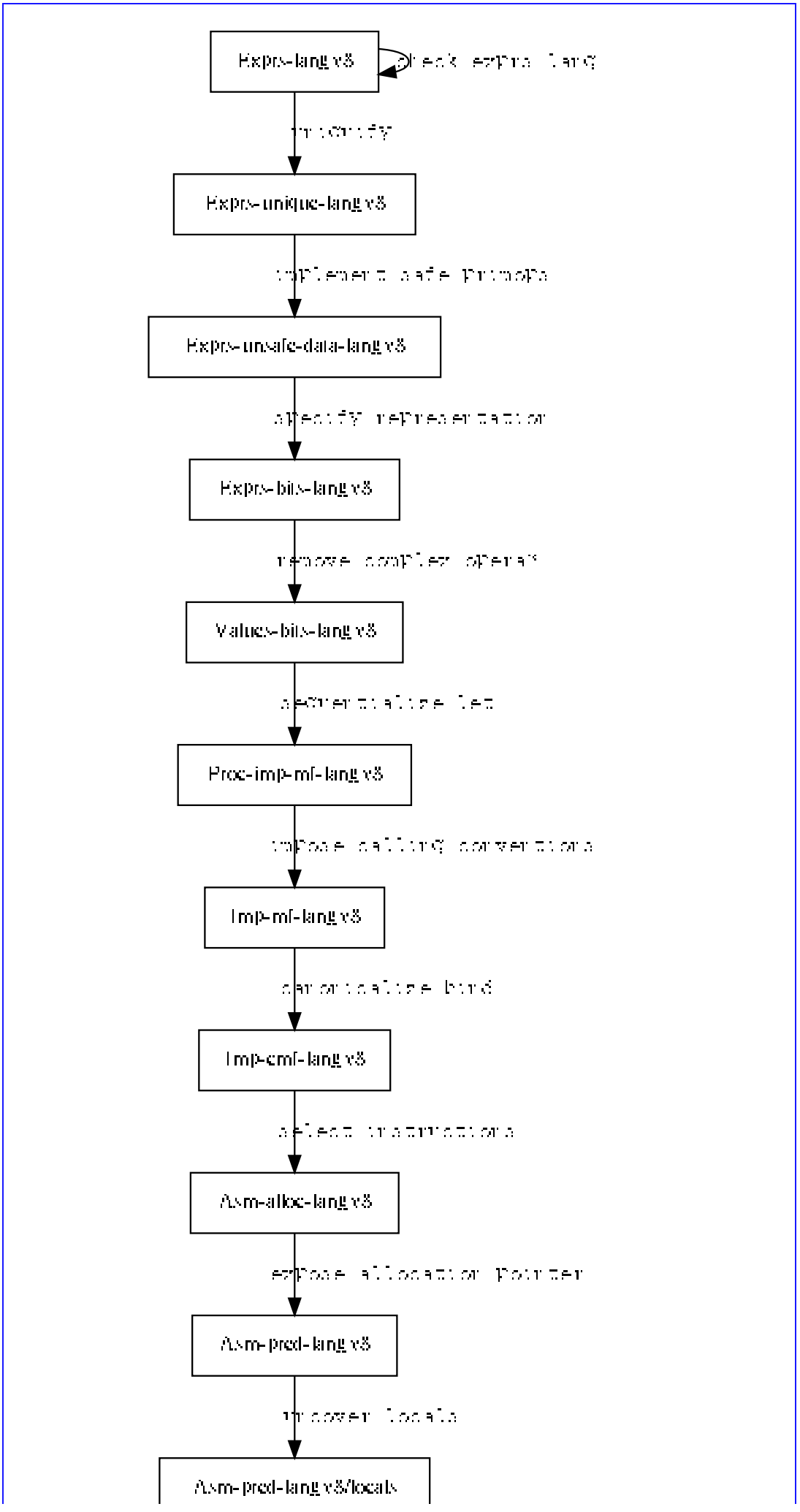
4.10.2.8 uniquify

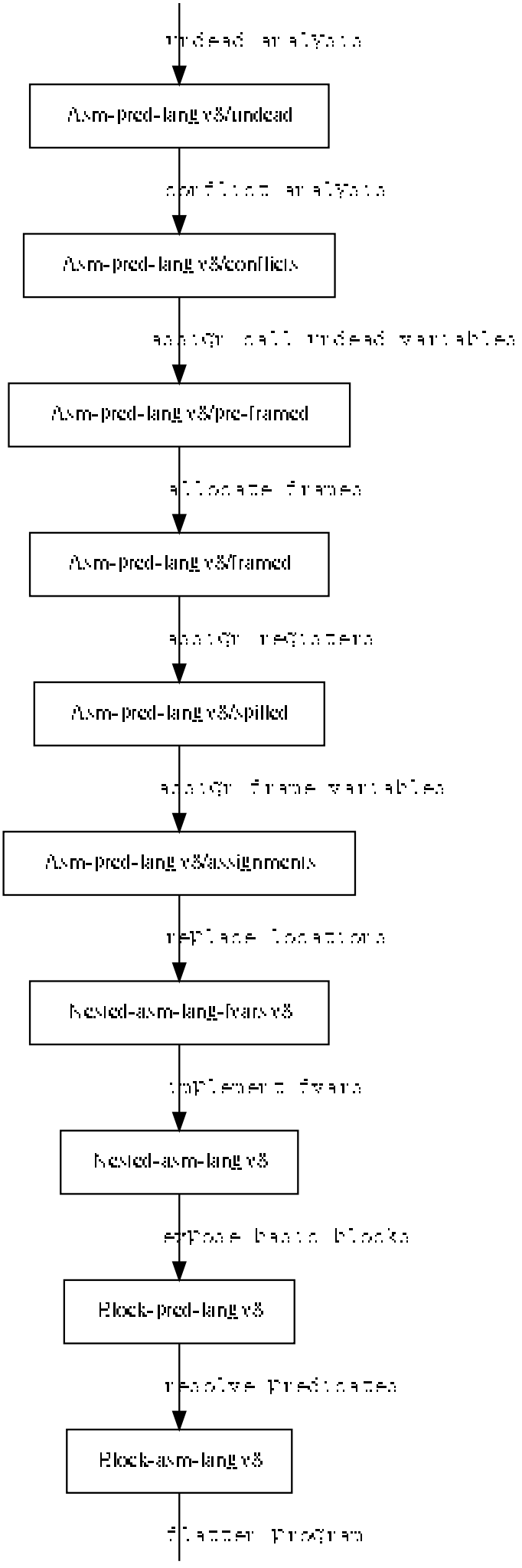
Finally, we define the source language. Below we define *Exprs-lang v8*.

```
                p ::= (module b ... e)

                b ::= (define x (lambda (x ...) e))

                e ::= v
                    | (call e e ...)
                    | (let ([x e] ...) e)
                    | (if e e e)

                x ::= name?
                    | prim-f

                v ::= x
                    | fixnum
                    | #t
                    | #f
                    | empty
                    | (void)
                    | (error uint8)
                    | ascii-char-literal

          prim-f+ ::= *
                    | +
                    | -
                    | <
                    | <=
                    | >
                    | >=
```

```
                | eq?
                | fixnum?
                | boolean?
                | empty?
                | void?
                | ascii-char?
                | error?
                | not
                | pair?
                | vector?
                | cons
                | car
                | cdr
                | make-vector
                | vector-length
                | vector-set!
                | vector-ref

        prim-f¯ ::= binop
                | unop

         binop¯ ::= *
                | +
                | -
                | eq?
                | <
                | <=
                | >
                | >=

          unop¯ ::= fixnum?
                | boolean?
                | empty?
                | void?
                | ascii-char?
                | error?
                | not

         fixnum ::= int61?

          uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

4.10.3 Appendix: Overview


Expr-lang v8
check expr-lang
uniquify
Expr-unique-lang v8
implement safe primops
Expr-unsafe-data-lang v8
specify representation
Expr-bits-lang v8
remove complex opera*
Values-bits-lang v8
sequentialize let
Proc-imp-mf-lang v8
impose calling conventions
Imp-mf-lang v8
flatten set!
Imp-cmf-lang v8
select instructions
Asm-alloc-lang v8
expose allocation pointer
Asm-pred-lang v8
uncover locals
Asm-pred-lang v8/locals


undead analysis
Asm-pred-lang v8/undead
conflict analysis
Asm-pred-lang v8/conflicts
assign call-undead variables
Asm-pred-lang v8/pre-framed
allocate frames
Asm-pred-lang v8/framed
assign registers
Asm-pred-lang v8/spilled
assign frame variables
Asm-pred-lang v8/assignments
replace locations
Nested-asm-lang-fvars v8
implement fvars
Nested-asm-lang v8
expose basic blocks
Block-pred-lang v8
resolve predicates
Block-asm-lang v8
flatten program

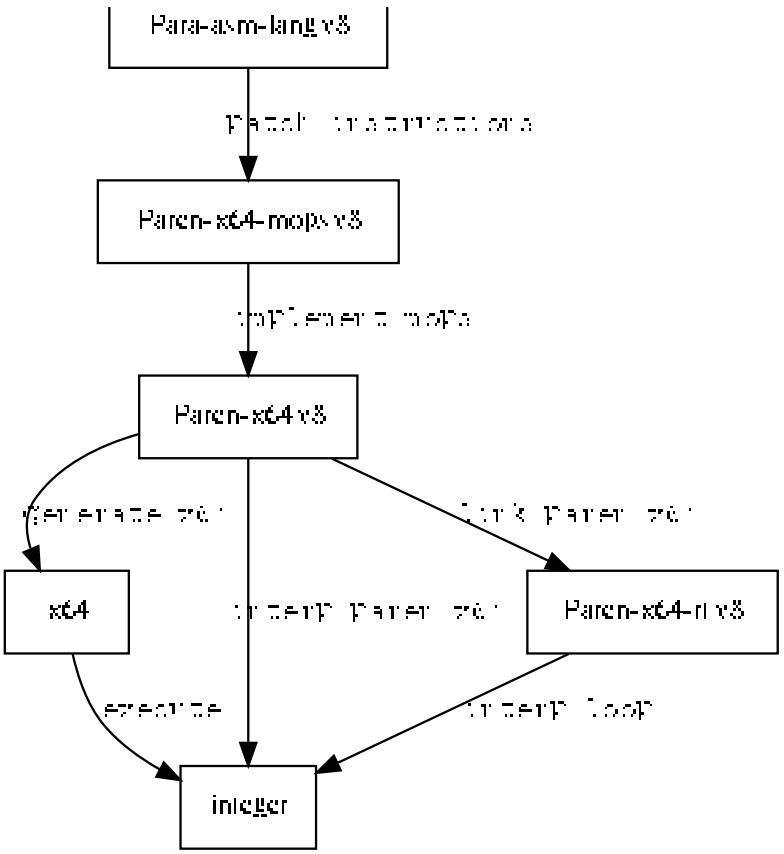


Figure 8: Overview of Compiler Version 8

4.10.4 Appendix: Languages

`(exprs-lang-v8? a) → boolean?` procedure

a : any/c

Decides whether *a* is a valid program in the `exprs-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-lang-v8 : grammar?

```
p ::= (module b ... e)

b ::= (define x (lambda (x ...) e))

e ::= v
    | (call e e ...)
    | (let ([x e] ...) e)
    | (if e e e)

x ::= name?
    | prim-f

v ::= x
    | fixnum
```

```
                  | #t
                  | #f
                  | empty
                  | (void)
                  | (error uint8)
                  | ascii-char-literal

          prim-f ::= *
                  | +
                  | -
                  | <
                  | <=
                  | >
                  | >=
                  | eq?
                  | fixnum?
                  | boolean?
                  | empty?
                  | void?
                  | ascii-char?
                  | error?
                  | not
                  | pair?
                  | vector?
                  | cons
                  | car
                  | cdr
                  | make-vector
                  | vector-length
                  | vector-set!
                  | vector-ref

          fixnum ::= int61?

           uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(exprs-unique-lang-v8? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unique-lang-v8 grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-unique-lang-v8 : grammar?

```
p ::= (module b ... e)

b ::= (define label (lambda (aloc ...) e))
```

```
                e ::= v
                    | (call e e ...)
                    | (let ([aloc e] ...) e)
                    | (if e e e)

                v ::= label
                    | aloc
                    | prim-f
                    | fixnum
                    | #t
                    | #f
                    | empty
                    | (void)
                    | (error uint8)
                    | ascii-char-literal

           prim-f ::= *
                    | +
                    | -
                    | <
                    | <=
                    | >
                    | >=
                    | eq?
                    | fixnum?
                    | boolean?
                    | empty?
                    | void?
                    | ascii-char?
                    | error?
                    | not
                    | pair?
                    | vector?
                    | cons
                    | car
                    | cdr
                    | make-vector
                    | vector-length
                    | vector-set!
                    | vector-ref

            aloc ::= aloc?

           label ::= label?

           fixnum ::= int61?

            uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(**exprs-unsafe-data-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unsafe-data-lang-v8 grammar. The first non-terminal in the grammar defines valid programs.

exprs-unsafe-data-lang-v8 : grammar?

```
    p ::= (module b ... e)

    b ::= (define label (lambda (aloc ...) e))

 pred ::= e
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc e] ...) pred)
        | (if pred pred pred)

    e ::= v
        | (primop e ...)
        | (call e e ...)
        | (let ([aloc e] ...) e)
        | (if pred e e)
        | (begin effect ... e)

effect ::= (primop e ...)
        | (begin effect ... effect)

    v ::= label
        | aloc
        | fixnum
        | #t
        | #f
        | empty
        | (void)
        | (error uint8)
        | ascii-char-literal

primop ::= unsafe-fx*
        | unsafe-fx+
        | unsafe-fx-
        | eq?
        | unsafe-fx<
        | unsafe-fx<=
        | unsafe-fx>
        | unsafe-fx>=
        | fixnum?
        | boolean?
        | empty?
        | void?
```

```
                    | ascii-char?
                    | error?
                    | not
                    | pair?
                    | vector?
                    | cons
                    | unsafe-car
                    | unsafe-cdr
                    | unsafe-make-vector
                    | unsafe-vector-length
                    | unsafe-vector-set!
                    | unsafe-vector-ref

              aloc ::= aloc?

             label ::= label?

            fixnum ::= int61?

             uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(exprs-bits-lang-v8/contexts? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `exprs-bits-lang-v8/contexts` grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-bits-lang-v8/contexts : grammar?

```
      p ::= (module b ... tail)

      b ::= (define label (lambda (aloc ...) tail))

   pred ::= (relop value value)
          | (true)
          | (false)
          | (not pred)
          | (let ([aloc value] ...) pred)
          | (if pred pred pred)
          | (begin effect ... pred)

   tail ::= value
          | (let ([aloc value] ...) tail)
          | (if pred tail tail)
          | (begin effect ... tail)

  value ::= triv
          | (binop value value)
          | (mref value value)
```

```
        |  (alloc value)
        |  (call value value ...)
        |  (let ([aloc value]) value)
        |  (if pred value value)
        |  (begin effect ... value)

 effect ::= (mset! value value value)
        |  (begin effect ... effect)

   triv ::= label
        |  aloc
        |  int64

  binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

   aloc ::= aloc?

  label ::= label?

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?
```

(values-bits-lang-v8? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `values-bits-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

values-bits-lang-v8 : grammar?

```
     p ::= (module (define label (lambda (aloc ...) tail)) ... tail)

  pred ::= (relop opand opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (let ([aloc value] ...) pred)
        |  (if pred pred pred)
        |  (begin effect ... pred)
```

```
 tail ::= value
        | (let ([aloc value] ...) tail)
        | (if pred tail tail)
        | (call triv opand ...)
        | (begin effect ... tail)

value ::= triv
        | (binop opand opand)
        | (mref aloc opand)
        | (alloc opand)
        | (let ([aloc value] ...) value)
        | (if pred value value)
        | (call triv opand ...)
        | (begin effect ... value)

effect ::= (mset! aloc opand value)
        | (let ([aloc value] ...) effect)
        | (begin effect ... effect)

opand ::= int64
        | aloc

 triv ::= opand
        | label

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?
```

(**proc-imp-mf-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `proc-imp-mf-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

proc-imp-mf-lang-v8 : grammar?

```
      p ::= (module (define label (lambda (aloc ...) entry)) ...
              entry)

  entry ::= tail

   pred ::= (relop opand opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= value
          | (call triv opand ...)
          | (begin effect ... tail)
          | (if pred tail tail)

  value ::= triv
          | (binop opand opand)
          | (mref aloc opand)
          | (alloc opand)
          | (begin effect ... value)
          | (if pred value value)
          | (call triv opand ...)

 effect ::= (set! aloc value)
          | (mset! aloc opand value)
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | aloc

   triv ::= opand
          | label

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
          | arithmetic-shift-right

  relop ::= <
          | <=
          | =
          | >=
          | >
          | !=
```

```
  int64 ::= int64?

   aloc ::= aloc?

  label ::= label?
```

(**imp-mf-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-mf-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-mf-lang-v8 : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...))))

  frame ::= (aloc ...)

   pred ::= (relop opand opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

  value ::= triv
          | (binop opand opand)
          | (mref loc opand)
          | (alloc opand)
          | (begin effect ... value)
          | (if pred value value)
          | (return-point label tail)

 effect ::= (set! loc value)
          | (mset! loc opand value)
          | (begin effect ... effect)
          | (if pred effect effect)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc
```

```
  trg ::= label
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(imp-cmf-lang-v8? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `imp-cmf-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

imp-cmf-lang-v8 : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-frames (frame ...))))

frame ::= (aloc ...)

 pred ::= (relop opand opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)
```

```
 value ::= triv
         | (binop opand opand)
         | (mref loc opand)
         | (alloc opand)

effect ::= (set! loc value)
         | (mset! loc opand triv)
         | (begin effect ... effect)
         | (if pred effect effect)
         | (return-point label tail)

 opand ::= int64
         | loc

  triv ::= opand
         | label

   loc ::= rloc
         | aloc

   trg ::= label
         | loc

 binop ::= *
         | +
         | -
         | bitwise-and
         | bitwise-ior
         | bitwise-xor
         | arithmetic-shift-right

 relop ::= <
         | <=
         | =
         | >=
         | >
         | !=

 int64 ::= int64?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
         | fvar?
```

(asm-alloc-lang-v8? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-alloc-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

asm-alloc-lang-v8 : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (set! loc_1 (mref loc_2 index))
          | (set! loc (alloc index))
          | (mset! loc index triv)
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  index ::= int64
          | loc

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
          | arithmetic-shift-right

  relop ::= <
          | <=
```

```
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

  int32 ::= int32?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v8 grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8 : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  opand ::= int64
        |  loc
```

```
 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

index ::= int64
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(asm-pred-lang-v8/locals? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v8/locals` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/locals : grammar?

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (new-
          frames (frame ...)) (locals (aloc ...))))

frame ::= (aloc ...)
```

```
 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (set! loc_1 (mref loc_2 index))
        | (begin effect ... effect)
        | (mset! loc index triv)
        | (if pred effect effect)
        | (return-point label tail)

 opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

 index ::= int64
        | loc

 binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

 relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

 int64 ::= int64?

 int32 ::= int32?
```

```
 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v8/undead?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v8/undead grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/undead : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-
           frames (frame ...)) (locals (aloc ...)) (call-
           undead (loc ...)) (undead-out undead-set-tree/rloc?)))

frame ::= (aloc ...)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
        | (begin effect ... pred)
        | (if pred pred pred)

 tail ::= (jump trg loc ...)
        | (begin effect ... tail)
        | (if pred tail tail)

effect ::= (set! loc triv)
        | (set! loc_1 (binop loc_1 opand))
        | (set! loc_1 (mref loc_2 index))
        | (mset! loc index triv)
        | (begin effect ... effect)
        | (if pred effect effect)
        | (return-point label tail)

 opand ::= int64
        | loc

 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
```

```
        | loc

index ::= int64
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(asm-pred-lang-v8/conflicts? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v8/conflicts` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/conflicts : grammar?

```
    p ::= (module info (define label info tail) ... tail)

 info ::= (#:from-contract (info/c (new-
          frames (frame ...)) (locals (aloc ...)) (call-
          undead (loc ...)) (undead-out undead-set-
          tree/rloc?) (conflicts ((loc (loc ...)) ...))))

frame ::= (aloc ...)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
```

```
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

 opand ::= int64
        |  loc

  triv ::= opand
        |  label

   loc ::= rloc
        |  aloc

   trg ::= label
        |  loc

 index ::= int64
        |  loc

 binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

 relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

 int64 ::= int64?

 int32 ::= int32?

  aloc ::= aloc?

 label ::= label?

  rloc ::= register?
```

```
        | fvar?
```

(asm-pred-lang-v8/pre-framed? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v8/pre-framed` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/pre-framed : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (new-frames (frame ...)) (locals (aloc ...)) (call-
            undead (loc ...)) (undead-out undead-set-
            tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (set! loc_1 (mref loc_2 index))
          | (mset! loc index triv)
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  index ::= int64
          | loc
```

```
binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**asm-pred-lang-v8/framed?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v8/framed` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/framed : grammar?

```
   p ::= (module info (define label info tail) ... tail)

info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
         tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)
       | (begin effect ... pred)
       | (if pred pred pred)

tail ::= (jump trg loc ...)
       | (begin effect ... tail)
       | (if pred tail tail)

effect ::= (set! loc triv)
         | (set! loc_1 (binop loc_1 opand))
```

```
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  opand ::= int64
        |  loc

   triv ::= opand
        |  label

    loc ::= rloc
        |  aloc

    trg ::= label
        |  loc

  index ::= int64
        |  loc

  binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

  relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

  int32 ::= int32?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(asm-pred-lang-v8/spilled? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the asm-pred-lang-v8/spilled grammar. The first non-terminal in the grammar defines valid programs.

asm-pred-lang-v8/spilled : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (locals (aloc ...)) (undead-out undead-set-
            tree/rloc?) (conflicts ((loc (loc ...)) ...)) (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
          | (true)
          | (false)
          | (not pred)
          | (begin effect ... pred)
          | (if pred pred pred)

   tail ::= (jump trg loc ...)
          | (begin effect ... tail)
          | (if pred tail tail)

 effect ::= (set! loc triv)
          | (set! loc_1 (binop loc_1 opand))
          | (set! loc_1 (mref loc_2 index))
          | (mset! loc index triv)
          | (begin effect ... effect)
          | (if pred effect effect)
          | (return-point label tail)

  opand ::= int64
          | loc

   triv ::= opand
          | label

    loc ::= rloc
          | aloc

    trg ::= label
          | loc

  index ::= int64
          | loc

  binop ::= *
          | +
          | -
          | bitwise-and
          | bitwise-ior
          | bitwise-xor
          | arithmetic-shift-right

  relop ::= <
          | <=
```

```
        |  =
        |  >=
        |  >
        |  !=

  int64 ::= int64?

  int32 ::= int32?

   aloc ::= aloc?

  label ::= label?

   rloc ::= register?
        |  fvar?
```

(**asm-pred-lang-v8/assignments?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `asm-pred-lang-v8/assignments` grammar. The first non-terminal in the grammar defines valid programs.

---

asm-pred-lang-v8/assignments : grammar?

```
      p ::= (module info (define label info tail) ... tail)

   info ::= (#:from-contract (info/c (assignment ((aloc loc) ...))))

  frame ::= (aloc ...)

   pred ::= (relop loc opand)
        |  (true)
        |  (false)
        |  (not pred)
        |  (begin effect ... pred)
        |  (if pred pred pred)

   tail ::= (jump trg loc ...)
        |  (begin effect ... tail)
        |  (if pred tail tail)

 effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  opand ::= int64
        |  loc
```

```
 triv ::= opand
        | label

  loc ::= rloc
        | aloc

  trg ::= label
        | loc

index ::= int64
        | loc

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 aloc ::= aloc?

label ::= label?

 rloc ::= register?
        | fvar?
```

(**nested-asm-lang-fvars-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the nested-asm-lang-fvars-v8 grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-fvars-v8 : grammar?

```
    p ::= (module (define label tail) ... tail)

 pred ::= (relop loc opand)
        | (true)
        | (false)
        | (not pred)
```

```
        |  (begin effect ... pred)
        |  (if pred pred pred)

  tail ::= (jump trg)
        |  (begin effect ... tail)
        |  (if pred tail tail)

effect ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)
        |  (begin effect ... effect)
        |  (if pred effect effect)
        |  (return-point label tail)

  triv ::= opand
        |  label

 opand ::= int64
        |  loc

   trg ::= label
        |  loc

   loc ::= reg
        |  fvar

 index ::= int64
        |  loc

   reg ::= rsp
        |  rbp
        |  rax
        |  rbx
        |  rcx
        |  rdx
        |  rsi
        |  rdi
        |  r8
        |  r9
        |  r12
        |  r13
        |  r14
        |  r15

 binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right
```

```
relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 aloc ::= aloc?

 fvar ::= fvar?

label ::= label?
```

(**nested-asm-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `nested-asm-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

nested-asm-lang-v8 : grammar?

```
          p ::= (module (define label tail) ... tail)

       pred ::= (relop loc opand)
              | (true)
              | (false)
              | (not pred)
              | (begin effect ... pred)
              | (if pred pred pred)

       tail ::= (jump trg)
              | (begin effect ... tail)
              | (if pred tail tail)

     effect ::= (set! loc triv)
              | (set! loc_1 (binop loc_1 opand))
              | (set! loc_1 (mref loc_2 index))
              | (mset! loc index triv)
              | (begin effect ... effect)
              | (if pred effect effect)
              | (return-point label tail)

       triv ::= opand
              | label

      opand ::= int64
              | loc
```

```
  trg ::= label
        | loc

  loc ::= reg
        | addr

index ::= int64
        | loc

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

label ::= label?
```

(block-pred-lang-v8? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-pred-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

block-pred-lang-v8 : grammar?

```
   p ::= (module b ... b)

   b ::= (define label tail)

pred ::= (relop loc opand)
       | (true)
       | (false)
       | (not pred)

tail ::= (jump trg)
       | (begin s ... tail)
       | (if pred (jump trg) (jump trg))

   s ::= (set! loc triv)
       | (set! loc_1 (binop loc_1 opand))
       | (set! loc_1 (mref loc_2 index))
       | (mset! loc index triv)

triv ::= opand
       | label

opand ::= int64
       | loc

 trg ::= label
       | loc

 loc ::= reg
       | addr

index ::= int64
       | loc

 reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r12
       | r13
```

```
        |  r14
        |  r15

   binop ::= *
        |  +
        |  -
        |  bitwise-and
        |  bitwise-ior
        |  bitwise-xor
        |  arithmetic-shift-right

   relop ::= <
        |  <=
        |  =
        |  >=
        |  >
        |  !=

   int64 ::= int64?

   int32 ::= int32?

    addr ::= (fbp - dispoffset)

     fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?

   label ::= label?
```

(**block-asm-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `block-asm-lang-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

block-asm-lang-v8 : grammar?

```
      p ::= (module b ... b)

      b ::= (define label tail)

   tail ::= (jump trg)
        |  (begin s ... tail)
        |  (if (relop loc opand) (jump trg) (jump trg))

      s ::= (set! loc triv)
        |  (set! loc_1 (binop loc_1 opand))
        |  (set! loc_1 (mref loc_2 index))
        |  (mset! loc index triv)

   triv ::= opand
```

```
        | label

opand ::= int64
        | loc

  trg ::= label
        | loc

  loc ::= reg
        | addr

index ::= int64
        | loc

  reg ::= rsp
        | rbp
        | rax
        | rbx
        | rcx
        | rdx
        | rsi
        | rdi
        | r8
        | r9
        | r12
        | r13
        | r14
        | r15

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

dispoffset ::= dispoffset?
```

```
label ::= label?
```

(**para-asm-lang-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the para-asm-lang-v8 grammar. The first non-terminal in the grammar defines valid programs.

---

para-asm-lang-v8 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! loc triv)
              | (set! loc_1 (binop loc_1 opand))
              | (set! loc_1 (mref loc_2 index))
              | (mset! loc_1 index triv)
              | (with-label label s)
              | (jump trg)
              | (compare loc opand)
              | (jump-if relop trg)

        trg ::= label
              | loc

       triv ::= opand
              | label

      opand ::= int64
              | loc

        loc ::= reg
              | addr

      index ::= int64
              | loc

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
              | r9
              | r12
              | r13
              | r14
              | r15
```

```
 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

binop ::= *
        | +
        | -
        | bitwise-and
        | bitwise-ior
        | bitwise-xor
        | arithmetic-shift-right

relop ::= <
        | <=
        | =
        | >=
        | >
        | !=

int64 ::= int64?

int32 ::= int32?

label ::= label?

dispoffset ::= dispoffset?
```

(**paren-x64-mops-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the paren-x64-mops-v8 grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-mops-v8 : grammar?

```
   p ::= (begin s ...)

   s ::= (set! addr int32)
       | (set! addr trg)
       | (set! reg loc)
       | (set! reg triv)
       | (set! reg_1 (binop reg_1 int32))
       | (set! reg_1 (binop reg_1 loc))
       | (set! reg_1 (mref reg_2 index))
       | (mset! reg_1 index triv)
       | (with-label label s)
       | (jump trg)
       | (compare reg opand)
       | (jump-if relop label)

 trg ::= reg
```

```
       | label

 triv ::= trg
       | int64

opand ::= int64
       | reg

  loc ::= reg
       | addr

index ::= int32
       | reg

  reg ::= rsp
       | rbp
       | rax
       | rbx
       | rcx
       | rdx
       | rsi
       | rdi
       | r8
       | r9
       | r10
       | r11
       | r12
       | r13
       | r14
       | r15

 addr ::= (fbp - dispoffset)

  fbp ::= frame-base-pointer-register?

binop ::= *
       | +
       | -
       | bitwise-and
       | bitwise-ior
       | bitwise-xor
       | arithmetic-shift-right

relop ::= <
       | <=
       | =
       | >=
       | >
       | !=

int64 ::= int64?

int32 ::= int32?
```

```
      label ::= label?

  dispoffset ::= dispoffset?
```

(**paren-x64-v8?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `paren-x64-v8` grammar. The first non-terminal in the grammar defines valid programs.

---

paren-x64-v8 : grammar?

```
          p ::= (begin s ...)

          s ::= (set! addr int32)
              | (set! addr trg)
              | (set! reg loc)
              | (set! reg triv)
              | (set! reg_1 (binop reg_1 int32))
              | (set! reg_1 (binop reg_1 loc))
              | (with-label label? s)
              | (jump trg)
              | (compare reg opand)
              | (jump-if relop label)

        trg ::= reg
              | label

       triv ::= trg
              | int64

      opand ::= int64
              | reg

        loc ::= reg
              | addr

        reg ::= rsp
              | rbp
              | rax
              | rbx
              | rcx
              | rdx
              | rsi
              | rdi
              | r8
              | r9
              | r10
              | r11
              | r12
```

```
          |  r13
          |  r14
          |  r15

   addr ::= (fbp - dispoffset)
          |  (reg + int32)
          |  (reg + reg)

    fbp ::= frame-base-pointer-register?

  binop ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right

  relop ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int32 ::= int32?

  int64 ::= int64?

  label ::= label?

dispoffset ::= dispoffset?
```

4.11 First-class Procedures: Code is Data

4.11.1 Preface: What's wrong with Exprs-Lang v8?

Actually, not much. With structured data types, Exprs-lang v8 is a pretty good language now.

Exprs-bits-lang v8/contexts is sufficiently expressive to act as a reasonable compiler backend for many languages. It's roughly equivalent to C, although with more curvy parens.

Exprs-lang v8 adds safety on top of that language, although this safety does come at a cost. The main limitation in Exprs-lang v8 is the lack of abstractions over computation. We have lots of abstraction over data, but it's common to want to abstract over computation—first class functions, objects, function pointers, etc. Exprs-lang v8 prevents even passing function pointers to ensure safety.

In this chapter, we add the ability to easily abstract over computations at any point via first-class procedures. Many languages provide some version of this—Python, JavaScript, Ruby, Racket, Scheme, Java, and many more. They enable the programmer to create a suspended computation, and pass it around as a value. The procedure closes over the environment in which it was created, capturing any free variables and essentially creating an object with private fields. They can be used as the foundations for object systems, and provide a safe, lexically scoped alternative to function pointers.

In *Exprs-lang v9*, we add first-class procedures as values:

v8 Diff (excerpts) Full

```
p ::= (module b ... e)

b ::= (define x (lambda (x ...) e))

e ::= v
    | (call e e ...)
    | (let ([x e] ...) e)
    | (if e e e)
```

```
      x ::= name?
          | prim-f

      v ::= x
          | fixnum
          | #t
          | #f
          | empty
          | (void)
          | (error uint8)
          | ascii-char-literal
          | (lambda (x ...) e)+

prim-f ::= *
          | +
          | -
          | <
          | <=
          | >
          | >=
          | eq?
          | fixnum?
          | boolean?
          | empty?
          | void?
          | ascii-char?
          | error?
          | not
          | pair?
          | procedure?+
          | vector?
          | cons
          | car
          | cdr
          | make-vector
          | vector-length
          | vector-set!
          | vector-ref
          | procedure-arity+
```

Now, `lambda` can appear in any expression. We can still define procedures at the top-level using `define`, although the semantics will change slightly.

We add a new data structure to the language as well: *procedures*. These are implicitly constructed by `lambda`. They support two additional operations: `procedure?` and `procedure-arity`, for inspecting how many formal paramters they take.

This is a syntactically small change, but it has massive implications.

4.11.2 Procedures, Closures and Closure Conversion

So far, procedures in our language have been compiled directly to labeled *code*—a suspended computation that is closed execpt for it's declared parameters. We have not treated procedures as values, nor considered what happens if a procedure appears in value context. The closest representation of the value of a procedure we had was the label to its code. In earlier source languages, we disallowed passing procedures as values, to ensure safety.

4.11.2.1 The Procedure

To support first-class procedures, we need to compile procedures to a data structure. This data structure allows us to construct a procedure value, pass it around and return it, call it (safety), and captures both the procedure's code, but also any information we need about the procedure.

To add a new data structure, we need a new primary tag and a new collection of primitive operations. We use the tag `#b010` for procedures.

Here is our updated list of tags:

- `#b000`, *fixnums*, fixed-sized integers
- `#b001`, pairs
- `#b010`, procedures
- `#b011`, vectors
- `#b100`, *unused*
- `#b101`, *unused*
- `#b110`, non-fixnum immediates (booleans, etc)

- `#b111`, *unused*

In the source language, we expose the primitive operations `procedure?` and `procedure-arity`. However, the compiler intermediate languages will expose a few more operations that the compiler needs to make use of to implement procedure calls.

Every instance of `lambda` compiles to a procedure. The procedure now has three pieces of information: its arity for dynamic checking; the label to its *code*, the computation it executes when invoked; and its *environment*, the values of the free variables used in the definition of the procedure. We compile each application of a procedure to dereference and call the label of the procedure, but also to pass a reference to the procedure itself as a parameter. Essentially, the procedure is an object, and receives itself as an argument. Each "free variable" `x` is a field of that object, and are compiled to references to `self.x`.

The procedure interface is described below:

- `(make-procedure e_label e_arity e_size)`

  Creates a procedure whose label is *`e_label`*, which expects *`e_arity`* number of arguments, and has an environment of size *`e_size`*.

  `make-procedure` does not perform any error checking; it must be applied to a label and two fixnum ptrs. This is safe because no user can access `make-procedure` directly. Only the compiler generates uses of this operator, and surely our compiler uses it correctly.

  In the source language, `make-procedure` is not exposed directly; instead, `lambda` is compiled down to this primitive.

- `(unsafe-procedure-ref e_proc e_index)`

  Return the value at index *`e_index`* in the environment of the procedure *`e_proc`*.

  As with all unsafe operators, this does not perform any checking.

  In the source language, `unsafe-procedure-ref` is not exposed directly. This is used to access variables outside of the procedure's scope, but in scope at the time the procedure is created. We use this to implement closures, which we describe shortly.

- `(unsafe-procedure-set! e_proc e_index e_val)`

  Set the value at index *`e_index`* in the environment of the procedure

`e_proc` to be `e_val`.

In the source language, `unsafe-procedure-set!` is not exposed directly.

- `(unsafe-procedure-label e_proc)`

  Returns the label to the code for the procedure `e_proc`.

- `(call e_label es ...)`

  Call the code whose label is `e_label` with the arguments `es`.

  This is essentially the same as the `call` primitive in previous chapters, although we now allow labels to be computes and passed as values. It is unsafe and with no dynamic checks, so some earlier pass must insert dynamic checks to ensure safety.

Our procedure data structure is essentially a vector containing a label to the code and the values of each free variable in its environment.

The challenge in implement procedures is primarily in compiling `lambda` down to the procedure primitives, then specifying the representation of these procedure primitives in terms of calls to labelled code. All compiler passes below `specify-representation` remain unchanged.

Until now, all procedures were bound at the top-level in a set of mutually-recursive definitions. To work with first-class procedures in intermediate languages, we need to be able to represent sets of mutually recursive definitions that appear *as expressions*. We introduce the `letrec` construct to aid with this. `(letrec ([aloc e] ...) e_2)` binds each `aloc` in each `e`, including its own right-hand-side, as well as binding all `aloc`s in `e_2`. For now, we only consider a restricted form of `letrec` that only binds procedures.

4.11.2.2 The Closure

Our procedure data structure implements a *closure*, a procedure's code paired with the values of free variables from the environment in which the procedure was created. This allows us to create procedures that refer to variables outside of their own scope, but still retain references to those variables even when the procedure is passed to a different scope.

As an intermediate step in compiling first-class procedures, we

introduce explicit closure primitives which compile to the procedure primitives. There is no primary tag for this data strucure, since it will be implemented by the lower-level procedure data type.

Closures support two operations. First, you can call a closure with `(closure-call e es ...)`, which essentially extracts the label from the closure `e` and calls the procedure at that label with the argument `(es ...)`. Second, you can dereference an environment variable from the closure with `(closure-ref e e_i)`, extracting the value at index `e_i` from the environment of the closure `e`.

Because we want to implement safe procedure application, we add a third field to the closure: its *arity*, the number of arguments expected by the code of the closure.

The closure interface is described below:

- `(make-closure e_label e_arity e_i ...)`

  Creates a closure whose code is at label `e_label`, which expects `e_arity` number of arguments, and has the values `e_i` in its environment.

- `(closure-call e_c es ...)`

  Safely call the closure `e_c`, invoking its code, with the arguments `(es ...)`.

- `(closure-ref e_c e_i)`

  Deference the value at index `e_i` in the environment of the closure `e_c`. Since this dereference is only generated by the compiler, it always succeeds and performs no dynamic checks. The environment is 0-indexed.

Each of these primitives compile down to the analogous procedure primitives.

4.11.2.3 Closure Conversion

The main problem with compiling first-class procedure is that we need to lift their code to the top-level, but they have references to free variables which go out of scope if we move the procedure definition. We deal with this by converting all procedures to closures, rebinding the free variables in the code as explicit dereferences from the closure's environment, then lifting the now-closed code definitions to the top-

level. This process is called *closure conversion*.

> Closure conversion is not the only way to implement first-class procedures. An alternative that can avoid some of the allocation cost of closures is *defunctionalization*, but this does not work well with separate compilation.

Before we can perform closure conversion, we must discover which variables in a `lambda` are free with respect to the procedure's scope. We first annotation all `lambda` with their free variable sets.

```
`(lambda (,alocs ...) ,e)
=>
`(lambda ,(info-set '() 'free (set-subtract (free-var e) alocs))
         (,alocs ...) ,e)
```

We add a pass to perform this just prior to closure conversion.

A variable is considered *free* in a scope if it is not in the set of variables bound by that scope, if it is referenced in any expression in which the scope binds variables, and if the reference is not bound. A variable is *bound* if it is refernced inside a scope for which it is declared in the set of variables bound by that scope.

In our languages, `lambda`, `let`, and `letrec` introduce new scopes. Calcuating the free variables of an expression is relatively straightforward, but we have to be careful with the binding structures of `letrec` and `let`.

> Note that all variables are bound, which is enforced by `check-exprs-lang`, but they can be free relative to a particular scope.

There are two parts to closure conversion:

- Transform each `lambda`. Each `lambda` is transformed to take a new formal parameter, which is its closure, and to be bound to a *`label`* in its enclosing `letrec`. We can think of this as adding a `this` or `self` argument to each procedure.

  The abstract location to which the the `lambda` was previously bound must now be bound to a closure. The closure has `n + 2` fields, where `n` is the number of free variables in the `lambda`. The first field is the label to which the closure's code is bound. The final `n` fields are references to the lexical variables in the environment of the closure.

In essence, we transform

```
`(letrec ([,x (lambda ((free (,ys ...))) (,xs ...)  ,es)] ...)
   ,e)
=>
`(letrec ([,l (lambda (,c ,xs ...)
                (let ([,ys (closure-ref ,c ,i)] ...)
                  ,es))] ...)
    (cletrec ([,x (make-closure ,l ,(length xs) ,ys ...)] ...)
      ,e))
```

where `l` is a fresh label and `c` is a fresh abstract location. The `cletrec` form is like `letrec` but restricted to bind closures. We add the number of arguments as a field in the closure to implement safe application later.

- Transform each `call`. Every procedure now takes an extra argument, its closure, so we have to expand each `call`. The essence of the translation is:

```
`(call ,e ,es ...)
=>
`(let ([,x ,e])
   (closure-call ,x ,x ,es ...))
```

We use `closure-call` to call the (label of the) closure to the closure itself and its usual arguments. We need to bind the operator to avoid duplicating code.

We already have the low-level abstractions in place to deal with closures, so we design this assignment top-down.

4.11.3 Administrative Passes

Allowing procedures to be bound in two different ways is great for programmer convenience, but annoying for a compiler writer. Before we get to implementing procedures, we simplify and regularize how procedures appear in our language.

The first big benefit to the programmer comes in `check-exprs-lang`. Since we finally have a procedure data type, and procedure primitives to enable dynamic checking, we can finally stop type checking programs. Now, it is valid and does not cause undefined behaviour to pass procedures are arguments, return procedures, or call an arbitrary variables with an arbitrary number of arguments. The language will

dynamically check whether any of those expressions is safe before attempting to execute them

---

`(check-exprs-lang p)` → `exprs-lang-v9` procedure

`p` : `any`

Validates that input is a well-bound Exprs-lang v9 program. There are no other static restrictions.

As usual with `uniquify`, the only change is that all names *x* are replaced by abstract locations *aloc*.

Unlike in previous versions, there are no *labels* after `uniquify`. All of our procedures are data, not merely code, and cannot easily be lifted to the top level yet, so it is now the job of a later pass to introduce labels.

Below we define *Exprs-unique-lang v9*. We typeset the changes with respect to Exprs-lang v9.

v8 Diff (excerpts) Source/Target Diff (excerpts) Full

```
p ::= (module b ... e)

b ::= (define aloc+ label- (lambda (aloc ...) e))

e ::= v
    | (call e e ...)
    | (let ([aloc e] ...) e)
    | (if e e e)

v ::= label-
    | aloc
    | prim-f
    | fixnum
    | #t
    | #f
    | empty
    | (void)
    | (error uint8)
    | ascii-char-literal
    | (lambda (aloc ...) e)+

label- ::= label?
```

(**uniquify** *p*) → exprs-unique-lang-v9 procedure
p : exprs-lang-v9

Resolves top-level lexical identifiers into unique abstract locations.

Not much changes in `implement-safe-primops`.

The target language of the pass, *Exprs-unsafe-data-lang v9*, is defined below.

v8 Diff (excerpts) Source/Target Diff (excerpts) Full

```
      p ::= (module b ... e)

      b ::= (define aloc+ label- (lambda (aloc ...) e))

   pred ::= e
          | (true)
          | (false)
          | (not pred)
          | (let ([aloc e] ...) pred)
          | (if pred pred pred)

      e ::= v
          | (primop e ...)
          | (call e e ...)
          | (let ([aloc e] ...) e)
          | (if pred e e)
          | (begin effect ... e)

 effect ::= (primop e ...)
          | (begin effect ... effect)

      v ::= label-
          | aloc
          | fixnum
          | #t
          | #f
          | empty
          | (void)
          | (error uint8)
          | ascii-char-literal
          | (lambda (aloc ...) e)+
```

```
primop ::= unsafe-fx*
         | unsafe-fx+
         | unsafe-fx-
         | eq?
         | unsafe-fx<
         | unsafe-fx<=
         | unsafe-fx>
         | unsafe-fx>=
         | fixnum?
         | boolean?
         | empty?
         | void?
         | ascii-char?
         | error?
         | not
         | pair?
         | vector?
         | procedure?+
         | cons
         | unsafe-car
         | unsafe-cdr
         | unsafe-make-vector
         | unsafe-vector-length
         | unsafe-vector-set!
         | unsafe-vector-ref
         | unsafe-procedure-arity+

label- ::= label?
```

Note that this pass does not implement safe `call`, but can be safely applied to arbitrary data—a later pass will implement dynamic checking for application.

(**implement-safe-primops** *p*) → exprs-unsafe-data-lang-v9?
p : exprs-unique-lang-v9?

Implement safe primitive procedures by inserting procedure definitions for each primitive operation which perform dynamic tag checking, to ensure type and memory safety.

Now we implement `call` in terms of `unsafe-procedure-call` and `unsafe-procedure-arity`. Note that we cannot simply define `call` as a procedure, like we did with other safe wrappers, since it must count its

arguments, and we must support a variable number of arguments to the procedure.

Below we define *Exprs-unsafe-lang v9*.

Source/Target Diff (excerpts) Full

```
          p ::= (module b ... e)

          b ::= (define aloc (lambda (aloc ...) e))

       pred ::= e
              | (true)
              | (false)
              | (not pred)
              | (let ([aloc e] ...) pred)
              | (if pred pred pred)

          e ::= v
              | (primop e ...)
              | (unsafe-procedure-
                 call+ call- e e ...)
              | (let ([aloc e] ...) e)
              | (if pred e e)
              | (begin effect ... e)

     effect ::= (primop e ...)
              | (begin effect ... effect)

          v ::= aloc
              | fixnum
              | #t
              | #f
              | empty
              | (void)
              | (error uint8)
              | ascii-char-literal
              | (lambda (aloc ...) e)

     primop ::= unsafe-fx*
              | unsafe-fx+
              | unsafe-fx-
              | eq?
              | unsafe-fx<
              | unsafe-fx<=
              | unsafe-fx>
```

```
                  |  unsafe-fx>=
                  |  fixnum?
                  |  boolean?
                  |  empty?
                  |  void?
                  |  ascii-char?
                  |  error?
                  |  not
                  |  pair?
                  |  vector?
                  |  procedure?
                  |  cons
                  |  unsafe-car
                  |  unsafe-cdr
                  |  unsafe-make-vector
                  |  unsafe-vector-length
                  |  unsafe-vector-set!
                  |  unsafe-vector-ref
                  |  unsafe-procedure-arity

          aloc ::= aloc?

        fixnum ::= int61?

         uint8 ::= uint8?

    ascii-char- ::= ascii-char-literal?
        literal
```

We implement `call` in terms of `procedure?`, `unsafe-procedure-call` and `unsafe-procedure-arity`. The essence of the transformation is:

```
`(call ,e ,es ...)
=>
`(if (procedure? ,e)
     (if (eq? (unsafe-procedure-arity ,e) ,(length es))
         (unsafe-procedure-call ,e ,es ...)
         ,bad-arity-error)
     ,bad-proc-error)
```

> If you keep track of the arity of procedures, you can optimize this transformation in some cases.

```
(implement-safe-call p) → exprs-unsafe-lang-v9?  procedure
  p : exprs-unsafe-data-lang-v9?
```

Implement `call` as an unsafe procedure call with dynamic checks.

Some procedures now appear in local expressions, and some appear defined at the top-level. This presents two problems. First, our compiler later assumes that all *data* (as opposed to code) is *locally* defined—we have no way to define top-level, labelled data. Since procedures are data, we need to transform top-level bindings of procedures into local bindings, so the rest of the compiler will "just work". Second, we have to look for procedures in two different places to transform them: that's annoying.

To deal with these, we introduce two small administrative passes: `define->letrec` and `dox-lambdas`.

First, in `define->letrec`, we elaborate `define` into a local binding form `letrec`, which will be used to bind all procedures.

`letrec`, unlike `let`, supports multiple bindings in a single form, and each bound expression can refer to any variable in the set of bindings for the `letrec`. This is important to capture mutually-recursive functions, and has the same binding structure as our top-level `defines`.

> *Design digression:*
>
> In general, a language might impose additional semantics on `define`, such as allowing `defined` data to be exported and imported at module boundaries. This would require additional handling of `define`, and the ability to generate labelled data in the back-end of the compiler. We continue to ignore separate compilation and linking, so we treat `define` as syntactic sugar for `letrec`.

Below we define *Just-Exprs-lang v9*.

Source/Target Diff (excerpts) Full

```
      p ::= (module b⁻ ...⁻ e)

     b⁻ ::= (define aloc (lambda (aloc ...) e))

   pred ::= e
          | (true)
          | (false)
```

```
        |  (not pred)
        |  (let ([aloc e] ...) pred)
        |  (if pred pred pred)

    e ::= v
        |  (primop e ...)
        |  (unsafe-procedure-call e e ...)
        |  (letrec ([aloc (lambda (aloc ...) e)] ...)
           e)+
        |  (let ([aloc e] ...) e)
        |  (if pred e e)
        |  (begin effect ... e)

effect ::= (primop e ...)
        |  (begin effect ... effect)

    v ::= aloc
        |  fixnum
        |  #t
        |  #f
        |  empty
        |  (void)
        |  (error uint8)
        |  ascii-char-literal
        |  (lambda (aloc ...) e)

primop ::= unsafe-fx*
        |  unsafe-fx+
        |  unsafe-fx-
        |  eq?
        |  unsafe-fx<
        |  unsafe-fx<=
        |  unsafe-fx>
        |  unsafe-fx>=
        |  fixnum?
        |  boolean?
        |  empty?
        |  void?
        |  ascii-char?
        |  error?
        |  not
        |  pair?
        |  vector?
        |  procedure?
```

```
  | cons
  | unsafe-car
  | unsafe-cdr
  | unsafe-make-vector
  | unsafe-vector-length
  | unsafe-vector-set!
  | unsafe-vector-ref
  | unsafe-procedure-arity
```

(**define->letrec** *p*) → just-exprs-lang-v9? procedure
p : exprs-unsafe-lang-v9?

Explicitly binds all procedures to abstract locations.

Before we start compiling `lambda`s, we should try to get rid of them. *Direct calls* to `lambda`s, such as `(call (lambda (x) x) 1)`, are simple to rewrite to a `let` binding, such a `(let ([x 1]) x)`. A human programmer may not write this kind of code much, but most programs are not written by humans—compilers write far more programs. This optimization will speed-up compile time and run time for such simple programs.

> *Design digression:*
>
> This optimization is a special case of procedure inlining. A direct call to a procedure value guarantees the procedure occurs exactly once, and is therefore completely safe to inline. This pass could be replaced by a more general inlining optimization.

(**optimize-direct-calls** *p*) → just-exprs-lang-v9? procedure
p : just-exprs-lang-v9?

Inline all direct calls to first-class procedures.

Next, we explicitly name all lambdas to names in `dox-lambdas`. The source language supports anonymous procedures, that is, first-class procedure values that are not necessarily bound to names. For example, we can write the following in Racket, creating and using procedures without ever binding them to names in a *`letrec`* or *`let`* form.

Example:

```
> ((lambda (x f) (f x x)) 1 (lambda (x y) (+ x y)))
2
```

The equivalent in Exprs-lang v9 is:

```
(call (lambda (x f) (call f x x)) 1 (lambda (x y) (call + x y)))
```

This is great for functional programmers, who value freedom, but bad for compilers who want to keep track of everything.

We bind all procedures to names to simplify lifting code to the top-level and assigning labels later.

We transform each `` `(lambda (,alocs ...) ,e) `` into `` `(letrec ([,tmp (lambda (,alocs ...) ,e)]) ,tmp) ``, where `tmp` is a fresh *aloc*.

We define *Lam-opticon-lang v9*, in which we know the name of every procedure.

Source/Target Diff (excerpts) Full

```
        p ::= (module e)

     pred ::= e
            | (true)
            | (false)
            | (not pred)
            | (let ([aloc e] ...) pred)
            | (if pred pred pred)

        e ::= v
            | (primop e ...)
            | (unsafe-procedure-call e e ...)
            | (letrec ([aloc (lambda (aloc ...) e)] ...) e)
            | (let ([aloc e] ...) e)
            | (if pred e e)
            | (begin effect ... e)

   effect ::= (primop e ...)
            | (begin effect ... effect)

        v ::= aloc
            | fixnum
            | #t
            | #f
            | empty
            | (void)
```

```
          |  (error uint8)
          |  ascii-char-literal
          |  (lambda (aloc ...) e)¯

primop ::= unsafe-fx*
          |  unsafe-fx+
          |  unsafe-fx-
          |  eq?
          |  unsafe-fx<
          |  unsafe-fx<=
          |  unsafe-fx>
          |  unsafe-fx>=
          |  fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
          |  pair?
          |  vector?
          |  procedure?
          |  cons
          |  unsafe-car
          |  unsafe-cdr
          |  unsafe-make-vector
          |  unsafe-vector-length
          |  unsafe-vector-set!
          |  unsafe-vector-ref
          |  unsafe-procedure-arity

  aloc ::= aloc?

fixnum ::= int61?

 uint8 ::= uint8?

ascii- ::= ascii-char-literal?
 char-
literal
```

(**dox-lambdas** *p*) → lam-opticon-lang-v9? procedure

p : just-exprs-lang-v9?

Explicitly binds all procedures to abstract locations.

4.11.4 Closure Conversion

The rest of our compiler expects procedures to be little more than labeled blocks of code. Unfortunately, now our procedures can contain references to free-variables in their lexical scope. This means we cannot simply lift procedure definitions to the top-level, stick on a label, and generate a labelled procedure.

First, we uncover the free variables in each `lambda`. We add these as an annotation on the `lambda`, which the next pass will use to generate closures.

Below we define *Lambda-free-lang v9*.

Source/Target Diff (excerpts) Full

```
      p ::= (module e)

  info+ ::= ((free (aloc ...)) any ...)

      e ::= v
          | (primop e ...)
          | (unsafe-procedure-call e e ...)
          | (letrec ([aloc (lambda info+ (aloc ...) e)] ...) e)
          | (let ([aloc e] ...) e)
          | (if pred e e)
          | (begin effect ... e)

      v ::= aloc
          | fixnum
          | #t
          | #f
          | empty
          | (void)
          | (error uint8)
          | ascii-char-literal
```

To find the free abstract locations, we traverse the body of each `lambda` remembering any abstract locations that have been bound (by `let`, `lambda`, or `letrec`), and return the set of abstract locations that have been used but were not in the defined set. On entry to the `(lambda (aloc ...) e)`, only the formal parameters `(aloc ...)` are considered

bound.

(**uncover-free** *p*) → lam-free-lang-v9? procedure
  p : lam-opticon-lang-v9?

Explicitly annotate procedures with their free variable sets.

The only complicated case is for `letrec`. Even a variable bound in a `letrec` is considered free in the body of a `lambda`.

Example:

```
> (uncover-free
    `(module
       (letrec ([x.1 (lambda () (unsafe-procedure-call x.1))])
         x.1)))
'(module
   (letrec ((x.1 (lambda ((free (x.1))) () (unsafe-procedure-call x.1)))) x.1))
```

However, the `letrec` does bind those variables, so they do not contribute to the free variable set for the context surrounding the `letrec`.

Example:

```
> (uncover-free
    `(module
       (letrec ([f.1 (lambda ()
                       (letrec ([x.1 (lambda () (unsafe-procedure-call x.1))])
                         x.1))])
         f.1)))
'(module
   (letrec ((f.1
             (lambda ((free ()))
               ()
               (letrec ((x.1
                         (lambda ((free (x.1)))
                           ()
                           (unsafe-procedure-call x.1))))
                 x.1))))
     f.1))
```

Now, we make closures explicit.

Strictly speaking, all the previous languages had *closures*—procedures that (implicitly) close over their lexical environment. However, our

earlier languages forbid us from ever creating procedures that had a non-empty set environment, so all our closures were trivial to compile to labelled code. Closure conversion is the process of compiling first-class procedures into an explicit data type.

Below, we define *Closure-lang v9*.

Source/Target Diff (excerpts) Full

```
        p ::= (module e)

   info⁻ ::= ((free (aloc ...)) any ...)

        e ::= v
            | (primop e ...)
            | (closure-ref e e)⁺
            | (closure-call e e ...)⁺
            | (call⁺ unsafe-procedure-call⁻ e e ...)
            | (letrec ([label⁺ aloc⁻ (lambda info⁻ (aloc ...) e)] ...) e)
            | (cletrec ([aloc (make-closure label e ...)] ...) e)⁺
            | (let ([aloc e] ...) e)
            | (if pred e e)
            | (begin effect ... e)

        v ::= label⁺
            | aloc
            | fixnum
            | #t
            | #f
            | empty
            | (void)
            | (error uint8)
            | ascii-char-literal

  label⁺ ::= label?
```

Closure conversion changes `letrec` to bind labels to procedure code. After this pass, the body of `lambda` will not contain any free variables, and will not be a procedure data type—it is just like a procedure from Values-lang v6.

To encode closures, we temporarily add a new data type for closures, which we compile to a lower-level data structure later. We add a new form, `cletrec`, which only binds closures. Closures can, in general,

have recursive self-references, so this is a variant of the `letrec` form. We also add a new form for dereferencing the value of lexical variables from the closure `(closure-ref e e)`. The next pass implements closures using the procedure data type.

We assume that the `cletrec` form only ever appears as the body of a `letrec` form, but we do not make this explicit in the syntax for readability. This assumption is not necessary for correctness, but simplifies an optimization presented later as a challenge exercise.

We add `call`, the primtive operation for calling a label directly, to enable optimizing closures, an important optimization in functional languages.

`(convert-closures p)` → `closure-lang-v9?` procedure
`p` : `lam-free-lang-v9?`

Performs closure conversion, converting all procedures into explicit closures.

> If the operator is already a *aloc*, avoid introducing an extra `let`:
>
> ```
> `(unsafe-procedure-call ,aloc ,es ...)
> =>
> `(closure-call ,aloc ,aloc ,es ...)
> ```
>
> This also simplifies the optimization `optimize-known-calls`.

Closures can cause a lot of indirection, and thus performance penalty, in a functional language. We essentially transform all calls into *indirect calls*. This causes an extra memory dereference and indirect jump, both of which can have performance penalties.

Many calls, particularly to named functions, can be optimized to direct calls. We essentially perform the following transformation on all calls where we can determine the label of the operator:

```
`(closure-call ,e ,es ...)
=>
`(call ,l ,es ...)
```

where `l` is known to be the label of the closure `e`. Because `e` is already an *aloc*, we can safely discard it; we do not need to force evaluation to preserve any side effects.

We perform this optimization by recognizing `letrec` and `cletrec` as a single composite form:

```
`(letrec ([,label_l ,lam])
   (cletrec ([,aloc_c (make-closure ,label_c ,es ...)])
     ,e))
```

All uses of `(closure-call aloc_c es ...)` in `e` and `lam` can be transformed into `(call label_c es ...)`. We have to recognize these as a single composite form to optimize recursive calls inside `lam`, which will benefit the most from the optimization. This relies on the name *aloc_c* binding in two places: once to define the closure, and once when dereferenced in a recursive closure.

(optimize-known-calls *p*) → closure-lang-v9? procedure
p : closure-lang-v9?

Optimizes calls to known closures.

Now that all `lambda`s are closed and labelled, we can lift them to top-level `define`s.

We define *Hoisted-lang v9* below.

Source/Target Diff (excerpts) Full

```
   p ::= (module b+ ...+ e)

  b+ ::= (define label (lambda (aloc ...) e))

   e ::= v
       | (primop e ...)
       | (closure-ref e e)
       | (closure-call e e ...)
       | (call e e ...)
       | (letrec ([label (lambda (aloc ...) e)] ...) e)
       | (cletrec ([aloc (make-
         closure label e ...)] ...) e)
       | (let ([aloc e] ...) e)
       | (if pred e e)
       | (begin effect ... e)
```

The only difference is the `letrec` is remove and `define` blocks are re-added.

(**hoist-lambdas** *p*) → hoisted-lang-v9? procedure
  p : closure-lang-v9?

Hoists code to the top-level definitions.

Now we implement closures as the procedure data structure.

A *procedure* is a data structure representing a value that can be called. Essentially, it is a wrapper around labels so we can check applications. We construct a procedure using `(make-procedure e_1 e_2)`, where *e_1* must evaluate to a label and *e_2* is the number of expected arguments. The predicate `procedure?` should return `#t` for any value constructed this way, and #f for any other value—`(eq? (procedure? (make-procedure e_1 e_2)) #t)`.

We extract the label of a procedure with `(unsafe-procedure-label e_1)`, where *e_1* is a procedure. We get the arity of a procedure with `(unsafe-procedure-arity e_1)`, where *e_1* is a procedure.

A procedure looks like an extension of a vector. It has at least three fields: the label, the arity, and a size. The size indicates how large the environment of the procedure is. The environment will be uninitialized after `make-procedure`, and instead the environment will be initialized manually using `unsafe-procedure-set!`, similar to vector initialization. As with closures, `unsafe-procedure-label` dereference the label and `unsafe-procedure-ref` dereferences a value from the procedure's environment, given an index into the environment. However, we also have `unsafe-procedure-arity` to dereference the arity of a procedure.

The language *Proc-exposed-lang v9* is defined below.

Source/Target Diff (excerpts) Full

```
p ::= (module b ... e)

b ::= (define label (lambda (aloc ...) e))

e ::= v
    | (primop e ...)
    | (closure-ref e e)⁻
    | (closure-call e e ...)⁻
    | (call e e ...)
    | (cletrec ([aloc (make-closure label e ...)]
      ...) e)⁻
```

```
        |  (let ([aloc e] ...) e)
        |  (if pred e e)
        |  (begin effect ... e)

    v ::= label
        |  aloc
        |  fixnum
        |  #t
        |  #f
        |  empty
        |  (void)
        |  (error uint8)
        |  ascii-char-literal

primop ::= unsafe-fx*
        |  unsafe-fx+
        |  unsafe-fx-
        |  eq?
        |  unsafe-fx<
        |  unsafe-fx<=
        |  unsafe-fx>
        |  unsafe-fx>=
        |  fixnum?
        |  boolean?
        |  empty?
        |  void?
        |  ascii-char?
        |  error?
        |  not
        |  pair?
        |  vector?
        |  procedure?
        |  cons
        |  unsafe-car
        |  unsafe-cdr
        |  unsafe-make-vector
        |  unsafe-vector-length
        |  unsafe-vector-set!
        |  unsafe-vector-ref
        |  make-procedure+
        |  unsafe-procedure-arity
        |  unsafe-procedure-label+
        |  unsafe-procedure-ref+
```

```
        |  unsafe-procedure-set!+
label ::= label?
```

To transform closures into procedures, we do a three simple translations:

- Transform `make-closure`

```
`(cletrec ([,aloc (make-closure ,label ,arity ,es ...)] ...)
   ,e)
=>
`(let ([,aloc (make-procedure ,label ,arity ,n)] ...)
   (begin
     (unsafe-procedure-set! ,aloc 0 ,(list-ref es 0))
     ...
     (unsafe-procedure-set! ,aloc ,n ,(list-ref es n))
     ,e))
```

  where `n` is `(length es)`, the number of values in the environment.

- Transform `closure-ref`.

```
`(closure-ref ,c ,i)
=>
`(unsafe-procedure-ref ,c ,i)
```

  We can use `unsafe-procedure-ref` since we generate all uses of `closure-ref`.

- Transform `closure-call`.

```
`(closure-call ,c ,es ...)
=>
`(call (unsafe-procedure-label ,c) ,es ...)
```

  Recall that `closure-call` is generated from an `unsafe-procedure-call`, so it is equally safe to call `unsafe-procedure-label`.

(implement-closures *p*) → proc-exposed-lang-v9 procedure
  p : hoisted-lang-v9

Implement closures in terms of the procedure data structure.

Finally, we need to implement procedure data type. It is intentionally designed to be similar to the vector data type.

The target language is Exprs-bits-lang v8/contexts, which is unchanged from the previous chapter.

Source/Target Diff (excerpts) Full

```
        p ::= (module b ... tail+ e−)

        b ::= (define label (lambda (aloc ...) tail+ e−))

     pred ::= (relop value value)+
            | e−
            | (true)
            | (false)
            | (not pred)
            | (let ([aloc value+ e−] ...) pred)
            | (if pred pred pred)
            | (begin effect ... pred)+

    tail+ ::= value+
            | e−
            | v−
            | (primop e ...)−
            | (call e e ...)−
            | (let ([aloc value+ e−] ...) tail+ e−)
            | (if pred tail+ tail+ e− e−)
            | (begin effect ... tail+ e−)

  value+ ::= triv
            | (binop value value)
            | (mref value value)
            | (alloc value)
            | (call value value ...)
            | (let ([aloc value]) value)
            | (if pred value value)
            | (begin effect ... value)

  effect ::= (mset! value value value)+
            | (primop e ...)−
            | (begin effect ... effect)

    triv+ ::= label
            | aloc
```

```
          |  int64

binop+ ::= *
          |  +
          |  -
          |  bitwise-and
          |  bitwise-ior
          |  bitwise-xor
          |  arithmetic-shift-right

     v- ::= label
          |  aloc
          |  fixnum
          |  #t
          |  #f
          |  empty
          |  (void)
          |  (error uint8)
          |  ascii-char-literal

primop- ::= unsafe-fx*
          |  unsafe-fx+
          |  unsafe-fx-
          |  eq?
          |  unsafe-fx<
          |  unsafe-fx<=
          |  unsafe-fx>
          |  unsafe-fx>=
          |  fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
          |  pair?
          |  vector?
          |  procedure?
          |  cons
          |  unsafe-car
          |  unsafe-cdr
          |  unsafe-make-vector
          |  unsafe-vector-length
```

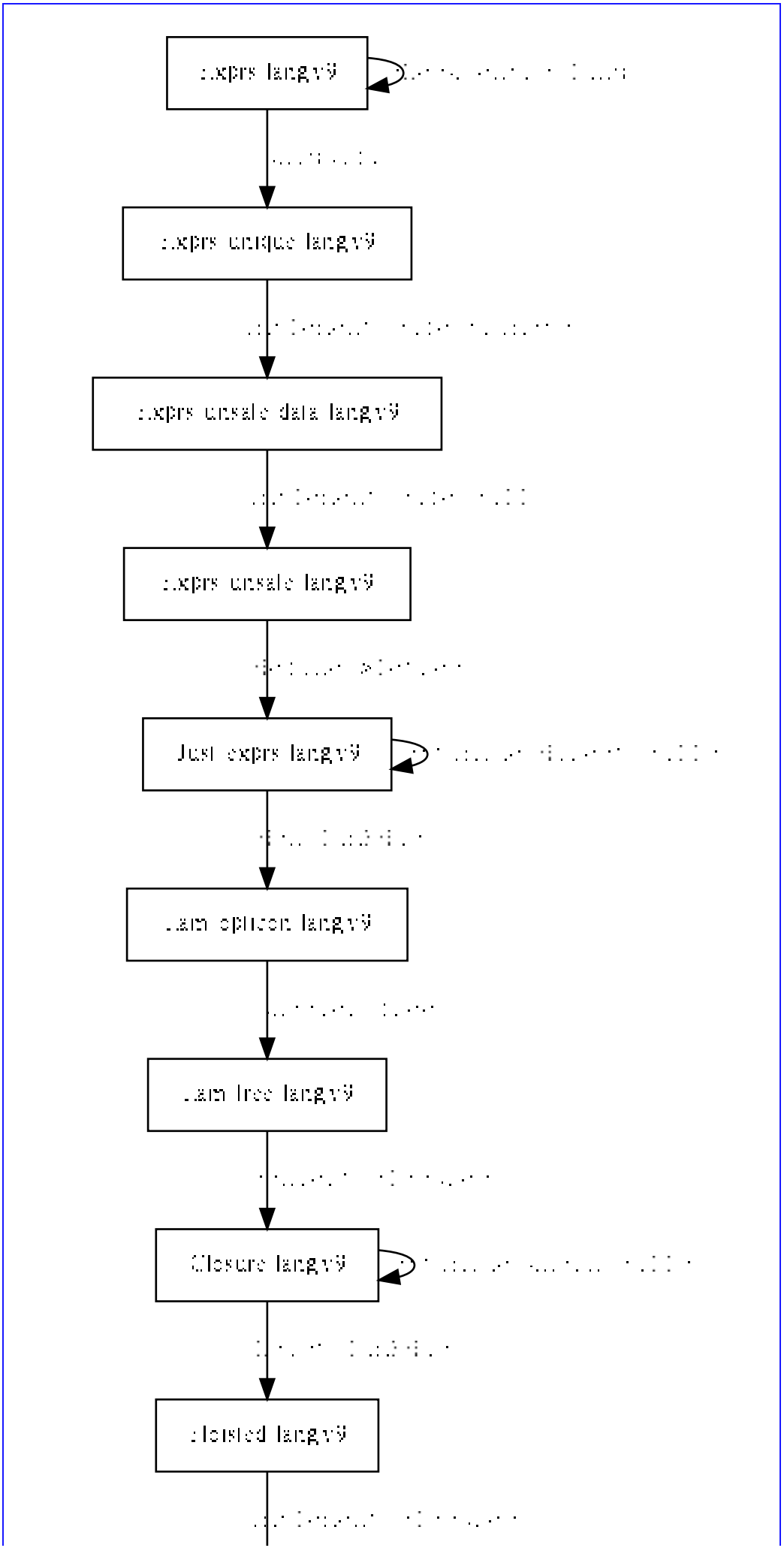
```
          |  unsafe-vector-set!
          |  unsafe-vector-ref
          |  make-procedure
          |  unsafe-procedure-arity
          |  unsafe-procedure-label
          |  unsafe-procedure-ref
          |  unsafe-procedure-set!

    aloc ::= aloc?

   label ::= label?

  relop+ ::= <
          |  <=
          |  =
          |  >=
          |  >
          |  !=

  int64+ ::= int64?

 fixnum- ::= int61?

  uint8- ::= uint8?

  ascii-  ::= ascii-char-literal?
   char-
literal-
```

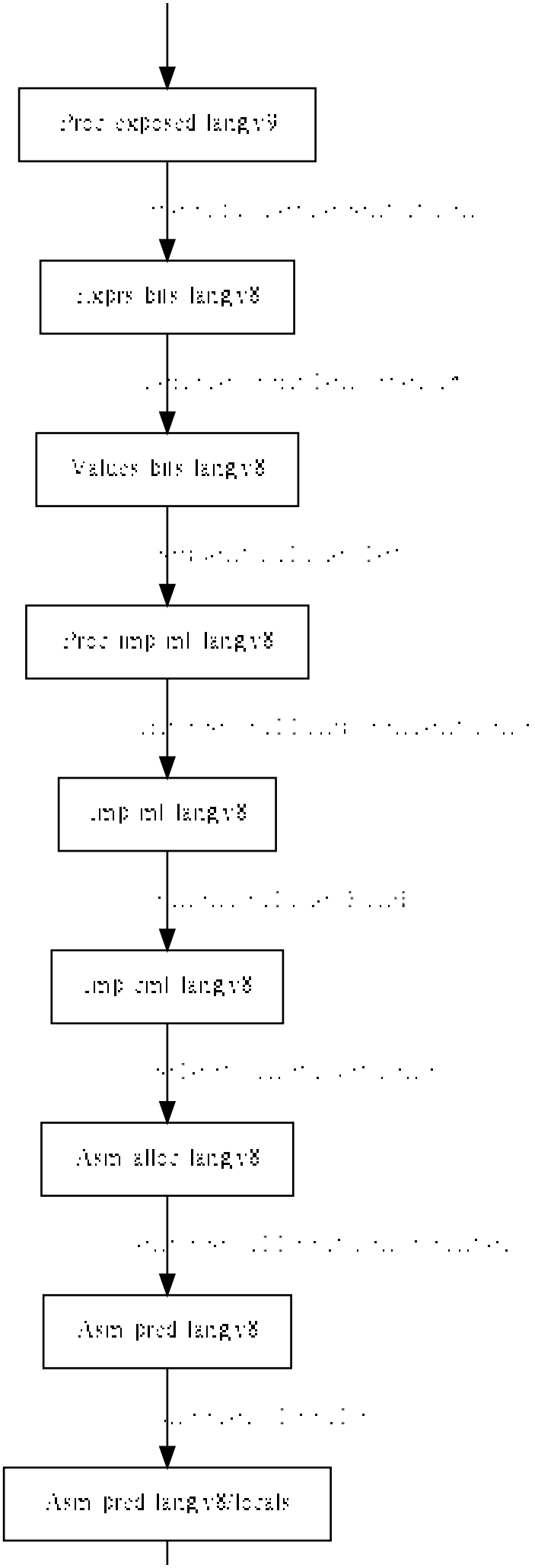
When implementing `make-procedure`, we assume the size of the environment is a fixnum constant, since this is guaranteed by how our compiler generates `make-procedure`

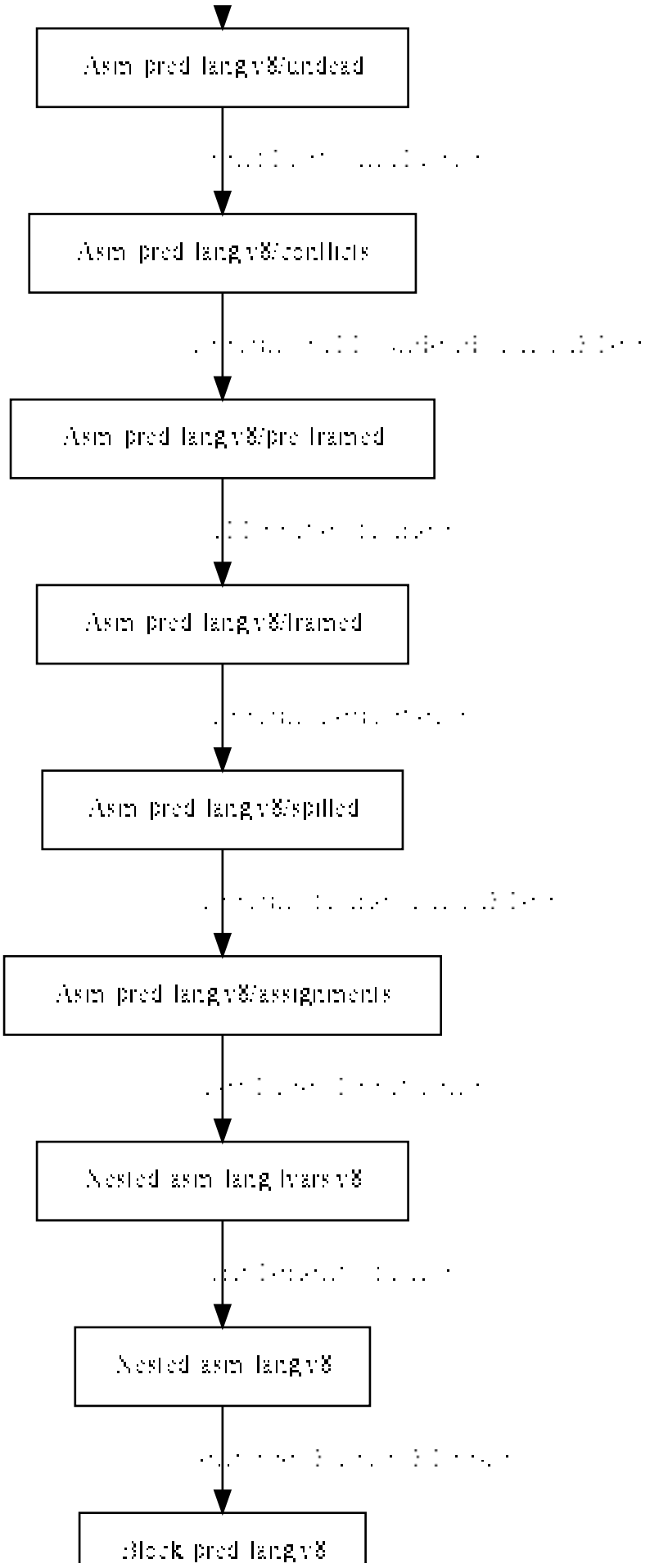
(**specify-representation** *p*) → exprs-bits-lang-v8/procedure?
p : proc-exposed-lang-v9?

Compiles data types and primitive operations into their implementations as ptrs and primitive bitwise operations on ptrs.

4.11.5 Appendix: Overview


Expr lang v9
Expr unique lang v9
Expr unsafe data lang v9
Expr unsafe lang v9
Just expr lang v9
Jam option lang v9
Jam free lang v9
Closure lang v9
Closed lang v9


Proc exposed lang-v8
Expr bits lang-v8
Values bits lang-v8
Proc imp ml lang-v8
Imp ml lang-v8
Imp cml lang-v8
Asm alloc lang-v8
Asm pred lang-v8
Asm pred lang-v8/locals


Asm pred lang:v8/undead
Asm pred lang:v8/conflicts
Asm pred lang:v8/pre framed
Asm pred lang:v8/framed
Asm pred lang:v8/spilled
Asm pred lang:v8/assignments
Nested asm lang:vars:v8
Nested asm lang:v8
Block pred lang:v8

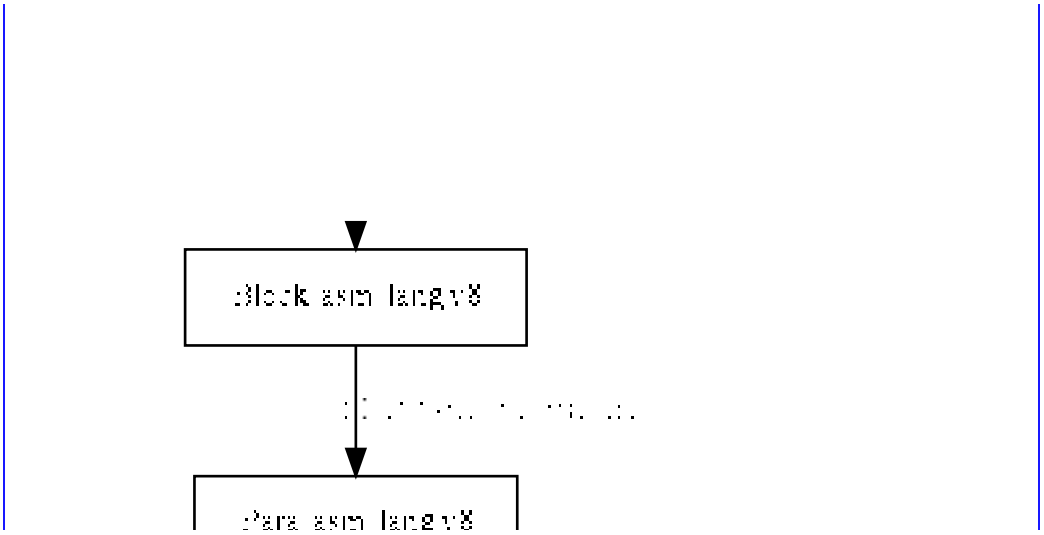


Figure 9: Overview of Compiler Version 0

4.11.6 Appendix: Languages

(**exprs-lang-v9?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `exprs-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

exprs-lang-v9 : grammar?

p ::= (module *b* ... *e*)

b ::= (define *x* (lambda (*x* ...) *e*))

```
e ::= v
    | (call e e ...)
    | (let ([x e] ...) e)
    | (if e e e)

x ::= name?
    | prim-f

v ::= x
    | fixnum
    | #t
    | #f
    | empty
    | (void)
    | (error uint8)
    | ascii-char-literal
    | (lambda (x ...) e)

prim-f ::= *
         | +
         | -
         | <
         | <=
         | >
         | >=
         | eq?
         | fixnum?
         | boolean?
         | empty?
         | void?
         | ascii-char?
         | error?
         | not
         | pair?
         | procedure?
         | vector?
         | cons
         | car
         | cdr
         | make-vector
         | vector-length
         | vector-set!
         | vector-ref
```

```
                  |  procedure-arity

          fixnum ::= int61?

           uint8 ::= uint8?

  ascii-char-literal ::= ascii-char-literal?
```

(**exprs-unique-lang-v9?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unique-lang-v9 grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-unique-lang-v9 : grammar?

```
                p ::= (module b ... e)

                b ::= (define aloc (lambda (aloc ...) e))

                e ::= v
                   |  (call e e ...)
                   |  (let ([aloc e] ...) e)
                   |  (if e e e)

                v ::= aloc
                   |  prim-f
                   |  fixnum
                   |  #t
                   |  #f
                   |  empty
                   |  (void)
                   |  (error uint8)
                   |  ascii-char-literal
                   |  (lambda (aloc ...) e)

          prim-f ::= *
                   |  +
                   |  -
                   |  eq?
                   |  <
                   |  <=
                   |  >
```

```
                  |  >=
                  |  fixnum?
                  |  boolean?
                  |  empty?
                  |  void?
                  |  ascii-char?
                  |  error?
                  |  not
                  |  pair?
                  |  procedure?
                  |  vector?
                  |  cons
                  |  car
                  |  cdr
                  |  make-vector
                  |  vector-length
                  |  vector-set!
                  |  vector-ref
                  |  procedure-arity

          aloc ::= aloc?

        fixnum ::= int61?

         uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(exprs-unsafe-data-lang-v9? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unsafe-data-lang-v9 grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-unsafe-data-lang-v9 : grammar?

```
            p ::= (module b ... e)

            b ::= (define aloc (lambda (aloc ...) e))

         pred ::= e
                  |  (true)
```

```
          |  (false)
          |  (not pred)
          |  (let ([aloc e] ...) pred)
          |  (if pred pred pred)

     e ::= v
          |  (primop e ...)
          |  (call e e ...)
          |  (let ([aloc e] ...) e)
          |  (if pred e e)
          |  (begin effect ... e)

effect ::= (primop e ...)
          |  (begin effect ... effect)

     v ::= aloc
          |  fixnum
          |  #t
          |  #f
          |  empty
          |  (void)
          |  (error uint8)
          |  ascii-char-literal
          |  (lambda (aloc ...) e)

primop ::= unsafe-fx*
          |  unsafe-fx+
          |  unsafe-fx-
          |  eq?
          |  unsafe-fx<
          |  unsafe-fx<=
          |  unsafe-fx>
          |  unsafe-fx>=
          |  fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
          |  pair?
          |  vector?
          |  procedure?
          |  cons
```

```
                  |  unsafe-car
                  |  unsafe-cdr
                  |  unsafe-make-vector
                  |  unsafe-vector-length
                  |  unsafe-vector-set!
                  |  unsafe-vector-ref
                  |  unsafe-procedure-arity

            aloc ::= aloc?

          fixnum ::= int61?

           uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```

(exprs-unsafe-lang-v9? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the exprs-unsafe-lang-v9 grammar. The first non-terminal in the grammar defines valid programs.

---

exprs-unsafe-lang-v9 : grammar?

```
                p ::= (module b ... e)

                b ::= (define aloc (lambda (aloc ...) e))

             pred ::= e
                  |  (true)
                  |  (false)
                  |  (not pred)
                  |  (let ([aloc e] ...) pred)
                  |  (if pred pred pred)

                e ::= v
                  |  (primop e ...)
                  |  (unsafe-procedure-call e e ...)
                  |  (let ([aloc e] ...) e)
                  |  (if pred e e)
                  |  (begin effect ... e)

           effect ::= (primop e ...)
```

```
          |  (begin effect ... effect)

      v ::= aloc
          |  fixnum
          |  #t
          |  #f
          |  empty
          |  (void)
          |  (error uint8)
          |  ascii-char-literal
          |  (lambda (aloc ...) e)

 primop ::= unsafe-fx*
          |  unsafe-fx+
          |  unsafe-fx-
          |  eq?
          |  unsafe-fx<
          |  unsafe-fx<=
          |  unsafe-fx>
          |  unsafe-fx>=
          |  fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
          |  pair?
          |  vector?
          |  procedure?
          |  cons
          |  unsafe-car
          |  unsafe-cdr
          |  unsafe-make-vector
          |  unsafe-vector-length
          |  unsafe-vector-set!
          |  unsafe-vector-ref
          |  unsafe-procedure-arity

   aloc ::= aloc?

 fixnum ::= int61?

  uint8 ::= uint8?
```

```
ascii-char-literal ::= ascii-char-literal?
```

(just-exprs-lang-v9? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `just-exprs-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

---

just-exprs-lang-v9 : grammar?

```
          p ::= (module e)

        pred ::= e
               | (true)
               | (false)
               | (not pred)
               | (let ([aloc e] ...) pred)
               | (if pred pred pred)

          e ::= v
               | (primop e ...)
               | (unsafe-procedure-call e e ...)
               | (letrec ([aloc (lambda (aloc ...) e)] ...) e)
               | (let ([aloc e] ...) e)
               | (if pred e e)
               | (begin effect ... e)

     effect ::= (primop e ...)
               | (begin effect ... effect)

          v ::= aloc
               | fixnum
               | #t
               | #f
               | empty
               | (void)
               | (error uint8)
               | ascii-char-literal
               | (lambda (aloc ...) e)

     primop ::= unsafe-fx*
```

```
          |  unsafe-fx+
          |  unsafe-fx-
          |  eq?
          |  unsafe-fx<
          |  unsafe-fx<=
          |  unsafe-fx>
          |  unsafe-fx>=
          |  fixnum?
          |  boolean?
          |  empty?
          |  void?
          |  ascii-char?
          |  error?
          |  not
          |  pair?
          |  vector?
          |  procedure?
          |  cons
          |  unsafe-car
          |  unsafe-cdr
          |  unsafe-make-vector
          |  unsafe-vector-length
          |  unsafe-vector-set!
          |  unsafe-vector-ref
          |  unsafe-procedure-arity

    aloc ::= aloc?

  fixnum ::= int61?

   uint8 ::= uint8?

  ascii-  ::= ascii-char-literal?
   char-
 literal
```

(**lam-opticon-lang-v9?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the lam-opticon-lang-v9 grammar. The first non-terminal in the grammar defines valid programs.

lam-opticon-lang-v9 : grammar?

```
          p ::= (module e)

       pred ::= e
              | (true)
              | (false)
              | (not pred)
              | (let ([aloc e] ...) pred)
              | (if pred pred pred)

          e ::= v
              | (primop e ...)
              | (unsafe-procedure-call e e ...)
              | (letrec ([aloc (lambda (aloc ...) e)] ...) e)
              | (let ([aloc e] ...) e)
              | (if pred e e)
              | (begin effect ... e)

     effect ::= (primop e ...)
              | (begin effect ... effect)

          v ::= aloc
              | fixnum
              | #t
              | #f
              | empty
              | (void)
              | (error uint8)
              | ascii-char-literal

     primop ::= unsafe-fx*
              | unsafe-fx+
              | unsafe-fx-
              | eq?
              | unsafe-fx<
              | unsafe-fx<=
              | unsafe-fx>
              | unsafe-fx>=
              | fixnum?
              | boolean?
              | empty?
              | void?
              | ascii-char?
```

```
            |  error?
            |  not
            |  pair?
            |  vector?
            |  procedure?
            |  cons
            |  unsafe-car
            |  unsafe-cdr
            |  unsafe-make-vector
            |  unsafe-vector-length
            |  unsafe-vector-set!
            |  unsafe-vector-ref
            |  unsafe-procedure-arity

      aloc ::= aloc?

    fixnum ::= int61?

     uint8 ::= uint8?

    ascii-  ::= ascii-char-literal?
     char-
   literal
```

(lam-free-lang-v9? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `lam-free-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

---

lam-free-lang-v9 : grammar?

```
     p ::= (module e)

  info ::= ((free (aloc ...)) any ...)

  pred ::= e
         |  (true)
         |  (false)
         |  (not pred)
         |  (let ([aloc e] ...) pred)
         |  (if pred pred pred)
```

```
     e ::= v
         | (primop e ...)
         | (unsafe-procedure-call e e ...)
         | (letrec ([aloc (lambda info (aloc ...) e)] ...) e)
         | (let ([aloc e] ...) e)
         | (if pred e e)
         | (begin effect ... e)

effect ::= (primop e ...)
         | (begin effect ... effect)

     v ::= aloc
         | fixnum
         | #t
         | #f
         | empty
         | (void)
         | (error uint8)
         | ascii-char-literal

primop ::= unsafe-fx*
         | unsafe-fx+
         | unsafe-fx-
         | eq?
         | unsafe-fx<
         | unsafe-fx<=
         | unsafe-fx>
         | unsafe-fx>=
         | fixnum?
         | boolean?
         | empty?
         | void?
         | ascii-char?
         | error?
         | not
         | pair?
         | vector?
         | procedure?
         | cons
         | unsafe-car
         | unsafe-cdr
         | unsafe-make-vector
         | unsafe-vector-length
         | unsafe-vector-set!
```

```
          |  unsafe-vector-ref
          |  unsafe-procedure-arity

   aloc ::= aloc?

 fixnum ::= int61?

  uint8 ::= uint8?

 ascii- ::= ascii-char-literal?
  char-
literal
```

(closure-lang-v9? *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `closure-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

---

closure-lang-v9 : grammar?

```
        p ::= (module e)

     pred ::= e
           |  (true)
           |  (false)
           |  (not pred)
           |  (let ([aloc e] ...) pred)
           |  (if pred pred pred)

        e ::= v
           |  (primop e ...)
           |  (closure-ref e e)
           |  (closure-call e e ...)
           |  (call e e ...)
           |  (letrec ([label (lambda (aloc ...) e)] ...) e)
           |  (cletrec ([aloc (make-
              closure label e ...)] ...) e)
           |  (let ([aloc e] ...) e)
           |  (if pred e e)
           |  (begin effect ... e)

   effect ::= (primop e ...)
```

```
        |  (begin effect ... effect)

     v ::= label
        |  aloc
        |  fixnum
        |  #t
        |  #f
        |  empty
        |  (void)
        |  (error uint8)
        |  ascii-char-literal

primop ::= unsafe-fx*
        |  unsafe-fx+
        |  unsafe-fx-
        |  eq?
        |  unsafe-fx<
        |  unsafe-fx<=
        |  unsafe-fx>
        |  unsafe-fx>=
        |  fixnum?
        |  boolean?
        |  empty?
        |  void?
        |  ascii-char?
        |  error?
        |  not
        |  pair?
        |  vector?
        |  procedure?
        |  cons
        |  unsafe-car
        |  unsafe-cdr
        |  unsafe-make-vector
        |  unsafe-vector-length
        |  unsafe-vector-set!
        |  unsafe-vector-ref
        |  unsafe-procedure-arity

  aloc ::= aloc?

 label ::= label?

fixnum ::= int61?
```

```
   uint8 ::= uint8?

  ascii- ::= ascii-char-literal?
   char-
literal
```

(**hoisted-lang-v9?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `hoisted-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

---

hoisted-lang-v9 : grammar?

```
          p ::= (module b ... e)

          b ::= (define label (lambda (aloc ...) e))

       pred ::= e
              | (true)
              | (false)
              | (not pred)
              | (let ([aloc e] ...) pred)
              | (if pred pred pred)

          e ::= v
              | (primop e ...)
              | (closure-ref e e)
              | (closure-call e e ...)
              | (call e e ...)
              | (cletrec ([aloc (make-
                closure label e ...)] ...) e)
              | (let ([aloc e] ...) e)
              | (if pred e e)
              | (begin effect ... e)

     effect ::= (primop e ...)
              | (begin effect ... effect)

          v ::= label
              | aloc
              | fixnum
```

```
            | #t
            | #f
            | empty
            | (void)
            | (error uint8)
            | ascii-char-literal

    primop ::= unsafe-fx*
            | unsafe-fx+
            | unsafe-fx-
            | eq?
            | unsafe-fx<
            | unsafe-fx<=
            | unsafe-fx>
            | unsafe-fx>=
            | fixnum?
            | boolean?
            | empty?
            | void?
            | ascii-char?
            | error?
            | not
            | pair?
            | vector?
            | procedure?
            | cons
            | unsafe-car
            | unsafe-cdr
            | unsafe-make-vector
            | unsafe-vector-length
            | unsafe-vector-set!
            | unsafe-vector-ref
            | unsafe-procedure-arity

      aloc ::= aloc?

     label ::= label?

    fixnum ::= int61?

     uint8 ::= uint8?

ascii-char- ::= ascii-char-literal?
    literal
```

(**proc-exposed-lang-v9?** *a*) → boolean? procedure

a : any/c

Decides whether *a* is a valid program in the `proc-exposed-lang-v9` grammar. The first non-terminal in the grammar defines valid programs.

---

proc-exposed-lang-v9 : grammar?

```
    p ::= (module b ... e)

    b ::= (define label (lambda (aloc ...) e))

 pred ::= e
        | (true)
        | (false)
        | (not pred)
        | (let ([aloc e] ...) pred)
        | (if pred pred pred)

    e ::= v
        | (primop e ...)
        | (call e e ...)
        | (let ([aloc e] ...) e)
        | (if pred e e)
        | (begin effect ... e)

effect ::= (primop e ...)
        | (begin effect ... effect)

    v ::= label
        | aloc
        | fixnum
        | #t
        | #f
        | empty
        | (void)
        | (error uint8)
        | ascii-char-literal

primop ::= unsafe-fx*
        | unsafe-fx+
        | unsafe-fx-
        | eq?
```

```
                    |  unsafe-fx<
                    |  unsafe-fx<=
                    |  unsafe-fx>
                    |  unsafe-fx>=
                    |  fixnum?
                    |  boolean?
                    |  empty?
                    |  void?
                    |  ascii-char?
                    |  error?
                    |  not
                    |  pair?
                    |  vector?
                    |  procedure?
                    |  cons
                    |  unsafe-car
                    |  unsafe-cdr
                    |  unsafe-make-vector
                    |  unsafe-vector-length
                    |  unsafe-vector-set!
                    |  unsafe-vector-ref
                    |  make-procedure
                    |  unsafe-procedure-arity
                    |  unsafe-procedure-label
                    |  unsafe-procedure-ref
                    |  unsafe-procedure-set!

          aloc ::= aloc?

         label ::= label?

        fixnum ::= int61?

         uint8 ::= uint8?

ascii-char-literal ::= ascii-char-literal?
```