QBE Intermediate Language

Table of Contents

1. Basic Concepts

The intermediate language (IL) is a higher-level language than the machine's assembly language. It smoothes most of the irregularities of the underlying hardware and allows an infinite number of temporaries to be used. This higher abstraction level lets frontend programmers focus on language design issues.

Input Files

The intermediate language is provided to QBE as text. Usually, one file is generated per each compilation unit from the frontend input language. An IL file is a sequence of Definitions for data, functions, and types. Once processed by QBE, the resulting file can be assembled and linked using a standard toolchain (e.g., GNU binutils).

Here is a complete "Hello World" IL file which defines a function that prints to the screen. Since the string is not a first class object (only the pointer is) it is defined outside the function's body. Comments start with a # character and finish with the end of the line.

```
# Define the string constant.
data $str = { b "hello world", b 0 }

export function w $main() {
@start
        # Call the puts function with $str as argument.
        %r =w call $puts(l $str)
        ret 0
}
```

If you have read the LLVM language reference, you might recognize the example above. In comparison, QBE makes a much lighter use of types and the syntax is terser.

BNF Notation

The language syntax is vaporously described in the sections below using BNF syntax. The different BNF constructs used are listed below.

- Keywords are enclosed between quotes;
- `... | ...` expresses alternatives;
- `( ... )` groups syntax;

- `[ ... ]` marks the nested syntax as optional;
- `( ... ),` designates a comma-separated list of the enclosed syntax;
- `...*` and `...+` are used for arbitrary and at-least-once repetition respectively.

Sigils

The intermediate language makes heavy use of sigils, all user-defined names are prefixed with a sigil. This is to avoid keyword conflicts, and also to quickly spot the scope and nature of identifiers.

- `:` is for user-defined Aggregate Types
- `$` is for globals (represented by a pointer)
- `%` is for function-scope temporaries
- `@` is for block labels

In this BNF syntax, we use `?IDENT` to designate an identifier starting with the sigil `?`.

Spacing

```
NL := '\n'+
```

Individual tokens in IL files must be separated by one or more spacing characters. Both spaces and tabs are recognized as spacing characters. In data and type definitions, newlines may also be used as spaces to prevent overly long lines. When exactly one of two consecutive tokens is a symbol (for example `,` or `=` or `{`), spacing may be omitted.

2. Types

Simple Types

```
BASETY := 'w' | 'l' | 's' | 'd' # Base types
EXTTY  := BASETY | 'b' | 'h'    # Extended types
```

The IL makes minimal use of types. By design, the types used are restricted to what is necessary for unambiguous compilation to machine code and C

interfacing. Unlike LLVM, QBE is not using types as a means to safety; they are only here for semantic purposes.

The four base types are `w` (word), `l` (long), `s` (single), and `d` (double), they stand respectively for 32-bit and 64-bit integers, and 32-bit and 64-bit floating-point numbers. There are no pointer types available; pointers are typed by an integer type sufficiently wide to represent all memory addresses (e.g., `l` on 64-bit architectures). Temporaries in the IL can only have a base type.

Extended types contain base types plus `b` (byte) and `h` (half word), respectively for 8-bit and 16-bit integers. They are used in Aggregate Types and Data definitions.

For C interfacing, the IL also provides user-defined aggregate types as well as signed and unsigned variants of the sub-word extended types. Read more about these types in the Aggregate Types and Functions sections.

Subtyping

The IL has a minimal subtyping feature, for integer types only. Any value of type `l` can be used in a `w` context. In that case, only the 32 least significant bits of the word value are used.

Make note that it is the opposite of the usual subtyping on integers (in C, we can safely use an `int` where a `long` is expected). A long value cannot be used in word context. The rationale is that a word can be signed or unsigned, so extending it to a long could be done in two ways, either by zero-extension, or by sign-extension.

3. Constants and Vals

```
CONST :=
    ['-'] NUMBER  # Decimal integer
  | 's_' FP       # Single-precision float
  | 'd_' FP       # Double-precision float
  | $IDENT        # Global symbol

DYNCONST :=
    CONST
  | 'thread' $IDENT  # Thread-local symbol

VAL :=
```

```
  DYNCONST
| %IDENT
```

Constants come in two kinds: compile-time constants and dynamic constants. Dynamic constants include compile-time constants and other symbol variants that are only known at program-load time or execution time. Consequently, dynamic constants can only occur in function bodies.

The representation of integers is two's complement. Floating-point numbers are represented using the single-precision and double-precision formats of the IEEE 754 standard.

Constants specify a sequence of bits and are untyped. They are always parsed as 64-bit blobs. Depending on the context surrounding a constant, only some of its bits are used. For example, in the program below, the two variables defined have the same value since the first operand of the subtraction is a word (32-bit) context.

```
%x =w sub -1, 0
%y =w sub 4294967295, 0
```

Because specifying floating-point constants by their bits makes the code less readable, syntactic sugar is provided to express them. Standard scientific notation is prefixed with `s_` and `d_` for single and double precision numbers respectively. Once again, the following example defines twice the same double-precision constant.

```
%x =d add d_0, d_-1
%y =d add d_0, -4616189618054758400
```

Global symbols can also be used directly as constants; they will be resolved and turned into actual numeric constants by the linker.

When the `thread` keyword prefixes a symbol name, the symbol's numeric value is resolved at runtime in the thread-local storage.

Vals are used as arguments in regular, phi, and jump instructions within function definitions. They are either constants or function-scope temporaries.

4. Linkage

```
LINKAGE :=
    'export' [NL]
  | 'thread' [NL]
  | 'section' SECNAME [NL]
  | 'section' SECNAME SECFLAGS [NL]

SECNAME  := '"' .... '"'
SECFLAGS := '"' .... '"'
```

Function and data definitions (see below) can specify linkage information to be passed to the assembler and eventually to the linker.

The `export` linkage flag marks the defined item as visible outside the current file's scope. If absent, the symbol can only be referred to locally. Functions compiled by QBE and called from C need to be exported.

The `thread` linkage flag can only qualify data definitions. It mandates that the object defined is stored in thread-local storage. Each time a runtime thread starts, the supporting platform runtime is in charge of making a new copy of the object for the fresh thread. Objects in thread-local storage must be accessed using the `thread $IDENT` syntax, as specified in the Constants and Vals section.

A `section` flag can be specified to tell the linker to put the defined item in a certain section. The use of the section flag is platform dependent and we refer the user to the documentation of their assembler and linker for relevant information.

```
section ".init_array"
data $.init.f = { l $f }
```

The section flag can be used to add function pointers to a global initialization list, as depicted above. Note that some platforms provide a BSS section that can be used to minimize the footprint of uniformly zeroed data. When this section is available, QBE will automatically make use of it and no section flag is required.

The section and export linkage flags should each appear at most once in a definition. If multiple occurrences are present, QBE is free to use any.

5. Definitions

Definitions are the essential components of an IL file. They can define three types of objects: aggregate types, data, and functions. Aggregate types are never

exported and do not compile to any code. Data and function definitions have file scope and are mutually recursive (even across IL files). Their visibility can be controlled using linkage flags.

Aggregate Types

```
TYPEDEF :=
    # Regular type
    'type' :IDENT '=' ['align' NUMBER]
    '{'
        ( SUBTY [NUMBER] ),
    '}'
  | # Union type
    'type' :IDENT '=' ['align' NUMBER]
    '{'
        (
            '{'
                ( SUBTY [NUMBER] ),
            '}'
        )+
    '}'
  | # Opaque type
    'type' :IDENT '=' 'align' NUMBER '{' NUMBER '}'

SUBTY := EXTTY | :IDENT
```

Aggregate type definitions start with the `type` keyword. They have file scope, but types must be defined before being referenced. The inner structure of a type is expressed by a comma-separated list of types enclosed in curly braces.

```
type :fourfloats = { s, s, d, d }
```

For ease of IL generation, a trailing comma is tolerated by the parser. In case many items of the same type are sequenced (like in a C array), the shorter array syntax can be used.

```
type :abyteandmanywords = { b, w 100 }
```

By default, the alignment of an aggregate type is the maximum alignment of its members. The alignment can be explicitly specified by the programmer.

```
type :cryptovector = align 16 { w 4 }
```

Union types allow the same chunk of memory to be used with different layouts. They are defined by enclosing multiple regular aggregate type bodies in a pair of curly braces. Size and alignment of union types are set to the maximum size and alignment of each variation or, in the case of alignment, can be explicitly specified.

```
type :un9 = { { b } { s } }
```

Opaque types are used when the inner structure of an aggregate cannot be specified; the alignment for opaque types is mandatory. They are defined simply by enclosing their size between curly braces.

```
type :opaque = align 16 { 32 }
```

Data

```
DATADEF :=
    LINKAGE*
    'data' $IDENT '=' ['align' NUMBER]
    '{'
        ( EXTTY DATAITEM+
        | 'z'   NUMBER ),
    '}'

DATAITEM :=
    $IDENT ['+' NUMBER]  # Symbol and offset
  |  '"' ... '"'         # String
  |  CONST               # Constant
```

Data definitions express objects that will be emitted in the compiled file. Their visibility and location in the compiled artifact are controlled with linkage flags described in the Linkage section.

They define a global identifier (starting with the sigil `$`), that will contain a pointer to the object specified by the definition.

Objects are described by a sequence of fields that start with a type letter. This letter can either be an extended type, or the `z` letter. If the letter used is an extended type, the data item following specifies the bits to be stored in the field. When several data items follow a letter, they initialize multiple fields of the same size.

The members of a struct will be packed. This means that padding has to be emitted by the frontend when necessary. Alignment of the whole data objects can be manually specified, and when no alignment is provided, the maximum alignment from the platform is used.

When the `z` letter is used the number following indicates the size of the field; the contents of the field are zero initialized. It can be used to add padding between fields or zero-initialize big arrays.

Here are various examples of data definitions.

```
# Three 32-bit values 1, 2, and 3
# followed by a 0 byte.
data $a = { w 1 2 3, b 0 }

# A thousand bytes 0 initialized.
data $b = { z 1000 }

# An object containing two 64-bit
# fields, one with all bits sets and the
# other containing a pointer to the
# object itself.
data $c = { l -1, l $c }
```

Functions

```
FUNCDEF :=
    LINKAGE*
    'function' [ABITY] $IDENT '(' (PARAM), ')' [NL]
    '{' NL
        BLOCK+
    '}'

PARAM :=
    ABITY %IDENT  # Regular parameter
  | 'env' %IDENT  # Environment parameter (first)
  | '...'         # Variadic marker (last)

SUBWTY := 'sb' | 'ub' | 'sh' | 'uh'  # Sub-word types
ABITY  := BASETY | SUBWTY | :IDENT
```

Function definitions contain the actual code to emit in the compiled file. They define a global symbol that contains a pointer to the function code. This pointer can be used in `call` instructions or stored in memory.

The type given right before the function name is the return type of the function. All return values of this function must have this return type. If the return type is missing, the function must not return any value.

The parameter list is a comma separated list of temporary names prefixed by types. The types are used to correctly implement C compatibility. When an argument has an aggregate type, a pointer to the aggregate is passed by the caller. In the example below, we have to use a load instruction to get the value of the first (and only) member of the struct.

```
type :one = { w }

function w $getone(:one %p) {
@start
        %val =w loadw %p
        ret %val
}
```

If a function accepts or returns values that are smaller than a word, such as `signed char` or `unsigned short` in C, one of the sub-word type must be used. The sub-word types `sb`, `ub`, `sh`, and `uh` stand, respectively, for signed and unsigned 8-bit values, and signed and unsigned 16-bit values. Parameters associated with a sub-word type of bit width N only have their N least significant bits set and have base type `w`. For example, the function

```
function w $addbyte(w %a, sb %b) {
@start
        %bw =w extsb %b
        %val =w add %a, %bw
        ret %val
}
```

needs to sign-extend its second argument before the addition. Dually, return values with sub-word types do not need to be sign or zero extended.

If the parameter list ends with `...`, the function is a variadic function: it can accept a variable number of arguments. To access the extra arguments provided by the caller, use the `vastart` and `vaarg` instructions described in the Variadic section.

Optionally, the parameter list can start with an environment parameter `env %e`. This special parameter is a 64-bit integer temporary (i.e., of type `l`). If the function

does not use its environment parameter, callers can safely omit it. This parameter is invisible to a C caller: for example, the function

```
export function w $add(env %e, w %a, w %b) {
@start
        %c =w add %a, %b
        ret %c
}
```

must be given the C prototype `int add(int, int)`. The intended use of this feature is to pass the environment pointer of closures while retaining a very good compatibility with C. The Call section explains how to pass an environment parameter.

Since global symbols are defined mutually recursive, there is no need for function declarations: a function can be referenced before its definition. Similarly, functions from other modules can be used without previous declaration. All the type information necessary to compile a call is in the instruction itself.

The syntax and semantics for the body of functions are described in the Control section.

6. Control

The IL represents programs as textual transcriptions of control flow graphs. The control flow is serialized as a sequence of blocks of straight-line code which are connected using jump instructions.

Blocks

```
BLOCK :=
    @IDENT NL      # Block label
    ( PHI NL )*    # Phi instructions
    ( INST NL )*   # Regular instructions
    JUMP NL        # Jump or return
```

All blocks have a name that is specified by a label at their beginning. Then follows a sequence of instructions that have "fall-through" flow. Finally one jump terminates the block. The jump can either transfer control to another block of the same function or return; jumps are described further below.

The first block in a function must not be the target of any jump in the program. If a jump to the function start is needed, the frontend must insert an empty prelude block at the beginning of the function.

When one block jumps to the next block in the IL file, it is not necessary to write the jump instruction, it will be automatically added by the parser. For example the start block in the example below jumps directly to the loop block.

```
function $loop() {
@start
@loop
        %x =w phi @start 100, @loop %x1
        %x1 =w sub %x, 1
        jnz %x1, @loop, @end
@end
        ret
}
```

Jumps

```
JUMP :=
    'jmp' @IDENT                # Unconditional
  | 'jnz' VAL, @IDENT, @IDENT   # Conditional
  | 'ret' [VAL]                 # Return
  | 'hlt'                       # Termination
```

A jump instruction ends every block and transfers the control to another program location. The target of a jump must never be the first block in a function. The three kinds of jumps available are described in the following list.

1. Unconditional jump.

   Simply jumps to another block of the same function.

2. Conditional jump.

   When its word argument is non-zero, it jumps to its first label argument; otherwise it jumps to the other label. The argument must be of word type; because of subtyping a long argument can be passed, but only its least significant 32 bits will be compared to 0.

3. Function return.

Terminates the execution of the current function, optionally returning a value to the caller. The value returned must be of the type given in the function prototype. If the function prototype does not specify a return type, no return value can be used.

4. Program termination.

   Terminates the execution of the program with a target-dependent error. This instruction can be used when it is expected that the execution never reaches the end of the block it closes; for example, after having called a function such as `exit()`.

7. Instructions

Instructions are the smallest piece of code in the IL, they form the body of Blocks. The IL uses a three-address code, which means that one instruction computes an operation between two operands and assigns the result to a third one.

An instruction has both a name and a return type, this return type is a base type that defines the size of the instruction's result. The type of the arguments can be unambiguously inferred using the instruction name and the return type. For example, for all arithmetic instructions, the type of the arguments is the same as the return type. The two additions below are valid if `%y` is a word or a long (because of Subtyping).

```
%x =w add 0, %y
%z =w add %x, %x
```

Some instructions, like comparisons and memory loads have operand types that differ from their return types. For instance, two floating points can be compared to give a word result (0 if the comparison succeeds, 1 if it fails).

```
%c =w cgts %a, %b
```

In the example above, both operands have to have single type. This is made explicit by the instruction suffix.

The types of instructions are described below using a short type string. A type string specifies all the valid return types an instruction can have, its arity, and the

type of its arguments depending on its return type.

Type strings begin with acceptable return types, then follows, in parentheses, the possible types for the arguments. If the N-th return type of the type string is used for an instruction, the arguments must use the N-th type listed for them in the type string. When an instruction does not have a return type, the type string only contains the types of the arguments.

The following abbreviations are used.

- `T` stands for `wlsd`
- `I` stands for `wl`
- `F` stands for `sd`
- `m` stands for the type of pointers on the target; on 64-bit architectures it is the same as `l`

For example, consider the type string `wl(F)`, it mentions that the instruction has only one argument and that if the return type used is long, the argument must be of type double.

Arithmetic and Bits

- `add`, `sub`, `div`, `mul` -- `T(T,T)`
- `neg` -- `T(T)`
- `udiv`, `rem`, `urem` -- `I(I,I)`
- `or`, `xor`, `and` -- `I(I,I)`
- `sar`, `shr`, `shl` -- `I(I,ww)`

The base arithmetic instructions in the first bullet are available for all types, integers and floating points.

When `div` is used with word or long return type, the arguments are treated as signed. The unsigned integral division is available as `udiv` instruction. When the result of a division is not an integer, it is truncated towards zero.

The signed and unsigned remainder operations are available as `rem` and `urem`. The sign of the remainder is the same as the one of the dividend. Its magnitude is smaller than the divisor one. These two instructions and `udiv` are only available with integer arguments and result.

Bitwise OR, AND, and XOR operations are available for both integer types. Logical operations of typical programming languages can be implemented using Comparisons and Jumps.

Shift instructions `sar`, `shr`, and `shl`, shift right or left their first operand by the amount from the second operand. The shifting amount is taken modulo the size of the result type. Shifting right can either preserve the sign of the value (using `sar`), or fill the newly freed bits with zeroes (using `shr`). Shifting left always fills the freed bits with zeroes.

Remark that an arithmetic shift right (`sar`) is only equivalent to a division by a power of two for non-negative numbers. This is because the shift right "truncates" towards minus infinity, while the division truncates towards zero.

Memory

- Store instructions.
    - `stored -- (d,m)`
    - `stores -- (s,m)`
    - `storel -- (l,m)`
    - `storew -- (w,m)`
    - `storeh -- (w,m)`
    - `storeb -- (w,m)`

    Store instructions exist to store a value of any base type and any extended type. Since halfwords and bytes are not first class in the IL, `storeh` and `storeb` take a word as argument. Only the first 16 or 8 bits of this word will be stored in memory at the address specified in the second argument.

- Load instructions.
    - `loadd -- d(m)`
    - `loads -- s(m)`
    - `loadl -- l(m)`
    - `loadsw`, `loaduw -- I(mm)`
    - `loadsh`, `loaduh -- I(mm)`
    - `loadsb`, `loadub -- I(mm)`

For types smaller than long, two variants of the load instruction are available: one will sign extend the loaded value, while the other will zero extend it. Note that all loads smaller than long can load to either a long or a word.

The two instructions `loadsw` and `loaduw` have the same effect when they are used to define a word temporary. A `loadw` instruction is provided as syntactic sugar for `loadsw` to make explicit that the extension mechanism used is irrelevant.

- Blits.
  - `blit -- (m,m,w)`

  The blit instruction copies in-memory data from its first address argument to its second address argument. The third argument is the number of bytes to copy. The source and destination spans are required to be either non-overlapping, or fully overlapping (source address identical to the destination address). The byte count argument must be a nonnegative numeric constant; it cannot be a temporary.

  One blit instruction may generate a number of instructions proportional to its byte count argument, consequently, it is recommended to keep this argument relatively small. If large copies are necessary, it is preferable that frontends generate calls to a supporting `memcpy` function.

- Stack allocation.
  - `alloc4 -- m(l)`
  - `alloc8 -- m(l)`
  - `alloc16 -- m(l)`

  These instructions allocate a chunk of memory on the stack. The number ending the instruction name is the alignment required for the allocated slot. QBE will make sure that the returned address is a multiple of that alignment value.

  Stack allocation instructions are used, for example, when compiling the C local variables, because their address can be taken. When compiling Fortran, temporaries can be used directly instead, because it is illegal to take the address of a variable.

The following example makes use of some of the memory instructions. Pointers are stored in long temporaries.

```
%A0 =l alloc4 8      # stack allocate an array A of 2 words
%A1 =l add %A0, 4
storew 43,  %A0      # A[0] <- 43
storew 255, %A1      # A[1] <- 255
%v1 =w loadw  %A0    # %v1 <- A[0] as word
%v2 =w loadsb %A1    # %v2 <- A[1] as signed byte
%v3 =w add %v1, %v2  # %v3 is 42 here
```

Comparisons

Comparison instructions return an integer value (either a word or a long), and compare values of arbitrary types. The returned value is 1 if the two operands satisfy the comparison relation, or 0 otherwise. The names of comparisons respect a standard naming scheme in three parts.

1. All comparisons start with the letter `c`.
2. Then comes a comparison type. The following types are available for integer comparisons:
    - `eq` for equality
    - `ne` for inequality
    - `sle` for signed lower or equal
    - `slt` for signed lower
    - `sge` for signed greater or equal
    - `sgt` for signed greater
    - `ule` for unsigned lower or equal
    - `ult` for unsigned lower
    - `uge` for unsigned greater or equal
    - `ugt` for unsigned greater

    Floating point comparisons use one of these types:

    - `eq` for equality
    - `ne` for inequality
    - `le` for lower or equal
    - `lt` for lower

    - `ge` for greater or equal
    - `gt` for greater
    - `o` for ordered (no operand is a NaN)
    - `uo` for unordered (at least one operand is a NaN)

    Because floating point types always have a sign bit, all the comparisons available are signed.
3. Finally, the instruction name is terminated with a basic type suffix precising the type of the operands to be compared.

For example, `cod` (`I(dd,dd)`) compares two double-precision floating point numbers and returns 1 if the two floating points are not NaNs, or 0 otherwise. The `csltw` (`I(ww,ww)`) instruction compares two words representing signed numbers and returns 1 when the first argument is smaller than the second one.

Conversions

Conversion operations change the representation of a value, possibly modifying it if the target type cannot hold the value of the source type. Conversions can extend the precision of a temporary (e.g., from signed 8-bit to 32-bit), or convert a floating point into an integer and vice versa.

- `extsw`, `extuw` -- `l(w)`
- `extsh`, `extuh` -- `I(ww)`
- `extsb`, `extub` -- `I(ww)`
- `exts` -- `d(s)`
- `truncd` -- `s(d)`
- `stosi` -- `I(ss)`
- `stoui` -- `I(ss)`
- `dtosi` -- `I(dd)`
- `dtoui` -- `I(dd)`
- `swtof` -- `F(ww)`
- `uwtof` -- `F(ww)`
- `sltof` -- `F(ll)`
- `ultof` -- `F(ll)`

Extending the precision of a temporary is done using the `ext` family of instructions. Because QBE types do not specify the signedness (like in LLVM),

extension instructions exist to sign-extend and zero-extend a value. For example, `extsb` takes a word argument and sign-extends the 8 least-significant bits to a full word or long, depending on the return type.

The instructions `exts` (extend single) and `truncd` (truncate double) are provided to change the precision of a floating point value. When the double argument of `truncd` cannot be represented as a single-precision floating point, it is truncated towards zero.

Converting between signed integers and floating points is done using `stosi` (single to signed integer), `stoui` (single to unsigned integer, `dtosi` (double to signed integer), `dtoui` (double to unsigned integer), `swtof` (signed word to float), `uwtof` (unsigned word to float), `sltof` (signed long to float) and `ultof` (unsigned long to float).

Because of Subtyping, there is no need to have an instruction to lower the precision of an integer temporary.

Cast and Copy

The `cast` and `copy` instructions return the bits of their argument verbatim. However a `cast` will change an integer into a floating point of the same width and vice versa.

- `cast` -- `wlsd(sdwl)`
- `copy` -- `T(T)`

Casts can be used to make bitwise operations on the representation of floating point numbers. For example the following program will compute the opposite of the single-precision floating point number `%f` into `%rs`.

```
%b0 =w cast %f
%b1 =w xor 2147483648, %b0  # flip the msb
%rs =s cast %b1
```

Call

```
CALL := [%IDENT '=' ABITY] 'call' VAL '(' (ARG), ')'

ARG :=
```

```
      ABITY VAL  # Regular argument
    | 'env' VAL  # Environment argument (first)
    | '...'      # Variadic marker

SUBWTY := 'sb' | 'ub' | 'sh' | 'uh'  # Sub-word types
ABITY  := BASETY | SUBWTY | :IDENT
```

The call instruction is special in several ways. It is not a three-address instruction and requires the type of all its arguments to be given. Also, the return type can be either a base type or an aggregate type. These specifics are required to compile calls with C compatibility (i.e., to respect the ABI).

When an aggregate type is used as argument type or return type, the value respectively passed or returned needs to be a pointer to a memory location holding the value. This is because aggregate types are not first-class citizens of the IL.

Sub-word types are used for arguments and return values of width less than a word. Details on these types are presented in the Functions section. Arguments with sub-word types need not be sign or zero extended according to their type. Calls with a sub-word return type define a temporary of base type `w` with its most significant bits unspecified.

Unless the called function does not return a value, a return temporary must be specified, even if it is never used afterwards.

An environment parameter can be passed as first argument using the `env` keyword. The passed value must be a 64-bit integer. If the called function does not expect an environment parameter, it will be safely discarded. See the Functions section for more information about environment parameters.

When the called function is variadic, there must be a `...` marker separating the named and variadic arguments.

Variadic

The `vastart` and `vaarg` instructions provide a portable way to access the extra parameters of a variadic function.

- `vastart -- (m)`
- `vaarg -- T(mmmm)`

The `vastart` instruction initializes a *variable argument list* used to access the extra parameters of the enclosing variadic function. It is safe to call it multiple times.

The `vaarg` instruction fetches the next argument from a variable argument list. It is currently limited to fetching arguments that have a base type. This instruction is essentially effectful: calling it twice in a row will return two consecutive arguments from the argument list.

Both instructions take a pointer to a variable argument list as sole argument. The size and alignment of variable argument lists depend on the target used. However, it is possible to conservatively use the maximum size and alignment required by all the targets.

```
type :valist = align 8 { 24 }  # For amd64_sysv
type :valist = align 8 { 32 }  # For arm64
type :valist = align 8 { 8 }   # For rv64
```

The following example defines a variadic function adding its first three arguments.

```
function s $add3(s %a, ...) {
@start
        %ap =l alloc8 32
        vastart %ap
        %r =s call $vadd(s %a, l %ap)
        ret %r
}

function s $vadd(s %a, l %ap) {
@start
        %b =s vaarg %ap
        %c =s vaarg %ap
        %d =s add %a, %b
        %e =s add %d, %c
        ret %e
}
```

Phi

```
   PHI := %IDENT '=' BASETY 'phi' ( @IDENT VAL ),
```

First and foremost, phi instructions are NOT necessary when writing a frontend to QBE. One solution to avoid having to deal with SSA form is to use stack allocated

variables for all source program variables and perform assignments and lookups using Memory operations. This is what LLVM users typically do.

Another solution is to simply emit code that is not in SSA form! Contrary to LLVM, QBE is able to fixup programs not in SSA form without requiring the boilerplate of loading and storing in memory. For example, the following program will be correctly compiled by QBE.

```
@start
        %x =w copy 100
        %s =w copy 0
@loop
        %s =w add %s, %x
        %x =w sub %x, 1
        jnz %x, @loop, @end
@end
        ret %s
```

Now, if you want to know what phi instructions are and how to use them in QBE, you can read the following.

Phi instructions are specific to SSA form. In SSA form values can only be assigned once, without phi instructions, this requirement is too strong to represent many programs. For example consider the following C program.

```
int f(int x) {
        int y;
        if (x)
                y = 1;
        else
                y = 2;
        return y;
}
```

The variable `y` is assigned twice, the solution to translate it in SSA form is to insert a phi instruction.

```
@ifstmt
        jnz %x, @ift, @iff
@ift
        jmp @retstmt
@iff
        jmp @retstmt
@retstmt
```

```
%y =w phi @ift 1, @iff 2
ret %y
```

Phi instructions return one of their arguments depending on where the control came from. In the example, `%y` is set to `1` if the `@ift` branch is taken, or it is set to `2` otherwise.

An important remark about phi instructions is that QBE assumes that if a variable is defined by a phi it respects all the SSA invariants. So it is critical to not use phi instructions unless you know exactly what you are doing.

8. Instructions Index

- Arithmetic and Bits:
  - `add`
  - `and`
  - `div`
  - `mul`
  - `neg`
  - `or`
  - `rem`
  - `sar`
  - `shl`
  - `shr`
  - `sub`
  - `udiv`
  - `urem`
  - `xor`
- Memory:
  - `alloc16`
  - `alloc4`
  - `alloc8`
  - `blit`
  - `loadd`
  - `loadl`
  - `loads`

    - loadsb
    - loadsh
    - loadsw
    - loadub
    - loaduh
    - loaduw
    - loadw
    - storeb
    - stored
    - storeh
    - storel
    - stores
    - storew
- Comparisons:
    - ceqd
    - ceql
    - ceqs
    - ceqw
    - cged
    - cges
    - cgtd
    - cgts
    - cled
    - cles
    - cltd
    - clts
    - cned
    - cnel
    - cnes
    - cnew
    - cod
    - cos
    - csgel
    - csgew
    - csgtl
    - csgtw

    - cslel
    - cslew
    - csltl
    - csltw
    - cugel
    - cugew
    - cugtl
    - cugtw
    - culel
    - culew
    - cultl
    - cultw
    - cuod
    - cuos
- Conversions:
    - dtosi
    - dtoui
    - exts
    - extsb
    - extsh
    - extsw
    - extub
    - extuh
    - extuw
    - sltof
    - ultof
    - stosi
    - stoui
    - swtof
    - uwtof
    - truncd
- Cast and Copy :
    - cast
    - copy

- Call:
  - `call`
- Variadic:
  - `vastart`
  - `vaarg`
- Phi:
  - `phi`
- Jumps:
  - `hlt`
  - `jmp`
  - `jnz`
  - `ret`