Identifiers

local value: `%[-a-zA-Z$._][-a-zA-Z$._0-9]*`

global value: `@[-a-zA-Z$._][-a-zA-Z$._0-9]*`

Type System

Void Type

Syntax: `void`

Integer Type

Syntax: `iN`

Examples:

```
i1 ;a single-bit integer.
i32 ;a 32-bit integer.
```

Pointer Type

Examples:

```
i32*
i8*
[4 x i32]*
```

Array Type

Syntax: `[<# elements> x <elementtype>]`

Examples:

```
[40 x i32] ;Array of 40 32-bit integer values.
[12 x [10 x float]] ;12x10 array of single precision floating-point values.
```

Structure Type

Syntax:

```
%T1 = type { <type list> }     ; Identified normal struct type
%T2 = type <{ <type list> }>   ; Identified packed struct type
```

Examples:

```
{ i32, i32, i32 } ;A triple of three i32 values
{ float, ptr } ;A pair, where the first element is a float and the second
element is a pointer.
<{ i8, i32 }> ;A packed struct known to be 5 bytes in size.
```

Instruction Reference

Terminator Instructions

Terminator Instructions are used to control flows.

ret

Syntax:

```
ret <type> <value>       ; Return a value from a non-void function
ret void                 ; Return from void function
```

Examples:

```
ret i32 5                       ; Return an integer value of 5
ret void                        ; Return from a void function
ret { i32, i8 } { i32 4, i8 2 } ; Return a struct of values 4 and 2
```

br

Syntax:

```
br i1 <cond>, label <iftrue>, label <iffalse> ; Conditional branch
br label <dest>          ; Unconditional branch
```

Examples:

```
Test:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

Binary Operations

add

Syntax:

```
<result> = add <ty> <op1>, <op2>          ; yields ty:result
```

Examples:

```
<result> = add i32 4, %var          ; yields i32:result = 4 + %var
```

sub

Syntax:

```
<result> = sub <ty> <op1>, <op2>          ; yields ty:result
```

Examples:

```
<result> = sub i32 4, %var          ; yields i32:result = 4 - %var
<result> = sub i32 0, %val          ; yields i32:result = -%var
```

mul

Syntax:

```
<result> = mul <ty> <op1>, <op2>          ; yields ty:result
```

Examples:

```
<result> = mul i32 4, %var          ; yields i32:result = 4 * %var
```

udiv

Syntax:

```
<result> = udiv <ty> <op1>, <op2>         ; The value produced is the unsigned
integer quotient of the two operands.
```

Examples:

```
<result> = udiv i32 4, %var          ; yields i32:result = 4 / %var
```

sdiv

Syntax:

```
<result> = sdiv <ty> <op1>, <op2>         ; The value produced is the signed
integer quotient of the two operands rounded towards zero.
```

urem

Syntax:

```
<result> = urem <ty> <op1>, <op2>   ; the remainder from the unsigned division
```

Examples:

```
<result> = urem i32 4, %var          ; yields i32:result = 4 % %var
```

srem

Syntax:

```
<result> = srem <ty> <op1>, <op2>   ; the remainder from the signed division
```

Examples:

```
<result> = srem i32 4, %var          ; yields i32:result = 4 % %var
```

shl

Syntax:

```
<result> = shl <ty> <op1>, <op2>           ; the first operand shifted to the
left a specified number of bits.
```

Examples:

```
<result> = shl i32 4, %var   ; yields i32: 4 << %var
<result> = shl i32 4, 2      ; yields i32: 16
```

lshr

Syntax:

```
<result> = lshr <ty> <op1>, <op2>         ; logical shift right
```

Examples:

```
<result> = lshr i32 4, 1   ; yields i32:result = 2
<result> = lshr i32 4, 2   ; yields i32:result = 1
<result> = lshr i8  4, 3   ; yields i8:result = 0
<result> = lshr i8 -2, 1   ; yields i8:result = 0x7F
```

ashr

Syntax:

```
<result> = ashr <ty> <op1>, <op2>         ; arithmetic shift right
```

Examples:

```
<result> = ashr i32 4, 1   ; yields i32:result = 2
<result> = ashr i32 4, 2   ; yields i32:result = 1
<result> = ashr i8  4, 3   ; yields i8:result = 0
<result> = ashr i8 -2, 1   ; yields i8:result = -1
```

and

Syntax:

```
<result> = and <ty> <op1>, <op2>   ; the bitwise logical and of its two
operands.
```

Examples:

```
<result> = and i32 4, %var         ; yields i32:result = 4 & %var
<result> = and i32 15, 40          ; yields i32:result = 8
<result> = and i32 4, 8            ; yields i32:result = 0
```

or

Syntax:

```
<result> = or <ty> <op1>, <op2>   ; the bitwise logical inclusive or of its two
operands.
```

Examples:

```
<result> = or i32 4, %var         ; yields i32:result = 4 | %var
<result> = or i32 15, 40          ; yields i32:result = 47
<result> = or i32 4, 8            ; yields i32:result = 12
```

xor

Syntax:

```
<result> = xor <ty> <op1>, <op2>   ; the bitwise logical exclusive or of its two
operands
```

Examples:

```
<result> = xor i32 4, %var         ; yields i32:result = 4 ^ %var
<result> = xor i32 15, 40          ; yields i32:result = 39
<result> = xor i32 4, 8            ; yields i32:result = 12
<result> = xor i32 %V, -1          ; yields i32:result = ~%V
```

Memory and Addressing Operations

alloca

Syntax:

```
<result> = alloca <type>
```

The `alloca` instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller.

Examples:

```
%ptr = alloca i32                             ; yields ptr
%ptr = alloca i32, i32 4                      ; yields ptr
%ptr = alloca i32, i32 4, align 1024          ; yields ptr
%ptr = alloca i32, align 1024
```

load

Syntax:

```
<result> = load [volatile] <ty>, ptr <pointer>
<result> = load atomic [volatile] <ty>, ptr <pointer>
```

The `load` instruction is used to read from memory.

Examples:

```
%ptr = alloca i32                               ; yields ptr
store i32 3, i32* %ptr                          ; yields void
%val = load i32, i32* %ptr                      ; yields i32:val = i32 3
```

store

Syntax:

```
store [volatile] <ty> <value>, ptr <pointer>
store atomic [volatile] <ty> <value>, ptr <pointer>
```

The '`store`' instruction is used to write to memory.

Examples:

```
%ptr = alloca i32                               ; yields ptr
store i32 3, i32* %ptr                          ; yields void
%val = load i32, i32* %ptr                      ; yields i32:val = i32 3
```

getelementptr

Syntax:

```
<result> = getelementptr <ty>, ptr <ptrval>{, <ty> <idx>}*
```

The `getelementptr` instruction is used to get the address of a subelement of an aggregate data structure.

Examples:

```
%t4 = getelementptr [10 x [20 x i32]], ptr %t3, i32 0, i32 5
%t5 = getelementptr [20 x i32], ptr %t4, i32 0, i32 13
```

Conversion Operations

bitcast .. to

Syntax:

```
<result> = bitcast <ty> <value> to <ty2>             ;  converts value to type
ty2 without changing any bits.
```

Examples:

```
%Y = bitcast i32* %x to i16*      ; yields i16*:%x
```

zext..to

Syntax:

```
<result> = zext <ty> <value> to <ty2>             ;  converts value to type ty2
with zero extending.
```

trunc..to

Syntax:

```
<result> = trunc <ty> <value> to <ty2>             ;  converts value to type ty2
with truncating.
```

Other Operations

icmp

Syntax:

```
<result> = icmp <cond> <ty> <op1>, <op2>   ; yields i1 or <N x i1>:result
```

The `icmp` instruction returns a boolean value or a vector of boolean values based on comparison of its two integer, integer vector, pointer, or pointer vector operands.

The `cond` can be:

1. `eq`: equal
2. `ne`: not equal
3. `ugt`: unsigned greater than
4. `uge`: unsigned greater or equal
5. `ult`: unsigned less than
6. `ule`: unsigned less or equal
7. `sgt`: signed greater than
8. `sge`: signed greater or equal
9. `slt`: signed less than
10. `sle`: signed less or equal

Examples:

```
<result> = icmp eq i32 4, 5          ; yields: result=false
<result> = icmp ne ptr %X, %X        ; yields: result=false
<result> = icmp ult i16  4, 5        ; yields: result=true
<result> = icmp sgt i16  4, 5        ; yields: result=false
<result> = icmp ule i16 -4, 5        ; yields: result=false
<result> = icmp sge i16  4, 5        ; yields: result=false
```

call

Syntax:

```
<result> = <ty>|<fnty> <fnptrval>(<function args>) ;a simple function call.
```

Examples:

```
%retval = call i32 @test(i32 %argc)
call void %foo(i8 97)
```