Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the `Meta.parse` function), and manipulated by macros. It is a structured representation of code as it is written, constructed by `julia-parser.scm` from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by `julia-syntax.scm`.

First we will focus on the AST, since it is needed to write macros.

Surface syntax AST

Front end ASTs consist almost entirely of `Expr`s and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an Expr, where the first element is the head. For example `(call f x)` corresponds to `Expr(:call, :f, :x)` in Julia.

Calls

Input	AST
`f(x)`	`(call f x)`
`f(x, y=1, z=2)`	`(call f x (kw y 1) (kw z 2))`
`f(x; y=1)`	`(call f (parameters (kw y 1)) x)`
`f(x...)`	`(call f (... x))`

`do` syntax:

```
f(x) do a,b
    body
end
```

parses as `(do (call f x) (-> (tuple a b) (block body)))`.

Operators

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as "syntactic operators". Some operators (`+` and `*`) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Input	AST
`x+y`	`(call + x y)`
`a+b+c+d`	`(call + a b c d)`
`2x`	`(call * 2 x)`
`a&&b`	`(&& a b)`
`x += 1`	`(+= x 1)`
`a ? 1 : 2`	`(if a 1 2)`
`a:b`	`(: a b)`
`a:b:c`	`(: a b c)`
`a,b`	`(tuple a b)`
`a==b`	`(call == a b)`
`1<i<=n`	`(comparison 1 < i <= n)`
`a.b`	`(. a (quote b))`
`a.(b)`	`(. a (tuple b))`

Bracketed forms

Input	AST
`a[i]`	`(ref a i)`
`t[i;j]`	`(typed_vcat t i j)`

`t[i j]`	`(typed_hcat t i j)`
`t[a b; c d]`	`(typed_vcat t (row a b) (row c d))`
`a{b}`	`(curly a b)`
`a{b;c}`	`(curly a (parameters c) b)`
`[x]`	`(vect x)`
`[x,y]`	`(vect x y)`
`[x;y]`	`(vcat x y)`
`[x y]`	`(hcat x y)`
`[x y; z t]`	`(vcat (row x y) (row z t))`
`[x for y in z, a in b]`	`(comprehension x (= y z) (= a b))`
`T[x for y in z]`	`(typed_comprehension T x (= y z))`
`(a, b, c)`	`(tuple a b c)`
`(a; b; c)`	`(block a (block b c))`

Macros

Input	AST
`@m x y`	`(macrocall @m (line) x y)`
`Base.@m x y`	`(macrocall (. Base (quote @m)) (line) x y)`
`@Base.m x y`	`(macrocall (. Base (quote @m)) (line) x y)`

Strings

Input	AST
`"a"`	`"a"`

`x"y"`	`(macrocall @x_str (line) "y")`
`x"y"z`	`(macrocall @x_str (line) "y" "z")`
`"x = $x"`	`(string "x = " x)`
`` `a b c` ``	`(macrocall @cmd (line) "a b c")`

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Imports and such

Input	AST
`import a`	`(import (. a))`
`import a.b.c`	`(import (. a b c))`
`import ...a`	`(import (. . . . a))`
`import a.b, c.d`	`(import (. a b) (. c d))`
`import Base: x`	`(import (: (. Base) (. x)))`
`import Base: x, y`	`(import (: (. Base) (. x) (. y)))`
`export a, b`	`(export a b)`

`using` has the same representation as `import`, but with expression head `:using` instead of `:import`.

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Input	AST

`1111111111111111111`	`(macrocall @int128_str (null) "1111111111111111111")`
`0xfffffffffffffffff`	`(macrocall @uint128_str (null) "0xfffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str (null) "1111....")`

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else
    e
end
```

parses as:

```
(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
            (block (line 5 e))))
```

A `while` loop parses as `(while condition body)`.

A `for` loop parses as `(for (= var iter) body)`. If there is more than one iteration specification, they are parsed as a block: `(for (block (= v1 iter1) (= v2 iter2)) body)`.

`break` and `continue` are parsed as 0-argument expressions `(break)` and `(continue)`.

`let` is parsed as `(let (= var val) body)` or `(let (block (= var1 val1) (= var2 val2) ...) body)`, like `for` loops.

A basic function definition is parsed as `(function (call f x) body)`. A more complex example:

```
function f(x::T; k = 1) where T
    return x+1
end
```

parses as:

```
(function (where (call f (parameters (kw k 1))
                       (:: x T))
                 T)
          (block (line 2) (return (call + x 1))))
```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(struct true (curly Foo (<: T S))
        (block (line 2) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as `(try try_block var catch_block finally_block)`. If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

Quote expressions

Julia source syntax forms for code quoting (`quote` and `:( )`) support interpolation with `$`. In Lisp terminology, this means they are actually "backquote" or "quasiquote" forms. Internally, there is also a need for code quoting without interpolation. In Julia's scheme code, non-interpolating quote is represented with the expression head `inert`.

`inert` expressions are converted to Julia `QuoteNode` objects. These objects wrap a single value of any type, and when evaluated simply return that value.

A `quote` expression whose argument is an atom also gets converted to a `QuoteNode`.

Line numbers

Source location information is represented as `(line line_num file_name)` where the third component is optional (and omitted when the current line number, but not file name, changes).

These expressions are represented as `LineNumberNode`s in Julia.

Macros

Macro hygiene is represented through the expression head pair `escape` and `hygienic-scope`. The result of a macro expansion is automatically wrapped in `(hygienic-scope block module)`, to represent the result of the new scope. The user can insert `(escape block)` inside to interpolate code from the caller.

Lowered form

Lowered form (IR) is more important to the compiler, since it is used for type inference, optimizations like inlining, and code generation. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

In addition to `Symbol`s and some number types, the following data types exist in lowered form:

- `Expr`

  Has a node type indicated by the `head` field, and an `args` field which is a `Vector{Any}` of subexpressions. While almost every part of a surface AST is represented by an `Expr`, the IR uses only a limited number of `Expr`s, mostly for calls, conditional branches (`gotoifnot`), and returns.

- `Slot`

  Identifies arguments and local variables by consecutive numbering. `Slot` is an abstract type with subtypes `SlotNumber` and `TypedSlot`. Both types have an integer-valued `id` field giving the slot index. Most slots have the same type at all uses, and so are represented with `SlotNumber`. The types of these slots are found in the `slottypes` field of their `MethodInstance` object. Slots that require per-use type annotations are represented with `TypedSlot`, which has a `typ` field.

- `CodeInfo`

  Wraps the IR of a group of statements. Its `code` field is an array of expressions to execute.

- `GotoNode`

  Unconditional branch. The argument is the branch target, represented as an index in the code array to jump to.

- `QuoteNode`

  Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a `QuoteNode` whose `value` field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

- `GlobalRef`

Refers to global variable `name` in module `mod`.

- `SSAValue`

  Refers to a consecutively-numbered (starting at 1) static single assignment (SSA) variable inserted by the compiler. The number (`id`) of an `SSAValue` is the code array index of the expression whose value it represents.

- `NewvarNode`

  Marks a point where a variable (slot) is created. This has the effect of resetting a variable to undefined.

`Expr` types

These symbols appear in the `head` field of `Expr`s in lowered form.

- `call`

  Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

- `invoke`

  Function call (static dispatch). `args[1]` is the MethodInstance to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

- `static_parameter`

  Reference a static parameter by index.

- `gotoifnot`

  Conditional branch. If `args[1]` is false, goes to the index identified in `args[2]`.

- `=`

  Assignment. In the IR, the first argument is always a Slot or a GlobalRef.

- `method`

  Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 3-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

The 3-argument form has the following arguments:

- `args[1]`

  A function name, or `false` if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is `false`, it means a method is being added strictly by type, `(::T)(x) = x`.

- `args[2]`

  A `SimpleVector` of argument type data. `args[2][1]` is a `SimpleVector` of the argument types, and `args[2][2]` is a `SimpleVector` of type variables corresponding to the method's static parameters.

- `args[3]`

  A `CodeInfo` of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a `:lambda` expression.

- `struct_type`

  A 7-argument expression that defines a new `struct`:

  - `args[1]`

    The name of the `struct`

  - `args[2]`

    A `call` expression that creates a `SimpleVector` specifying its parameters

  - `args[3]`

    A `call` expression that creates a `SimpleVector` specifying its fieldnames

  - `args[4]`

    A `Symbol`, `GlobalRef`, or `Expr` specifying the supertype (e.g., `:Integer`, `GlobalRef(Core, :Any)`, or `:(Core.apply_type(AbstractArray, T, N))`)

  - `args[5]`

    A `call` expression that creates a `SimpleVector` specifying its fieldtypes

  - `args[6]`

    A Bool, true if `mutable`

    - `args[7]`

      The number of arguments to initialize. This will be the number of fields, or the minimum number of fields called by an inner constructor's `new` statement.

- `abstract_type`

  A 3-argument expression that defines a new abstract type. The arguments are the same as arguments 1, 2, and 4 of `struct_type` expressions.

- `primitive_type`

  A 4-argument expression that defines a new primitive type. Arguments 1, 2, and 4 are the same as `struct_type`. Argument 3 is the number of bits.

  > **Julia 1.5**
  >
  > `struct_type`, `abstract_type`, and `primitive_type` were removed in Julia 1.5 and replaced by calls to new builtins.

- `global`

  Declares a global binding.

- `const`

  Declares a (global) variable as constant.

- `new`

  Allocates a new struct-like object. First argument is the type. The `new` pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary `new` expressions can easily segfault.

- `splatnew`

  Similar to `new`, except field values are passed as a single tuple. Works similarly to `Base.splat(new)` if `new` were a first-class function, hence the name.

- `return`

  Returns its argument as the value of the enclosing function.

- `isdefined`

  `Expr(:isdefined, :x)` returns a Bool indicating whether `x` has already been defined in the current scope.

- `the_exception`

Yields the caught exception inside a `catch` block, as returned by `jl_current_exception()`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error. Yields a token which is consumed by `pop_exception`.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `pop_exception`

  Pop the stack of current exceptions back to the state at the associated `enter` when leaving a catch block. `args[1]` contains the token from the associated `enter`.

  > **Julia 1.1**
  >
  > `pop_exception` is new in Julia 1.1.

- `inbounds`

  Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

- `boundscheck`

  Has the value `false` if inlined into a section of code marked with `@inbounds`, otherwise has the value `true`.

- `loopinfo`

  Marks the end of the a loop. Contains metadata that is passed to `LowerSimdLoop` to either mark the inner loop of `@simd` expression, or to propagate information to LLVM loop passes.

- `copyast`

  Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

- `meta`

  Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

  - `:inline` and `:noinline`: Inlining hints.

- `foreigncall`

Statically-computed container for `ccall` information. The fields are:

- `args[1]` : name

  The expression that'll be parsed for the foreign function.
- `args[2]::Type` : RT

  The (literal) return type, computed statically when the containing method was defined.
- `args[3]::SimpleVector` (of Types) : AT

  The (literal) vector of argument types, computed statically when the containing method was defined.
- `args[4]::Int` : nreq

  The number of required arguments for a varargs function definition.
- `args[5]::QuoteNode{Symbol}` : calling convention

  The calling convention for the call.
- `args[6:length(args[3])]` : arguments

  The values for all the arguments (with types of each given in args[3]).
- `args[(length(args[3]) + 1):end]` : gc-roots

  The additional objects that may need to be gc-rooted for the duration of the call. See Working with LLVM for where these are derived from and how they get handled.

Method

A unique'd container describing the shared metadata for a single method.

- `name`, `module`, `file`, `line`, `sig`

  Metadata to uniquely identify the method for the computer and the human.
- `ambig`

  Cache of other methods that may be ambiguous with this one.
- `specializations`

  Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.
- `source`

The original source code (if available, usually compressed).

- `generator`

  A callable object which can be executed to get specialized source for a specific method signature.

- `roots`

  Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

- `nargs`, `isva`, `called`, `isstaged`, `pure`

  Descriptive bit-fields for the source code of this Method.

- `primary_world`

  The world age that "owns" this Method.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- `specTypes`

  The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

- `def`

  The `Method` that this function describes a specialization of. Or a `Module`, if this is a top-level Lambda expanded in Module, and which is not part of a Method.

- `sparam_vals`

  The values of the static parameters in `specTypes` indexed by `def.sparam_syms`. For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

- `uninferred`

  The uncompressed source code for a toplevel thunk. Additionally, for a generated function, this is one of many places that the source code might be found.

- `backedges`

  We store the reverse-list of cache dependencies for efficient tracking of incremental reanalysis/recompilation work that may be needed after a new method definitions. This works by keeping a list of the other `MethodInstance` that have been inferred or optimized to contain a possible call to this `MethodInstance`. Those optimization results might be stored somewhere in the `cache`, or it might have been the result of something we didn't want to cache, such as constant propagation. Thus we merge all of those backedges to various cache entries here (there's almost always only the one applicable cache entry with a sentinal value for max_world anyways).

- `cache`

  Cache of `CodeInstance` objects that share this template instantiation.

CodeInstance

- `def`

  The `MethodInstance` that this cache entry is derived from.

- `rettype`/`rettype_const`

  The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

- `inferred`

  May contain a cache of the inferred source for this function, or it could be set to `nothing` to just indicate `rettype` is inferred.

- `ftpr`

  The generic jlcall entry point.

- `jlcall_api`

  The ABI to use when calling `fptr`. Some significant ones include:

  - 0 - Not compiled yet
  - 1 - JL*CALLABLE `jl*value*t ()(jl*function*t *f, jl*value*t *args[nargs], uint32*t nargs)`
  - 2 - Constant (value stored in `rettype_const`)
  - 3 - With Static-parameters forwarded `jl_value_t *(*)(jl_svec_t *sparams, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 4 - Run in interpreter `jl_value_t *(*)(jl_method_instance_t *meth, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

- `min_world` / `max_world`

  The range of world ages for which this method instance is valid to be called. If max_world is the special token value `-1`, the value is not yet known. It may continue to be used until we encounter a backedge that requires us to reconsider.

CodeInfo

A (usually temporary) container for holding lowered source code.

- `code`

  An `Any` array of statements

- `slotnames`

  An array of symbols giving names for each slot (argument or local variable).

- `slotflags`

  A `UInt8` array of slot properties, represented as bit flags:

  - 2 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 8 - const (currently unused for local variables)
  - 16 - statically assigned once
  - 32 - might be used before assigned. This flag is only valid after type inference.

- `ssavaluetypes`

  Either an array or an `Int`.

  If an `Int`, it gives the number of compiler-inserted temporary locations in the function (the length of `code` array). If an array, specifies a type for each location.

- `ssaflags`

  Statement-level flags for each expression in the function. Many of these are reserved, but not yet implemented:

  - 0 = inbounds
  - 1,2 = <reserved> inlinehint,always-inline,noinline
  - 3 = <reserved> strict-ieee (strictfp)
  - 4-6 = <unused>
  - 7 = <reserved> has out-of-band info

- `linetable`

  An array of source location objects

- `codelocs`

An array of integer indices into the `linetable`, giving the location associated with each statement.

Optional Fields:

- `slottypes`

  An array of types for the slots.

- `rettype`

  The inferred return type of the lowered form (IR). Default value is `Any`.

- `method_for_inference_limit_heuristics`

  The `method_for_inference_heuristics` will expand the given method's generator if necessary during inference.

- `parent`

  The `MethodInstance` that "owns" this object (if applicable).

- `min_world`/`max_world`

  The range of world ages for which this code was valid at the time when it had been inferred.

Boolean properties:

- `inferred`

  Whether this has been produced by type inference.

- `inlineable`

  Whether this should be eligible for inlining.

- `propagate_inbounds`

  Whether this should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

- `pure`

  Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

« Initialization of the Julia runtime More about types »