rough Adscript Spec

This specification is not very complete, but still documents Adscript.

adn

Adscript Data Notation is a modified version of EDN. It may become its own spec in the future, but at the moment it is embedded here.

Data types

adn type	implementation	example
`id`	none (resolved at compile time)	`f`
`char`	`unsigned int`	`\A`
`int`	`int64_t`	`42`
`float`	`double`	`13.37`
`str`	`char*`	`"hi"`
`list`	none (resolved at compile time)	`(f 1)`
`hetvec`	`void**`	`["hi" 2]`
`homovec`	`T*`	`#[1 2]`

Maps are to be defined. (Probably `{1 2}`)

Adscript

Functions

In Adscript there is another native data type: The function. All functions that are not natively implemented are "first-class values".

Functions can be created using the `fn` function and called using a `list`:

```
(fn [<parameters>] <return type> <body>)

;; arguments of variable length
(fn [<parameters>]' <return type> <body>)
```

For example:

```
((fn [int i] int i) 1)
```

Structs (not implemented yet)

By quoting a list you can create a struct data type.

```
'(int a int b int c)
```

Additional builtin functions

`def` Defines a compile-time constant.

```
(def <identifier> <value>)
```

`deft` Defines a data type.

```
(deft <identifier> <data type>)
```

```
(deft int32 i32)
(deft xy '(int32 x int32 y))
```

`defn` Defines non-anonymous functions.

```
(defn <identifier> [<parameters>] <return type> <body>)
```

`let` (not implemented yet)

Defines a "final variable"/"run time constant", works like `let` in Clojure.

```
(let <identifier> <value>)
(let a 42)
```

`var`

Defines a variable that can be changed later.

```
(var <identifier> <value>)
(var a 42)
```

`set`

Sets the value for a variable or array element. It can also be used for setting a pointer's value, please use `setptr` instead.

```
(set <id> <value>)
(set <element> <value>)
```

```
(set a 42)
```

```
(var b #[1 2 4 8 1])
(set (b 4) 16)
```

`setptr`

Sets the value for a pointer.

```
(setptr <pointer> <value>)
```

```
(var i 21)
(setptr (ref i) 42)
```

(fn []

)

`+`, `-`, `*`, `/`, `%`, `|`, `&`, `^`, `~`, `=`, `<`, `>`, `<=`, `>=`, `or`, `and`, `xor`, `not`

These functions are so obvious that they will be documented later.

`if`

A conditional expression, exactly like in Clojure.

```
(if <condition> <then> <else>)
(if 1 42 10)
```

`ref`

Creates a pointer to a reference.

```
(ref <value>)
(ref (#[1 2 3] 1))
```

`deref`

Dereferences a pointer.

```
(deref <pointer>)
(deref (ref ("ABC*" 3)))
```

native-llvm (not implemented yet)

Equivalent to the asm "function" in c with llvm IR.

native-c (not implemented yet)

Equivalent to the asm "function" in c but with c code.

native-c++ (not implemented yet)

Equivalent to the asm "function" in c but with c++ code.