Simple, but ugly

Here is a simple way to add built-in functions to your language. It’s ugly, but it works. For example, let’s add two builtin functions: `println` and `makeArray`. One does so by adding some tests to the `evalCall` function. Originally, this function looks like:

```javascript
function evalCall(pt, env)
{
    var name = getCallFunction(pt);
    var args = getCallArgs(pt);
    var eargs = evalArgs(args, env);
    var closure = eval(name, env);
    var params = getClosureParams(closure);
    var body = getClosureBody(closure);
    var denv = getClosureEnvironment(closure);
    var xenv = EnvExtend(denv, params, eargs);

    return eval(body, xenv);
}
```

We modify the function to check for a call to the built-in function:

```javascript
function evalCall(pt, env)
{
    var name = getCallFunction(pt);
    var args = getCallArgs(pt);
    var eargs = evalArgs(args, env);
    // check for built-in functions here
    if (isString(name) && identifierEquals(name, "println"))
        return evalPrintln(eargs);
    else if (isString(name) && identifierEquals(name, "makeArray"))
        return evalMakeArray(eargs);
    else
    {
        var closure = eval(name, env);
        var params = getClosureParams(closure);
        var body = getClosureBody(closure);
        var denv = getClosureEnvironment(closure);
        var xenv = EnvExtend(denv, params, eargs);

        return eval(body, xenv);
    }
}
```

and dispatch to the appropriate handler for the built-in. Finally, we add the `evalPrintln` and `evalMakeArray` evaluators to our evaluation module:

```javascript
function evalPrintln(eargs)
{
    while (eargs != null)
    {
        arg = eargs.left;
        display(arg);
        eargs = eargs.right;
    }
    return arg;
}
```
function evalMakeArray(eargs)
{
    var a = new Lexeme(ARRAY);
    a.aval = new Lexeme[eargs.left.ival]; // or something
    return a;
}

You can perform similar actions for each of your built-ins.

A better way

We start by adding a new value field to our lexeme, which we will call fval. This field will hold a pointer to an evaluation function. Next, we first create some *builtin* lexemes:

```
var printlnB = new Lexeme(BUILTIN);
printlnB.fval = evalPrint;
var makeArrayB = new Lexeme(BUILTIN);
makeArrayB.fval = evalMakeArray;
```

We also build some identifier lexemes:

```
var printlnID = new Lexeme(ID);
printlnID.sval = "println";
var makeArrayID = new Lexeme(ID);
makeArrayID.sval = "makeArray;"
```

Our next step is to insert our builtin lexemes into the global environment, binding them to the appropriate identifiers:

```
insertEnv(global,printlnID,printlnB);
insertEnv(global,makeArrayID,makeArrayB);
```

Since we will do this for a possibly large set of builtin functions, it will be useful to define a function to assist us in this task:

```
function addBuiltin(env,name,evaluator)
{
    var b = new Lexeme(BUILTIN);
    b.fval = evaluator;
    var v = new Lexeme(ID);
    v.sval = name;
    insertEnv(env,v,b);
}
```

We would use this function like so:

```
addBuiltin(global,"println",evalPrint);
addBuiltin(global,"makeArray",evalMakeArray);
```

Finally, we modify evalCall to handle builtin lexemes:

```
function evalCall(t,env)
{
    // this code assumes a function call of the form f(x,y)
    var name = getCallFunction(pt);
    var args = getCallArgs(pt);
    var eargs = evalArgs(args,env);
    var closure = eval(name,env);

    if (closure.type == BUILTIN)
    {
        var evaluator = closure.fval
        return evaluator(eargs);
    }
    else
    {
        var params = getClosureParams(closure);
```
The advantage of this method is two-fold:

1. We can easily add new builtins with a minimum of modification to existing code.
2. We can now override builtin functions to extend their behaviors