JL

JSON Manipulation Language

Json Objects and JL's Motivation

```
[
  {
    name: "John",
    from: "New York"
   },
  {
    name:"Bob",
    from:"California"
   }
]
```

Typically access using obj[0][“from”]

Json Objects

- But what if you have and you want all of Johns Roommates?

```
[
  {
    "name": "John",
    "homes": [
            {
                "city": "New York",
                "roommates": ["Bill","James"]
            },
            {
               "city": "San Francisco",
                  "roommates": ["Chris"]
            }
     ]
   }
   ....
];
object[i][“homes”][j][“roommates”] ??
```

JL Language Design

- General
  - General purpose language for json manipulation + other tasks
- Simple
  - No variable declaration, no nested function, only two scopes (local & global), garbage collected.
  - Types are automatically promoted. Lists and Hashmaps can be concatenated to other Lists and Hashmaps
- Dynamic
- Platform neutral
- JSON support
  - Lists in JL are defined as “[]”. Hashmaps are defined using “{}”
  - Inside our language there is a JSON parser
  - An idiom: apply selector to parse(get(“http://…”))

Full Feature Language

```
fun fib(x) {
      if (x == 0) then
            return 0;
      if (x == 1) then
            return 1;
      return fib(x-1) + fib(x-2);
}
```

Easy Manipulation of Lists + Maps

```
result = [1,2,3] + [2,3];
print(result);

[1,2,3,2,3]

result = {“test”:2} + {“test2”:3};
print(result);

{“test”:2, “test2”:3}
```

Selectors

- `object[:string]`
  - Descend one level from the root
- `object[::string]`
  - Get all which match
- `object[::string>]`
  - Get parents
- `object[::string, restriction]`
  - Get all which match and obey the restriction

Back to John's roommates..

```
[
  {
    "name": "John",
    "homes": [
              {
                  "city": "New York",
                  "roommates": ["Bill","James"]
              },
              {
                 "city": "San Francisco",
                 "roommates": ["Chris"]
              }
     ]
   }
];
```

Using JL Selectors

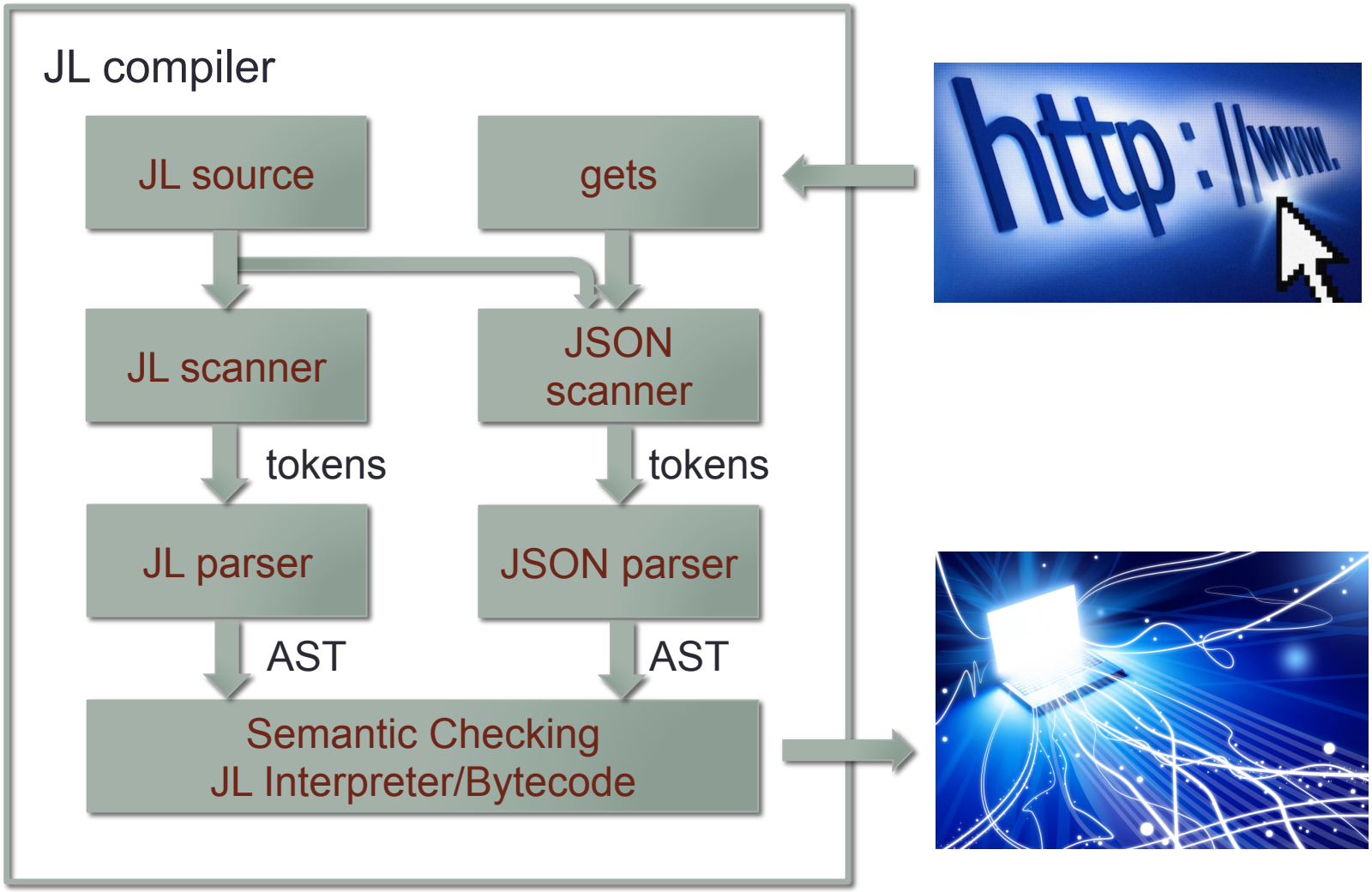
```
//restriction function
fun findJohn(obj) {
    return obj[:"name"] == "John";
}

//the selector
result = [];
johnList = object[::"name">, findJohn];
john = johnList[:"0"];
foreach roommateList : john[::"roommates"] do{
    result = result + roommateList;
}
print(result);
```

Architectural Design

- Lexical Analyzer
- Parser
- Semantic Analyzer
- Translator and Executor
- Top Level – “JL”

Architecture



Bytecode

New Codes:

Lis	Sel
Has	Slc
Rsn	Nit

```
a = "hello";
b = {"value":4};
fun listAdd() {
   c = 3;
   return [a,2]+[c]+ [b[:"value"]];
}
print(listAdd());
```

```
2 global variables
0 Jsr 2
1 Hlt
2 Ent 0
3 Lit hello
4 Str 1
5 Drp
6 Lit 4
7 Lit "value”
8 Has 1        #hashmap
9 Str 0
10 Drp
11 Jsr 16
12 Jsr -1
13 Drp
14 Lit 0
15 Rts 0
16 Ent 1
17 Lit 3
18 Sfp 1
19 Drp
20 Lod 1
21 Lit 2
22 Lis 2    #list
23 Lfp 1
24 Lis 1
25 Add
26 Lis 0
27 Lit
28 Lit "value”
29 Co
30 Lod 0
31 Slc 1  #selector
32 Lis 1
33 Add
34 Rts 0
35 Lit 0
36 Rts 0
```

Selector Bytecode

```
result = a[::"name">,atLeastXFollowers];
```

```
8 Rsn 63   //jump to restriction function
9 Lis 1  //list of restrictions
10 Lit >  //postfix
11 Lit "name”  //selector name
12 Dco  //double colon
13 Lod 2 //load the object
14 Slc 1 //selector length + start selection
```

Lessons Learned

- Get “Hello World” working as soon as possible. Everything will flow from that.
- Do an interpreter; it really does help to test newly added features.
- Use an incremental development strategy – don’t try to put everything together in one go
- Build big code examples before starting

Thanks!

And a live demo....