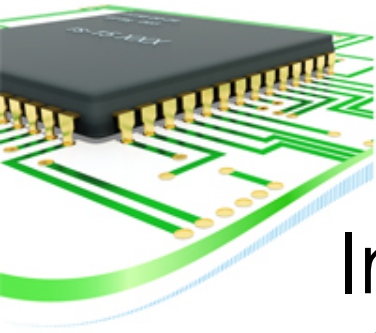


**Semantic Analysis**  
on top of Flex + Bison

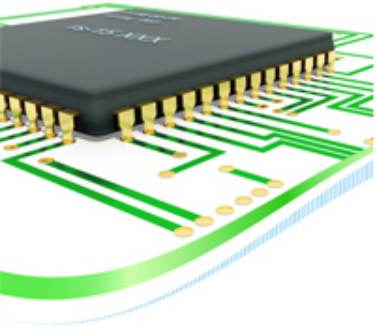


# Problem

## JSON-like translation

Implement a tool to verify and load the value contained in a text file.

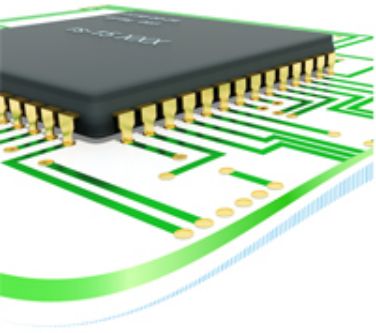
The value is encoded in JSON-like syntax. Make the value available for application specific purposes.



# Problem

## Details and constraints

- A value can be simple or complex
- Simple values are numbers and strings
- Complex values are dictionaries and arrays
- A dictionary is sequence of members enclosed by “{” and “}”
- A dictionary without member is possible
- Members are separated by commas
- A member can be either
  - in the form of “<member-name>” “:” value
  - or, in the form of <member-name> “:” value
- An array is sequence of values enclosed by “[” and “]”
- An array without value is possible
- Values are separated by commas



# Problem

## Sample Input

```
{  
  m: "Test\x30\nAbc",  
  "field": [1, "Anv", {id:"Test", arr:[]}],  
  "anothermember": {},  
  val: 0.5E2  
}
```



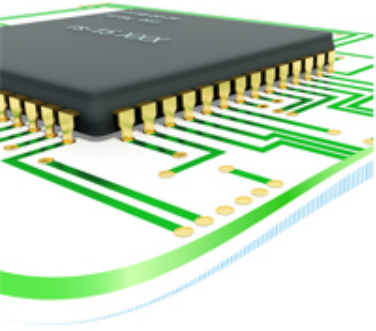
# Solution Steps

**Analyze the problem statement and the example to**

- **Identify Tokens**
- **Develop Grammar**
- **Design Semantic Representation**

**The tokens are**

**“{”, “}”, “[”, “]”, comma, colon, identifier, string, number**



# Solution Steps

Develop grammar

```
dictionary: SOB memberlist EOB;
```

```
array: OB valuelist CB;
```

```
memberlist: member | member COMMA memberlist | ;
```

```
valuelist: jvalue | jvalue COMMA valuelist | ;
```

```
member: name COLON jvalue;
```

```
name: ID | STR;
```

```
jvalue: STR | NUM | dictionary | array;
```

```
%token SOB
```

```
%token EOB
```

```
%token OB
```

```
%token CB
```

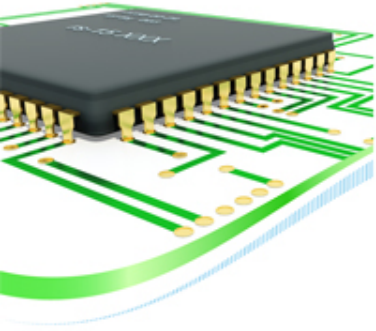
```
%token ID
```

```
%token STR
```

```
%token NUM
```

```
%token COLON
```

```
%token COMMA
```



# Solution Steps

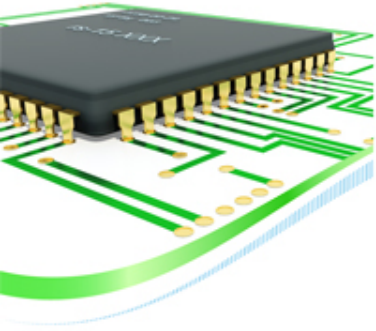
Design semantic representation

The representation is hierarchy of values!

Dictionary and array may have children

```
enum class JValueType
{
    Str = 0,
    Num = 1,
    Dict = 32,
    Arr = 33
};
```

```
class JValue
{
    JValueType type;
public:
    JValue(JValueType t);
    JValueType getType();
    virtual void
report(ofstream *os)=0;
    virtual ~JValue();
};
```



# Solution Steps

Design semantic representation

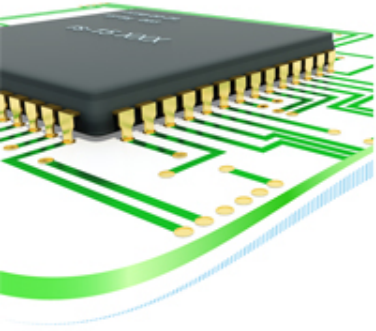
The representation is hierarchy of values!

A simple value without children can be either a double or a string.

```
class JValueDouble : public
JValue
{
    private:
        double    val;
    public:
        JValueDouble(double d);
        virtual void
report(ofstream *os);
};
```

```
class JValueStr : public JValue
{
    private:
        string *str;
    public:
        JValueStr(string *s);
        virtual ~JValueStr();
        virtual void
report(ofstream *os);
};
```





# Solution Steps

Design semantic representation

The representation is hierarchy of values!

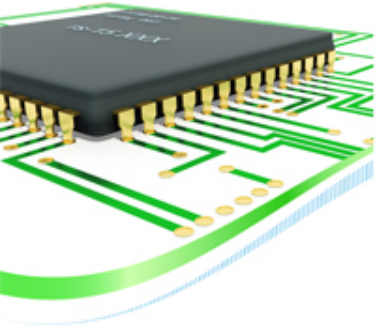
A dictionary is represented by a vector of members.

```
class JMember
{
    private:
        string      *id;
        JValue      *value;
    public:
        JMember(string *pId,
JValue *pValue);
        virtual ~JMember();
        virtual void
report(ofstream *os);
};
```

```
class JDictionary : public JValue
{
    std::vector<JMember *> *arr;

    public:
        JDictionary();
        ~JDictionary();

        void addMember(JMember *m);
        virtual void report(ofstream
*os);
};
```



# Solution Steps

Design semantic representation

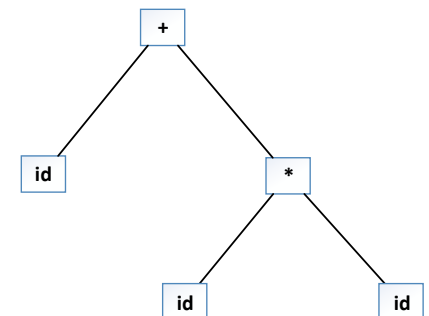
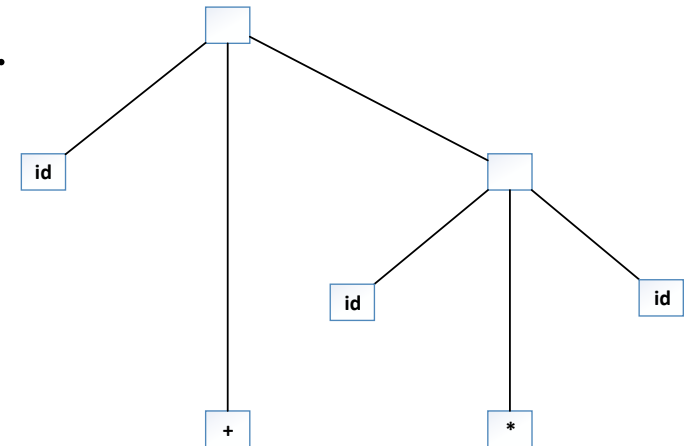
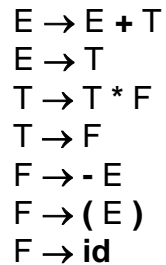
The representation is hierarchy of values!

```
class JArray : public JValue
{
    private:
        std::vector<JValue *> *arr;
    public:
        JArray() ;
        ~JArray() ;

        void addElement(JValue *v) ;
        virtual void report(ofstream *os) ;
};
```



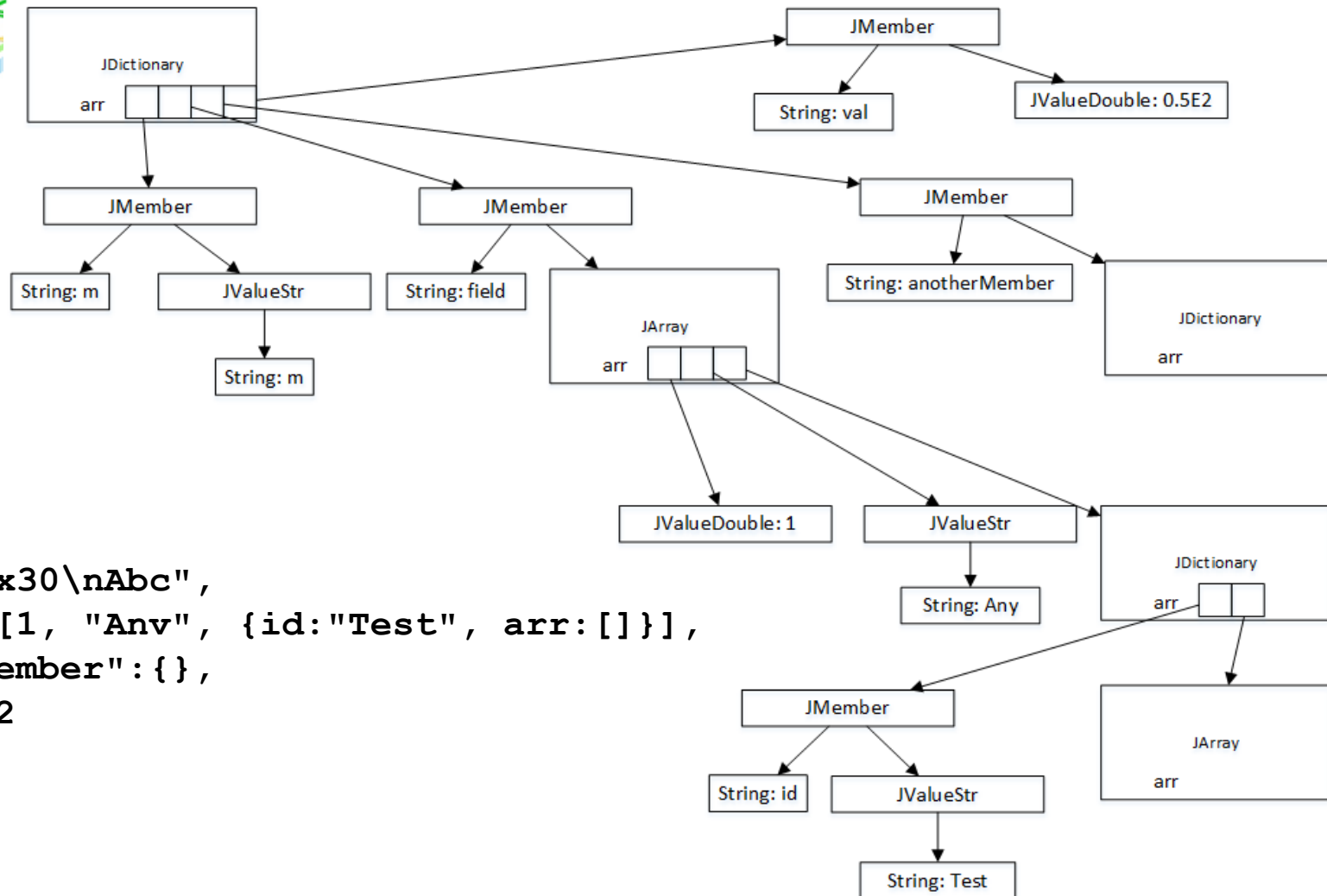
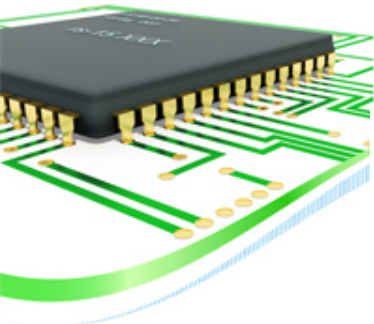
**Input:** id + id \* id





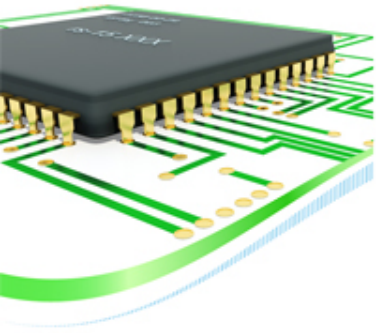
# Abstract Syntax Tree (AST)

Bridging Syntax to Semantic Representation



Input:

```
{
  m: "Test\x30\nAbc",
  "field": [1, "Anv", {id:"Test", arr:[]}],
  "anothermember": {},
  val: 0.5E2
}
```



# Flex + Bison

## Objects for Parsing

**It is a fine wiring!**

### **MyParser**

This is the driver.

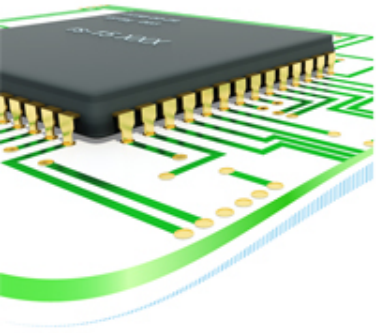
Encompasses the parser, the lexer, and the whole state and semantics.

#### **MyParserBase**

Generated by Bison

#### **MyFlexLexer**

Generated by Flex as subclass of FlexLexer



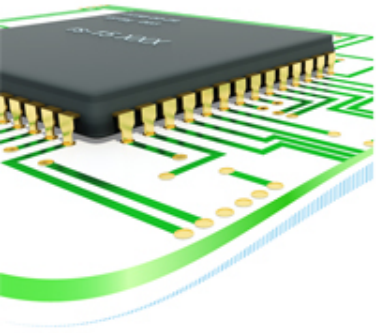
# Flex + Bison

Scraping semantics from lexemes

- Application requests parse using driver
  - The driver sets up configuration and uses auto-generated parser's parse method
    - The parse method runs SR method and requests lexer to report a lexeme
      - The lexeme is reported back to the action code!
        - The action code calculates and sets the semantics for the lexeme.
        - The action code returns the relevant token identifier.

The driver has actions for both lexer and the parser.

The driver is made accessible from both the lexer and the parser.

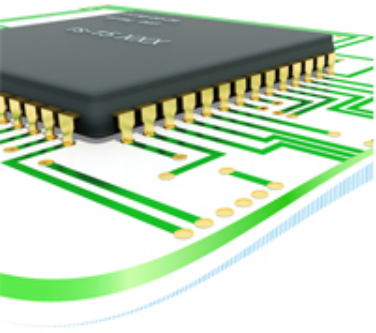


# Flex + Bison

## Synthesizing semantics

The art(!) of returning semantics using fine-wiring.

- The parser calls driver's lex method because it was instructed to do so in .y (see line 19 and its effects). The generated parser passes a pointer where the semantics can be saved! **This is the opportunity to save the pointer** ( lval ).
  - The driver's lex method calls MyFlexLexer's lex method, which is **not** in MyFlexLexer.cpp module! It is hijacked by the YY\_DECL macro through the definition in the .l file (see the lines in .l and its effects).
    - When the driver's action method is called by the lexer, the saved pointer ( lval ) can be used **thread-safely** to store semantics! From this point on, the extracted semantics is in the game field of bison parser.



# Types for Semantics

Bison's `semantic_type`

Use of the union semantic type.

Define data types to represent semantics for both tokens and non-terminals!

```
%define api.value.type union
```

```
%nterm <JValue *> jvalue
```

```
%nterm <JDictionary *> dictionary
```

```
%nterm <JDictionary *> memberlist
```

```
%nterm <JArray *> array
```

```
%nterm <JArray *> valuelist
```

```
%nterm <string *> name
```

```
%nterm <JMember *> member
```

```
%token SOB
```

```
%token EOB
```

```
%token OB
```

```
%token CB
```

```
%token <string *>ID
```

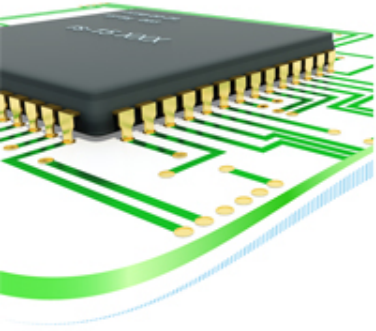
```
%token <string *>STR
```

```
%token <double> NUM
```

```
%token COLON
```

```
%token COMMA
```





# Actions for Semantics

Bison's semantics game!

- See use of semantics in combination with actions and the methods called.
- No globals in solution!
- Download, compile, and observe by using debugger!
- See the output `sample03output.txt` as a means of verification.
- This is also the Intermediate Representation and the Target for this problem.
- What is missing to serve potential application needs!

## QUESTIONS!