Expression Parsing
In C++

Amarendra Joshi

What is Numeric Expression

- Numeric Expressions are composed of
  - Numbers
  - The operators +, −, /, *, ^, %, =
  - Parentheses (,)
  - Variables

- Example:

  10—2*3

  (12-3)*(2/4)

Parsing Expression : The Problem

- 10-2*3

```
a = get first variable/number
while (variable/number present) {
op = get operator
b = get second variable/number
a = a op b
}
```

Parsing Expression : Rules

- Recursive Descent Parser
- Expression Production Rules
  - expression −> term [+ term] [− term]
  - term −> factor [* factor] [/ factor]
  - factor −> variable, number, or (expression)

- Example:
  - 10-5*6
  - 14*(7-c)

Some Assumptions

- Assume this precedence for each operator:
  - highest + – (unary)
  - ^
  - * / %
  - + –
  - lowest =

Recursive Descent Parser : Examples

- 3 Versions
  - Parser for floating numerical values
  - Use of variables
  - Generic Parser

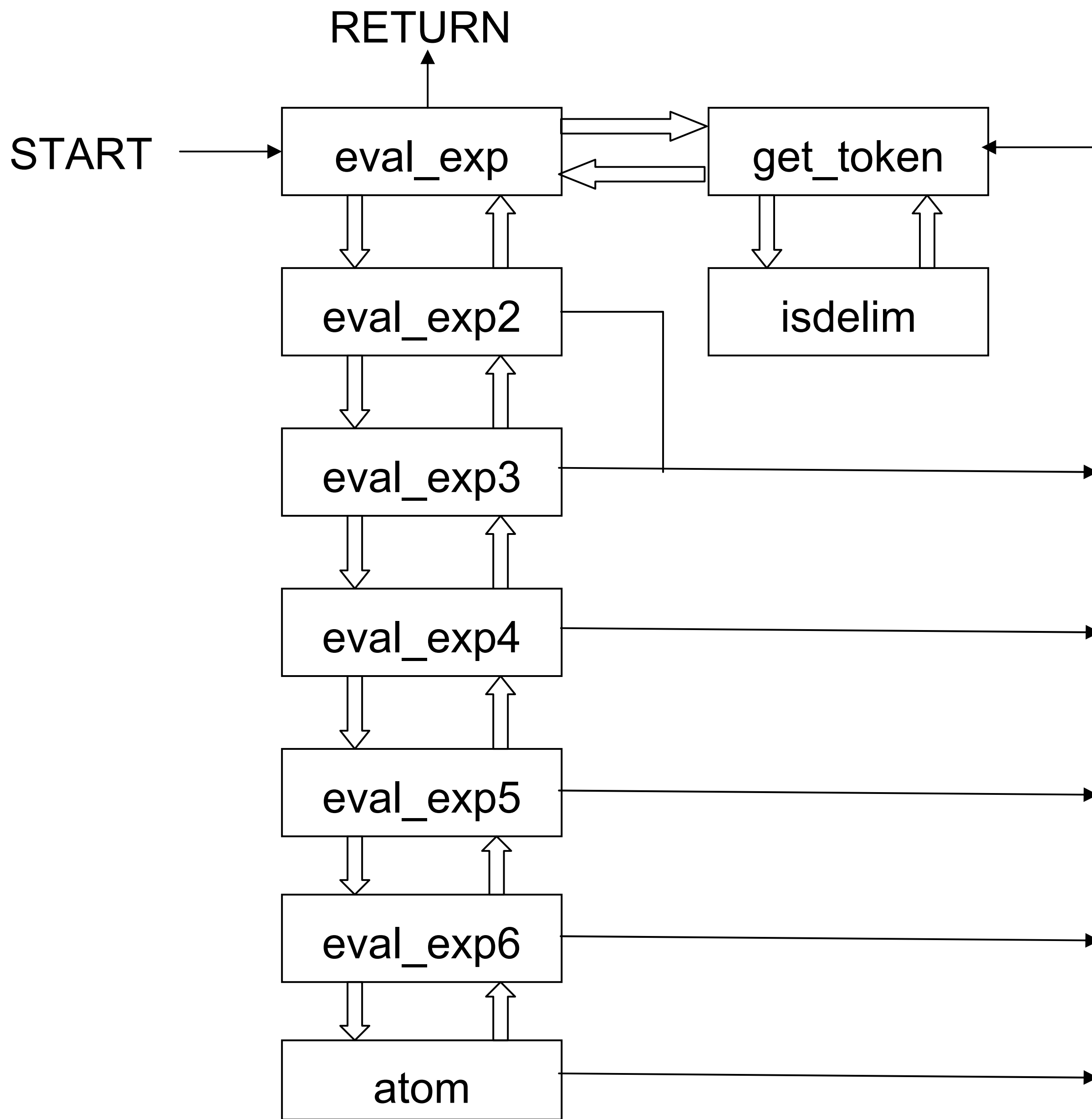
Recursive Descent Parser : Version 1

- Parser for floating point numbers
- No use of variables
- Implement expression production rules
- Minimal error checking

The recursive descent parser

Class Parser

```
class parser
{
    private:
        char *exp_ptr;          // points to the expression
        char token[80];         // holds current token
        char tok_type;          // holds token's type
        void eval_exp2(double &result);
        void eval_exp3(double &result);
        void eval_exp4(double &result);
        void eval_exp5(double &result);
        void eval_exp6(double &result);
        void atom(double &result);
        void get_token();
        void serror(int error);
        int isdelim(char c);
    public:
        parser();
        double eval_exp(char *exp);
};
```


RETURN
START
eval_exp
get_token
eval_exp2
isdelim
eval_exp3
eval_exp4
eval_exp5
eval_exp6
atom

Recursive Descent Parser : Version 2

- Few changes to last version
- Use single variables from A to Z
- Not a case sensitive
- Variables will be stored in an array inside class
- Each variable uses one array location in a 26-element array of doubles
- If you type string, it only takes first letter
- Minimal error checking

Class Parser : Version 2

```
class parser
{
    private:
            char *exp_ptr;              // points to the expression
            char token[80];             // holds current token
            char tok_type;              // holds token's type
            double vars [NUMVARS];  // hold variables' values
            void eval_exp1(double &result);
            void eval_exp2(double &result);
            void eval_exp3(double &result);
            void eval_exp4(double &result);
            void eval_exp5(double &result);
            void eval_exp6(double &result);
            void atom(double &result);
            void get_token();
            void putback();
            void serror(int error);
            double find_var(char *s);
            int isdelim(char c);
    public:
            parser();
            double eval_exp(char *exp);
};
```

Recursive Descent Parser : Version 3

- Generic Parser
- Last two parsers use double values
- Create a parser class template which can use both double and integer values

Generic Parser Class

```
template <class PType>
class parser
 {
    char *exp_ptr; // points to the
                      expression
    char token[80]; // holds current
                      token
    char tok_type; // holds token's type
    PType vars[NUMVARS]; // holds
                      variable's values
    void eval_exp1(PType &result);
    void eval_exp2(PType &result);
    void eval_exp3(PType &result);
    void eval_exp4(PType &result);
    void eval_exp5(PType &result);
    void eval_exp6(PType &result);
    void atom(PType &result);
    void get_token(), putback();
    void serror(int error);
    PType find_var(char *s);
    int isdelim(char c);

public:
    parser();
    PType eval_exp(char *exp);
};
```

Thank You