Advantage

1. easy to read, especially for algorithm code

   Because our function is separated from data type, you don't need to care data type when you are coding functions, this makes the function easy to write and read, or in other words, because we don't write any thing related to data type, your function would be far more short, thus easy to read. The code of this type of programming should be concise as math equations.

   And now I'm a freshman of many language such as python, java. I always find it really hard to read the documentations, often they are hard to get start with. I think this will be different in DFS programming because our document should be well organized. Take the small document below for example.

```
Documentation for DSF programming language
function fun1
{
# description for fun1(x)
need operators:
    {
    operator1
    operatot2
    ...
    }
...
}

struct x1
{
# description for x1
operator1:
    {
    ...
    }
operator2:
    {
    ...
    }
...
}
```

   Actually it feel like I just put the code to the documentation, but this is just for an example, we may omit something above, and we will put some examples in it.

   Now take a look at it, if you want to learn how fun(x1) actually works you just go to fun1's definition, learn what does it do, maybe that is enough for those who want to just use the function, and next time they want use the same function but to a different type, just check the deferent type's operators, if it provide all the operators fun1 need, and all the operators is right for you, just type fun1(different type). So now find it more easier to learn using functions? Actually I want to say if you learn fun1 you have learned fun1, no matter which type of data you use as parameter.

2. Easy for syntax check

Your operators are defined in your data struct, they will also tell functions which type can be operated with them. Below is an example, you define an addition operator in your struct, and the data type that can be add with them. If you try $sum(x1, x2)$ and $x1$ is a real number, $x2$ is a matrix, the checker will tell you they can not be add because the addition.object in real don't have 'matrix'.

```
struct real
{whici
addition:
    {object : 'real' 'integer'}
    {procedure:
        x1 + x2
    }
}

function sum(x1,x2){
    addition(x1,x2)
}
```

And it's easy to check all the operators used in your function.

3. Convenient to share

   For this aspect I will use an example to show you, just imagine you are a data scientist, you want design a algorithm for one type of task, such as sort, we just let it be sort.

   So you begin to program, the first thing is to ask: sort what? And based on the data type you want sort you write you code.

   And others may see it and they really appreciate your algorithm, and they want use it, but guess what? They want to use it for another data type. Unluckily they have to read your code and correct all the things related to data type in your code.

   Now using our DFS programming those unlucky guys find their hope. They don't have to revise your code, they just provide some operators to your functions and it begin to work.

   More specifically, your code designed for sort use some basic operators, such as sequence(which means this operator give the serial number), compare , switch. Any data type contains these three operators can use it, so for users, they don't have to look your code, they just need to know the function of the function you give to them and the operators your function needs.

   This will be really helpful if the algorithm is very long.

4. Fast operation speed

   All the operations is defined in Data Structure, and our functions use these operation to accomplish its function, so if you make sure the functions are the best algorithms, your operations defined in your data structures are the fastest ones, you should get the fastest code.

5. Easy for Parallel Processing

   DFS naturally support Parallel Processing, just arrange some marks to tell compiler the function can be use for multiprocessing. For example, there' s two algorithms, one is **f1** that support parallel processing, another is **f2** which is not, add mark like this to tell compiler.

```
function f1
{
parallel: true
...
}

function f2
{
parallel: false
}
```

%TODO I need to think whether is Function or Data Structure should Parallel mark belong to.

6. Easy to generate new code

You may have use this idea in your code, but have you generated it every structure in your code?