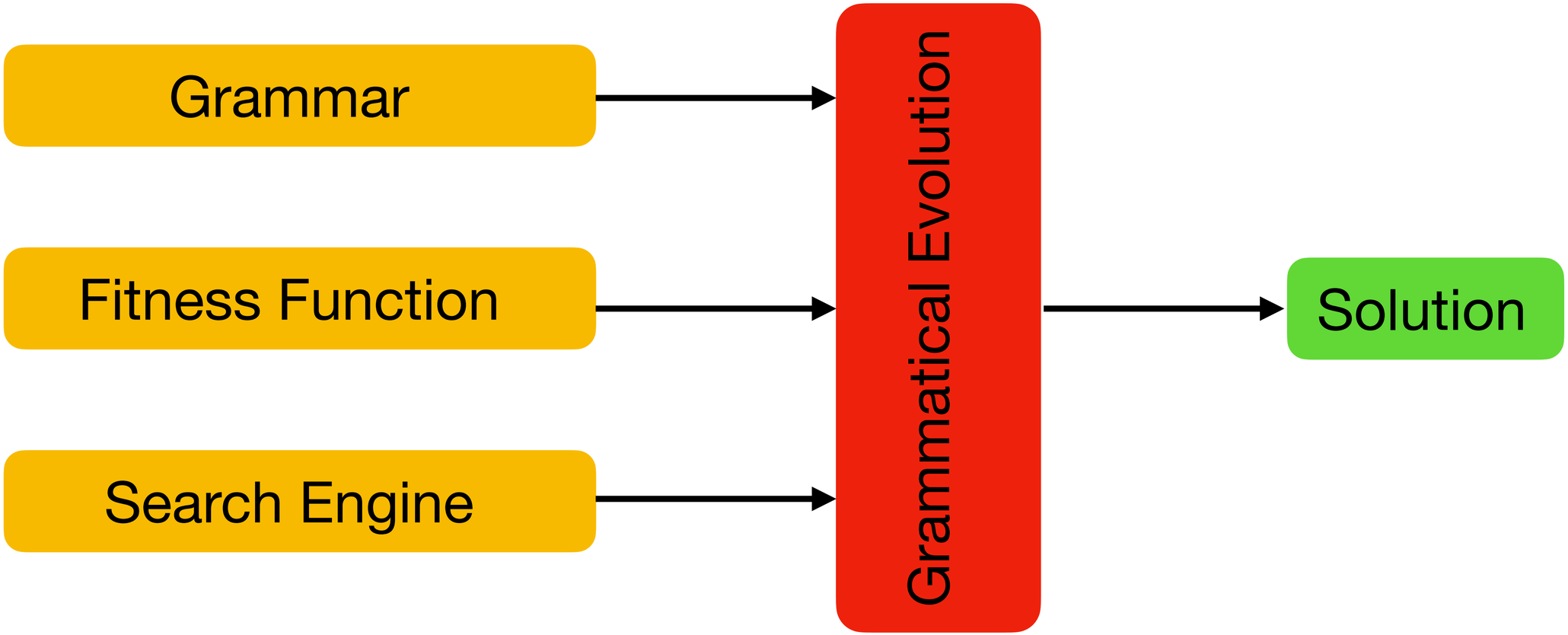
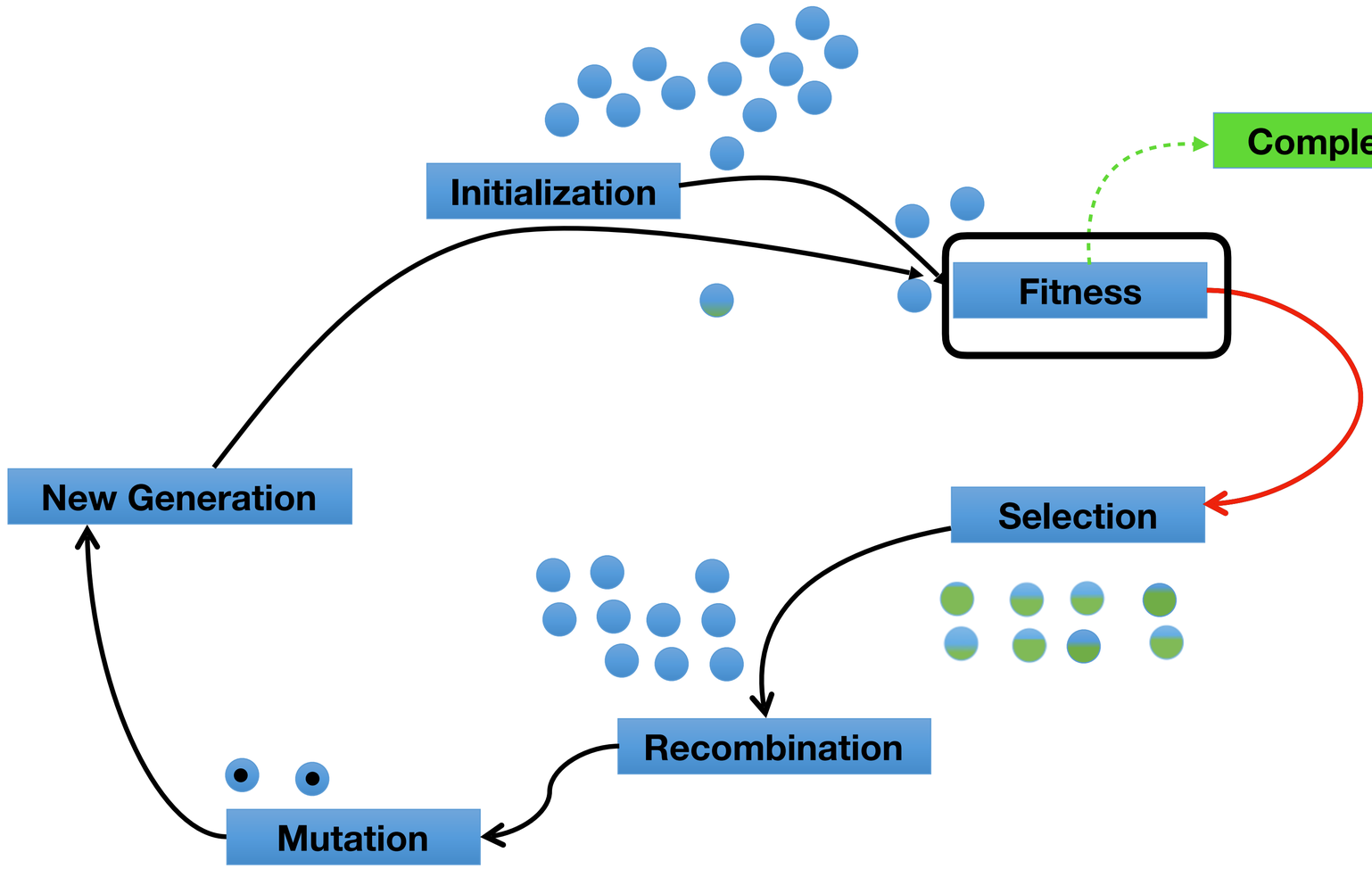

CS6271
5.1 Grammatical Evolution

Overview


Grammar
Fitness Function
Search Engine
Grammatical Evolution
Solution



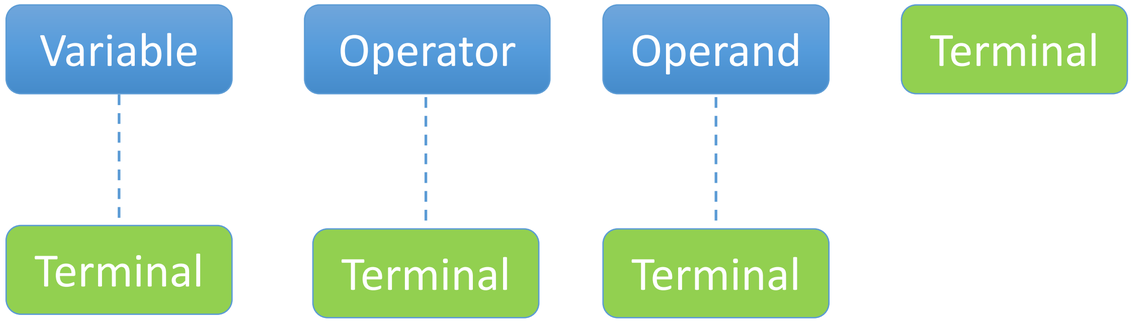
1. Map to program
2. Compile (maybe)
3. Evaluate fitness

Grammars

- Specify legal *sentences* in a language

x+=3;

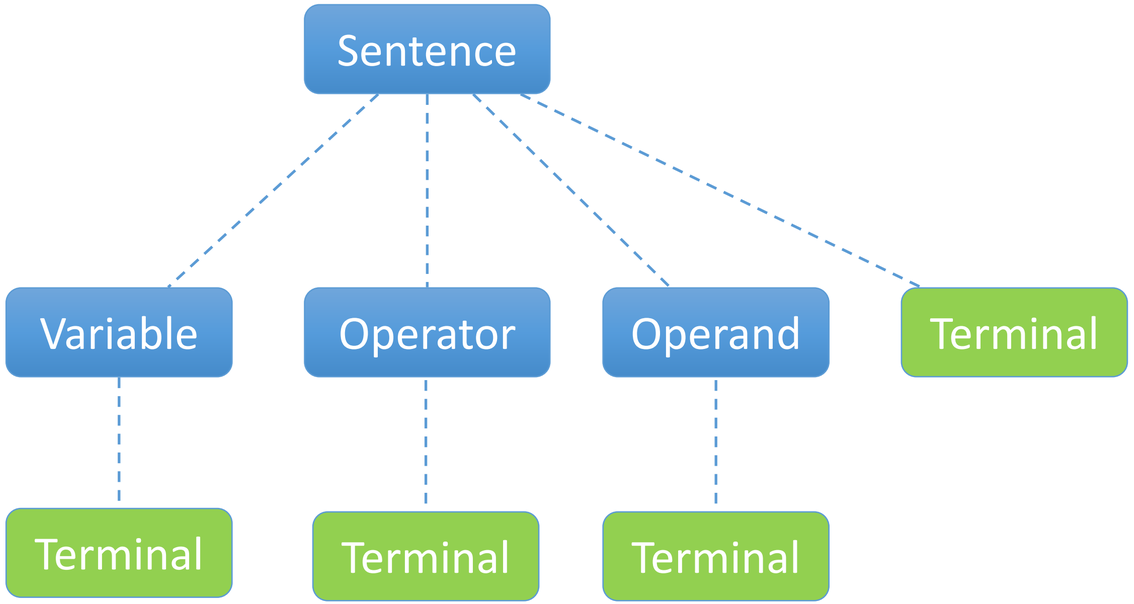
- Tokenize sentence



- Legal sentences map to all terminals

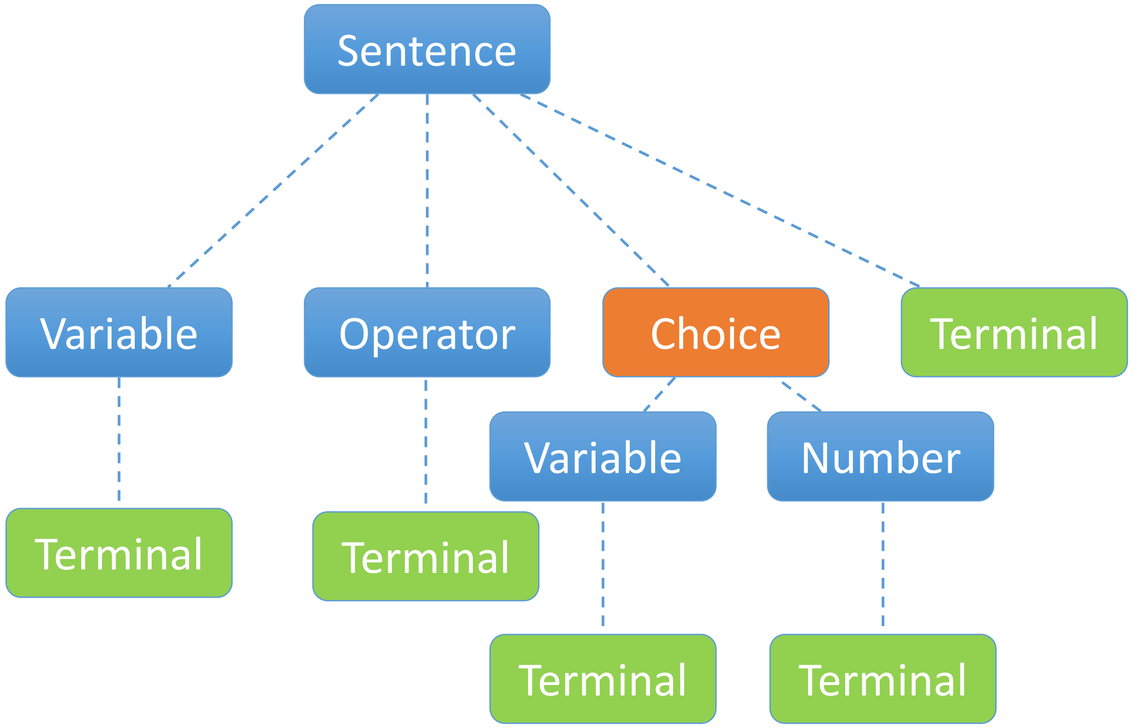
Grammars

- Specify legal *sentences* in a language



Grammars

- Specify legal *sentences* in a language



Formally

- Grammar described by <*S, T, N, P*>
- *S* start symbol
- *T* terminals
- *N* non-terminals
- *P* production rules

Sample Grammar

<e> := <v> | <o> <e> <e>

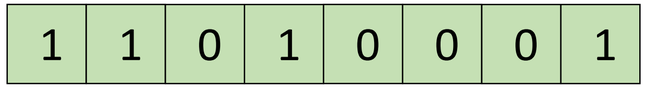
<o> := + | - | * | /

<v> := x | y | z

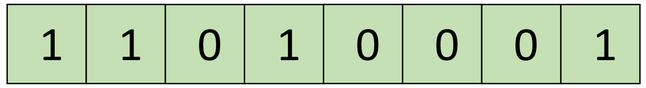
Grammatical Evolution

- Individuals use grammar to generate program
  - Attribute Grammar describing legal syntax
- Application can be changed by
  - New grammar
  - New fitness function
- Advantages
  - Generate complex structure
  - Impose constraints

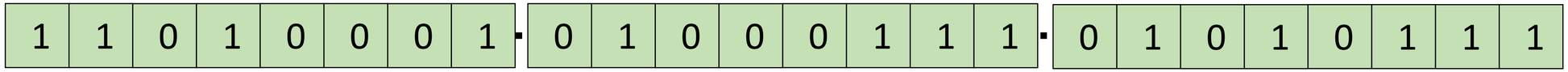
Grammatical Evolution



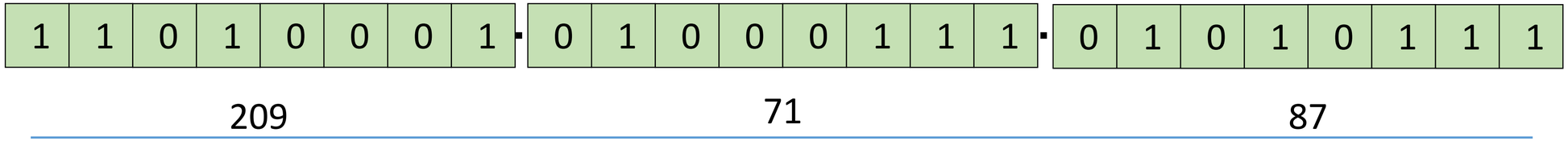
Grammatical Evolution



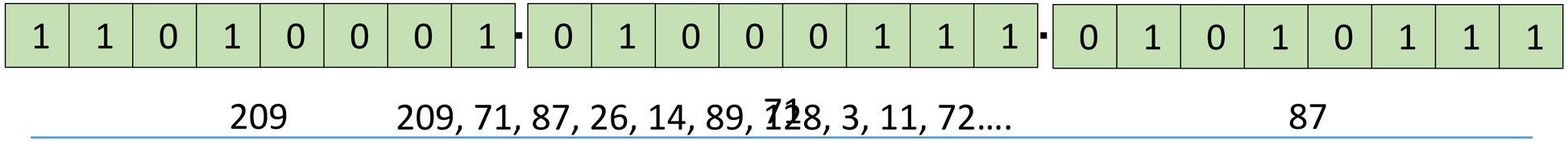
Grammatical Evolution



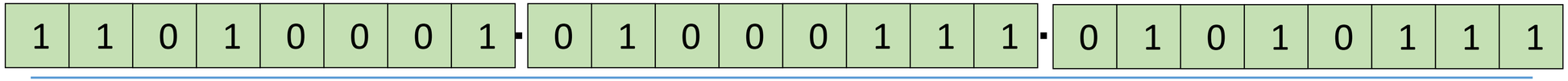
Grammatical Evolution



Grammatical Evolution



Grammatical Evolution



209, 71, 87, 26, 14, 89, 128, 3, 11, 72….

Grammar

<e> := <v> | <o> <e> <e>
<o> := + | - | * | /
<v> := x | y | z

<e><e><e>

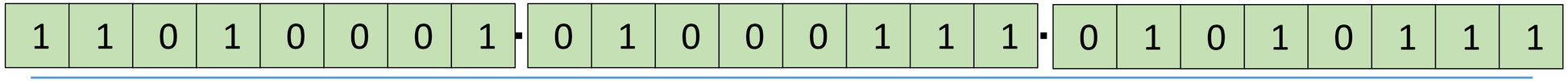
Mapping

Calculate **mod** of codon value;
Select that choice for current non-terminal

<e>:= <v> [Choice 0]
|<o><e><e> [Choice 1]

209 **mod** 2 = 1

Grammatical Evolution



209, 71, 87, 26, 14, 89, 128, 3, 11, 72….

Grammar

<e> := <v> | <o> <e> <e>

<o> := + | - | * | /

<v> := x | y | z

<e> <o> <e> <e>

Mapping

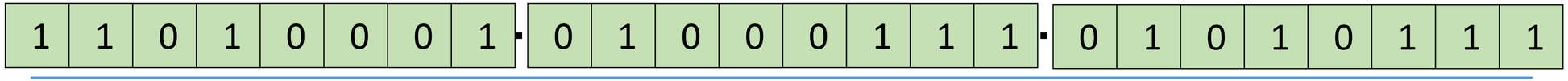
Calculate **mod** of codon value;

Select that choice for current non-terminal

<o>:= + [Choice 0]
| - [Choice 1]
| * [Choice 2]
| / [Choice 3]

71 **mod** 4 = 3

Grammatical Evolution



209, 71, 87, 26, 14, 89, 128, 3, 11, 72….

Grammar

<e> := <v> | <o> <e> <e>
<o> := + | - | * | /
<v> := x | y | z

Mapping

Calculate **mod** of codon value;
Select that choice for current non-terminal

/<e><e>
/<o><e><e><e>
/*<e><e><e>
/*<v><e><e>
/*x<e><e>
/*x y x
(/ (* x y) x)

More complex grammars yield more sophisticated structures…
Programs, music, artwork, etc.

Multicore Attribute Grammatical Evolution

- Automatically generating optimized recursive parallel programs for multi-core architectures

- Prize winner at GECCO *Humies* competition

  - Human Competitive results from Evolutionary Computation tools

Why is parallel programming difficult?

- “If we simply added more than 16 cores, we would get diminishing returns, because the threads and data traffic would not be used properly, so the cores get in the way of each other. It’s like having too many cooks in the kitchen.”
  - *Jerry Bautista*, director of Intel’s tera-scale research program.
- Thread scheduling, synchronization, locking and optimizing the parallelism, etc.
- Efficient parallel programming requires (highly skilled!) human expertise
- ***Automatic Native Parallel Code Generation!***

Recursive Test Problems

#	Problem	Type		Local Variables	Range
		Input	Output		
1	Sum-of-N	int	int	3	[1, 1000]
2	Factorial	int	unsigned long long	3	[1, 60]
3	Fibonacci	int	unsigned long long	3	[1, 60]
4	Binary-Sum	int [], int, int	int	2	[1, 1000]
5	Reverse	int [], int, int	void	2	[1, 1000]
6	Quicksort	int [], int, int	void	3	[1, 1000]

Why Recursion? – *Easy to express but takes longer to execute.*

Recursive Test Problems

#	Problem	Type		Local Variables	Range
		Input	Output		
1	Sum-of-N	int	int	3	[1, 1000]
2	Factorial	int	unsigned long long	3	[1, 60]
3	**Fibonacci**	**int**	**unsigned long long**	**3**	**[1, 60]**
4	Binary-Sum	int [], int, int	int	2	[1, 1000]
5	Reverse	int [], int, int	void	2	[1, 1000]
6	Quicksort	int [], int, int	void	3	[1, 1000]

Why Recursion? – *Easy to express but takes longer to execute.*

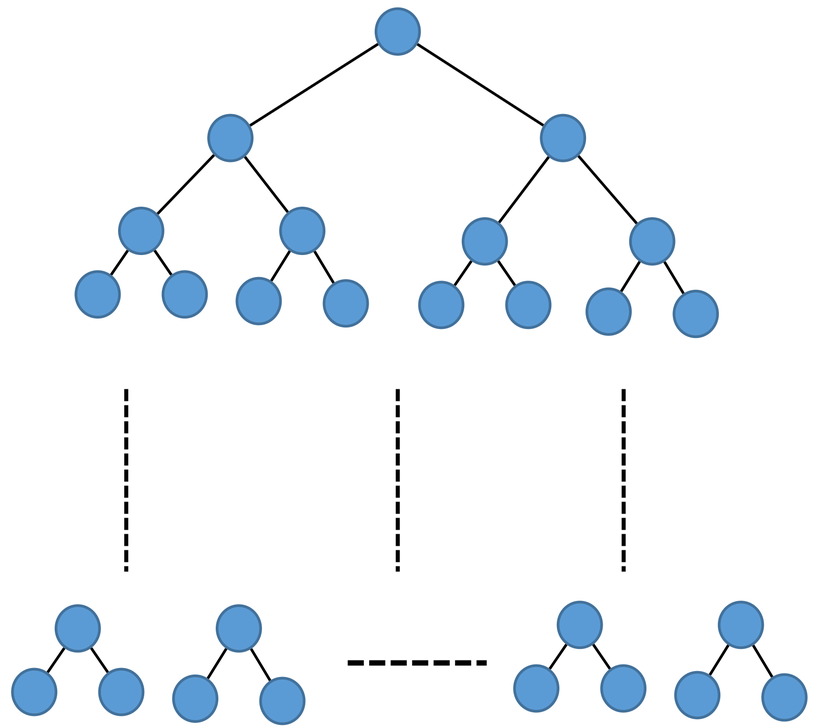
Human generated code

Human Program

```
int i, j;

if (n <= 2)  {
        return n;
}
else
{
        #pragma omp parallel
sections  \
        shared(i, j)
        {
        #pragma omp section
        {
                i = fib(n−1);
        }
        #pragma omp section
        {
        j = fib(n−2);
                }
        }
        return (i+j);
}
```

n



Maximizing Parallelism

$2^{(n+1)}$ threads

Grammar

- Create OpenMP function using #pragmas
- Create parallel for loop
  - Private and shared variables
- Create sequential code
  - Recursive calls

Fitness Function

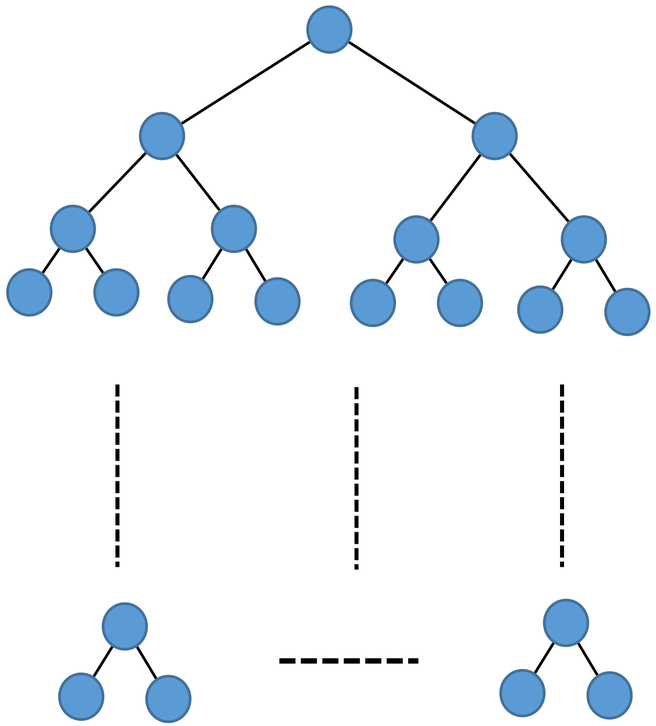
- Ability to generate Fibonacci numbers
  - 60 cases; calculate Mean Square Error across all
- Ability to avoid infinite recursion
- Minimize execution time

Optimizing Parallelism

MCGE-II Program

```
if (n <= 2)     {
        temp = n;
res += temp;
}
else if (n >= C)     {
res = temp+fib(n-1)+fib(n-2);
}
else            {
        #pragma omp parallel sections \
        private (a) shared(n, temp, res)
        {
        #pragma omp section
        {
                a = fib(n−1);
                #pragma omp atomic
                        res += temp+a;
        }
        #pragma omp section
        {
        a = fib(n−2);
        #pragma omp atomic
                res += temp+a;
                }
        }
} return res;
```

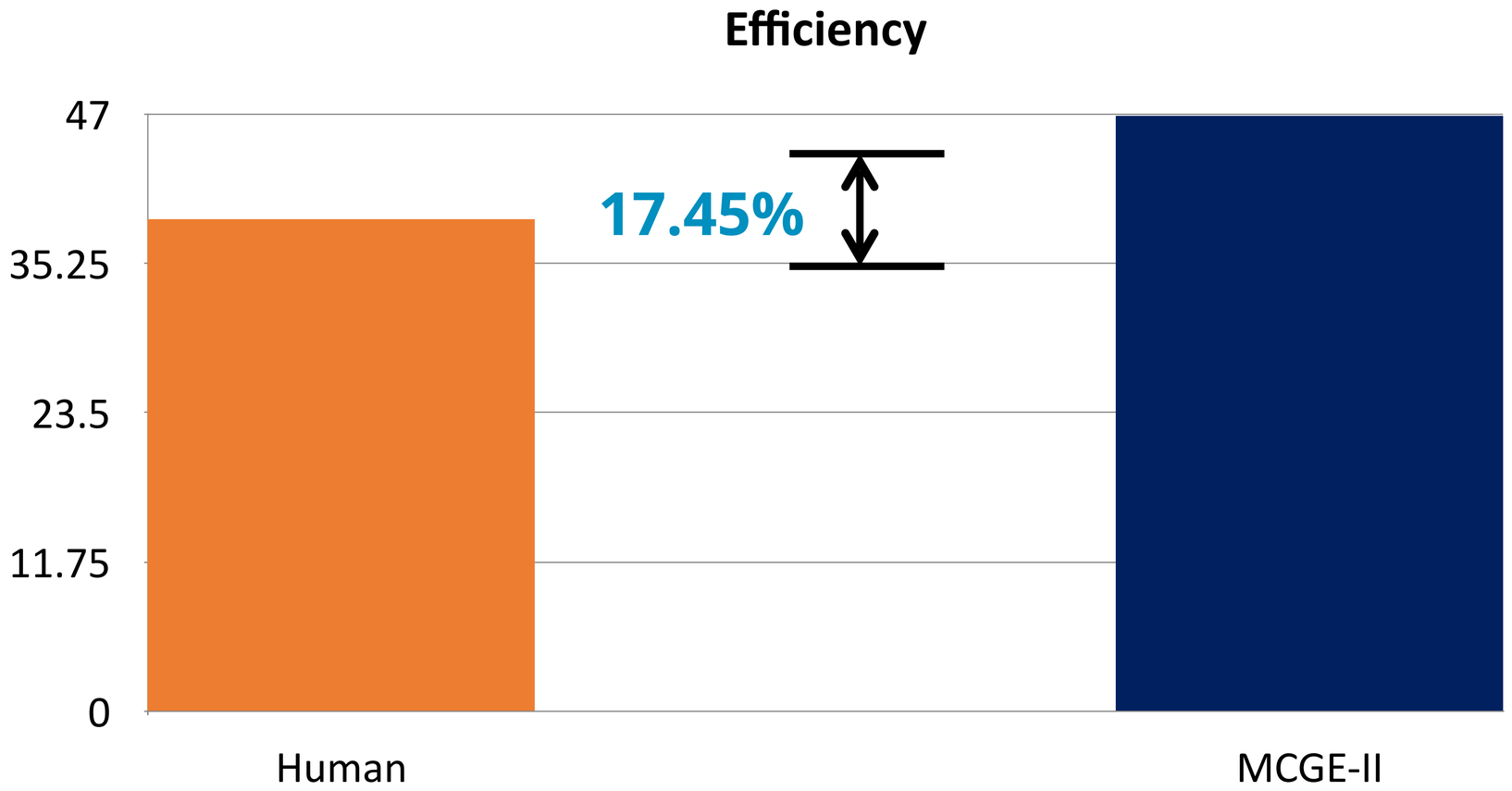
C



Optimal Parallelism

$2^{(c+1)}$ threads

Human Competitive



Locks

- Locks guarantee the correctness of a program by synchronising access to shared variables
  - **But**, they can degrade the performance by causing delays
- Difficult even for experts to write *lock-free* programs
- Automatic lock-free programming

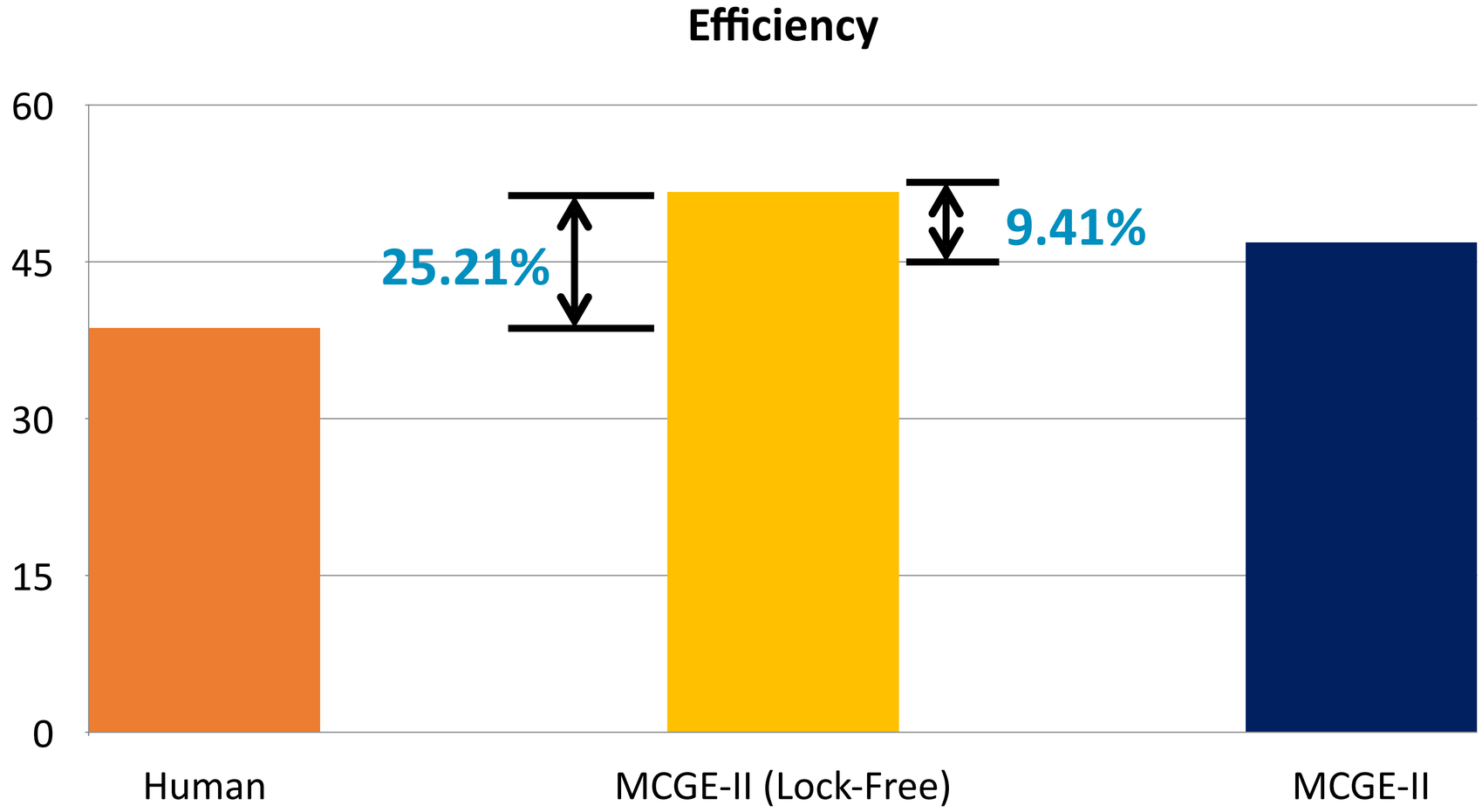
Grammar Modification

- Allow GE to use GCC atomic primitives
  - __sync_fetch_add, __sync_fetch_sub, etc.
- Incorporate use of pointers to share variables
- GE chooses whether or not to use these
  - Only feedback is from the fitness function

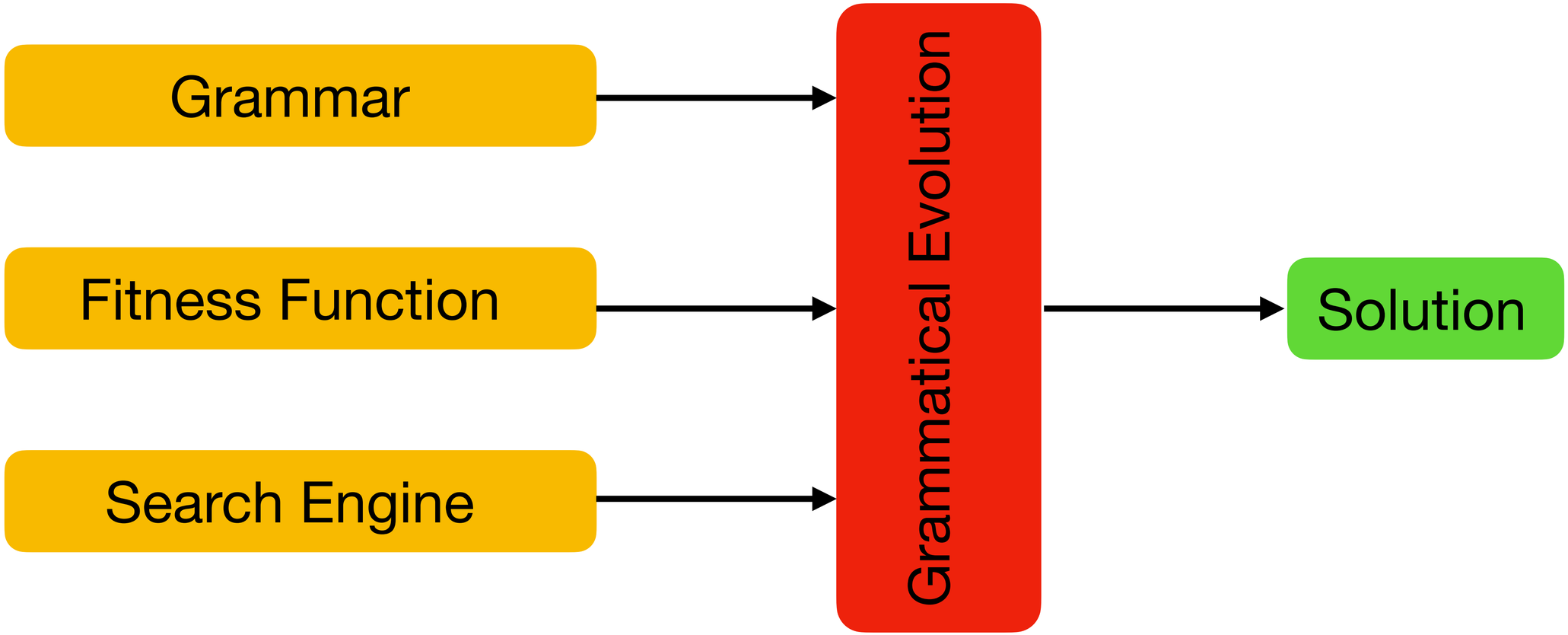
Fitness Function

- Exactly as before!

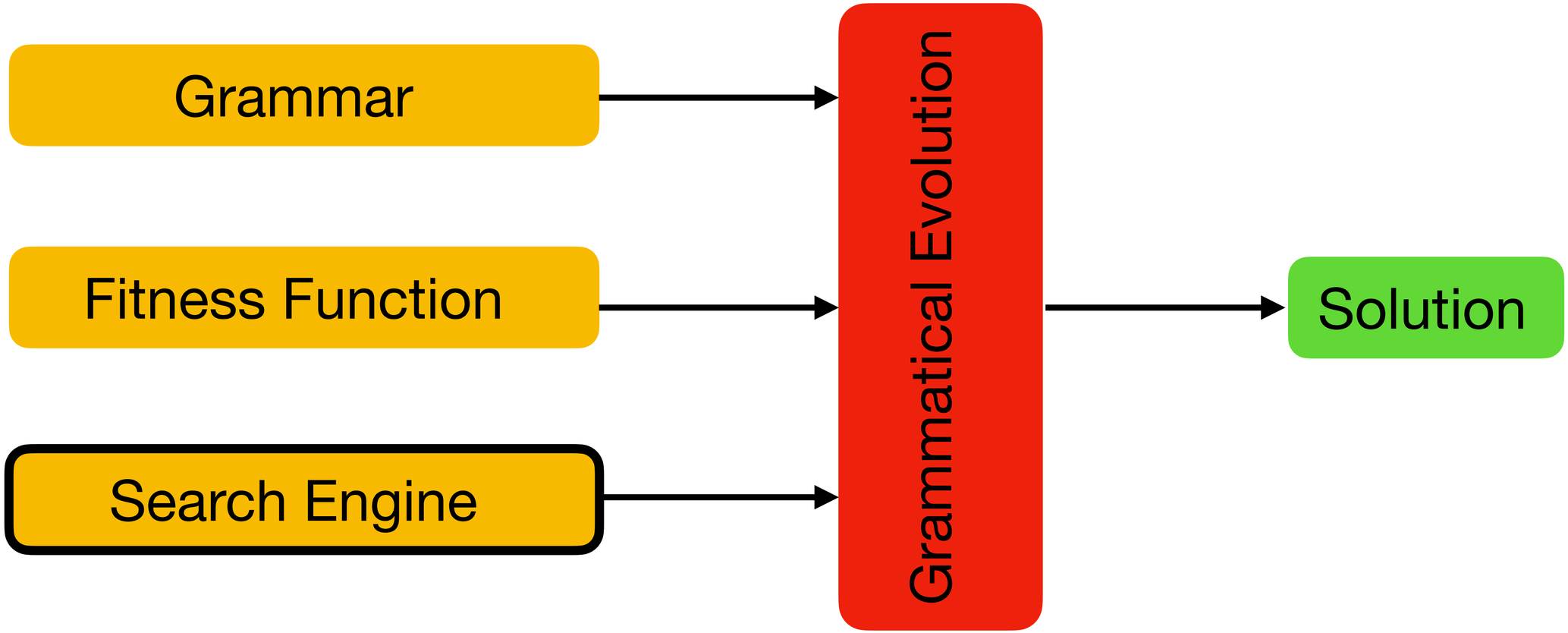
Lock-free results



Overview


Grammar
Fitness Function
Search Engine
Grammatical Evolution
Solution

Overview


Grammar
Fitness Function
Search Engine
Grammatical Evolution
Solution