Lecture 4: Static Analysis – Notions, Techniques and Formal Methods

Passive Testing Techniques for Communication Protocols

Dr. Jorge López, PhD.
jorgelopezcoronado[at]gmail.com

National Research Tomsk State University

February 17, 2016

OUTLINE

HOMEWORK & LAB INFORMATION

Remove ambiguity from:

Homework & Lab Information

Remove ambiguity from:

- $S \mapsto SS|a|c$; ca has priority over ac

HOMEWORK & LAB INFORMATION

Remove ambiguity from:

- $S \mapsto SS|a|c$; ca has priority over ac
  - $S \mapsto a|c|aS|S2\ S|S2$
  - $S2 \mapsto ca$

Homework & Lab Information

Remove ambiguity from:

- $S \mapsto SS|a|c$; ca has priority over ac
  - $S \mapsto a|c|aS|S2\ S|S2$
  - $S2 \mapsto ca$

Lab Information

Homework & Lab Information

Remove ambiguity from:

- $S \mapsto SS|a|c$; ca has priority over ac
  - $S \mapsto a|c|aS|S2\ S|S2$
  - $S2 \mapsto ca$

Lab Information

- Given the hosts endianess no guarantee the correct transmission of data from the hosts to network functions to transform data from the hosts representation to network and vice-versa are necessary. Use the functions:
  - `ntohs // network to host short (2 bytes)`
  - `htons // host to network short (2 bytes)`
  - `ntohl // network to host long (4 bytes)`
  - `htonl // host to network short (4 bytes)`

An AST as a data structure

An AST

An AST as a data structure

An AST

- Is a **Parse Tree**

AN AST AS A DATA STRUCTURE

An AST

- Is a **Parse Tree**
- A directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure)

AN AST AS A DATA STRUCTURE

An AST

- Is a **Parse Tree**
- A directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure)
- **Without irrelevant symbols!**

AN AST AS A DATA STRUCTURE

An AST

- Is a **Parse Tree**
- A directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure)
- **Without irrelevant symbols!**

As a data structure

AN AST AS A DATA STRUCTURE

An AST

- Is a **Parse Tree**
- A directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure)
- **Without irrelevant symbols!**

As a data structure

- Each non-terminal is a data structure with *pointers* to other terminal and non-terminals data structures according to the non-terminal production rules of the [abstract] grammar

AN AST AS A DATA STRUCTURE

An AST

- Is a **Parse Tree**
- A directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure)
- **Without irrelevant symbols!**

As a data structure

- Each non-terminal is a data structure with *pointers* to other terminal and non-terminals data structures according to the non-terminal production rules of the [abstract] grammar
- Terminal symbols **might** be represented as primitive data types

AN AST AS A DATA STRUCTURE (CONT.)

AN AST AS A DATA STRUCTURE (CONT.)

(Hacked) Grammar

- $mop = /|*$
- $aop = +|-$
- $int = [0-9]^+$,
  $opar = ($,$cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$
4. $Term \mapsto Term\ mop\ Num$
5. $Term \mapsto Num$
6. $Num \mapsto int$
7. $Num \mapsto opar\ Expr\ cpar$

AN AST AS A DATA STRUCTURE (CONT.)

(Hacked) Grammar

- $mop = /|*$
- $aop = +|-$
- $int = [0-9]^+$,
  $opar = ($, $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$
4. $Term \mapsto Term\ mop\ Num$
5. $Term \mapsto Num$
6. $Num \mapsto int$
7. $Num \mapsto opar\ Expr\ cpar$

Data structure

AN AST AS A DATA STRUCTURE (CONT.)

(Hacked) Grammar

- $mop = /|*$
- $aop = +|-$
- $int = [0-9]^{+}$, $opar = ($, $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$
4. $Term \mapsto Term\ mop\ Num$
5. $Term \mapsto Num$
6. $Num \mapsto int$
7. $Num \mapsto opar\ Expr\ cpar$

Data structure

```
typedef enum optypetag
{
   VALUE,
   MUL,
   DIV,
   PLUS,
   MIN
} optype;

typedef struct exprtag
{
   optype type; // type of operation

   int value; // for VALUE
   struct exprtag *left;
   struct exprtag *right;
} expr;
```

AN AST AS A DATA STRUCTURE (CONT. 2)

Keep in mind that:

AN AST AS A DATA STRUCTURE (CONT. 2)

Keep in mind that:

- Operations to create AST nodes are important

An AST as a data structure (cont. 2)

Keep in mind that:

- Operations to create AST nodes are important
- Operations to delete AST nodes are important

AN AST AS A DATA STRUCTURE (CONT. 2)

Keep in mind that:

- Operations to create AST nodes are important
- Operations to delete AST nodes are important

What will be the root of the AST data structure for the general case?

An AST as a data structure (cont. 2)

Keep in mind that:

- Operations to create AST nodes are important
- Operations to delete AST nodes are important

What will be the root of the AST data structure for the general case?

- The data structure associated to the CFG start symbol!

Let's see how we use the AST then…

VISITING THE AST

Everybody knows that transversing a tree-like structure is done via

VISITING THE AST

Everybody knows that transversing a tree-like structure is done via recursion…

Visiting the AST

Everybody knows that transversing a tree-like structure is done via recursion…

- There are a few differences with an AST, those are:

VISITING THE AST

Everybody knows that transversing a tree-like structure is done via recursion…

- There are a few differences with an AST, those are:
  - Depending on the data structure design of the AST, the transversing could be different

VISITING THE AST

Everybody knows that transversing a tree-like structure is done via recursion…

- There are a few differences with an AST, those are:
  - Depending on the data structure design of the AST, the transversing could be different
  - An AST has different data structures for different non-terminal symbols of the grammar, the tree transversal algorithm needs to be aware of those types and perform the transversing accordingly

Visiting the AST

Everybody knows that transversing a tree-like structure is done via recursion…

- There are a few differences with an AST, those are:
  - Depending on the data structure design of the AST, the transversing could be different
  - An AST has different data structures for different non-terminal symbols of the grammar, the tree transversal algorithm needs to be aware of those types and perform the transversing accordingly

Let's take a look of each case for the grammar $P \mapsto P\ P|(P)|()$

Visiting the AST (cont.)

Visiting the AST (cont.)

Data structure design 1, AST of $((())())()$

Visiting the AST (cont.)

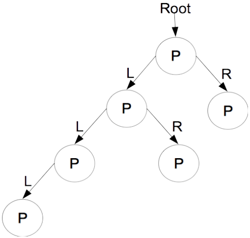
Data structure design 1, AST of ((())())()

```
typedef struct partag
{
        struct partag *L;
        struct partag *R;
}P;
```

Visiting the AST (cont.)

Data structure design 1, AST of ((())())()

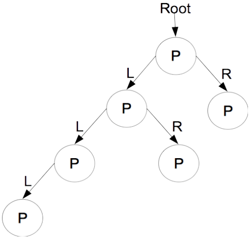
```
typedef struct partag
{
        struct partag *L;
        struct partag *R;
}P;
```



VISITING THE AST (CONT.)

Data structure design 1, AST of ((())())()

```
typedef struct partag
{
        struct partag *L;
        struct partag *R;
}P;
```

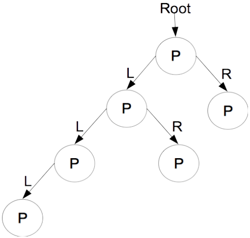


Data structure design 2, AST of (((())()))

Visiting the AST (cont.)

Data structure design 1, AST of ((())())()

```
typedef struct partag
{
        struct partag *L;
        struct partag *R;
}P;
```



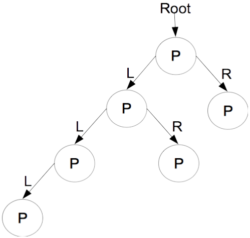
Data structure design 2, AST of ((())()))

```
typedef enum ttag{
        PLIST,
        PAR
}astType;
typedef struct listtag{
        list L; //initial
}plist;
typedef struct partag{
        struct partag *in;
        list L; //null in case of in
}P;
```

VISITING THE AST (CONT.)

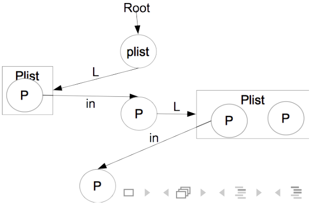
Data structure design 1, AST of ((())())()

```
typedef struct partag
{
        struct partag *L;
        struct partag *R;
}P;
```



Data structure design 2, AST of ((())()))

```
typedef enum ttag{
        PLIST,
        PAR
}astType;
typedef struct listtag{
        list L; //initial
}plist;
typedef struct partag{
        struct partag *in;
        list L; //null in case of in
}P;
```



VISITING THE AST (CONT. 2)

VISITING THE AST (CONT. 2)

Single recursive visitor

```
void visitAST(void *astNode, astType t){
   if(t == PLIST){
      ... //list instructions
      foreach p = (list*)astNode
         visitAST(p, PAR);
   }
   else if(t == PAR){
      ... // P instructions
      if(p->in != NULL)
         visitAST(p->in, PAR);
      else
         visitAST(p->L, PLIST);
   }
   ...
}
```

Visiting the AST (cont. 2)

Single recursive visitor

```
void visitAST(void *astNode, astType t){
   if(t == PLIST){
      ... //list instructions
      foreach p = (list*)astNode
         visitAST(p, PAR);
   }
   else if(t == PAR){
      ... // P instructions
      if(p->in != NULL)
         visitAST(p->in, PAR);
      else
         visitAST(p->L, PLIST);
   }
   ...
}
```

Multiple *recursive* visitors

```
void visitAST(list *astRoot){
   visitASTList(astRoot);
}
void visitASTList(list L){
   ... //list instructions
   foreach p = (list*)astNode
      visitASTPars(p);
}

void visitASTPars(P p){
   ... // P instructions
   if(p->in != NULL)
      visitASTPar(p->in);
   else
      visitASTList(p->L);
}
...
```

Visiting the AST (cont. 3)

From a simple AST visiting function you can

VISITING THE AST (CONT. 3)

From a simple AST visiting function you can

- Print the structured source code

VISITING THE AST (CONT. 3)

From a simple AST visiting function you can

- Print the structured source code
- look for a specific function call(!)

Visiting the AST (cont. 3)

From a simple AST visiting function you can

- Print the structured source code
- look for a specific function call(!)
- Check for simple overflows

Visiting the AST (cont. 3)

From a simple AST visiting function you can

- Print the structured source code
- look for a specific function call(!)
- Check for simple overflows
- Check for certain divisions by 0

VISITING THE AST (CONT. 3)

From a simple AST visiting function you can

- Print the structured source code
- look for a specific function call(!)
- Check for simple overflows
- Check for certain divisions by 0

From a simple AST visiting function you can build your very first static program analyzer

Static Program Analysis 101

Static Program Analysis

We can look for anything…

Static Program Analysis

We can look for anything…

- Anything which is not *functional* related

Static Program Analysis

We can look for anything. . .

- Anything which is not *functional* related
- Usually we check properties

Static Program Analysis

We can look for anything…

- Anything which is not *functional* related
- Usually we check properties
  - For instance, for the simple grammar $P \mapsto P\ P | (P) | ()$, ensure that on the even parenthesis list elements, the depth level is not more than 2. $(())()()((()))$ violates the property, but $((((()))))()$ is does not (i.e., is valid)

STATIC PROGRAM ANALYSIS

We can look for anything…

- Anything which is not *functional* related
- Usually we check properties
  - For instance, for the simple grammar $P \mapsto P\ P|(P)|()$, ensure that on the even parenthesis list elements, the depth level is not more than 2. $(())()()((()))$ violates the property, but $(((())))()$ is does not (i.e., is valid)
  - Ensure that starting from the N-th element of the list at least m levels are found, for example for $N = 2, m = 2$, $()()()$ violates the property, and $()(())((()))$ does not

Static Program Analysis

We can look for anything...

- Anything which is not *functional* related
- Usually we check properties
  - For instance, for the simple grammar $P \mapsto P\ P|(P)|()$, ensure that on the even parenthesis list elements, the depth level is not more than 2. $(())()()((()))$ violates the property, but $((()))()$ is does not (i.e., is valid)
  - Ensure that starting from the N-th element of the list at least m levels are found, for example for $N = 2, m = 2$, $()()()$ violates the property, and $()(())((()))$ does not
  - Ensure that in starting from the N-th element of the list, no sub-lists are found...

Static Program Analysis

We can look for anything…

- Anything which is not *functional* related
- Usually we check properties
  - For instance, for the simple grammar $P \mapsto P\ P|(P)|()$, ensure that on the even parenthesis list elements, the depth level is not more than 2. $(())()()((()))$ violates the property, but $((((()))))()$ is does not (i.e., is valid)
  - Ensure that starting from the N-th element of the list at least m levels are found, for example for $N = 2, m = 2$, $()()()$ violates the property, and $()(())((()))$ does not
  - Ensure that in starting from the N-th element of the list, no sub-lists are found…
  - Imagine the possibilities for a more complex grammar

STATIC PROGRAM ANALYSIS

We can look for anything...

- Anything which is not *functional* related
- Usually we check properties
  - For instance, for the simple grammar $P \mapsto P\,P|(P)|()$, ensure that on the even parenthesis list elements, the depth level is not more than 2. $(())()()((()))$ violates the property, but $((((()))))()$ is does not (i.e., is valid)
  - Ensure that starting from the N-th element of the list at least m levels are found, for example for $N = 2, m = 2$, $()()()$ violates the property, and $()(())((()))$ does not
  - Ensure that in starting from the N-th element of the list, no sub-lists are found...
  - Imagine the possibilities for a more complex grammar
- Let's look at more real-world examples...

Static Analysis for security properties

Security issues in code

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks

STATIC ANALYSIS FOR SECURITY PROPERTIES

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks
  - SQL injection (SQLI) attacks

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks
  - SQL injection (SQLI) attacks
  - Buffer overflow attacks

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks
  - SQL injection (SQLI) attacks
  - Buffer overflow attacks

XSS

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks
  - SQL injection (SQLI) attacks
  - Buffer overflow attacks

XSS

- Essentially works by supplying other users data which can lead to insecure actions, for example visiting a link or executing some code (javascript), or even adding a sub-site filled with publicity or others

Static Analysis for security properties

Security issues in code

- Many problems arise from not checking the user input
  - Cross-Site Scripting (XSS) attacks
  - SQL injection (SQLI) attacks
  - Buffer overflow attacks

XSS

- Essentially works by supplying other users data which can lead to insecure actions, for example visiting a link or executing some code (javascript), or even adding a sub-site filled with publicity or others
- An easy example is to allow a user in a forum to insert a comment and then display it. If the comment contains HTML and it is displayed as-is, the attacker would successfully execute the attack on users seeing that page

Static Analysis for security properties II

SQLI

Static Analysis for security properties II

SQLI

- Essentially works by supplying data to the database that the database will execute, for example, the user inputs a search criteria, and the database looks for the users matching that search criteria:

  ```
  SELECT * from users where name='$CRIT';
  ```

  What if entered criteria is:

  ```
  a'; DROP TABLE users
  ```

Static Analysis for security properties II

SQLI

- Essentially works by supplying data to the database that the database will execute, for example, the user inputs a search criteria, and the database looks for the users matching that search criteria:

  ```
  SELECT * from users where name='$CRIT';
  ```

  What if entered criteria is:

  ```
  a'; DROP TABLE users
  ```

How to protect using static analysis?

Static Analysis for security properties II

SQLI

- Essentially works by supplying data to the database that the database will execute, for example, the user inputs a search criteria, and the database looks for the users matching that search criteria:

  ```
  SELECT * from users where name='$CRIT';
  ```

  What if entered criteria is:

  ```
  a'; DROP TABLE users
  ```

How to protect using static analysis?

- A very simple approach is to guarantee that a **sanitization** function is called before the storing or displaying of the input. Many languages provide such functions built-in, e.g., PHP provides htmlspecialchars() function

STATIC ANALYSIS FOR SECURITY PROPERTIES III

Buffer Overflow

STATIC ANALYSIS FOR SECURITY PROPERTIES III

Buffer Overflow

- The canonical example:

```
#include <string.h>
#define BUFFSIZE 100
void load (char *userdata){
   char  buff[BUFFSIZE];
   strcpy(buff, userdata);  //not good
}

int main (int argc, char **argv){
   load(argv[1]);
   ...
}
```

Static Analysis for security properties III

Buffer Overflow

- The canonical example:

```
#include <string.h>
#define BUFFSIZE 100
void load (char *userdata){
   char  buff[BUFFSIZE];
   strcpy(buff, userdata);  //not good
}

int main (int argc, char **argv){
   load(argv[1]);
   ...
}
```

- A string which is lager than BUFFSIZE will write inside the memory space of the function load, potentially overwriting the return address

STATIC ANALYSIS FOR SECURITY PROPERTIES III

Buffer Overflow

- The canonical example:

```
#include <string.h>
#define BUFFSIZE 100
void load (char *userdata){
   char  buff[BUFFSIZE];
   strcpy(buff, userdata);  //not good
}

int main (int argc, char **argv){
   load(argv[1]);
   ...
}
```

- A string which is lager than BUFFSIZE will write inside the memory space of the function load, potentially overwriting the return address
- A string which contains code and the memory address of this code in the position of the return address will do the trick

Static Analysis for security properties IV

Buffer Overflow

Static Analysis for security properties IV

Buffer Overflow

- If the program with a potentially susceptible stack overflow runs with admin privileges, consequences can be devastating

STATIC ANALYSIS FOR SECURITY PROPERTIES IV

Buffer Overflow

- If the program with a potentially susceptible stack overflow runs with admin privileges, consequences can be devastating

Many approaches proposed to detect such an attack

Static Analysis for security properties IV

Buffer Overflow

- If the program with a potentially susceptible stack overflow runs with admin privileges, consequences can be devastating

Many approaches proposed to detect such an attack

- Some rely on guaranteeing calling some specific functions (sometimes replaced safe functions)

STATIC ANALYSIS FOR SECURITY PROPERTIES IV

Buffer Overflow

- If the program with a potentially susceptible stack overflow runs with admin privileges, consequences can be devastating

Many approaches proposed to detect such an attack

- Some rely on guaranteeing calling some specific functions (sometimes replaced safe functions)
- Others propose a mathematical approach of calculating the bound automatically

STATIC ANALYSIS FOR SECURITY PROPERTIES IV

Buffer Overflow

- If the program with a potentially susceptible stack overflow runs with admin privileges, consequences can be devastating

Many approaches proposed to detect such an attack

- Some rely on guaranteeing calling some specific functions (sometimes replaced safe functions)
- Others propose a mathematical approach of calculating the bound automatically
- Any new approach for such a problem will be welcomed!

Static Analysis 101 – Final Remarks

STATIC ANALYSIS 101 – FINAL REMARKS

- Static Analysis programs can be complex and can try guarantee <<**generic**>> properties, However, new analysis even of single properties can be important and incorporated to known solutions

Static Analysis 101 – Final Remarks

- Static Analysis programs can be complex and can try guarantee <<**generic**>> properties, However, new analysis even of single properties can be important and incorporated to known solutions
- Providing false-positives is an issue, however, in some cases accepted

Static Analysis 101 – Final Remarks

- Static Analysis programs can be complex and can try guarantee <<**generic**>> properties, However, new analysis even of single properties can be important and incorporated to known solutions
- Providing false-positives is an issue, however, in some cases accepted
- Sound approaches have been proposed, they are based on formal methods, let's get a quick overview of them. . .

Formal Methods Overview

DATA FLOW ANALYSIS

In principle look for data flow and data dependencies, assume the following code:

DATA FLOW ANALYSIS

In principle look for data flow and data dependencies, assume the following code:

```
void func (int x){
   int y = 10;
   int z = 2 + y;

   if(x > 10){
      z=10;
      x = y + 1;
   }
   print(z);
}
```

Data Flow Analysis

In principle look for data flow and data dependencies, assume the following code:

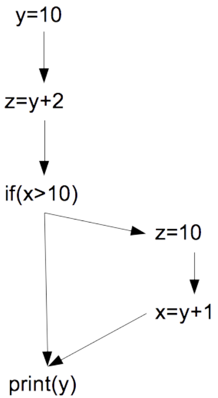
```
void func (int x){
   int y = 10;
   int z = 2 + y;

   if(x > 10){
      z=10;
      x = y + 1;
   }
   print(z);
}
```

What does the flow analysis can help us analyze?

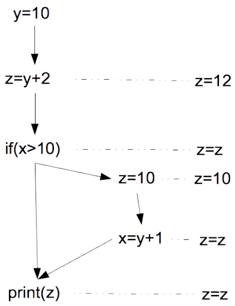
Data Flow Analysis (cont.)

The data flow



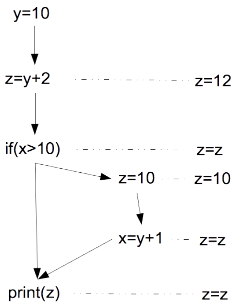
Data Flow Analysis (cont.)

The data flow



DATA FLOW ANALYSIS (CONT.)

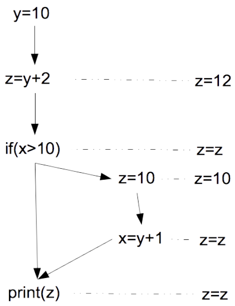
The data flow



Potential values

Data Flow Analysis (cont.)

The data flow

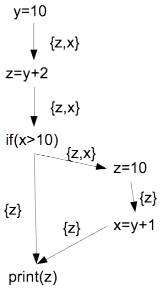


Potential values

- Ranges can be useful, e.g., negatives for array indexing

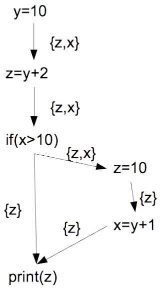
Data Flow Analysis (cont. 2)

The data flow



Data Flow Analysis (cont. 2)

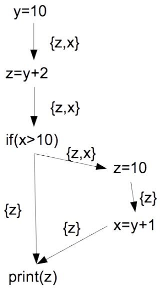
The data flow



Data dependencies

Data Flow Analysis (cont. 2)

The data flow



Data dependencies

- Useful for security testing, for example, or to check useless code

Data Flow Analysis – Final Remarks

DATA FLOW ANALYSIS – FINAL REMARKS

- It can be used for many things, including statement reachability

DATA FLOW ANALYSIS – FINAL REMARKS

- It can be used for many things, including statement reachability
- Popular use for test generation, based on the data flow, get such inputs that will build a test case that will execute ALL statements

DATA FLOW ANALYSIS – FINAL REMARKS

- It can be used for many things, including statement reachability
- Popular use for test generation, based on the data flow, get such inputs that will build a test case that will execute ALL statements
- Many others. . . For the moment, let's try to get our hands on to get a better understanding :)

Lex & Bison $\mapsto$ AST Demo