**Parser Report**

* **Group member : 钟艾彤 (5230000530)**
* **Test :**

**Implemented grammars:**

《program》

<program> => (<stmt>)+

《expression》

<expr> =>  <non-op-expr>  | <binop-expr> | <unop-expr>

<non-op-expr> => <id> | <literal> | <fun-call> | <paren-expr> | <if-expr> | <block> | <construct-expr> | <lambda-expr>

<unop-expr> => <unop> ( <non-op-expr> |  <paren-expr> )

<unop> => not

<binop-expr> => <non-op-expr> <binop> ( <non-op-expr> | <paren-expr> )

<binop> => [+ , - , \* , / , %, < , <= , > , >= ,  ==]

<paren-expr> => \( <expr> \)

<if-expr> =>

   if <expr> : <block> (<else-if>)+  (else: <block>)? end |

   if <expr> : <block> (<else-if>)\*  else: <block> end

‹else-if› =>  else if <expr> : <block>

<fun-call> => <id>  \(  (e  | (<expr> ,)\* <expr> )  \)

<block> => <stmt> | block :  <stmt>\* end

<construct-expr> => <array> | <list>

<list> => \[ list: (e  | (<expr> ,)\* <expr> ) \]

<array> => \[ array: (e  | (<expr> ,)\* <expr> ) \]

<lambda-expr>  => lam <fun-header> : <block> end

《statement》

<stmt> =>  <when-stmt> | <assign-stmt> | <expr> |

            <decl-stmt> | <binding-stmt>

<stmts> => (<stmt>) \*

<assign-stmt> => <id> := <expr>

<when-stmt> => when <expr> : <block> end

<decl-stmt> => <var-decl> | <fun-decl>

<var-decl> => (shadow)? (rec)? var <id> = <expr>

<fun-decl> => (shadow)? (rec)? fun <id> <fun-header> : <stmts> end

<fun-header> => \( < params>  \) (<return-part>)

<return-part> =>  -> <type-name>

<params> => e | <param\_list>

<param\_list> => (<param> ,)\* <param>

<param> =>  (shadow)? <id> [:: <type-name>]

<binding-stmt> => <id> = <expr>

《literal》

<literal> => <boolean> | <string> | <nothing> | <number>

<boolean> =>  true | false

<string> => <single\_quote\_str> | <double\_quote\_str> | <multi\_line\_str>

<single\_quote\_str> => ' ([ASCII]-[' \n] | \')\* '

<double\_quote\_str> => " ([ASCII]-[" \n] | \")\* "

<nothing> => e

<number>  =>  <integer>

<integer> => [- \+]? [0-9]+

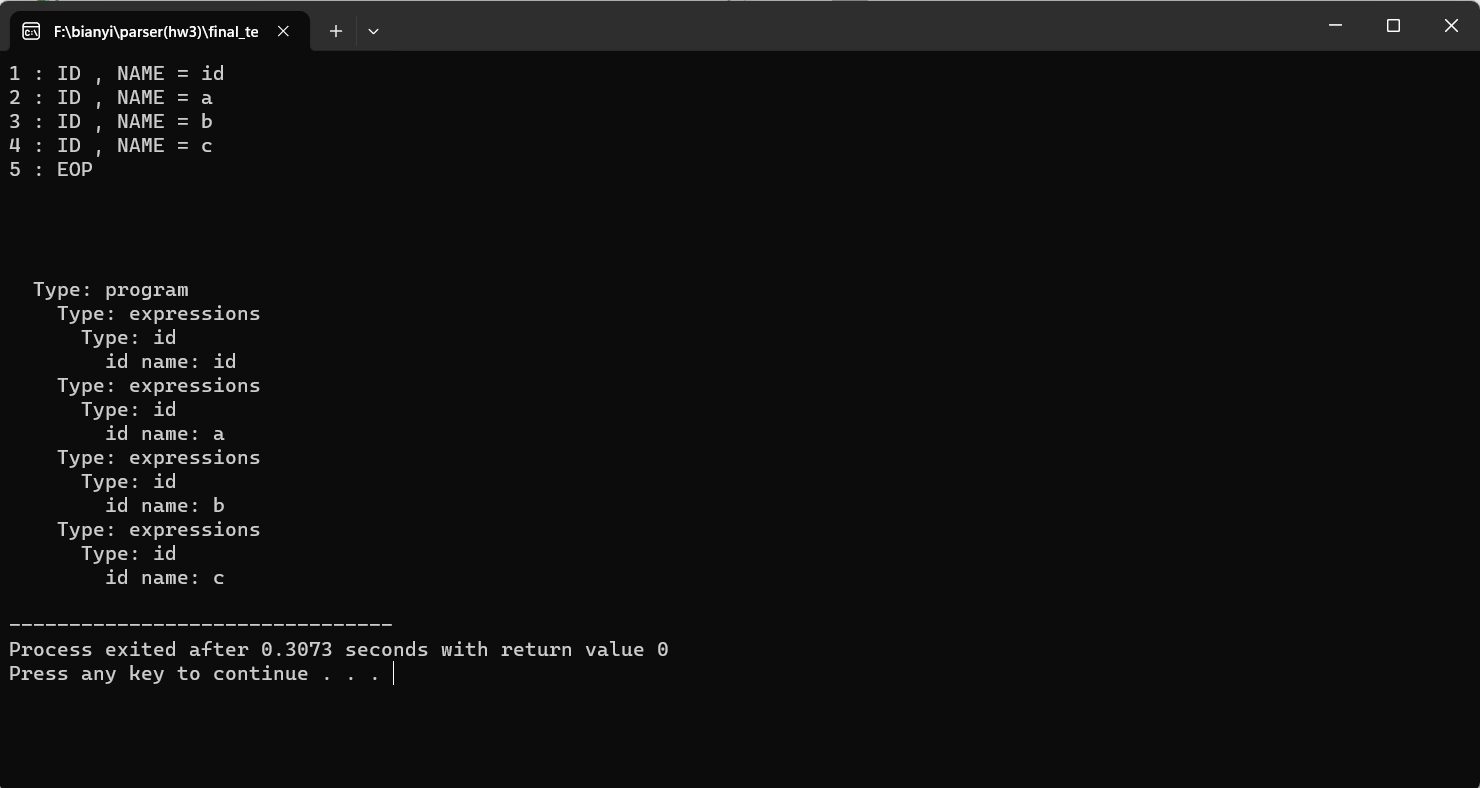
《names》

<id> => (<name>) – (<keyword>)

<type-name> => Any | Number | String | Boolean | Nothing | Function

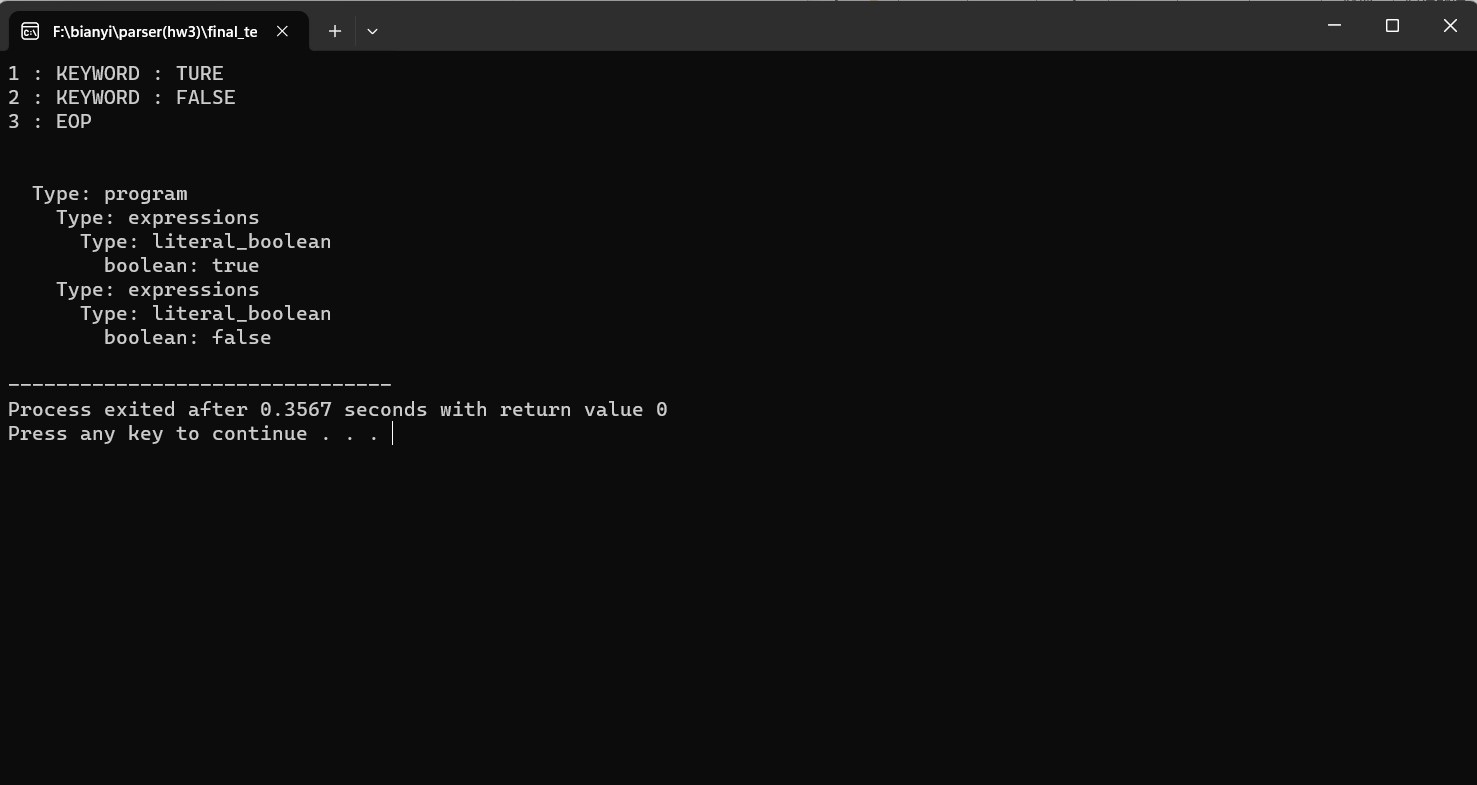
**Test results with simple correct Pyr code:**

**<id> => (<name>) – (<keyword>)**



**<literal> => <boolean> | <string> | <nothing> | <number>**

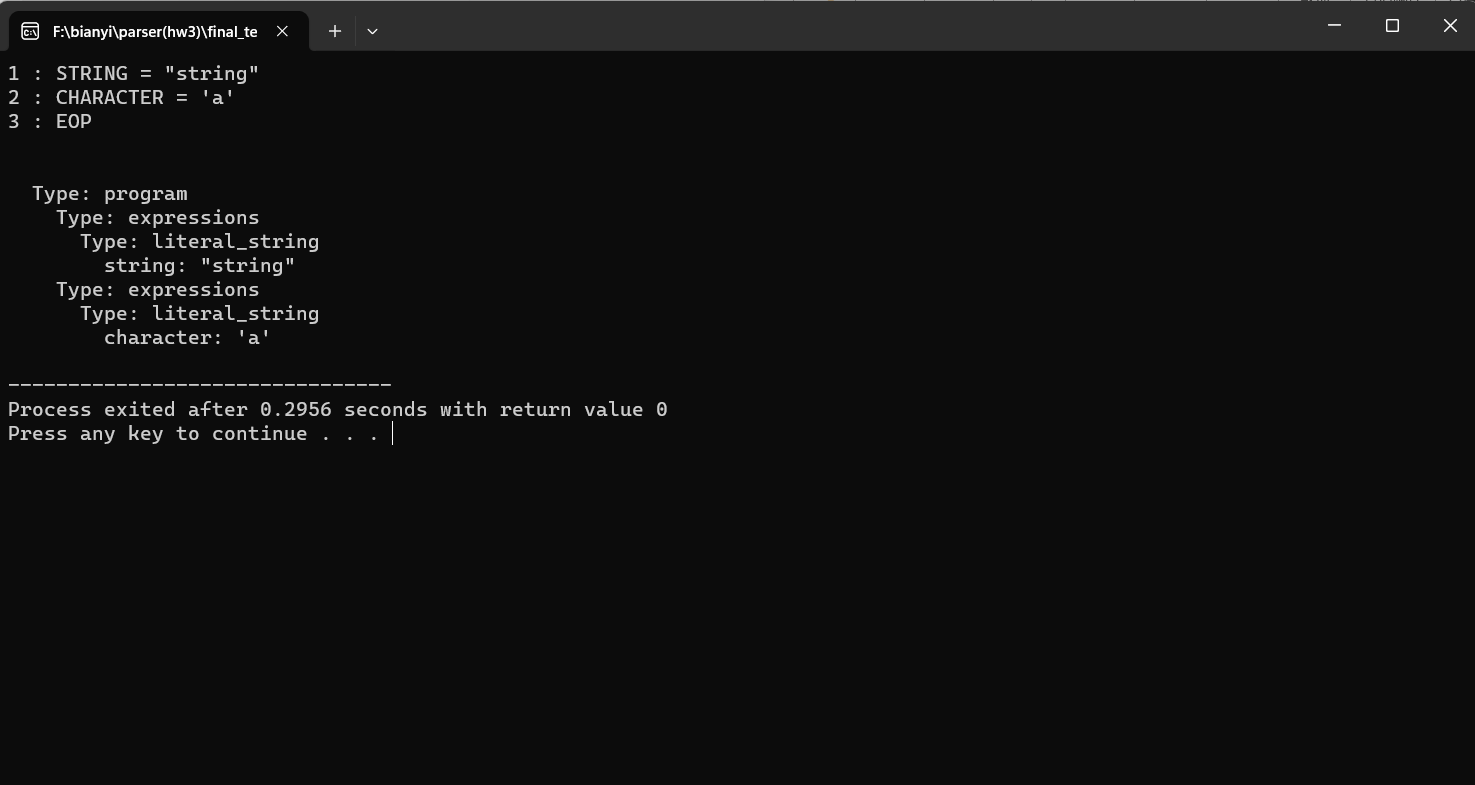
**<boolean> =>  true | false**



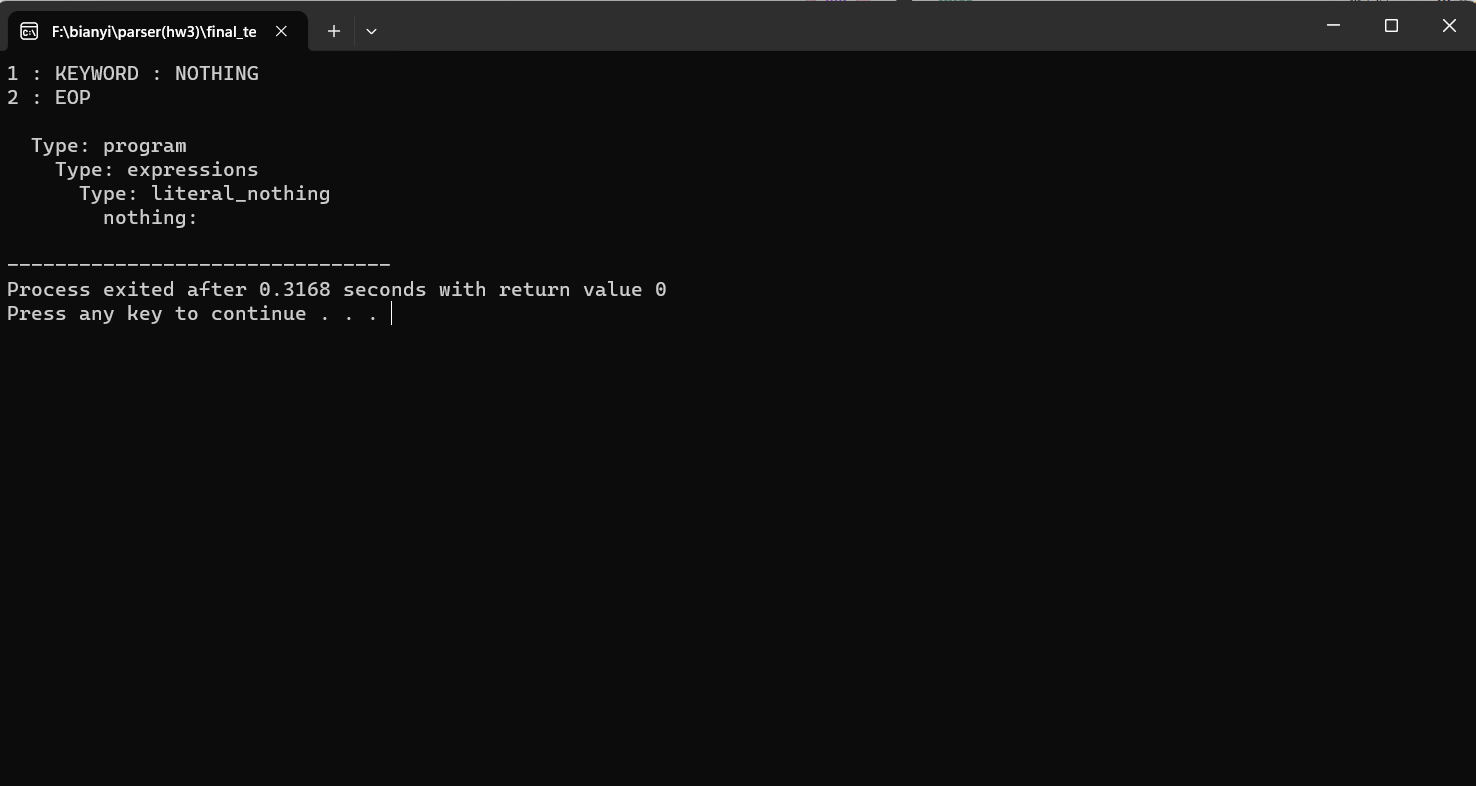
**<string> => <single\_quote\_str> | <double\_quote\_str> | <multi\_line\_str>**

**<single\_quote\_str> => ' ([ASCII]-[' \n] | \')\* '**

**<double\_quote\_str> => " ([ASCII]-[" \n] | \")\* "**

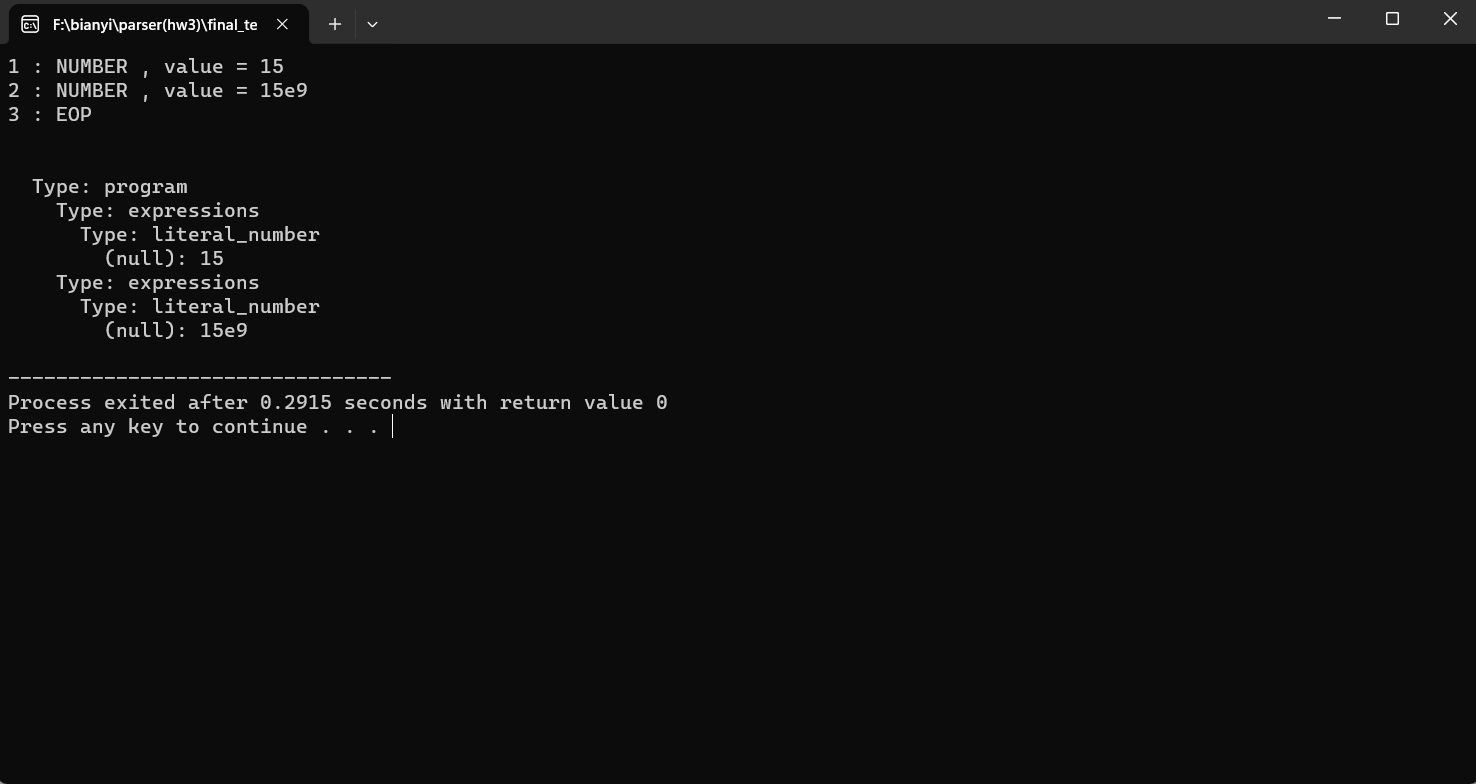


**<nothing> => e**

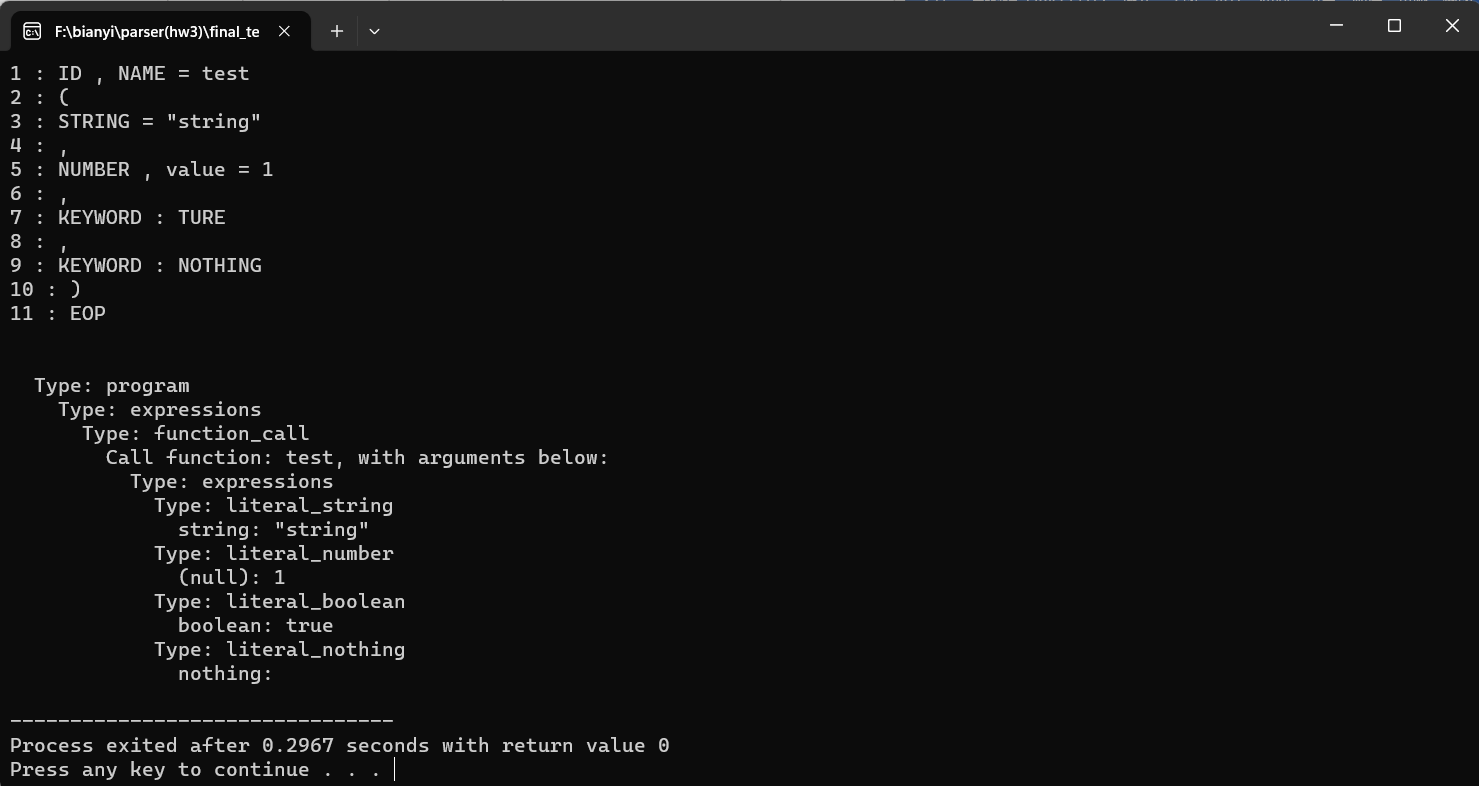


**<number>  =>  <integer>**

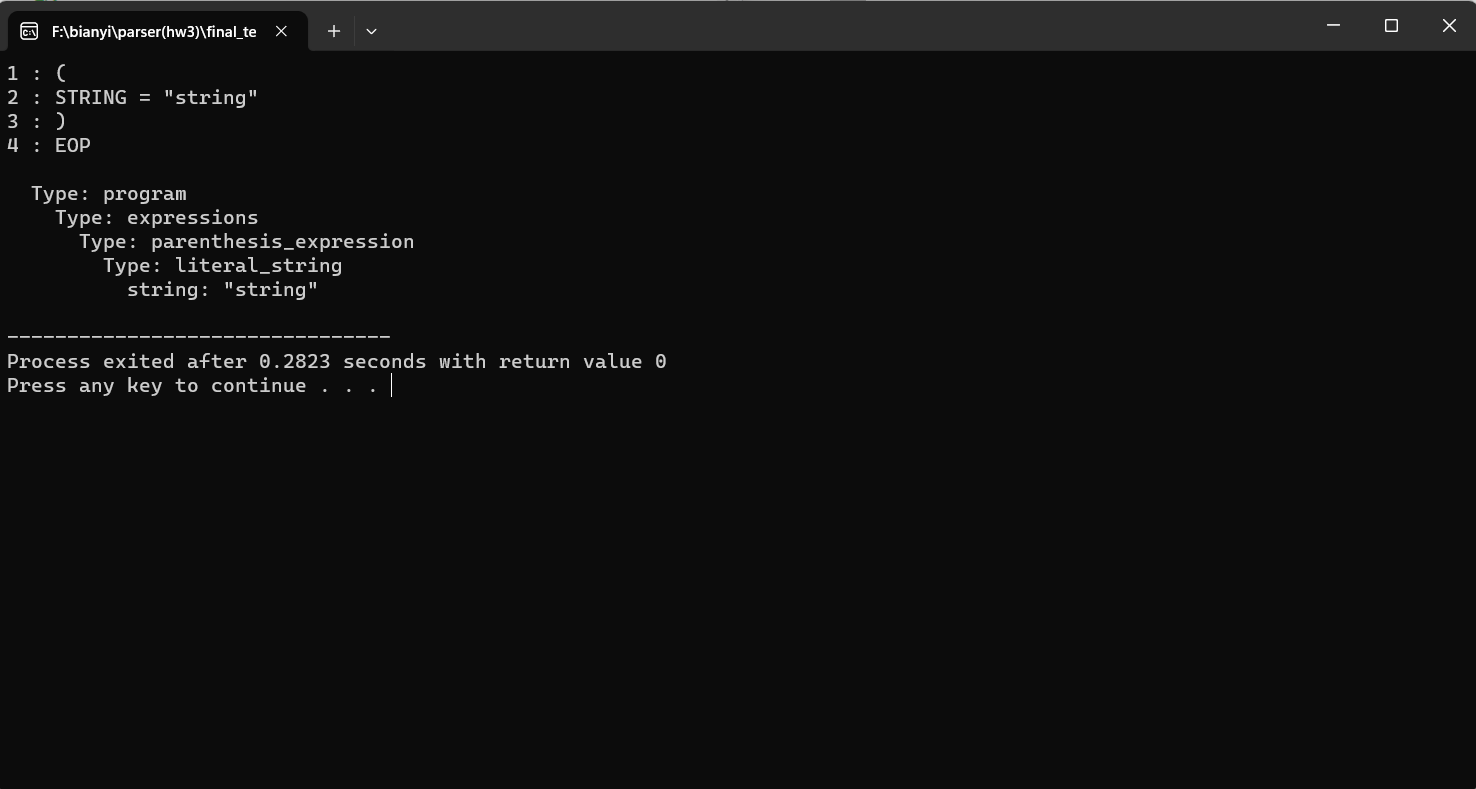
**<integer> => [0-9]+**



**<fun-call> => <id>  \(  (e  | (<expr> ,)\* <expr> )  \)**

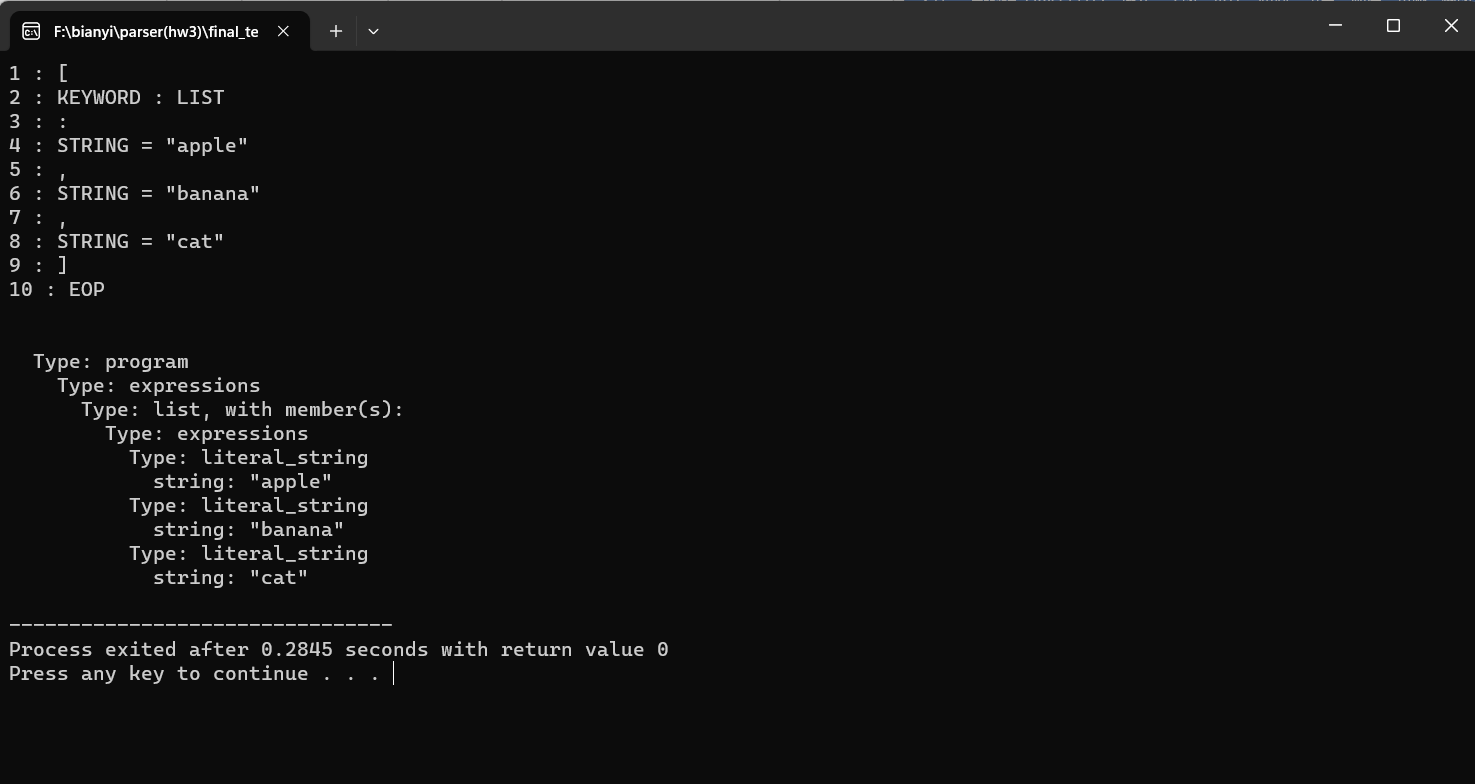


**<paren-expr> => \( <expr> \)**

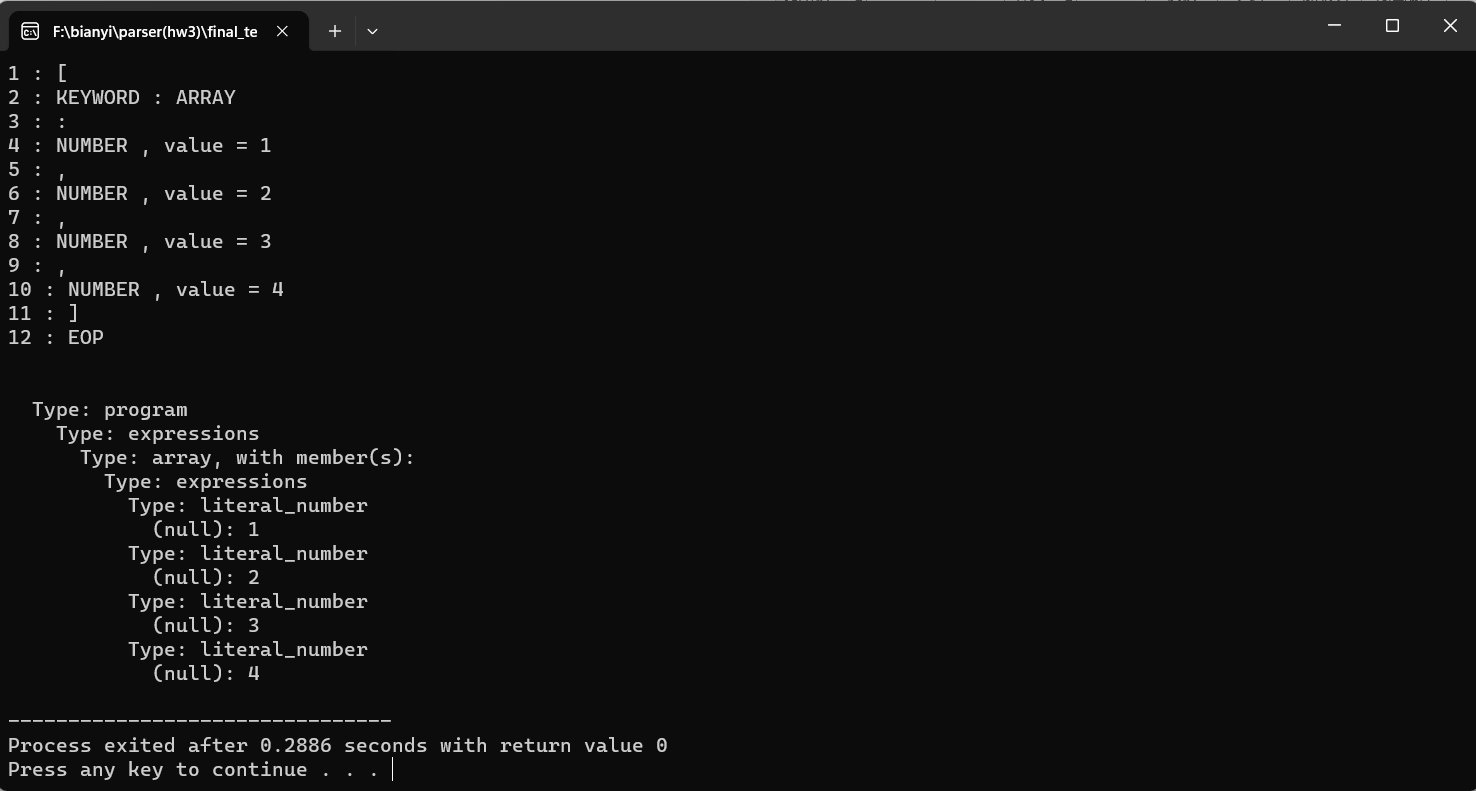


**<construct-expr> => <array> | <list>**

**<list> => \[ list: (e  | (<expr> ,)\* <expr> ) \]**

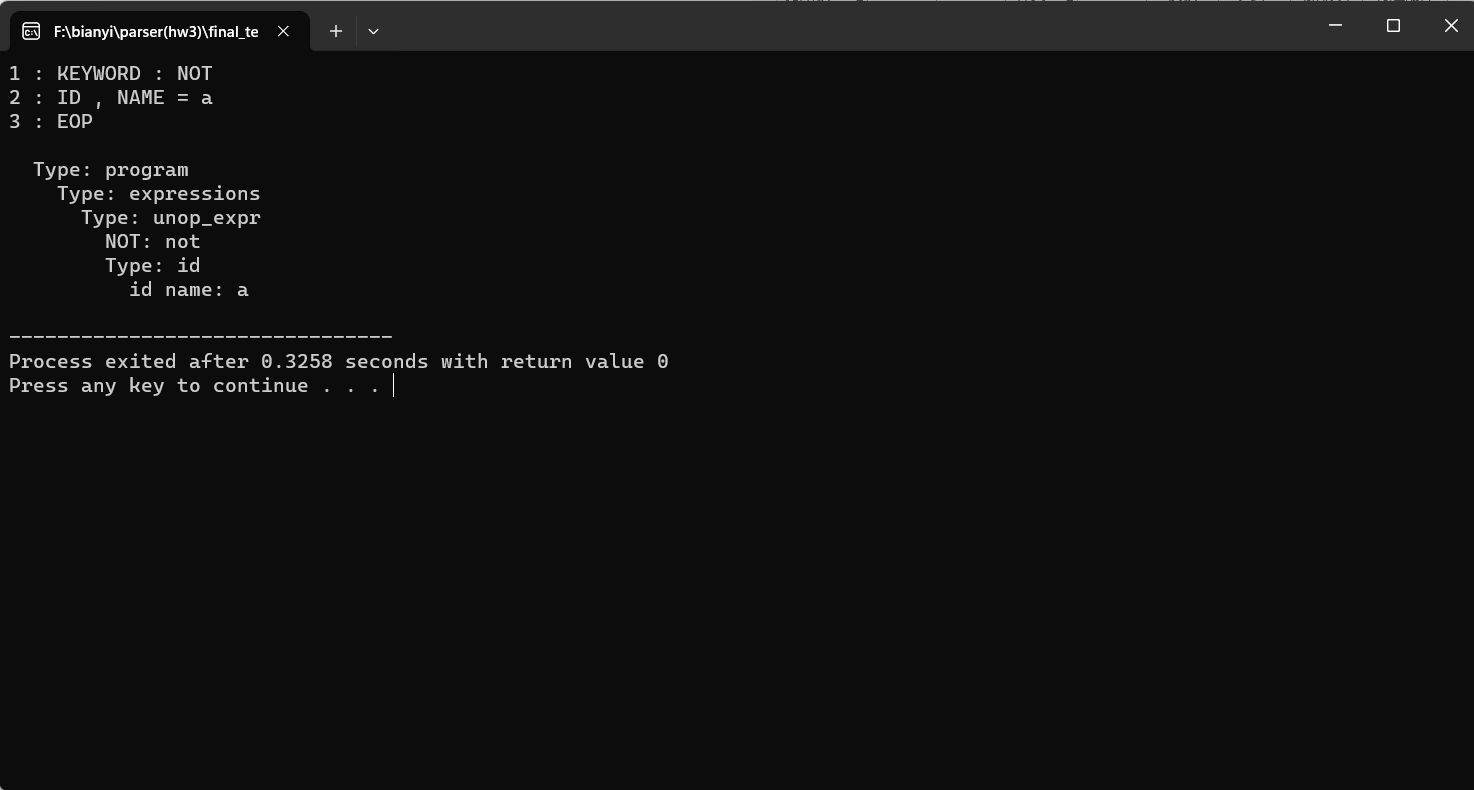


**<array> => \[ array: (e  | (<expr> ,)\* <expr> ) \]**



**<unop-expr> => <unop> ( <non-op-expr> |  <paren-expr> )**

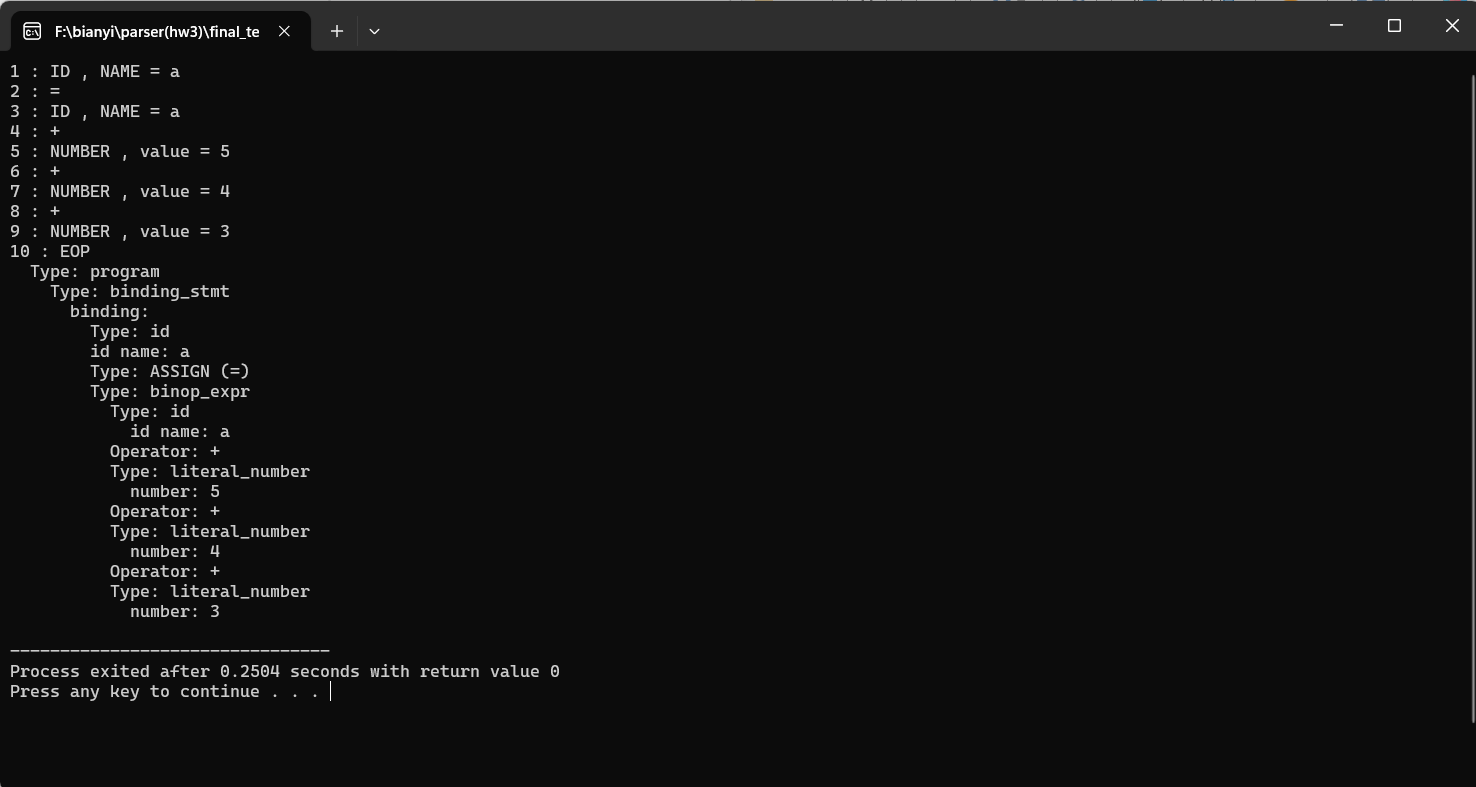
**<unop> => not**



**<binop-expr> => <non-op-expr> <binop> ( <non-op-expr> | <paren-expr> )**

**<binop> => [+ , - , \* , / , %, < , <= , > , >= ,  ==]**

**<binding-stmt> => <id> = <expr>**



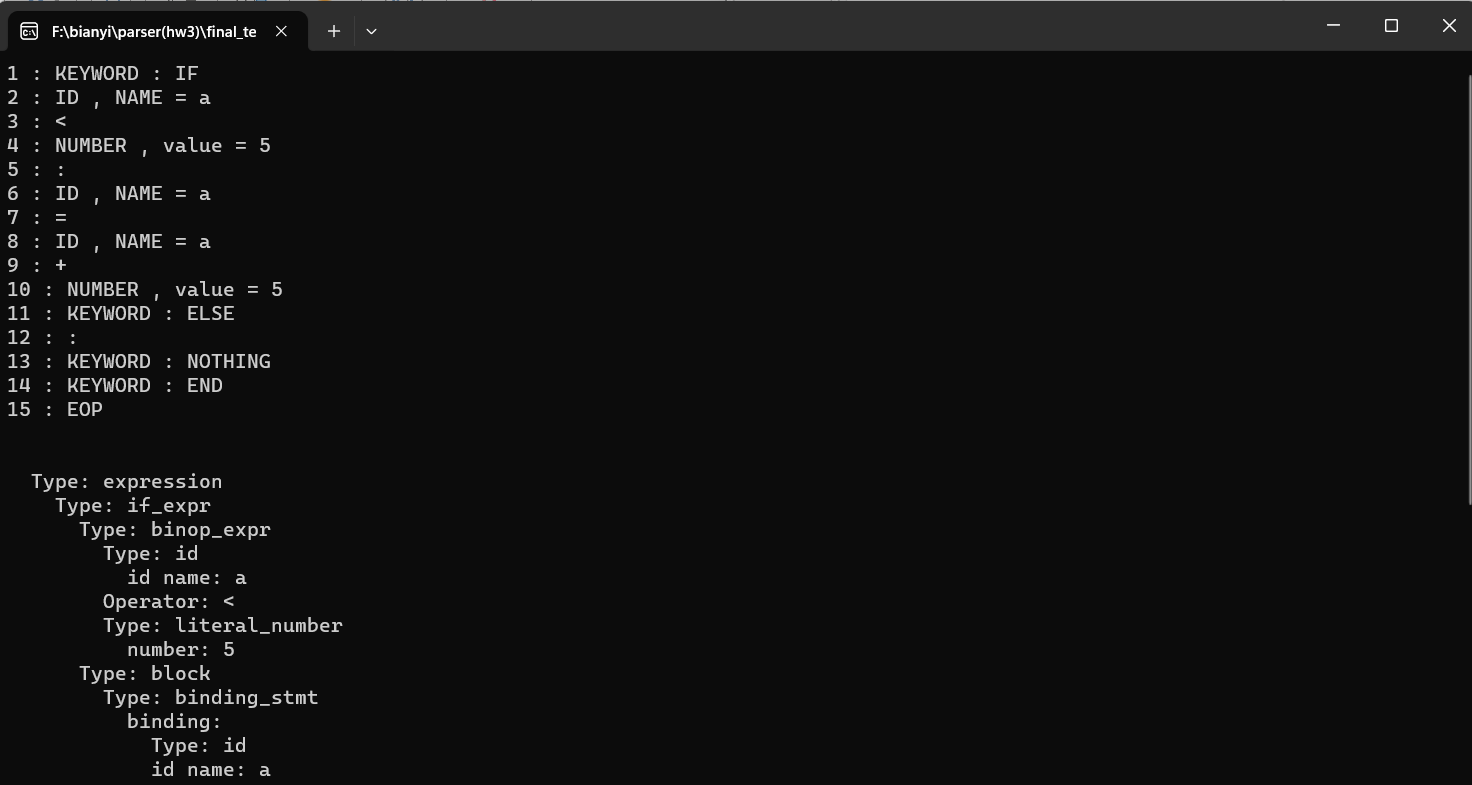
**<if-expr> =>**

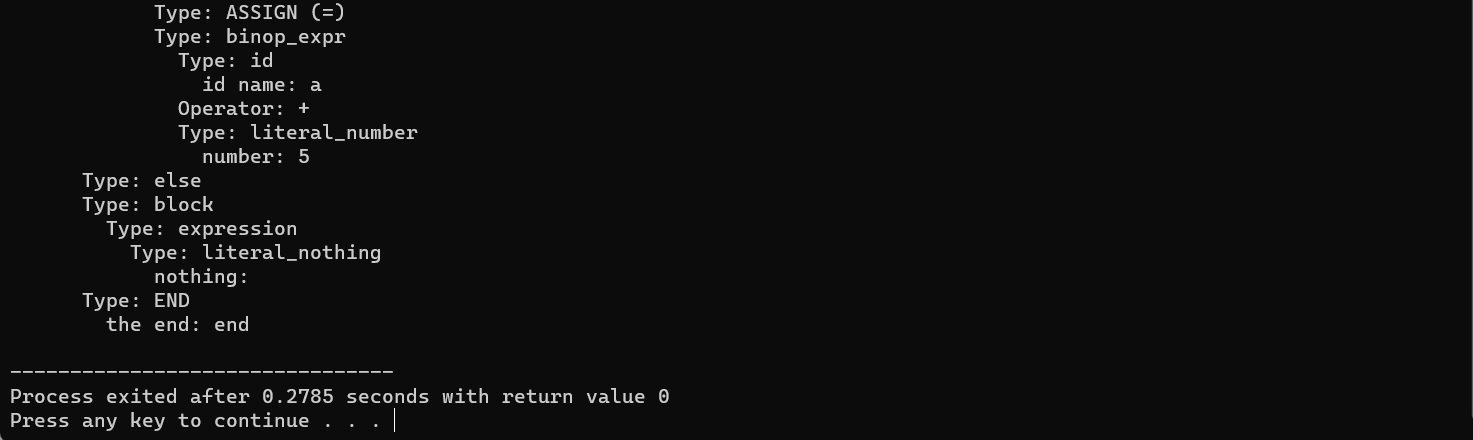
**if <expr> : <block> (<else-if>)+  (else: <block>)? end |**

**if <expr> : <block> (<else-if>)\*  else: <block> end**

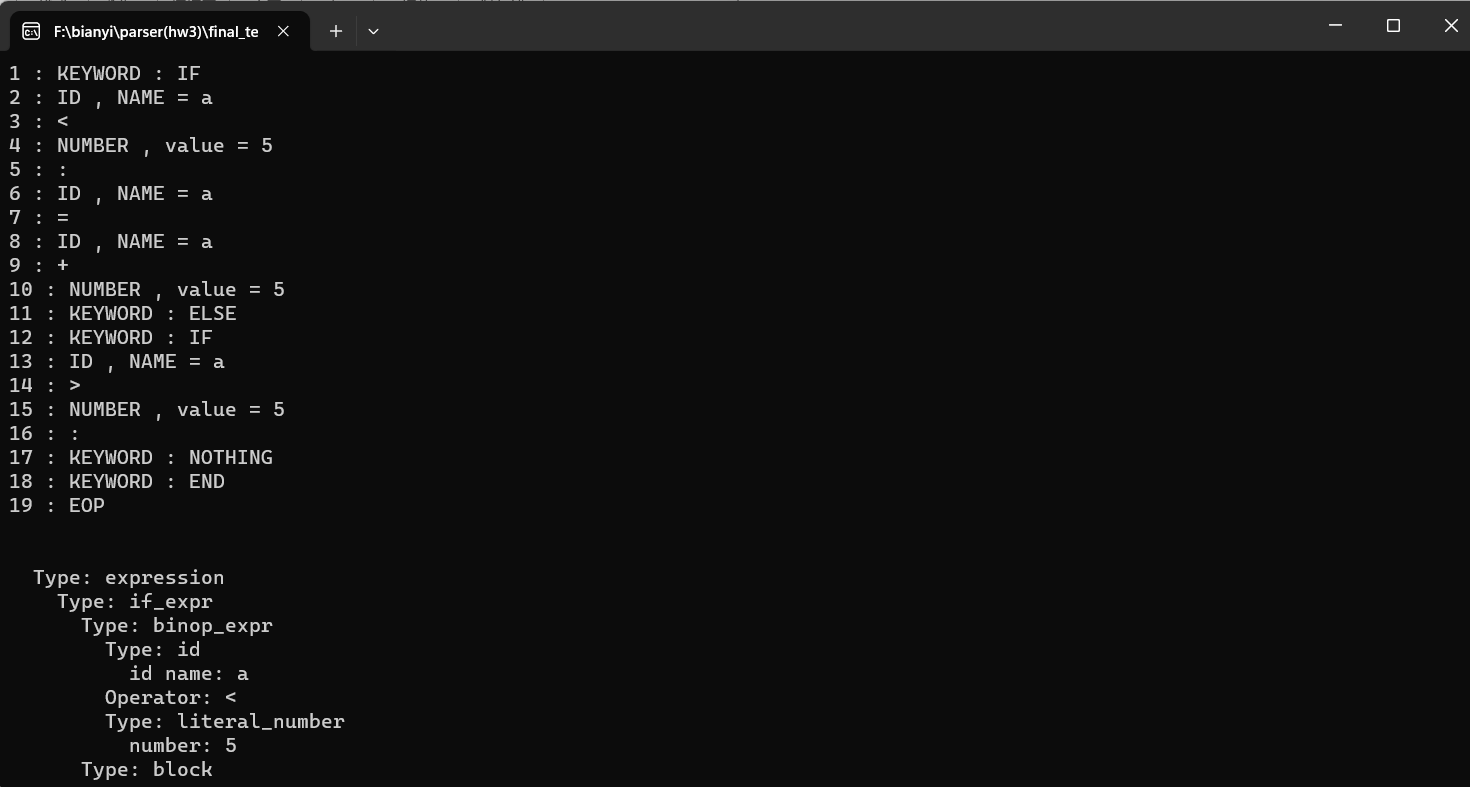
**‹else-if› =>  else if <expr> : <block>**

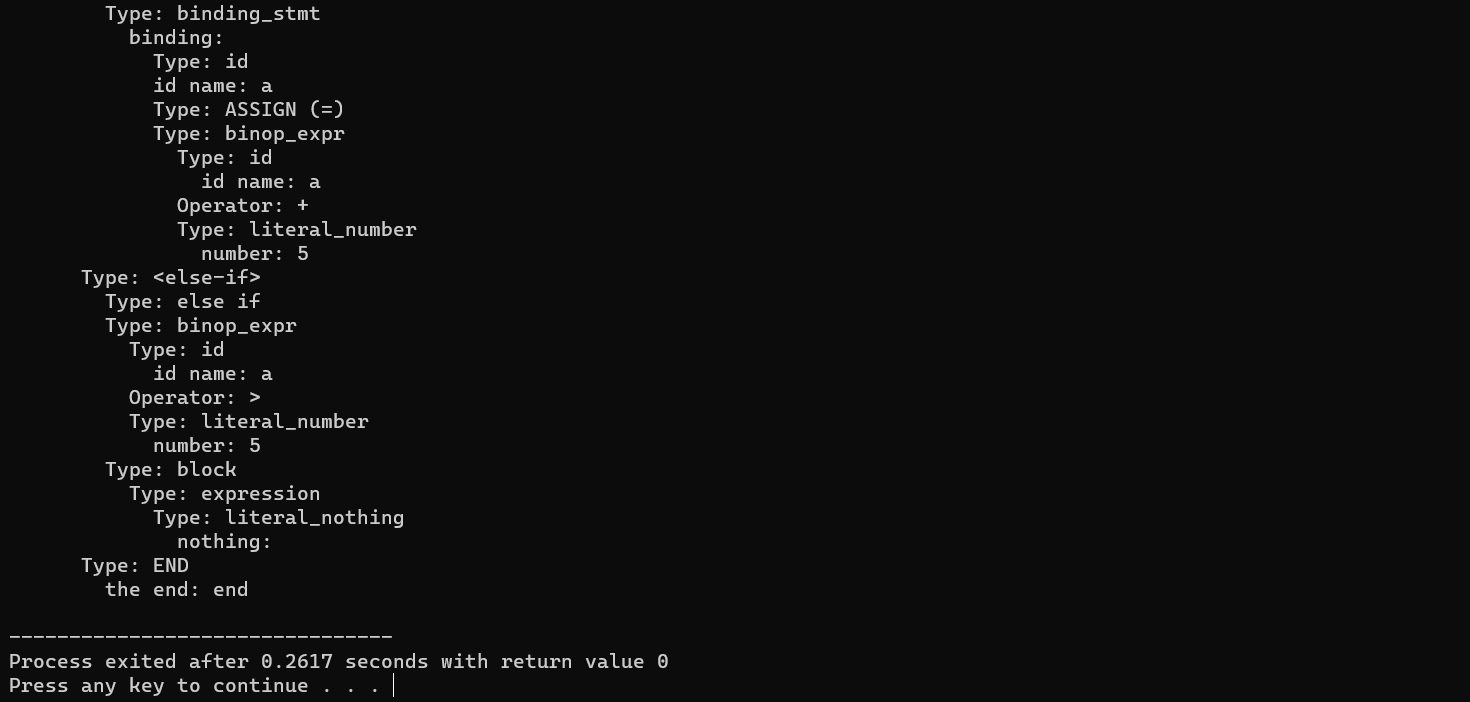
**[if -else]**



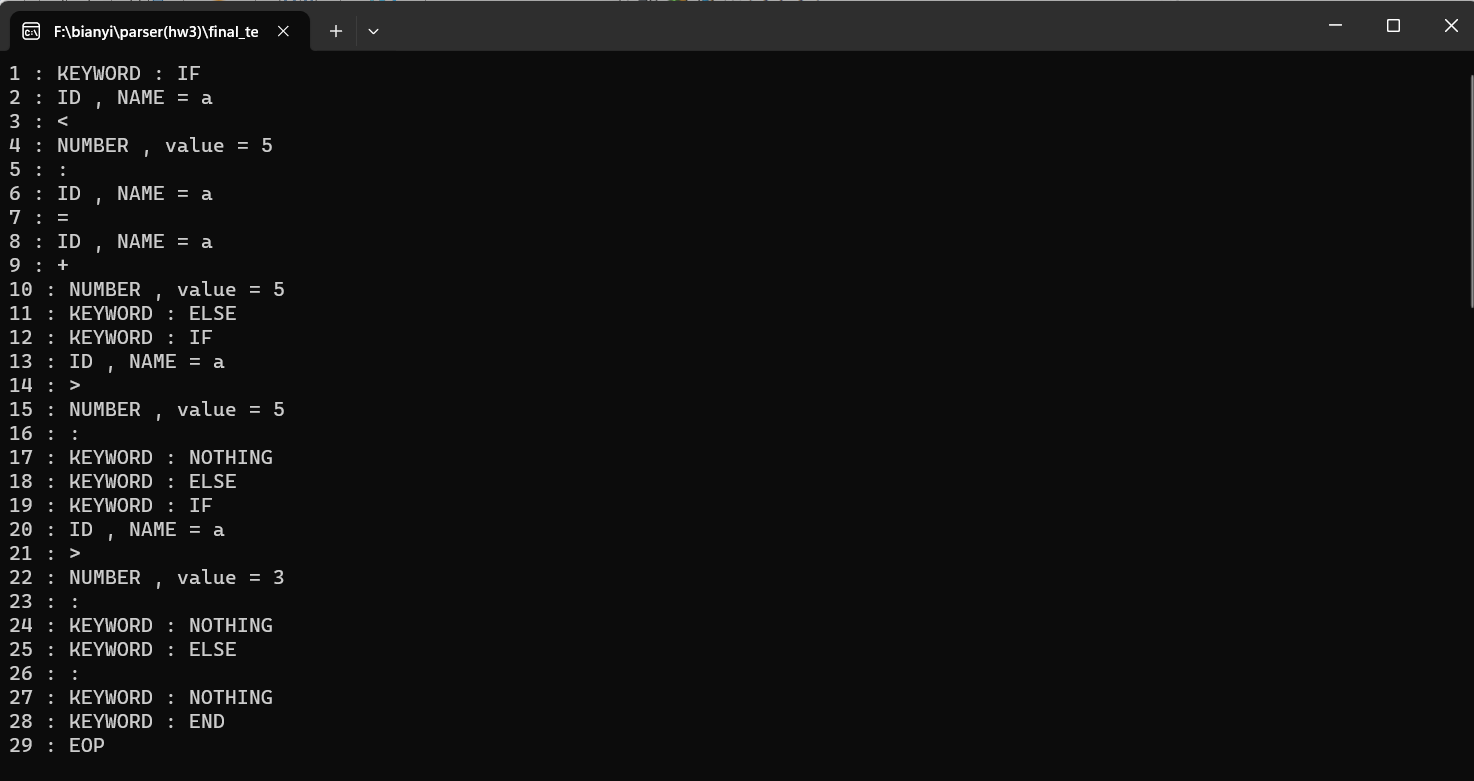


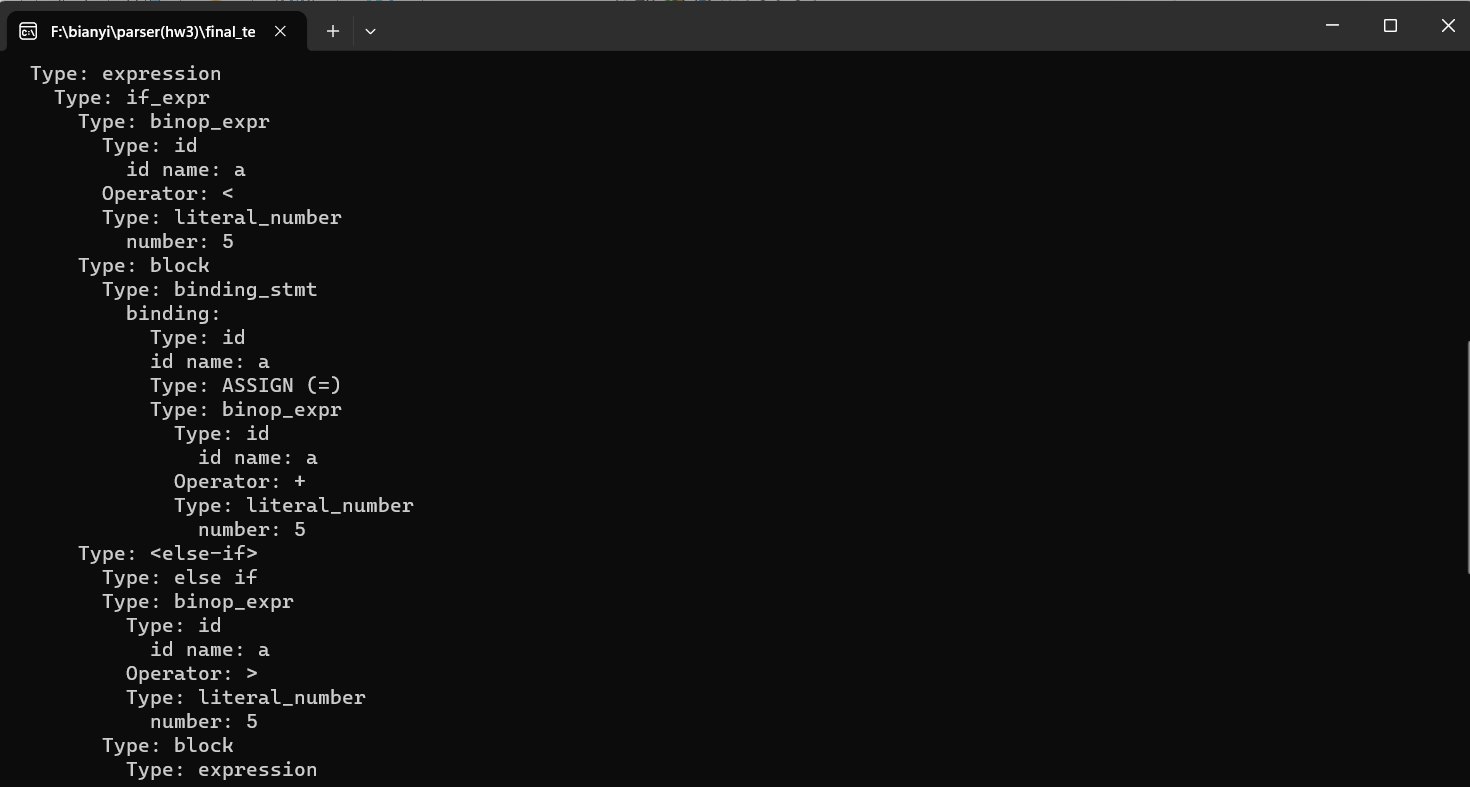
**[if-elseif]**





**[if-((elseif)+)-else]**

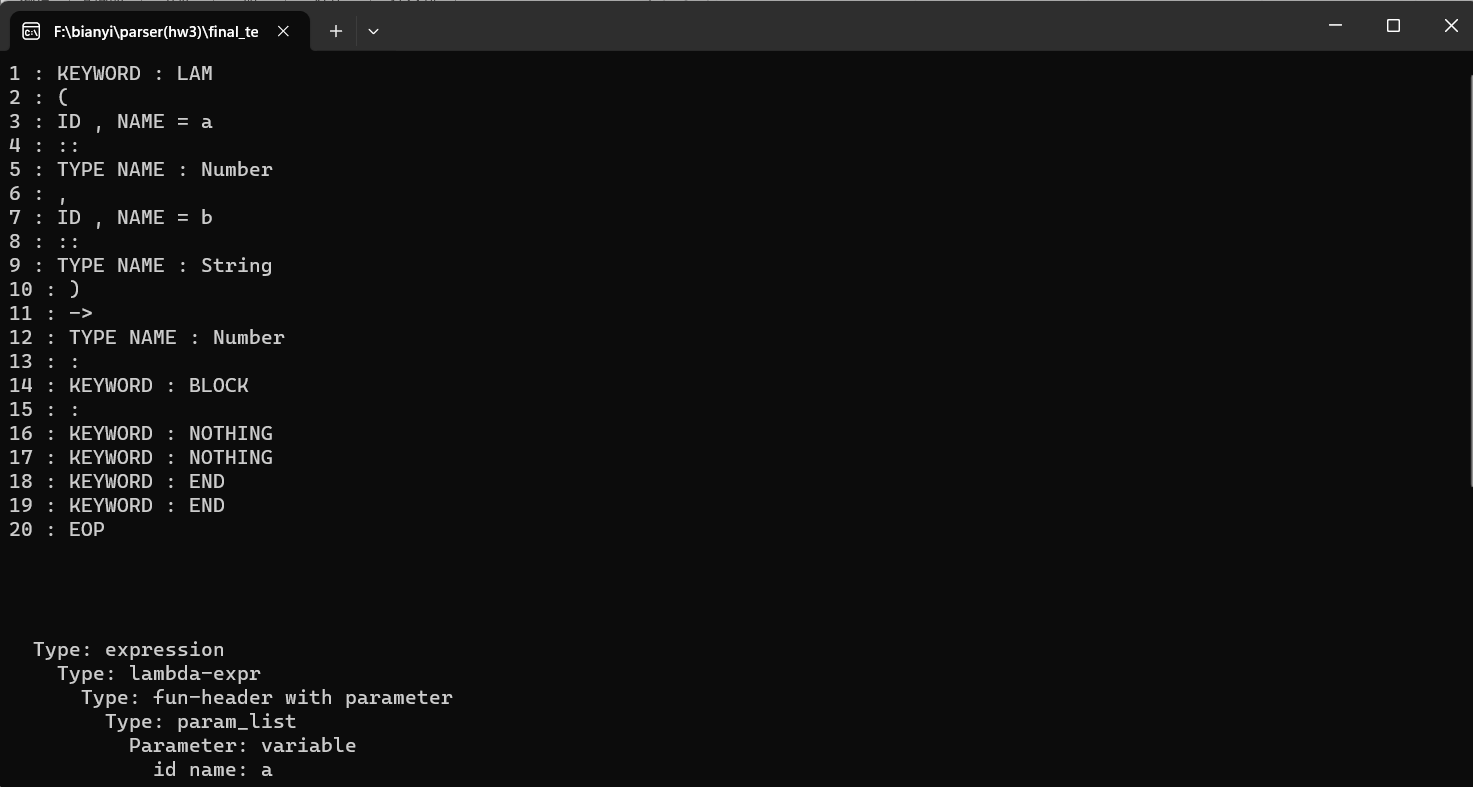






**<lambda-expr>  => lam <fun-header> : <block> end**

**<block> => <stmt> | block :  <stmt>\* end**



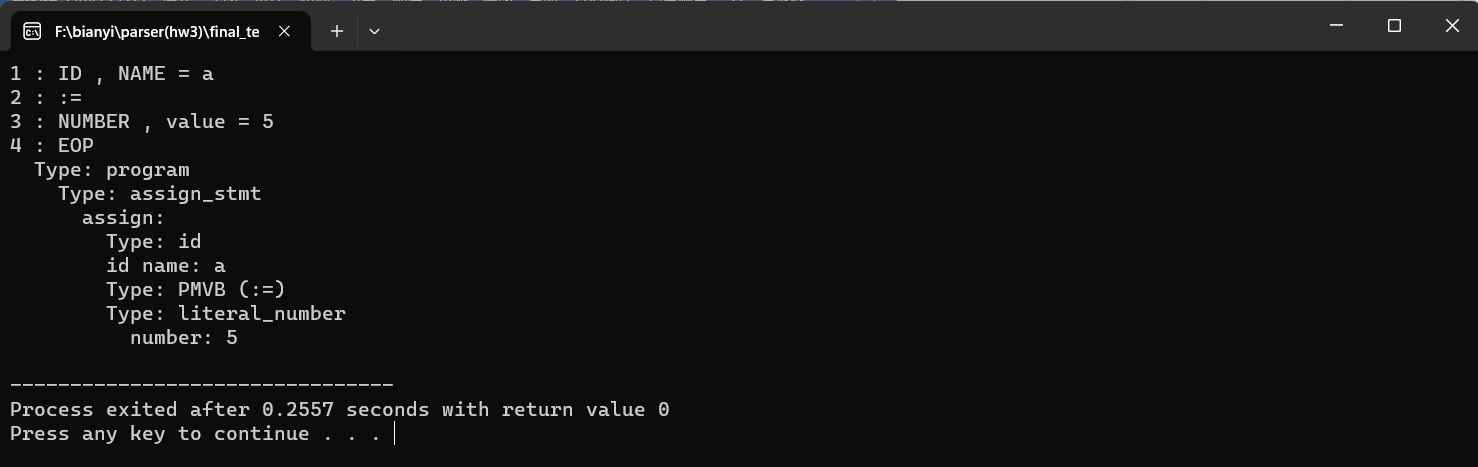


**<stmts> => (<stmt>) \***

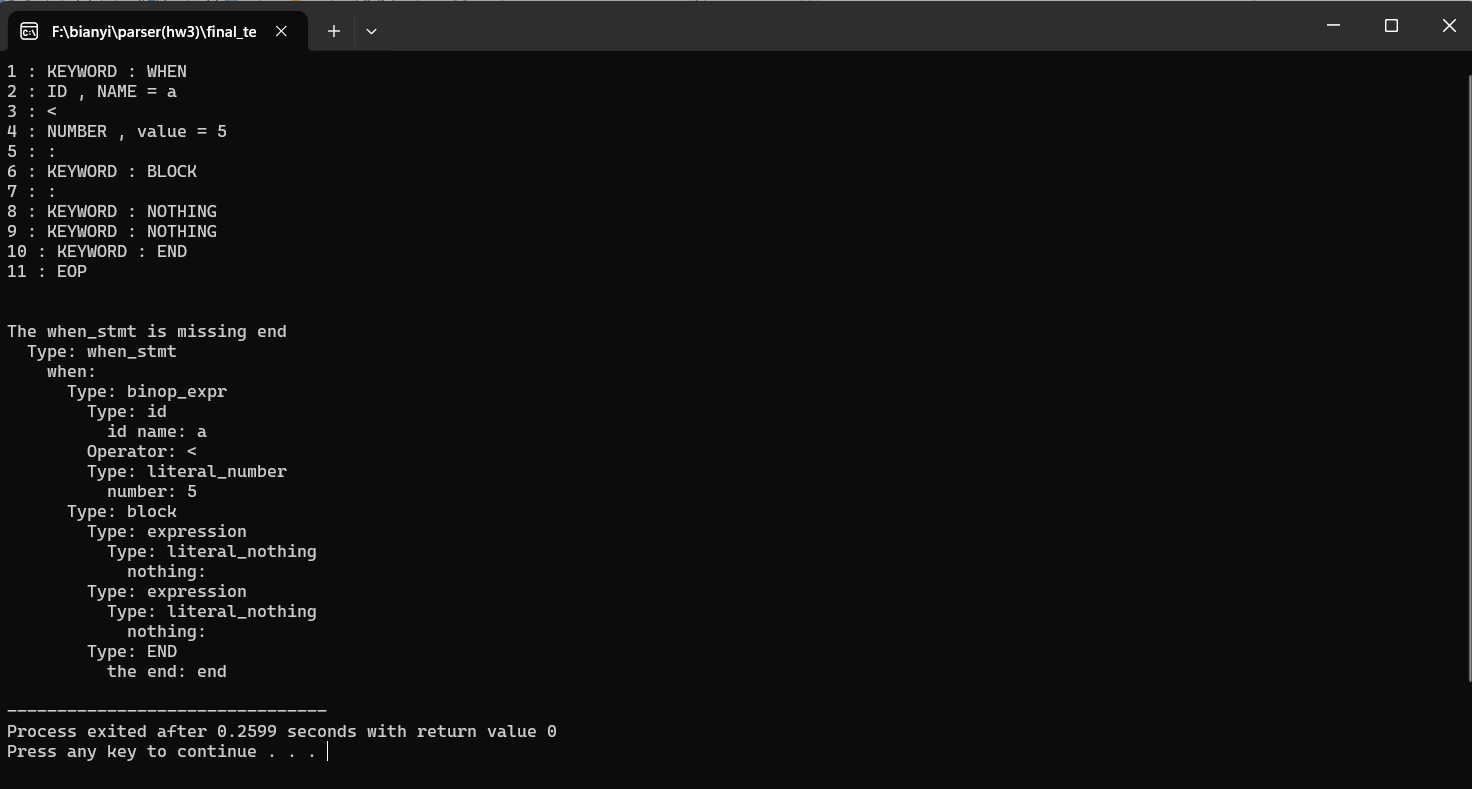
**<stmt> =>  <when-stmt> | <assign-stmt> | <expr> |**

**<decl-stmt> | <binding-stmt>**

**<assign-stmt> => <id> := <expr>**



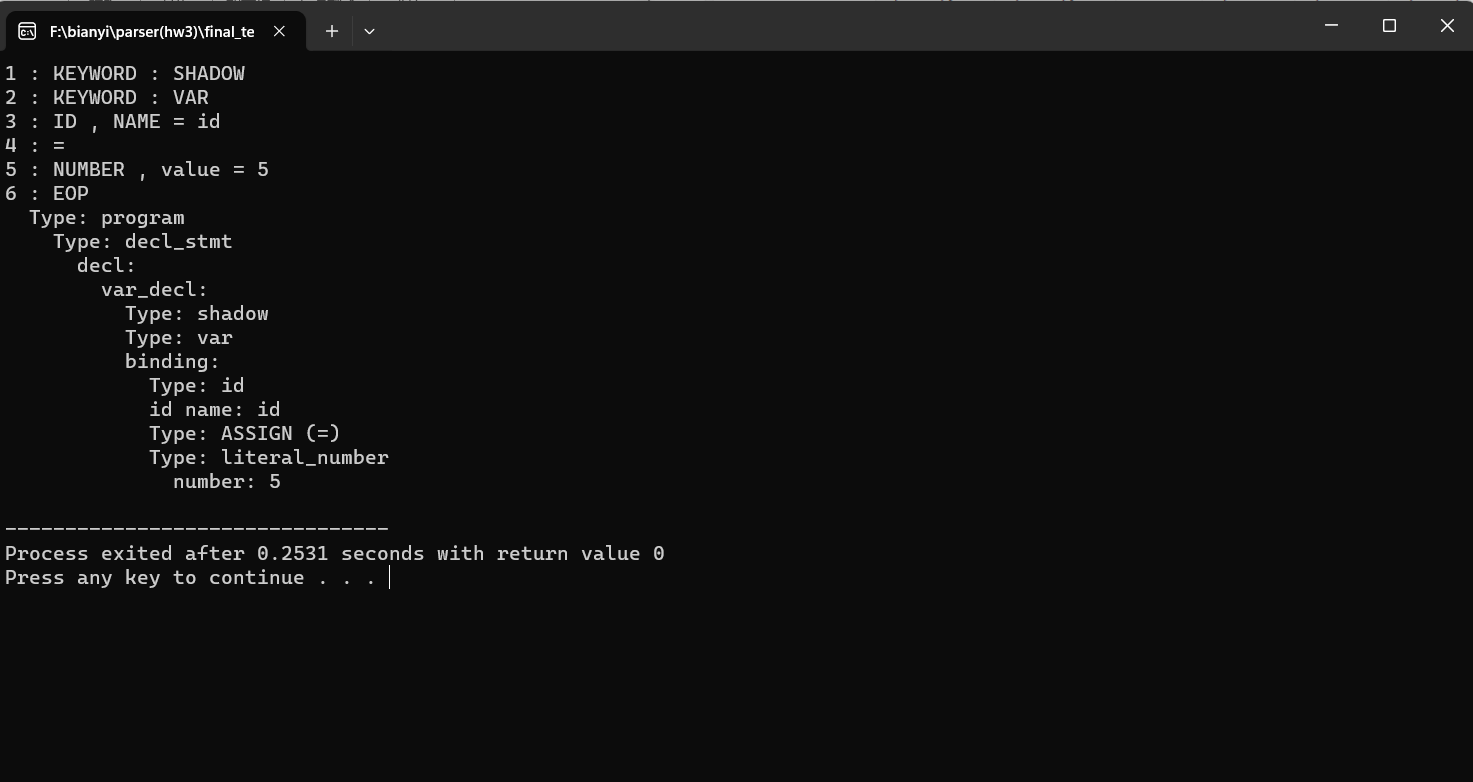
**<when-stmt> => when <expr> : <block> end**



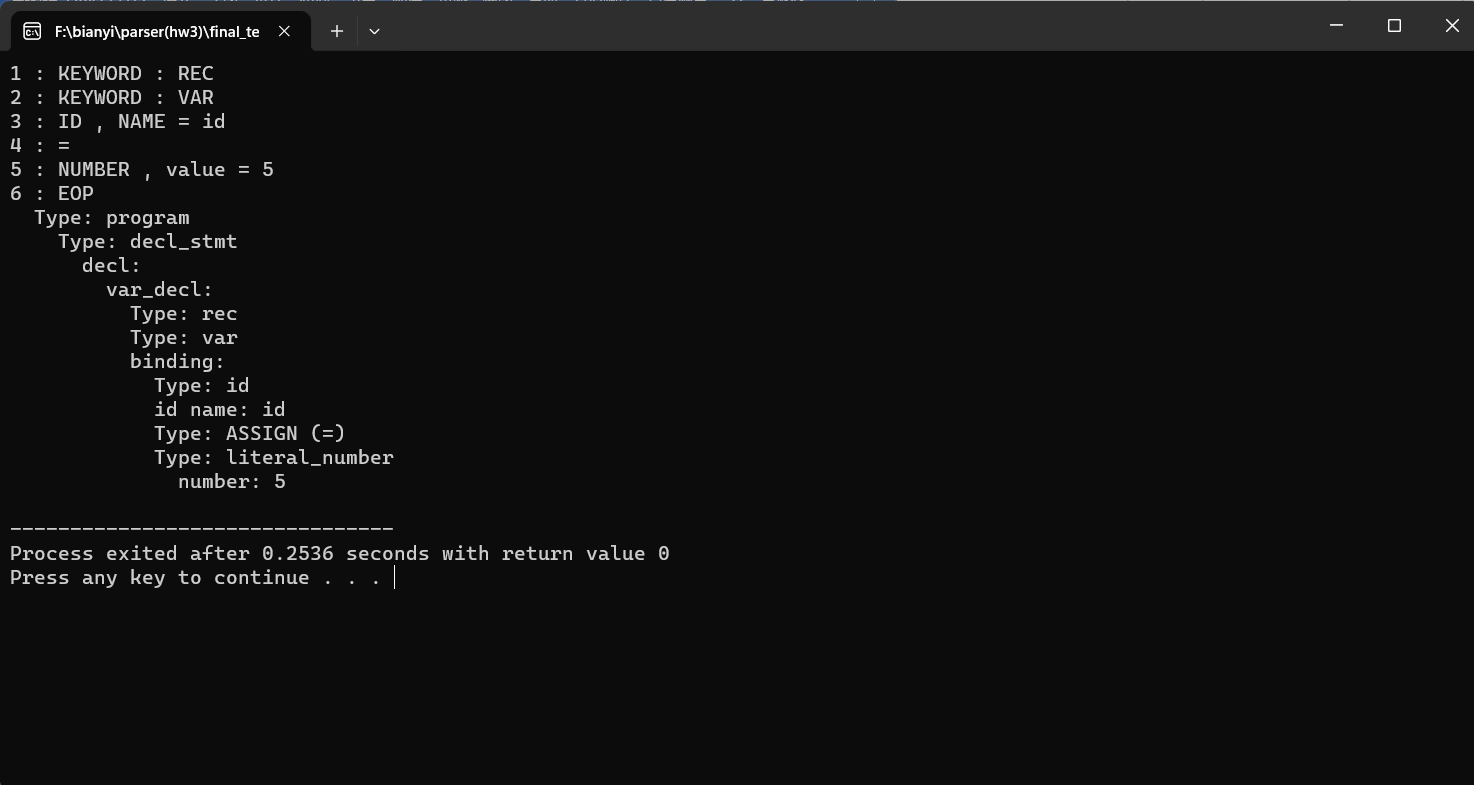
**<decl-stmt> => <var-decl> | <fun-decl>**

**<var-decl> => (shadow)? (rec)? var <id> = <expr>**

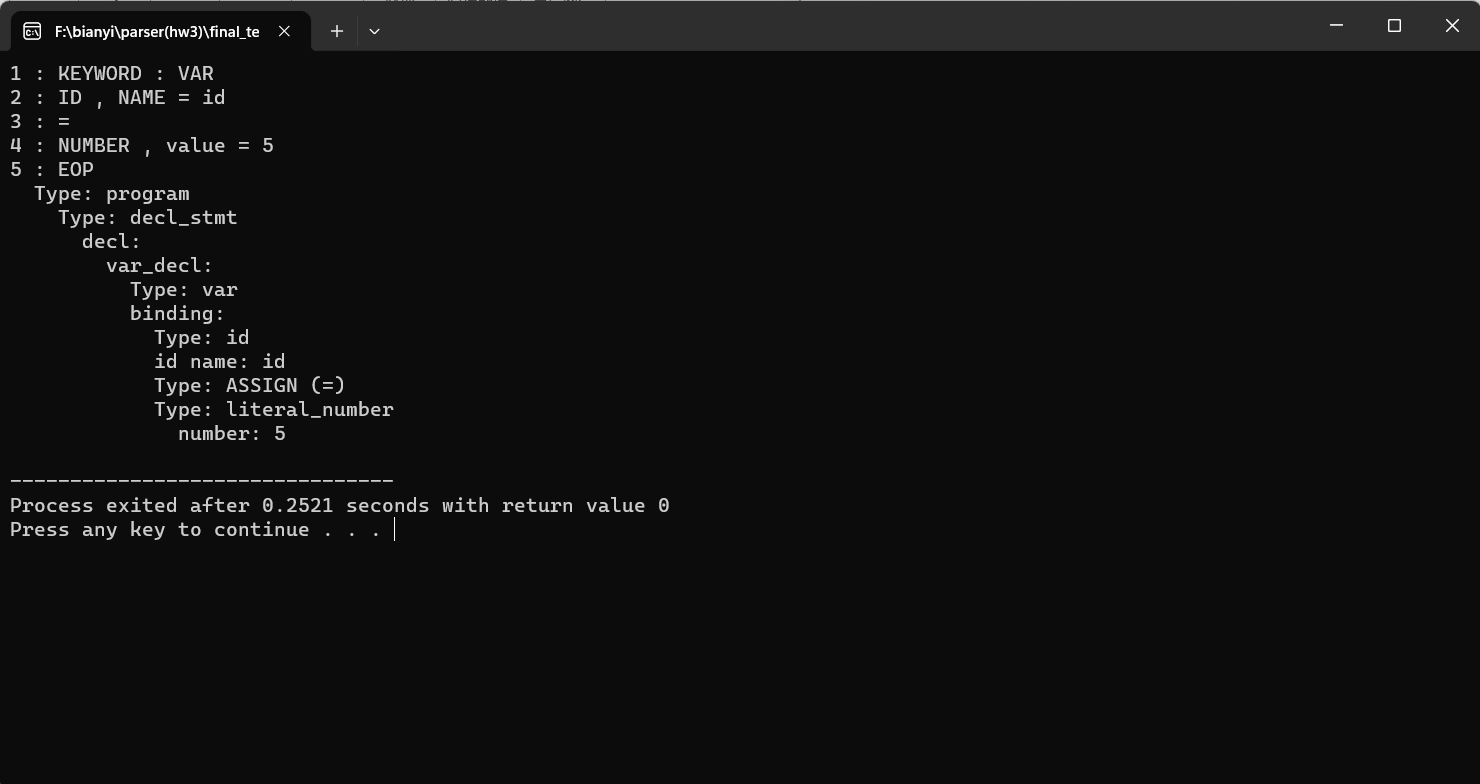
**[shadow var]**



**[rec var]**



**[var]**



**<fun-decl> => (shadow)? (rec)? fun <id> <fun-header> : <stmts> end**

**<fun-header> => \( < params>  \) (<return-part>)**

**<return-part> =>  -> <type-name>**

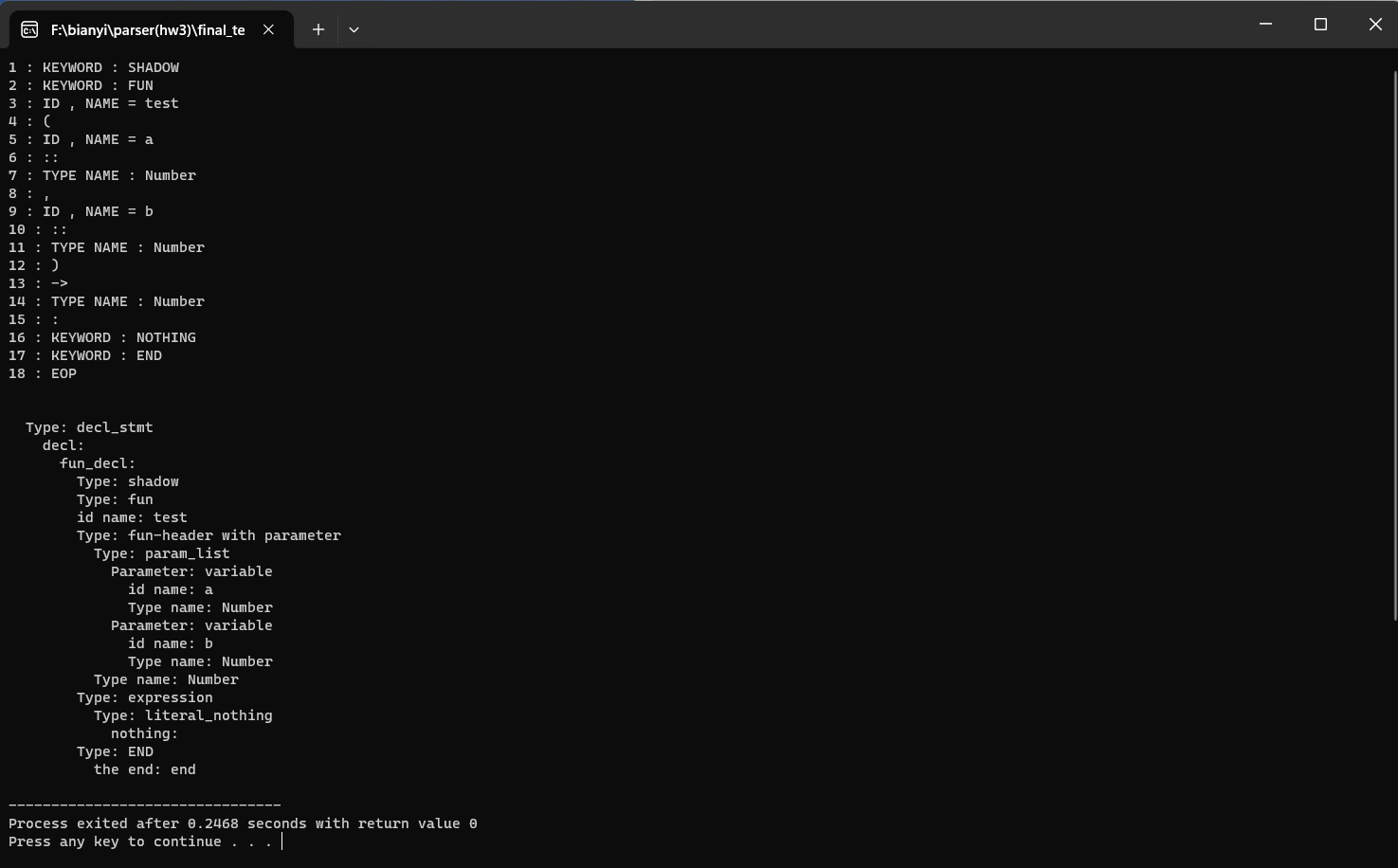
**<params> => e | <param\_list>**

**<param\_list> => (<param> ,)\* <param>**

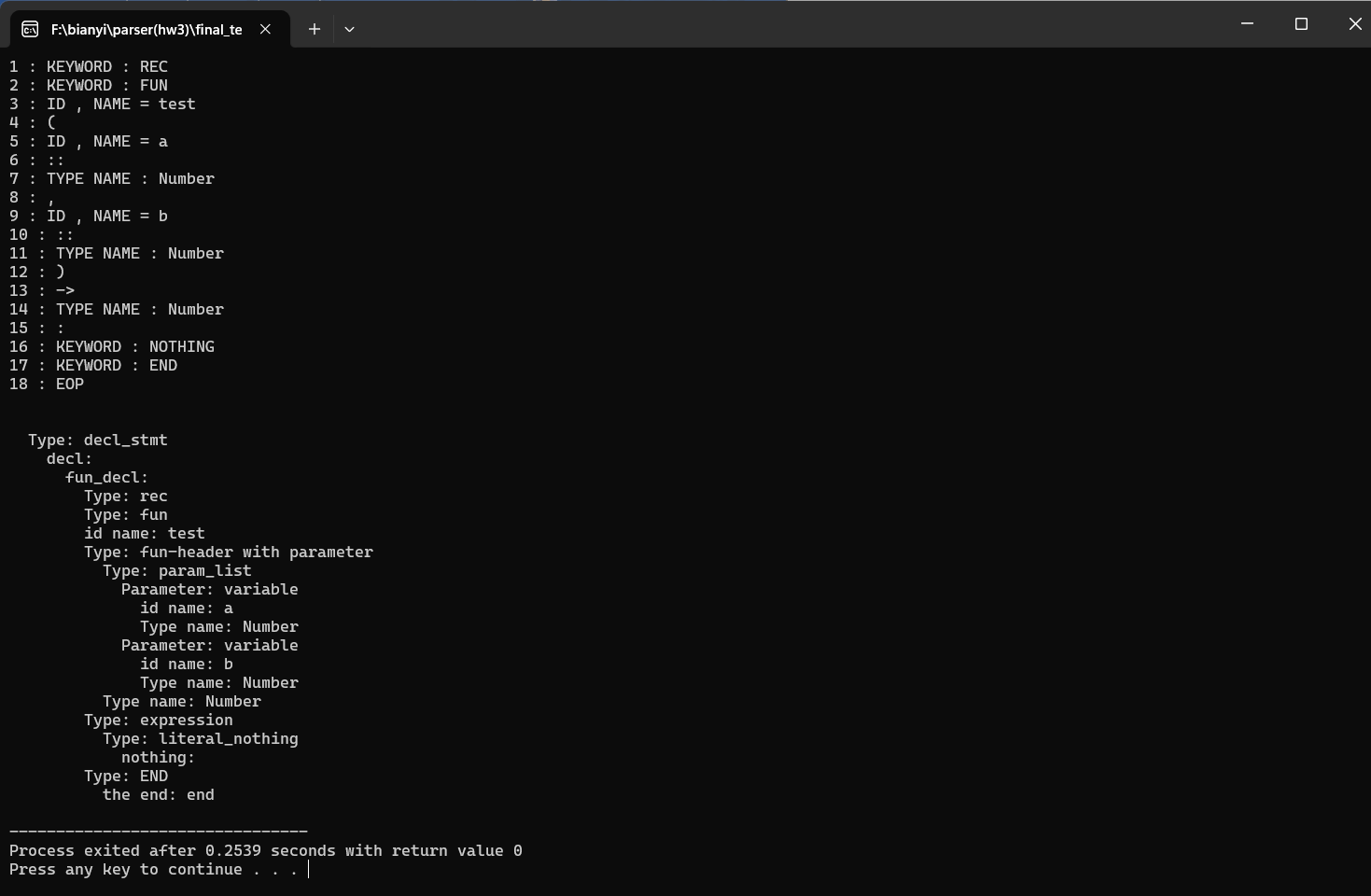
**<param> =>  (shadow)? <id> [:: <type-name>]**

**<type-name> => Any | Number | String | Boolean | Nothing | Function**

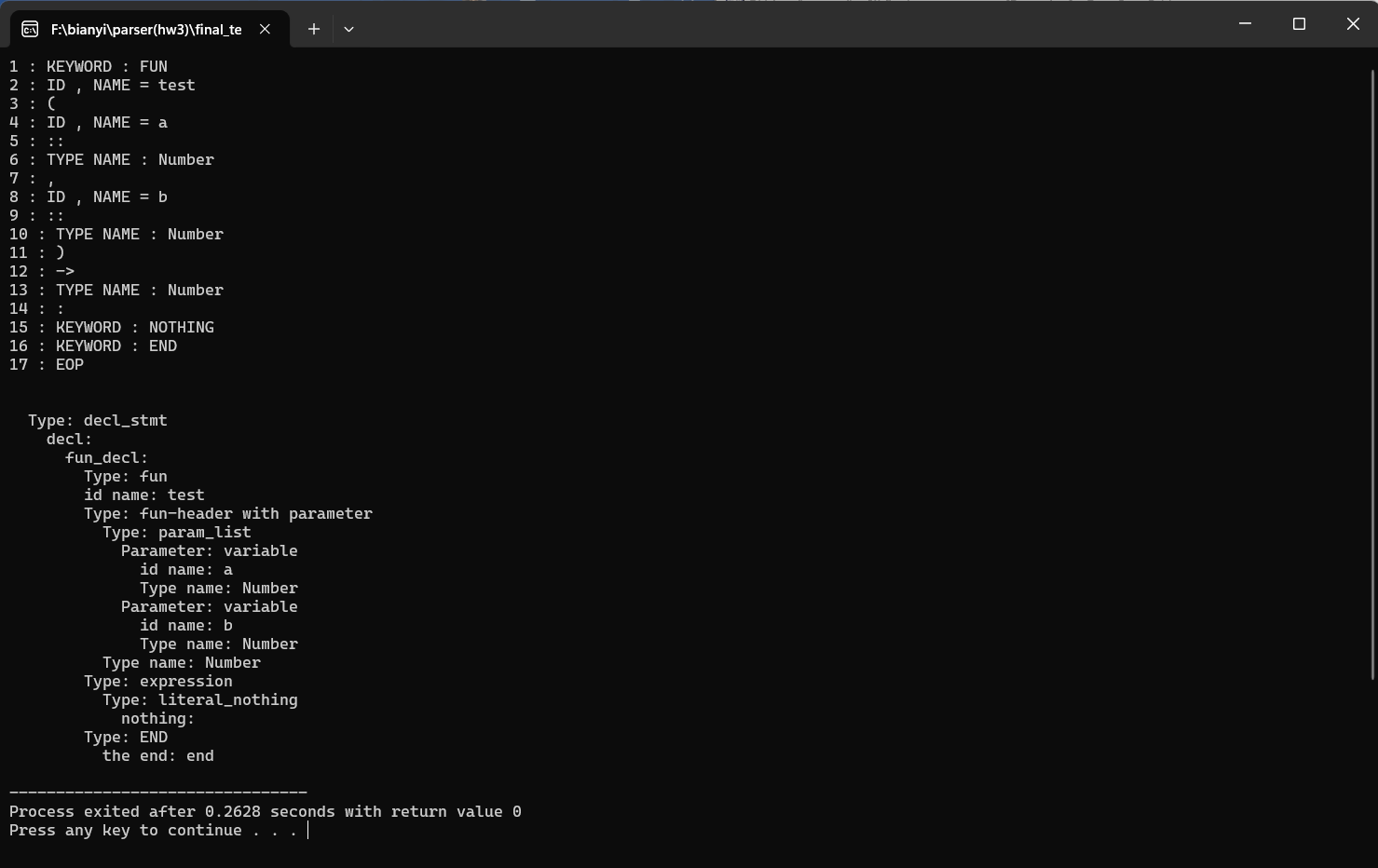
**[shadow fun]**



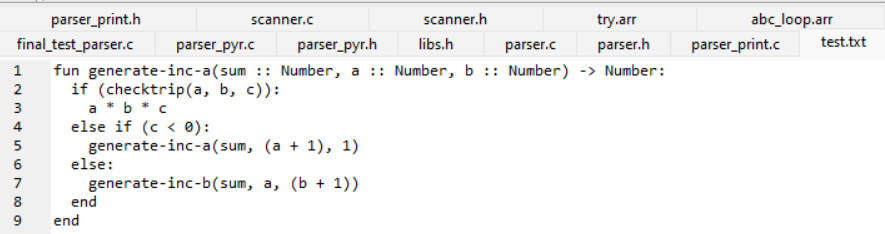
**[rec fun]**

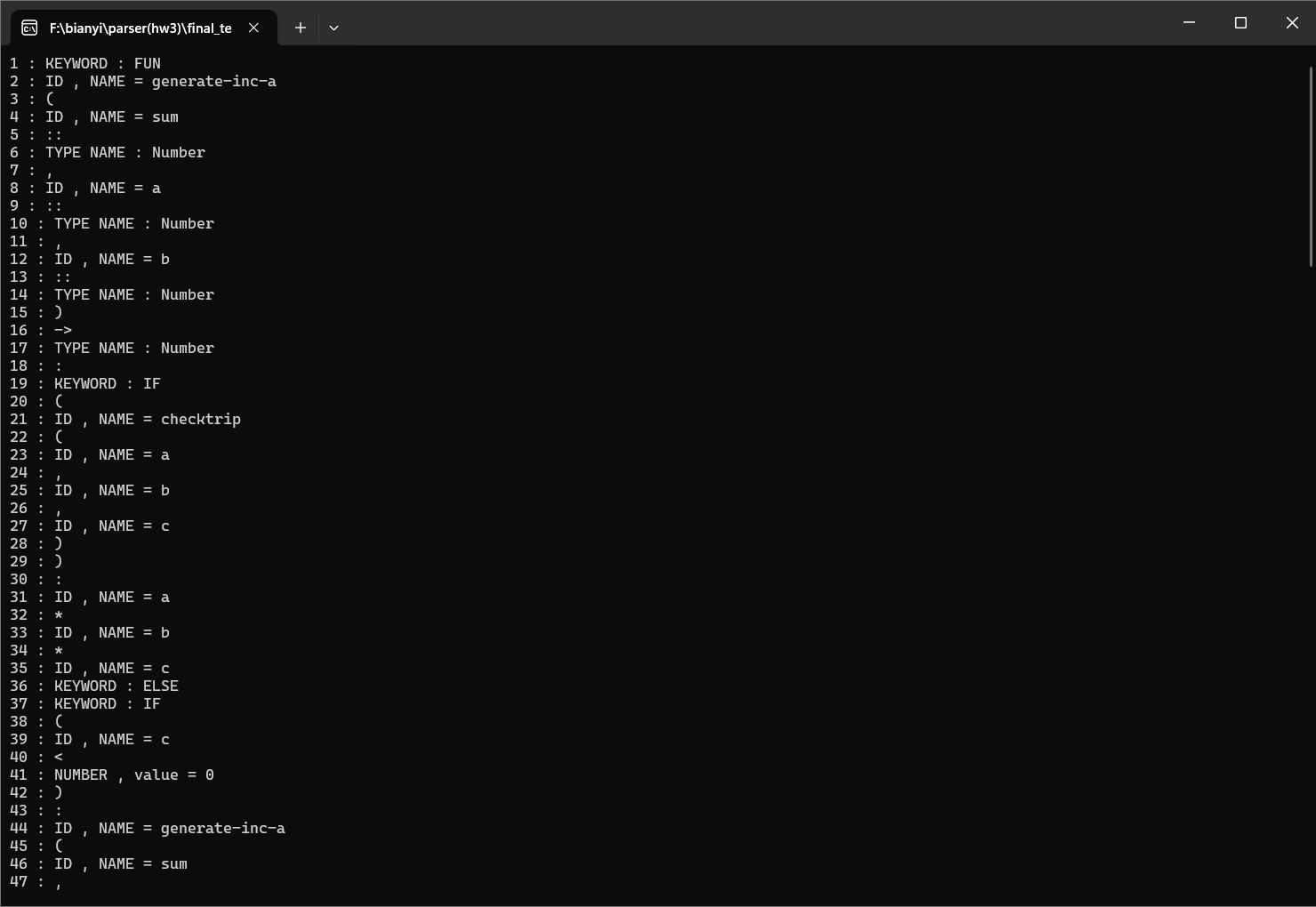


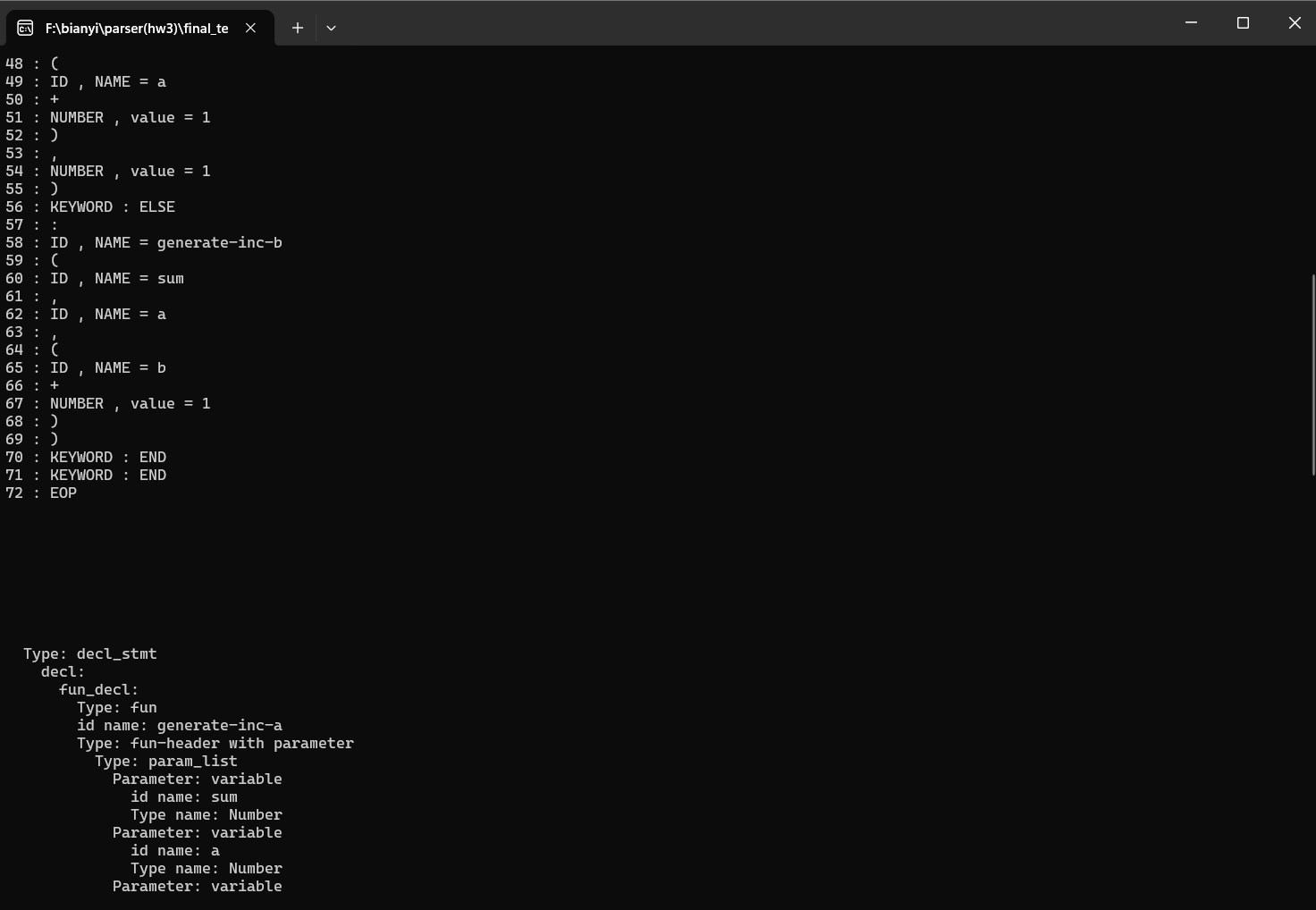
**[fun]**

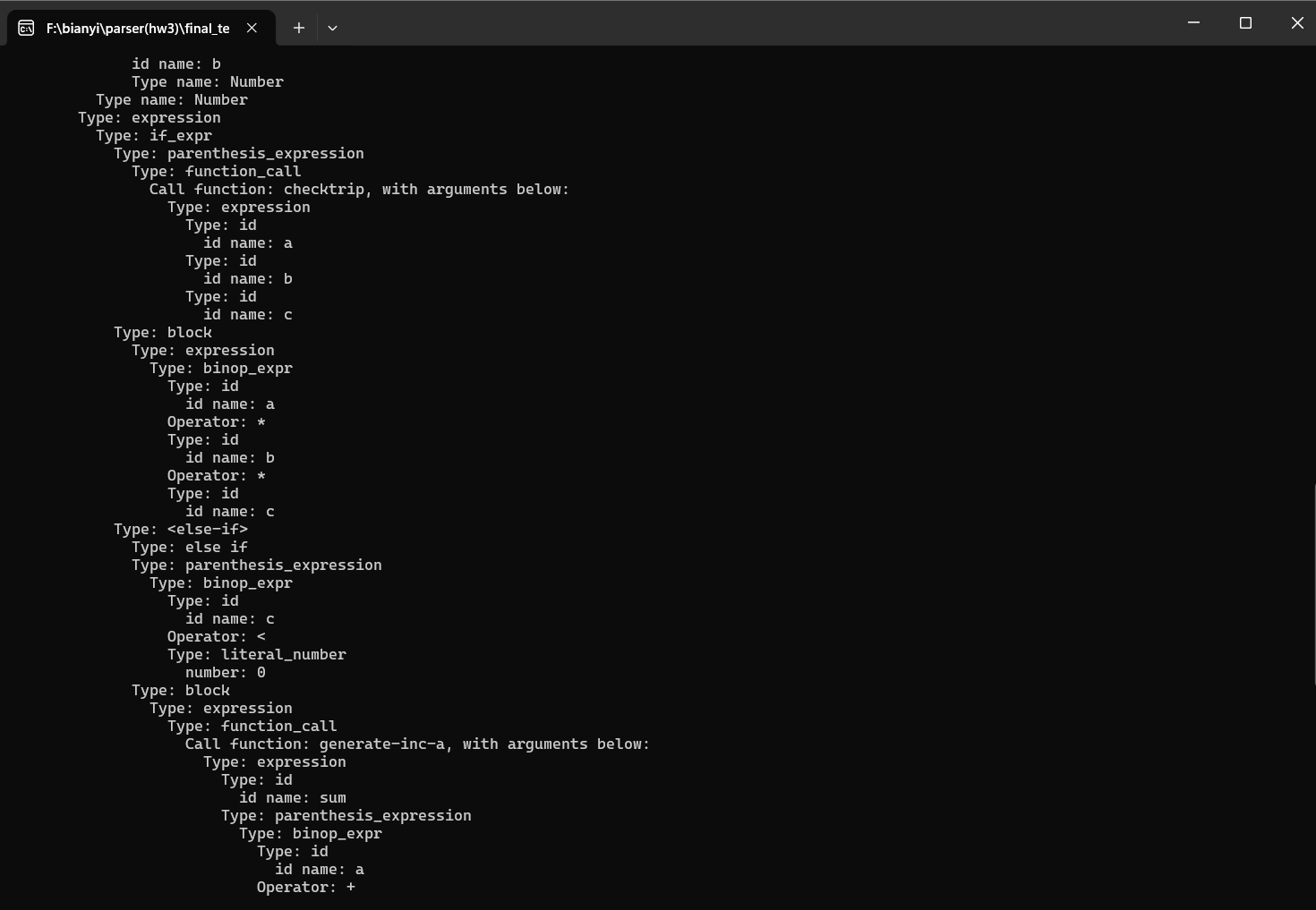


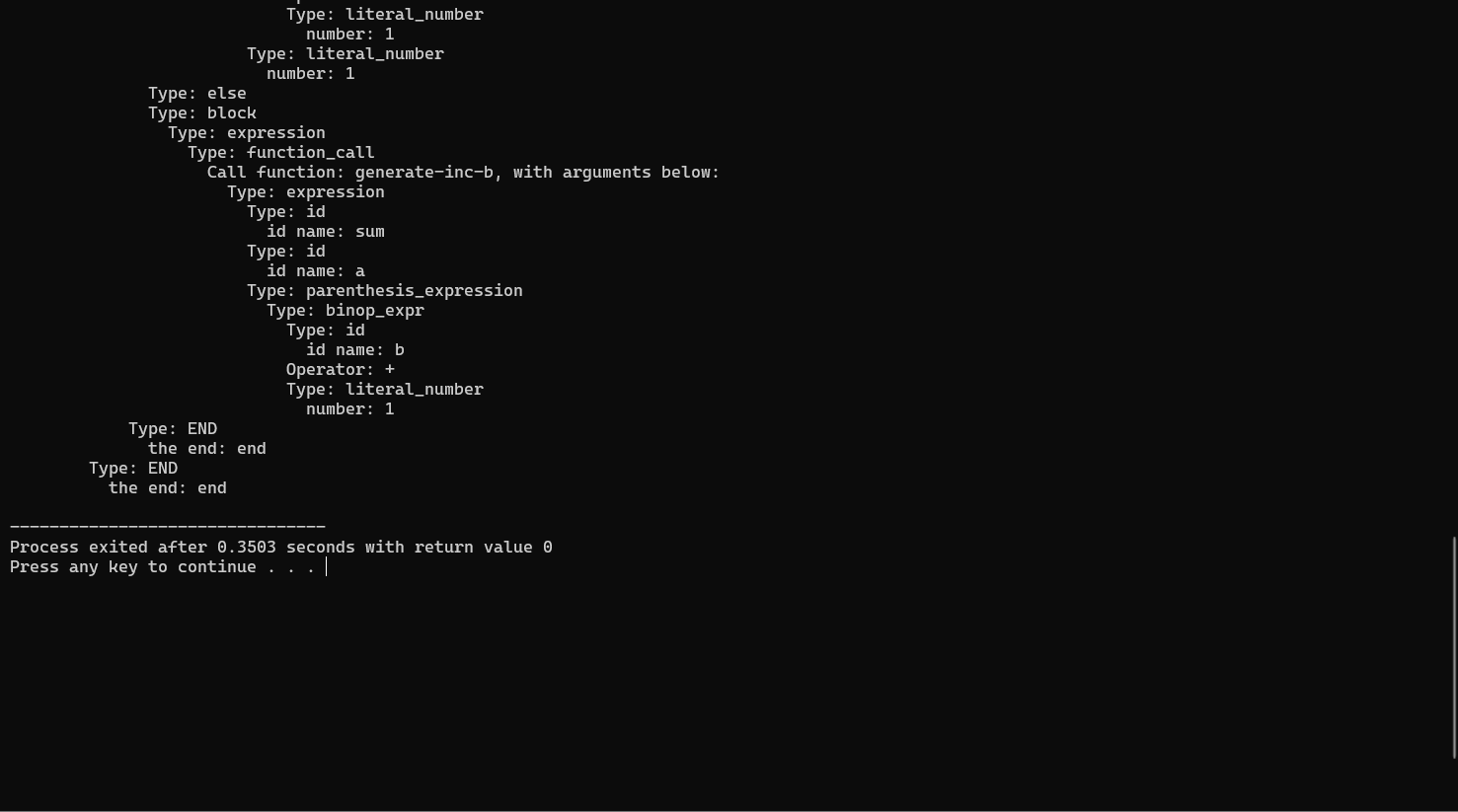
**Test results with complex correct Pyr code:**



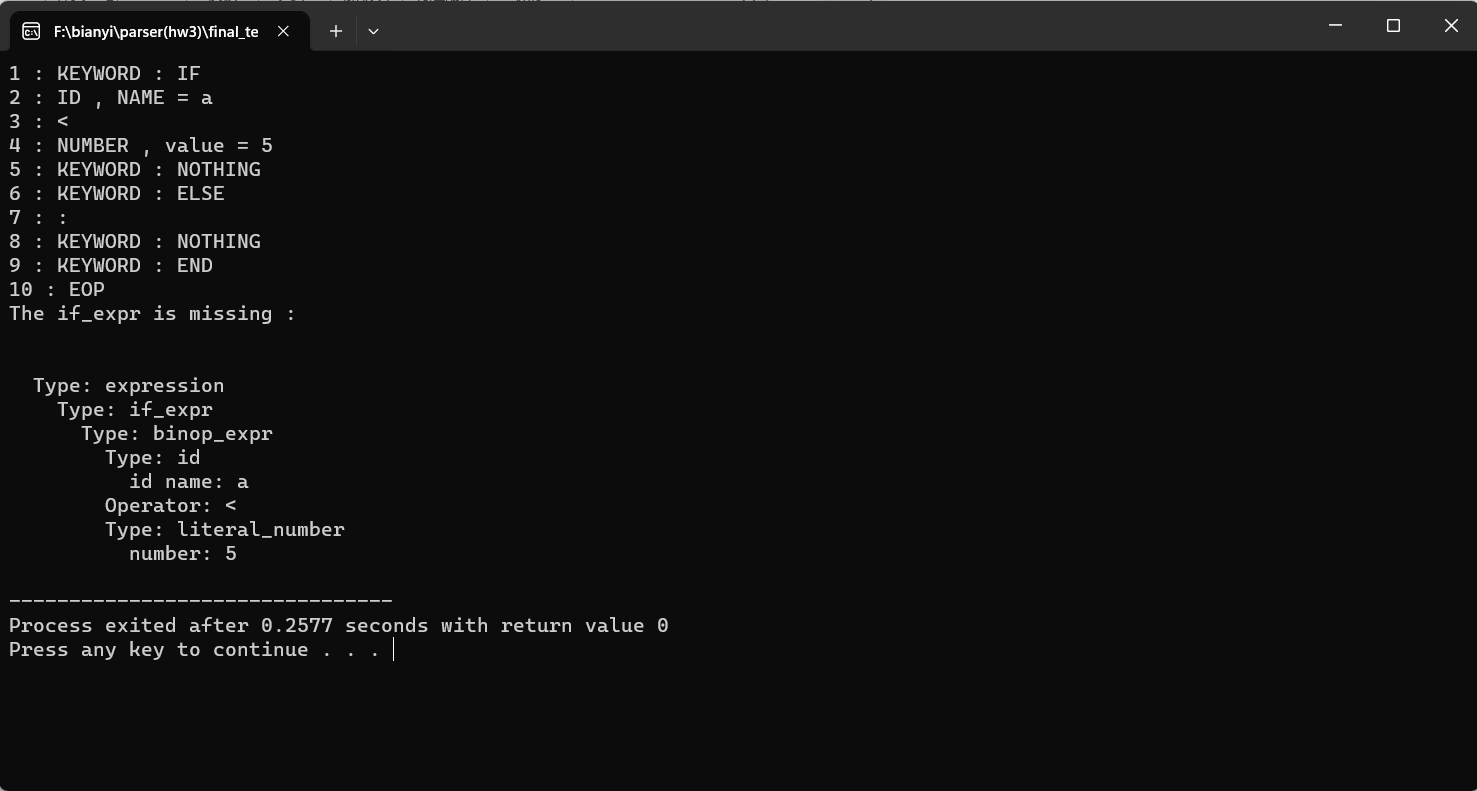


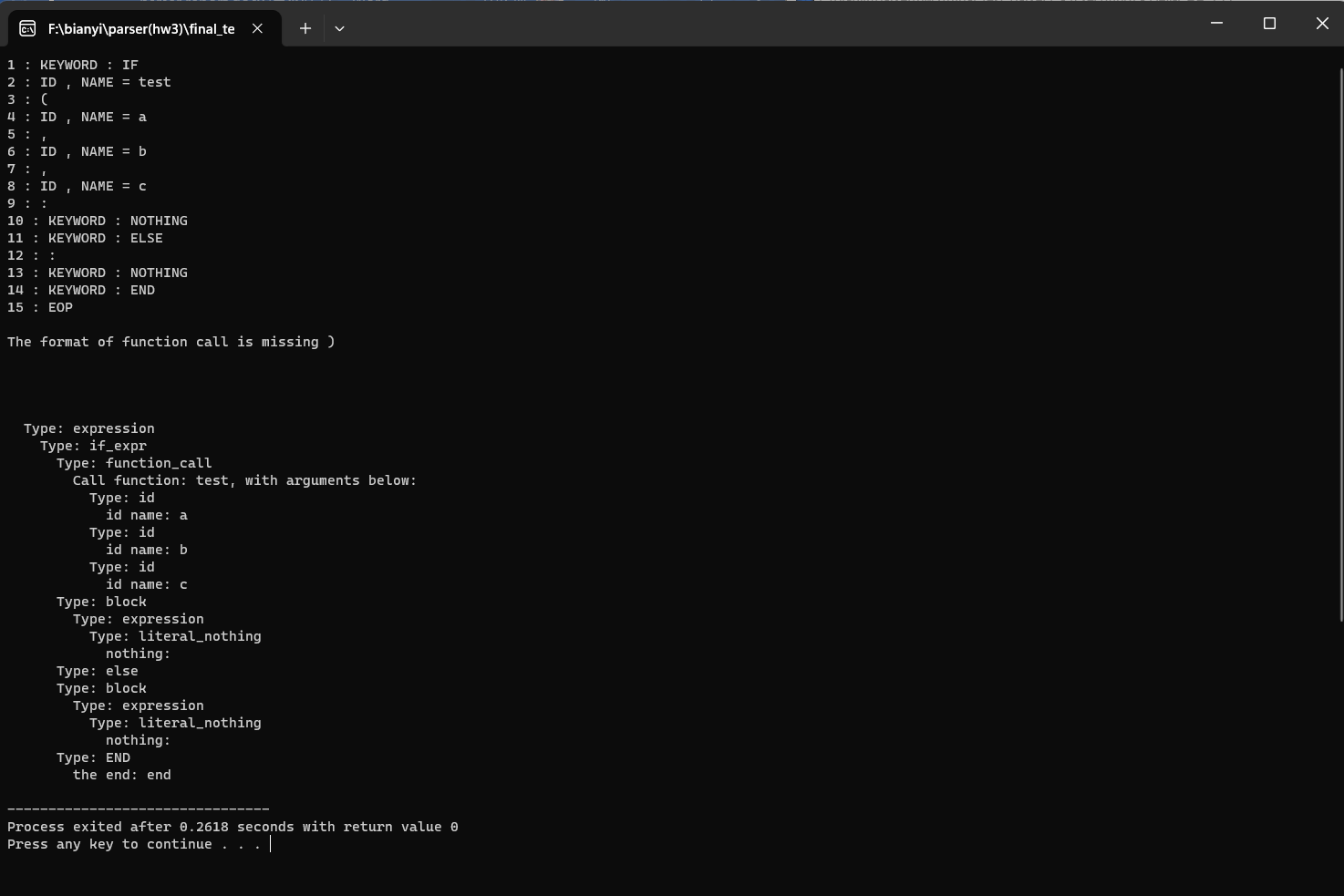


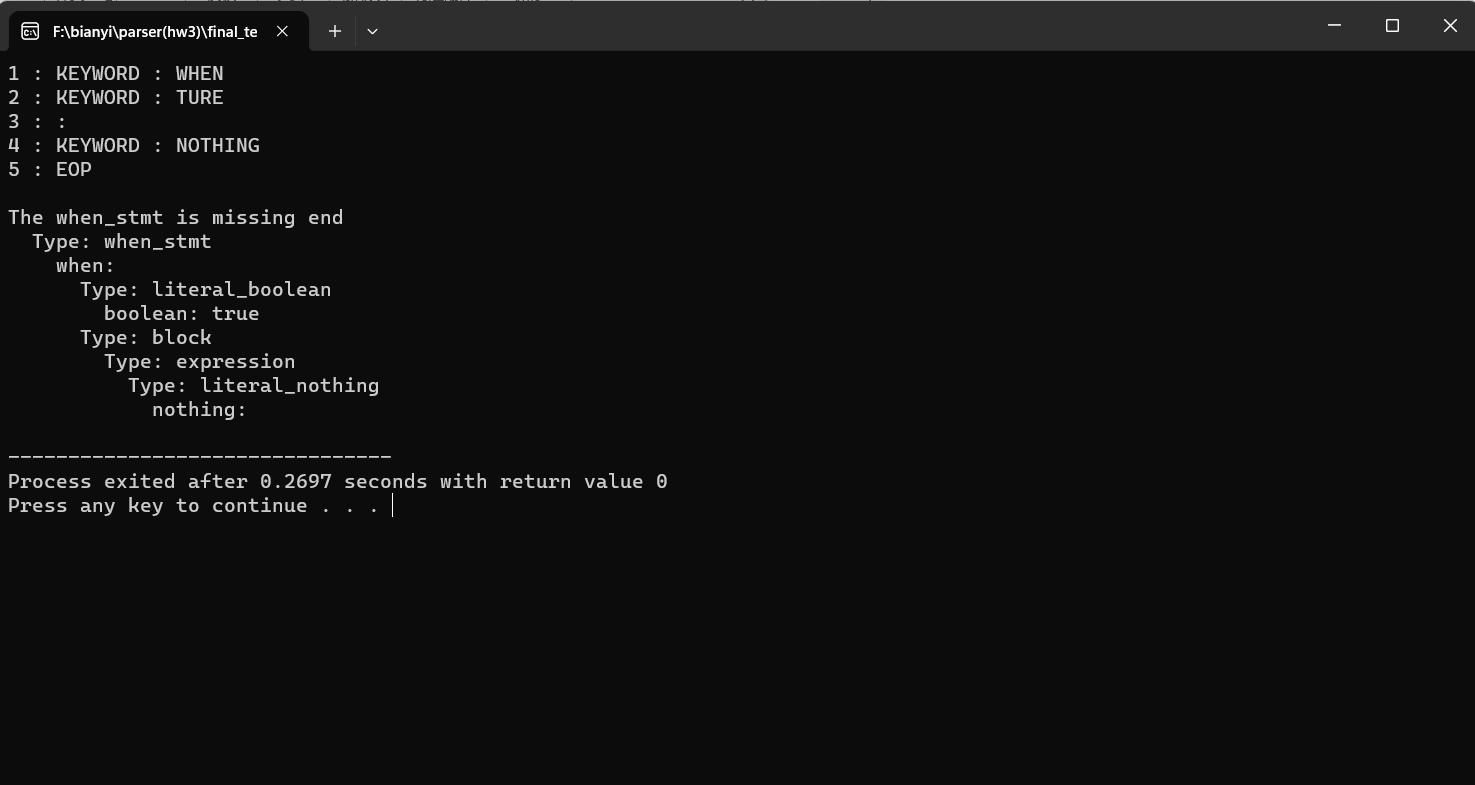




**Test results with syntex errors:**



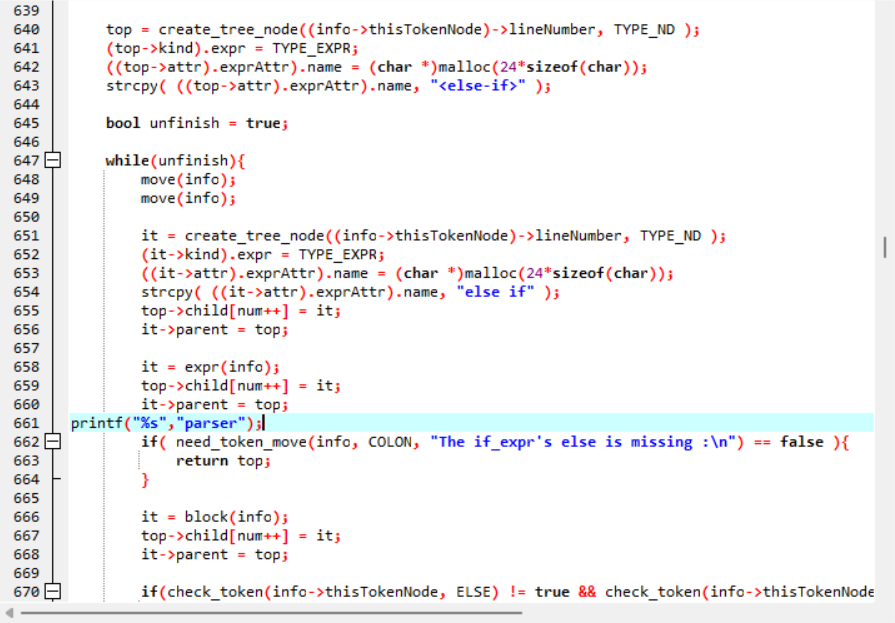




* **Descriptions(run and complie) :**

**The way I use to find where is the error that cause problems:**

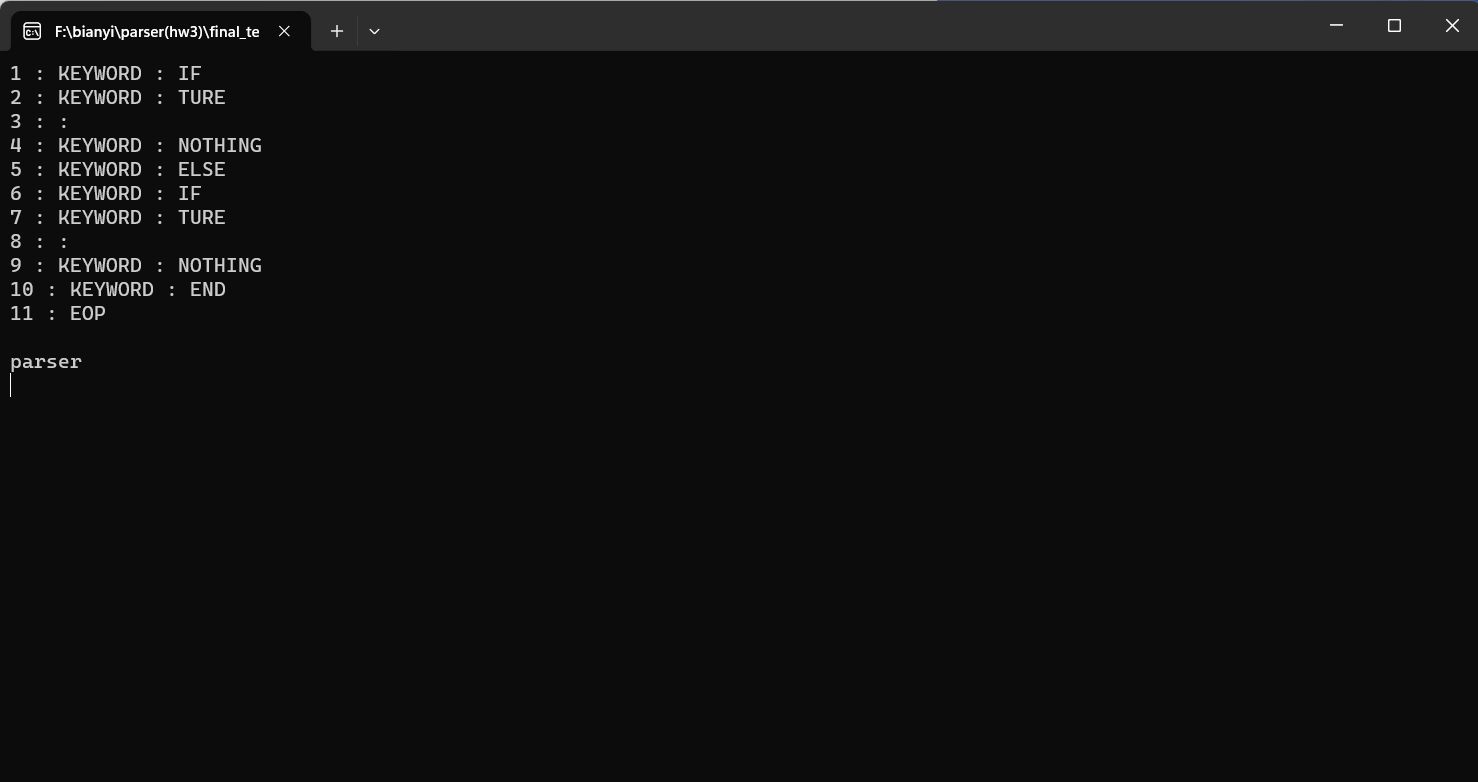
**During compling, sometimes I can’t get the expected parser tree from the list according to the grammar, so I use the printf(“%s”,”parser); to locate where the error occurred like the way shown in below:**



**and then use the text.txt file to print out the result.**

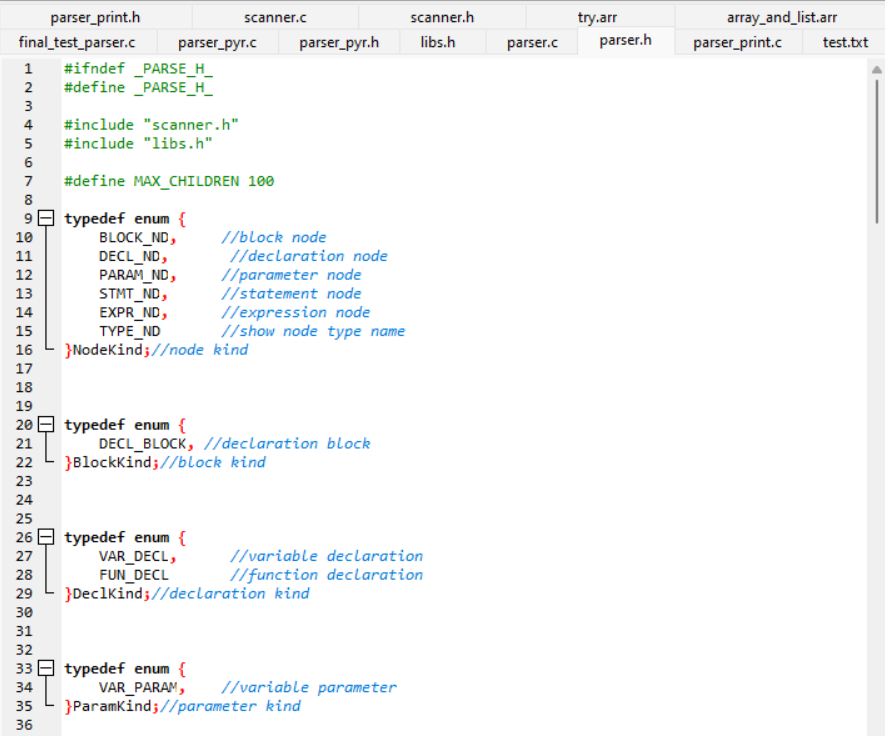
**If there is parser shown, the means the part above the printf sentence is ok; otherwise, it means something is wrong with the above codes.**

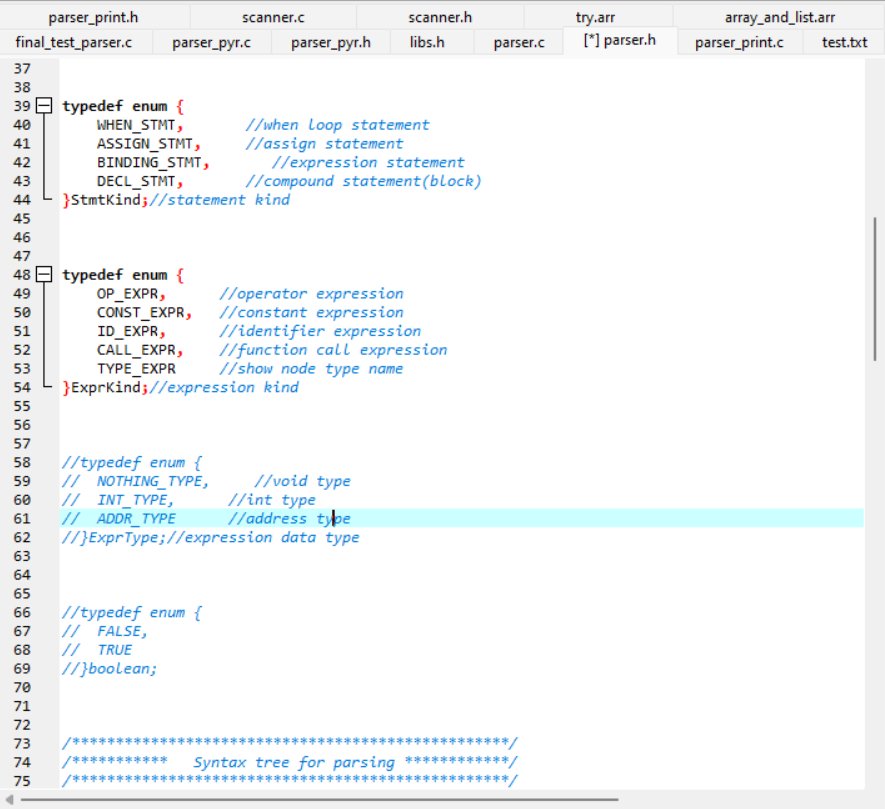
**With several checks around the codes that seems to have problems, this method helped me the narrow down to where the error was.**

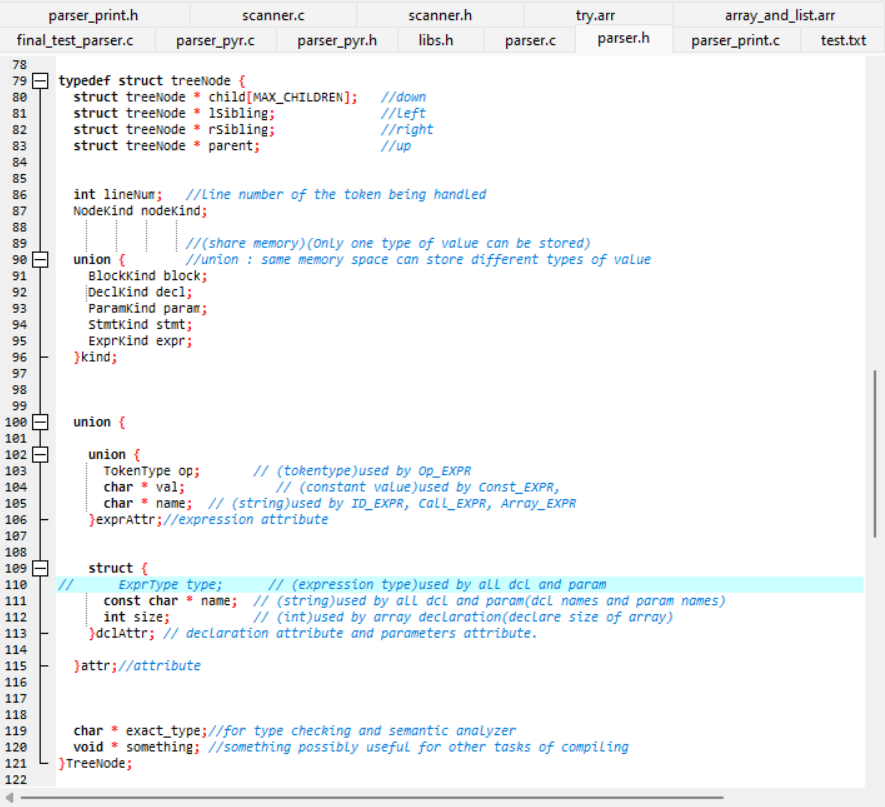


**The changes I make according to the provides helpful code:**

**[parser.h]**

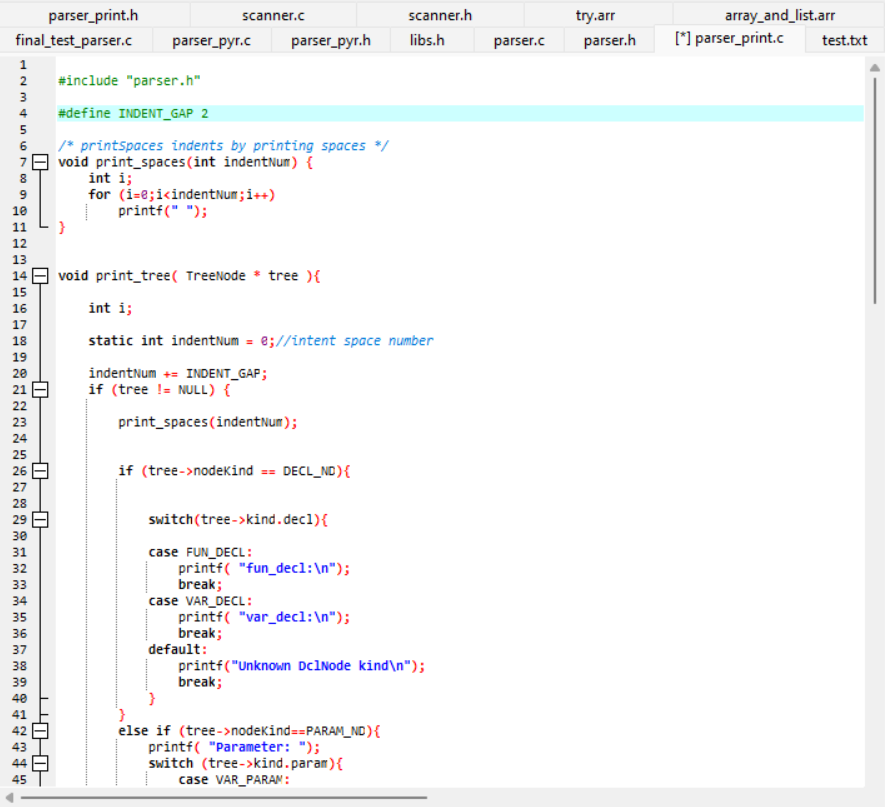


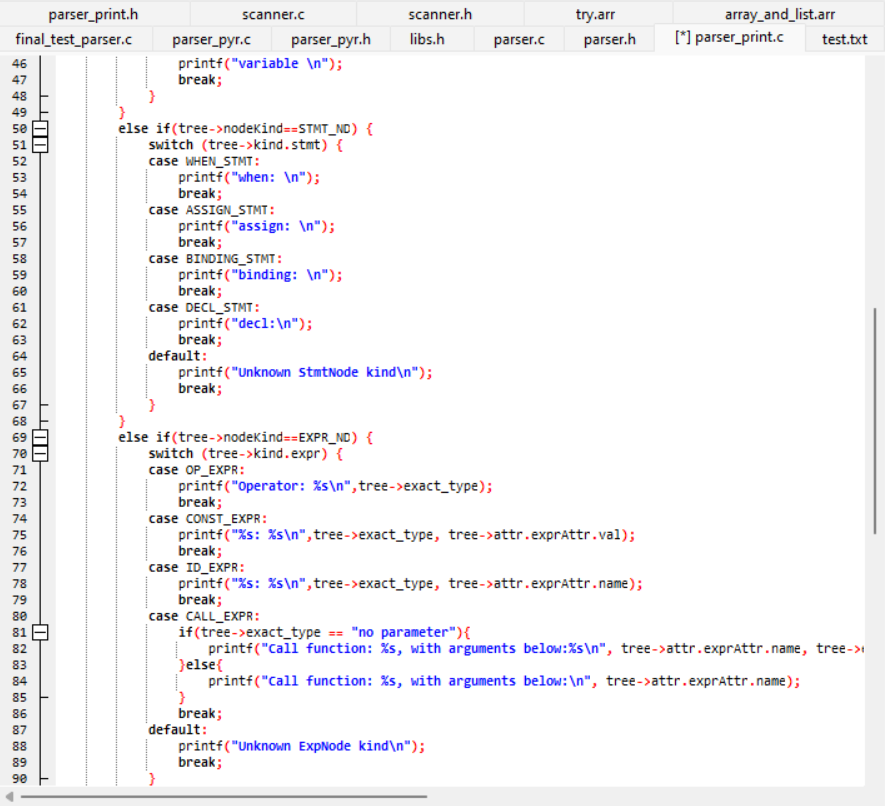


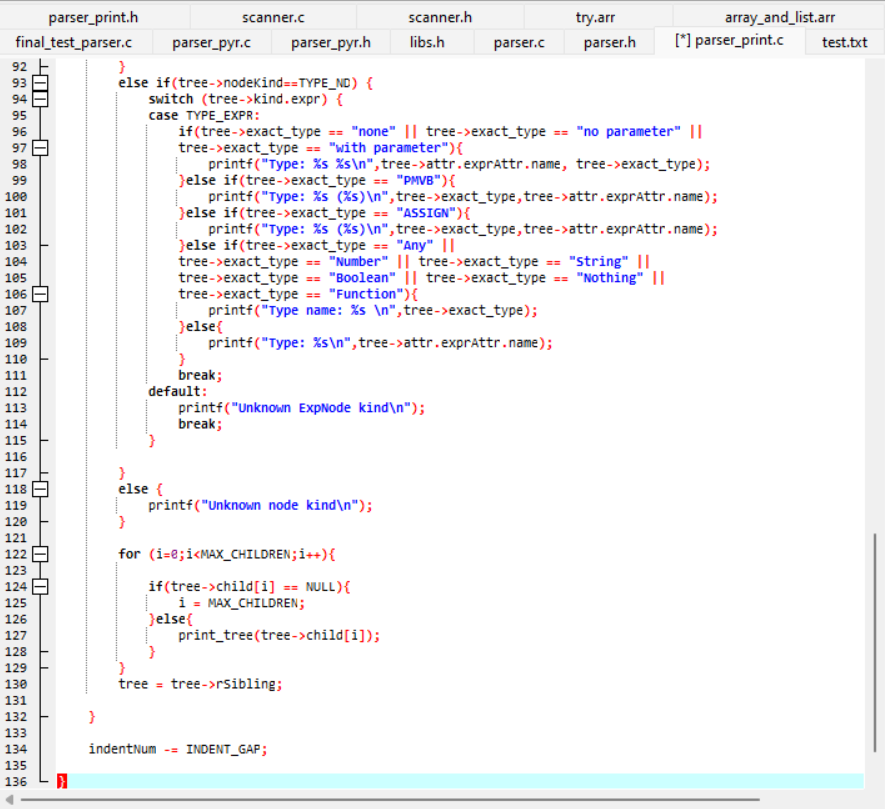


I add an char \* exact\_type; to show the details information of the current node

**[parser\_print.c]**







**The file I used to test the result:**

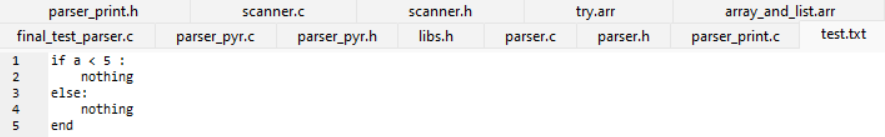


* **Problems :**

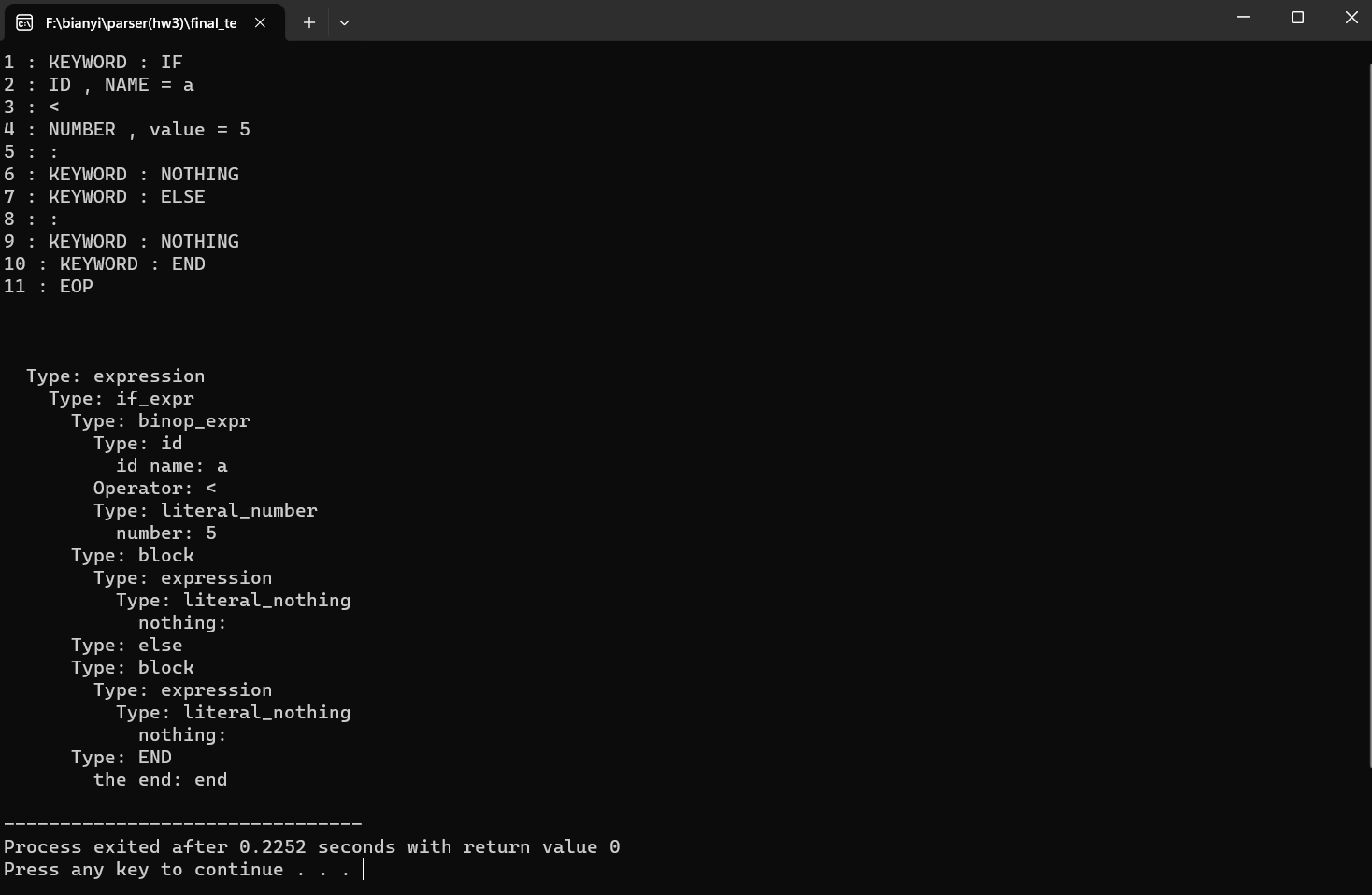
**According to the limited of time and my code ability, the above contents are all I have done. However, I still have many problems to be solved.**

1. **A parse tree can be produced from only one piece of code**

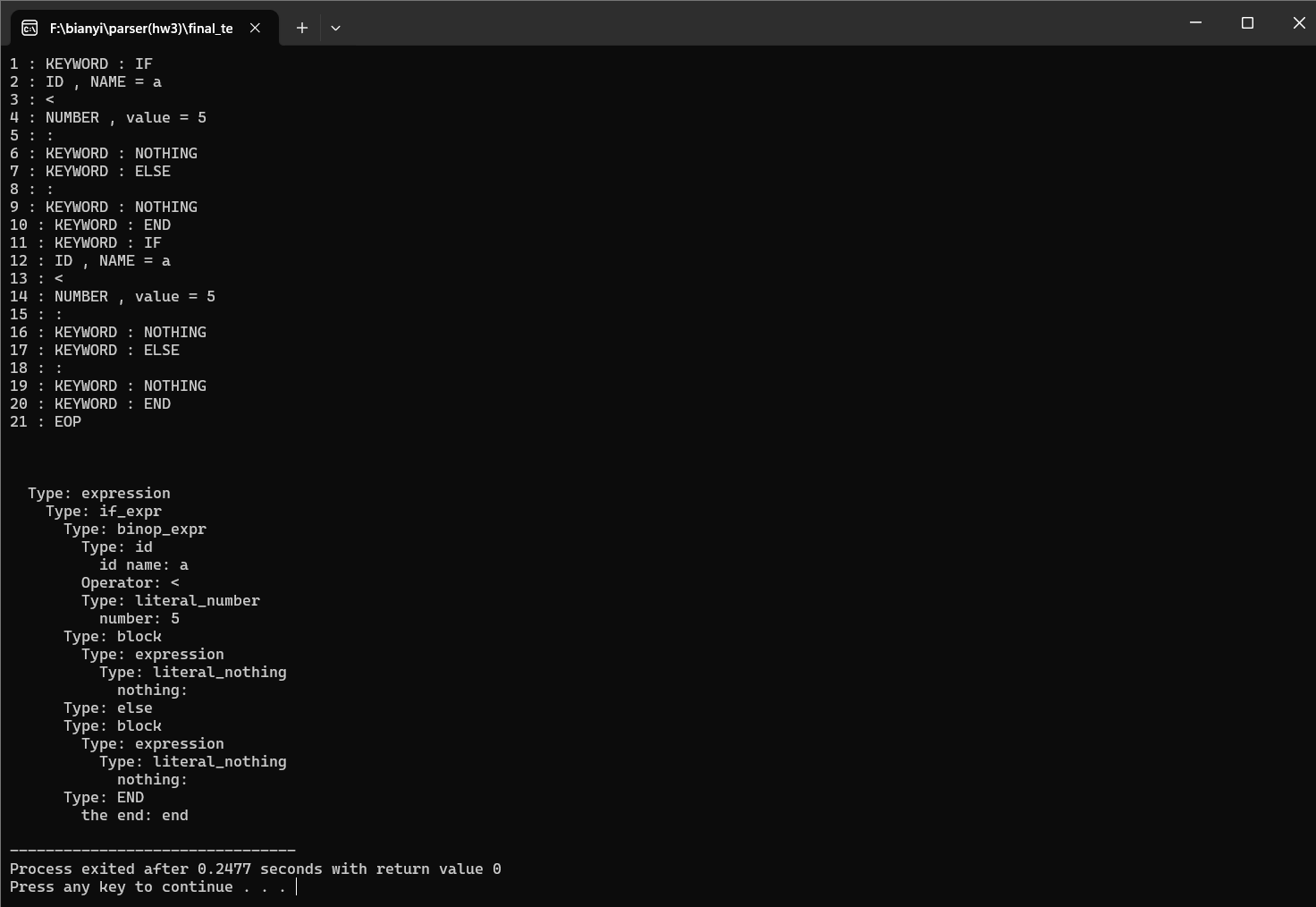
**Here is a test to show the problem**



**When there is only one piece of code in the program, the parser tree can be produced.**



**However, when there are two pieces of code, only one parser tree can be generated.**



1. **I am not sure the way to deal with these grammars :**

<expr> =>  <non-op-expr>  | <op-expr>

<non-op-expr> => <id> | <literal> | <fun-call> |

                   \( <expr> \) | <if-expr> | <block> |

                   <construct-expr> | <lambda-expr>

<op-expr> => <binop-expr> | <unop-expr>

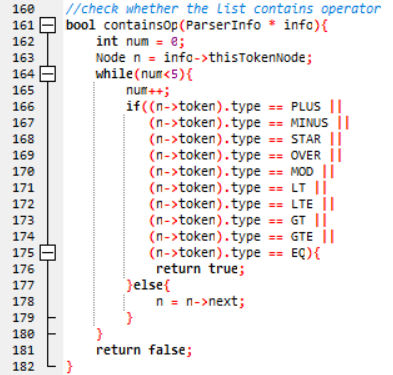
<unop-expr> => <unop> ( <non-op-expr> |  <paren-expr> )

<unop> => not

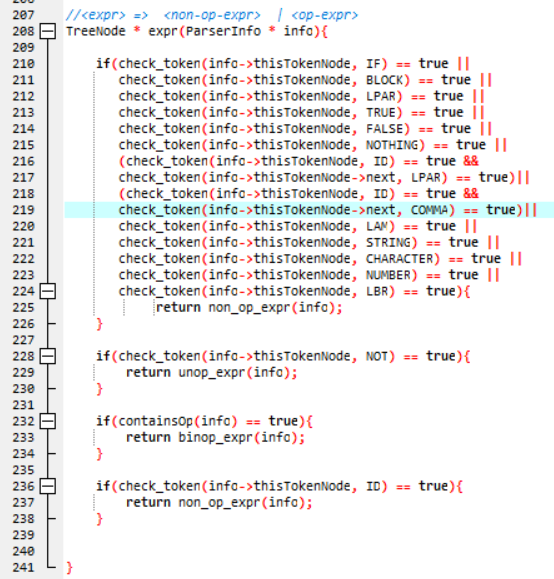
<binop-expr> => <non-op-expr> <binop> ( <non-op-expr> | <paren-expr> )

<binop> => [+ , - , \* , / , < , <= , > , >= ,  == ]

**I wrote a containsOP() function to check whether the coming list contains operator or not, and use it to distinguish the** <binop-expr> **and** <non-op-expr>



**And here is the expr() function**



**I know that the way I used will cause many problems if the number of tests are big enough but it’s the only way I can figure it out by myself. Hope to get more information about the solution in the future.**

1. **The parser is not perfect enough.**

**There are many grammars that I haven’t got time to finish and some problems that exists but I haven’t found it. There are still a lot of work to do in the future.**