Solidity\_language is used to read in solidity and convert it to C++ mode, solidity\_grammar defines an enumeration of types for type determination and conversion, solidity\_convert is responsible for specific conversions and handles traversals and conversions, solidity\_convert\_ literals is specifically responsible for converting integer boolean strings and hexadecimal, and pattern detects solidity code.

The solidity\_language class uses solidity\_convert to convert ASTs. solidity\_convert relies on solidity\_grammar to understand the structure of the Solidity code it is converting. solidity\_convert\_literal supports converting literals and is used by solidity\_convert.

**Solidity\_convert**

nlohmann::json &ast\_json; // json for Solidity AST. Use vector for multiple contracts

**bool solidity\_convertert::convert(){**

Perform pattern-based verification

Populate the context with symbols annotated based on each AST node, and hence prepare for the GOTO conversion.**}**

**Using:**

convert\_ast\_nodes：Iterate over and then process the nodes in the JSON AST tree.

get\_noncontract\_defition：Non-contractual definitions.

get\_struct\_class：Structure Definition.

add\_implicit\_constructor：Responsible for adding an implicit constructor if one is not explicitly defined in the contract.

multi\_transaction\_verification: Multi-transaction validation for –contract.

multi\_contract\_verification: Validate multiple contracts across the file.

**bool convert\_ast\_nodes(const nlohmann::json &contract\_def);**

**Using:**

get\_decl: used to process each declaration node in the AST. get decl in rule contract-body-element. get decl in rule variable-declaration-statement, e.g. function local declaration.

**bool get\_decl(const nlohmann::json &ast\_node, exprt &new\_expr);**

**using:**

bool get\_var\_decl(const nlohmann::json &ast\_node, exprt &new\_expr); Deals with variable declarations

bool get\_function\_definition(const nlohmann::json &ast\_node) //Deals with function definitions

bool get\_struct\_class(const nlohmann::json &ast\_node); Deals with structure declarations

bool get\_error\_definition(const nlohmann::json &ast\_node); Deals with error statements

//Handling implicit constructors

1. add\_implicit\_constructor()//Adding an implicit constructor to a class or structure
2. get\_implicit\_ctor\_call(exprt &new\_expr, const std::string &contract\_name)// Getting the invocation expression of an implicit constructor
3. get\_struct\_class\_fields(const nlohmann::json &ast\_node, struct\_typet &type)// Parses the fields in the structure definition and adds those fields to the internal type representation
4. get\_struct\_class\_method(const nlohmann::json &ast\_node, struct\_typet &type)// Parses the methods defined in the structure and processes them accordingly.
5. get\_access\_from\_decl(const nlohmann::json &ast\_node, struct\_typet::componentt &comp)// Extract the access rights information from the declaration and set the appropriate attributes
6. get\_block(const nlohmann::json &expr, exprt &new\_expr)// Parsing Code Blocks Consisting of Multiple Statements.
7. get\_statement(const nlohmann::json &block, exprt &new\_expr)
8. get\_binary\_operator\_expr(const nlohmann::json &expr, exprt &new\_expr)//Parsing binary arithmetic expressions.
9. get\_compound\_assign\_expr(const nlohmann::json &expr, exprt &new\_expr)// Parsing Compound Assignment Expressions
10. get\_unary\_operator\_expr(const nlohmann::json &expr, const nlohmann::json &literal\_type, exprt &new\_expr)// Parsing unary arithmetic expressions
11. get\_conditional\_operator\_expr(const nlohmann::json &expr, exprt &new\_expr)// Parsing Conditional Expressions
12. get\_cast\_expr(const nlohmann::json &cast\_expr, exprt &new\_expr, const nlohmann::json literal\_type = nullptr)// Parsing Type Conversion Expressions
13. get\_var\_decl\_ref(const nlohmann::json &decl, exprt &new\_expr)// Getting references to variable declarations
14. get\_func\_decl\_ref(const nlohmann::json &decl, exprt &new\_expr)// Getting a reference to a function declaration
15. get\_enum\_member\_ref(const nlohmann::json &decl, exprt &new\_expr)// Getting a reference to an enumeration member
16. get\_decl\_ref\_builtin(const nlohmann::json &decl, exprt &new\_expr)// Getting references to built-in functions or variables
17. get\_type\_description(const nlohmann::json &type\_name, typet &new\_type)// Parsing type descriptions and converting to internal type representations
18. get\_func\_decl\_ref\_type(const nlohmann::json &decl, typet &new\_type)// Getting information about the type of a function declaration
19. get\_array\_to\_pointer\_type(const nlohmann::json &decl, typet &new\_type)// Converting an array type to a pointer type
20. get\_elementary\_type\_name(const nlohmann::json &type\_name, typet &new\_type)// Get elementary type name
21. get\_parameter\_list(const nlohmann::json &type\_name, typet &new\_type)// Parsing the argument list of a function or method
22. get\_state\_var\_decl\_name(const nlohmann::json &ast\_node, std::string &name, std::string &id)// Getting State Variable Declarations
23. get\_var\_decl\_name(const nlohmann::json &ast\_node, std::string &name, std::string &id)// Get the name of the variable declaration
24. get\_function\_definition\_name(const nlohmann::json &ast\_node, std::string &name, std::string &id)// Get the name of the function definition
25. get\_constructor\_call(const nlohmann::json &ast\_node, exprt &new\_expr)// Parsing Constructor Calls
26. get\_current\_contract\_name(const nlohmann::json &ast\_node, std::string &contract\_name)// Get the name of the current contract
27. get\_empty\_array\_ref(const nlohmann::json &ast\_node, exprt &new\_expr)// Handling empty array references

**key function:**

**bool get\_expr(const nlohmann::json &expr, exprt &new\_expr);**

Function overloading, as a proxy or interface simplifier.

**bool get\_expr(const nlohmann::json &expr, const nlohmann::json &expr\_common\_type, exprt &new\_expr);**

Populate the out parameter with the expression based on the solidity expression grammar. More specifically, parse each expression in the AST json and convert it to a exprt ("new\_expr"). The expression may have sub-expression.

@param expr The expression that is to be converted to the IR

@param literal\_type Type information ast to create the the literal

type in the IR (only needed for when the expression is a literal).

A literal\_type is a "typeDescriptions" ast\_node.

we need this due to some info is missing in the child node.

@param new\_expr Out parameter to hold the conversion

@return true iff the conversion has failed

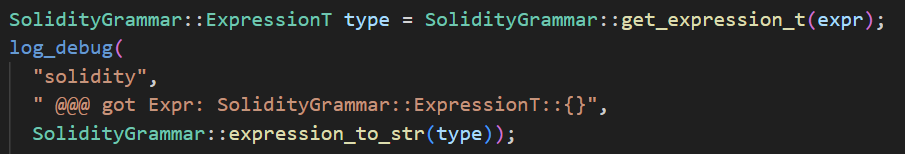
@return false iff the conversion was successful

**Details:**

locationt location;

get\_start\_location\_from\_stmt(expr, location);// Initialising location information

Get expression type.



Type Branch Handling:

Binary Operator: BinaryOperatorClass

Unary operators: UnaryOperatorClass

Conditional operator: ConditionalOperatorClass

Declared Reference Expressions: DeclRefExprClass

Literal: literral

Tuple expression: Tuple

Function Call: CallExprClass

Implicit conversion expression: ImplicitCastExprClass

Index Access: IndexAccess

Create new object: NewExpression

Member Calls (including Contract, Struct, Enum Member Calls): ContractMemberCall, StructMemberCall, EnumMemberCall

Type name expression: ElementaryTypeNameExpression

// line number and locations. Processing location information in AST nodes

1. get\_location\_from\_decl // Extracting location information from declared nodes
2. get\_start\_location\_from\_stmt // Extracting start position information from statement nodes
3. get\_final\_location\_from\_stmt // Extracting end position information from statement nodes
4. get\_line\_number // Extract line numbers based on AST nodes
5. add\_offset// Add Offset。
6. get\_src\_from\_json // some nodes may have "src" inside a member json object. we need to deal with them case by case based on the node type
7. move\_symbol\_to\_context
8. multi\_transaction\_verification// Verify Multi-Trading Conditions
9. multi\_contract\_verification// Validating Multi-Contract Structures

//Helper function to handle conversions in Json files

1. std::string get\_modulename\_from\_path(std::string path); Extract module name from file path。
2. std::string get\_filename\_from\_path(std::string path); Extracting filenames from full paths。
3. const nlohmann::json &solidity\_convertert::find\_decl\_ref(int ref\_decl\_id, std::string &contract\_name)// find declaration reference
4. const nlohmann::json &find\_constructor\_ref(int ref\_decl\_id); Look up nodes in the AST。
5. void convert\_expression\_to\_code(exprt &expr); Converting expression nodes into executable code form
6. bool check\_intrinsic\_function(const nlohmann::json &ast\_node)// Check if it is a built-in function

// Type and expression handling

1. make\_implicit\_cast\_expr//Create implicit type conversion expressions.
2. make\_return\_type\_from\_typet//Create return types from types.
3. make\_pointee\_type// Since Solidity function call node does not have enough information, we need to make a JSON object manually create a JSON object to complete the conversions of function to pointer decay
4. make\_array\_elementary\_type// Function used to extract the type of the array and its elements
5. make\_array\_to\_pointer\_type // Function to replace the content of ["typeIdentifier"] with "ArrayToPtr"
6. get\_array\_size //Get the array size from the type description.
7. is\_dyn\_array//Determines if the array is dynamic.
8. add\_dyn\_array\_size\_expr//Add a size expression to a dynamic array type.

//literal conversion functions.

bool convert\_integer\_literal(

const nlohmann::json &integer\_literal,

std::string the\_value,

exprt &dest); //integer conversion

bool convert\_bool\_literal(

const nlohmann::json &bool\_literal,

std::string the\_value,

exprt &dest); //boolean conversion

bool convert\_string\_literal(std::string the\_value, exprt &dest); //string conversion。

bool convert\_hex\_literal(std::string the\_value, exprt &dest, const int n = 0); //hexadecimal conversion

//Auxiliary data structures:

Mapping from the Contract\_id to the Contract\_Name

std::unordered\_map<int, std::string> exportedSymbolsList;

Inheritance Order Record <contract\_name, Contract\_id>

std::unordered\_map<std::string, std::vector<int>> linearizedBaseList;

Store the ast\_node["id"] of contract/struct/function/...

std::unordered\_map<int, std::string> scope\_map;

Private：

Receive basic types from solidity and convert them

get\_elementary\_type\_name\_uint: unsigned integer

get\_elementary\_type\_name\_int: signed integer

get\_elementary\_type\_name\_bytesn: byte sequence

**Solidity\_grammar**

enum ContractBodyElementT

{

VarDecl = 0, // rule variable-declaration

FunctionDef, // rule function-definition

StructDef, // rule struct-definition

EnumDef, // rule enum-definition

ErrorDef, // rule error-definition

ContractBodyElementTError

};// In a similar way, enumeration types are created to express the different elements of a solidity contract.

//The following function gets the enumeration type first：

ContractBodyElementT get\_contract\_body\_element\_t(const nlohmann::json &element)

TypeNameT get\_type\_name\_t(const nlohmann::json &type\_name)

ElementaryTypeNameT get\_elementary\_type\_name\_t(const nlohmann::json &type\_name)

ParameterListT get\_parameter\_list\_t(const nlohmann::json &type\_name)

BlockT get\_block\_t(const nlohmann::json &block)

StatementT get\_statement\_t(const nlohmann::json &stmt)

ExpressionT get\_expression\_t(const nlohmann::json &expr)

VarDeclStmtT get\_var\_decl\_stmt\_t(const nlohmann::json &stmt)

FunctionDeclRefT get\_func\_decl\_ref\_t(const nlohmann::json &decl)

ImplicitCastTypeT get\_implicit\_cast\_type\_t(std::string cast)

//Then the following functions convert the different enumeration types to strings：

const char\* contract\_body\_element\_to\_str(ContractBodyElementT type)

const char\* type\_name\_to\_str(TypeNameT type)

const char\* elementary\_type\_name\_to\_str(ElementaryTypeNameT type)

const char\* parameter\_list\_to\_str(ParameterListT type)

const char\* block\_to\_str(BlockT type)

const char\* statement\_to\_str(StatementT type)

const char\* expression\_to\_str(ExpressionT type)

const char\* var\_decl\_statement\_to\_str(VarDeclStmtT type)

const char\* func\_decl\_ref\_to\_str(FunctionDeclRefT type)

const char\* implicit\_cast\_type\_to\_str(ImplicitCastTypeT type)

//These functions calculate the type size：

unsigned int uint\_type\_name\_to\_size(ElementaryTypeNameT)

unsigned int int\_type\_name\_to\_size(ElementaryTypeNameT)

unsigned int bytesn\_type\_name\_to\_size(ElementaryTypeNameT)

//Operator acquisition, binary and unary

get\_expression\_t(const nlohmann::json &expr) // received expression

ExpressionT get\_expr\_operator\_t(const nlohmann::json &expr) //binary

ExpressionT get\_unary\_expr\_operator\_t(const nlohmann::json &expr, bool uo\_pre)//unary

VisibilityT get\_access\_t(const nlohmann::json &ast\_node); // should be an access modifier

**Solidity\_convert\_literals**

convert\_integer\_literal

convert\_bool\_literal

convert\_string\_literal

convert\_hex\_literal

//Converts integers, booleans, strings and hexadecimal to the types used internally.

**solidity\_language.cpp**

parse //Read the AST file

typecheck //check the syntax.

convert\_intrinsics//Mapping solidity syntax to C++ for post-processing

**pattern\_check.cpp**

safety check