Porcelli Gianluca

Formal languages and compilers

C++ to Python translator

The present work aims to translate a program understandable by a c++ compiler into a sequence of equivalent instructions (where possible) written in python language

Index

1. Design requirements

Type of managed information:

- integer variables;
- real variables;
- string variables;
- one-dimensional vectors;
- functions with or without parameters.

Implementable constructs:

- sequence of instructions;
- selection: if statement, if-then-else statement;
- iteration: do statement, while statement, for statement.

Operators:

- arithmetic operators: sum, subtraction, division, multiplication;
- conditional operators:>, <, ==,> =, <=,! =;
- logical operators: AND, OR, NOT.

Built-in functions:

- input function;
- output function.

Declaration and call of integer, real and string functions.

2. Design limits

- **Definition of variables**: The definition of variables for all types is envisaged. The assignment of an initial value is allowed for all data types except for the vectors, which will be automatically initialized with all elements equal to zero. A multiple variable declaration on the same line will be possible only for integer type variables.
  Note: The exponential form is not provided for float variables.
- **Assignment between variables**: The assignment between variables is always possible for the cases provided by the C ++ language. Respecting these constraints, errors due to bad assignments will be managed.
- **Assignment of a constant value**: The assignment of a constant value in the post declarative phase is not foreseen for non-integer variables (of course it is not even foreseen for arrays). To allow this operation it would be necessary to provide new nodes in the abstract syntax tree containing string and real constants with the related check and translation functions.
- **Expression statement**: it is possible to handle errors concerning incorrect assignments between variables of different types and binary operations in general.
- **Input function**: Entry patterns of one variable at a time are recognized; the case of insertion of a carrier (or one of its elements) is not evaluated. The acquisition of variables can only occur sequentially.
- **Output function**: The output function may contain variables of any type. It will also be possible to output a string constant to the user.

- **Declaration and call of a function**: The declarable functions in this translator can contain non-vectorial data types. There is a check on the number of arguments in the call phase without comparing the variable's type with the related formal parameter's type (this kind of error will not be notified). For instance, incompatibilities could occur with string data.
- **Not operator**: The logical and conditional operations will include all combination of logical operators but a "not" statement can be present in the expression only as a prefix. For ease of management it will not be possible to include this operator in nested conditions.
- **Return statement**: In the main function it is possible to leave out the return operation, but this is not allowable for the other functions. There isn't an apposite error message in case of a failed return statement.

3. Tricks in the translation phase

Variables initialization

Python variables don’t need to be declared but they must be initialized in order to be used. Declarations without initialization in source language will be converted into an initialization with 0 default value in the case of integer variables, 0.000000 in the case of float variables and for string variables with "null" value. In the source language the default of initial value depends on the compiler chosen.

For statement translation

The for loop used in python has a very different syntax from the one expected in C ++ language; for this reason only one example pattern is envisaged.

Do loop translation

The do loop is not provided in the python syntax but has been simulated by adding a selection operation within a while true loop and an appropriate break statement.

Input acquisition translation

It will be possible to acquire all types of data except one-dimensional arrays, because the control on the single element to be acquired can’t be managed.

Note: A different type of data has not been provided for arrays; for that reason it’s impossible to discriminate a bad input or output acquisition containing an array. In the symbol table vectorial data and simple data are collected in the same way.

Output statement translation

User visualization of non-vector data type and constant strings will be provided by this translator with the related error handling (one at a time). In a similar manner, all the other constant types could be printed by editing the related translation function.

Note: Looking at the abstract syntax tree is easy to denote that it is not be possible to avoid or detect the printing of data defined as vectors without the use of round brackets; the previous pattern will print the address containing the first element of the vector for c++ program while in python the entire content of the selected vector will be displayed.

Expression translation

Expressions like "x + = 1;" are provided in both languages for all types of variables, but if they are applied to string data an error in target language is produced; therefore if this operation has an element of string type as an argument, it is not translated into target language. The case in which the operand to the right of the equal is not a constant is also managed.

#warning

User alerts are provided in the form of python comments for any programming advice. They do not represent syntactic or semantic errors. The notified warnings are foreseen in case of:

- Lost **return statement** (at the end of the list) or if the return value type is different from the one declared by the function;
- **For_statement** pattern not provided by this translator;
- Cycle conditions containing assignment operations in the cycles **while** and **do**.

4. Main functions useful for implementation

AST node

*typedef struct **ast_node** {*

enum {

/ type of node */*

} type;

union {

/ definition of the structure about single nodes*/*

};

int type_id;

char fg;

} ast_node;

type_id: it saves the variable type contained in the node (integer, float, string).

fg: character that specify the visibility which can be global (g) or not (f).

Parsing functions

*int **parse_int_hex**(char const *s);*

It is called when this kind of input is recognized: 0 [xX] [a-fA-F0-9] +.

It performs the conversion from a vector of input characters containing a hexadecimal number with the equivalent decimal number stored in a integer variable.

int ***parse_int_oct****(char const *s);*

It’s called when this input sequence is recognized: 0 [0-7] +.

It performs the conversion from a vector of input characters containing a base eight number into an integer variable with the equivalent decimal number.

int ***parse_int****(char const *s);*

It takes care of calling up the appropriate parsing function by observing the base of the number.

float ***parse_float****(char const *s);*

It is called if one of these patterns is recognized:

1. {NUMBER} + \. {NUMBER} *

2. {NUMBER} * \. {NUMBER} +

In both cases the parse_float function is called and returns the real number corresponding to the translated string.

*char * **strdup** (const char * s);*

Return a pointer to a string of bytes with null termination, which is a duplicate of the string indicated by "s". The returned pointer must be released at the end of the procedure to avoid memory leaks. If an error occurs, a null pointer is returned and a report_error can be set. This function is called every time a string constant between double quotes is recognized. Any spaces in the character string will also be included.

Tables

symbol table

```
struct sym_node
{
  char str[1024];
  int sym_id;
  struct sym_node *next;
};
struct sym_node *sym_table = NULL;
static int last_sym_id = 0;
```

The symbol table is initially empty and every node entered is thus defined.

```
struct sym_node* new_sym_node(const char *s, struct sym_node *next)
{
    struct sym_node *p = malloc(sizeof(struct sym_node));
    strcpy(p->str, s);
    p->sym_id = ++last_sym_id;
    p->next = next;
    return p;
}
```

Dynamically allocates a new node of the symbol table. Id generation is managed simply by increasing an integer. last_sym_id stores the last integer assigned as id. The name of the instantiated node is passed to it by argument in the function. Each entry in the table is related to the next entry in order to create a sequence.

*int **sym** (const char *s);*

This function is called when a variable identifier is detected from the lexer. The function takes care of searching for the variable name in the symbol table by comparing the "str" field; if it is found, the corresponding integer id will be returned, otherwise a new element of the table will be created returning the corresponding id similarly.

char **symname**(int sym_id);*

The inverse work of the sym function is performed by the symname procedure which receives the id of the symbol table as an argument and returns its related declarative name. If the search was not successful, an error message would be generated. It is used to detect the duplicate declaration or the usage of a not declared variable.

Scope

Vscope

*struct **vscope***

{

    *struct vscope *parent;*

    *struct vscope_node *list;*

};

NOTE: if (current_vscope->parent == NULL) n->fg = 'g';

If no v_scope has been initialized the variable is considered global.

List: field containing the last node added to the v_scope updated every time a node is inserted. From this node it will be possible to scroll through all the nodes previously added.

*struct **vscope_node***

{

*struct vscope *scope;*

int sym_id;

int type_id;

int offset;

*struct vscope_node *next;*

};

Offset: represents the space occupied by the node. Each entry of a new variable is decreased.

Next: pointer to the next node in sequence.

*void **begin_vscope**();*

This function instance the root node of the variable scope. The initial node is originally empty and will subsequently contain a list of child nodes. Set as "v_scope parent" the current "v_scope". It will be recalled whenever a new visibility space is defined, delimited by the context of a function.

*void **end_vscope**() ;*

I go back up the tree of a node to identify the end of the previously described context.

*void **vscope_addnode**(struct vscope *v, int sym_id, int type_id, int offset);*

This function will be called whenever the definition of a variable within a context (field of vision) will be required. The new node inserted is the next one in the list of the current_vscope and the new node entered is assigned to the list field of the current_vscope.

*static struct vscope_node * **find_vnode_rec(**struct vscope *c_vscope, int sym_id);*

It deals with searching for the context to which a variable belongs by providing the name of the scope in which to search and the sym_id. If the context does not exist an error is returned, otherwise the pointer to the corresponding v_scope is returned. In the event of a mismatch, the variable in the parent scope is searched by retracing the entire scope tree recursively.

*struct vscope_node * **find_vnode**(int sym_id);*

It performs the same search defined previously but does not require the passage of the v_scope from which to start the search for the variable but by default it starts the recursive procedure from the current_vscope.

*enum **TYPE** {FUNC};*

The type of node can therefore be represented by an integer value (variable used to discriminate integers, real and strings data) or from the **as_func** structure:

```
struct type_node
{
  int type_id;
  enum TYPE ty;
    struct
    {
      int ret_type;
      int para_num;
      int para_type[1000];
    } as_func;
  struct type_node *next;
};
```

ret_type: this field is defined by the parser when the function is declared based on the return type.

In this example the only purpose of the structures described is to verify that the type of data defined is present in those provided by the translator. Otherwise an error will be generated.

struct type_node **find_type_node**(int type_id);*

Search for the type of variable corresponding to the type_id. If it's found, it returns the pointer to the found node, otherwise an exception will be started.

Each new function definition is defined a new type_table entry containing the following information:

```
int func_type(ast_node *funchead)
{
   struct type_node *n = malloc(sizeof(struct type_node));
   n->type_id = ++last_type_id;
   n->ty = FUNC;
   n->as_func.para_num = 0;
   ast_node *p = funchead->funchead.paras;
   for (; p; p = p->list.tail)
   {
      n->as_func.para_type[n->as_func.para_num++] = p->list.head->para.para_type;
   }
   n->next = type_table;
   type_table = n;
   return n->type_id;
}
```

Last_type_id: Contains an id of a different type for each new defined function. The initial value of this variable is 3.

5.Description of the translation process

static ast_node ***g_parse_tree**;

Definition of the node used to explore the AST (after having previously created the tree with yyparse. Only if at least one tree node is found, the following functions are subsequently called:

1. check_semantics(g_parse_tree);
2. transtext_ast(g_parse_tree);
3. print_ast(g_parse_tree);

1)**check_semantics**(g_parse_tree);

This procedure calls the check_ast function that uses "static void init_ast_checkers ()" for associates a check function to each type of node. For each node, by checking the field containing the node type, it will call the corresponding check function. Each check function will recursively call the check_ast (ast) for control all child nodes. In case of absence of nodes the check procedure can be considered completed.

typedef void (*ast_checker)(ast_node *n);

Defines a function pointer type that returns no value but to which an ast_node is passed as an argument.

In this case a vector of these functions is defined as one will be called for each different ast_node:

static ast_checker **g_ast_checkers**[AST_TYPE_LIMIT];

This vector contains one parsing function for each type of node provided by the translator.

static const char* **g_ast_names**[AST_TYPE_LIMIT];

Another vector containing the related name of the parsing function.

2) **transtext_ast**(g_parse_tree);

Similarly to what was said for the previous check procedure, the same dynamics are defined to implement the translation process.

3) **print_ast**(g_parse_tree);

The print_ast function deals to the translation of the AST produced by BISON into an html list. An .htm file is created containing all nodes described by its fields and the related child nodes.

6. Useful procedure for inserting a statement

c++2python.l

Recognize a token using regular expressions.

c++2python.h

This file must include the type of **ast_node** (in **enum**erative mode) and its composition (in the **union** region). Furthermore the functions used in the parser for the insertion of the new node will be declared and they will be exported with the **extern** directive.

c++2python.y

The parser must contain the tokens (**% token**) used in the tree structures and the relative types (**% type <n>**) of nodes that will be described later. Finally, it will be necessary to specify how the **parser** should recognize the new statement.

ast.c

The file containing the AST must be explicit about what the **parsing functions** defined in file.y will do.

check_ast.c

In the function **init_ast_checkers ()** the individual checking functions will be defined for each different type of node. Only after recognizing the node, the compiler will follow the appropriate procedure.

translate_ast.c

In the function **init_ast_translators ()** the individual translation procedures must be explained. Only after recognizing the node, the compiler will follow the appropriate translation procedure.

print_ast.c

It isn’t a fundamental module but it would help in the case of a search for an error in the constitution of the syntactic tree.