Programming language translation: An overview

Duy Anh D. Hoang
FPT University

***Abstract*—**

***Index Terms*—Programming languages, lexical analysis, syntax analysis, semantic analysis, code generation, compiler, interpreter, abstract syntax tree, bytecode, assembly.**

I. Introduction

-

II. Evolution of programming languages

-

III. Methods used for translating programming languages: Compilation and Interpretation

-

IV. Principles of language: Essential definitions for programming language translation

-

A. Syntax

-

B. Semantics

-

V. Translation process: Compilation

-

A. Syntax analyzer

-

B. Semantic analyzer

-

C. Code generation

-

VI. Practical demonstration in translating high-level programming language

A. C programming language

Source program

```c
#include <stdio.h>


int main() {
        printf("Hello, World!\n");

        return 0;
}
```

- Compiler: GCC

1) Lexical analysis: The GCC's main function is to analyze the source code and divide it into tokens, with instructions closely intertwined within the compiler and primarily linked to the compilation procedure. A helpful way to find keywords is by referring to detailed language documentation like the GNU C REFERENCE MANUAL [1] and C REFERENCE [2]. These resources do not cover lexical analysis in a theoretical manner but provide guidance on utilizing the compiler to influence the compilation process and leveraging the programming language.

2) Syntax analysis: The GCC compiler offers the "-fdump-tree-original-raw" combined flag to generate a thorough linear representation of the Abstract Syntax Tree (AST) used in the compilation process. The result file "a-hello.c.005t.original" is quite comprehensive with a lot of information about the program.

File: a-hello.c.005t.original

```
;; Function main (null)
;; enabled by -tree-original

@1      statement_list   0    : @2        1    : @3
@2      bind_expr        type: @4         body: @5
@3      return_expr      type: @4         expr: @6
@4      void_type        name: @7         algn: 8
@5      statement_list   0    : @8        1    : @9
...
@56     integer_type     name: @57        size: @30
                                          algn: 64
                         prec: 64
                         sign: unsigned
                         min : @54        max : @58
@57     identifier_node  strg: sizetype   lngt: 8
@58     integer_cst      type: @56        int: -1
```

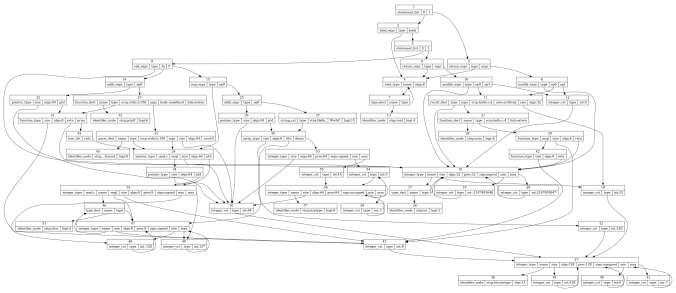
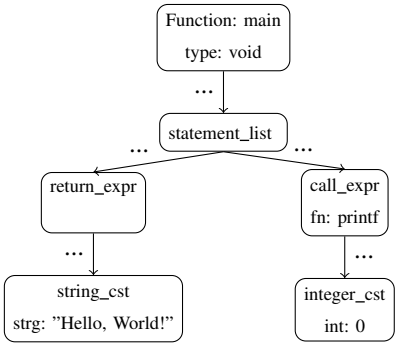


Fig. 1. AST of hello.c.

The graph view of this AST is (Fig. 1.), therefore, super detailed. A more straightforward demonstration could be the following diagram.



3) Semantic analysis: The AST generated in the syntax analysis phase is used for semantic checking. To assure semantic consistency, the Linux "man" command and other detailed documentation explain the functions and types in detail, even if much of the work is not visible. They offer not just the standard language information but also documentation for other libraries. In practical usage, language manuals are helpful because one can not assume how something actually works. For instance, it is handy to use the method "print(1)" in Python to display the number 1 on the screen. However, when using "printf()" in C, calling "printf(1)" will result in a warning during compilation and a segmentation fault during execution.

Semantically incorrect program

```c
#include <stdio.h>


int main() {
        printf(1);

        return 0;
}
```

Output

```
gcc main.c -o main
main.c: In function `main':
main.c:5:16: warning: passing argument 1 of `printf'
                      makes pointer from
                      integer without a
                      cast [-Wint-conversion]

    5 |         printf(1);
      |                ^
      |                |
      |                int
In file included from main.c:1:
         /usr/include/stdio.h:356:43:
    note: expected `const char * restrict'
          but argument is of type `int'

  356 | extern int printf
      | (const char *__restrict __format, ...);
      |   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

main.c:5:9: warning: format not a string literal
                     and no format arguments
                     [-Wformat-security]

    5 |         printf(1);
      |         ^~~~~~

./main
[1]29653 segmentation fault (core dumped)  ./main
```

This occurs because the "printf" function is documented to expect an argument of type "const char *format", which is a string with optional formatting.

4) Code generation: The syntactically correct source code is compiled into the executable file, making it comprehensible and executable by a computer. To view the machine code, you can use the "xxd" command to generate a hexadecimal dump of the executable file.

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000
00000010: 0300 3e00 0100 0000 6010 0000 0000 0000
00000020: 4000 0000 0000 0000 9836 0000 0000 0000
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00
00000040: 0600 0000 0400 0000 4000 0000 0000 0000
00000050: 4000 0000 0000 0000 4000 0000 0000 0000
...
00003e20: 0000 0000 0000 0000 0000 0000 0000 0000
00003e30: 7b35 0000 0000 0000 1a01 0000 0000 0000
00003e40: 0000 0000 0000 0000 0100 0000 0000 0000
00003e50: 0000 0000 0000 0000
```

The hexadecimal instructions consist of around one thousand lines to execute five lines of C code. For a clearer understanding, programmers could examine the assembly representation of the file, which consists of over a hundred lines of code by using "objdump -d hello".

```
hello:     file format elf64-x86-64
Disassembly of section .init:

0000000000001000 <_init>:
    1000:   f3 0f 1e fa
            endbr64
    1004:   48 83 ec 08
            sub    $0x8,%rsp
    1008:   48 8b 05 d9 2f 00 00
            mov    0x2fd9(%rip),%rax
            # 3fe8 <__gmon_start__@Base>
    100f:   48 85 c0
            test   %rax,%rax
    1012:   74 02
            je     1016 <_init+0x16>
    1014:   ff d0
            call   *%rax
    ...
0000000000001168 <_fini>:
    1168:   f3 0f 1e fa
            endbr64
    116c:   48 83 ec 08
            sub    $0x8,%rsp
```

```
    1170:    48 83 c4 08
             add     $0x8,%rsp
    1174:    c3
             ret
```

Despite the difficulties, the code's output is anticipated from the start and is straightforward. Displaying the text "Hello, World!" on the screen signifies the completion of converting a high-level language into machine code execution.

B. Python programming language

CPython, a widely used interpreter for Python, is written in C and is activated by typing python or python3 in the terminal. The python translation process is managed internally, similar to C. Python language provides various number of libraries that reveal those hidden processes, which can be accessed by default or installed via pip, a Python package installer. Package documentation can be accessed on the pypi website or locally with pydoc.

1) Lexical analysis:

Script

```
import tokenize
from io import BytesIO

code = 'print("Hello, World!")'
tokens = tokenize.tokenize(BytesIO(code.encode())
                                        .readline)

for token in tokens:
    print(token)
```

Output

```
TokenInfo(
    type=63 (ENCODING),
    string='utf-8',
    start=(0, 0),
    end=(0, 0),
    line=''
    )
...
```

2) Syntax analysis and semantic analysis:

Script

```
import ast

code_ast = ast.parse("print('Hello, World!')")
print(ast.dump(code_ast, indent=4))
```

Output

```
Module(
    body=[
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    Constant(value='Hello, World!')
                        ],
                keywords=[]))],
    type_ignores=[])
```

3) Code generation: Python is an interpreted language; therefore, in this last step, translated bytecode is handled by the Python virtual machine, which takes most of the heavy lifting work.

Script

```
import dis

compiled_code = compile("print('Hello, World!')",
                        '<string>',
                        'exec')
dis.dis(compiled_code)

print(compiled_code)
```

Output

```
  0           0 RESUME                   0

  1           2 PUSH_NULL
              4 LOAD_NAME                0 (print)
              6 LOAD_CONST               0 ('Hello,
                                           World!')
              8 PRECALL                  1
             12 CALL                     1
             22 POP_TOP
             24 LOAD_CONST               1 (None)
             26 RETURN_VALUE
<code object <module> at 0x000001C6ED80E790,
file "<string>", line 1>
```

C. Comparative Analysis

Compiled languages like C are often used for system-level programming, applications where performance is critical, and situations where direct hardware manipulation is required. Compilation allows for extensive code analysis and optimization by the compiler, potentially resulting in faster and more efficient executables.

Interpreted languages like Python excel in web development, data analysis, scripting, and rapid prototyping, where development speed and portability are more important than raw execution speed. Interpretation focuses on flexibility and ease of use, with performance optimizations typically happening at a higher level or relying on just-in-time compilation techniques.

VII. Conclusion

References

[1] Free Software Foundation. "The GNU C Reference Manual" (accessed February 23, 2024). [Online]. GNU Operating System. Available: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

[2] Cppreference.com . "C reference" (accessed February 23, 2024). [Online]. Cppreference.com. Available: https://en.cppreference.com/w/c