50006 - Compilers - (Prof Kelly) Lecture 1

Oliver Killane

05/01/22

Lecture Recording

Lecture recording is available here

Course Introduction

Lecturers

- **Paul Kelly**
  - Introduction to syntax analysis & toy compiler
  - Code Generation
  - Generating better code using registers
  - Register Allocation
  - Optimisation and data flow analysis
  - Loop optimisations
- **Naranker Dulay**
  - Lexical Analysis (characters to tokens)
  - LR (Bottom-up) parsers (tokens to AST)
  - LL (top-down parsing)
  - Semantic Analysis, checking program validity
  - Runtime organisation (memory)

Materials

- Scientia
- Course Textbook
- Course Webpages for Paul Kelly and Naranker Dulay

Recommended Textbooks:

- **The Dragon Book: 'Compilers: Principles, Techniques and Tools'**
  The definitive book on compilers. Called the dragon book due to its front illustration (allusion to taming the dragon of compiler complexity).

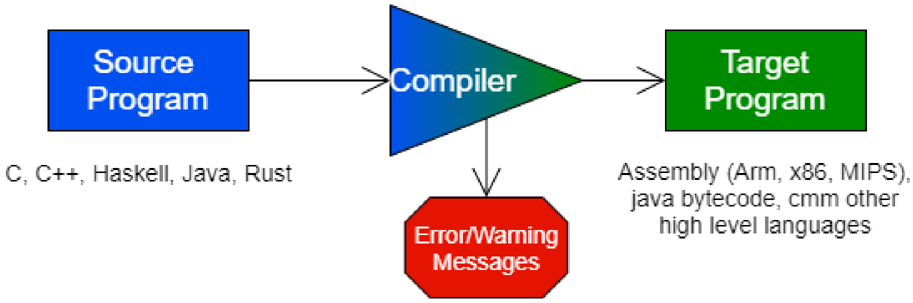
  Has all required information and lots more (very thick book). (Amazon link)
- **'Modern Compiler Implementation in Java'**
  Details a java compiler implementation, with reasoning behind design choiced. (Amazon link)
- **Most Recommended: 'Engineering a Compiler'**
  Most closely follows course & well reviewed. (Amazon link)

Compilers Introduction

A particular class of program called **language processors**. Compilers can be:

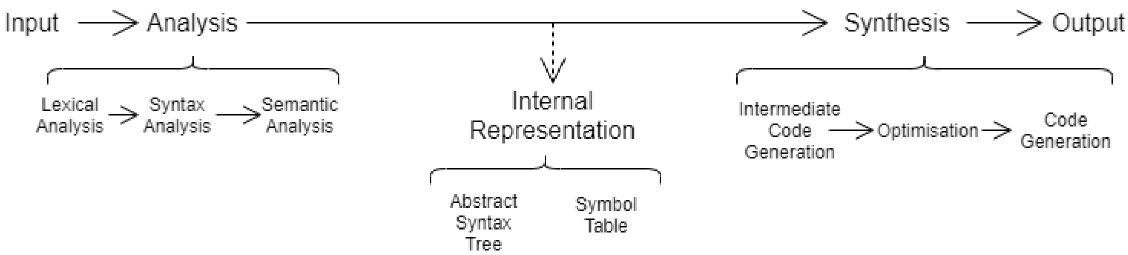
- **Processes** programs written in some language.
- **Writes** programs in some language.
- **Translates** programs from one language to an equivalent program in another. (e.g from hier to lower level language)

A tool to enable programs written using high-level concepts to be translated into a low-level implementation.



- Typically from a high level language to a lower level one.
- Error messages ensure source program can be fixed easily when failing to compile.
- Can compiler from & to a high-level language (e.g compile to C, or between languages)
- Analyse the source program to provide the programmer useful information.

Basic Compiler Structure



- **Input** Take program written in some language.
- **Analysis**
  Create an internal representation (IR) of the source which encodes its meaning (semantics).

  This often starts with a tree (abstract syntax tree), but graphs may be used to represent control flow.

  - **Lexical Analysis**
    Tokenization, converting text to a series of tokens representing keywords, identifiers, etc.
  - **Syntax Analysis**
    Analysis of the nested structure of a program (e.g if-else within if-else within loop block).
  - **Semantic Analysis**
    Analysing the meaning of a program such as type checking, determining overloading (e.g is + between variables correct & what action (e.g add integers, or concantenate strings)).
- **Synthesis**
  Use the IR to create the program in the target language (retaining the same meaning/semantics).

  The IR may be analysed and transformed to provide extra information to the programmer (e.g warnings) as well as to optimise (e.g inlining, loop hoisting, loop unrolling, removing dead code).

  - **Intermediate Code Generation**
    Encodes more information, and is used in sphisticated compilers for optimisation. For example determinign loop invariant code (code that can be hoisted out of the loop & only be evaluated once).
  - **Optimisation**
    Improving code performance without altering its meaning.

- **Output** Output the target program. (e.g x86 assembly or Java bytecode)

Symbol Table

Contains information on all declared identifiers, and is used to check its use (e.g type checking) and generate code for access.

- Name
- Type of identifier (function, class, variable, constant)
- Size (Memory to reserve & where (e.g stack, data segment, heap))

Code

```c
#include <stdio.h>
#include <stdlib.h>


int a = 6;

void pretty_print(int num) {
    printf("The number is: %d", num);
}

int another_cool_function(int* num) {
    *num++;
    return *num - 1;
}

int main(int argc, char **argv) {
    int b = 7;

    pretty_print(a + b);

    return EXIT_SUCCESS;
}
```

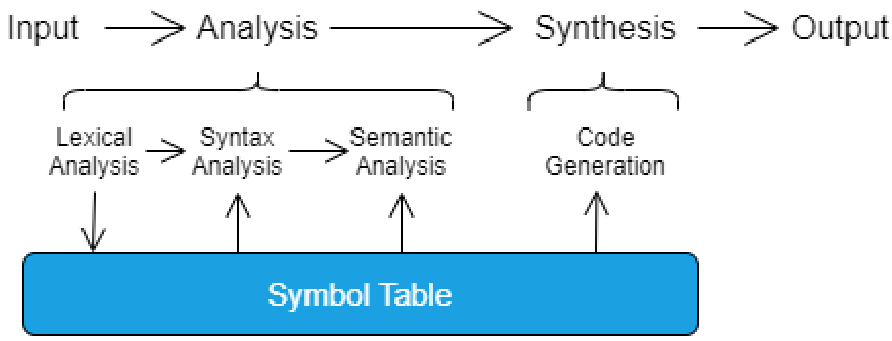
GCC symbol table

```
0000000000003dc8 d _DYNAMIC
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000000021b4 r __FRAME_END__
0000000000002018 r __GNU_EH_FRAME_HDR
0000000000004018 D __TMC_END__
0000000000004014 B __bss_start
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000001100 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003db8 d __frame_dummy_init_array_entry
                 w __gmon_start__
0000000000003dc0 d __init_array_end
0000000000003db8 d __init_array_start
0000000000001240 T __libc_csu_fini
00000000000011d0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000004014 D _edata
0000000000004018 B _end
0000000000001248 T _fini
0000000000001000 t _init
0000000000001060 T _start
0000000000004010 D a
0000000000001171 T another_cool_function
0000000000004014 b completed.8060
0000000000004000 W data_start
0000000000001090 t deregister_tm_clones
0000000000001140 t frame_dummy
0000000000001194 T main
0000000000001149 T pretty_print
                 U printf@@GLIBC_2.2.5
00000000000010c0 t register_tm_clones
```

GCC Symbol Table

Simply compile, and then use **nm** on the resulting binary to get a basic representation of GCC's symbol table (that is included with the executable).

The symbol table is used in different ways through different phases of the compiler:



Code

```c
int a;
int b;

void test_fun(void) {
    a = b+42;
}
```

⇒

Assembly

```asm
test_fun:
        movl    b(%rip), %eax
        addl    $42, %eax
        movl    %eax, a(%rip)
        ret
b:
        .zero   4
a:
        .zero   4
```

Boilerplate

Not lots of boilerplate assembly is ommited, if you try this locally you will see this.

GodBolt

The compiler explorer is a great tool for generating & exploring assembly generated by compilers.

Compiler Phases

We can show the phases of the compiler through an example. Converting **C** code into **AT&T/GAS** assembly.

$int_{space}a;_{newline}\ int_{space}b;_{newline}\ _{newline}void_{space}text\_fun(void)_{space}\{_{newline}a_{space} =_{space} b{+}42;_{newline}\ \}$

[INTtok,
IDENTtok(pointer to 'a' entry in symbol table),
SEMICOLONtok,
INTtok,
IDENTtok(pointer to 'b' entry in symbol table),
SEMICOLONtok,
VOIDtok,
IDENTtok(pointer to 'test_fun' entry in symbol table),
LBRACKETtok,
VOIDtok,
RBRACKETtok,
LCURLYBRACKETtok,
IDENTtok(pointer to 'a' entry in symbol table),

EQtok,
IDENTtok(pointer to 'b' entry in symbol table),
PLUStok,
CONSTINTtok(123),
SEMICOLONtok,
RCURLYBRACKETtok]

Lexical Analysis

Also known as tokenization. The input sequence of characters is tokenized and any new identifiers added to the symbol table during tokenization.

Tokens to identifiers include pointers to their corresponding entry in the symbol table.

Syntax Analysis

Also known as parsing. Using a grammatical structure to generate an **Abstract Syntax Tree** from the sequence of tokens provided by the previous stage.

1. Ensure the structure is correct, report an error otherwise.
2. Build **Abstract Syntax Tree** from tokens to represent program.

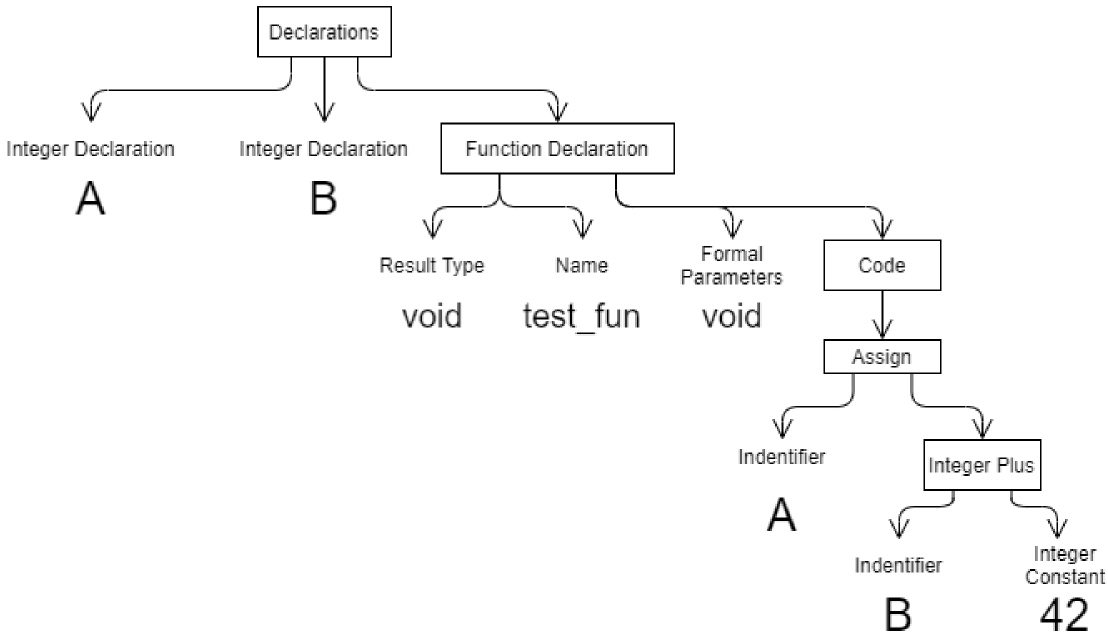
The grammar of the language can be expressed in a notation such as **Backus-Naur Form**. This is much the same as the definitions used for the simple while language in **Models of Computation**.

A rudimentary example could be:

$$
\begin{array}{rcl}
Cond & ::= & 'true' \mid 'false' \mid Expr == Expr \mid Expr < Expr \mid Cond \& Cond \mid \neg Cond \ldots \\
Expr & ::= & Var \mid Const \mid Expr + Expr \mid Expr \times Expr \ldots \\
Stat & ::= & x := Expr \mid 'if'\ Cond\ 'then'\ Stat\ 'else'\ Stat \mid Stat;\ Stat \mid 'skip' \mid 'while'\ Cond\ 'do'\ Stat
\end{array}
$$

The **Abstract Syntax Tree** is implemented as a tree of linked objects. It must be designed to be efficient to construct & contain all require information for subsequent stages.

The **Abstract Syntax Tree** for the basic C program would be something like:



Domain Specific Languages

Many programming languages are **domain specific**, meaning they are designed for a specific use.

- **Latex** Creating documents
- **Markdown** Simple text files with image, font support (e.g readmes)
- **YAML & TOML** Configuration files
- **Verilog** Digital Circuit Design
- **R** Statistics
- **Simulink** Dynamic System Modelling
- **Yacc** (Yet another compiler-compiler) parser generator language

There are also many language processors that are not compilers:

- **FindBugs** Finds bugs in java code
- **Pylint** Linter for python
- **Valgrind** Detecting bugs & memory leaks