COMP124

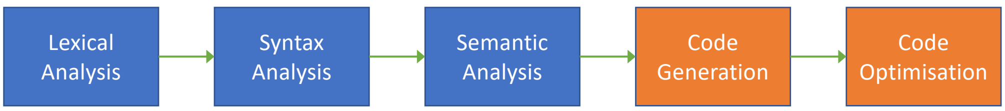
Computer Systems

# 25 | Compilation | Parsing | Code Generation

Dr Stuart Thomason

## The Compilation Process

- Compilation involves five main steps
  - The entire code is analysed at each step
  - Compiler can look at every line of code with surrounding context
  - Efficient and optimised machine code (object code) can be generated




- Interpreters use the first three steps to find the next instruction
  - Cannot do any optimisation because they do not 'look ahead' at the whole code
  - Might encounter an error in the code while the program is running

- The Java compiler uses all five steps to produce optimised bytecode that is then interpreted by the Java virtual machine

## Lexical Analysis

- Lexical analysis is also known as scanning
- The source code is just a text file containing a long string of characters
  - The scanner uses whitespace and punctuation to identify sequences of characters
  - Often implemented as a finite state machine
  - Some scanners use backtracking to look at surrounding context
  - This turns the code into a series of lexemes (or tokens)
- After this step, the compiler will have a list of the tokens that make up the program
  - Identifiers (variable, parameter and method names chosen by the programmer)
  - Keywords (that have some pre-defined meaning in the language)
  - Separators (such as brackets and punctuation)
  - Operators (mathematical symbols that can be evaluated)
  - Literals (such as strings, constants, numbers)
  - Comments (stripped out at this stage because they are not needed any more)

## Lexical Analysis Example

- Imagine a simple line of source code

```
total = (a + b) * vat;
```

- The scanner will split this into a list of tokens based on whitespace and punctuation

  IDENTIFIER total
  EQUALS
  OPEN_PAREN
  IDENTIFIER a
  PLUS
  IDENTIFIER b
  CLOSE_PAREN
  MULTIPLY
  IDENTIFIER vat
  SEMICOLON

- The source file will be turned into a long list of tokens that are passed to the next step

## Syntax Analysis

- Syntax analysis is also known as parsing
- The list of tokens is checked to see if it forms a valid program
  - The parser knows all the grammar rules for the language
  - It checks the sequence of tokens to see which rules they match
  - Often implemented as a recursive algorithm with backtracking
  - This generates an abstract syntax tree (AST)
- Syntax is correct if the entire list of tokens can be matched to a sequence of valid rules
- If all the tokens cannot be matched to a set of rules, there must be a syntax error, so the compiler stops and outputs a message

- The AST arranges tokens into a structure that represents different parts of the code (variable declarations, conditions, loops, functions, parameters, etc.)
- This abstract syntax tree structure is passed to the next step

## Abstract Syntax Tree Example

- We can define a simple modulo function and run the compiler to see its syntax tree

```
int modulo(int x, int y) {
    int result = (x % y);
    return result;
}
```

```
Stuart@Noctis comp124 % clang -Xclang -ast-dump -fsyntax-only modulo.c
`-FunctionDecl 0x7fe0e4886560 <modulo.c:1:1, line:4:1> line:1:5 modulo 'int (int, int)'
  |-ParmVarDecl 0x7fe0e4886400 <col:12, col:16> col:16 used x 'int'
  |-ParmVarDecl 0x7fe0e4886480 <col:19, col:23> col:23 used y 'int'
  `-CompoundStmt 0x7fe0e48867e8 <col:25, line:4:1>
    |-DeclStmt 0x7fe0e4886788 <line:2:3, col:23>
    | `-VarDecl 0x7fe0e4886670 <col:3, col:22> col:7 used result 'int' cinit
    |   `-ParenExpr 0x7fe0e4886768 <col:16, col:22> 'int'
    |     `-BinaryOperator 0x7fe0e4886748 <col:17, col:21> 'int' '%'
    |       |-ImplicitCastExpr 0x7fe0e4886718 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x7fe0e48866d8 <col:17> 'int' lvalue ParmVar 0x7fe0e4886400 'x' 'int'
    |       `-ImplicitCastExpr 0x7fe0e4886730 <col:21> 'int' <LValueToRValue>
    |         `-DeclRefExpr 0x7fe0e48866f8 <col:21> 'int' lvalue ParmVar 0x7fe0e4886480 'y' 'int'
    `-ReturnStmt 0x7fe0e48867d8 <line:3:3, col:10>
      `-ImplicitCastExpr 0x7fe0e48867c0 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x7fe0e48867a0 <col:10> 'int' lvalue Var 0x7fe0e4886670 'result' 'int'
```

## Semantic Analysis

- Semantic analysis adds meaning to the program
  - Identifiers are added to a symbol table
  - Performs type checking (in strongly typed languages)
  - Variable usage and function calls are matched to their definitions (object binding)
  - Memory requirements will be determined for each variable, parameter, etc.
  - Nodes of the syntax tree will be augmented with this semantic information
- The compiler will stop and output a message if semantic checks fail
  - Type mismatch (variables and parameters)
  - Using undeclared variables
  - Using a reserved keyword as an identifier
  - Multiple declarations of the same identifier
  - Using a variable that is out of scope
- The augmented syntax tree and symbol table are passed to the next step

## Code Generation

- Code generation turns the abstract syntax tree into actual machine code instructions
  - Adds instructions to reserve memory for each variable in the symbol table
  - Walks through each node in the syntax tree and turns it into CPU instructions
  - Walking node by node will generate code in the correct order
- The code generator must…
  - Specify the correct instructions (from the CPU instruction set)
  - Select appropriate CPU registers to hold variables during calculations
  - Create a subroutine (with label) for each high level function in the syntax tree
  - Add jump instructions to implement each condition and loop in the syntax tree
  - Add call, push and pop instructions to set up subroutines with correct parameters
  - Add debugging information (optionally) so the programmer can step through code
- The generated assembly code will be passed to the next step
- Most compilers generate machine code directly without going via assembly code first

## Assembly Code Generation Example

- We can view the generated assembly code for the modulo function (subroutine)

```
Stuart@Noctis comp124 % clang -Xclang -S -fsyntax-only -masm=intel modulo.c
        .globl  _modulo                         ## -- Begin function modulo
        .p2align        4, 0x90
_modulo:                                ## @modulo
        .cfi_startproc
## %bb.0:
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        mov     dword ptr [rbp - 8], esi
        mov     eax, dword ptr [rbp - 4]
        cdq
        idiv    dword ptr [rbp - 8]
        mov     dword ptr [rbp - 12], edx
        mov     eax, dword ptr [rbp - 12]
        pop     rbp
        ret
        .cfi_endproc
                                        ## -- End function
```

## Code Optimisation

- During the optimisation step the code is analysed to see if there are ways to…
  - Reduce the amount of code
  - Eliminate repeated operations
  - Reorganise parts of the program to execute faster

- There are lots of optimisation techniques
  - Most are beyond the scope of this module
  - See https://en.wikipedia.org/wiki/Optimizing_compiler

- Optimisation can happen in multiple places during the compilation process
  - Scanning
  - Parsing
  - Code generation

## Code Optimisation Techniques

- Remove redundancy – Store calculated results so they can be used again later instead of performing the calculation again
- Remove code – Such as unnecessary calculations and intermediate variables
- Unroll loops – Generate the same instructions in a linear manner instead of jumping back in the code (in some cases this helps with CPU instruction pre-fetching)
- Reverse loops – So the counter works in the opposite direction (we saw in an earlier lecture how this can reduce the number of instructions required)
- Increase locality – Put related code and data next to each other in memory to help with the principle of locality
- Use parallelism – Make use of multiple CPU cores by identifying and arranging instructions that can be carried out concurrently
- Fold constants – Replace calculations with their results instead of performing the calculation in the code (eg. replace 6 + 5 with 11)

## Loop Invariant Code

- Consider this fragment of code

```
for(int i = 0; i <= n; i++) {
     foo = amp + 5;
     sum = sum + (foo * i);
}
```

- The calculation of foo happens each time round the loop
  - But the value of amp is never changed inside the loop
  - So foo will always be the same on each iteration
  - Wastes CPU time doing the same calculation again and again
- The code optimisation step will look for loop invariant code and rearrange it

```
foo = amp + 5;
for(int i = 0; i <= n; i++) {
     sum = sum + (foo * i);
}
```

## Strength Reduction

- Consider this fragment of code

```
for(int i = 0; i <= n; i++) {
    sum = sum + i;
}
```

- The loop can be removed completely and replaced with a simple calculation

```
sum = n * (1 + n) / 2;
```

- This does exactly the same thing (sums all integers from 1 to n)
  - Uses fewer instructions and variables
  - Eliminates the need to jump back in the machine code (to implement the loop)
- Strength reduction replaces less efficient code with more efficient code where possible

- Code optimisation can make the code harder to understand (if you look at the generated assembly code or disassemble a compiled program)

## Linking Phase and Symbol Resolution

- A large program will be split across multiple source files
  - Each file is compiled separately into its own object file
  - Source files often refer to each other or to libraries (via #include, import, etc.)
  - Symbols (variables, functions) defined in one file can be used in another

- The linker is responsible for combining individual object files into the final executable
  - It can see the symbol table in each object file (and in each imported library)
  - It will look for a symbol called main (ie. the starting point of the program)
  - It will cross-check symbol tables to make sure every reference is valid

- The linker will stop and output an error message if…
  - The code uses a symbol that isn't defined in any of the source files or libraries
  - There is no symbol called main

## We’ve Come Full Circle

- We started the module by looking at how the CPU uses the fetch-execute cycle to bring instructions from memory into its registers

- Then we looked at the Intel instruction set and explored how to write programs using low level assembly code

- Finally we looked at how a compiler takes high level source code and turns it into the CPU instructions we covered at the start of the module