

CA448 Compiler Construction 1

Lecturer: Geoff Hamilton

Office: L255

Phone: 5017

Email: hamilton@computing.dcu.ie

WWW: <http://www.computing.dcu.ie/~hamilton>

Course web page: ["/teaching/CA448](http://www.computing.dcu.ie/~hamilton/teaching/CA448)

Essential Text: *"Compilers: Principles, Techniques and Tools"*, Aho, Lam, Sethi and Ullman, Pearson, ISBN 978-0-321-49169-5

Other Recommended Reading: *"Modern Compiler Implementation in Java"*, Second Edition, Andrew W. Appel, Cambridge University Press, ISBN 0-521-82060-X
"Generating Parsers With JavaCC", Tom Copeland, Centennial Books, ISBN 0-9762214-3-8

Important Qualities of a Compiler

You have used several compilers

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your feelings about the correct contents of this course

Compilers

What is a compiler?

- a program that translates an *executable* program in one language into an *executable* program in another language
- we expect the program produced by the compiler to be better, in some way, than the original

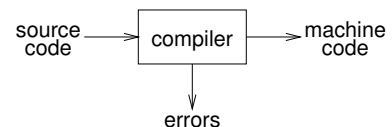
What is an interpreter?

- a program that reads an *executable* program and produces the results of running that program
- usually, this involves executing the source program in some fashion

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters*

Abstract view



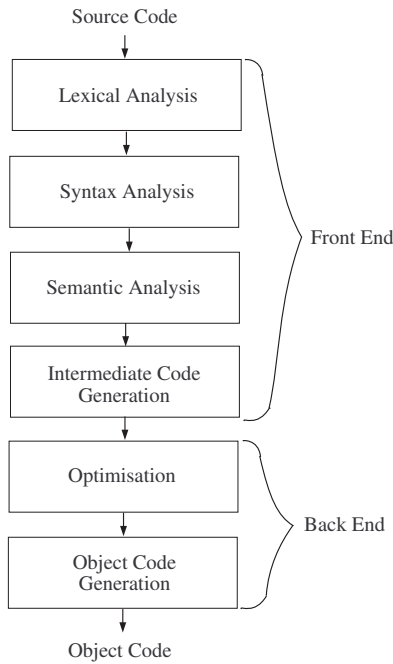
Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

Big step up from assembler — higher level notations

Compilation Phases

The compilation process is usually partitioned into a number of *phases*:



Compilation Phases

Lexical Analysis

Input: stream of characters from source program

Output: stream of *tokens* (groups of logically related characters e.g. identifiers, reserved words, punctuation, numbers, etc.).

In the case of tokens such as identifiers and numbers an additional quantity (the *lexeme*) is required to indicate which identifier or number is represented by this instance of the token.

For example:

Input: $x + 2 - y$

Output: $id(x), +, num(2), -, id(y)$

Compilation Phases

Syntax Analysis (Parsing)

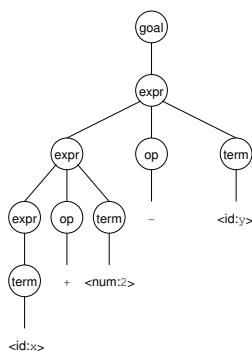
Input: stream of tokens from lexical analysis.

Output: parse tree (groups tokens to show the structure of the given sentence).

For example:

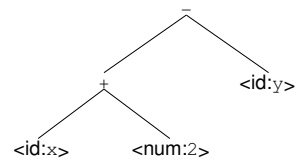
Input: $id(x), +, num(2), -, id(y)$

Output:



Compilation Phases

Parse trees often contain a lot of unnecessary information, so compilers often use an abstract syntax tree:



This is much more concise. Abstract syntax trees (ASTs) are often used as an IR between front end and back end.

Semantic Analysis

This phase checks the source program for semantic errors, and gathers type information for the intermediate code generation phase.

Compilation Phases

Intermediate Code Generation

Intermediate code is a kind of abstract machine code. which does not rely on a particular target machine by specifying the registers or memory locations to be used for each operation.

This separates compilation into a mostly language dependent *front end*, and a mostly machine-dependent *back end*.

For example:

```
loop: JLE x 0 end
      SUB x 1 temp
      MOV temp x
      JMP loop
end:  ...
```

Grouping of Phases

Several phases of compilation can be implemented in a single *pass* of the compiler, with the activity of the phases interleaved during the pass. A one-pass compiler only looks through the program once. A terrible methodology:

- ignores natural modularity
- gives unnatural scope rules
- limits optimisations

Used to be popular:

- fast (if your machine is slow)
- space efficient (if memory is very limited)

A modern compiler uses 5-15 passes.

Compilation Phases

Code Optimisation

This is an optional phase which can be used to improve the intermediate code to make it run faster and/or use less memory.

For example, the variable `temp` in the previous fragment of intermediate code is not required. This can be removed to give the following:

```
loop: JLE x 0 end
      SUB x 1 x
      JMP loop
end:  ...
```

Object Code Generation

This phase translates intermediate code into object code, allocating memory allocations for data, and selecting registers.