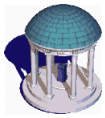


The University of North Carolina at Chapel Hill

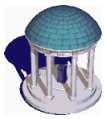
COMP 144 Programming Language Concepts

Spring 2002

Lecture 31: Building a Runnable Program

Felix Hernandez-Campos

April 10

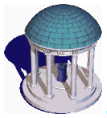


From Source Code to Executable Code

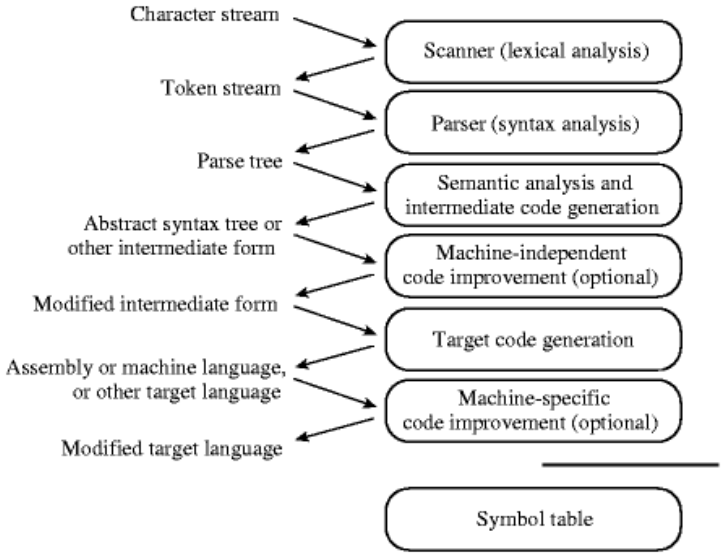
```
program gcd(input, output);
var i, j: integer;
begin
        read(i, j);
        while i <> j do
                if i > j then i := i - j;
                else j := j - i;
        writeln(i)
end.
```

Compilation

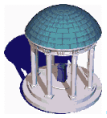
```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```



Phases of Compilation

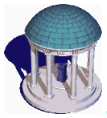


3

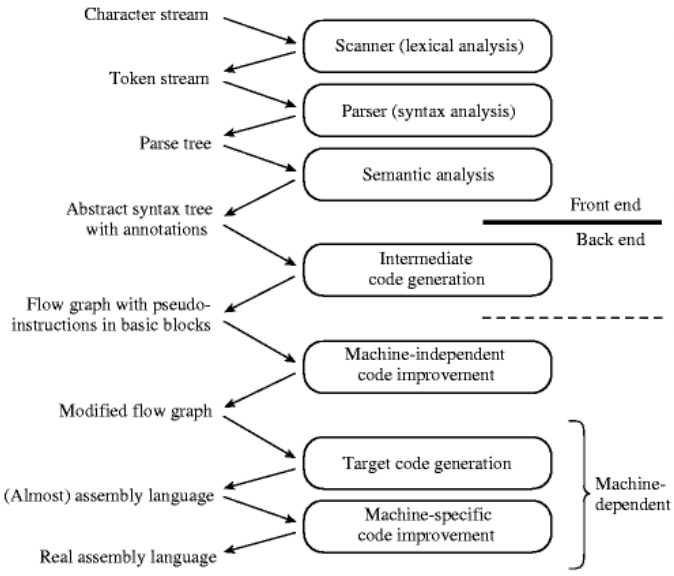


Compiler Structure

- Lexical, syntax and semantic analyses are the ***front-end*** of a compiler
  - These ***phases*** serve to figure out the meaning of the program
- The rest of the phases are considered part of the ***back-end*** of a compiler
  - They are responsible for the generation of the target program

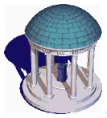


Phases of Compilation



- The first three phases are *language-dependent*
- The last two are *machine-dependent*
- The middle two dependent on neither the language nor the machine

5

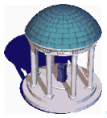


Example

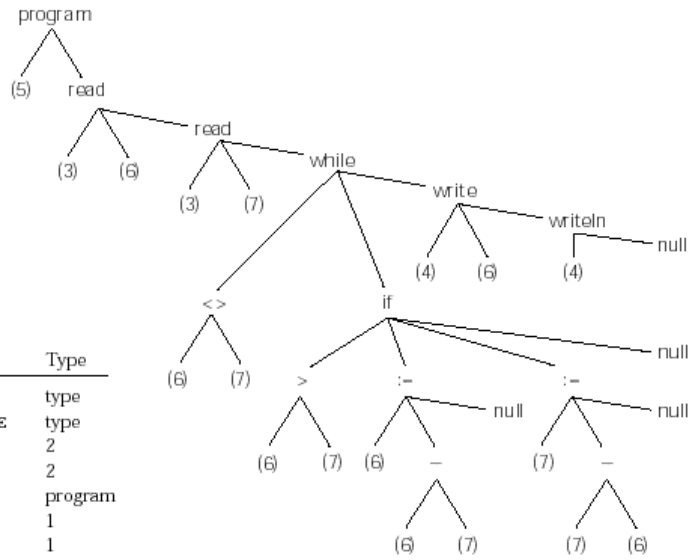
```
program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
      if i > j then i := i - j;
      else j := j - i;
  writeln(i)
end.
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

6

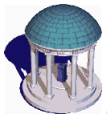


Example
Syntax Tree and Symbol Table

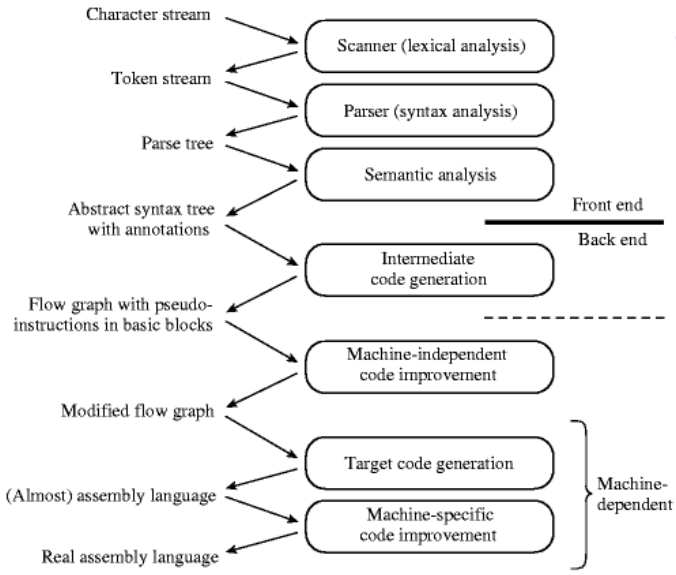


Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

7

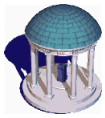


Phases of Compilation



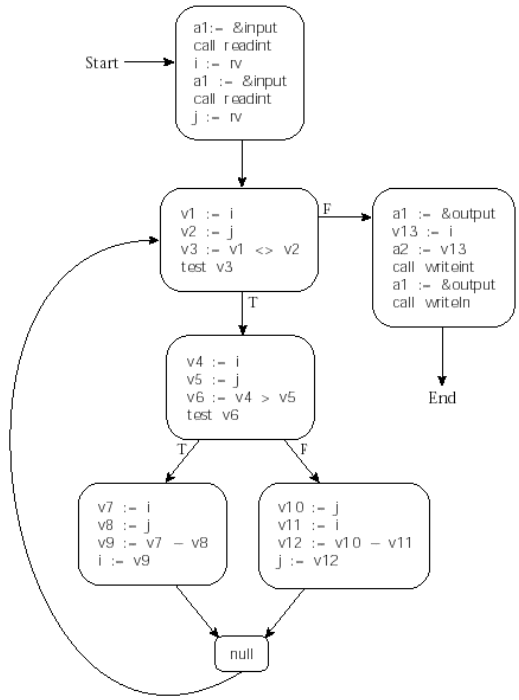
- ***Intermediate code generation*** transforms the abstract syntax tree into a less hierarchical representation: a control flow graph

8

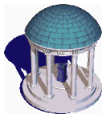


Example
Control Flow Graph

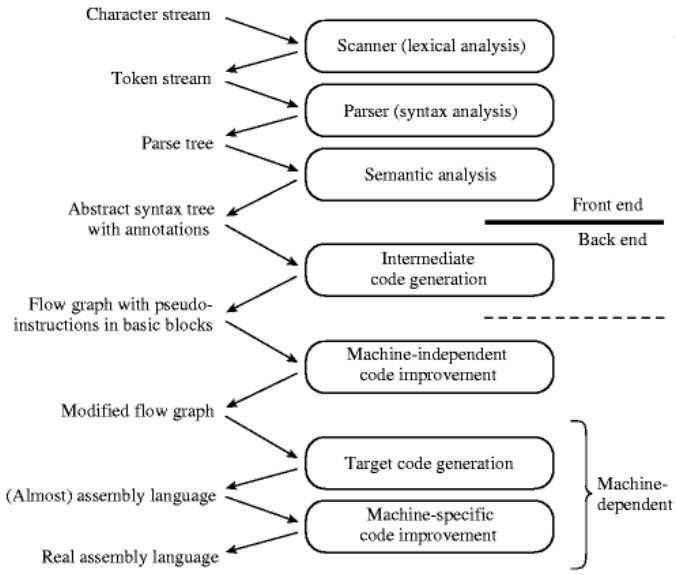
- ***Basic blocks*** are maximal-length set of sequential operations
  - Operations on a set of *virtual registers*
    - » Unlimited
    - » A new one for each computed value
- Arcs represent interblock control flow



COMP 144 I
Fe

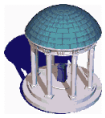


Phases of Compilation

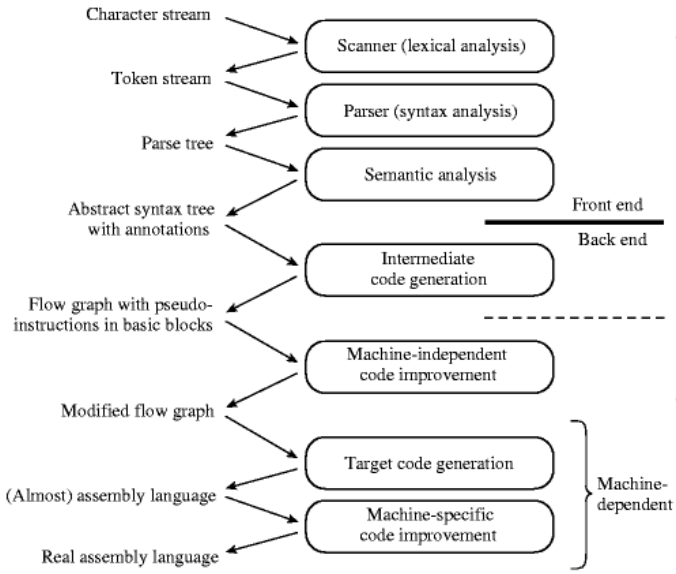


- ***Machine-independent code improvement*** performs a number of transformations:
  - Eliminate redundant loads stores and arithmetic computations
  - Eliminate redundancies across blocks

10

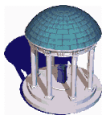


Phases of Compilation

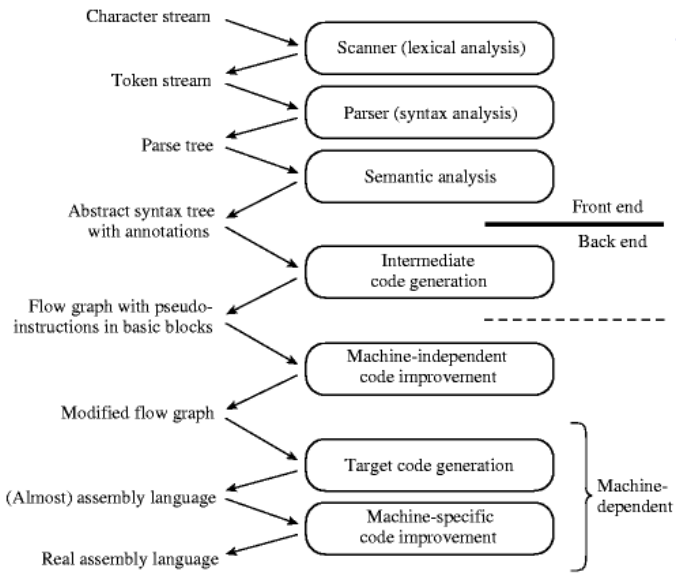


- ***Target Code Generation*** translates block into the instruction set of the target machine, including branches for the arc
- It still relies in the set of virtual registers

11

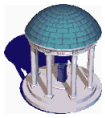


Phases of Compilation



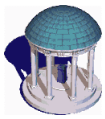
- ***Machine-specific code improvement*** consists of:
  - Register allocation (mapping of virtual register to physical registers and multiplexing)
  - Instruction scheduling (*fill the pipeline*)

12



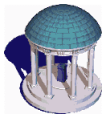
Compilation Passes

- A ***pass*** of compilation is a phase or sequence of phases that is serialized with respect to the rest of the compilation
  - It may be written as separate program that relies on files for input and output
- Two-pass compilers are very common
  - Front-end and back-end passes, or intermediate code generation and global code improvement
- Most compilers generate assembly, so the assembler behaves as an extra pass
- Assembly requires linking that may take place at compilation, load or run-time



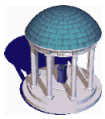
Compilation Passes

- Why are compilers divided in passes?
- Sharing the front-end among the compilers of more than one machine
- Sharing the back-end among the compilers of more than one language
- Historically, passes help reducing memory usage



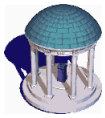
Intermediate Forms

- Front-end and back-end are linked using an abstract representation known as the Intermediate Format (IF)
  - The IF is propagated through the back-end phases
- They classified according to their level of machine dependence
- High-level IFs are usually trees or directed acyclic graphs that capture the hierarchy of the program
  - They are useful for machine-independent code improvement, interpretation and other operations



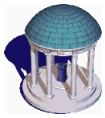
Intermediate Forms
Stack-based Language

- Stack-based language are another type of IFs
  - *E.g.* JVM, Pascal's P-code
- They are simple and compact
  - They resemble post-order tree enumeration
- Operations
  - Take their operands from an implicit stack
  - Return their result to an implicit stack
- These languages tend to make language easy to port and the result code is very compact
  - Ideal for network transfer of applets



Java Virtual Machine

- JVM spec
  - http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- Tutorial
  - http://www-106.ibm.com/developerworks/library/it-haggar_bytecode/index.html



Reading Assignment

- Read Scott
  - Sect. 9 Intro
  - Sect. 9.1
  - Sect. 9.2 intro, and glance at IDL and RTL