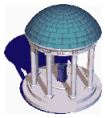


The University of North Carolina at Chapel Hill

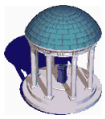
COMP 144 Programming Language Concepts

Spring 2002

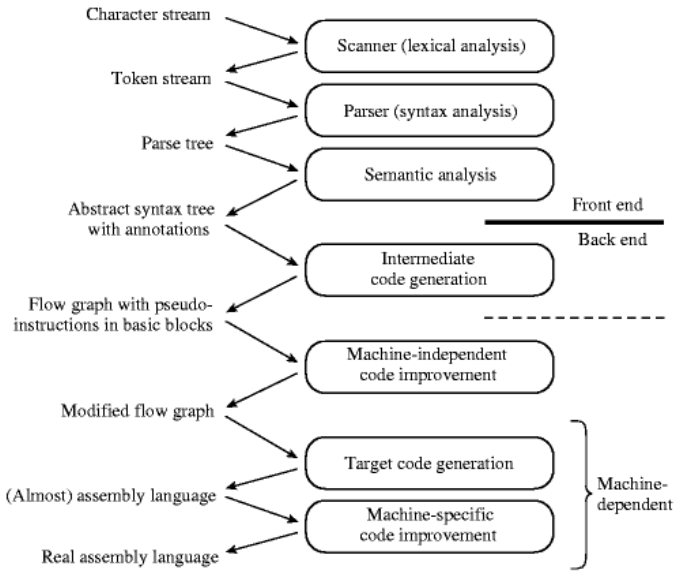
Lecture 33: Code Generation and Linking

Felix Hernandez-Campos

April 15

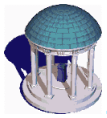


Phases of Compilation



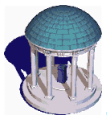
- The first three phases are *language-dependent*
- The last two are *machine-dependent*
- The middle two dependent on neither the language nor the machine

2



Example

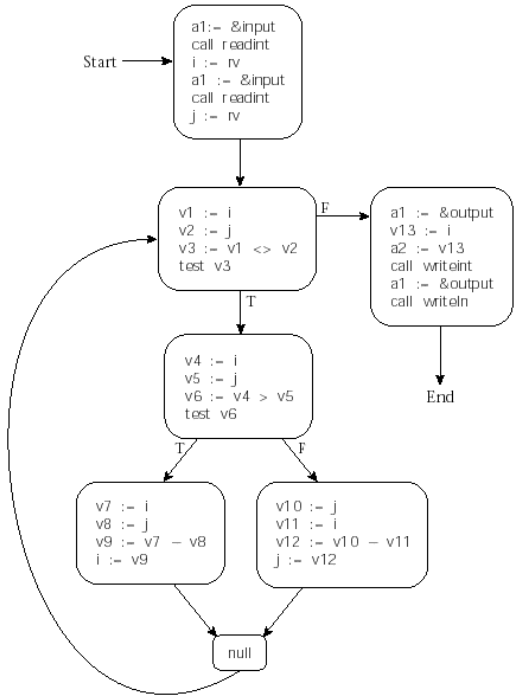
```
program gcd(input, output);
var i, j: integer;
begin
  read(i, j);
  while i <> j do
      if i > j then i := i - j;
      else j := j - i;
  writeln(i)
end.
```

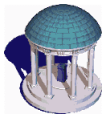



Example

Control Flow Graph

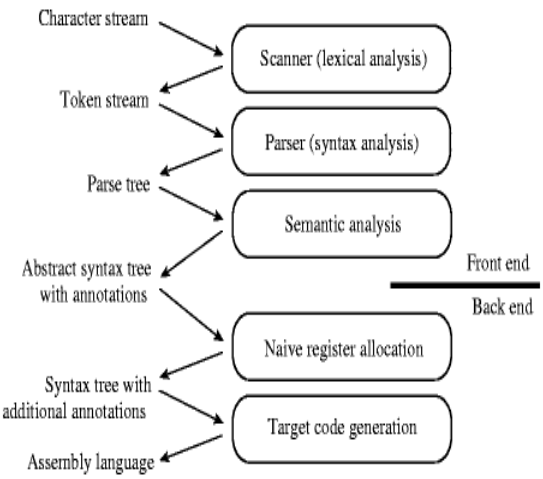
- ***Basic blocks*** are maximal-length set of sequential operations
  - Operations on a set of *virtual registers*
    - » Unlimited
    - » A new one for each computed value
- Arcs represent interblock control flow

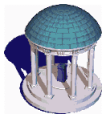




Code Generation

- We will illustrate the back-end with a simple compiler
- There is not control flow graph
  - No optimizations
- Two main tasks:
  - Register allocation
  - Instruction Scheduling



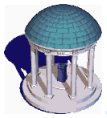


Code Generation Example

```
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.
```

Compilation

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

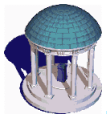


- Reserved registers: a1, a2, sp and rv
- General purpose: r1..r*k*
- Expression code generation: `(a+b) ◈ (c - (d/e)`

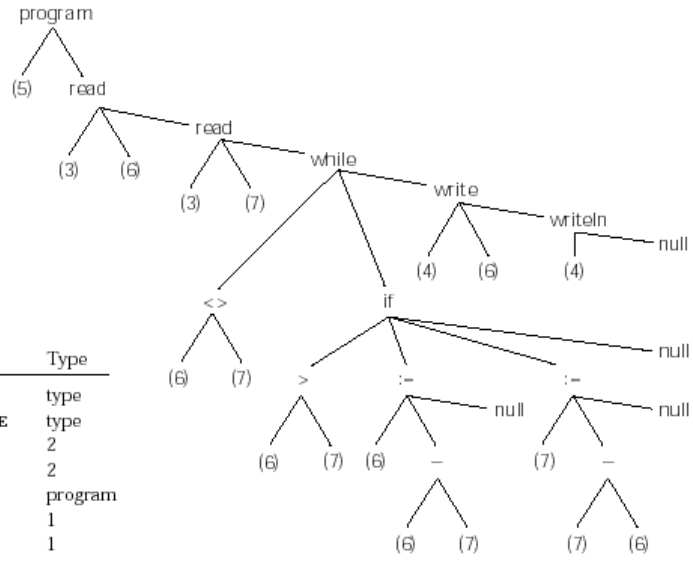
```
r1 := a          -- push a
r2 := b          -- push b
r1 := r1 + r2    -- add
r2 := c          -- push c
r3 := d          -- push d
r4 := e          -- push e
r3 := r3 / r4    -- divide
r2 := r2 - r3    -- subtract
r1 := r1 × r2    -- multiply
```

R*i* used as expression evaluation stack

7

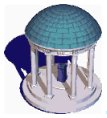


Code Generation Example
Syntax Tree and Symbol Table



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

8



Code Generation Example
Attribute Grammar

reg_names : array [0..$k-1$] of register_name := ["r1", "r2", ..., "rk"]
-- ordered set of temporaries

program $\longrightarrow$ *id stmt*
- ▷ stmt.next_free_reg := 0
- ▷ program.code := ["main:"] + stmt.code + ["goto exit"]
- ▷ program.name := id.stp→name

Code generation

while : stmt$_1$ $\longrightarrow$ *expr stmt*$_2$ *stmt*$_3$

stmt nodes

- ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_3$.next_free_reg := stmt$_1$.next_free_reg
- ▷ L1 := new_label (); L2 := new_label ()
  stmt$_1$.code := ["goto" L1] + [L2 ":"] + stmt$_2$.code + [L1 ":"] + expr.code
  + ["if" expr.reg "goto" L2] + stmt$_3$.code

if : stmt$_1$ $\longrightarrow$ *expr stmt*$_2$ *stmt*$_3$ *stmt*$_4$
- ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_3$.next_free_reg := stmt$_4$.next_free_reg := stmt$_1$.next_free_reg
- ▷ L1 := new_label (); L2 := new_label ()
  stmt$_1$.code := expr.code + ["if" expr.reg "goto" L1] + stmt$_3$.code + ["goto" L2]
  + [L1 ":"] + stmt$_2$.code + [L2 ":"] + stmt$_4$.code

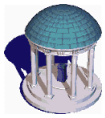
Code Generation Example

assign : stmt$_1$ $\longrightarrow$ *id expr stmt*$_2$
- ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_1$.next_free_reg
- ▷ stmt$_1$.code := expr.code + [id.stp→name ":=" expr.reg] + stmt$_2$.code

read : stmt$_1$ $\longrightarrow$ *id*$_1$ *id*$_2$ *stmt*$_2$
- ▷ stmt$_1$.code := ["a1 := &" id$_1$.stp→name] -- file
  + ["call" if id$_2$.stp→type = int then "readint" else ...]
  + [id$_2$.stp→name ":= rv"] + stmt$_2$.code

write : stmt$_1$ $\longrightarrow$ *id expr stmt*$_2$
- ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_1$.next_free_reg
- ▷ stmt$_1$.code := ["a1 := &" id.stp→name] -- file
  + ["a2 :=" expr.reg] -- value
  + ["call" if id.stp→type = int then "writeint" else ...] + stmt$_2$.code

writeln : stmt$_1$ $\longrightarrow$ *id stmt*$_2$
- ▷ stmt$_1$.code := ["a1 := &" id.stp→name] + ["call writeln"] + stmt$_2$.code



Code Generation Example

null : *stmt* $\longrightarrow \epsilon$
▷ stmt.code := nil

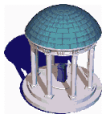
'<>' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ handle_op (expr$_1$, expr$_2$, expr$_3$, "<>")

'>' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ handle_op (expr$_1$, expr$_2$, expr$_3$, ">")

'−' : $expr_1 \longrightarrow expr_2\ expr_3$
▷ handle_op (expr$_1$, expr$_2$, expr$_3$, "−")

id : *expr* $\longrightarrow \epsilon$
▷ expr.reg := reg_names[expr.next_free_reg mod k]
▷ expr.code := [expr.reg ":=" expr.stp→name]

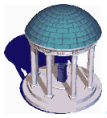
Allocation of next register



Code Generation Example

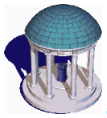
```
macro handle_op (ref result, L_operand, R_operand, op : syntax_tree_node)
    result.reg := L_operand.reg
    L_operand.next_free_reg := result.next_free_reg
    R_operand.next_free_reg := result.next_free_reg + 1
    if R_operand.next_free_reg < k
        spill_code := restore_code := nil
    else
        spill_code := ["*sp :=" reg_name[R_operand.next_free_reg mod k]]
                    + ["sp := sp − 4"]
        restore_code := ["sp := sp + 4"]
                    + [reg_names[R_operand.next_free_reg mod k] ":= *sp"]
    result.code := L_operand.code + spill_code + R_operand.code
                 + [result.reg ":=" L_operand.reg op R_operand.reg] + restore_code
```

Register Spill



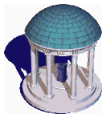
Code Generation Example

```
    -- first few lines generated during symbol table traversal
    .data          -- begin static data
    .word i        -- reserve one word to hold i
    .word j        -- reserve one word to hold j
    .text          -- begin text (code)
    -- remaining lines accumulated into program.code
main:
    a1 := &input -- "input" and "output" are file control blocks
                 -- located in a library, to be found by the linker
    call readint -- "readint", "writeint", and "writeln" are library subroutines
    i := rv
    a1 := &input
    call readint
    j := rv
    goto L1
```

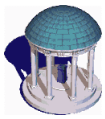
Code Generation Example

```
L2: r1 := i          -- body of while loop
    r2 := j
    r1 := r1 > r2
    if r1 goto L3
    r1 := j          -- "else" part
    r2 := i
    r1 := r1 - r2
    j := r1
    goto L4
L3: r1 := i          -- "then" part
    r2 := j
    r1 := r1 - r2
    i := r1
L4:
```



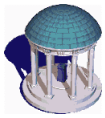
Code Generation Example

```
L1:  r1 := i          -- test terminating condition
     r2 := j
     r1 := r1 <> r2
     if r1 goto L2
     a1 := &output
     r1 := i
     a2 := r1
     call writeint
     a1 := &output
     call writeln
     goto exit        -- return to operating system
```

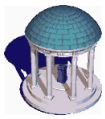
Address Space Organization

- Assemblers, linker and loader typically operate on a pair of related file formats:
- *Relocatable* object code
  - Input to linker
  - Multiple file are combined to create an executable program
  - It includes the following information:
    - » Import table (named locations with unknown addresses)
    - » Relocation table (instruction that refer to current file)
    - » Export table
  - Imported and exported names are known as *external symbols*



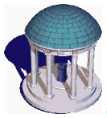
Address Space Organization

- *Executable* object code
  - Input to loader
  - Loaded in memory and run
- A running program is divided into segments
  - Code
  - Constants
  - Initialized data
  - Uninitialized data
  - Stack
  - Heap
  - Files (mapped to memory)



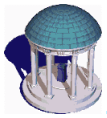
Linking

- Large program must be divided into separate compilation units
  - *E.g.* main program and libraries
- The *linker* is in charge of joining together those compilation units
  - Each compilation unit must a relocatable object file
- Linking involves two subtasks
  - Relocation
  - Resolution of external references

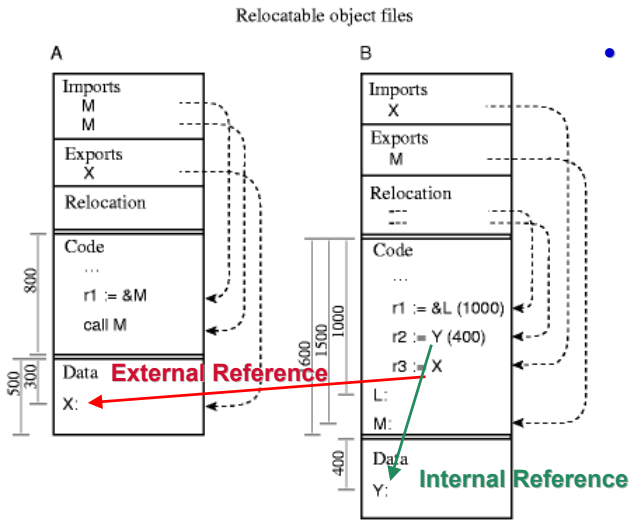


Relocation

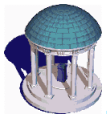
- Two phases
- In the first phase:
  - Gather all the of the compilation units
  - Choose an order in memory and note addresses
- In the second phase:
  - Resolve external references on each unit
  - Modify instruction set as required
- Linking also involved type checking using module headers



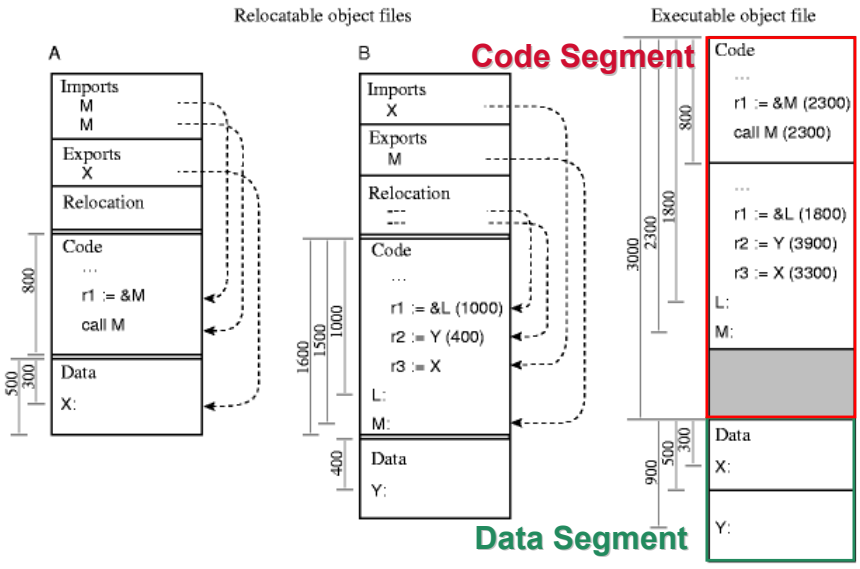
Linking Example

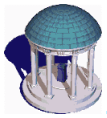


- Two relocatable object files are linked in this example, A and B
  - M and X are *external references*
  - L and Y are *internal refereces*



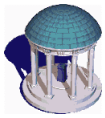
Linking Example





Dynamic Linking

- In many modern systems, many programs share the same libraries
  - It is a waste to create a copy of each library at run-time
- Most operating system support *dynamically linked libraries* (DLLs)
  - Each library resides in its own code and data segment
  - Program instances that uses the library has private copy of the data segment
  - Program instances share a *system-wide read-only copy* of the the library's code segment
- DLLs support backward-compatible without recompilation



Reading Assignment

- Read Scott
  - Sect. 9.3
  - Sect. 9.4
  - Sect. 9.6
  - Sect. 9.7 intro