

Programming Languages

Building an Executable Program

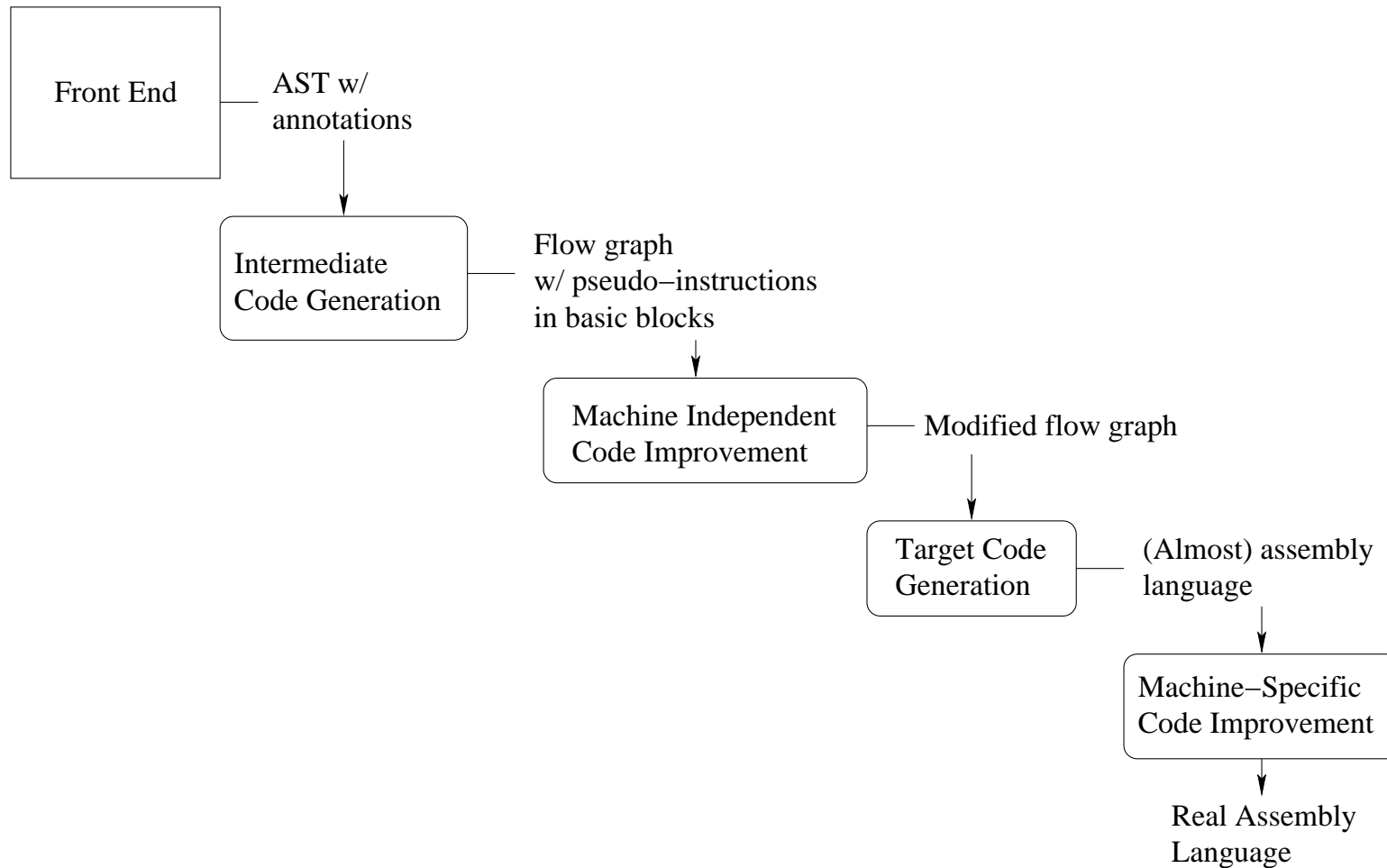
Benjamin J. Keller

Department of Computer Science, Virginia Tech

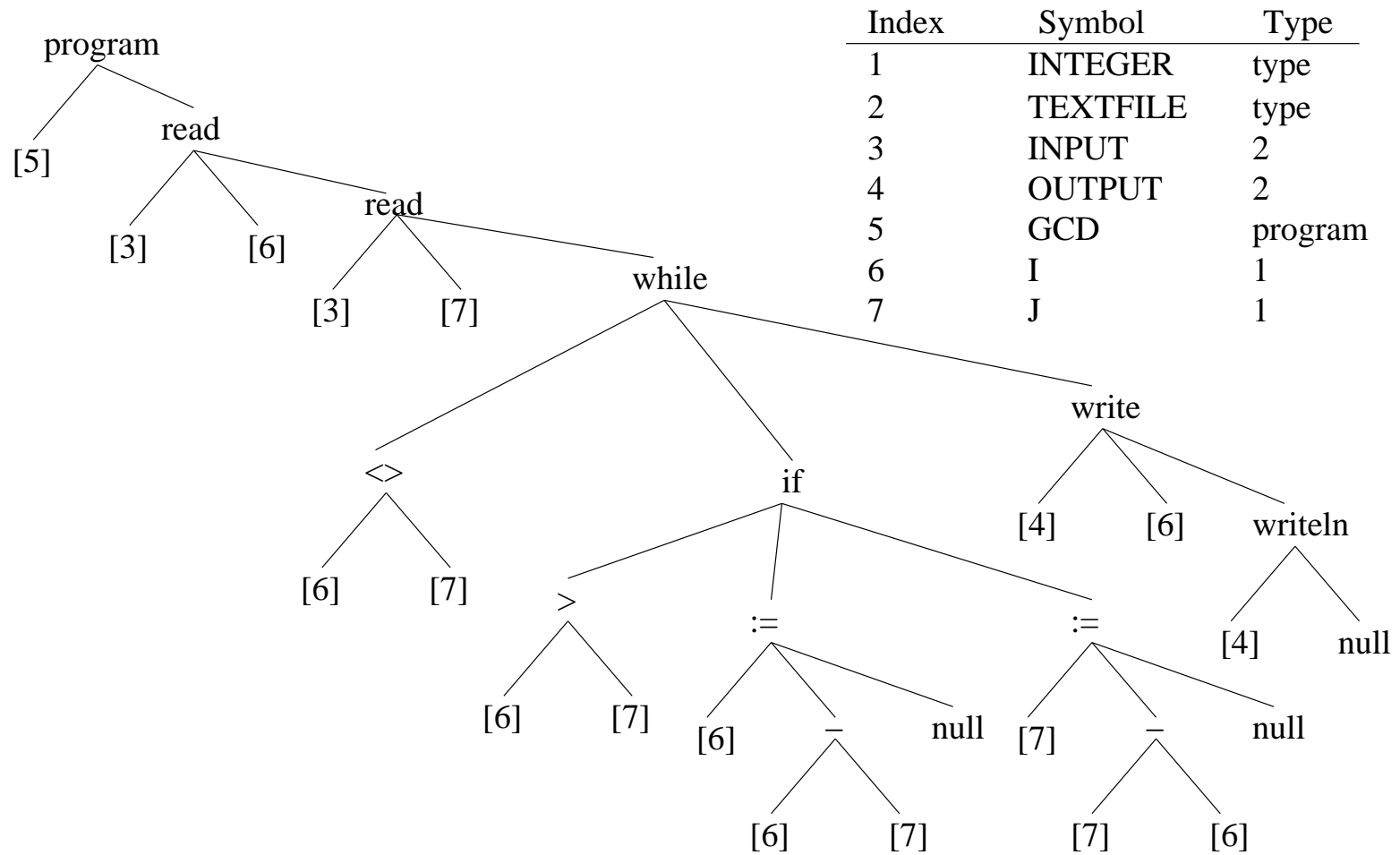
Overview

- General Back-End Structure
- Intermediate Forms
- Code generation with attribute grammar

Compiler Back-End Structure



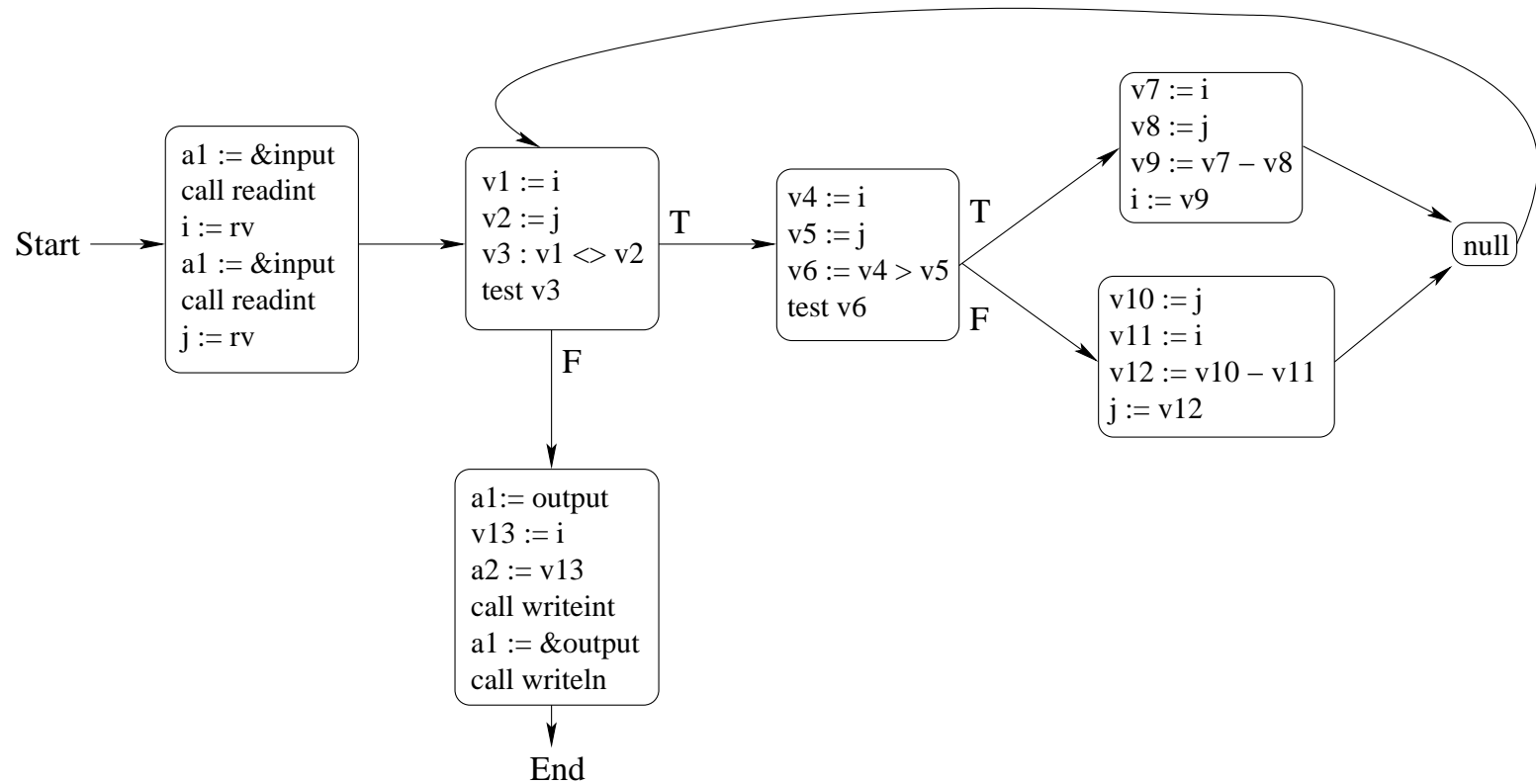
Syntax Tree



Intermediate Code Generation

- Code improvements easier on form other than AST
- *Basic block* — maximal length sequence of nonbranch instructions
- *Control Flow Graph*
 - Vertices: basic blocks
 - Arcs: control flow between graphs
- Instructions for an idealized processor
- Assume unlimited set of *virtual* registers

Control Flow Graph



Machine Independent Code Improvement

- Transformations on control flow graph
- Local improvements (to basic blocks):
 - Eliminate unnecessary loads and stores
 - Simplify and combine arithmetic expressions
- Global improvements:
 - Eliminating recomputations inside conditional blocks
 - Eliminating computation of invariant value inside loop

Final Phases

- Target code generation:
 - Translate control flow graph into assembly language for target machine
 - Strings together blocks, adding branches as needed
- Machine-specific code improvement:
 - Register allocation — assigning virtual registers to physical registers
 - Instruction scheduling — reordering instructions in basic block to try to keep pipeline(s) of machine full

Intermediate Forms

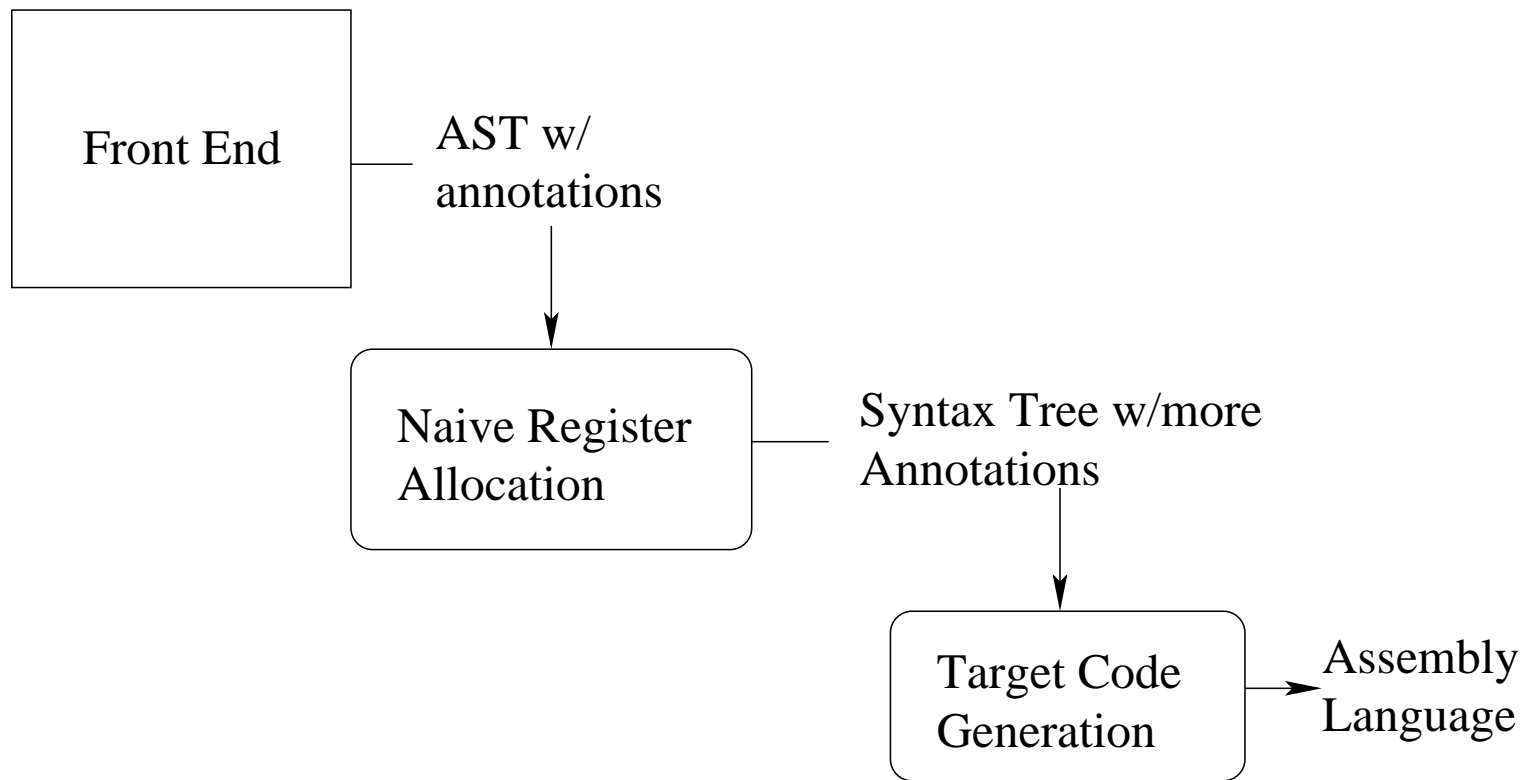
- Different compilers use different intermediate forms
- Vary in *level* or degree of abstraction from machine language
- High-level:
 - Tree-based forms
 - Stack-based: operands stored on stack
- Mid-level:
 - Control flow graph
 - Quadruples: opcode, two operands and destination

Example: GNU RTL

- Intermediate form for `gcc`
- Lisp like expressions: operator followed by list of operands
- Implemented using structs and pointers (text form not common)

```
(insn 8 6 10 (set (reg:SI 2)
                 (mem:SI (symbol_ref:SI ("a")))))
(insn 10 8 12 (set (reg:SI 3)
                 (mem:SI (symbol_ref:SI ("b")))))
(insn 12 10 14 (set (reg:SI 2)
                  (plus:SI (reg:SI 2)
                          (reg:SI 3))))
(insn 14 12 15 (set (reg:SI 3)
                  (mem:SI (symbol_ref:SI ("c")))))
(insn 15 14 17 (set (reg:SI 2)
                  (mult:SI (reg:SI 2)
                          (reg:SI 3))))
(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
                  (reg:SI 2)))
```

A Simple Compiler



Code Generation

program → *id stmt*

- ▷ `stmt.next_free_reg := 0`
- ▷ `program.code := ["main:"] +
stmt.code + ["goto exit"]`
- ▷ `id.stp->name`

Code Generation

$if : stmt_1 \rightarrow expr\ stmt_2\ stmt_3$

- ▷ `expr.next_free_reg := stmt2.next_free_reg`
`:= stmt3.next_free_reg`
`:= stmt4.next_free_reg`
`:= stmt1.next_free_reg`
- ▷ `L1 := new_label(); L2 := new_label()`
`stmt1.code := expr.code +`
`["if" expr.reg "goto" L1] +`
`stmt3.code + ["goto" L2]`
`+ [L1 ":"] + stmt2.code`
`+ [L2 ":"] + stmt4.code`

Code Generation

$id : expr \rightarrow \epsilon$

- ▷ `expr.reg := reg_names[expr.next_free_reg mod k]`
- ▷ `expr.code := [expr.reg " := " expr.stp->name]`

Register Allocation

- Maintain a pool of registers and cycle through them to select
- When return to parent, effectively pop registers off of “stack”
- If run out of registers, must *spill* contents to memory
- This is an extremely naive approach — would not be used

Example

Code for $(a+b)*(c-(d/e))$

```
r1 := a
```

```
r2 := b
```

```
r1 := r1 + r2
```

```
r2 := c
```

```
r3 := d
```

```
r4 := e
```

```
r3 := r3 / r4
```

```
r2 := r2 - r3
```

```
r1 := r1 * r2
```