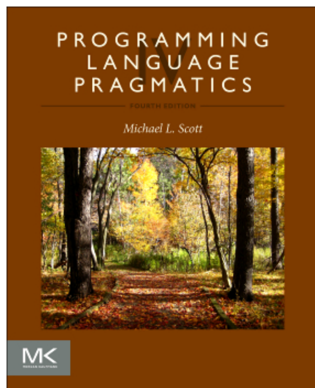
Intermediate Code Generation

17-363/17-663: Programming Language Pragmatics

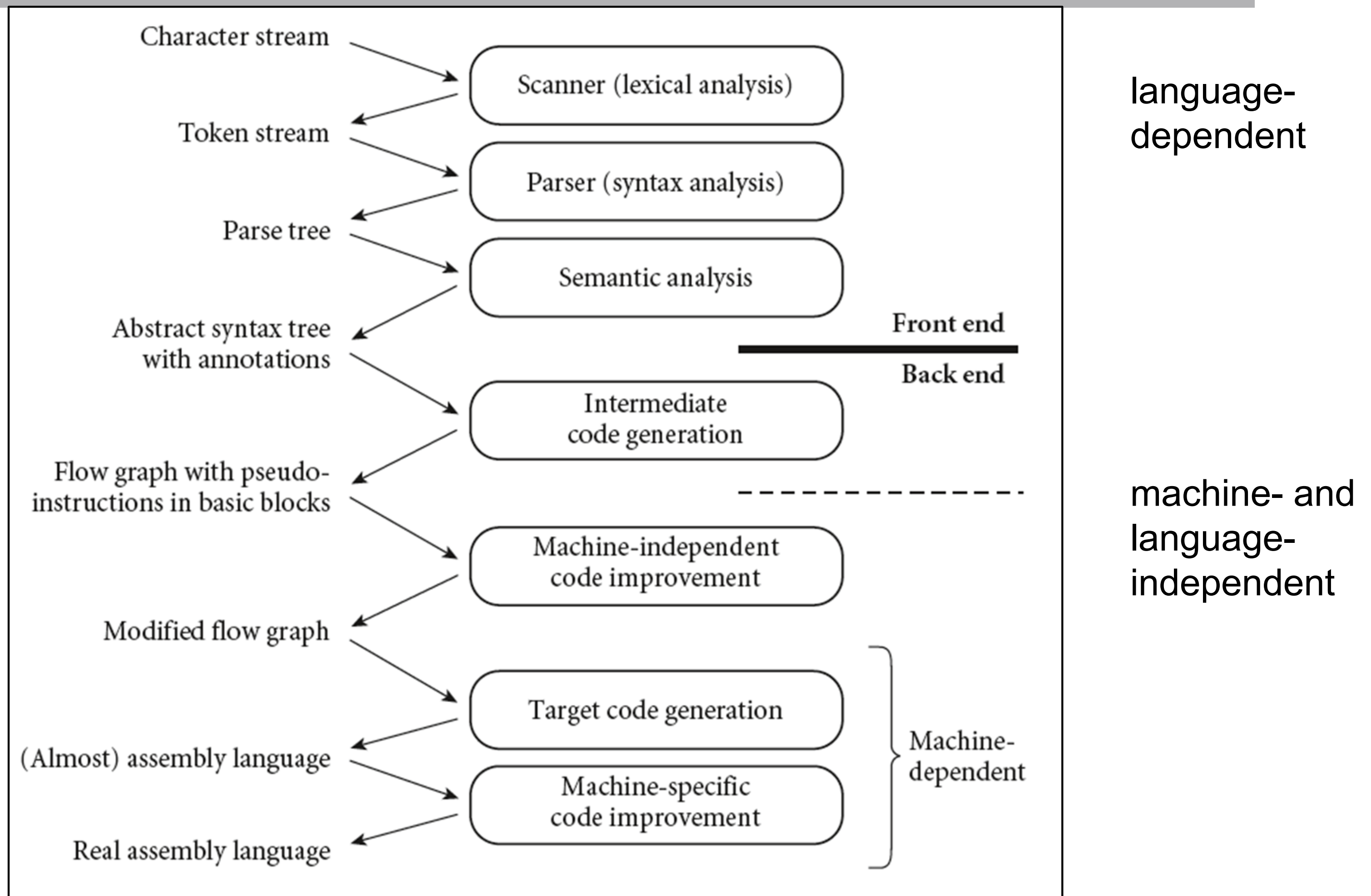


Reading: PLP chapter 15

Prof. Jonathan Aldrich

Copyright © 2016 Elsevier

Review: Compiler Structure



language-
dependent

machine- and
language-
independent

- Review: phases of compilation
  - Machine-independent phases form a “middle end” in addition to front end and back end

ELSEVIER

Intermediate Representations

- Many compilers have multiple intermediate representations
  - Different compilation steps work at different levels of abstraction
- High-level: Abstract Syntax Trees
- Mid-level: Control Flow Graphs
  - Nodes are **basic blocks**: instruction sequences with no jumps in or out
    - Idealized, machine-independent instructions are given in 3-address code: `r1 := r2` *`op`* `r3`
  - Edges are jumps
- Low Level: Instructions for an idealized machine
  - May be the same notation used inside basic blocks, above
  - Often with infinite "virtual registers" instead of finite physical ones
- Note: there are no hard boundaries between these levels

ELSEVIER

Stack-Based Bytecode

- Bytecode is an IR optimized for compactness, interpretability
  - Compactness is important for code sent over a network
    - The name comes from the typical "one instruction per byte"
  - Can build a fast interpreter by branching on the byte
    - Or more sophisticated techniques, like direct threaded code, jump tables, and computed gotos
    - In commercially important language, it usually gets compiled "just in time" for performance anyway
  - Still simple & portable, like other IRs
- Examples: Java bytecode, Pascal p-code, Microsoft CIL

ELSEVIER

Stack-Based vs. Pseudo-Assembly

- Heron's formula: `A = sqrt [s(s-a)(s-b)(s-c)] where s = (a+b+c)/2`

```
stack-based:          3-address pseudo-assembly:
push a                r2 := a
push b                r3 := b
push c                r4 := c
add                   r1 := r2 + r3
add                   r1 := r1 + r4
push 2                r1 := r1 / 2       -- s
divide
pop s
push s
push s                r2 := r1 - r2      -- s-a
push a
subtract
push s                r3 := r1 - r3      -- s-b
push b
subtract
push s                r4 := r1 - r4      -- s-c
push c
subtract
multiply              r3 := r3 * r4
multiply              r2 := r2 * r3
multiply              r1 := r1 * r2
push sqrt             call sqrt
call
```

Tradeoff: space vs. time

- Bytecode is more compact
  - 23 instructions in 25 bytes
  - Most instructions fit in a byte
    - including push small integers or first few variables
  - 2 extra bytes to specify `sqrt`
- 3-address easier to optimize
  - Reorder / replace instructions, considering registers or pipeline are all easier
  - But: most instructions 4 bytes
    - The call instruction is 8 bytes
  - 13 instructions in 56 bytes

ELSEVIER

WebAssembly

- Intermediate representation we'll use as a target in class
- Goals
  - Portable
  - Browser platform (interop with JavaScript)
  - Machine-independent
  - Memory-safe (all errors → exceptions, read/write only within process)
  - Compact (for sending over the network)
  - Fast (to interpret, to compile, and to execute compiled code)
  - Typed (but low-level: almost everything is an int32 or float)

- Two formats
  - Portable binary format (.wasm)
  - Textual equivalent based on S-expressions (.wat)
    - *S* ::= int_const | str_const | symbol | id | (*S**)

- Semantics specified with inference rules

Basic WebAssembly Bytecode

Instructions	Stack afterward
	(empty)
i32.const 1	1
i32.const 2	1 2
i32.add	3

ELSEVIER

Variables & Main in WebAssembly

TypeScript Source

```
let x:number = 1;
x = x + 1
```

WASM

```
(func (export "main") (local $x i32)
                                  (empty)
    i32.const 1                   i32
    local.set $x                  (empty)
    local.get $x                  i32
    i32.const 1                   i32 i32
    i32.add                       i32
    local.set $x                  (empty)
))
```

If in WebAssembly

`see if_false.ts / if_false.wat`

ELSEVIER

Typechecking WebAssembly

- Stacks must match at control flow merges!

```
                              (empty)
local.get $condition          i32
if                            (empty)
i32.const 1                   i32
else
                              (empty)
end                           ??? type error!
i32.const 2
i32.add                       // error if took else branch!
```

ELSEVIER

Typechecking WebAssembly

- Stacks must match at control flow merges! Fixed now.

```
                              (empty)
local.get $condition          i32
if                            (empty)
i32.const 1                   i32
else
i32.const 2                   i32
end                           i32
i32.const 2                   i32 i32
i32.add                       i32
```

ELSEVIER

Practice!

- Translate the following pseudocode to WebAssembly:

if x > 0 then x else –x

- Some useful instructions: local.get, i32.const, i32.gt_s, i32.sub, if/else/end,

ELSEVIER

Loops, Functions, and Nonlocal Returns

```
see while_count
see return
```

Imports, Memories, Running from JavaScript

`see hello and run.js`

- Compiling and running

```
wat2wasm wat/part1/hello.wat -o wasm/part1/hello.wasm

node run.js wasm/part1/hello.wasm 1
```

- argument `1` indicates 1 page of memory (64k)
  - see `run.js` code for how this is passed in
- Shortcut: `./single.sh part1/hello`

ELSEVIER

Global variables

```
(module
  (import "console" "log_int" (func $log_int (param i32)))
  (global $tmp (mut i32) (i32.const 0))
  (func (export "main")
    global.get $tmp
    i32.const 1
    i32.add
    global.set $tmp
))
```

ELSEVIER

Memories

```
(module
  (import "console" "log_int" (func $log_int (param i32)))
  (import "js" "mem" (memory 1))
  (func (export "main")
    i32.const 0
    i32.const 1
    i32.store
    i32.const 0
    i32.load
))
```

- See also `run.js`

ELSEVIER

Tables, Types, Indirect Calls

```
(module
  (import "console" "log_int" (func $log_int (param i32)))
  (import "js" "mem" (memory 1))
  (table 2 funcref)
  (elem (i32.const 0) $foo $bar)
  (type $fn1arg (func (param i32) (result i32)))
  (func (export "main")
    i32.const 10
    i32.const 0
    call_indirect (type $fn1arg)
    i32.const 1
    call_indirect (type $fn1arg)
  )
  (func $foo (param $x i32) (result i32) ...)
  (func $bar (param $x i32) (result i32) ...)
)
```

Generating code

- Generally a tree traversal
  - Producing instructions (or S-expressions, for WebAssembly)
- May need some information from typechecker
  - Or just (re-)compute, if it's simple
- May need to collect information to return along with code
  - E.g. in Assignment 7, this includes a list of all local variables declared (must be declared at the top of a function in WebAssembly)
  - If not already computed during typechecking

ELSEVIER

Other handy instructions

```
drop
```