CS251: Introduction to Language Processing

Code Generation and Optimizations

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

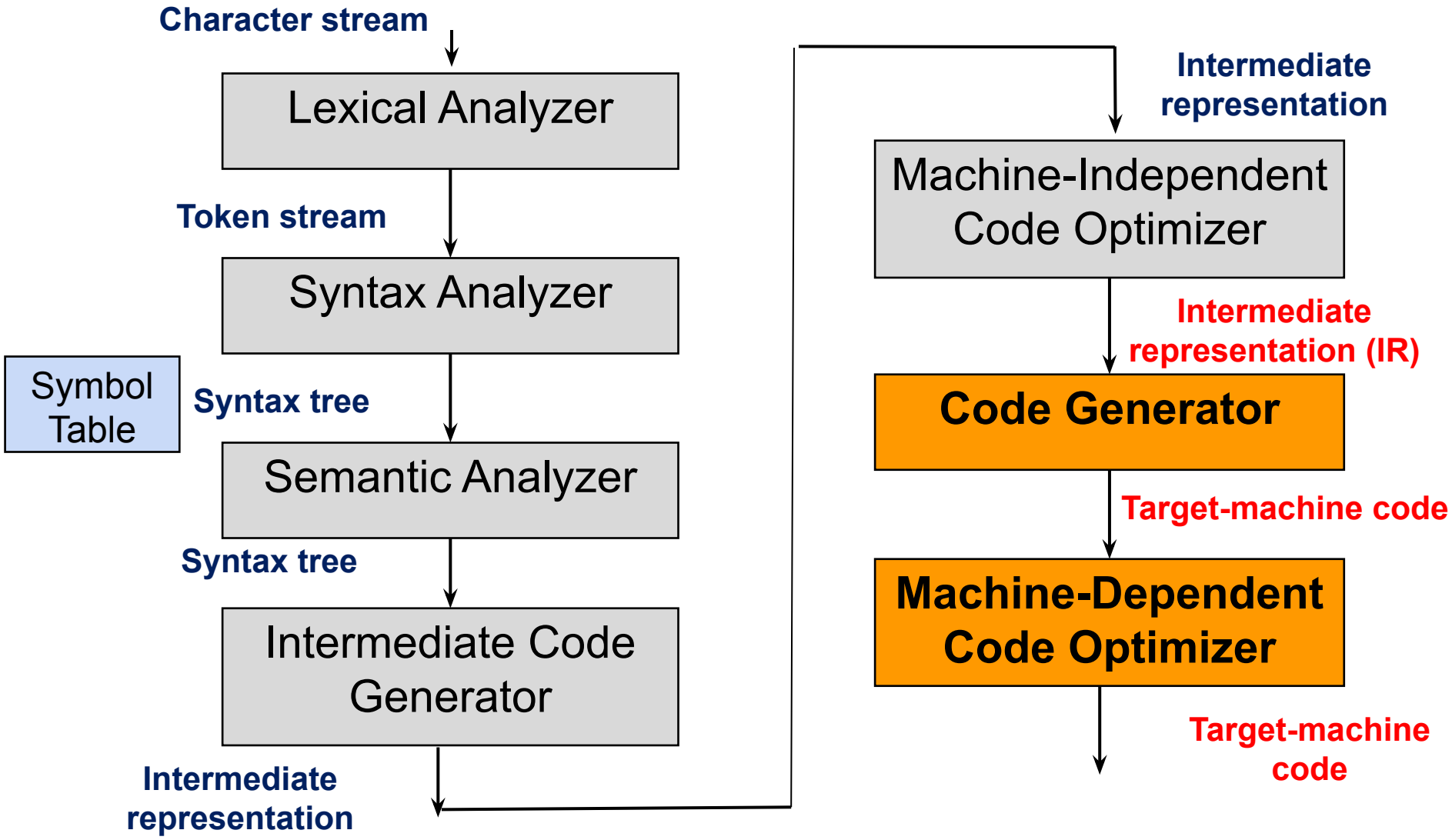
vishwesh@iitbhilai.ac.in

2023-24 M

Acknowledgement

- References for today's slides
  - *Prof. Y. N Srikant, IISc Bangalore*
    - *https://nptel.ac.in/content/storage2/courses/106108052/module4/code-gen-part-3.pdf*
  - *Course textbook*

Compiler Design



Target Language

- Assume a simple target assembly code
- Instructions:
  - Load: *LD dst, addr*
  - Store: *ST x, reg*
  - Computation operations: *OP dst, src1, src2*
    - OP can be ADD, SUB, MUL, etc.
  - Unconditional jumps: *BR L*
  - Conditional jumps: *BCond reg, L*

A Simple Code Generator: Overview

- Generates the code for each basic block
- Consider each three-address-code in turn
- For each instruction
  - Decides registers
  - Loads are required to get the operands
  - Generates the operation
- Keeps track of registers

A Simple Code Generator

- Uses few data structures
- Register descriptor:
  - Keeps track of variables names whose current value is in registers
- Address descriptor:
  - Keeps track of locations where the current value of variable can be found
    - Register
    - Memory address, etc.

Code Generation Algorithm

- Uses *getReg(I): will discuss in later slides*
  - Selects registers for *I*
  - Assumes enough registers

```
I: x=y+z
```

```
getReg(I): Rx, Ry, Rz
```

```
If y is not in Ry
generate LD Ry, y'
```

```
If z is not in Rz
generate LD Rz,z'
```

```
Gen Add Rx, Ry, Rz
```

Update descriptors!

Code Generation Algorithm: Updating descriptors

```
LD R0, x
```

Register Descriptor

R0	R1	R2	R3	R4	R5
x					

Address Descriptor

x	y	z	p	q	r
R0,x	y	z	p	q	r

Code Generation Algorithm: Updating descriptors

```
x=y+z
```

```
Add R3, R1, R2
```

Register Descriptor

R0	R1	R2	R3	R4	R5
	y	z	**x**		

Address Descriptor

x	y	z	p	q	r
R3	R1, y	R2, z	p	q	r

Code Generation Algorithm

```
I: x=y
```

```
getReg(I): Rx = Ry
```

```
If y is not in Ry
generate LD Ry, y'
```

- Issue store instructions at the end of basic block for live variables

Example

```
t = a-b
u = a-c
v = t+u
a = d
d = v+u
```

t, u, v: Temporary variables local to block
a, b, c, d: Live after exit of the block

Basic block instructions

Example

```
t = a-b
u = a-c
v = t+u
a = d
d = v+u
```

Register Descriptor

R1	R2	R3

Address Descriptor

a	b	c	d	t	u	v
a	b	c	d			

t=a-b

```
LD R1, a
LD R2, b
SUB R2, R1, R2
```

R1	R2	R3
a	**t**	

a	b	c	d	t	u	v
a,**R1**	b	c	d	**R2**		

u=a-c

```
LD R3, c
SUB R1, R1, R3
```

R1	R2	R3
u	t	**c**

a	b	c	d	t	u	v
a	b	c, **R3**	d	R2	**R1**	

Example

```
t = a-b
u = a-c
v = t+u
a = d
d = v+u
```

Register Descriptor

R1	R2	R3
u	t	c

Address Descriptor

a	b	c	d	t	u	v
a	b	c, R3	d	R2	R1	

v=t+u

```
ADD R3, R2, R1
```

R1	R2	R3
u	t	**v**

a	b	c	d	t	u	v
a	b	c	d	R2	R1	**R3**

a=d

```
LD R2, d
```

R1	R2	R3
u	**a,d**	v

a	b	c	d	t	u	v
R2	b	c	d, **R2**		R1	R3

Example

```
t = a-b
u = a-c
v = t+u
a = d
d = v+u
```

Register Descriptor **Address Descriptor**

R1	R2	R3
u	a,d	v

a	b	c	d	t	u	v
R2	b	c	d, R2		R1	R3

d=v+u

```
ADD R1, R3, R1
```

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	**R1**			R3

exit

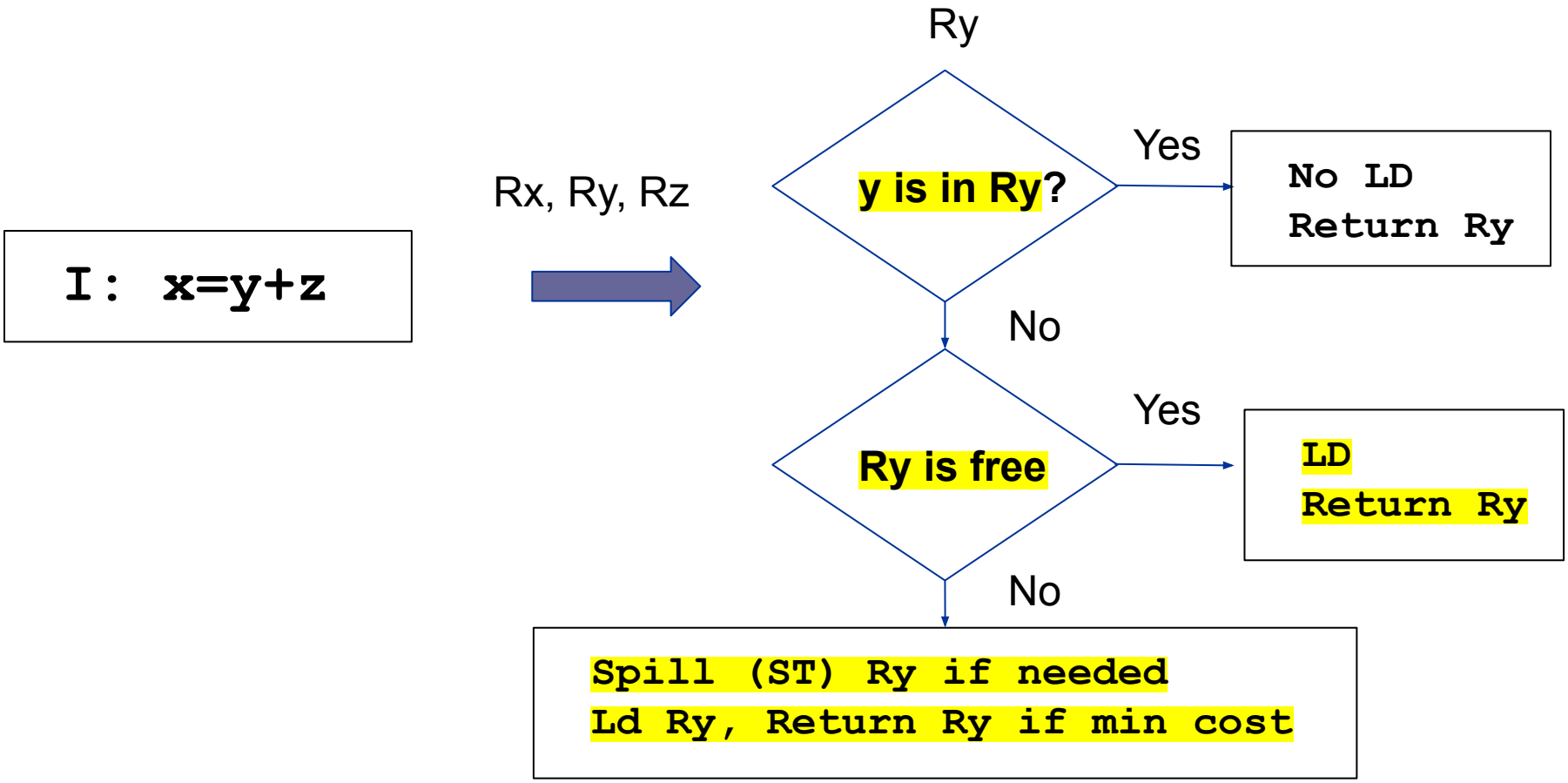
```
ST a, R2
ST d, R3
```

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a, R2	b	c	**d,** R1			R3

Design of getReg()

- Goal: For a TAC instruction I, gets the registers

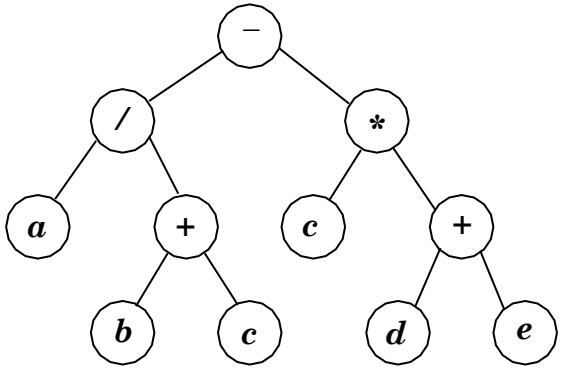


Sethi-Ullman Algorithm – Introduction

- Generates optimal code for expression trees.
  - Basic blocks containing single expression evaluation
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.

Expression Trees

- Here is the expression $a/(b + c) - c * (d + e)$ represented as a tree:



Target Machine Model

- We assume a machine with finite set of registers r0, r1, … rk , countable

1. $m \leftarrow r$ (store instruction)
2. $r \leftarrow m$ (load instruction)
3. $r \leftarrow r \; op \; m$ (the result of $r \; op \; m$ is stored in r)
4. $r_2 \leftarrow r_2 \; op \; r_1$ (the result of $r_2 \; op \; r_1$ is stored in r_2)

Note:

1. In instruction 3, the memory location is the right operand.
2. In instruction 4, the destination register is the same as the left operand register.

Overview

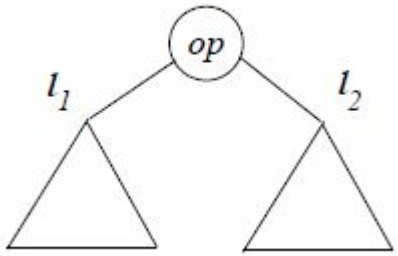
- Computes the minimum number of registers required to compute the expression tree -- labelling algorithm
- Generates the code

Overview

- Computes the minimum number of registers required to compute the expression tree -- labelling algorithm
- Generates the code

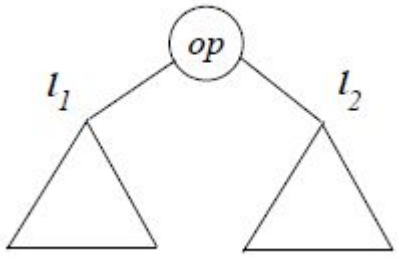
Order of Evaluation and Register Requirement

- Consider evaluation of a tree without stores. Assume that the left and right subtrees require up to l_1, and l_2 ($l_1 < l_2$) registers.
- In what order should we evaluate the subtrees to minimize register requirement?



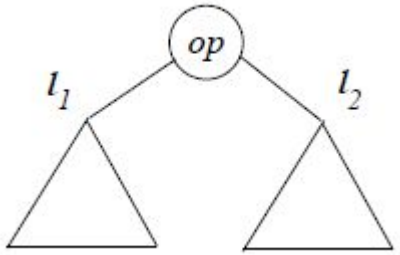
Order of Evaluation: Choice-1

- Left subtree first, leaving result in a register. This requires upto l_1 registers.
- Evaluate the right subtree. During this we require upto l_2 for evaluating the right subtree and one to hold value of the left subtree.
- Register requirement — $max\ (l_1, l_2 + 1) = l_2 + 1$.



Order of Evaluation: Choice-2

- Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require up to l_2 registers.
- Evaluate the left subtree. During this, we might require up to $l_1 + 1$ registers.
- Register requirement — $max\ (l_1 + 1,\ l_2) = l_2$



Therefore the subtree requiring more registers should be evaluated first.

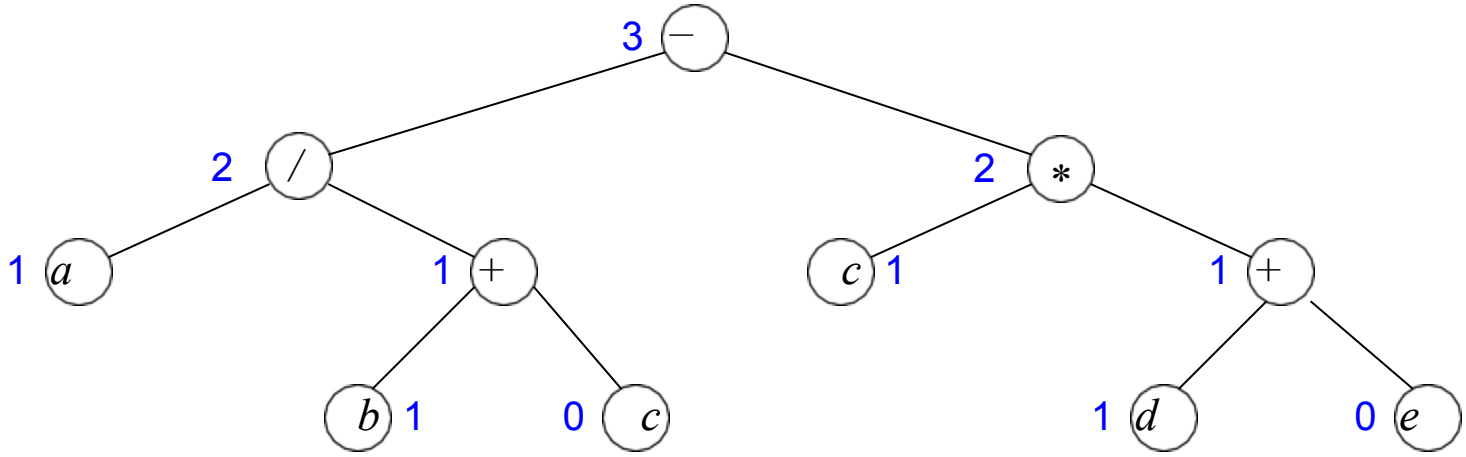
Labeling the Expression Tree

- Label each node by the number of registers required to evaluate it in a store free manner.
- Visit the tree in post-order. For every node visited do:
  1. Label each left leaf by 1 and each right leaf by 0.
     left leaf must necessarily be in a register
     the right leaf can reside in memory.
  2. If the labels of the children of a node *n* are l_1 and l_2 respectively, then

$$label(n) = max(l_1, l_2), \text{ if } l_2 \neq l_2$$
$$= l_1 + 1, \text{ otherwise}$$

Labeling the Expression Tree

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



Exercise

- Computes the minimum number of registers required to compute the expression tree using labelling algorithm

  ((a+b)*(c-d*e))+((f*g)/(h+i))

Next Lecture

- Computes the minimum number of registers required to compute the expression tree -- labelling algorithm
- Generates the code