CS251: Introduction to Language Processing

Code Generation and Optimizations

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

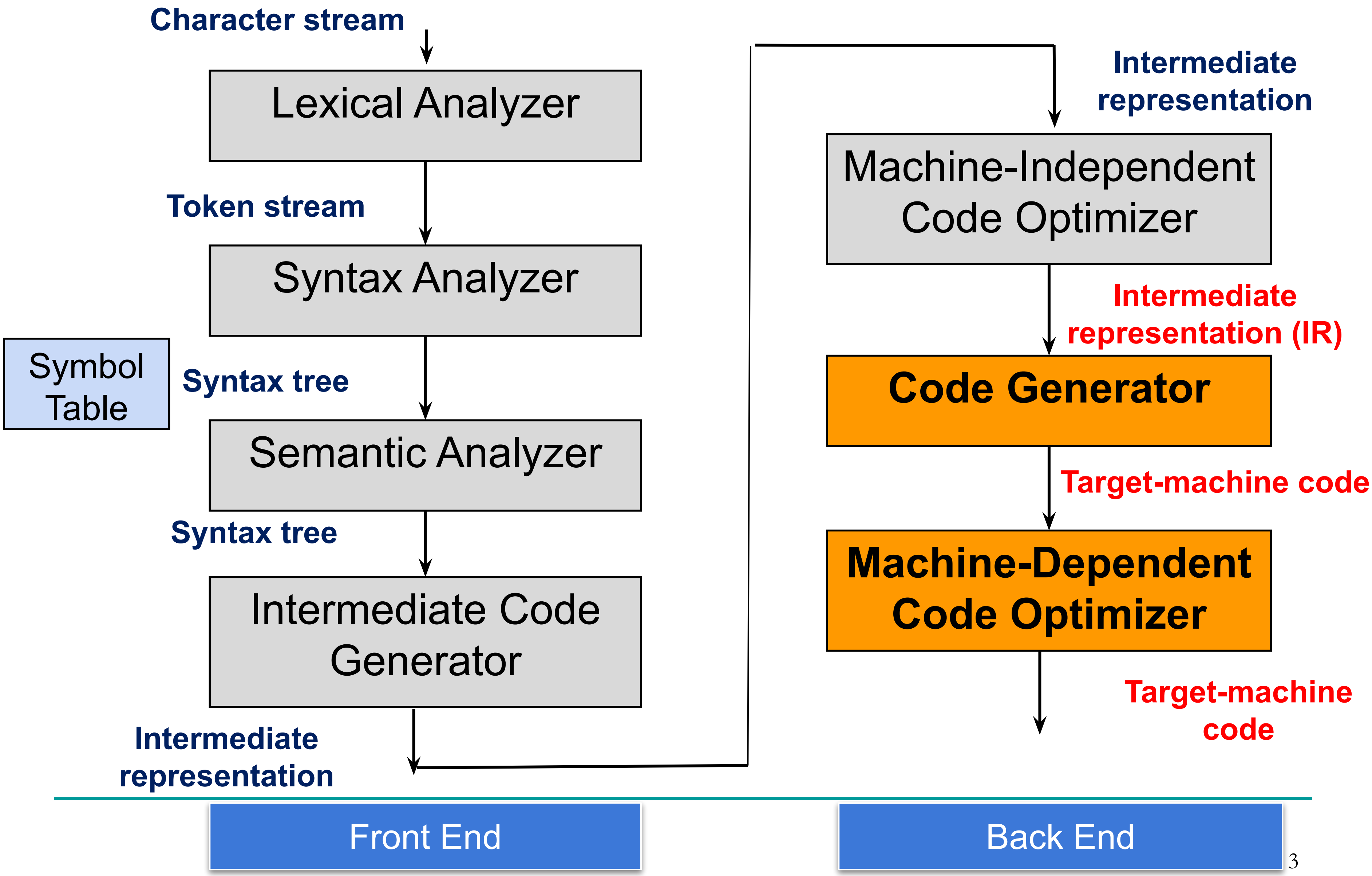
vishwesh@iitbhilai.ac.in

2023-24 M

Acknowledgement

- References for today's slides
  - *Prof. Amitabha Sanyal, IIT Bombay*
    - *https://www.cse.iitb.ac.in/~uday/courses/cs324-08/code-generation.pdf*
  - *Prof. Y. N Srikant, IISc Bangalore*
    - *https://nptel.ac.in/content/storage2/courses/106108052/module4/code-gen-part-2.pdf*
  - *Course textbook*

Compiler Design


Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Symbol Table
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation (IR)
Code Generator
Target-machine code
Machine-Dependent Code Optimizer
Target-machine code
Front End
Back End

Outline

- Code generation algorithms
  - Sethi-Ullman Algorithm

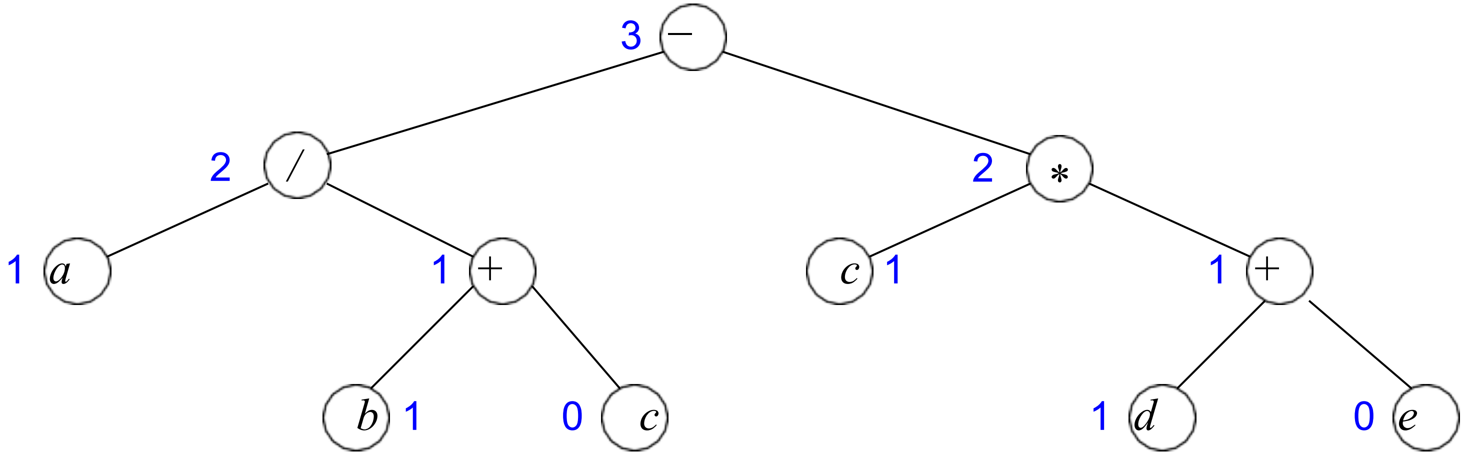
Overview

- 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

Labeling the Expression Tree



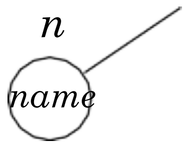
Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(*n*), which produces code to evaluate the node labeled *n*.
2. Register allocation is done from a stack of register names *rstack* , initially containing r_0, r_1, . . . , r_k (with r_0 on top of the stack).
3. *gencode*(*n*) evaluates *n* in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names *tstack*, initially containing t_0, t_1, . . . , t_k (with t_0 on top of the stack).
5. *swap*(*rstack*) swaps the top two registers on the stack.

The Algorithm

gencode(*n*) described by case analysis on the type of the node *n*.

1. *n is a left leaf:*

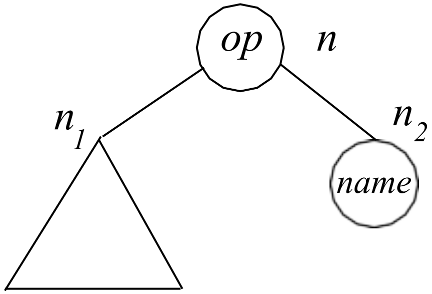


gen(*top*(*rstack*) ← *name*)

Comments: n is named by a variable say *name*. Code is generated to load *name* into a register.

The Algorithm

2. *n's right child is a leaf:*

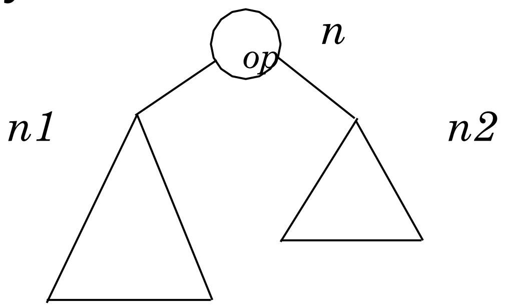


```
gencode($n_1$);
gen(top(rstack) ← top(rstack) op name)
```

Comments: n_1 is first evaluated in the register on the top of the stack, followed by the operation *op* leaving the result in the same register.

The Algorithm

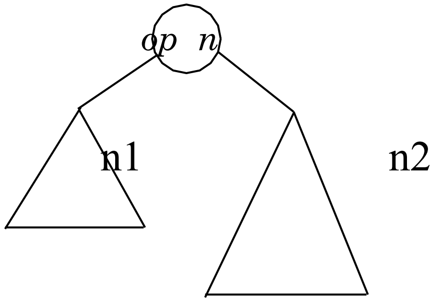
3. *The right child of n is lighter or as heavy as the left child. Its requirement is strictly less than the available number of registers*



```
gencode($n_1$);
R := pop(rstack );
gencode($n_2$);
gen(R ← R op top(rstack ));
push(rstack , R )
```

The Algorithm

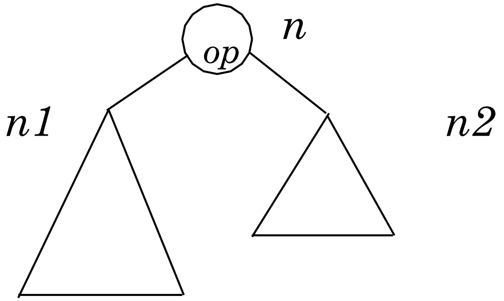
4. *The left child is the lighter subtree. This requirement is strictly less than the available number of registers*



```
swap(rstack );
gencode($n_2$);                               Evaluate right child
R := pop(rstack );
gencode($n_1$);                               Evaluate left child
gen(top(rstack ) ← top(rstack ) op R );       Issue op
push(rstack, R );
swap(rstack )                                 Restore register stack
```

The Algorithm

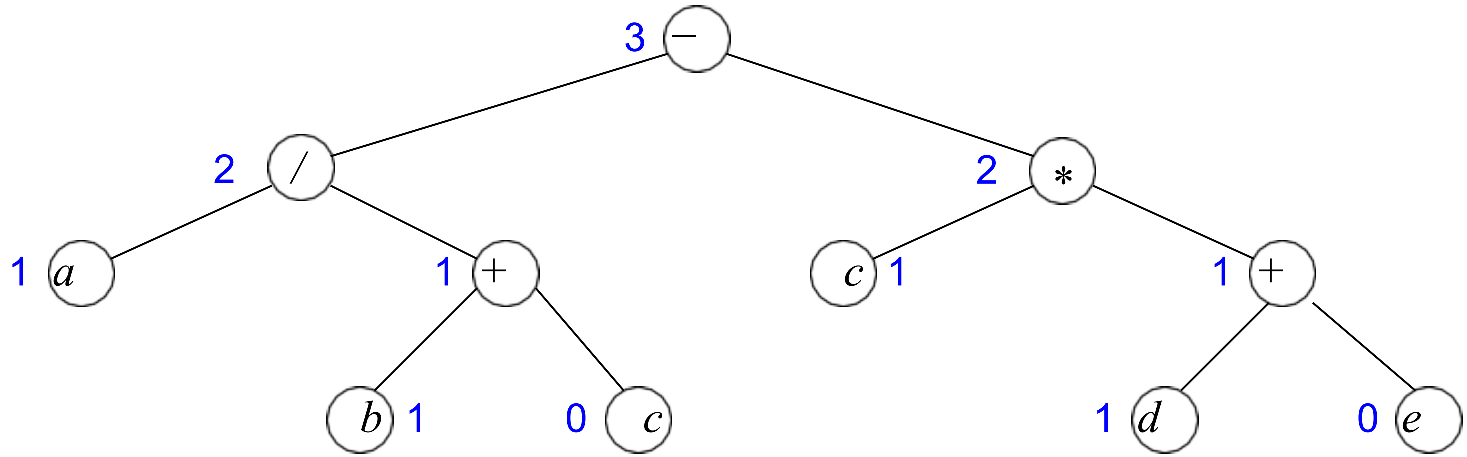
5. *Both the children of n require registers greater or equal to the available number of registers.*



```
gencode($n_2$);
$T$ := pop(tstack);
gen($T \leftarrow$ top(rstack));
gencode($n_1$);
push(tstack, $T$);
gen(top(rstack) $\leftarrow$ top(rstack) op $T$ ;
```

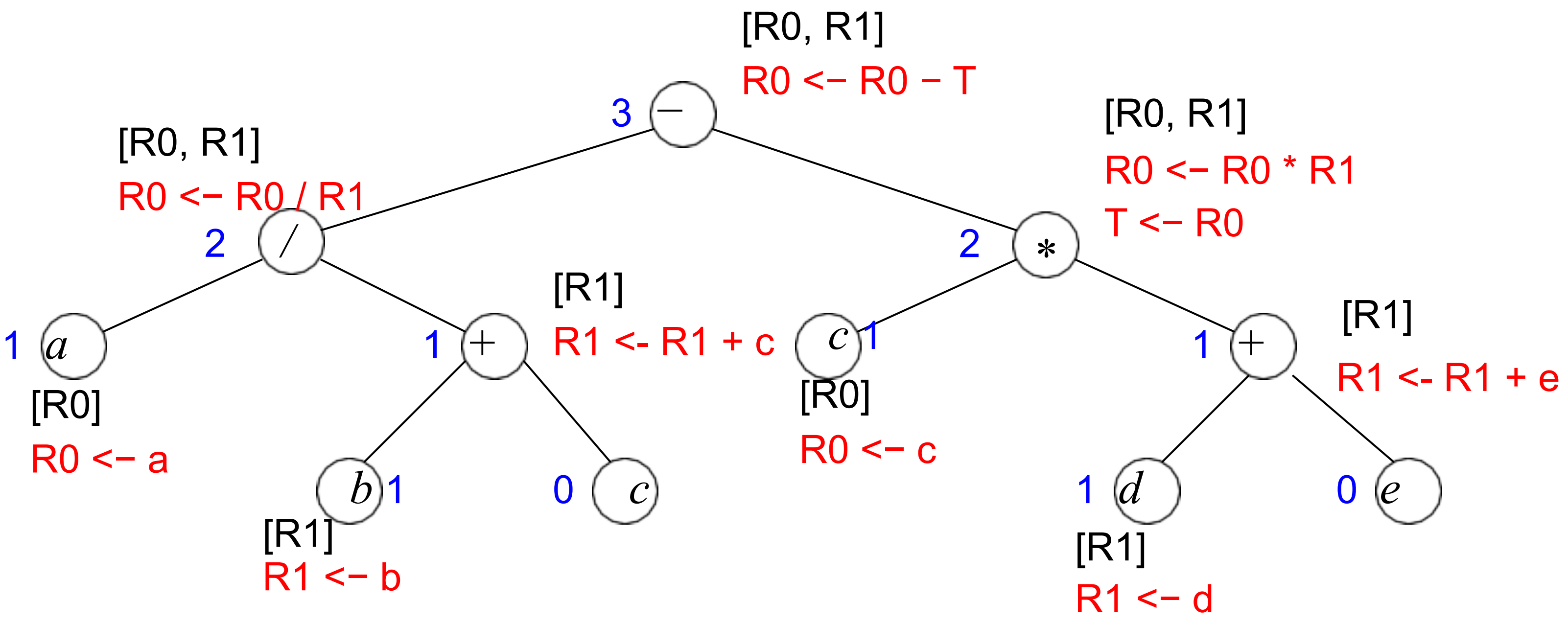
Comments: Evaluate the right sub-tree into a temporary. Then evaluate the left sub-tree and *n* into the register on top of stack.

Example

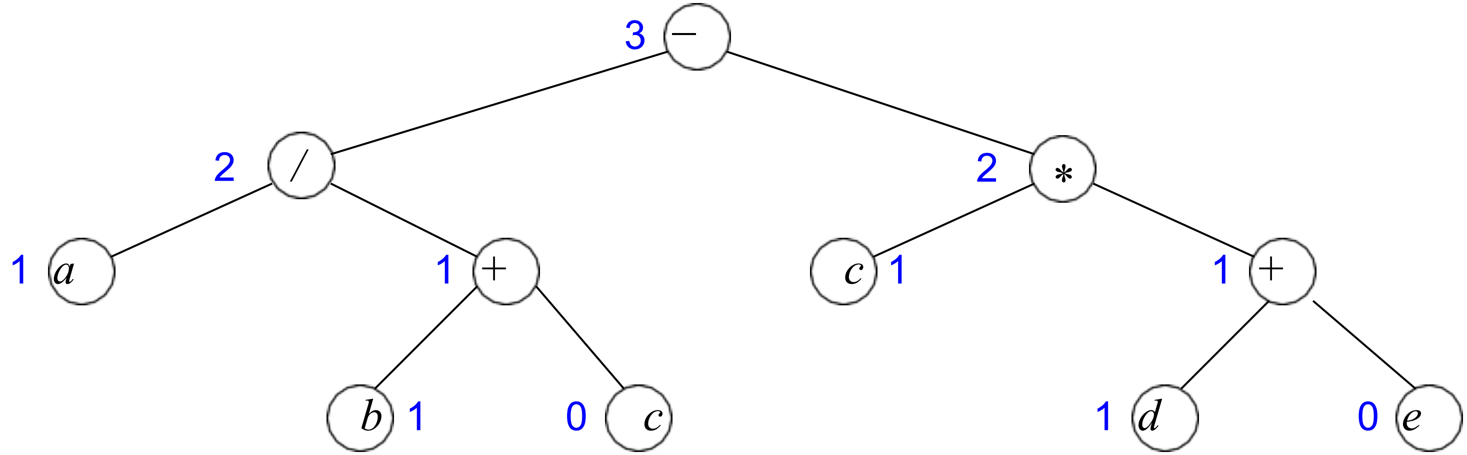


assuming two available registers r_0 and r_1, the calls to gencode and the generated code are shown below.

Example


[R0, R1]
R0 <− R0 − T
3
−
[R0, R1]
R0 <− R0 / R1
2
/
[R0, R1]
R0 <− R0 * R1
T <− R0
2
*
1
a
[R0]
R0 <− a
1
+
[R1]
R1 <- R1 + c
b
1
[R1]
R1 <− b
0
c
c
1
[R0]
R0 <− c
1
+
[R1]
R1 <- R1 + e
1
d
[R1]
R1 <− d
0
e

Home Exercise

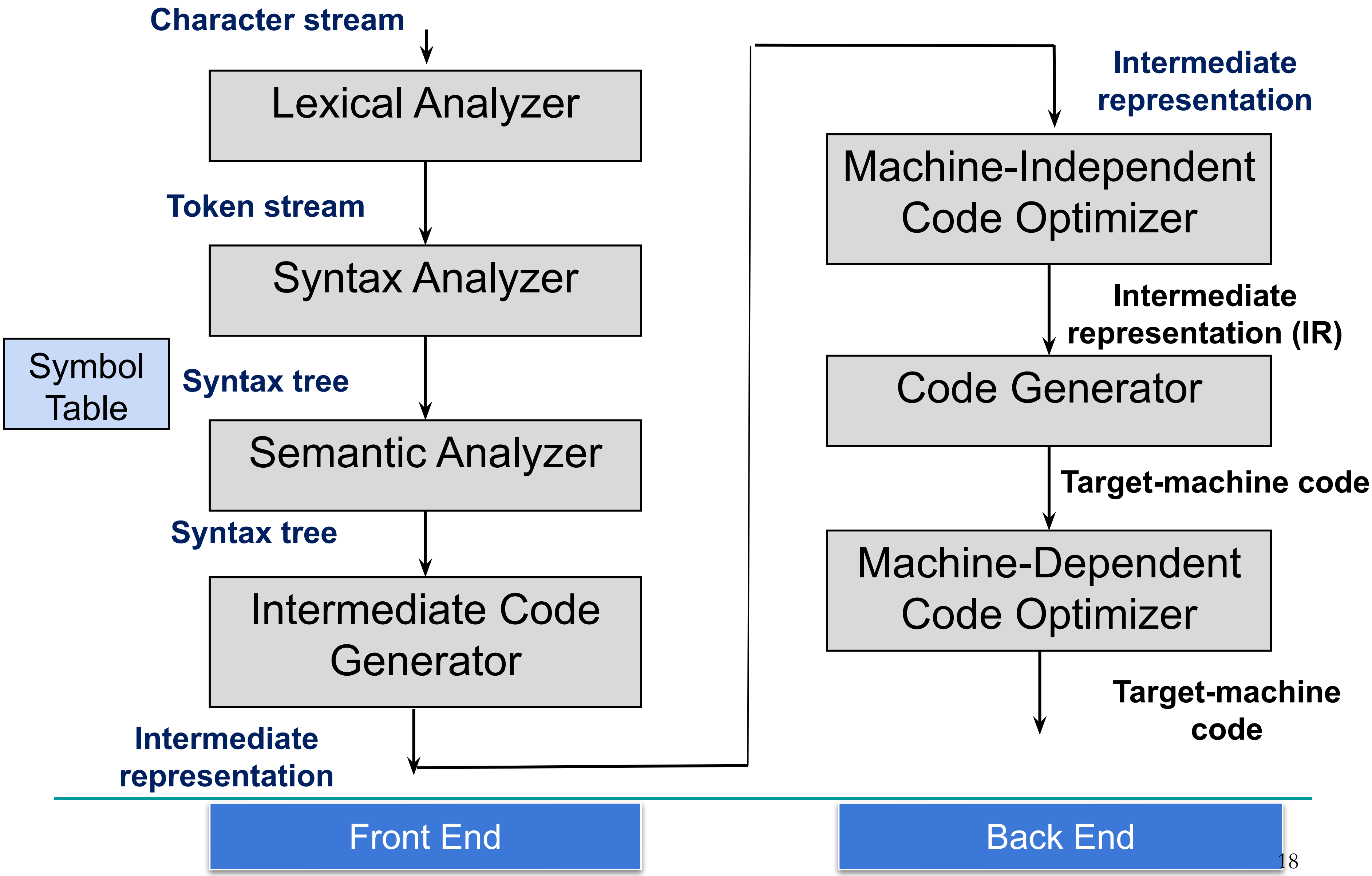


Assuming *three* available registers $r_{0,}$ r_1 and r_2 what would be the generated code?

Summary

- Code generation algorithm
  - Sethi-Ullman Algorithm

Summary of Course



Character stream

Lexical Analyzer

Token stream

Syntax Analyzer

Symbol Table

Syntax tree

Semantic Analyzer

Syntax tree

Intermediate Code Generator

Intermediate representation

Intermediate representation

Machine-Independent Code Optimizer

Intermediate representation (IR)

Code Generator

Target-machine code

Machine-Dependent Code Optimizer

Target-machine code

Front End

Back End

Summary of Course

- Concepts of compiler design
  - Theory
  - Practice

Questions?