

CS 335: Code Generation

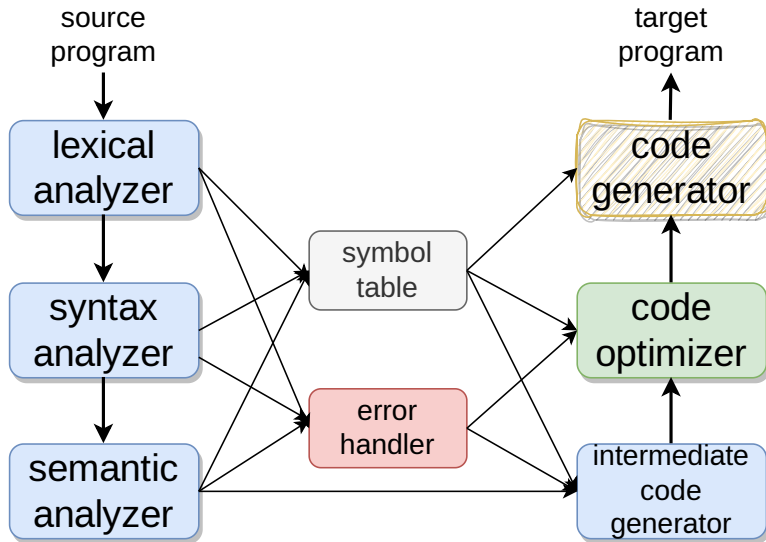
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

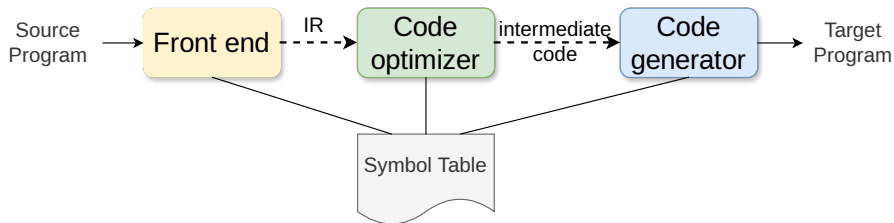
Sem 2023-24-II



An Overview of Compilation



Code Generation



- (i) Generated output code must be correct
- (ii) Generated code must be of “good” quality
 - ▶ Notion of good can vary
 - ▶ Should make efficient use of resources on the target machine
- (iii) Code generation should be efficient

Generating optimal code is undecidable, compilers make use of well-designed heuristics

Code Generation

Input

- Intermediate representation (IR) generated by the front end
 - ▶ Linear IRs like 3AC or stack machine representations, or graphical IRs
- Symbol table information

Assumptions

- Code generation does not bother with error checking
- Code generation assumes that types in the IR can be operated on by target machine instructions
 - ▶ For example, bits, integers, and floats

Code Generation

Output

- Absolute machine code
 - ▶ Generated **addresses are fixed** and work when loaded at fixed locations in memory
 - ▶ Efficient to execute, now primarily used in embedded systems
- Relocatable machine code
 - ▶ Code can be broken down into separate sections and loaded anywhere in memory that meets size requirements
 - ▶ Allows for separate compilation but requires a **separate linking and loading** phase
- Assembly language
 - ▶ Simplifies code generation, but **requires assembling** the generated code

Steps in Code Generation

- Compiler back end performs three steps to translate IR to executable code
 - Instruction selection** Choose appropriate target machine instructions while generating code
 - Register allocation** Decide what values to keep in which registers
 - Instruction scheduling** Decide in what order to schedule the execution of instructions
- Need to also emit code to **manage memory** during execution

Instruction Selection

- Possible idea: Match patterns and replace them with a pre-decided template
 - (i) Devise a target code skeleton for every 3AC IR instruction
 - (ii) Replace every 3AC instruction with the skeleton

$x = y + z$

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

$x = y + z$

$w = x + v$

```
LD R0, y
ADD R0, R0, z
ST x, R0
LD R0, x
ADD R0, R0, v
ST w, R0
```

redundant

Instruction Selection

- Each IR instruction can be translated in several ways, the **challenge** is to pick an **efficient** variant

a = a + 1	LD R ₀ , a
	ADD R ₀ , R ₀ , #1
	ST a, R ₀
	INC a

- Need a cost model and heuristics for instruction selection
 - ▶ Influential factors are the level of abstraction of the IR, speed of instructions, energy consumption, and space overhead
- Target ISA also influences instruction selection
 - Scalar RISC machine simple mapping from IR to assembly
 - CISC machine may need to fuse multiple IR operations for effectively using CISC instructions
 - Stack machine needs to translate implicit names and destructive instructions to assembly

Register Allocation

- Instructions operating on register operands are more efficient

Register allocation Choose which variables will reside in registers

Register assignment Choose which registers to assign to each variable

- Architectures may impose restrictions on the usage of registers
- Finding an optimal assignment of registers to variables is NP-complete

Architectures such as IBM 370 may require register pairs to be used for some instructions

```
MUL x, y
```

- x is in the even register, y is in the odd register
- Product occupies the entire even/odd register pair

```
DIV x, y
```

- 64-bit dividend occupies the even/odd register pair
- Even register holds the remainder, odd register the quotient

Instruction Scheduling

- Order of evaluating the instructions also affects the efficiency of the target code
- Instruction scheduling reorders instructions to maximize utilization of hardware resources and minimize cycles
- Selecting the best order across inputs is an NP-complete problem

Target Machine for Code Generation

- Efficient code generation requires a good understanding of the target ISA
- **Assumptions**
 - ▶ Three-address machine, byte-addressable with four-byte words
 - ▶ n general-purpose registers
 - ▶ Limited instruction set
 - ▶ `OP dst, src1, src2`
 - ▶ `LD dst, addr`
 - ▶ `ST dst, src`
 - ▶ `BR L`
 - ▶ `Bcond r, L`

Addressing Modes

Specifies how to **interpret the operands** of an instruction

Mode	Form	Address	Example
absolute	M	M	LD R ₀ , M
register	R	contents(R)	ADD R ₀ , R ₁ , R ₂
indexed	c(R)	contents(c + contents(R))	LD R ₁ , 4(R ₀)
indirect register	*R	contents(contents(R))	LD R ₁ , *R ₀
indirect indexed	*c(R)	contents(contents(c + contents(R)))	LD R ₁ , *100(R ₀)
immediate	#c	c	LD R ₁ , #1

Examples of Code Generation

$x = y - z$	LD R_1, y	// $R_1 = y$
	LD R_2, z	// $R_2 = z$
	SUB R_1, R_1, R_2	// $R_1 = R_1 - R_2$
	ST x, R_1	// $x = R_1$

if $x < y$ goto L	LD R_1, x	// $R_1 = x$
	LD R_2, y	// $R_2 = y$
	SUB R_1, R_1, R_2	// $R_1 = R_1 - R_2$
	BLTZ R_1, M	// if $R_1 < 0$ JMP M

$b = a[i]$	LD R_1, i	// $R_1 = i$
	MUL $R_1, R_1, 8$	// $R_1 = R_1 * 8$
	LD $R_2, a(R_1)$	// $R_2 = c(a+c(R_1))$
	ST b, R_2	// $b = R_2$

$a[j] = c$	LD R_1, c	// $R_1 = c$
	LD R_2, j	// $R_2 = j$
	MUL $R_2, R_2, 8$	// $R_2 = R_2 * 8$
	ST $a(R_2), R_1$	// $c(a+c(R_2)) = R_1$

$x = *p$	LD R_1, p	// $R_1 = p$
	LD $R_2, 0(R_1)$	// $R_2 = c(0+c(R_1))$
	ST x, R_2	// $x = R_2$

$*p = y$	LD R_1, p	// $R_1 = p$
	LD R_2, y	// $R_2 = y$
	ST $0(R_1), R_2$	// $c(0+c(R_1)) = R_2$

Runtime Storage Management

Assume that the first location in the activation record (given by `staticArea`) of the callee stores the return address of the caller

Static Allocation	
ST callee.staticArea, #here+20	Store return address in the first slot in the callee's activation record, assume 2 opcodes and 3 constants are each of 4 bytes
BR callee.codeArea	Transfer control to callee code
...	

return
address →

...	Callee's code area
BR *callee.staticArea	Transfer control to caller
...	

Determine Addresses in Target Code

Need to generate code to manage activation records at runtime

3AC
<pre>// code for func c action₁ call p action₂ halt // return to OS</pre>
<pre>// code for func p action₃ return</pre>

	Activation record for c (64 bytes)
0:	return address
4:	arr
56:	i
60:	j

	Activation record for p (88 bytes)
0:	return address
4:	buf
84:	n

Target Code for Static Allocation

text area

		// code for c
100:	ACTION ₁	// assume takes 20 bytes
120:	ST 364, #140	// save return address 140
132:	BR 200	// call p
140:	ACTION ₂	
160:	HALT	// terminate, return to OS
		// code for p
200:	ACTION ₃	
220:	BR *364	// return to address saved // in location 364

stack area with
activation records

		// 300-363 hold the activation // record for c
300:		// return address
304:		// local data for c
		// 364-451 hold the activation // record for p
364:	140	// return address
368:		// local data for p

Stack Allocation

Code for the caller

LD SP, #stackStart code HALT	// initialize the stack // terminate execution
------------------------------------	---

Code for procedure call

ADD SP, SP, #caller.ARSize ST *SP, #here + 20 BR callee.codeArea	// increment stack pointer // save return address in // callee's frame // jump to caller
--	---

Code for return sequence in the callee

BR *0(SP)	// return to caller
-----------	---------------------

Code for return sequence in the caller

SUB SP, SP, #caller.ARSize	// decrement stack pointer
----------------------------	----------------------------

Target Code for Stack Allocation

3AC
// code for s action ₁ call q action ₂ halt
// code for p action ₃ return
// code for q action ₄ call p action ₅ call q action ₆ call q return

		// code for s
100:	LD SP, #600	// initialize the stack
108:	ACTION ₁	// code for action ₁
128:	ADD SP, SP, #ssize	// call sequence begins
136:	ST 0(SP), #152	// push return address
144:	BR 300	// call q
152:	SUB SP, SP, #ssize	// restore SP
160:	ACTION ₂	
180:	HALT	

		// code for p
200:	ACTION ₃	
220:	BR *0(SP)	// return to caller

		// code for q
300:	ACTION ₄	// conditional jump to 456
320:	ADD SP, SP, #qsize	
328:	ST 0(SP), #344	// push return address
336:	BR 200	// call p
344:	SUB SP, SP, #qsize	// restore SP
352:	ACTION ₅	
372:	ADD SP, SP, #qsize	
380:	ST 0(SP), #396	// push return address
388:	BR 300	// call q
396:	SUB SP, SP, #qsize	// restore SP
404:	ACTION ₆	
424:	ADD SP, SP, #qsize	
432:	ST 0(SP), #448	// push return address
440:	BR 300	// call q
448:	SUB SP, SP, #qsize	// restore SP
456:	BR *0(SP)	// return to caller

600:		// stack starts
------	--	-----------------

Basic Blocks and Control Flow Graphs

Basic Block (BB)

Definition

A BB is a **maximal** sequence of instructions with only one entry and one exit point

- Entry is at the start of the BB, and exit is from the end of the BB
- Only the start/leader instruction can be the target of a JUMP instruction
- There are no jumps in or out of the middle of a BB

- Identifying BBs

- (i) The first instruction of the input code is a **leader**
- (ii) Instructions that are targets of conditional/unconditional jumps are leaders
- (iii) Instructions that immediately follow conditional/unconditional jumps are leaders

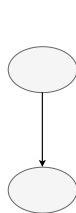
Identifying BBs

(1)	$i = 1$	
(2)	$j = 1$	
(3)	$t_1 = 10 \times i$	target
(4)	$t_2 = t_1 + j$	
(5)	$t_3 = 8 \times t_2$	
(6)	$t_4 = t_3 - 88$	
(7)	$a[t_4] = 0.0$	
(8)	$j = j + 1$	
(9)	if $j \leq 10$ goto (3)	
(10)	$i = i + 1$	
(11)	if $i \leq 10$ goto (2)	
(12)	$i = 1$	follows a conditional
(13)	$t_5 = i - 1$	
(14)	$t_6 = 88 \times t_5$	
(15)	$a[t_6] = 1.0$	
(16)	$i = i + 1$	
(17)	if $i \leq 10$ goto (13)	

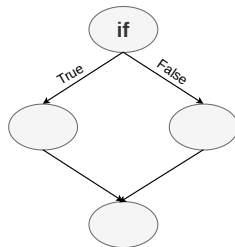
- Statements (1), (2), (3), (10), (12), and (13) are leaders
- There are six BBs: (1), (2), (3)–(9), (10)–(11), (12), (13)–(17)

Control Flow Graph (CFG)

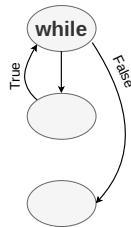
- Graphical representation of control flow during execution of a program
 - ▶ Each node represents a statement or a BB
 - ▶ An entry and an exit node are often added to a CFG for a function
 - ▶ An edge represents the **possible** transfer of control between nodes
- Used for static program analysis (e.g., compiler optimizations like instruction scheduling and global register allocation)



straight-line code



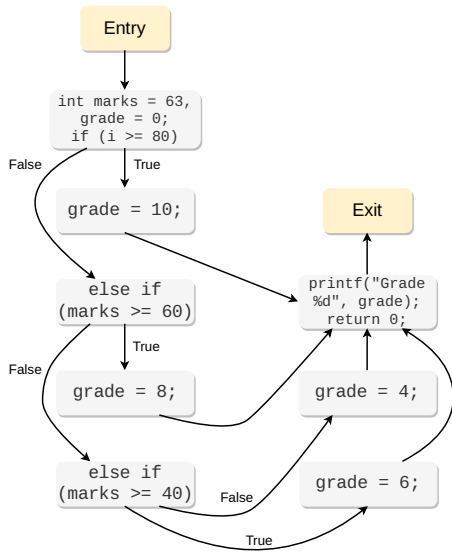
predicated code



loop-based code

Example of BBs and a CFG

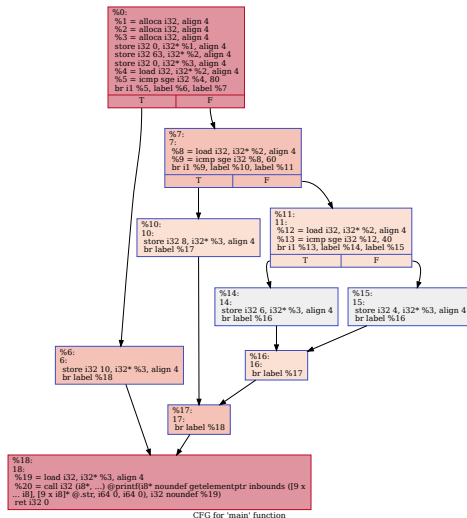
```
int main() {  
    int marks = 63, grade = 0;  
    if (marks >= 80)  
        grade = 10;  
    else if (marks >= 60)  
        grade = 8;  
    else if (marks >= 40)  
        grade = 6;  
    else  
        grade = 4;  
    printf("Grade %d", grade);  
    return 0;  
}
```



Example CFG Generated with LLVM

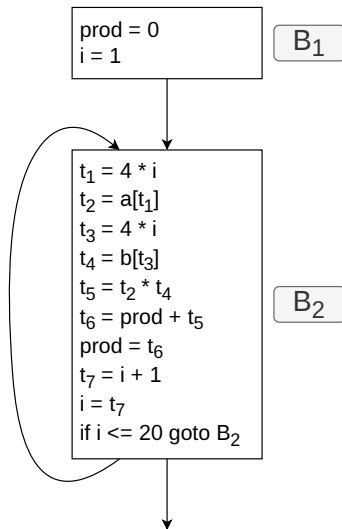
```
int main() {  
    int marks = 63, grade = 0;  
    if (marks >= 80)  
        grade = 10;  
    else if (marks >= 60)  
        grade = 8;  
    else if (marks >= 40)  
        grade = 6;  
    else  
        grade = 4;  
    printf("Grade %d", grade);  
    return 0;  
}
```

```
$ clang++ -S -emit-llvm ctrl-flow.cpp -o ctrl-flow.ll  
$ opt -analyze -enable-new-pm=0 -dot-cfg ctrl-flow.ll  
$ dot -Tpdf -o ctrl-flow.pdf .main.dot
```



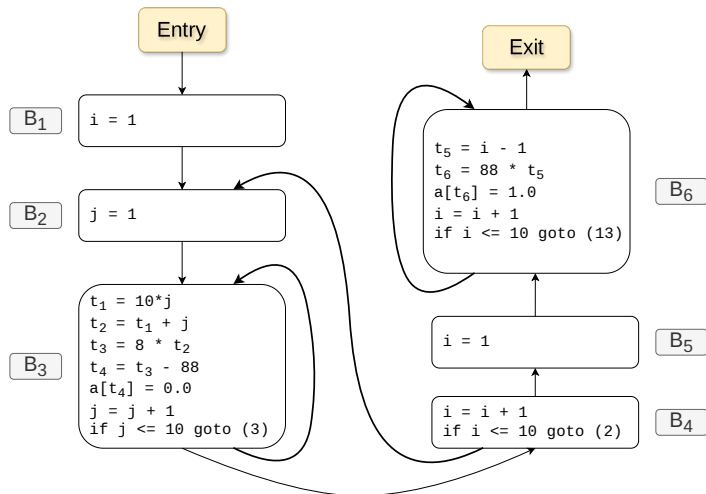
Loops in a CFG

- A set of CFG nodes L form a loop if L contains a unique loop entry node e such that
 - ▶ e is not the Entry node,
 - ▶ No node in L besides e has a predecessor outside L ,
 - ▶ Only way to reach a node in L from outside the loop is through e
 - ▶ Every node in L has a nonempty path to e that is completely within L
 - ▶ All nodes in the group are strongly connected



Example CFG

```
1  i = 1
2  j = 1
3  t1 = 10 * i
4  t2 = t1 + j
5  t3 = 8 * t2
6  t4 = t3 - 88
7  a[t4] = 0.0
8  j = j + 1
9  if j <= 10 goto (3)
10 i = i + 1
11 if i <= 10 goto (2)
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1.0
16 i = i + 1
17 if i <= 10 goto (13)
```



Optimizing BBs

Local Optimizations

Optimization of BBs

- Code optimizations can lead to substantial improvement in running time and/or energy consumption
- Global optimization analyzes control flow, data flow, and data dependence **among** BBs
- Local (i.e., intra-BB) optimizations can also provide **significant improvements** in code quality
 - ▶ Local transformations should **not change** the set of expressions computed by a block
 - ▶ Two BBs are equivalent if they compute the same set of expressions
 - ▶ **Expressions** are values of names that are **live on exit** from a BB

Next Use and Liveness

- Knowing when the value of a variable will be **used next** is important for generating good code
 - ▶ For example, can remove variables from registers if not used
- Consider the 3AC instruction $I: x = y + z$
 - ▶ We say I defines x and uses y and z
- If a statement J uses x and control can flow from I to J along a path where x is not redefined, then J **uses** the value of x defined at I

Definition

A name in a BB is **live** at a given point if its value is used after that point

- Given I and J , we say x is live at statement I

X is live at (5), X's next use at (5) is (15)

(5) $X = \dots$

(no redefinition of X)

(15) $\dots = \dots X \dots$

X is dead at (15) because there is no further use

(no use of X)

(25) $X = \dots$

Example of Next Use and Liveness

Intermediate Code	Live			Next Use		
	x	y	z	x	y	z
(1) $x = y + z$	T	F	F	(2)	-	-
(2) $z = x * 5$	F		T	-		(3)
(3) $y = z - 7$		T	T		(4)	(4)
(4) $x = z + y$	F	F	F	-	-	-

Determining Next Use and Liveness Information

- Input**
- A BB (say B) of 3AC
 - Assume symbol table shows all non-temporary variables in B as live on exit and all temporaries are dead on exit

- Output**
- Liveness and next use information for each instruction $I: x = y \text{ op } z$ in B

- Algorithm**
- Scan forward over B to initialize liveness and next use information for (i) each used variable in B , and (ii) each instruction I in B
 - Scan backward over B . For each instruction $I: x = y \text{ op } z$ in B , do
 - Copy the liveness and next use information for x , y , and z from the symbol table to tuple I
 - Update x , y , and z 's symbol table entries
 - Set $x.\text{live} = \text{FALSE}$ and $x.\text{next_use} = \text{NONE}$
 - Set $y.\text{live} = z.\text{live} = \text{TRUE}$ and $y.\text{next_use} = z.\text{next_use} = I$

Example Computation of Next Use and Liveness Information

Intermediate Code	Symbol Table Information						Instruction Information					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) $x = y + z$	F	F	F	-	-	-	F	F	F	-	-	-
(2) $z = x * 5$	F	F	F	-		-	F	F	F	-		-
(3) $y = z - 7$	F	F	F		-	-	F	F	F		-	-
(4) $x = z + y$	F	F	F	-	-	-	F	F	F	-	-	-

after the
forward pass

Example Computation of Next Use and Liveness Information

Intermediate Code	Symbol Table Information						Instruction Information					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(4) $x = z + y$	F	T	T	-	(4)	(4)	F	F	F	-	-	-
(3) $y = z - 7$	F	F	T	-	-	(3)	F	T	T	-	(4)	(4)
(2) $z = x * 5$	T	F	F	(2)	-	-	F	F	T	-	-	(3)
(1) $x = y + z$	F	T	T	-	(1)	(1)	T	F	F	(2)	-	-

after the
backward pass

Structure-Preserving Transformations

- **Common subexpression elimination**

- ▶ Instructions compute a value that has been computed

- **Dead code elimination**

- ▶ Remove instructions that define variables that are never used

- **Renaming temporary variables**

- ▶ Can always transform a BB into an equivalent block where each statement that defines a temporary uses a new name
- ▶ Such a BB is called a normal-form block

- **Reordering of dependence-free statements**

- ▶ Normal-form blocks permit statement interchanges without affecting the value of the block
- ▶ May improve latency of accesses and register usage

a = b + c	a = b + c
b = a - d	b = a - d
c = b + c	c = b + c
d = a - d	d = b

$t_1 = b + c$
$t_2 = x + y$

Algebraic Transformations

- Apply algebraic laws to simplify computation

Strength Reduction	
Expensive	Cheaper
x^2	$x * x$
$2 * x$	$x + x$
$x / 2$	$x >> 1$

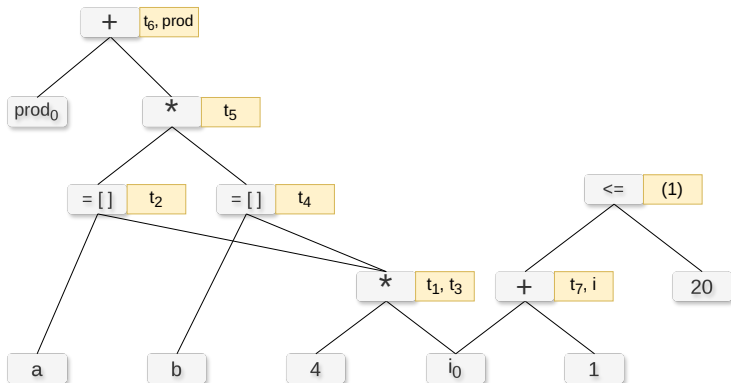
$$\begin{aligned}x + 0 &= 0 + x = x \\x * 1 &= 1 * x = x \\x - 0 &= x \\x / 1 &= x\end{aligned}$$

- Constant folding evaluates constants during compilation (e.g., $i = 2 * 3.14 * 300 * 300;$)
- Relational operators can generate common sub-expressions (e.g., $x > y$ and $x - y$)

DAG Representation of BBs

Many optimizations are easier to perform on a DAG representation of BBs

```
1  t1 = 4 * i
2  t2 = a[t1]
3  t3 = 4 * i
4  t4 = b[t3]
5  t5 = t2 * t4
6  t6 = prod + t5
7  prod = t6
8  t7 = i + 1
9  i = t7
10 if i <= 20 goto (1)
```



Representing BBs with DAGs

- Rules on the DAG structure
 - ▶ Leaf nodes are labeled with variable names or constants
 - ▶ Initial values for each variable are represented by a node
 - ▶ A node N is associated with each statement s in a BB
 - ▶ Children of N correspond to statements that last define the operands used in s
 - ▶ Inner nodes are labeled by an operator symbol
 - ▶ Node N is labeled by the operator applied at s
 - ▶ Nodes optionally have a sequence of identifiers for labels
 - ▶ Output nodes are those variables that are live on exit
- Each BB node in a CFG can be represented with a DAG

Constructing a DAG

Input

- A basic block (BB)

Output

- A DAG for the BB with the following information
 - ▶ a label for each node (ID for leaf nodes and operator symbols for interior nodes)
 - ▶ a list of identifiers (not constants) for each node

Assumptions

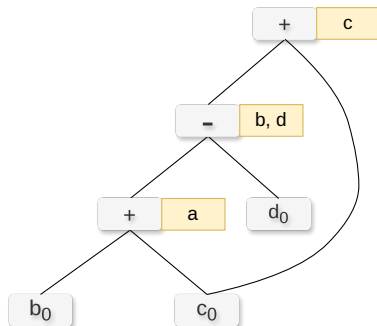
- Three kinds of 3AC: (i) $x = y \text{ op } z$, (ii) $x = \text{op } y$, and (iii) $x = y$
- Relational statements like `if $i \leq 20$ goto (1)` are treated like case (i)

Steps

- For each statement in the BB,
 - (i) If $node(y)$ is undefined, create a leaf labeled y and set $node(y)$ to the new node
 - (ii) For case (i), check if there is a node in the DAG labeled op with left child $node(y)$ and right child $node(z)$. If not, then create a node (denoted by n).
 - (iii) For case (ii), check if there is a node labeled op with $node(y)$ as the only child. If not, then create a node (denoted by n).
 - (iv) Delete x from the list of identifiers for $node(x)$. Append x to the list of identifiers for the node and set $node(x)$ to n .

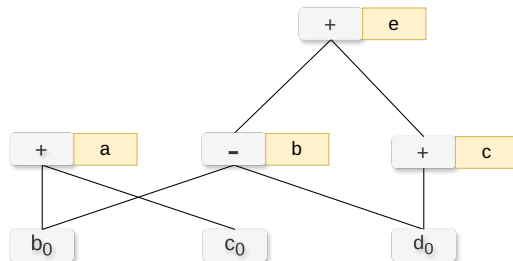
Local Common Subexpressions

$a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$



$a = b + c$
 $b = b - d$
 $c = c + d$
 $e = b + c$

DAG fails to capture that the 1st and 4th statements compute the same values

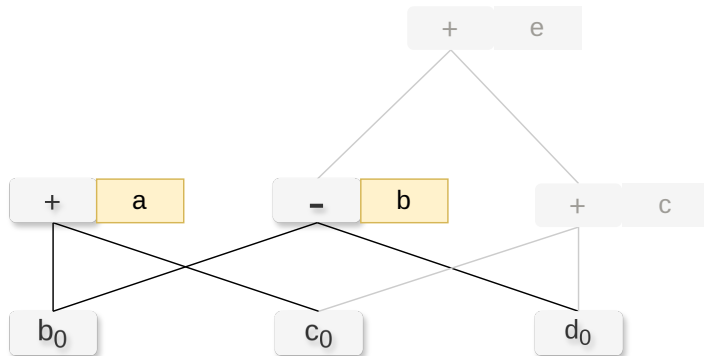


Dead Code Elimination

- Delete a root node from the DAG if it has no live variables
 - ▶ Repeat till there are no such nodes

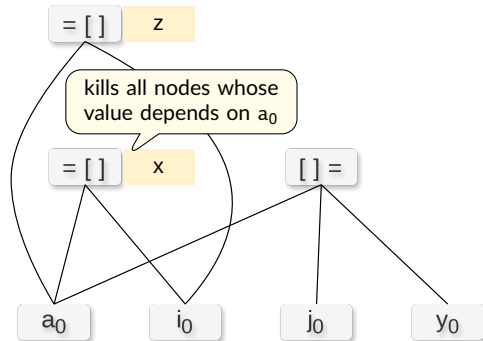
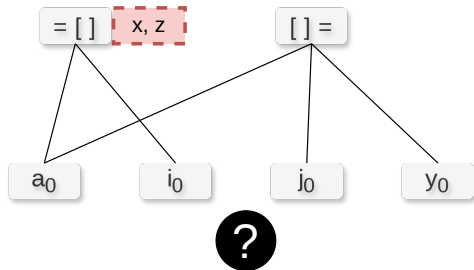
$a = b + c$
 $b = b - d$
 $c = c + d$
 $e = b + c$

Assume only a and b are live on exit



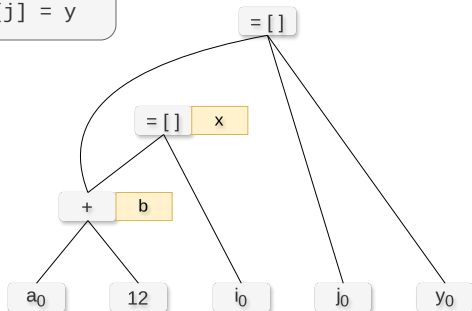
Representing Array References

```
x = a[i]  
a[j] = y  
z = a[i]
```



Consider Other Sources of Possible Aliasing

```
b = a + 12  
x = b[i]  
b[j] = y
```



```
// Use of every possible variable  
x = *p  
// Possible assignment to every variable  
*q = y
```

- `=*` must include all nodes for optimization analysis
- `*=` kills all other nodes
- Possible to use more precise pointer analysis
- Suppose there is a variable x defined at a node n that is in the scope of a procedure P
- We will conservatively assume that P uses x attached to n and kills node n

Code Generation Algorithm

Single Basic Blocks

Code Generation Strategy

Goal

- Generate target code for a sequence of 3AC **within** a BB

Assumption

- Every 3AC operator has an equivalent operator in the target language
- Computed values can reside in registers and only need to be saved when
 - (i) the register is required for another computation, or
 - (ii) just before a procedure call, jump, or a labeled statement
- Implies every register must be saved before the end of a BB

Steps

- For each 3AC,
 - ▶ Identify variables that need to be loaded into registers,
 - ▶ Load the variables into registers,
 - ▶ Generate code for the instruction,
 - ▶ Generate a store if the result needs to be saved in memory.

Challenges in Code Generation

Different Possibilities		
$a = b + c$	ADD R_i, R_i, R_j	b is in R_i , c is in R_j , b is no longer live on exit
	ADD R_i, R_i, c	b is in R_i , b is no longer live on exit
	MOV R_j, c ADD R_i, R_i, R_j	b is in R_i , b is no longer live on exit

Usually there will be numerous cases to consider

- An efficient choice depends on several factors (e.g., frequency of use of b and c later)
- Properties of the operator (e.g., commutativity) can add to the complexity

A Simple Code Generator

- Treat each IR quadruple as a “macro”
- Replace the macro with pre-existing code templates

Simple to implement but makes **inefficient use** of registers

Goal: Track values in registers and reuse them

Register and Address Descriptors

Register Descriptor

- Keeps track of **what name** is stored in each register, consulted whenever a new register is needed
- Each register holds the value of zero or more names at any time during an execution

Address Descriptor

- Keeps track of the **location(s)** where the current value of a name can be found at runtime
 - ▶ Location can be a register, a stack location, a memory address, or some combination of these (data can get copied)
- Information can be stored in the symbol table

Code Generation Algorithm

- For each 3AC instruction I of the form $x = y \text{ op } z$,
 - ▶ Invoke function $getreg(I)$ to select registers R_x , R_y , and R_z
 - ▶ If y is not in R_y according to the address descriptor, then generate instruction LD R_y, y'
 - ▶ y' is one of the memory locations for y
 - ▶ Perform the same steps for z
 - ▶ Generate the instruction OP R_x, R_y, R_z
- For a 3AC copy instruction $x = y$,
 - ▶ If y is not in R_y according to the address descriptor, then generate instruction LD R_y, y'
 - ▶ Adjust the register descriptor for R_y to include x

Managing Register and Address Descriptors

- For an instruction LD R, x ,
 - ▶ Change the register descriptor for R so it holds only x
 - ▶ Change the address descriptor for x by adding register R as an additional location
- For instruction ST x, R , change the address descriptor for x to include its own memory location
- For an instruction such as ADD R_x, R_y, R_z , implementing a 3AC $x = y + z$,
 - ▶ Change the register descriptor for R_x so that it holds only x
 - ▶ Change the address descriptor for x so that its only location is R_x
 - ▶ The memory location for x is no longer in the address descriptor for x
 - ▶ Remove R_x from the address descriptor of any variable other than x
- For a copy instruction $x = y$,
 - ▶ Process the load from y into a register, if needed
 - ▶ Add x to the register descriptor for R_y
 - ▶ Change the address descriptor for x so that its only location is R_y

Usage of Registers

- Leave the computed result in a register for **as long as possible**
- Store the result only at the end of a BB or when the register is needed for another computation
 - ▶ A variable is live at a point if it is used (possibly in later BBs) later, requires global dataflow analysis
 - ▶ On exit from a BB, store only live variables which are not already in their memory locations (use address descriptors to identify)
 - ▶ If liveness information is not available, then assume that all variables are live at all times

Defining Function `getreg()`

Input 3AC $l: x = y \text{ op } z$

Output Returns registers to hold the value of x , y , and z

Assumption There is no global register allocation

getreg(): Choosing R_y for y

1. If y is in a register, then return the register containing y as R_y
 2. If y is not in a register, but there is an empty register available, then pick one such register as R_y
 3. If y is not in a register and there are no empty registers, then
 - ▶ Let R be a candidate register and suppose v is one of the variables stored in R
 - ▶ Heuristic for candidate selection can be based on farthest references or fewest next use
 - ▶ If the address descriptor for v says that v is somewhere else beside R , then choose R
 - ▶ If v is x , and x is not an operand of I (i.e., $x \neq z$), then choose R
 - ▶ If v is not used later, then choose R
 - ▶ Else, generate ST v, R (called a **register spill**)
 - ▶ R may hold several variables, so we need to repeat the previous steps for each variable
 - ▶ Compute the number of store instructions generated for R (i.e., score) for each variable
 - ▶ Pick the register with the lowest score
- Selecting R_z for z is similar

getreg(): Choosing R_x for x

- In addition to the previous checks, try the following,
 - ▶ A register that holds only x is always an acceptable choice for R_x
 - ▶ If y is not used after instruction I , and R_y holds only y after being loaded, then R_y can also be used for R_x
 - ▶ Perform similar checks with R_z if required
- If I is a copy instruction, then always choose R_y

Code Generation Example

3AC	Generated Code	Address			Address Descriptor						
		R ₁	R ₂	R ₃	a	b	c	d	t	u	v
					a	b	c	d			
t = a - b	LD R ₁ , a LD R ₂ , b SUB R ₂ , R ₁ , R ₂										
u = a - c	LD R ₃ , c SUB R ₁ , R ₁ , R ₃										
v = t + u	ADD R ₃ , R ₂ , R ₁										
...

in memory, live at the end of the BB

temporaries, not live at the end of the BB

R₂ is reused because there is no next use of b

Code Generation Example

3AC	Generated Code	Address			Address Descriptor						
		R ₁	R ₂	R ₃	a	b	c	d	t	u	v
...
		u	t	v	a	b	c	d	R ₂	R ₁	R ₃
a = d	LD R ₂ , d										
		u	a,d	v	R ₂	b	c	d,R ₂		R ₁	R ₃
d = v + u	ADD R ₁ , R ₃ , R ₁										
		d	a	v	R ₂	b	c	R ₁			R ₃
exit	ST a, R ₂ ST d, R ₁										
		d	a	v	R ₂	b	c	R ₁			R ₃

Code Sequences for Indexed and Pointer Assignments

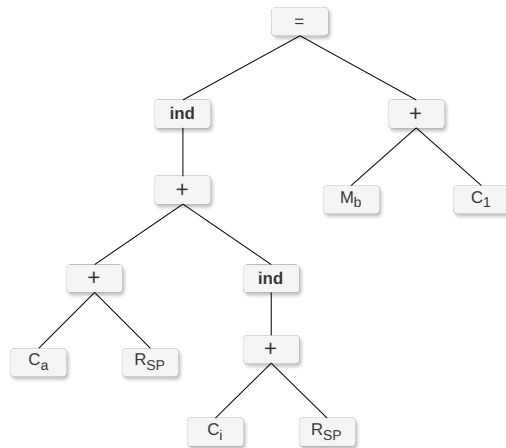
3AC	i in Register R_i	i in Memory M_i	i in Stack
$a = b[i]$	MOV $b(R_i)$, R	MOV M_i , R MOV $b(R)$, R	MOV $S_i(A)$, R MOV $b(R)$, R
$a[i] = b$	MOV b , $a(R_i)$	MOV M_i , R MOV b , $a(R)$	MOV $S_i(A)$, R MOV b , $a(R)$

3AC	p in Register R_p	p in Memory M_p	p in Stack
$a = *p$	MOV $*R_p$, a	MOV M_p , R MOV $*R$, R	MOV $S_p(A)$, R MOV $*R$, R
$*p = b$	MOV a , $*R_p$	MOV M_p , R MOV a , $*R$	MOV a , R MOV R , $*S_p(A)$

Code Generation with Tree Rewriting

Tree Representation

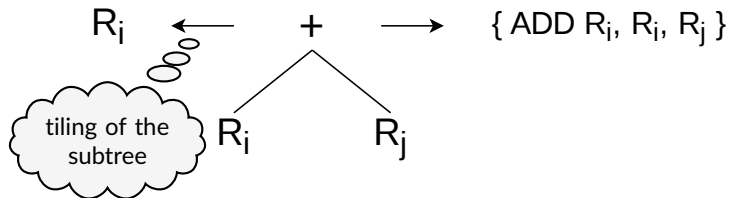
- Consider the statement $a[i] = b + 1$
 - ▶ Assume b is in memory location M_b
 - ▶ Array of chars a is a local variable and is stored on the stack
 - ▶ SP points to the beginning of the current activation record
 - ▶ Addresses of locals a and i are given as constant offsets C_a and C_i from the SP



Operator **ind** denotes indirection

Tree Rewriting

- Target code can be generated by applying a sequence of tree-rewriting rules to reduce the input tree to a single node
- Each rewrite rule is of the form $\text{replacement} \leftarrow \text{template} \{ \text{action} \}$, where *replacement* is a single node, *template* is a tree, and *action* is a code fragment like in a SDT
- A set of tree rewriting rules is called a tree-translation scheme

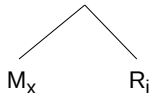


Tree Rewriting Rules

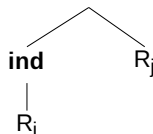
1 $R_i \leftarrow C_a$ { LD $R_i, \#a$ }

2 $R_i \leftarrow M_x$ { LD R_i, x }

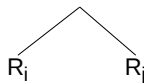
3 $M \leftarrow =$ { ST x, R_i }



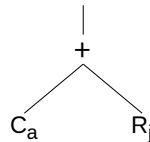
4 $M \leftarrow =$ { ST $*R_i, R_i$ }



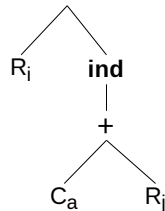
7 $R_i \leftarrow +$ { ADD R_i, R_i, R_j }



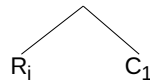
5 $R_i \leftarrow \text{ind}$ { LD $R_i, a(R_j)$ }



6 $R_i \leftarrow +$ { LD $R_i, a(R_j)$ }



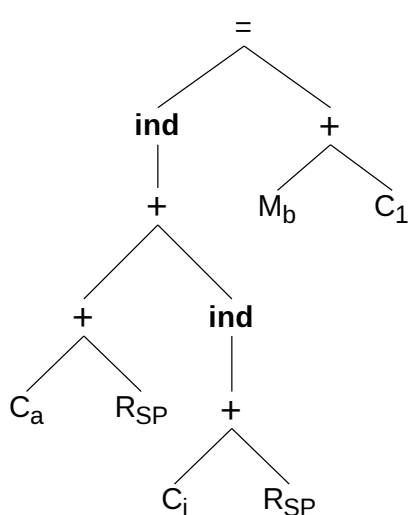
8 $R_i \leftarrow +$ { INC R_i }



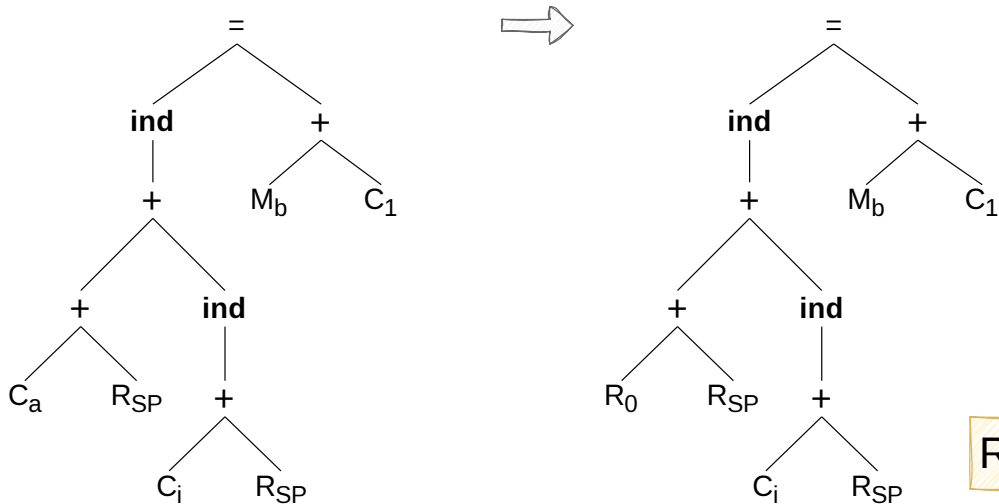
Code Generation by Tiling an Input Tree

- High-level steps in a tree-translation scheme
 - ▶ Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees
 - ▶ Tiling implies reducing a subtree with the *replacement* node
 - ▶ If a template matches, replace the matching subtree with the replacement node of the rule
 - ▶ Execute the action associated with the rule
 - ▶ If the action contains a sequence of instructions, the instructions are emitted
 - ▶ Repeat the above steps until the tree is reduced to a single node, or until no more templates match
- Output of the tree-translation scheme is the instruction sequence generated as the input tree is reduced to a single node

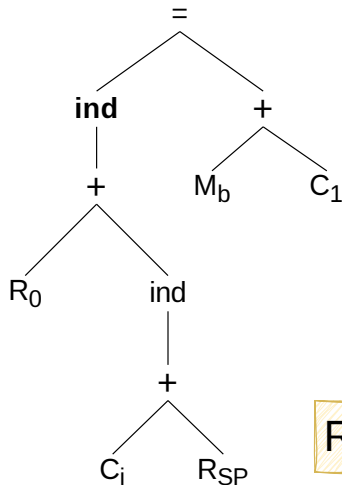
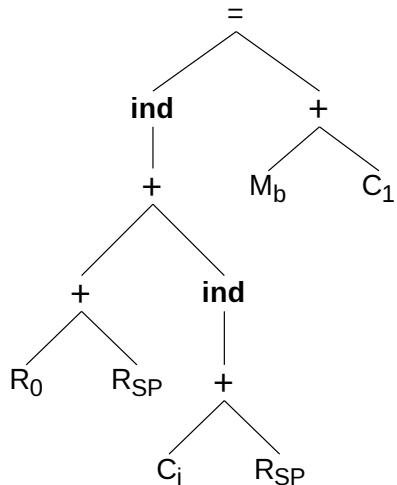
Example of Code Generation with Tree Rewriting



Example of Code Generation with Tree Rewriting

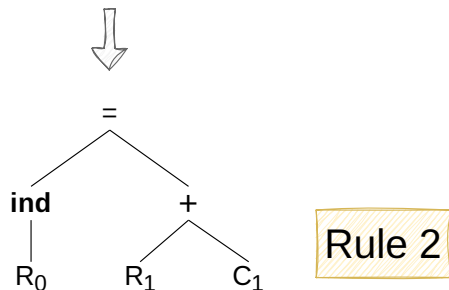
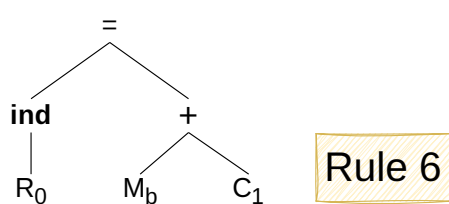
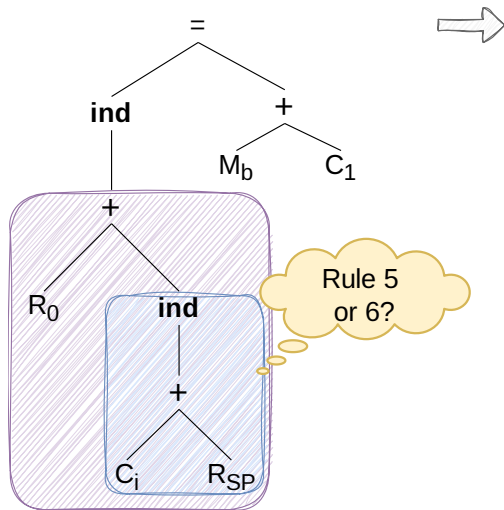


Example of Code Generation with Tree Rewriting

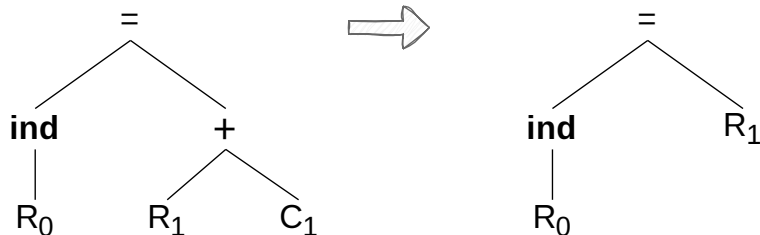


Rule 7

Example of Code Generation with Tree Rewriting



Example of Code Generation with Tree Rewriting



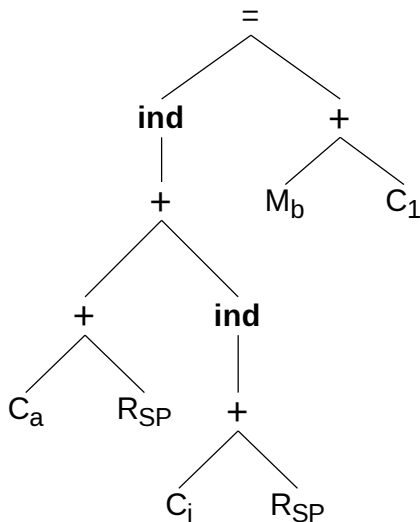
Rule 8



M

Rule 4

Example of Code Generation with Tree Rewriting



```
LD R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD R1, b
INC R1
ST *R0, R1
```

Considerations during Tree Reduction

- (i) Performance of tree matching impacts the efficiency of code generation at compile time
- (ii) Multiple templates may match during code generation
- (iii) Different match sequences of templates will lead to different codes being generated which can impact efficiency
- (iv) If no template matches, then the code-generation process blocks
 - ▶ Assume each operator in the intermediate code can be implemented by one or more target-machine instructions
 - ▶ Assume there are sufficient registers to compute each tree node by itself

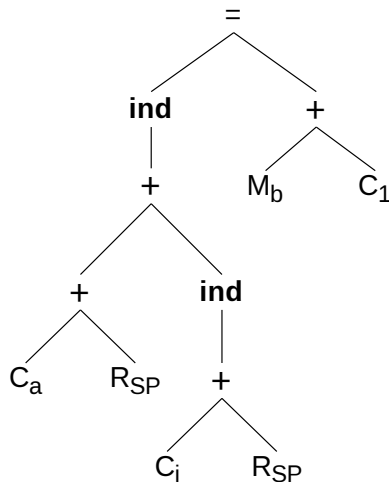
Pattern Matching with LR Parsing

● Idea

- ▶ Convert the input tree to a string using prefix (or postfix) form for comparison
- ▶ Use a parsing mechanism for pattern matching
- ▶ Come up with a syntax-directed translation (SDT) as an alternate for tree rewriting rules

Prefix representation =

ind + + C_aR_{SP} ind + C_iR_{SP} + M_bC₁





SDT for Tree Rewriting

- Terminal m represents a memory location
- Terminal sp represents register SP
- Terminal c represents a constant
- Design a code generator for a different architecture by rewriting the grammar
- Resolve conflicts using estimates of instruction costs, favoring larger reductions, and favoring shifts over reductions

Production	Semantic Action
$R_i \rightarrow c_a$	LD R_i , $\#a$
$R_i \rightarrow M_x$	LD R_i , x
$M \rightarrow = M_x R_i$	ST x , R_i
$M \rightarrow = \mathbf{ind} R_i R_j$	ST $*R_i$, R_j
$R_i \rightarrow \mathbf{ind} + c_a R_j$	LD R_i , $a(R_j)$
$R_i \rightarrow +R_i \mathbf{ind} + c_a R_j$	ADD R_i , R_i , $a(R_j)$
$R_i \rightarrow +R_i R_j$	ADD R_i , R_i , R_j
$R_i \rightarrow +R_i c_1$	INC R_i
$R \rightarrow \mathbf{sp}$	
$M \rightarrow \mathbf{m}$	

References

-  A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 8.1–8.6, 8.9, 8.10, 2nd edition, Pearson Education.
-  K. Cooper and L. Torczon. Engineering a Compiler. Chapter 11, 2nd edition, Morgan Kaufmann.