CS251: Introduction to Language Processing

Semantic Analysis and Intermediate Code Generation

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

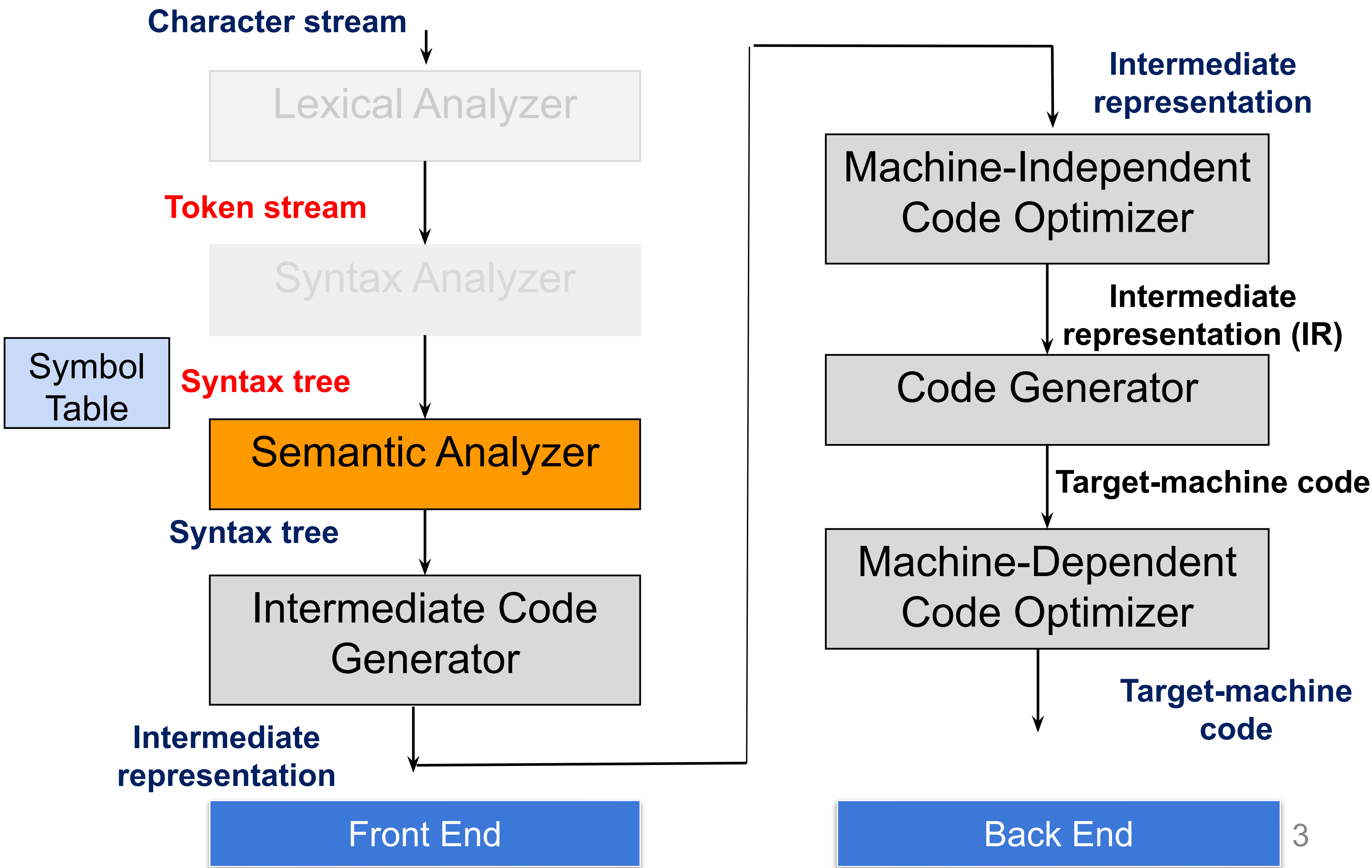
vishwesh@iitbhilai.ac.in

2023-24 Sem-1

Acknowledgement

- References for today's slides
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*
  - *IIT Madras (Prof. Rupesh Nasre)*
    - *http://www.cse.iitm.ac.in/~rupesh/teaching/compiler/aug15/schedule/4-sdt.pdf*
  - *Course textbook*
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

Compiler Design



Recap

- Express semantics:
  - Using attributed grammar
  - Synthesized attributes
  - Inherited attributes
  - Order of evaluation
    - Dependency graph
- S-attributed definition
- L-attributed definition

Syntax Directed Translations

Syntax Directed Translations

- Complementary notations to SDD
- Syntax Directed Translation scheme (SDT):
  - Context free grammar with program fragments embedded within production bodies
  - Program fragments: semantics

SDD for Calculator

Sr. No.	Production	Semantic Rules
1	$E' \rightarrow E$ \$	$E'.val = E.val$
2	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	...
4	$T \rightarrow T_1 * F$	...
5	$T \rightarrow F$	...
6	$F \rightarrow (E)$	...
7	$F \rightarrow$ *digit*	$F.val =$ *digit*.lexval

SDT for Calculator

Sr. No.	Production	Semantic Rules
1	$E' \rightarrow E\ \$$	**print(E.val)**
2	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3	$E \rightarrow T$	...
4	$T \rightarrow T_1 * F$	...
5	$T \rightarrow F$	...
6	$F \rightarrow (E)$	...
7	$F \rightarrow \textit{digit}$	$F.val = \textit{digit}.lexval$

SDT for Calculator

Postfix SDT

$E' \to E\ \$$	**{ print(E.val); }**
$E \to E_1 + T$	{ $E.val = E_1.val + T.val;$ }
$E \to T$	...
$T \to T_1 * F$	...
$T \to F$	...
$F \to (E)$	...
$F \to \mathit{digit}$	{ $F.val = \mathit{digit}.lexval;$ }

- SDTs with all the actions at the right ends of the production bodies are called *postfix SDTs*.
- Can be implemented during LR parsing by executing actions when reductions occur.
- The attribute values can be put on a stack and can be retrieved.

Actions within Productions

- Actions may be placed at any position within production body.

- For production $B \longrightarrow X \{action\} Y$, action is performed
  - as soon as X appears on top of the parsing stack in bottom-up parsing.
  - just before expanding Y in top-down parsing if Y is a non-terminal.
  - just before we check for Y on the input in top-down parsing if Y is a terminal.
- SDTs that can be implemented during parsing are
  - Postfix SDTs (S-attributed definitions) SDTs
  - implementing L-attributed definitions

SDT Example

$D \rightarrow T$ $\{L.in = T.type\}$ L

$T \rightarrow int$ $\{T.type = integer\}$
$T \rightarrow real$ $\{T.type = real\}$

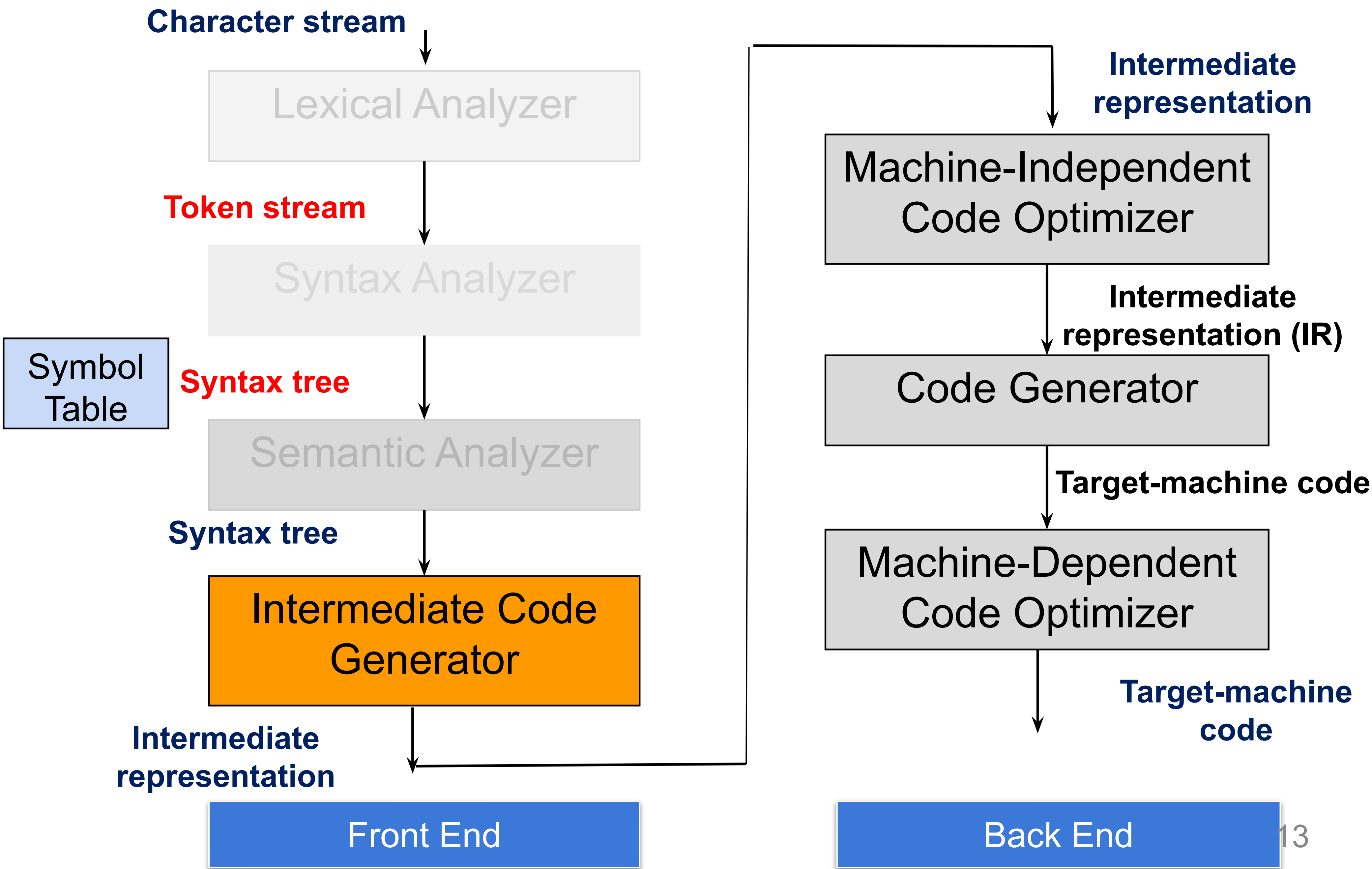
$L \longrightarrow \{L_1.in = L.in\}\ L_1, id$
$\{addtype(id.entry, L_{in})\}$

$L \longrightarrow id\ \{addtype(id.entry, L_{in})\}$

Quick Summary

- Express semantics:
  - Syntax Directed Definition (SDD)
  - Syntax Directed Translation (SDT)

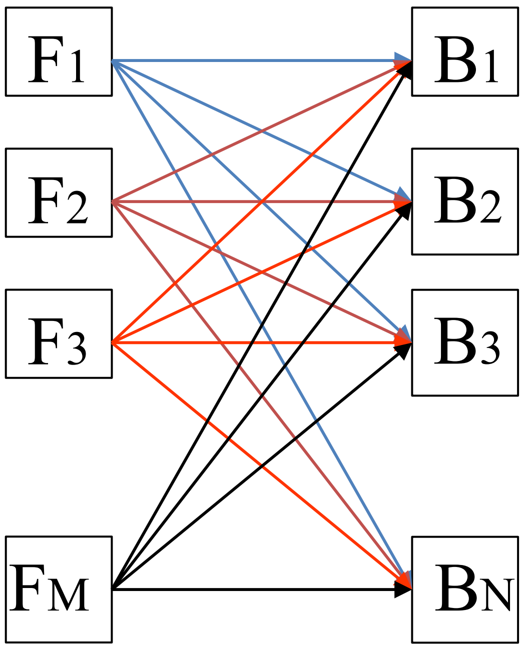
Next…


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

Compiler

Why Intermediate Code Generation?

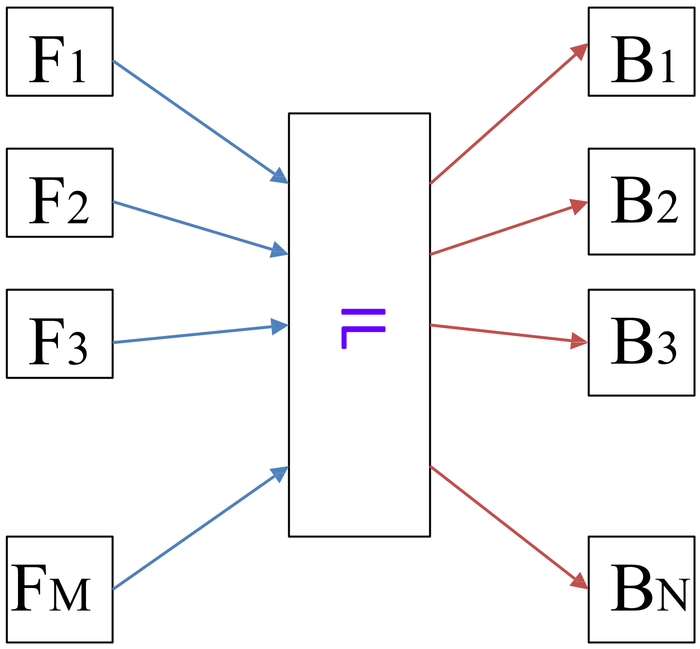
- M*N vs. M+N problem
  - Compilers are required for all the languages and all the machines
  - For M languages and N machines: Develop M*N compilers?
  - M *N optimizers, and M *N code generators
  - Repetition of work



Requires M*N compilers

Why Intermediate Code Generation?

Intermediate Language



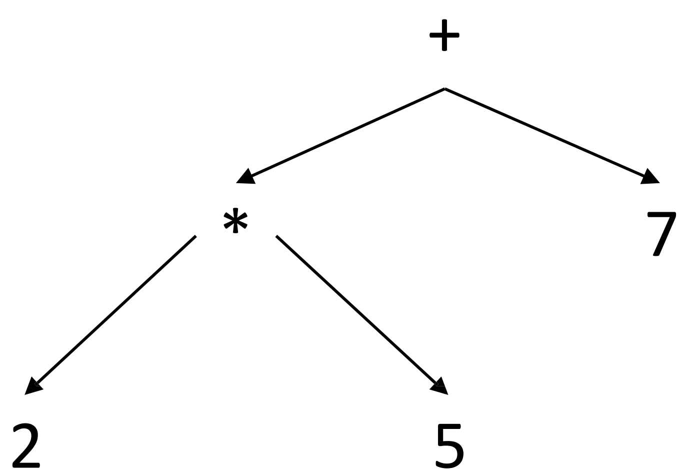
Requires M front ends
And N back ends

- M front ends, N back ends
- Facilitates machine independent code optimizers

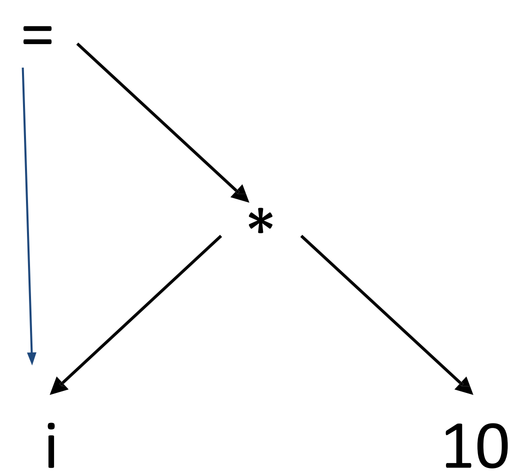
Intermediate Codes

- Maintains some high-level information
- Easy to generate
- Easy to translate to machine code
- Generated code should be based on application
- Should not contain machine dependent information
  - registers, addresses, stack..etc.

Intermediate Code Representations



Abstract Syntax Tree



DAG

Intermediate Code Representations

- Three address code (TAC)
  - Instructions are very simple
  - Maximum three addresses in an instruction
  - LHS is the target
  - RHS has at most two sources and one operator
  - address:
    - Name: programmer defined
    - Constant
    - Temporary variables

```
t = a + 5
p = t * b
q = p  - c
p = q
p = -e
q = p + q
```

Intermediate Code Representations

- Static single Assignment (SSA)
  - A variable is assigned exactly once

```
p = a + b
q = p  - c
p = q  * d
p = e - p
q = p + q
```

Three-address code

```
p1 = a + b
q1 = p1  - c
p2 = q1  * d
p3 = e - p2
q2 = p3 + q1
```

Static-single Assignment

We will use 3-address code in this course

Implementations of TAC

op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Quadruples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)

Triples

Three address code

- Assignment
  - x = y op z
  - x = op y
  - x = y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = &y
  - x = *y
  - *x = y

Intermediate Code Generation

- Expressions
- Statements
  - Simple statements
  - Conditional statements
  - Control flow statements
    - if, if-else, while.
  - Declarations
  - Arrays
  - Functions

Intermediate Code Generation

- Expressions
- Statements
  - Simple statements
  - Conditional statements
  - Control flow statements
    - if, If-else, while
  - Declarations
  - Arrays
  - Functions

Syntax directed translation of expression into 3-address code

Expression: a + b * c

Three-address code:

t1 = b * c

t2 = a + t1

Syntax directed translation of expression into 3-address code

- newtmp() -> creates a new temporary variable
- **gen(...):** produce sequence of three address statements
  - The statements themselves are kept in some data structure, e.g. list
  - SDD operations described using pseudo code

Syntax directed translation of expression into 3-address code

- Attribute:
  - ***E.place***, a name that will hold the value of E

Syntax directed translation of expression into 3-address code

$E \longrightarrow E_1 + E_2$

E.place:= newtmp()

gen(E.place := E_1.place + E_2.place)

Syntax directed translation of expression into 3-address code

$E \longrightarrow E_1 * E_2$

E.place:= newtmp()

gen(E.place := E_1.place * E_2.place)

Syntax directed translation of expression into 3-address code

$S \rightarrow id := E$

S.code := gen(id.place:= E.place)

Syntax directed translation of expression …

$E \rightarrow -E_1$

E.place := newtmp()

gen(E.place := - E_1.place)

$E \rightarrow (E_1)$

E.place := E_1.place

$E \rightarrow id$

E.place := id.place

Exercise

Generate the Intermediate representation for

$a = b * -c + b * c$

Exercise

Expression: $a = b * -c + b * c$

Generated code:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = b * c$

$t_4 = t_2 + t_3$

$a = t_4$

Boolean Expressions

E →

|	E relop E
|	E or E
|	E and E
|	not E
|	true
|	false

Numerical representation

- relational expression a < b is equivalent to if a < b then 1 else 0

1.if a < b goto 4.
2.t = 0
3. goto 5
4. t = 1
5.

Syntax directed translation of boolean expressions

E $\longrightarrow$ E1 < E2

E.place := newtmp
gen(if E1.place < E2.place goto nextstat+3)
gen(E.place = 0)
gen(goto nextstat+2)
gen(E.place = 1)

"nextstat" is a global variable; a pointer to the statement to be emitted. emit also updates the nextstat as a side-effect.

Syntax directed translation of boolean expressions

$E \longrightarrow E_1$ or E_2

E.place := newtmp

gen(E.place ':=' E_1.place 'or' E_2.place)

$E \longrightarrow E_1$ and E_2

E.place:= newtmp

gen(E.place ':=' E_1.place 'and' E_2.place)

$E \longrightarrow$ not E_1

E.place := newtmp

gen(E.place ':=' 'not' E_1.place)

Syntax directed translation of boolean expressions

$E \longrightarrow$ true

E.place := newtmp

gen(E.place = '1')

$E \longrightarrow$ false

E.place := newtmp

gen(E.place = '0')