CSL302: Compiler Design

Semantic Analysis

Vishwesh Jatala

Assistant Professor

Department of CSE

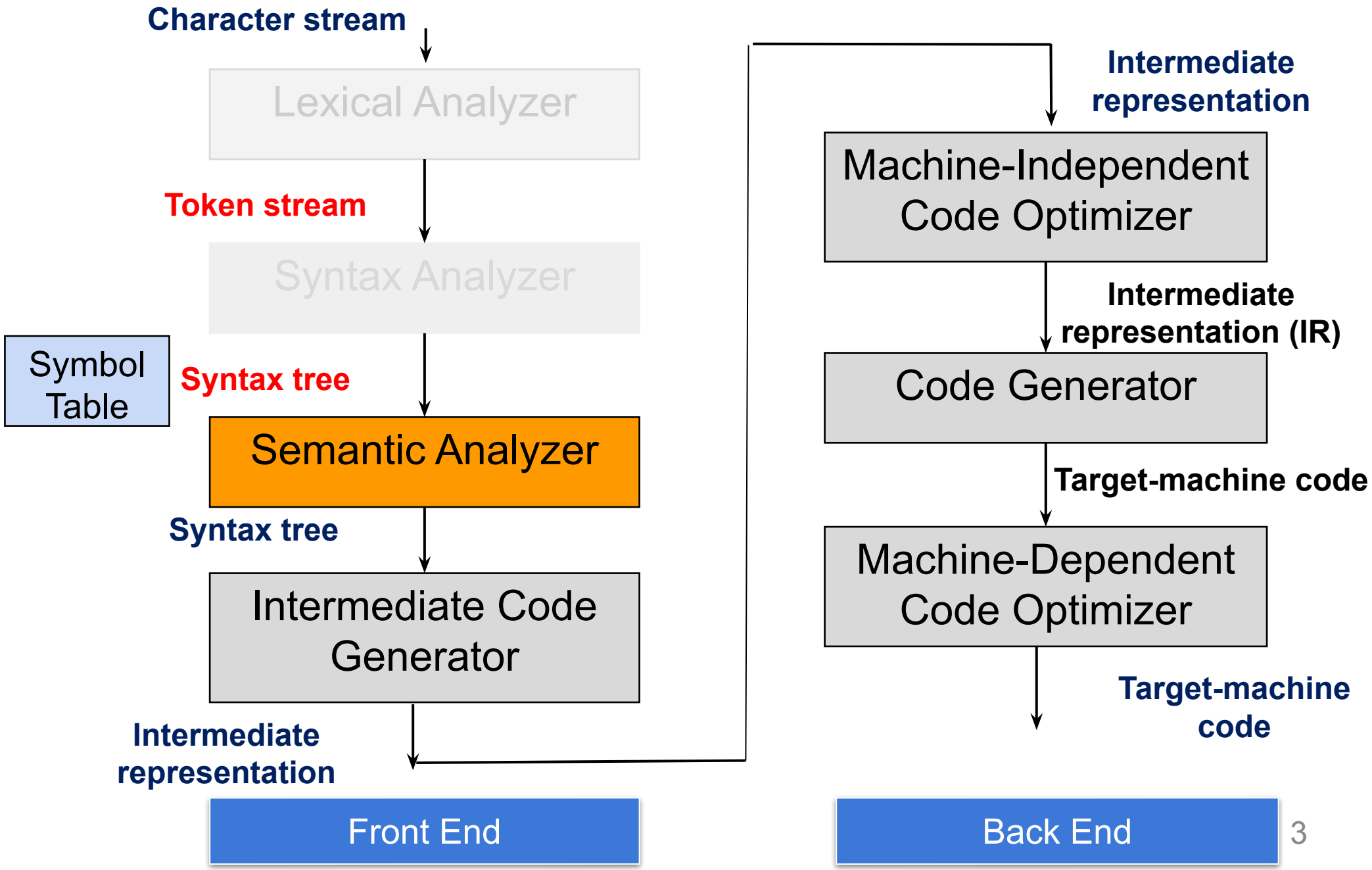
Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

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

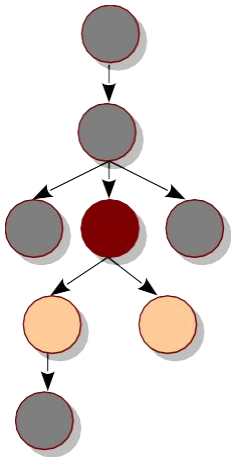


Recap

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

Attributes …

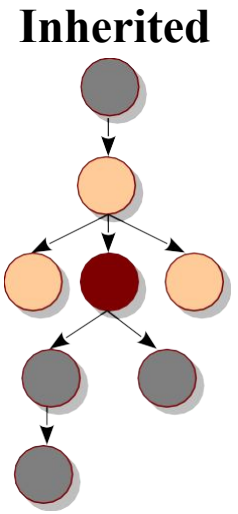
- Attributes fall into two classes: *Synthesized* and *Inherited*
- Value of a synthesized attribute is computed from the values of children nodes
  - Attribute value for LHS of a rule comes from attributes of RHS



Synthesized

Attributes …

- Value of an inherited attribute is computed from the sibling and parent nodes
  - Attribute value for a symbol on RHS of a rule comes from attributes of LHS and RHS symbols



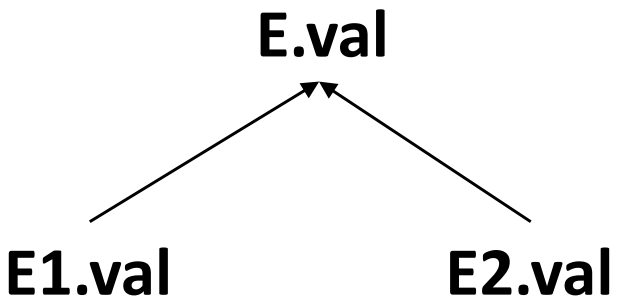
Synthesized Attributes

- A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
  - A topological evaluation order is well-defined.
  - Any bottom-up order of the parse tree nodes.

Example

- Consider the following production is used in a parse tree
  - $E \rightarrow E_1 + E_2$ $\quad\quad$ $E.val = E_1.val + E_2.val$

  we create a dependency graph



Issues with S-attributed SDD

- It is too strict!
- There exist reasonable non-cyclic orders that it disallows.
  - If a non-terminal uses attributes of its parent only (no sibling attributes)
  - If a non-terminal uses attributes of its left-siblings only (and not of right siblings).
- The rules may use information "from above" and "from left".

L attributed definitions

- Each attribute must be either
  - synthesized, or
  - inherited, but with restriction. For production A → X1 X2 … Xn with inherited attributed Xi.a computed by an action; then the rule may use only
    - inherited attributes of A.
    - either inherited or synthesized attributes of X1, X2, …, Xi-1.
    - inherited or synthesized attributes of Xi with no cyclic dependence.
- L is for left-to-right.

L attributed definitions

$A \rightarrow LM$ $L.i = f_1(A.i)$ $M.i = f_2(L.s)$ $A.s = f_3(M.s)$

$A \rightarrow QR$ $R.i = f4(A.i)$ $Q.i = f5(R.s)$ $A.s = f6(Q.s)$

L attributed definitions

- We can adapt the grammar to compute the L-attributes during LR parsing.

  More details, refer to textbook

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	$F \rightarrow$ *digit*	F.val = *digit*.lexval

SDT for Calculator

Postfix SDT

E' → E $	**{ print(E.val); }**
E → E_1 + T	{ E.val = E_1.val + T.val; }
E → T	...
T → T_1 * F	...
T → F	...
F → (E)	...
F → *digit*	{ F.val = *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 → T {L.in = T.type} L

T → int {T.type = integer}
T → real {T.type = real}

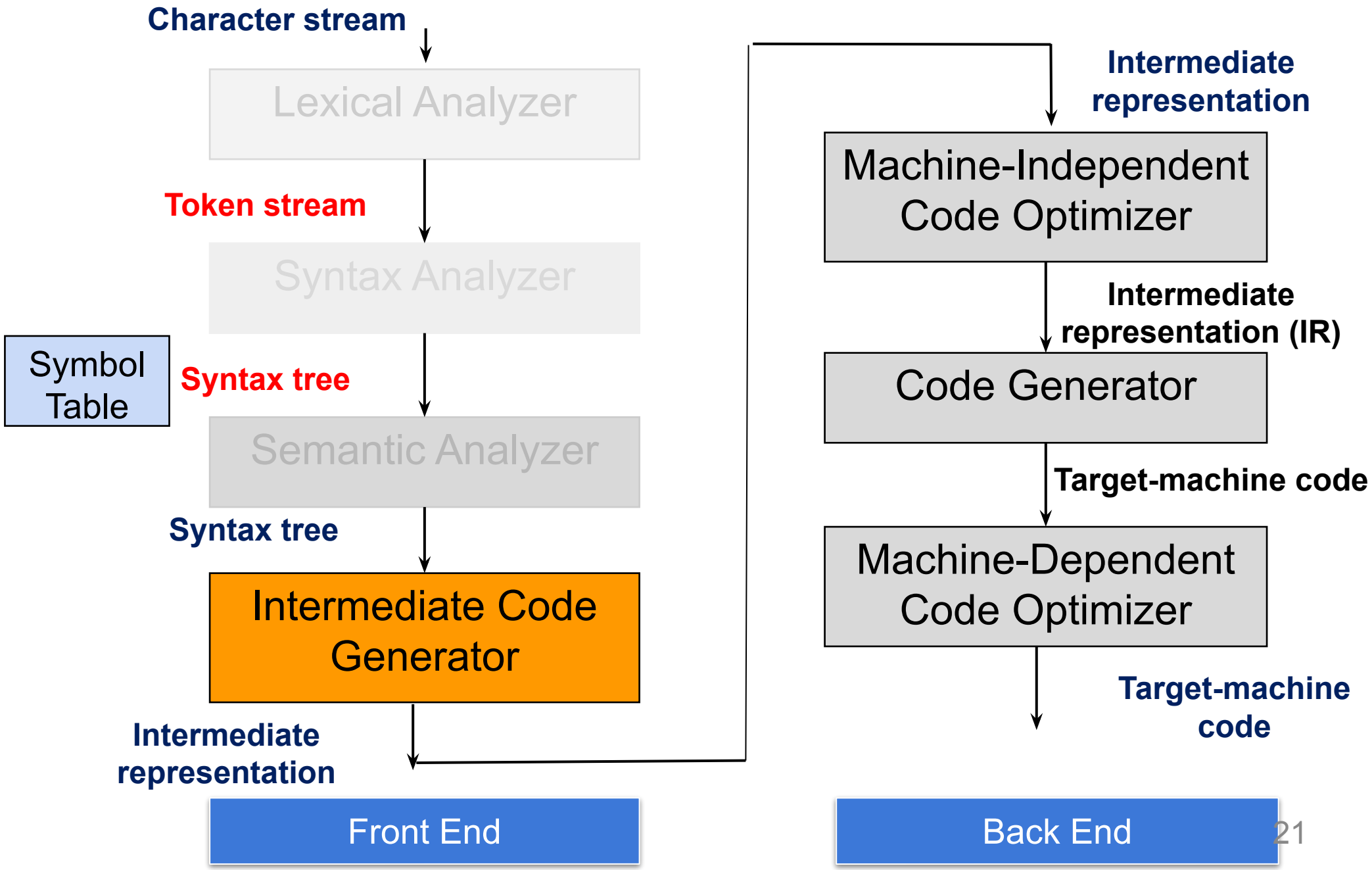
L → {L_1.in =L.in} L_1,id
{addtype(id.entry,L_{in})}

L → id {addtype(id.entry,L_{in})}

Quick Summary

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

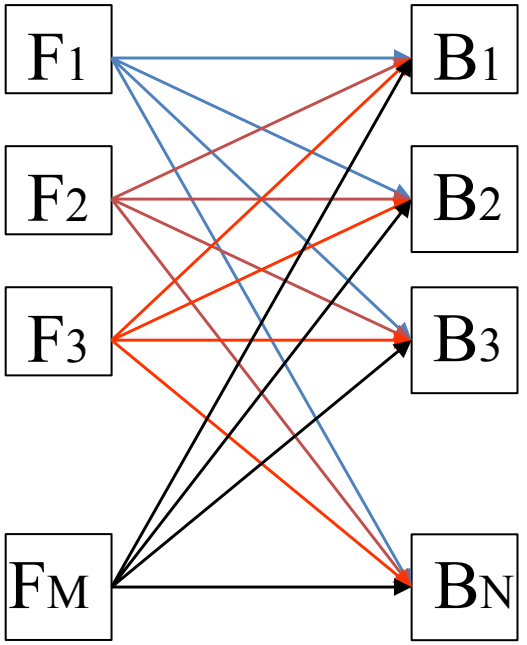
Next…


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

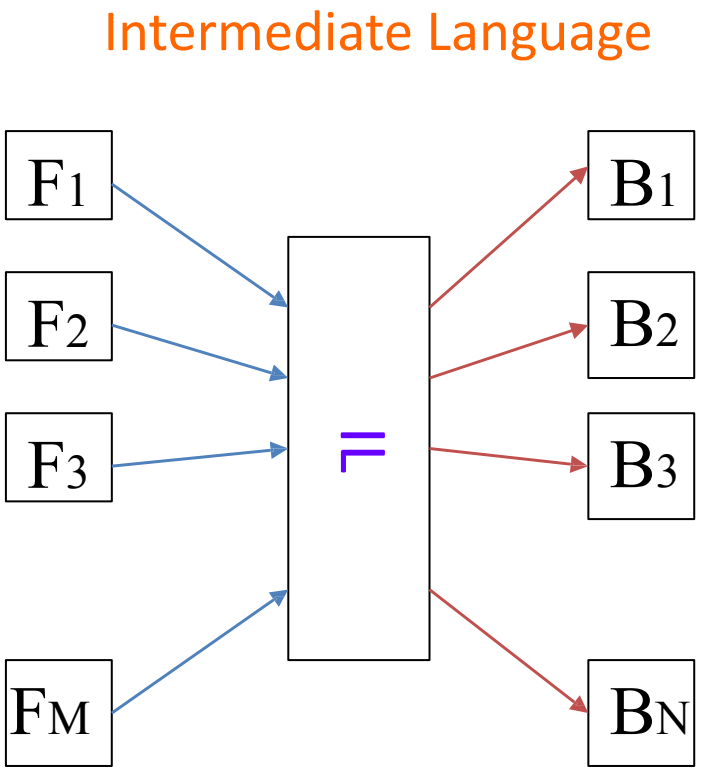
Why Intermediate Code Generation?

- 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?



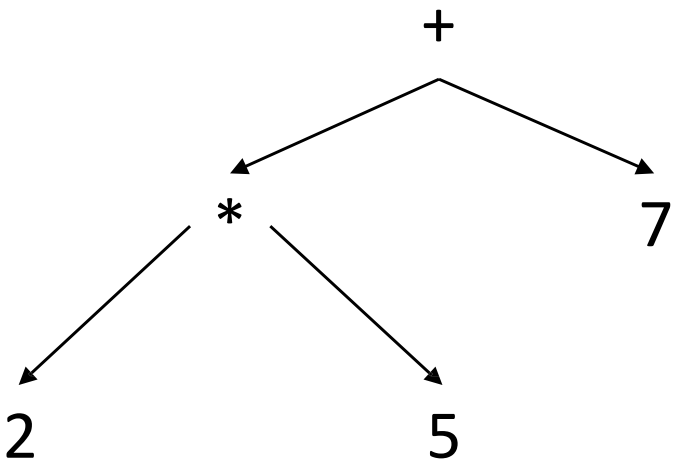
Requires M front ends
And N back ends

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

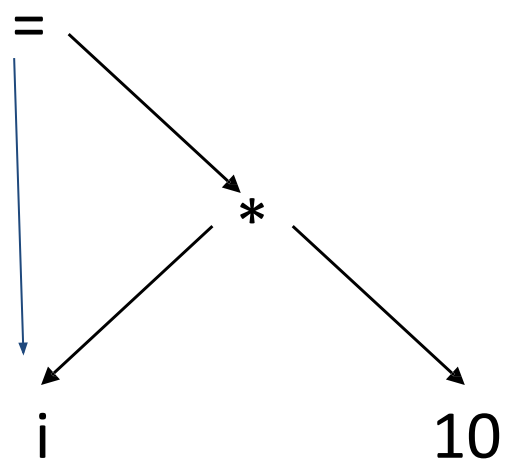
Intermediate Codes

- 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

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

Summary

- Express semantics:
  - S-attributed and L-attributed grammar
-
- Intermediate representation