Compiler Construction

BITS Pilani
Pilani Campus

Vinti Agarwal
March 2021

CS F363, Compiler Construction

Lecture topics: Recap

innovate achieve lead

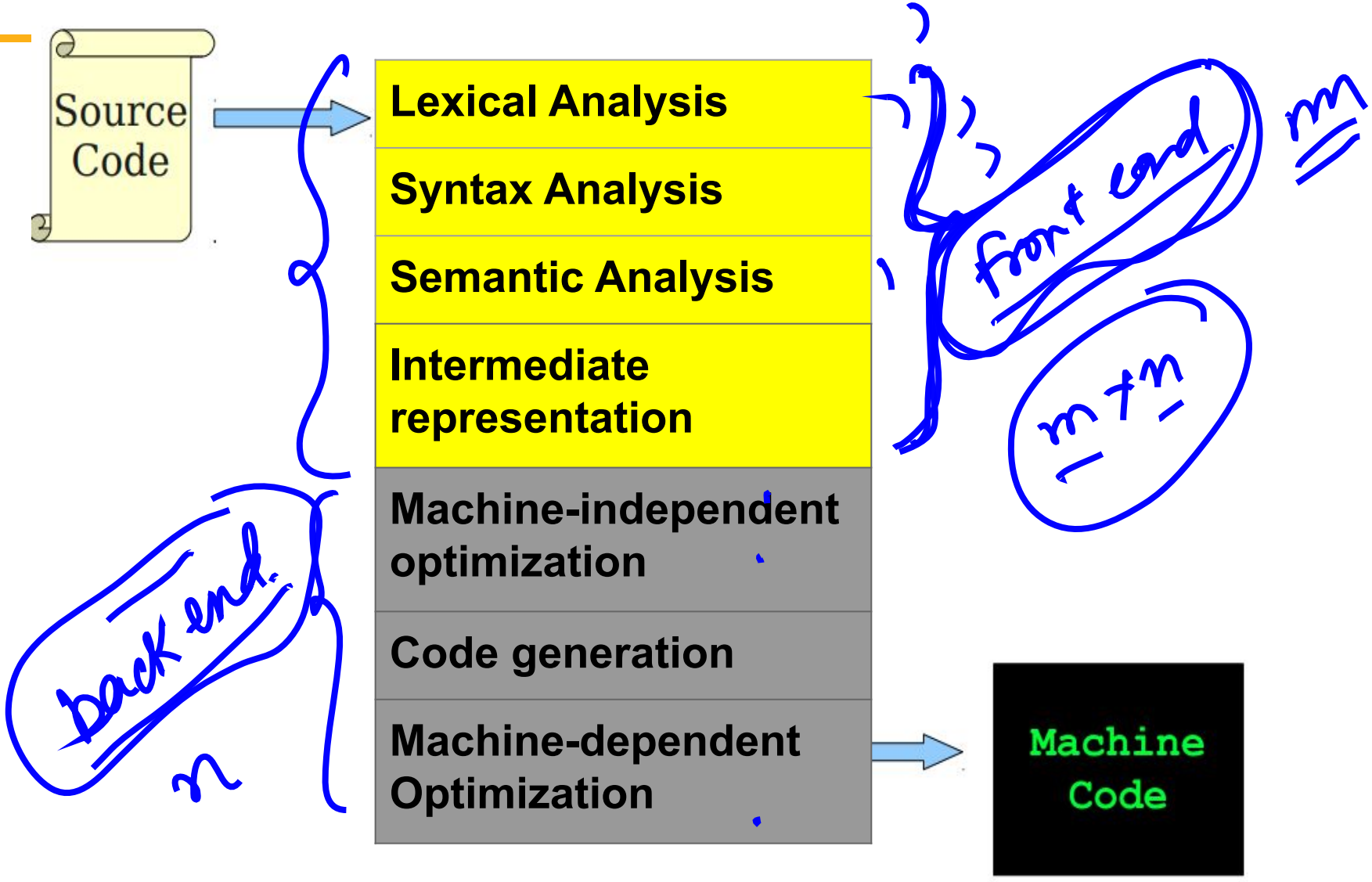
BITS Pilani

Pilani Campus

Some content of the slides are based on:

https://web.stanford.edu/courses/soe-ycscs1-compilers

Where we are ?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate representation
Machine-independent optimization
Code generation
Machine-dependent Optimization
Machine Code
front end
back end
m
n
m + n

Lexical Analysis?

- classify program substrings according to role
- communicate tokens to the parser (syntax analyzer)

Token class :

identifier- string of letters/digits

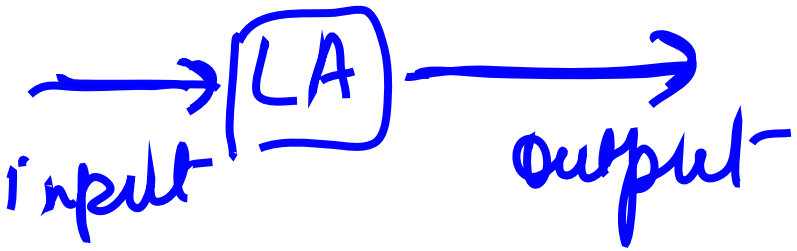
integer- non empty string of digits

keyword : else if begin

whitespace: non empty sequence of blanks, tabs, newline etc

Lexical Analysis?

```
if (i==j)
  z=1;
else
  z=0;
```



Lexical Analysis?

```
if (i==j)
  z=1;
else
  z=0;
```

```
\tif (i==j)\n\t\tz=1;\n\telse\n\t\tz=0;
```

Lexical specification

What set of strings is in token class?

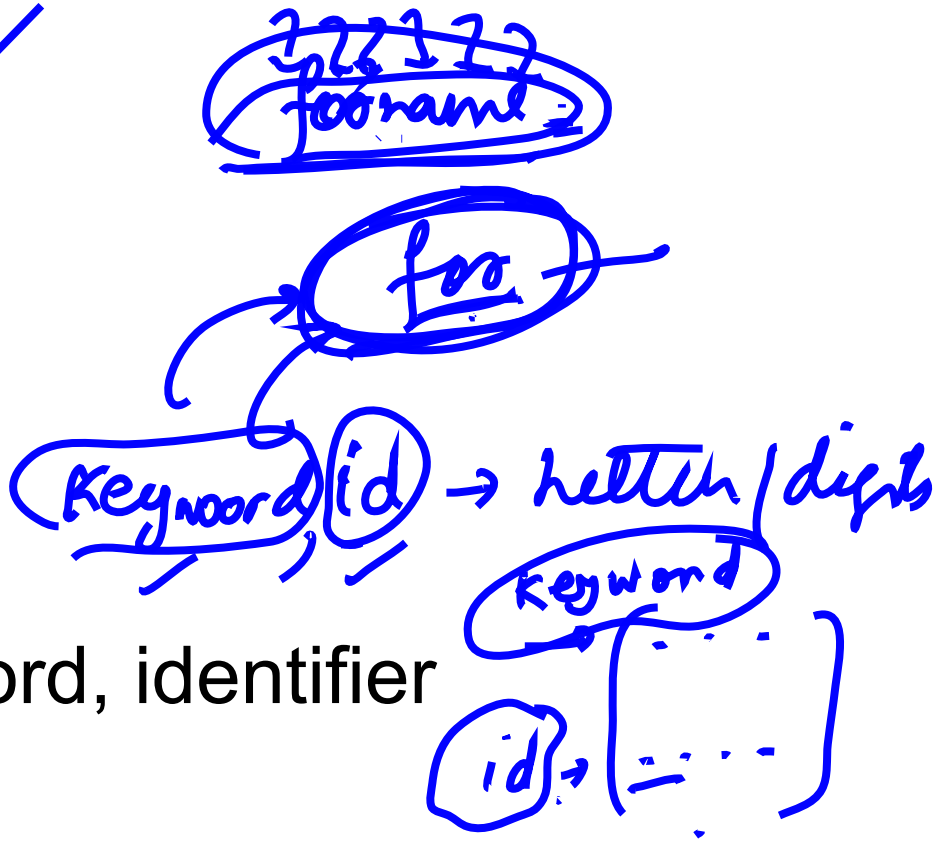
- use regular language

How much input is used?

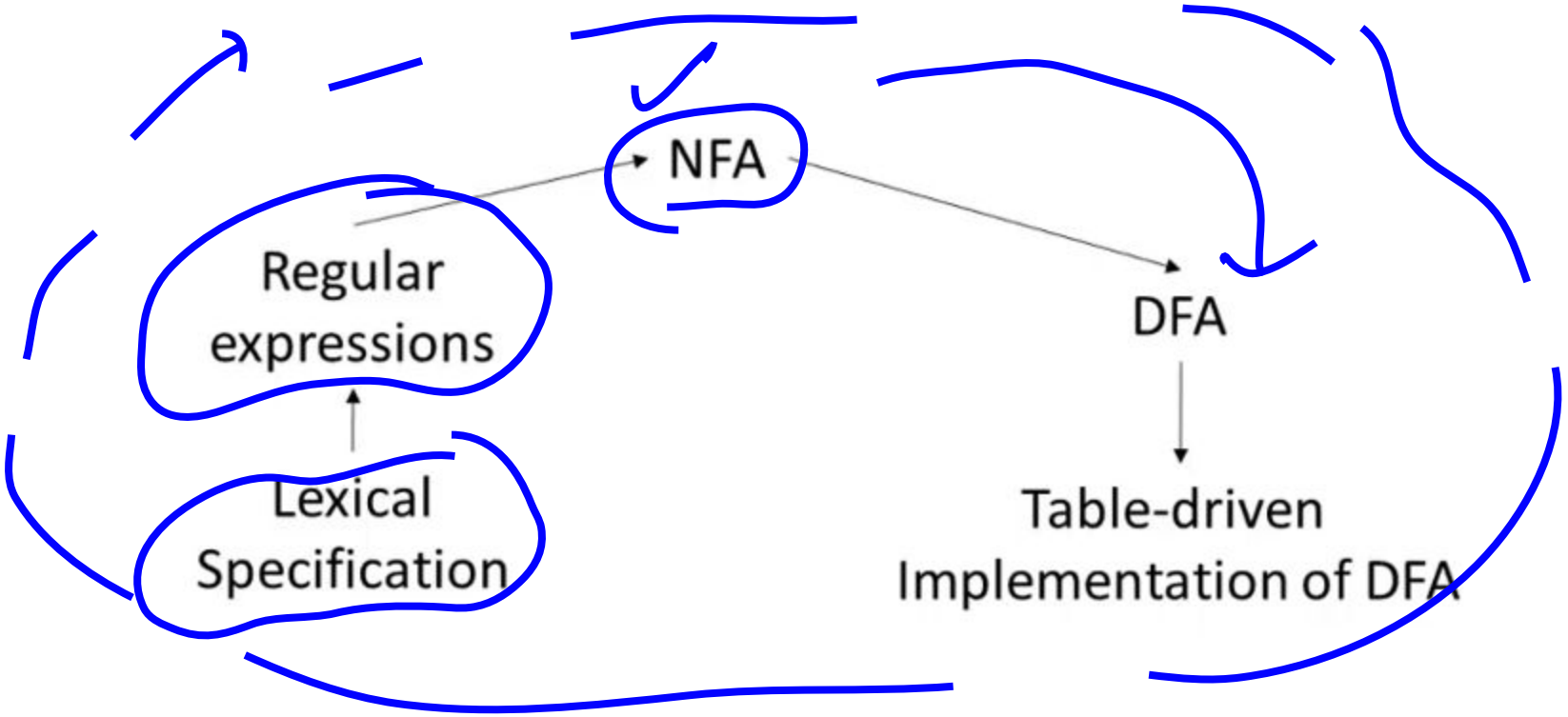
- Maximal munch

Which token is used?

- priority ordering: keyword, identifier



Lexical analysis

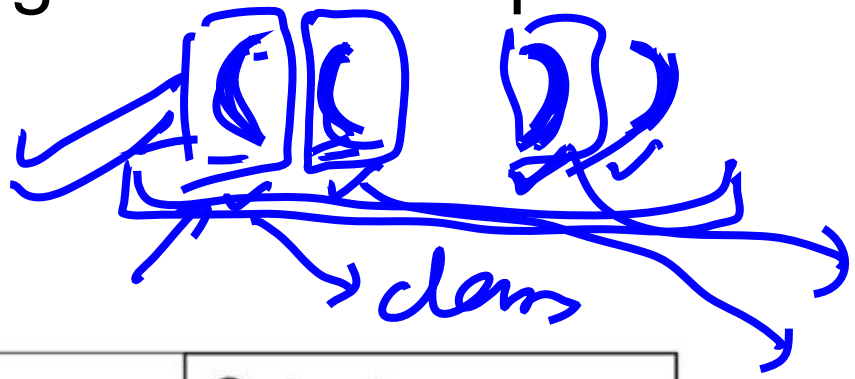


words → are I home a am at

Parsing/Syntax analysis

Some important language can't be expressed using finite automata.

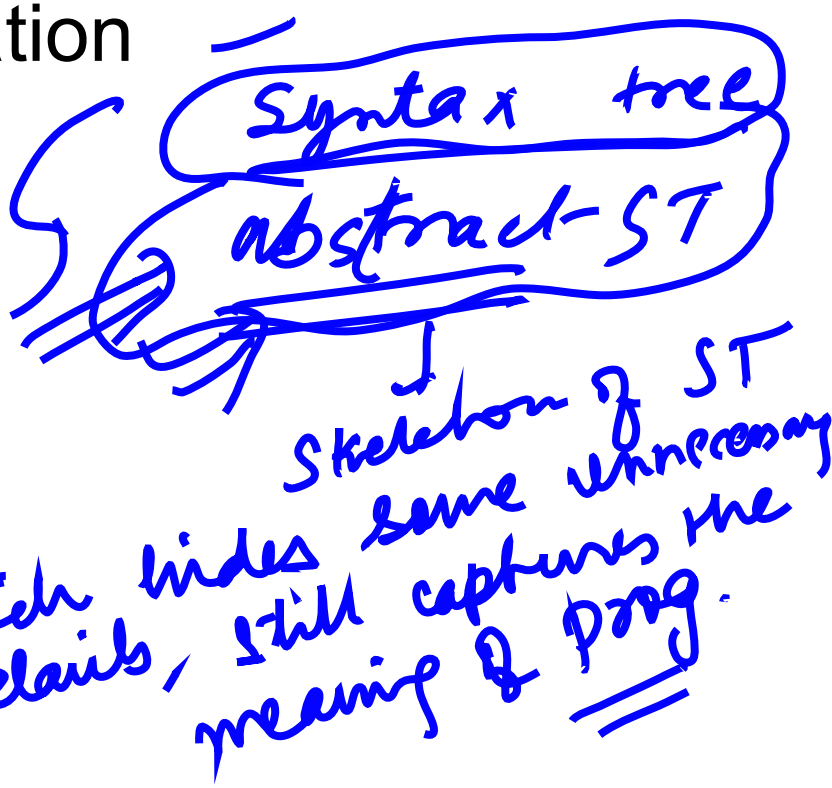
- e.g. $(^i)^i$ i>=0
-



Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

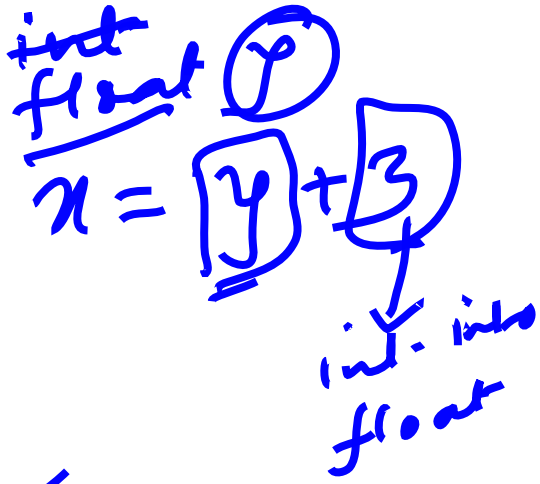
Parsing/Syntax analysis

- context free grammar
- left most/right most derivation
- ambiguity
- Abstract syntax tree
- LL/SR/SLR



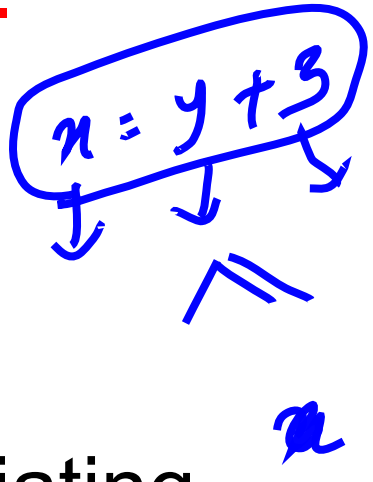
Semantic analysis

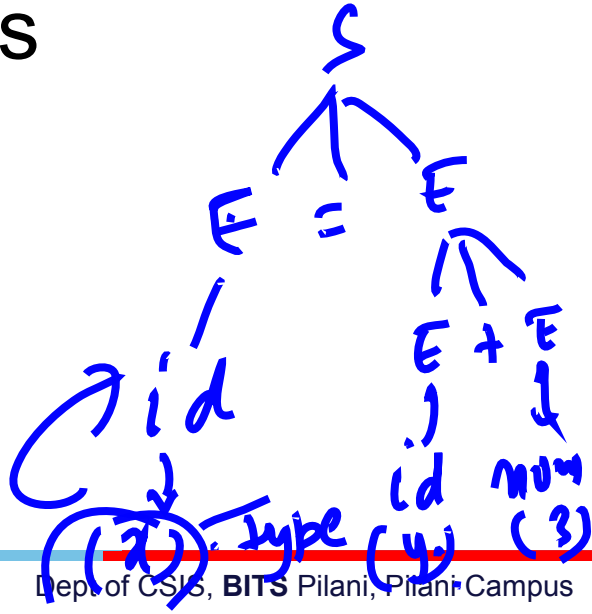
- parsing cannot catch some errors
- some language constructs are not context free
- checks of many kinds
  - All identifiers are declared
  - types
  - inheritance
  - class defined once
  - method in a class defined once
  - reserve identifier not misused



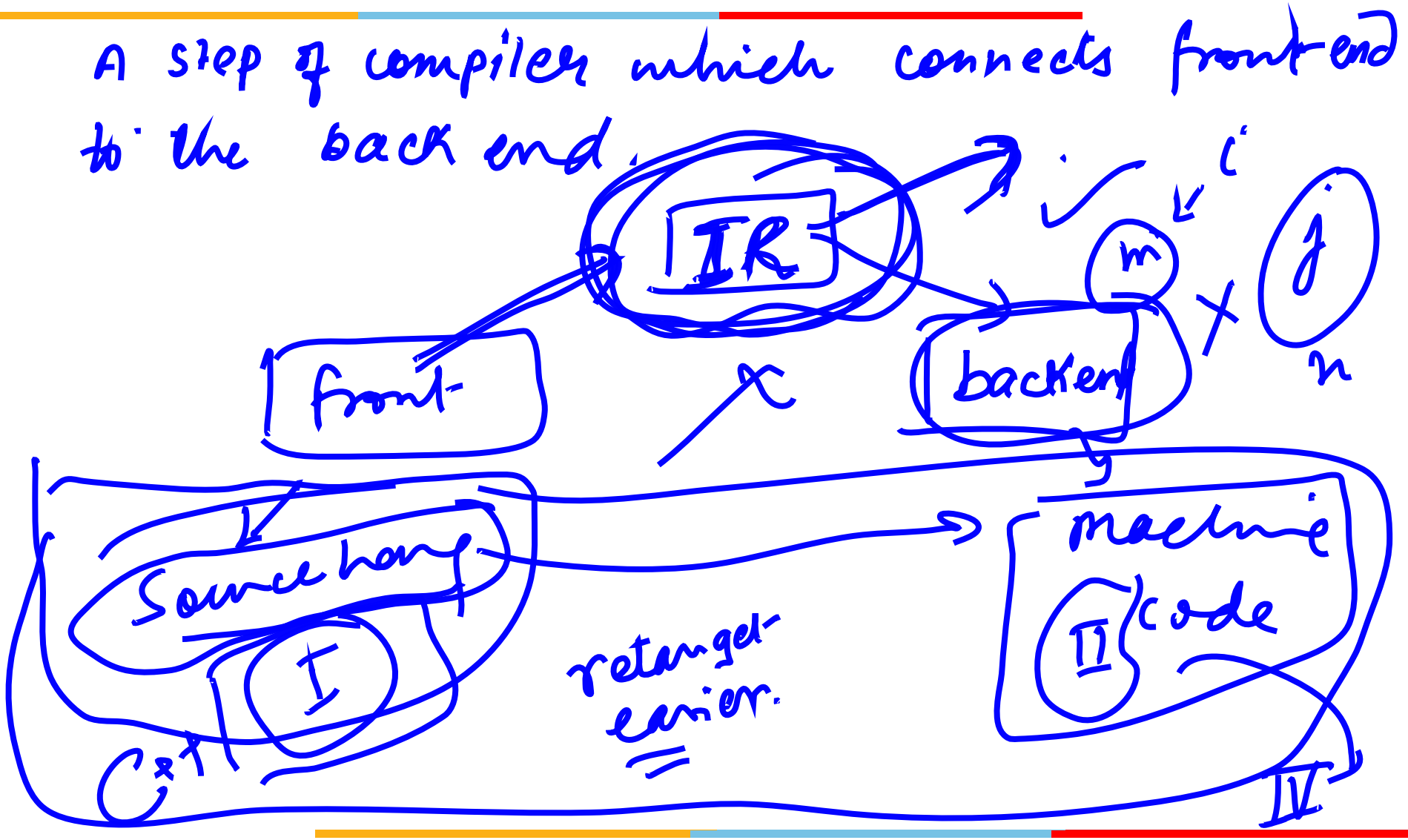
Semantic analysis

- Generalization of CFG: Attribute formalism
  - Synthesized Attributes
  - inherited attributes
- Syntax directed translation- associating semantic rules with productions



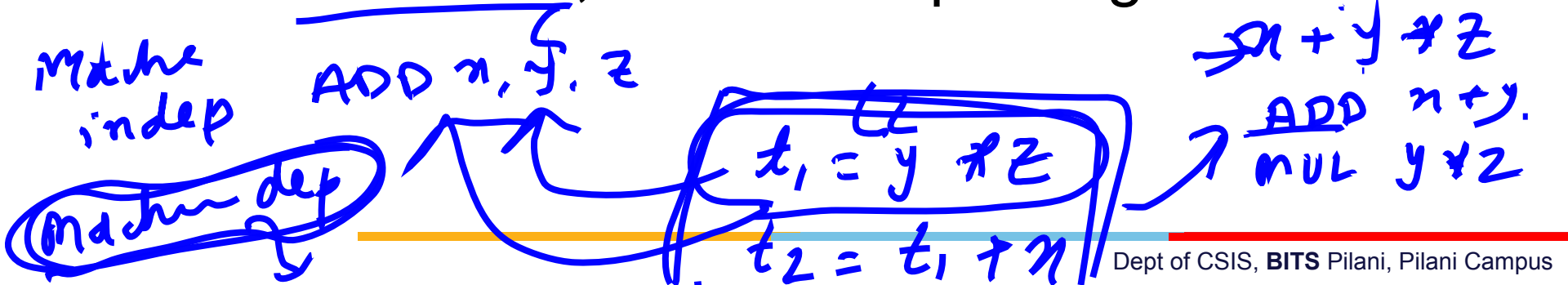


Intermediate representation (IR)



IR Language

- a language between source and target code
- provides an intermediate level of abstraction
  - more details than source
  - fewer details than target code
  - less machine dependent, easier to retarget
  - allow machine independent optimizations
  - can be implemented by syntax-directed translation, folded into parsing



IR Language classification

High-level representation

- closer to source
- easy to generate
- optimization is difficult since input program is not broken sufficiently

Low-level representation

- closer to target code
- easy to generate target code
- need more effort to translate source to IR

IR Language classification

High-level representation

- abstract syntax tree
- Directed Acyclic Graph

Low-level representation

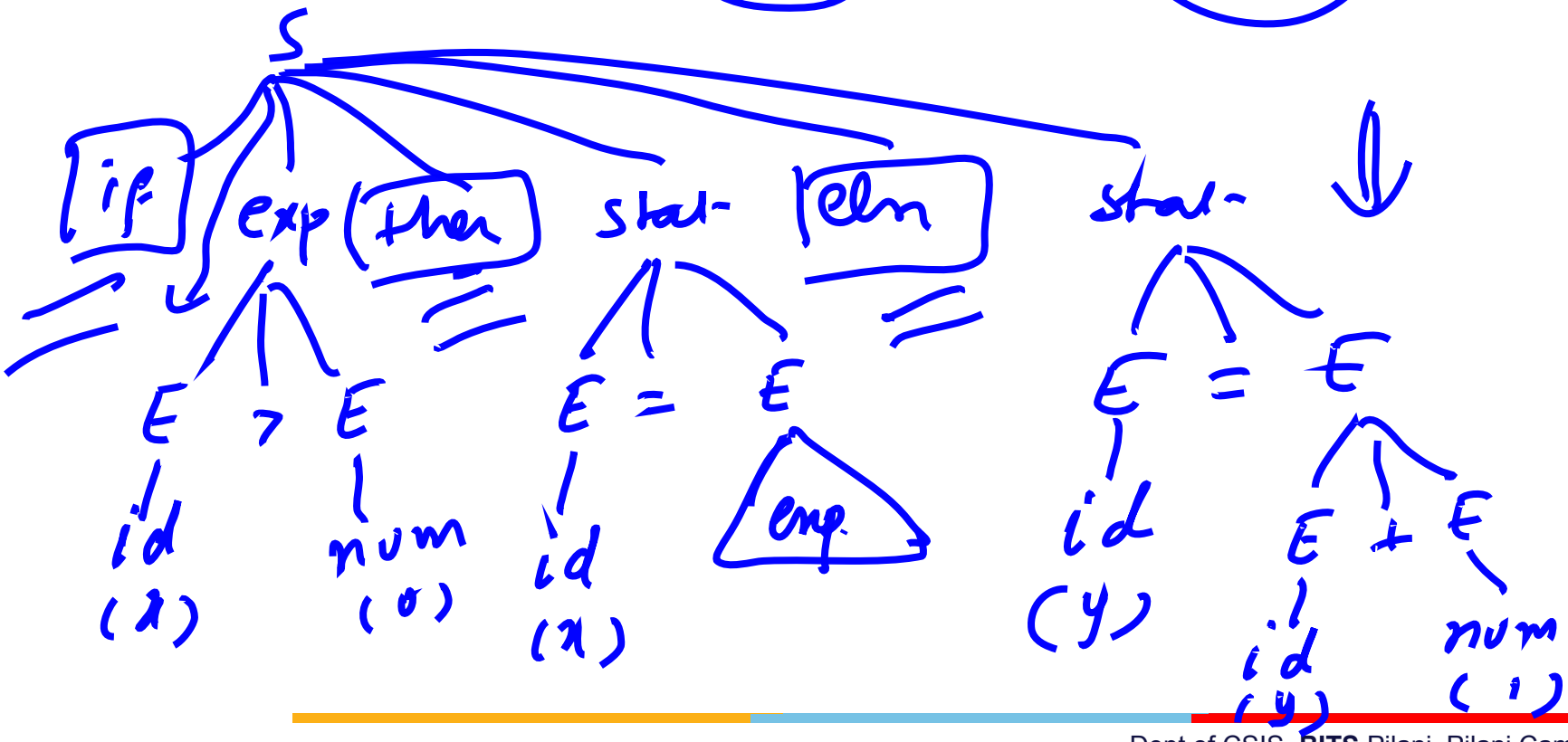
- Three address code

Abstract syntax tree

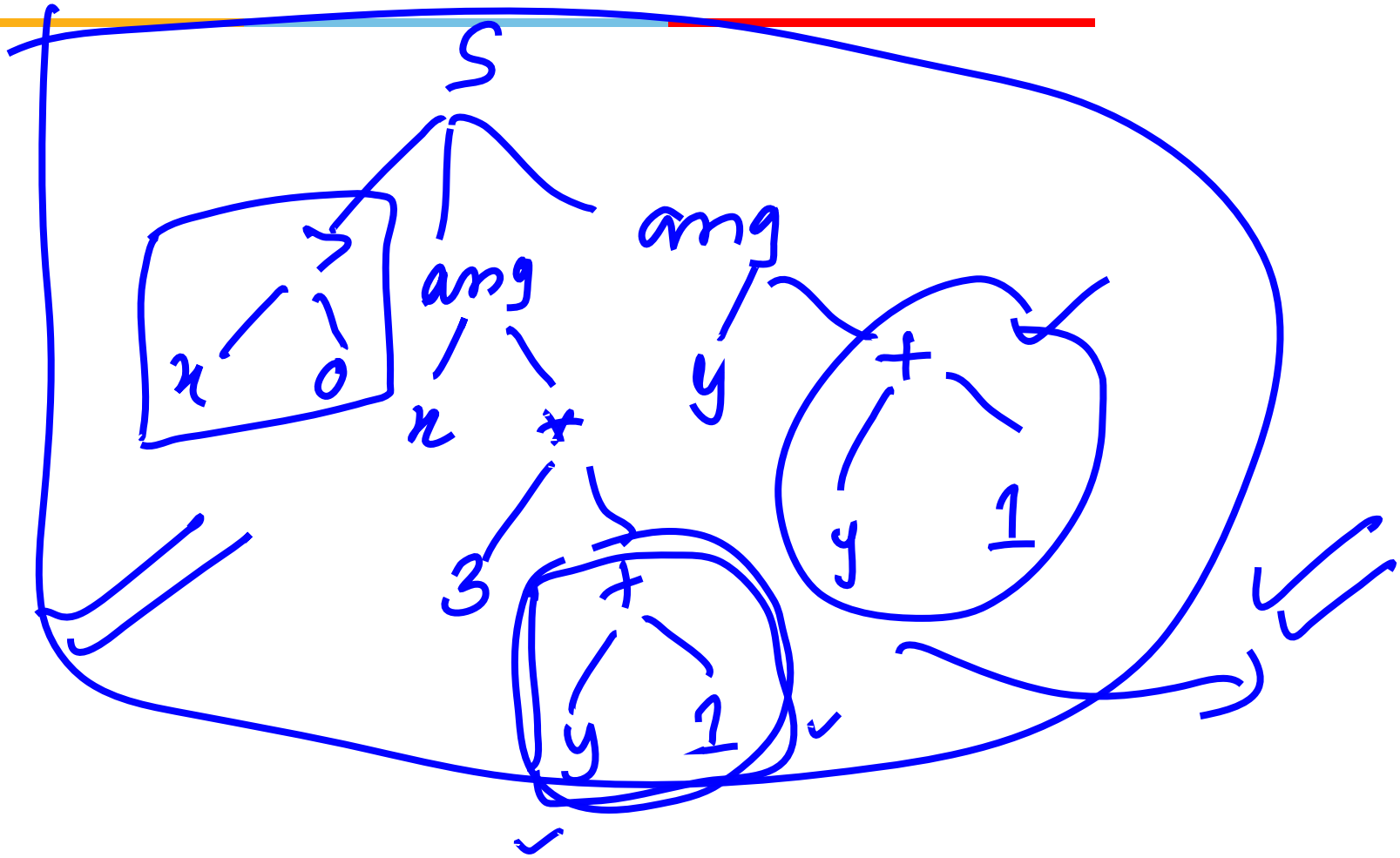
→ compact form of parse tree

→ Node are operator and children are operator

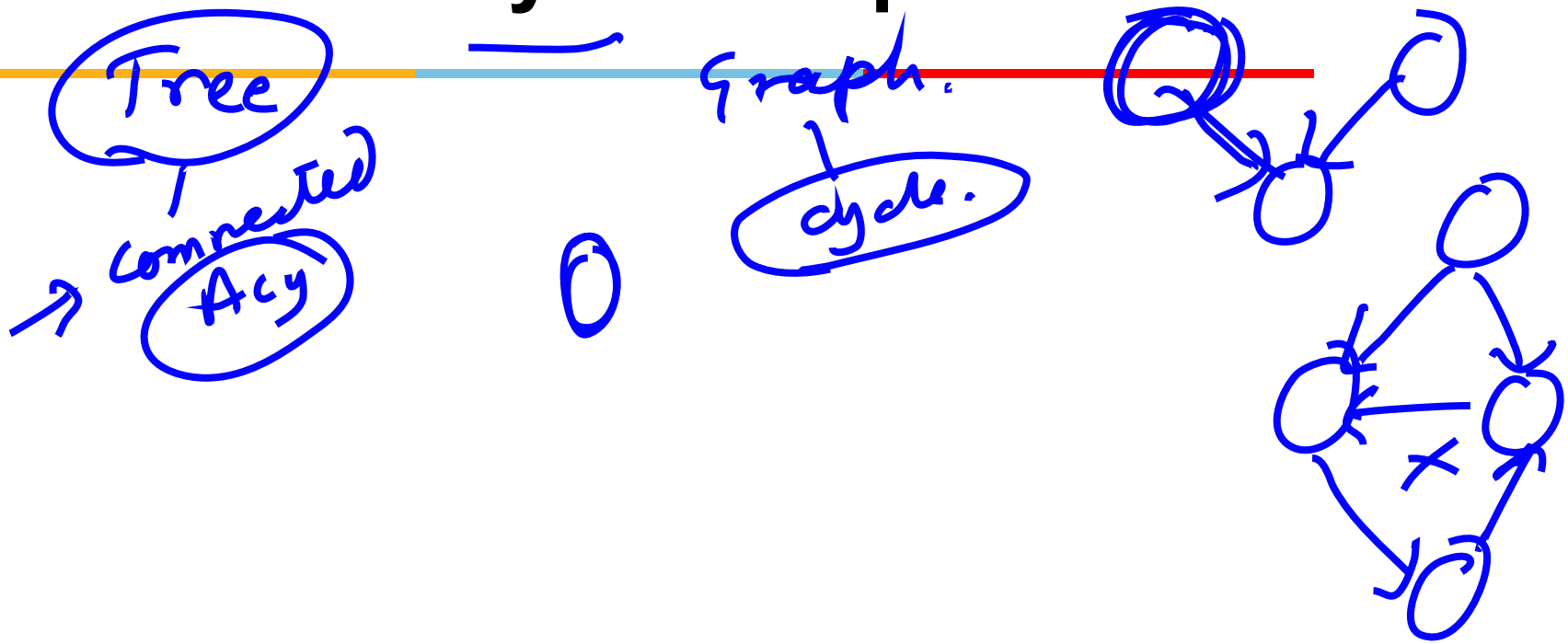
e.g. if x>0 then x = 3*(y+1) else y = y+1



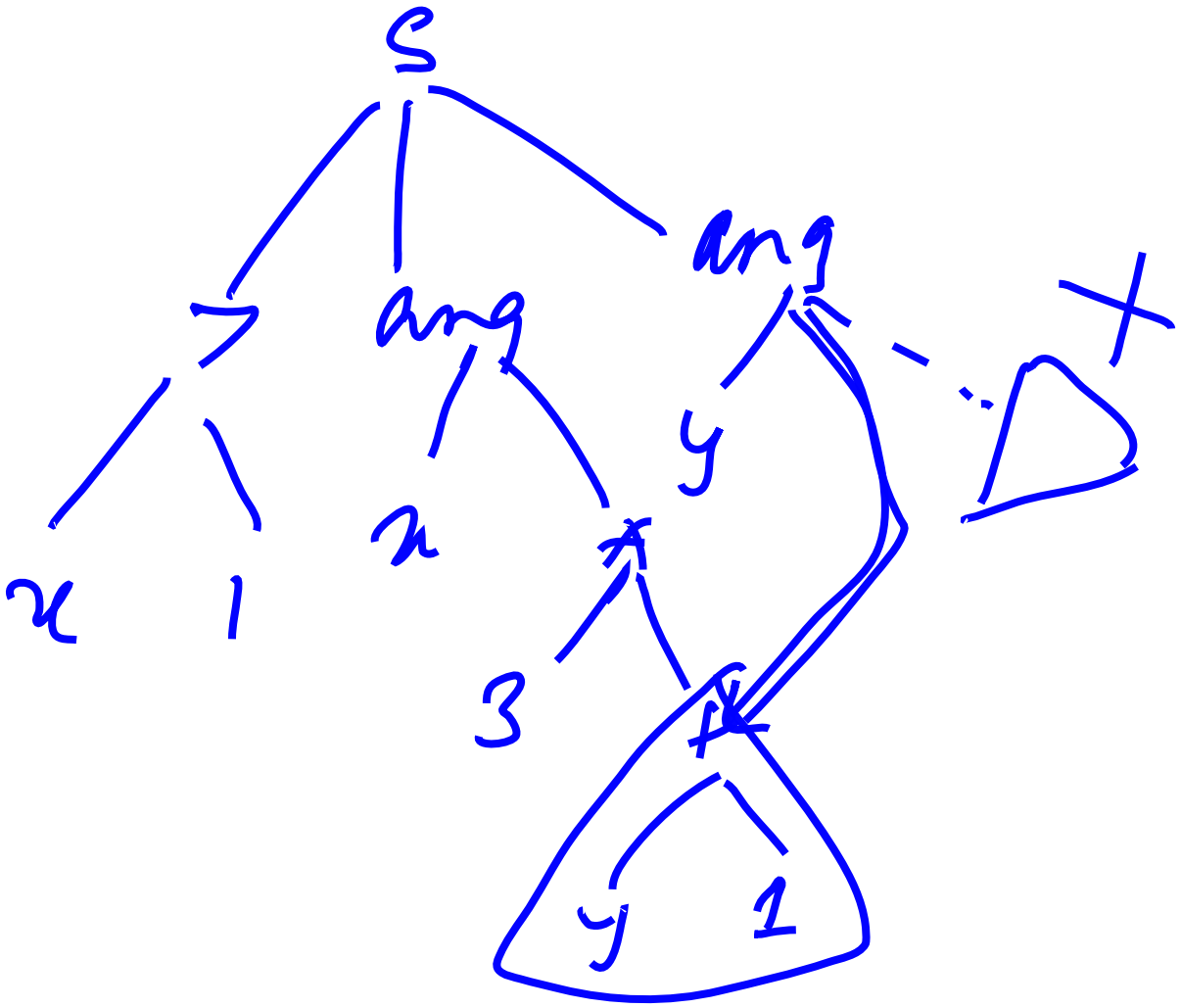
Abstract syntax tree



Directed Acyclic Graph



Directed Acyclic Graph



Low-level IR

- known as high-level assembly
- use register names, but has unlimited number
- use control structures like assembly language
- use opcodes, but some are high level
  - e.g push, translates to several assembly opcodes
  - most opcodes correspond directly to assembly opcodes

Low-level IR

- depends on target machine:
  - 0-address code for stack machines
  - 2-address code for machines with memory-register operations
  - 3-address code for RISC architectures

Three address code

- each instruction is of the form

  x:=y op z

  x:=op y

- y and z are registers or constants
- permit only one operator on RHS
- offers flexibility in terms of target code generation and optimization

Three address code

- The expression x+y*z is translated in

  t_1:= y*z

  t_2:= x + t_1

- each subexpression has a name

Three address code

- Similar to assembly code
- But use any number of IL registers to hold intermediate value.

Intermediate code

- Write a function igen(e,t) for intermediate code generation
- Compute the value of e in register t
- Example: igen(e_1+e_2, t)

  igen(e_1, t_1) (t_1 is a fresh register)

  igen(e_2, t_2) (t_2 is a fresh register)

  t := t_1 + t_2

Statements in three-address code

- assignments
- jumps
- pointer and address assignments
- procedure call/returns
- miscellaneous

Assignment Statements

- binary operator x = y op z
- unary operator x = op y
- x=y

for all operators in source language, there must be a counterpart in IR language

Index Assignments

- Only 1-d arrays are supported
- Higher dim arrays need to be converted into 1-d arrays
- e.g., x = a[i]

  x[i] = a

Jump Statements

- conditional and unconditional jumps
- goto L, L being a label
- if x relop y goto L

Address and Pointer assignments

- x = & y
- x=*y
- x = y, simple pointer assignments

Thank You!