Chap7 Semantic Processing

- **Syntax-directed translation**
  - **analysis: variable declarations, type errors**
  - **synthesis: IR or actual code**

- **The semantic action is attached to the productions (or subtrees of a syntax tree).**

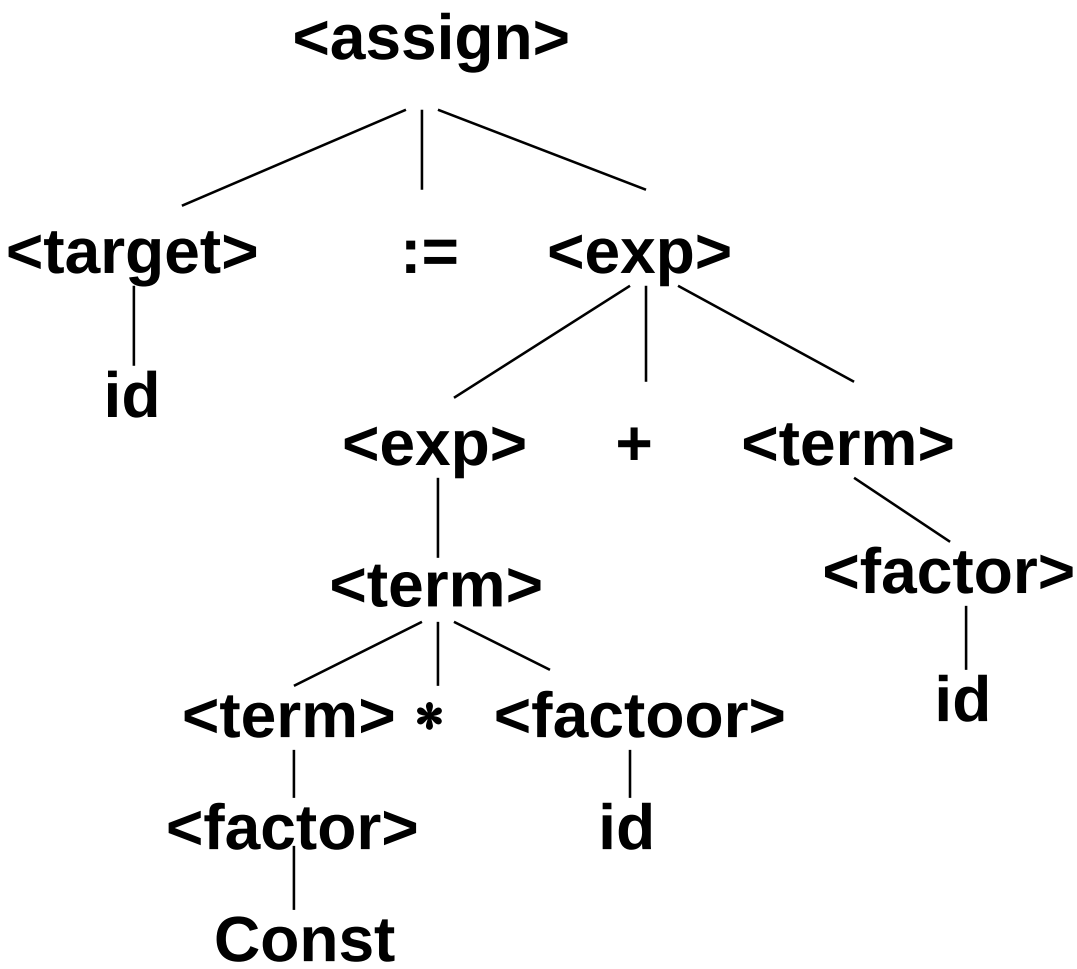
- **parsing: build the parse tree**
- **semantic processing: build and decorate the abstract syntax tree (AST)**

EX. Nonterminals for operator precedence and associativity need not be included.

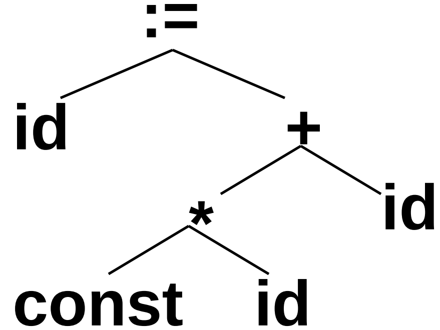
EX. Nonterminals used for ease of parsing may be omitted in the abstract syntax tree.

Chap7 Semantic Processing

Ex. parse tree



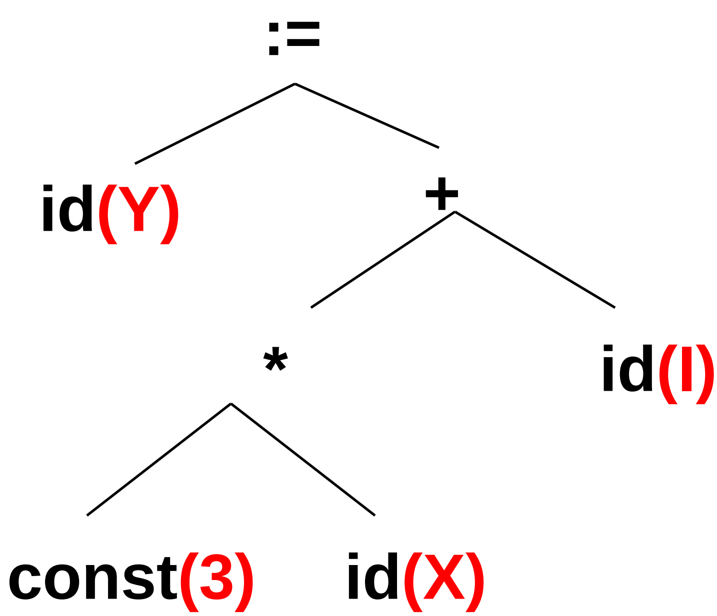
abstract syntax tree



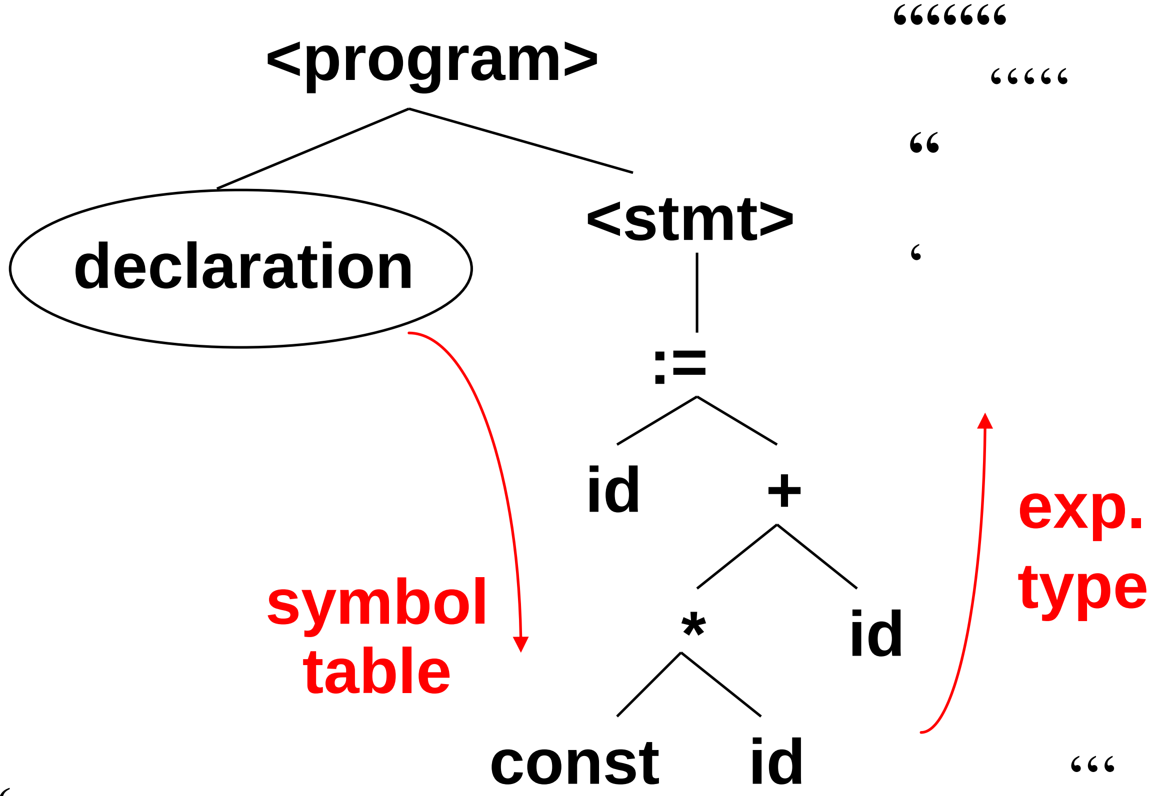
- **Semantic routines traverse the AST, computing attributes of the nodes of AST.**

- **Initially, only leaves (i.e. terminals, e.g. const, id) have attributes.**

Ex. Y := 3*X + I



- We then propagate the attributes to other nodes, using some functions, e.g.
  - build symbol table
  - attach attributes of nodes
  - check types, etc.
- bottom-up/top-down propagation



check types:

integer * or floating *

Need to consult symbol table for types of id's.

- **After attribute propagation is done, the tree is decorated and ready for code generation.**

- **We make another pass over the decorated AST to generate code.**

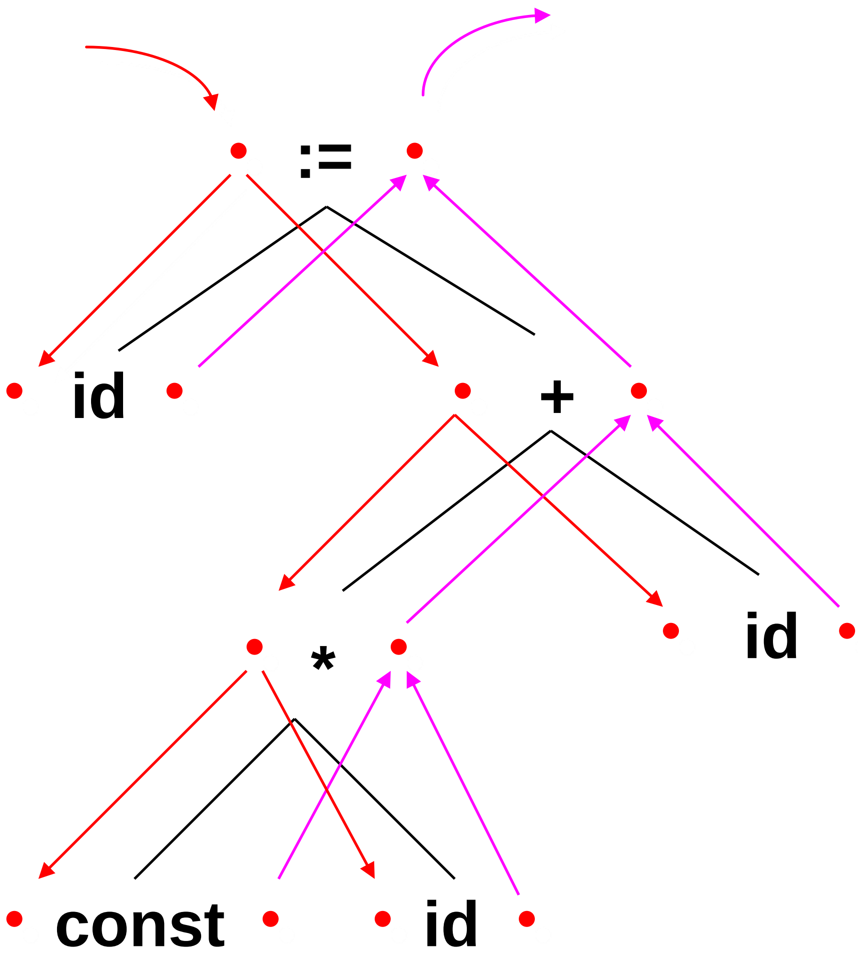
- **Actually,**

building the AST
decorating the AST
generating code

these can be combined in a single pass.

- **What we have described is essentially the attribute grammars(AG). Details in chap.14.**

Conceptually



Attributes flow in the AST.

§7.1.2 Compiler Organization Alternatives

<1> 1- pass analysis and synthesis

scanning
parsing
checking
translation

interleaved in a single pass.

Ex. Micro compiler in chap.2.

- Since code generation is limited to looking at one tuple at a time, few optimizations are possible.

Ex. Consider register allocation, which requires a more global view of the AST.

- We wish the code generator completely hide machine details and semantic routines become independent of machines.
- However, this is violated sometimes in order to produce better code.

Ex. Suppose there are several classes of registers, each for a different purpose.

Then register allocation is better done by semantic routines than code generator since semantic routines have a broader view of the AST.

<2> 1-pass + peephole

1 pass : generate code
1 pass : peephole optimization

- Peephole : looking at only a few instructions at a time

  - simple but effective
  - simplify code generator since there is a pass of post-processing.

<3> 1 pass + code gen pass

1st pass: analysis and IR
2nd pass: code generation

(+) flexible design for the code generator
(+) may use optimizations for IR
(+) greater independence of target machines (the front-end is quite independent of target machines.)
(+) re-targeting is easier.

<4> multipass analysis

- For limited addr. space,

scanner
parser
declaration
static checking

each is a pass.

- complete separation of analysis and synthesis.

<5> multipass synthesis

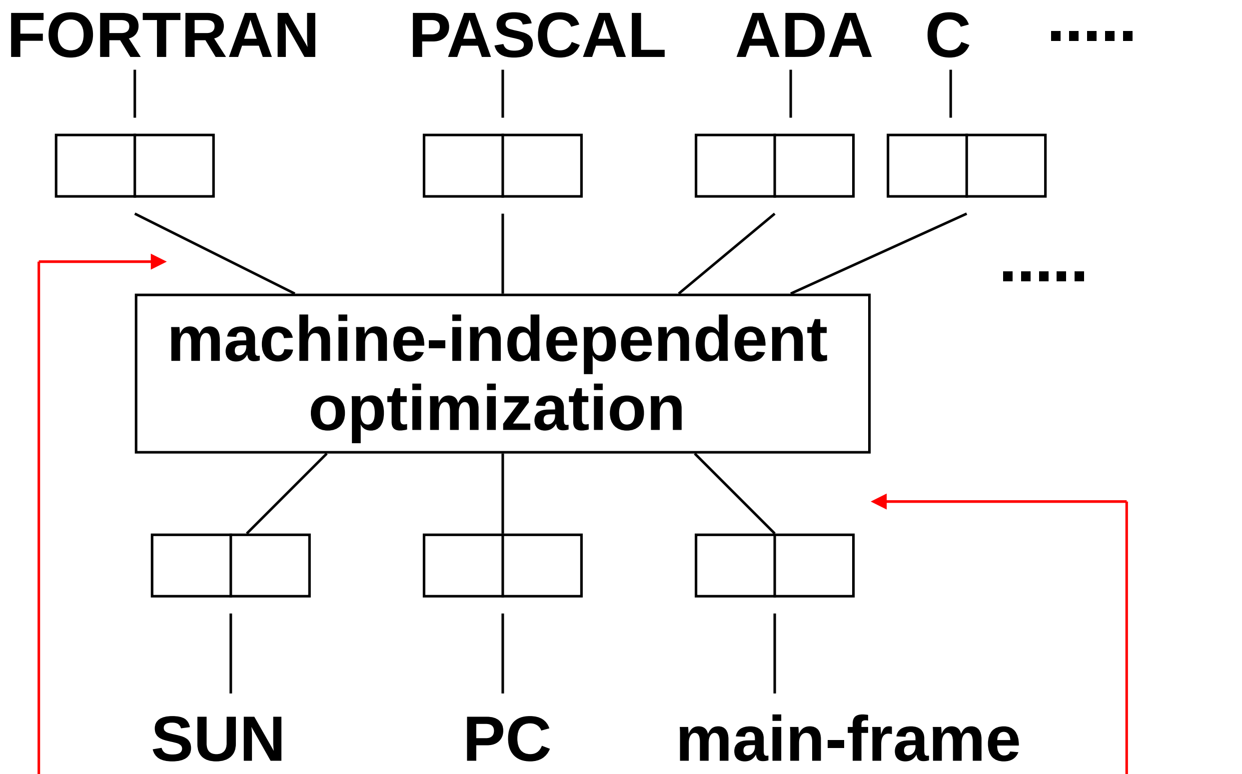
IR → machine-independent optimization passes → machine-dependent optimization passes

→ code gen passes → peephole →

- **Many complicated optimization and code generation algorithms require multiple passes.**

<6> multi-language and multi-target compilers

- Components may be shared and parameterized.



language- and machine-independent IRs

Ex. Ada uses Diana(language-dependent IR)

Ex. GCC uses two IRs.

- one is high-level tree-oriented
- the other(RTL) is more machine-oriented

§7.1.3 Single Pass

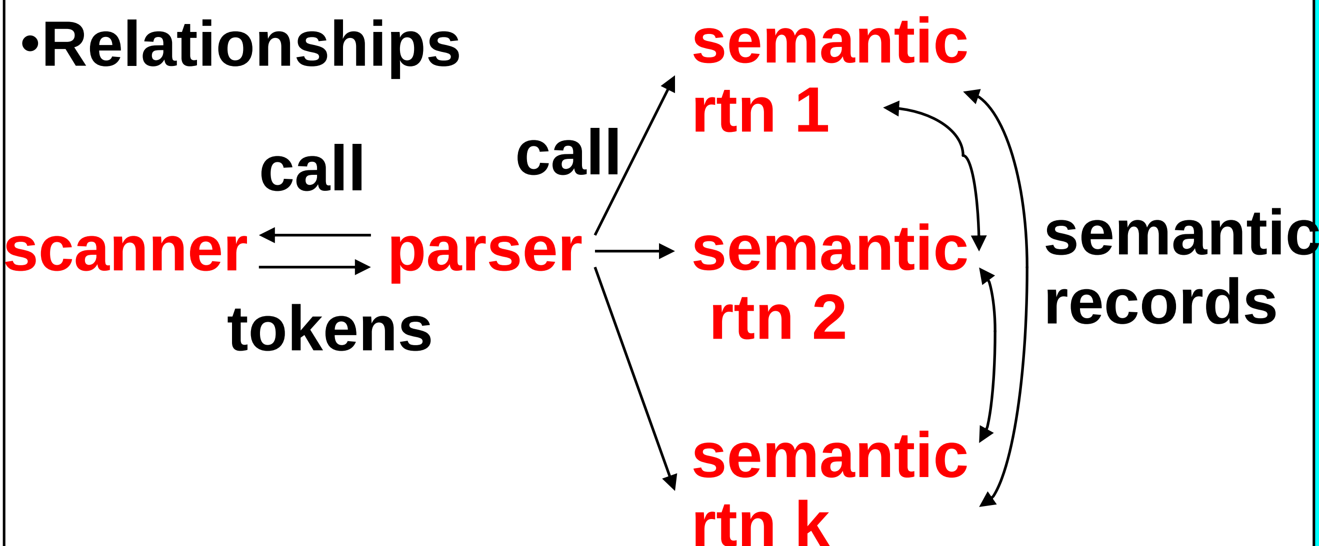
• In Micro of chap 2, scanning, parsing and semantic processing are interleaved in a single pass.

(+) simple front-end

(+) less storage if no explicit trees

(-) immediately available information is limited since no complete tree is built.

•Relationships



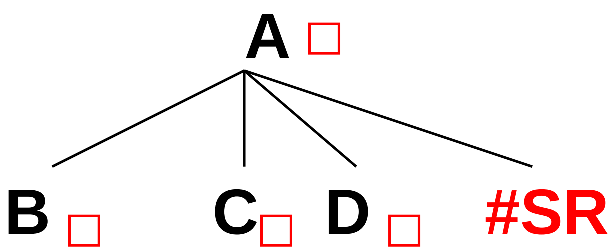
• Each terminal and nonterminal has a semantic record.

• Semantic records may be considered as the attributes of the terminals and non-terminals.

- For terminals, the semantic records are created by the scanner.

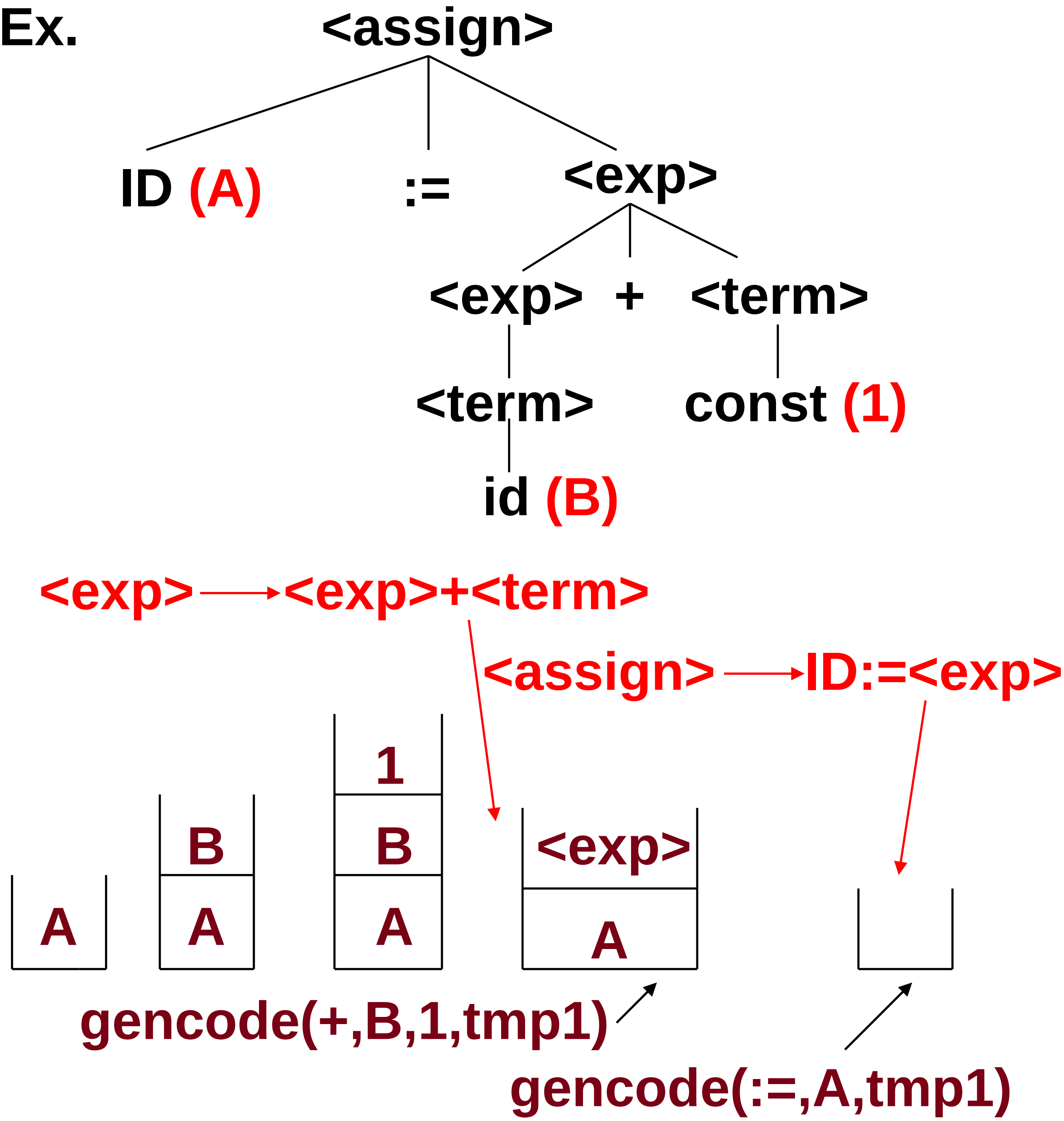
- For nonterminals, the semantic records are created by a semantic routine when a production is recognized.

ex. A $\longrightarrow$ B C D #SR



- Semantic records are transmitted among semantic routines via a **semantic stack**.

Ex.



1 pass = 1 post-order traversal of the parse tree

parsing actions —— build parse trees

semantic actions —— post-order traversal

Compare

<1> build a parse tree and then traverse

(+) flexible
(+) more powerful

<2> build and traverse the tree in an interleaved way

(+) simple
(-) limited

§ 7.2 Semantic Processing

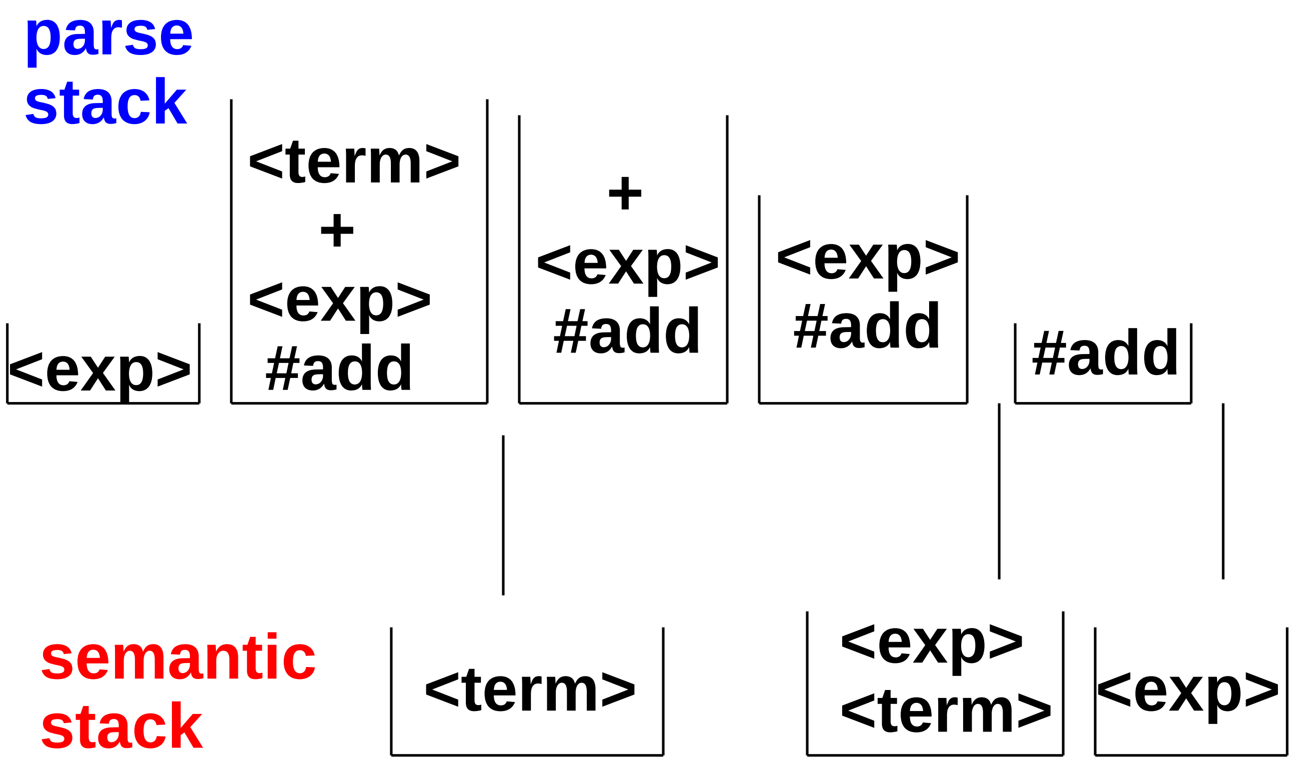
- **Semantic routines may be invoked in two ways:**

<1> By parsing procedures, as in the recursive descent parser in chap 2

<2> by the parser driver, as in LL and LR parsers.

§ 7.2.1 LL(1)

<exp> → <term> + <exp> #add



- Some productions have no action symbols; others may have several.

- Semantic routines are called when action symbols appear on stack top.

§ 7.2.2 LR(1)

- **Semantic routines are invoked only when a structure is recognized.**
- **In LR parsing, a structure is recognized when the RHS is reduced to LHS.**
- **Therefore, action symbols must be placed at the end.**

Ex. # ifThen
<stmt>
→ if <cond> then <stmt> end
→ if <cond> then <stmt> else <stmt> end
ifThenElse

After shifting " if <cond> " , the parser cannot decide which of #ifThen and #ifThenElse should be invoked.

- **cf. In LL parsing, the structure is recognized when a nonterminal is expanded.**

- **However, sometimes we do need to perform semantic actions in the middle of a production.**

Ex.

<stmt> → if <exp> • then <stmt> end

generate code for <exp>

Need a conditional jump here.

generate code for <stmt>

Solution: Use two productions:

<stmt> → <if head>
then <stmt> end #finishIf
<if head> → if <exp> #startIf

semantic hook
(only for semantic processing)

Another problem:

What if the action is not at the end?

Ex.

<prog> → #start begin <stmt> end

We need to call #start.

Solution: Introduce a new nonterminal.

<prog> → <head> begin <stmt> end

<head> → #start

- **YACC automatically performs such transformations.**

§ 7.2.3 Semantic Record Representation

- **Since we need to use a stack of semantic records, all semantic records must have the same type.**
  - » **variant record in Pascal**
  - » **union type in C**

Ex.

```
enum kind {OP, EXP, STMT, ERROR};
typedef struct {
      enum kind tag;
      union {
            op_rec_type        OP_REC;
            exp_rec_type       EXP_REC;
            stmt_rec_type      STMT_REC;
            ......
      }
} sem_rec_type;
```

- How to handle errors?

Ex. A semantic routine needs to create a record for each identifier in an expression.

What if the identifier is not declared?

Solution 1: make a bogus record

This method may create a chain of meaningless error messages due to this bogus record.

Solution 2: create an ERROR semantic record

No error message will be printed when ERROR record is encountered.

- WHO controls the semantic stack?
  » action routines
  » parser

§ 7.2.4 Action-controlled semantic stack

- Action routines take parameters from the semantic stack directly and push results onto the stack.

- Implementing stacks:
  1. array
  2. linked list

- Usually, the stack is transparent - any records in the stack may be accessed by the semantic routines.
  (-) difficult to change

Two other disadvantages:

(-) Action routines need to manage the stack.

(-) Control of the stack is distributed among the many action routines.

- **Each action routine pops some records and pushes 0 or 1 record.**
- **If any action routine makes a mistake, the whole stack is corrupt.**

Solution 1: Let parser control the stack

Solution 2: Introduce additional stack routines

parser → **stack routines** → **action routines**

- **If action routines do not control the stack, we can use opague (or abstract) stack: only push() and pop() are provided.**
  - **(+) clean interface**
  - **(-) less efficient**

§ 7.2.5 parser-controlled stack

• **LR**

Semantic stack and parse stack operate in parallel [shifts and reduces in the same way].

Ex. <stmt> → if <exp> then <stmt> end

parser stack		semantic stack
<stmt>	....	<stmt>
then	....	then
<exp>	....	<exp>
if	....	if
:		

parser stack **semantic stack**

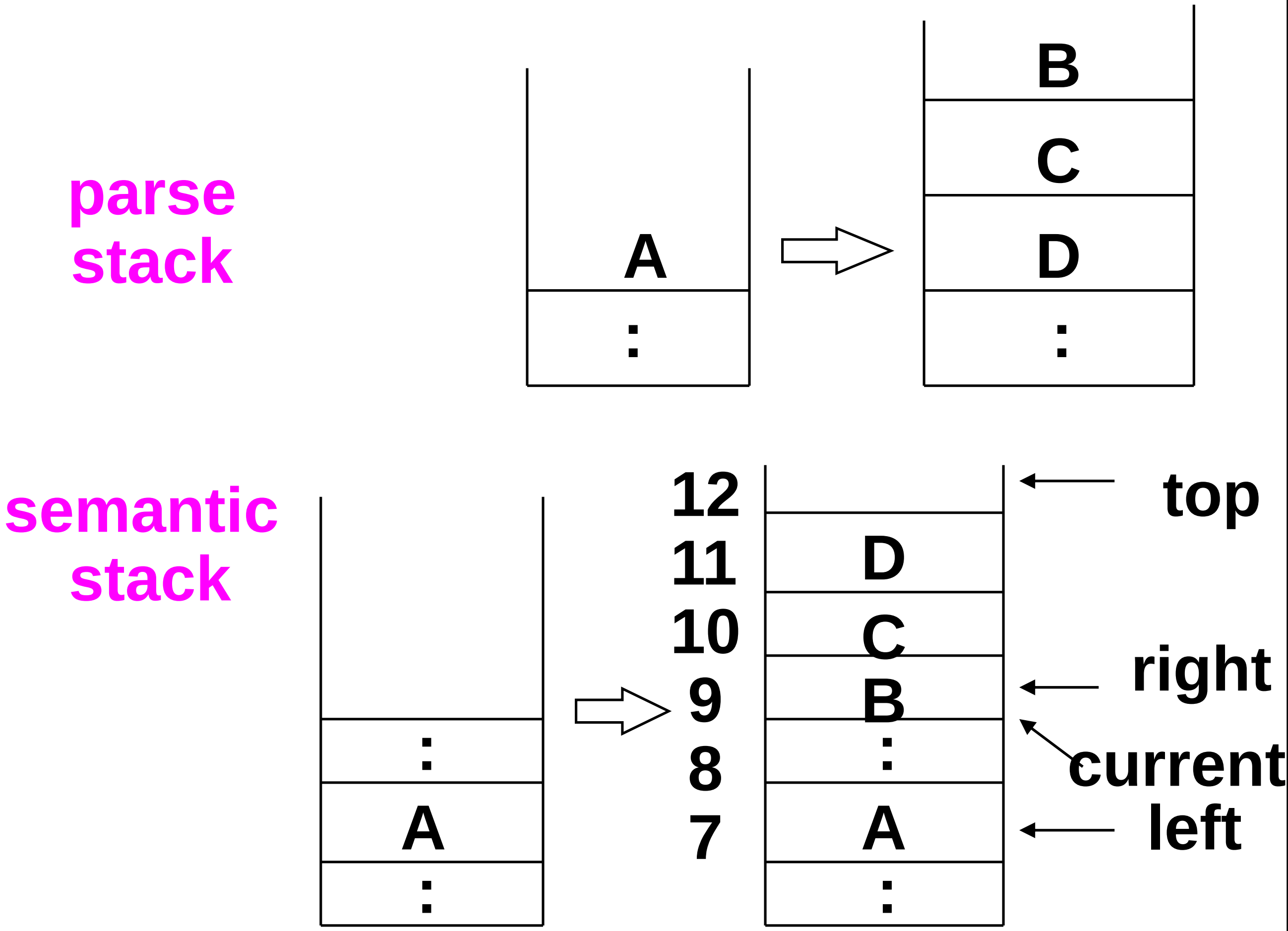
may be combined

Ex. YACC generates such parser-controlled semantic stack.

<exp> → <exp> + <term>

{ $$.value=$1.value+$3.value;}

- LL parser-controlled semantic stack
  - Every time a production A $\rightarrow$ B C D is predicted,

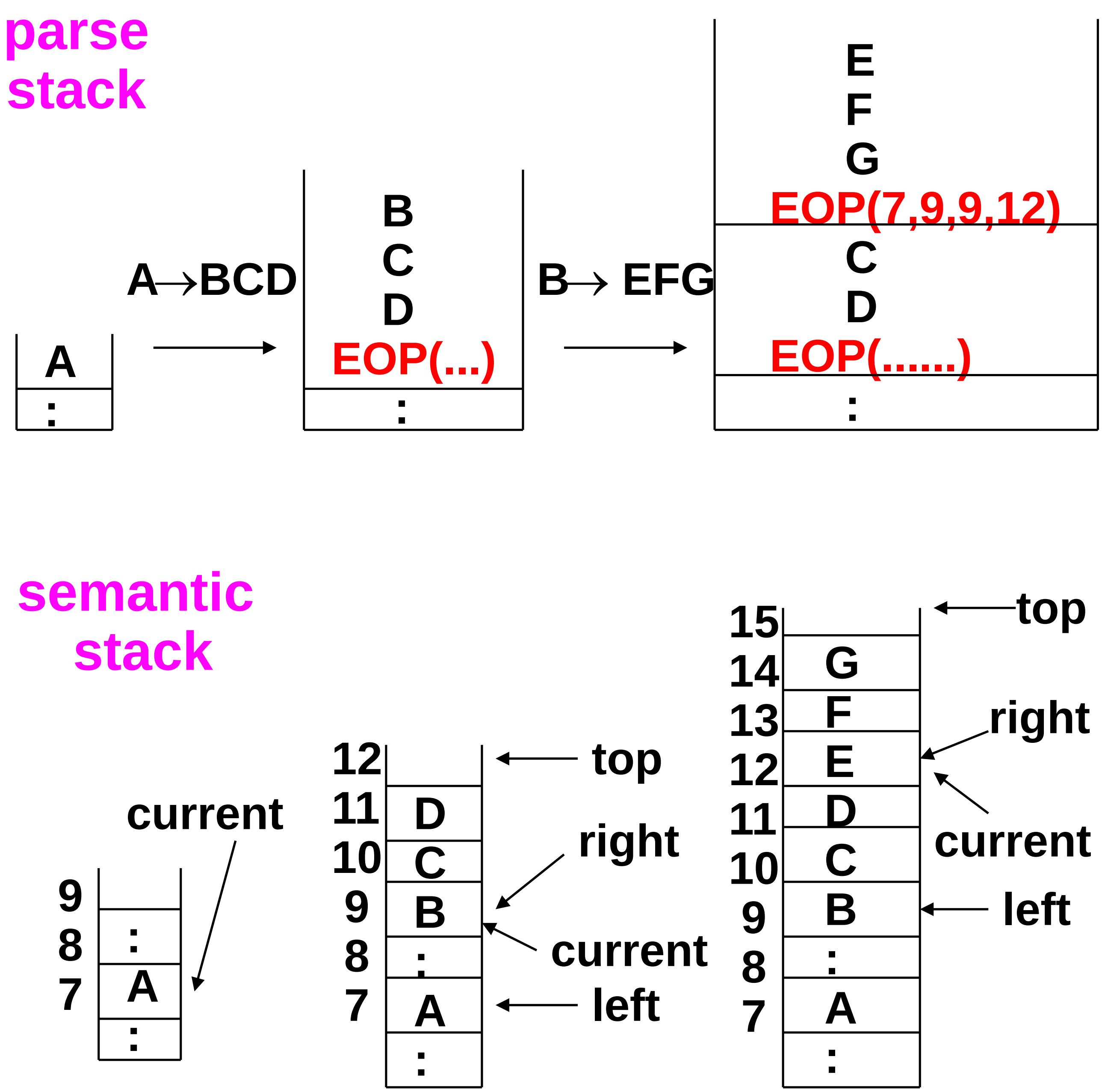


Need four pointers fir the semantic stack (left, right, current, top).

However, when a new production

$B \rightarrow E\ F\ G$

is predicted, the four pointers will be overwritten. Therefore, create a new EOP record for the four pointers on the parse stack.


parse
stack
A
:
A→BCD
B
C
D
EOP(...)
:
B→ EFG
E
F
G
EOP(7,9,9,12)
C
D
EOP(......)
:
semantic
stack
current
9
8
7
:
A
:
12
11
10
9
8
7
D
C
B
:
A
:
top
right
current
left
15
14
13
12
11
10
9
8
7
G
F
E
D
C
B
:
A
:
top
right
current
left

- When EOP record appears on stack top, restore the four pointers, which essentially pops off records from the semantic stack.

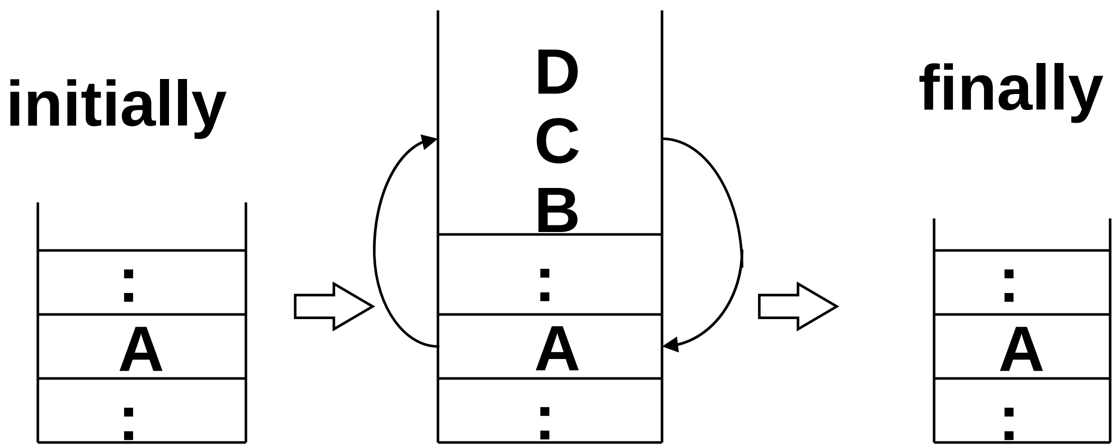
- Note that all push() and pop() are done by the parser, not by the action routines.
- Semantic records are passed to the action routines by parameters.

  Ex.

  <primary> $\longrightarrow$ (<exp>) #copy($2,$$)

- Initial information is stored in the semantic record of LHS.

  After the RHS is processed, the resulting information is stored back in the semantic record of LHS.



information flow
(attributes)

Figure 7.10 Micro grammar with parameterizd action symbols

Trace an example:

begin a := b + c end;

(-) Semantic stack may grow very big.

<fix> Certain nonterminals never use semantic records, e.g. <stmt list> and <id list>.

We may insert #reuse before the last nonterminal in each of their productions.

Ex.

<stmt list> $\rightarrow$ <stmt> #reuse <stmt tail>
<stmt tail> $\rightarrow$ <stmt> #reuse <stmt tail>
<stmt tail> $\rightarrow$

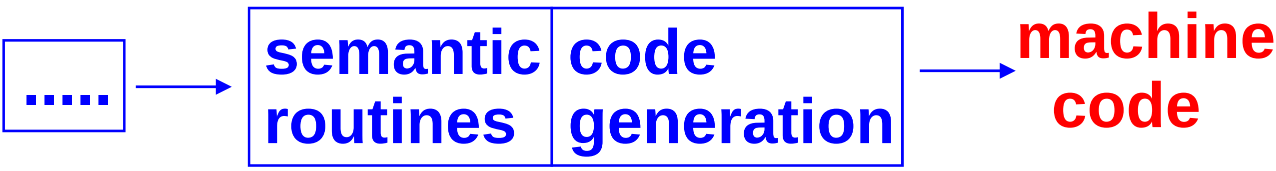
Evaluation:

Parser-controlled semantic stack is easy with LR, but not so with LL.

§ 7.3 Intermediate representation and code generation

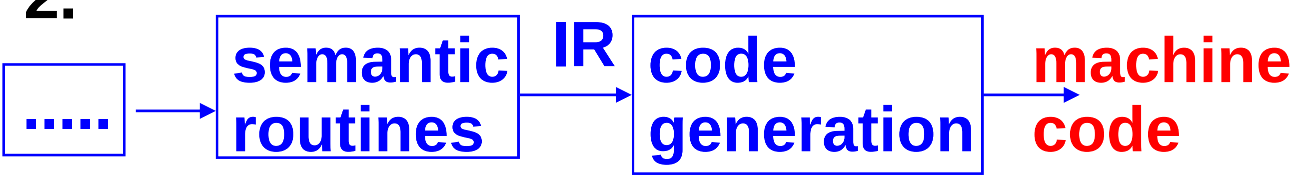
Two possibilities:

1.



(+) no extra pass for code generation
(+) allows simple 1-pass compilation

2.



(+) allows higher-level operations
e.g. open block, call procedures.
(+) better optimization because IR
is at a higher level.
(+) machine dependence is isolated
in code generation.

IR: good for optimization and portability

machine code: simple

§ 7.3.2

1. postfix form

Ex.

a+b	ab+
(a+b)*c	ab+c*
a+b*c	abc*+
a:=b*c+b*d	abc*bd*+:=

(+) simple and concise
(+) good for driving an interpreter
(-) NOT good for optimization or
code generation

2. 3-addr code

» triple : op arg1 arg2

» quadruple: op arg1 arg2 arg3

a := b*c + b*d

(1) (* b c)	(1) (* b c t1)
(2) (* b d)	(2) (* b d t2)
(3) (+ <u>(1)</u> <u>(2)</u>)	(3) (+ <u>t1</u> <u>t2</u> <u>t3</u>)
(4) (:= <u>(3)</u> a)	(4) (:= <u>t3</u> a _)
intermediate results are referenced by the instruction #	use temporary names

• triple: more concise

But what if instructions are deleted, moved or added during optimization?

• Triples and quadruples are more similar to machine code.

- **More detailed 3-addr code:**
  - » **Add type information**

Ex. **a := b*c + b*d**

Suppose b,c are integer type, d is float type.

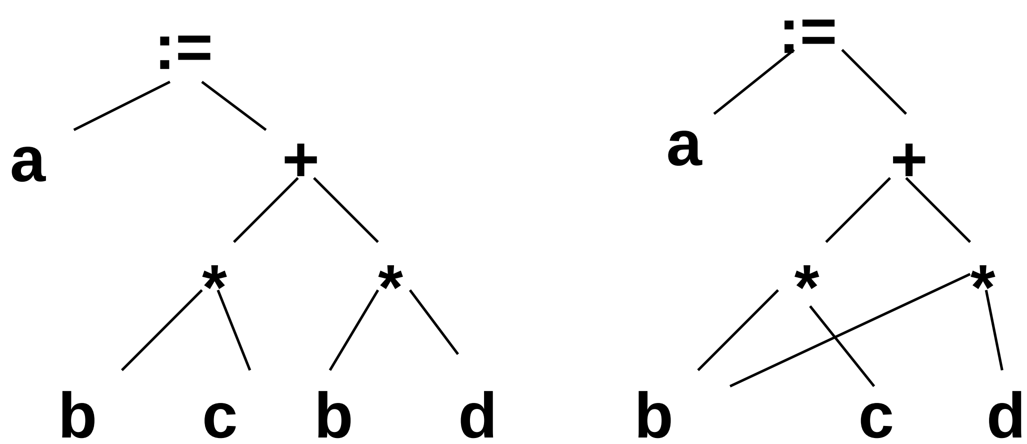
(1)	(I*	b	c)	(I*	b	c	t1)
(2)	(FLOAT	b	_)	(FLOAT	b	t2	_)
(3)	(F*	(2)	d)	(F*	t2	d	t3)
(4)	(FLOAT	(1)	_)	(FLOAT	t1	t4	_)
(5)	(*f+	(4)	(3))	(F+	t4	t3	t5)
(6)	(:=	(5)	a)	(:=	t5	a	_)

- Sometimes, the number of arguments to operators may vary. The generalized 3-addr code is called tuples.

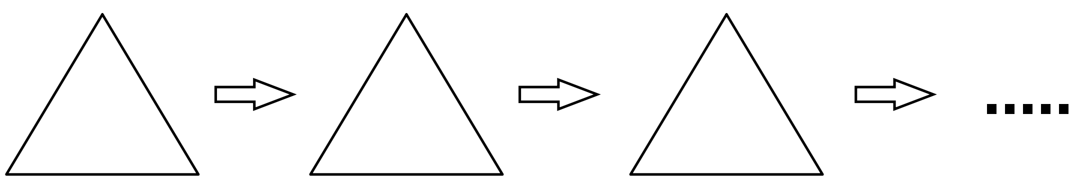
```
Ex.  (I*        b    c   t1)
     (FLOAT     b    t2)
     (F*        t2   d   t3)
     (FLOAT     t1   t4)
     (F+        t4   t3  t5)
     ( :=       t5   a)
```

3. Sometimes, we use **trees or DAG**

Ex. a := b*c + b*d



- More generally, we may use AST as IR. Machine-independent optimization is implemented as tree transformations.



Ex. Ada uses Diana.