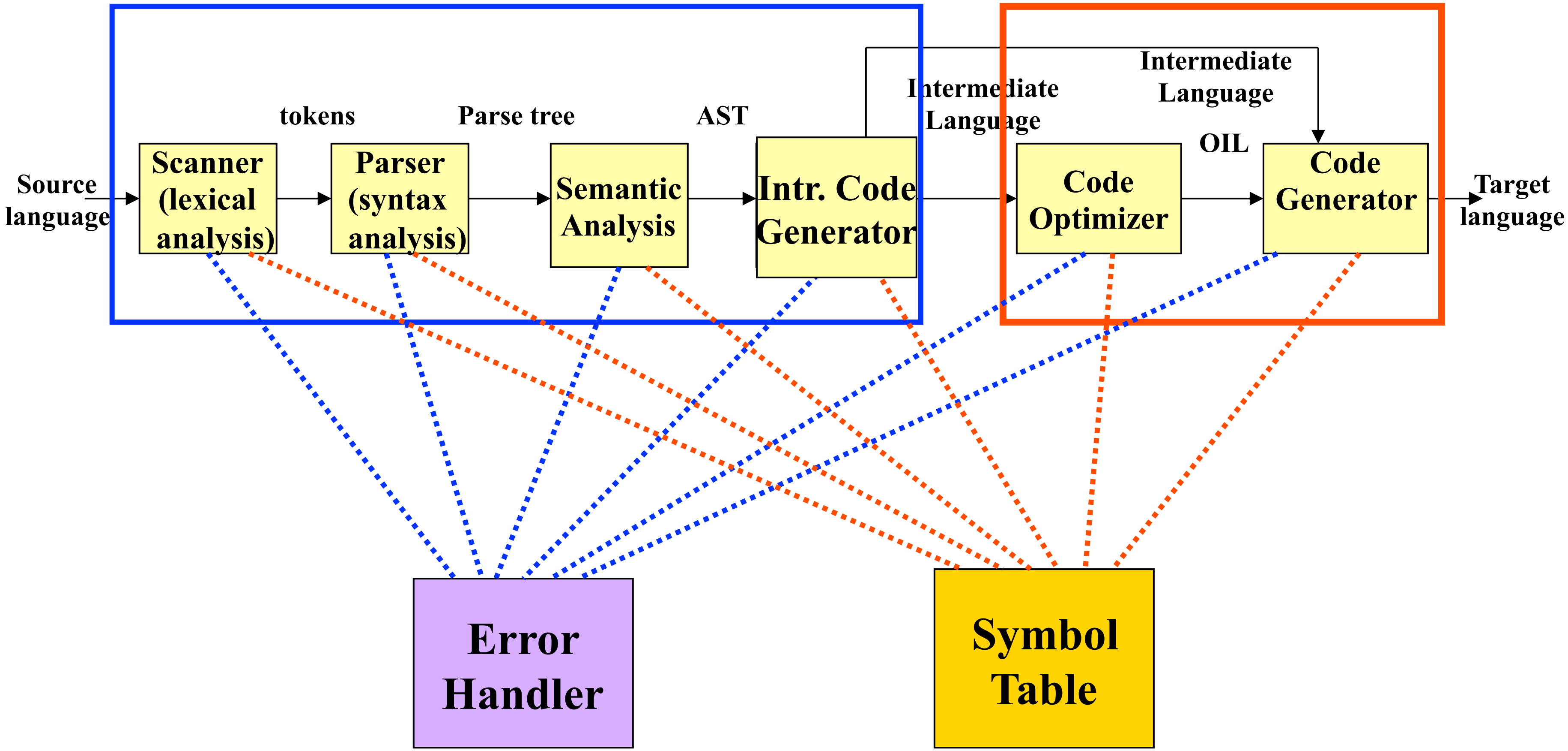
Language Processing Systems

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Intermediate/Code Generation



Intermediate Code Generation

Intermediate Code

- Similar terms: *Intermediate representation, intermediate language*
- Ties the front and back ends together
- Language and Machine neutral
- Many forms
- More than one intermediate language may be used by a compiler

Intermediate Representation

- There is no standard Intermediate Representation. IR is a step in expressing a source program so that machine understands it
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct
- Compiler writers want optimizations to be simple to write, easy to understand and easy to extend
- IR should be simple and light weight while allowing easy expression of optimizations and transformations.

Issues in new IR Design

- How much machine dependent
- Expressiveness: how many languages are covered
- Appropriateness for code optimization
- Appropriateness for code generation

Intermediate Languages Types

- Graphical IRs:
  - Abstract Syntax trees,
  - DAGs,
  - Control Flow Graphs,
  - etc.
- Linear IRs:
  - Stack based (postfix)
  - Three address code (quadruples)
  - etc.

Graphical IRs

- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well
- Control Flow Graphs (CFG) – explicitly model control flow

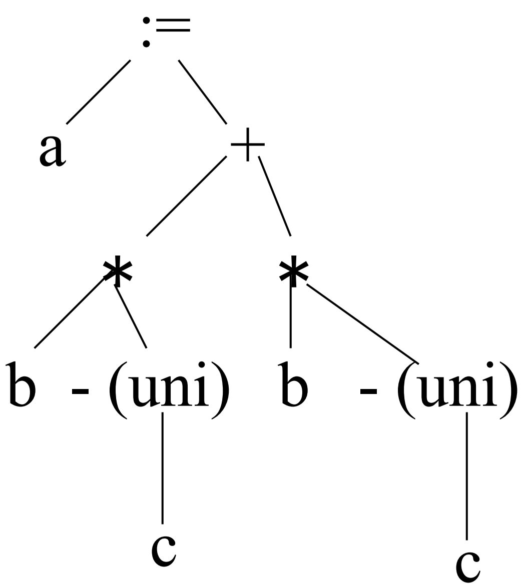
Abstract Syntax Tree/DAG

- Condensed form of parse tree
- Useful for representing language constructs
- Depicts the natural hierarchical structure of the source program
  - Each internal node represents an operator
  - Children of the nodes represent operands
  - Leaf nodes represent operands
- DAG is more compact than abstract syntax tree because common sub expressions are eliminated

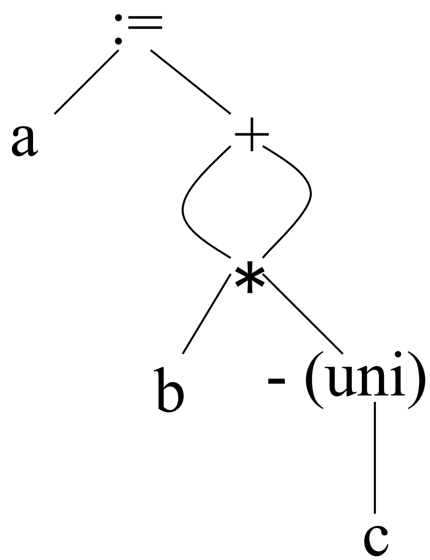
ASTs and DAGs:

Example: a := b *-c + b*-c

AST



DAG



Linear IR

- Low level IL before final code generation
  - A linear sequence of low-level instructions
  - Resemble assembly code for an abstract machine
    - Explicit conditional branches and goto jumps
  - Reflect instruction sets of the target machine
    - Stack-machine code and three-address code
  - Implemented as a collection (table or list) of tuples

- Linear IR examples
  - Stack based (postfix)
  - Three address code (quadruples)

Stack based: Postfix notation

- Linearized representation of a syntax tree
- List of nodes of the tree
- Nodes appear immediately after its children
- The postfix notation for an expression E is defined as follows:
  - **If E is a variable or constant then the postfix notation is E itself**
  - **If E is an expression of the form E_1 op E_2 where op is a binary operator then the postfix notation for E is**
    - **E_1' E_2' op where E_1' and E_2' are the postfix notations for E_1 and E_2 respectively**
  - **If E is an expression of the form (E_1) then the postfix notation for E_1 is also the postfix notation for E**

Stack based: Postfix notation

- No parenthesis are needed in postfix notation because
  - the position and parity of the operators permits only one decoding of a postfix expression

- Postfix notation for

  a = b * -c + b * - c

  is

  a b c - * b c - * + =

Stack-machine code

- Also called one-address code
  - Assumes an operand stack
  - Operations take operands from the stack and push results back onto the stack
  - Need special operations such as
    - **Swapping two operands on top of the stack**
- Compact in space, simple to generate and execute
    - **Most operands do not need names**
    - **Results are transitory unless explicitly moved to memory**
- Used as IR for Smalltalk and Java

Stack-machine code for x – 2 * y

```
Push 2
Push y
Multiply
Push x
subtract
```

Three address code

- It is a sequence of statements of the general form X := Y op Z where
  - X, Y or Z are names, constants or compiler generated temporaries
  - op stands for any operator such as a fixed- or floating-point arithmetic operator, or a logical operator

Three address code

- Only one operator on the right hand side is allowed
- Source expression like x + y * z might be translated into

  $t_1 := y * z$
  $t_2 := x + t_1$

  where t_1 and t_2 are compiler generated temporary names
- Unraveling of complicated arithmetic expressions and of control flow makes 3-address code desirable for code generation and optimization
- The use of names for intermediate values allows 3-address code to be easily rearranged
- Three address code is a linearized representation of a syntax tree where explicit names correspond to the interior nodes of the graph

Three address instructions

- 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

Three address code

- Every instruction manipulates at most two operands and one result. Typical forms include:
  - **Arithmetic operations: x := y op z | x := op y**
  - **Data movement: x := y [z] | x[z] := y | x := y**
  - **Control flow: if y op z goto x | goto x**
  - **Function call: param x | return y | call foo**

Example:

Three-address code for x – 2 * y

```
t1 := 2
t2 := y
t3 := t1*t2
t4 := x
t5 := t4-t3
```

Storing three-address code

- Store all instructions in a quadruple table
  - Every instruction has four fields: op, arg1, arg2, result
  - The label of instructions ➔ index of instruction in table

Example:

Three-address code

```
t1 := - c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Quadruple entries

	op	arg1	arg2	result
(0)	Uminus	c		t1
(1)	Mult	b	t1	t2
(2)	Uminus	c		t3
(3)	Mult	b	t3	t4
(4)	Plus	t2	t4	t5
(5)	Assign	t5		a

Alternative: store all the instructions in a singly/doubly linked list
What is the tradeoff?

Linear IR: Example

Example: **Linear IR for x – 2 * y**

Stack-machine code	two-address code	three-address code
Push 2 Push y Multiply Push x subtract	MOV 2 => t1 MOV y => t2 MULT t2 => t1 MOV x => t4 SUB t1 => t4	t1 := 2 t2 := y t3 := t1*t2 t4 := x t5 := t4-t3

Generating Intermediate Code

As we parse, generate IC for the given input. Use attributes to pass information about temporary variables up the tree

Example1: **a := b *-c + b*-c**

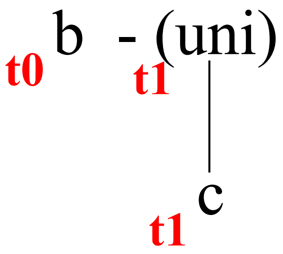
```
t0 = b
```

b

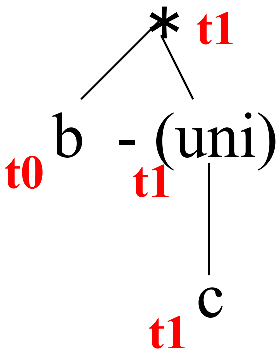
t0

Example1: a := b *-c + b*-c

```
t0 = b
t1 = -c
```

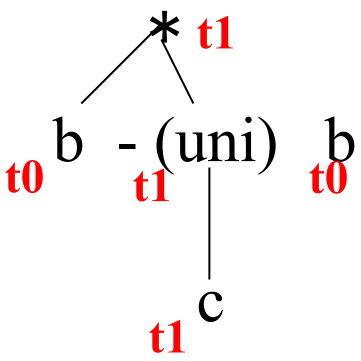


Example1: **a := b *-c + b*-c**



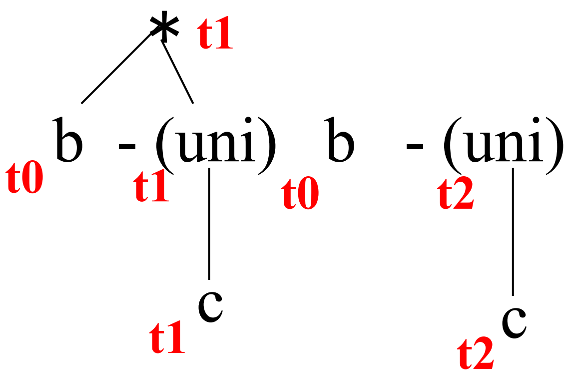

```
t0 = b
t1 = -c
t1 = t1 * t0
```

Example1: **a := b *-c + b*-c**



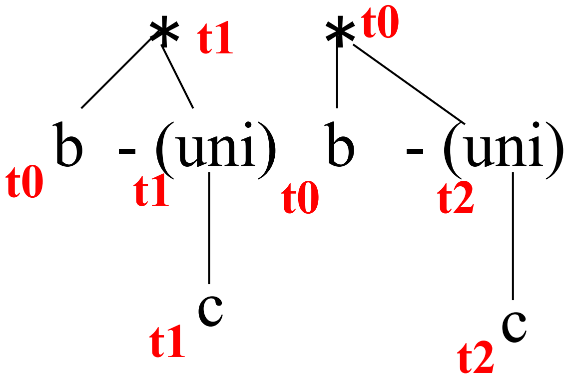

```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = b
```

Example1: **a := b *-c + b*-c**



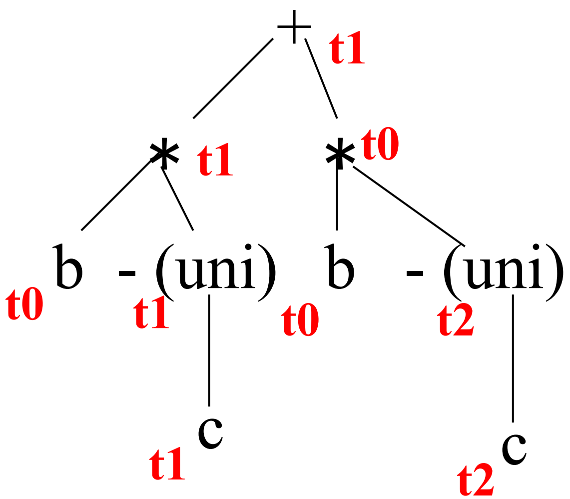

```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = b
t2 = -c
```

Example1: a := b *-c + b*-c



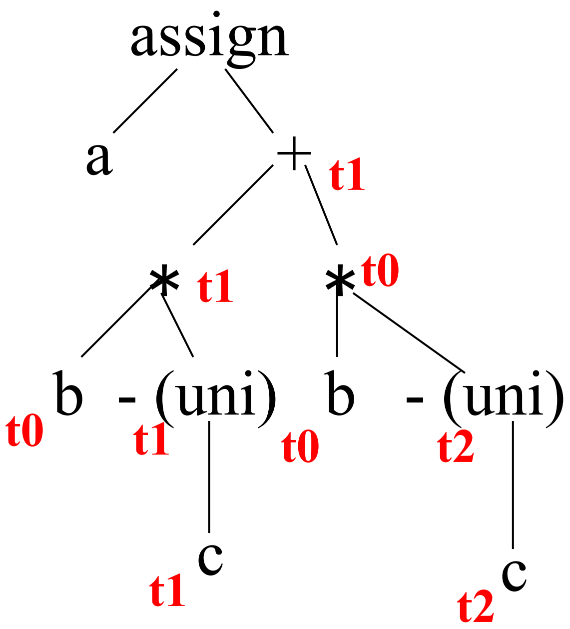

```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = b
t2 = -c
t0 = t0 * t2
```

Example1: a := b *-c + b*-c




```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = b
t2 = -c
t0 = t0 * t2
t1 = t0 + t1
```

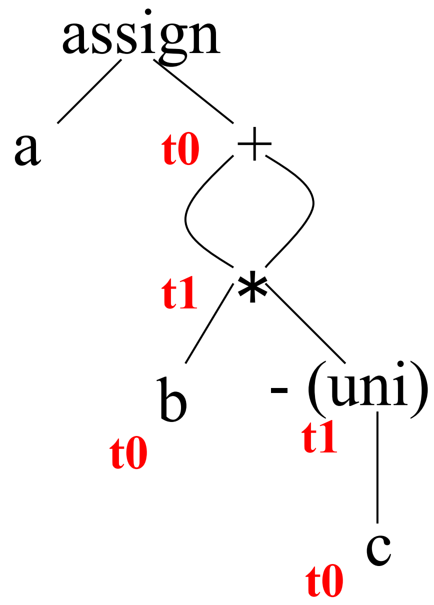
Example1: **a := b *-c + b*-c**




```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = b
t2 = -c
t0 = t0 * t2
t1 = t0 + t1
a = t1
```

Example1: **a := b *-c + b*-c**

```
t0 = b
t1 = -c
t1 = t1 * t0
t0 = t1 * t1
a = t10
```



Expressions

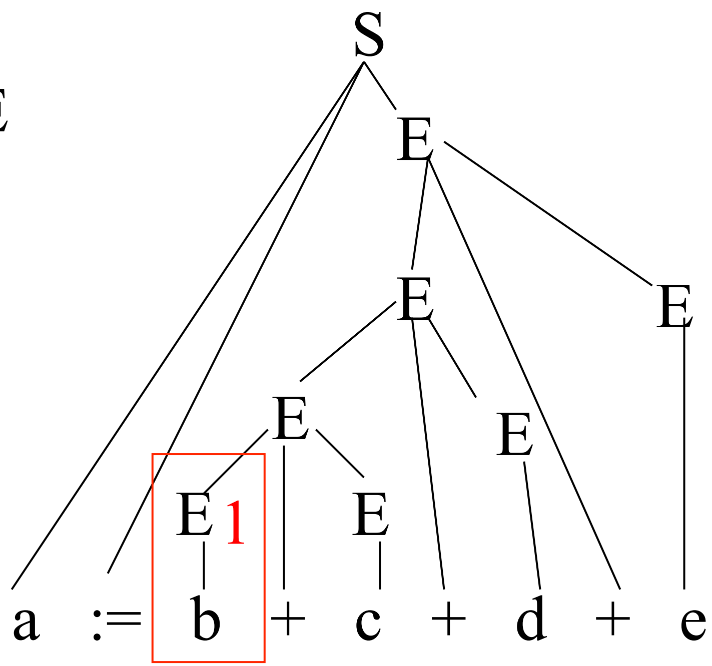
Example 2:

Grammar:

S → id := E

E → E + E

E → id



Generate:

```
t1 = b
```

Expressions

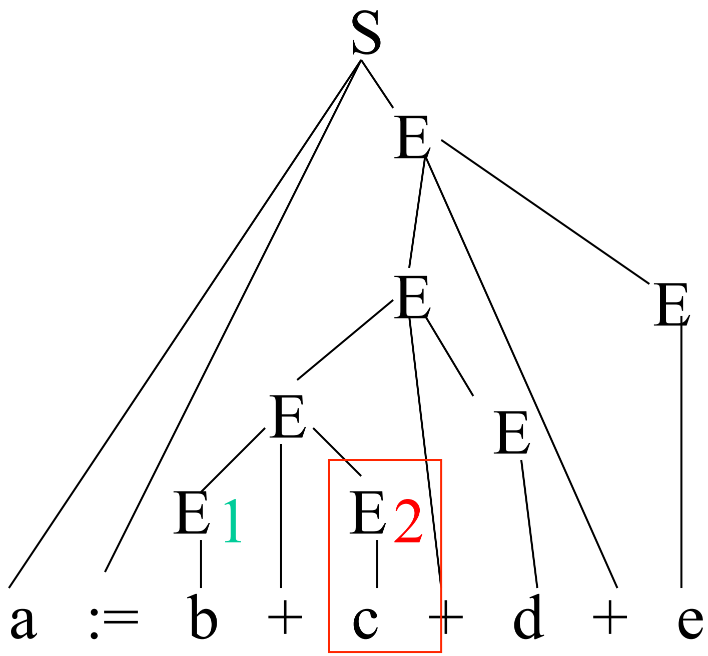
Example 2:

Grammar:

S → id := E

E → E + E

E → id




```
Generate:
 t1 = b
 t2 = c
```

Each number corresponds to a temporary variable.

Expressions

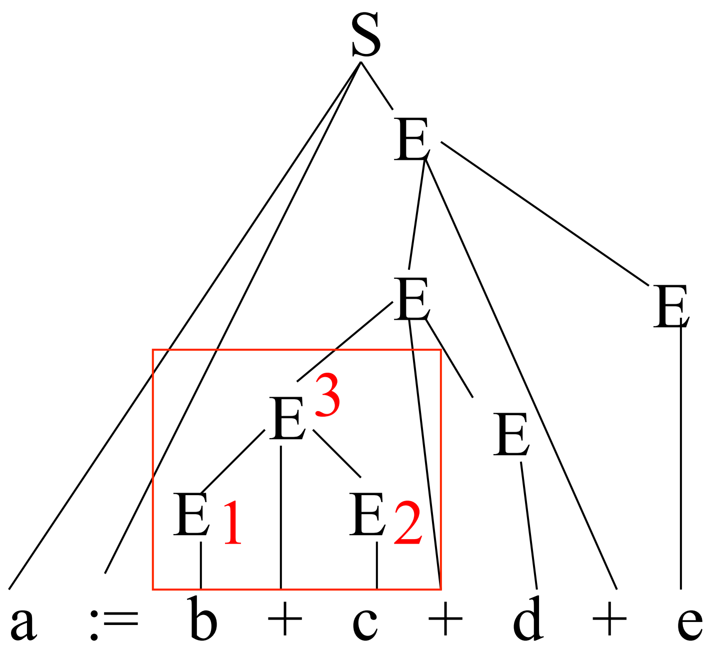
Example 2:

Grammar:

S → id := E

E → E + E

E → id



Generate:

```
t1 = b
t2 = c
t3 = t1 + t2
```

Each number corresponds to a temporary variable.

Expressions

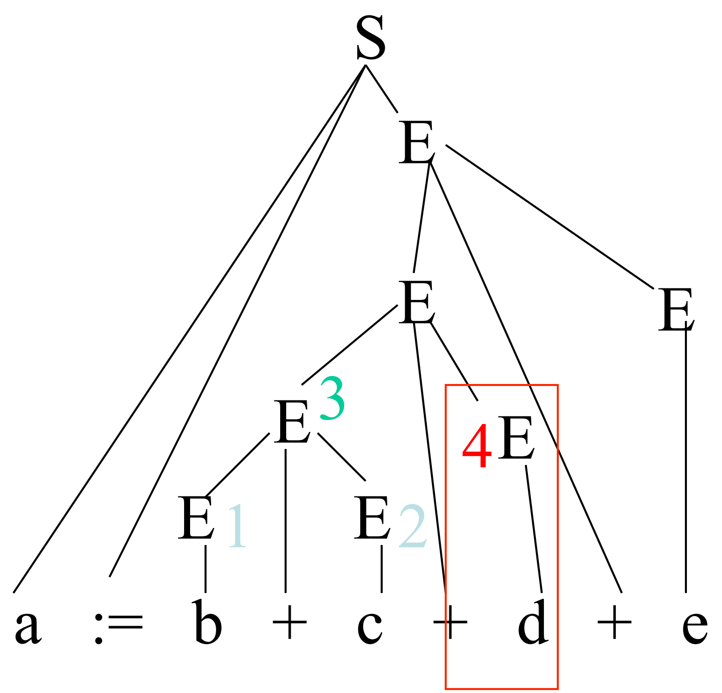
Example 2:

Grammar:

S → id := E

E → E + E

E → id



Generate:

```
t1 = b
t2 = c
t3 = t1 + t2
t4 = d
```

Each number corresponds to a temporary variable.

Expressions

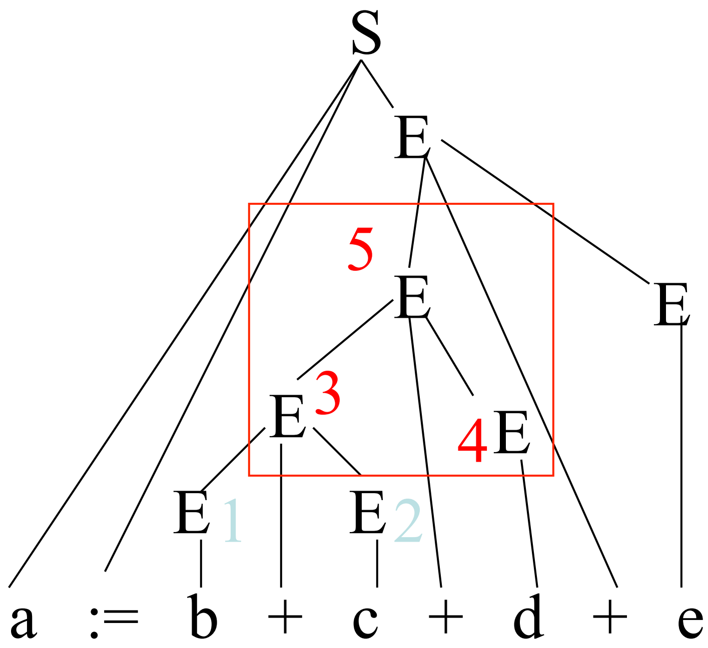
Example 2:

Grammar:

S → id := E

E → E + E

E → id



Generate:

```
t1 = b
t2 = c
t3 = t1 + t2
t4 = d
t5 = t3 + t4
```

Each number corresponds to a temporary variable.

Expressions

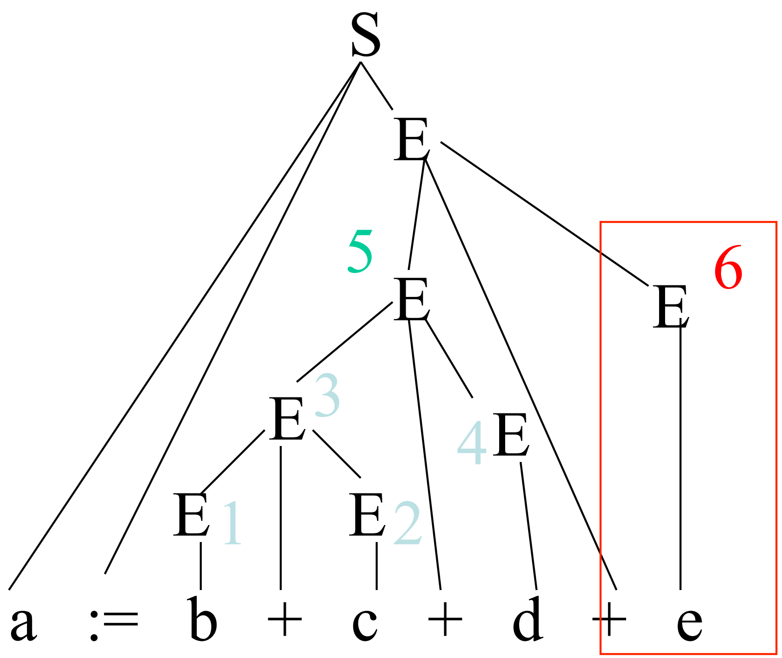
Example 2:

Grammar:

S → id := E

E → E + E

E → id




```
Generate:
  t1  =  b
  t2  =  c
  t3 = t1 + t2
  t4 = d
  t5 = t3 + t4
  t6  =  e
```

Each number corresponds to a temporary variable.

Expressions

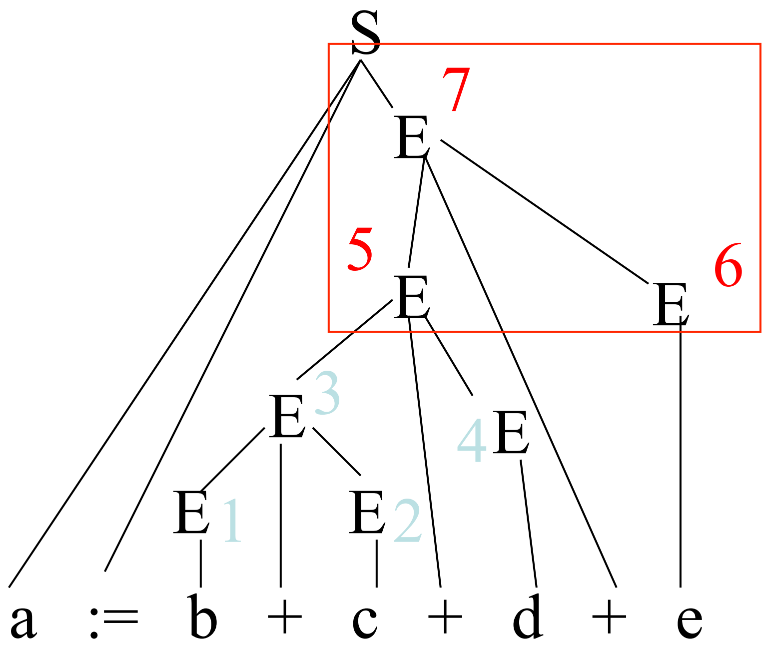
Example 2:

Grammar:

S → id := E

E → E + E

E → id



Each number corresponds to a temporary variable.

```
Generate:
  t1 = b
  t2 = c
  t3 = t1 + t2
  t4 = d
  t5 = t3 + t4
  t6 = e
  t7 = t5 + t6
```

Expressions

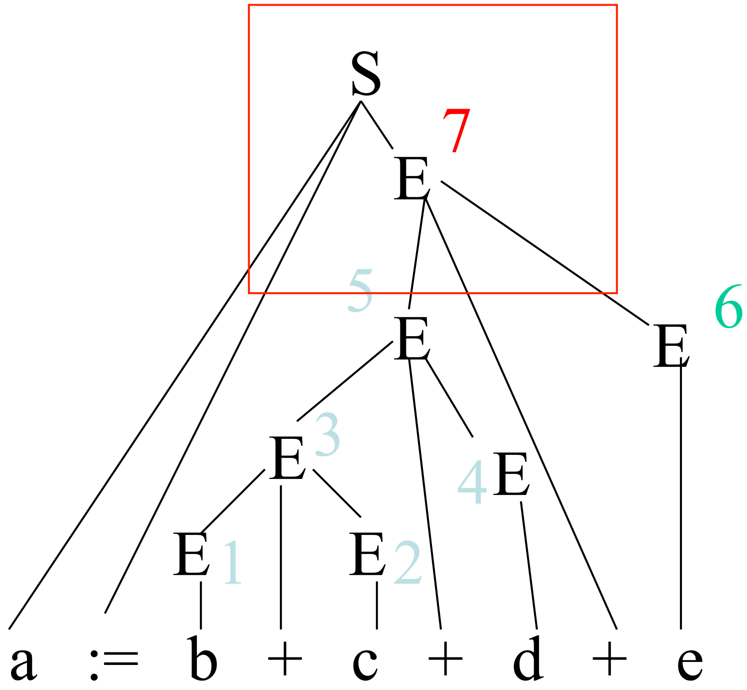
Example 2:

Grammar:

$S \rightarrow id := E$

$E \rightarrow E + E$

$E \rightarrow id$




```
Generate:
  t1 = b
  t2 = c
  t3 = t1 + t2
  t4 = d
  t5 = t3 + t4
  t6 = e
  t7 = t5 + t6
  a = t7
```

Each number corresponds to a temporary variable.

Other representations

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- The list goes on ......