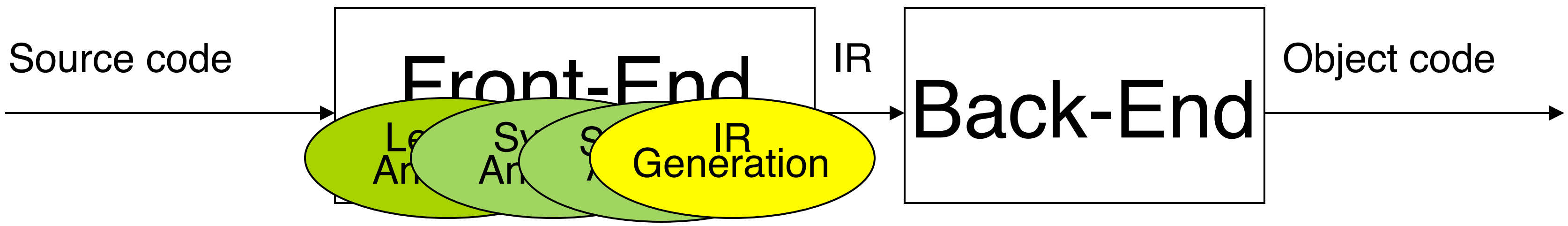
Compiler Design

Lecture 9:
Three-Address Code Generation

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Intermediate Representation Generation.



- IR Generation
  - Goal: Translate the program into the format expected by the compiler back-end.

Outline

- **Introduction**
- Syntax-Directed Translation
- Code Generation
- Representations
- More Structures of Code Generation

Steps

- Runtime Environments

- Three-Address Code IR

Intermediate Representation

- Intermediate code can be represented using different approaches:
  - Syntax tree
  - Postfix notation
  - Three-Address Code

Three-Address Code IR

- TAC can be used for different parts of the programming code:
  - TAC for simple expressions.
  - TAC for functions and function calls.
  - TAC for objects.
  - TAC for arrays.

Three-Address Code

- The IR that you will be using for the final programming project.
- High-level assembly where each operation has at most three operands.
- Uses explicit runtime stack for function calls.
- Uses vtables for dynamic dispatch.

Three-Address Code

x = y *op* z

- x, y, z are
  - Variables
  - Constants
  - Temporaries

- *op* is the operand

Sample TAC Code

```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```

Sample TAC Code

```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```

```
x2 = x * x;
y2 = y * y;
r2 = x2 + y2;
```

Temporary Variables

- The "three" in "three-address code" refers to the number of operands in any instruction.
- In the left hand side only one operand is allowed
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.

Sample TAC Code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

Sample TAC Code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

R-value and L-value

- The identifiers in right hand side and left hand of an assignment have different meanings
  - R-value is the exact value of the identifier
  - L-value is the place for holding that identifier

TAC Instructions

- Instructions:
  - Assignments
  - Unconditional jump
  - Conditional jumps
  - Procedure calls
  - Return statement
  - Indexed assignments (Arrays)
  - Address assignments
  - Pointers assignments

TAC Instructions

- Assignments:
  - x = y op z (op: binary arithmetic or logical operation)
  - x = op y (op: unary operation)
  - x = y

TAC Instructions

- Unconditional jump:
  - goto L (L is a symbolic label of a statement)

- Conditional jumps:
  - if x goto L
  - ifFalse x goto L
  - if x relop y goto L (relop: relation operator: <,==,<=)

TAC Instructions

- Procedure calls: p(x1,x2,…,xn)
  - param x1
  - param x2
  - …
  - param xn
  - call p, n
- Return statement:
  - return y

TAC Instructions

- Indexed assignments (Arrays):
  - x = y[i]
  - x[i] = y
- Address assignments:
  - x = &y (which sets x to the location of y)
- Pointers assignments:
  - x = *y (y is a pointer, sets x to the value pointed by y)
  - *x = y

TAC Instructions (detailed)

- **Variable assignment** allows assignments of the form
  - var = constant;
  - var1 = var2;
  - var1 = var2 **op** var3;
  - var1 = constant **op** var2;
  - var1 = var2 **op** constant;
  - var = constant1 **op** constant2;
- Permitted operators are +, **-**, *, /, **%**.

TAC Instructions

- Defining permitted commands and operands in the intermediate code generation is an important challenge
  - The smaller the operand set the easier to implement it in a machine
  - But it needs more commands to generate the corresponding IC
    - => It makes IC optimization and target code generation more difficult

TAC Instructions

- **Variable assignments:**
  - var1 = **op** var2;
  - var1 = **op** constant;

- How would you compile **y = -x;** without the above instructions?

TAC Instructions

- **Variable assignments:**
  - var1 = **op** var2;
  - var1 = **op** constant;

- How would you compile **y = -x;** without the above instructions?

  y = 0 – x;   y = -1 * x;

Postfix Notation

- The postfix notation for the statement E is defined as follows
  - If E is a variable or constant,

    the postfix notation is E
  - If E is in the form of E1 op E2

    the postfix notation is $E1^p$ $E2^p$ op
  - If E is in the form of (E1)

    the postfix notation of E is the same as the postfix notation of E1

Postfix Notation

- Using the postfix notation, the order of operations can be distinguished even without ()

- Example
  - (a - b) + c

  - a – (b + c)

Postfix Notation

- Using the postfix notation, the order of operations can be distinguished even without ()

- Example
  - (a - b) + c

    a b – c +
  - a – (b + c)

    a b c + -

Outline

- Introduction
- **Syntax-Directed Translation**
- Code Generation
- Representations
- More Structures of Code Generation

Syntax-Directed Translation

- The translation of languages guided by context-free grammars
  - Considering the grammar rules of the source language
  - Defining the semantic analysis of each rule
  - Defining the IC generation of each rule

Attributes

- To describe the translation in SDT, each symbol of the grammar is associated with a set of *attributes*
- Attributes may be of any kind
- Example:
  - E.code
  - E.type
  - E.val
  - E.place

Semantic Rules

- Semantic rules are defined for each grammar rule
- The value of attributes are specified by the semantic rules
- Defining semantic rules for grammar rules are done using two approaches
  - Syntax-directed definition
  - Translation scheme

Syntax-directed Definition

- Syntax-directed definition is a high level definition of specifying value of attributes
- It contains no detail and implementation
- The order of translation steps are not necessary to be defined

Translation Scheme

- In contrast, translation scheme includes the order of semantic rules
- Therefor more detail of implementation is considered in translation scheme

Semantic Rules

- Syntax-directed definitions
  - More readable
  - More useful for specifications
- Translation scheme
  - More efficient
  - More useful for implementations
- Both of the approaches will be used in this lecture

Syntax-directed Translation

- Constructing a parse tree or a syntax tree
- Computing the values of attributes at the nodes of the tree by visiting the nodes of the tree
- In many cases, translation can be done during parsing, without building an explicit tree

Syntax-Directed Definitions

- A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules
- Attributes are associated with grammar symbols
- Rules are associated with productions
- If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X.

Attributes

- We shall deal with two kinds of attributes for nonterminals:
  - Inherited attributes
  - Synthesized attributes

- They are defined based on the difference between the computation of their values.

Synthesized Attributes

- A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.
- The production must have A as its head.
- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

Inherited Attributes

- An *inherited attribute* for a nonterminal B at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N*.
- The production must have *B* as a symbol in its body.
- An inherited attribute at node *N* is defined only in terms of attribute values at N's parent, *N* itself, and *N's* siblings.

Inherited and Synthesized Attributes

- Since the synthesized attributes can be computed with a bottom-up traverse of a parse tree, it is more useful for compiler construction.

SDD Example

- In the SDD, each of the nonterminals has a single synthesized attribute, called *val*
- The terminal **digit** has a synthesized attribute *lexval,* which is an integer value returned by the lexical analyzer.

PRODUCTION	SEMANTIC RULES
L → E	*L.Val = E.val*
E → E_1 + T	*E.Val = E1.val + T.val*
E → T	*E.Val = T.val*
T → T_1 * F	*T.Val = T1.val* x *F.val*
T → F	*T.Val = F.val*
F → (E)	*F.Val = E.val*
F → digit	*F.Val =* **digit**.lexval

Syntax-directed Translation

- "L-attributed translations" (L for left-to-right)
  - A class of SDT
  - Encompass virtually all translations that can be performed during parsing
- "S-attributed translations" (S for synthesized)
  - A smaller class of SDT
  - Can be performed easily in connection with a bottom-up parse

Syntax-directed Definitions

- An SDD that involves only synthesized attributes is called *S-attributed*
- The SDD of the previous example has this property.
- In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.
- An S-attributed SDD can be implemented naturally in conjunction with an LR parser.

Syntax-Directed Definitions

- In practice, it is convenient to allow SDD's to have limited side effects
  - For example, printing the result computed by a desk calculator
- An SDD without side effects is sometimes called an *attribute grammar.*
- The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Syntax-Directed Definitions

- In case of an SDD with side effect, the first rule of the previous example can be change as follows:

PRODUCTION	SEMANTIC RULES
L → E	*L.Val = E.val*

PRODUCTION	SEMANTIC RULES
E	*Print (E.Val)*

- In production 1, the program prints the value *E.val* as a side effect, instead of defining the attribute *L.val*.

SDD for Postfix Notations

- Syntax-directed definitions can also translate grammar rules to postfix notations

PRODUCTION	SEMANTIC RULES
Exp → Exp_1 + Term	Exp.t = Exp_1.t || Term.t || '+'
Exp → Exp_1 - Term	Exp.t = Exp_1.t || Term.t || '-'
Exp → Term	Exp.t = Term.t
Term → 0	Term.t = '0'
Term → 1	Term.t = '1'
…	…
Term → 9	Term.t = '9'

't': The string used for representing postfix notation

||: is used for appending strings

TS for Postfix Notations

- Translation scheme can also do the same

PRODUCTION and SEMANTIC RULES
Exp → Exp_1 + Term { print ('+') }
Exp → Exp_1 - Term { print ('-') }
Exp → Term
Term → 0 { print ('0') }
Term → 1 { print ('1') }
…
Term → 9 { print ('9') }

Annotated Parse Tree

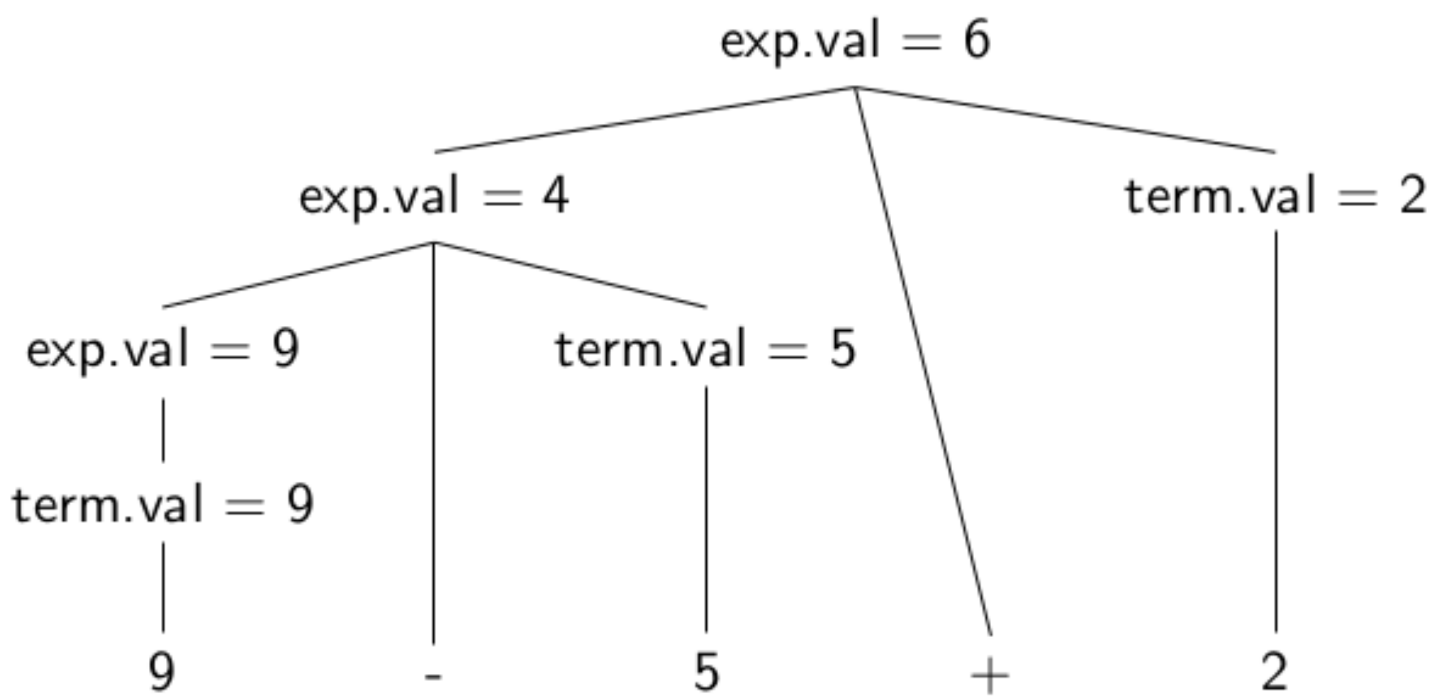
- Parse trees are very useful to visualize the translation specified by an SDD
  - Even though a translator need not actually build a parse tree.
- Therefore, the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree.*

Annotated Parse Tree

- If all attributes are synthesized, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom-up order
  - E.g., a post-order traversal of the parse tree

Example

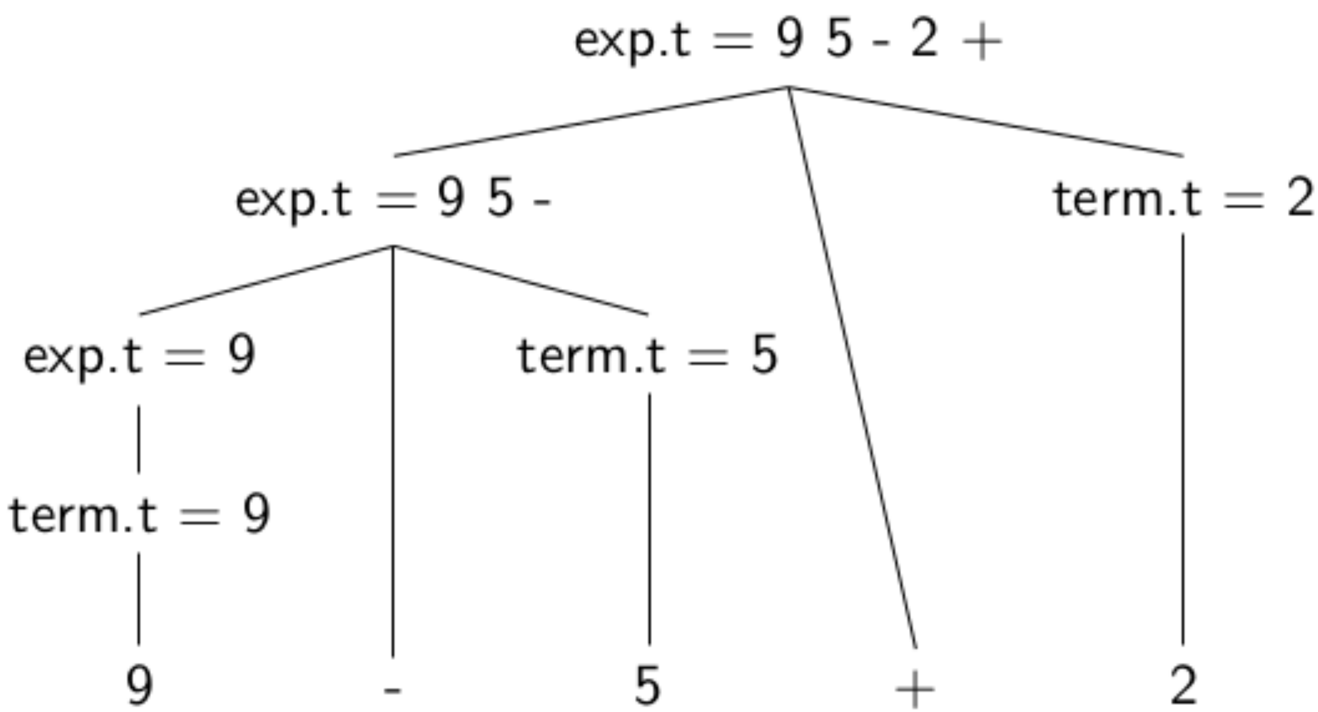
- 9 – 5 + 2



$\mathbf{digit}.lexval = 3$

Example

- $9 - 5 + 2$



$\mathbf{digit}.lexval = 3$

Annotated Parse Tree to Postfix Notation

- Traversing the parse tree using DFS
- Applying semantic rules of each subtree
  - Do no thing when we reach a terminal
  - Print the terminal after traversing both left and right subtrees

- Example
  - 9 5 – 2 +

Example

- Use the grammar rules in slide 40:
  - (3 + 4) * (5 + 6)
  - 1 * 2 * 3 * (4 + 5)
  - (9 + 8 * (7 + 6) + 5) * 4

Example

- Use the grammar rules in slide 40:

  - (3 + 4) * (5 + 6)

    3 4 + 5 6 + *

  - 1 * 2 * 3 * (4 + 5)

    1 2 * 3 * 4 5 + *

  - (9 + 8 * (7 + 6) + 5) * 4

    9 8 7 6 + * + 5 + 4 *

Outline

- Introduction
- Syntax-Directed Translation
- **Code Generation**
- Representations
- More Structures of Code Generation

Code Generation

- The syntax-directed definition builds up the three-address code for an assignment statement S using attributes addr and code
  - Attribute E.code denote the three-address code for E
  - Attribute E.addr denotes the address that will hold the value of E
    - An address can be a name, a constant, or a compiler-generated temporary

Code Generation for Assignment Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$	S. code = E.code || gen(top.get(id.lexeme) '= E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp () E.code = E_1.code || E_2.code || gen(E.addr '='E_1.addr '+' E_2.addr)
$E \rightarrow E_1 * E_2$	E.addr = new Temp () E.code = E_1.code || E_2.code || gen(E.addr '='E_1.addr '*' E_2.addr)
$E \rightarrow -E_1$	E.addr = new Temp () E.code = E_1.code || gen(E.addr '=' 'uminus' E_1.addr)
$E \rightarrow (E_1)$	E.addr = E_1.addr E. code = E_1.code
$E \rightarrow id$	E.addr = id.addr E.code = ''

Code Generation

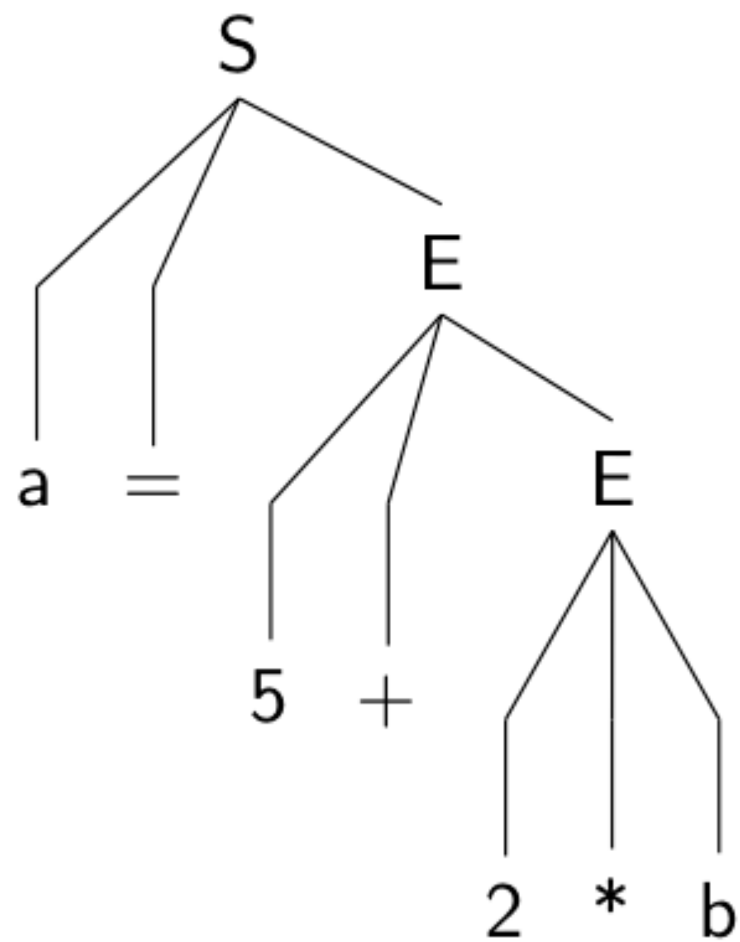
- Creating temporal variables for middle nodes of the parse tree

Code Generation

- Example:
  - a = 5 + 2 * b
  - Try syntax tree and abstract syntax tree

Code Generation

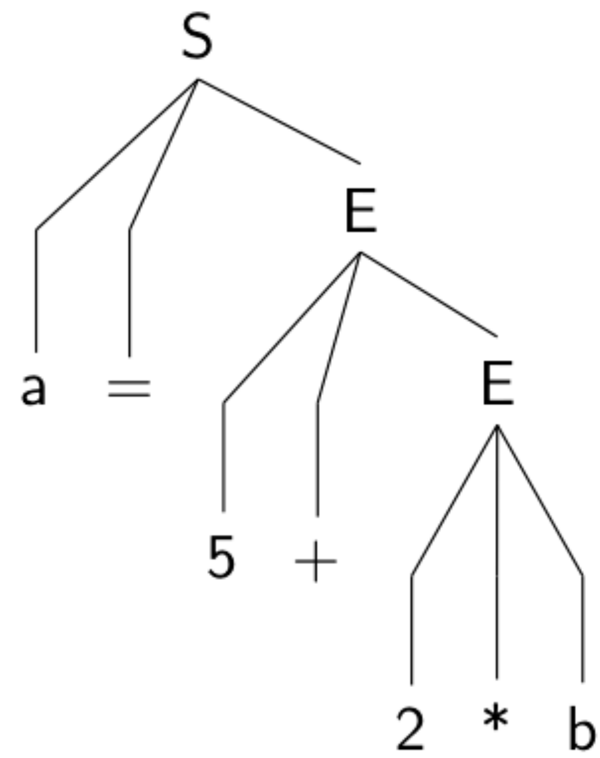
- Example:
  - a = 5 + 2 * b



Code Generation

- Example:
  - a = 5 + 2 * b

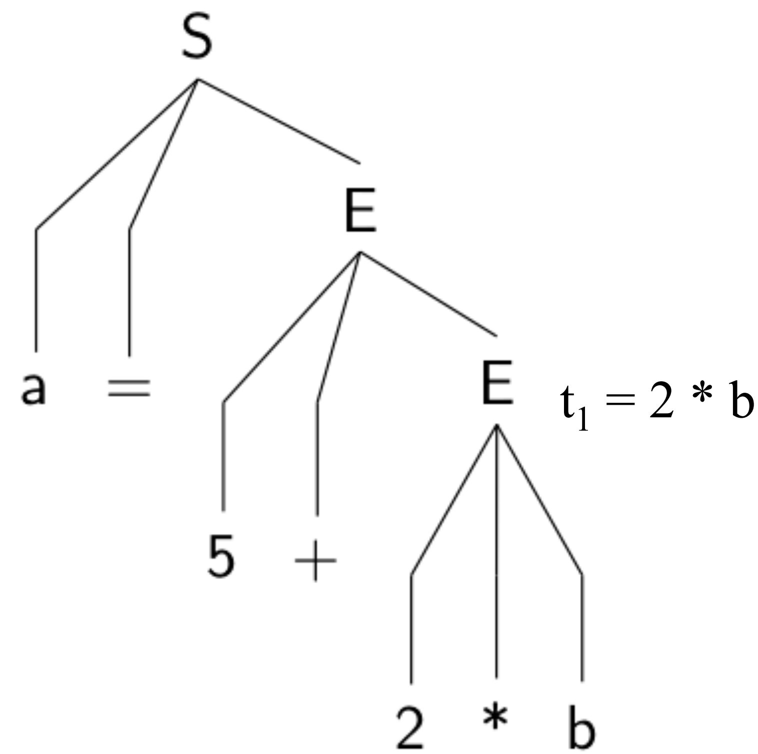
PRODUCTION	SEMANTIC RULES
S -> id = E	S. code = E.code || gen(top.get(id.lexeme) '= E.addr)
E —> E_1 + E_2	E.addr = new Temp () E.code = E_1.code II E_2.code II gen(E.addr '='E_1.addr '+' E_2.addr)
E —> E_1 * E_2	E.addr = new Temp () E.code = E_1.code II E_2.code II gen(E.addr '='E_1.addr '*' E_2.addr)
E —> -E_1	E.addr = new Temp () E.code = E_1.code II gen(E.addr '=' 'uminus' E_1.addr)
E —> (E_1)	E.addr = E_1.addr E. code = E_1.code
E —> id	E.addr = id.addr E.code = ''



Code Generation

- Example:
  - a = 5 + 2 * b

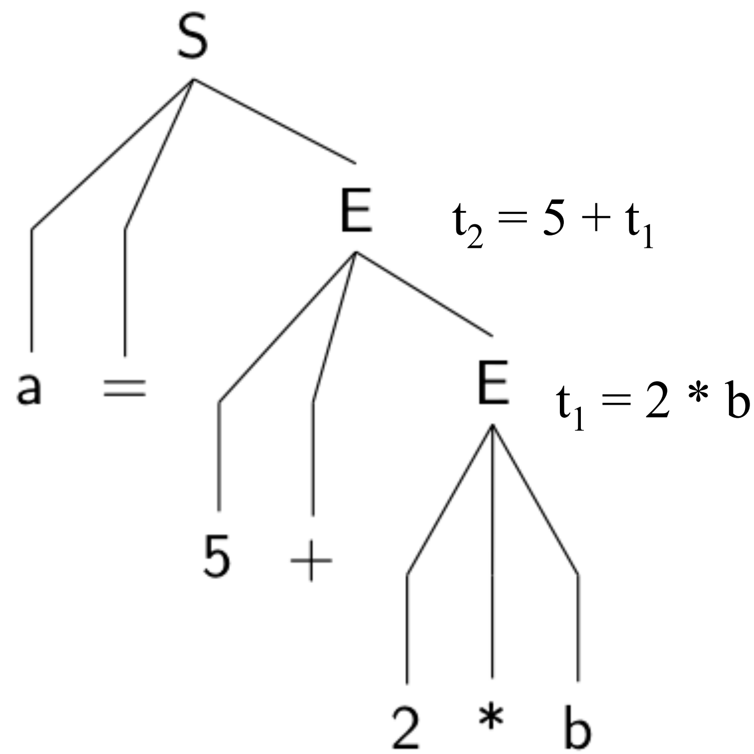
PRODUCTION	SEMANTIC RULES
S -> id = E	S. code = E.code || gen(top.get(id.lexeme) '= E.addr)
E —> E_1 + E_2	E.addr = new Temp () E.code = E_1.code ll E_2.code ll gen(E.addr '='E_1.addr '+' E_2.addr)
E —> E_1 * E_2	E.addr = new Temp () E.code = E_1.code ll E_2.code ll gen(E.addr '='E_1.addr '*' E_2.addr)
E —> -E_1	E.addr = new Temp () E.code = E_1.code ll gen(E.addr '=' 'uminus' E_1.addr)
E —> (E_1)	E.addr = E_1.addr E. code = E_1.code
E —> id	E.addr = id.addr E.code = ''



Code Generation

- Example:
  - a = 5 + 2 * b

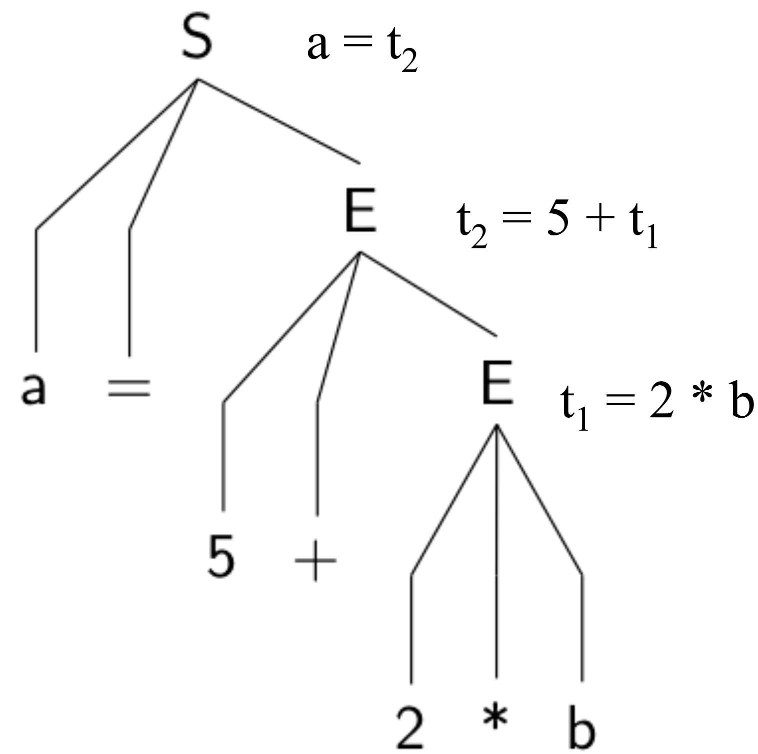
PRODUCTION	SEMANTIC RULES
S -> id = E	S. code = E.code || gen(top.get(id.lexeme) '= E.addr)
E —> E_1 + E_2	E.addr = new Temp () E.code = E_1.code II E_2.code II gen(E.addr '='E_1.addr '+' E_2.addr)
E —> E_1 * E_2	E.addr = new Temp () E.code = E_1.code II E_2.code II gen(E.addr '='E_1.addr '*' E_2.addr)
E —> -E_1	E.addr = new Temp () E.code = E_1.code II gen(E.addr '=' 'uminus' E_1.addr)
E —> (E_1)	E.addr = E_1.addr E. code = E_1.code
E —> id	E.addr = id.addr E.code = ''



Code Generation

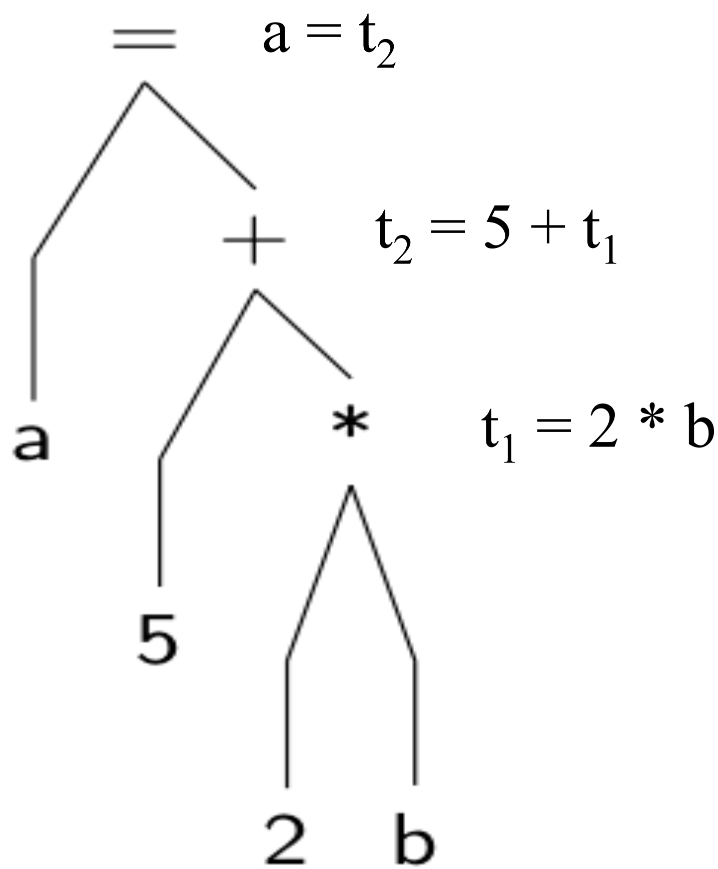
- Example:
  - $a = 5 + 2 * b$

PRODUCTION	SEMANTIC RULES
S -> id = E	S. code = E.code || gen(top.get(id.lexeme) '= E.addr)
E —> E_1 + E_2	E.addr = new Temp () E.code = E_1.code ll E_2.code ll gen(E.addr '='E_1.addr '+' E_2.addr)
E —> E_1 * E_2	E.addr = new Temp () E.code = E_1.code ll E_2.code ll gen(E.addr '='E_1.addr '*' E_2.addr)
E —> -E_1	E.addr = new Temp () E.code = E_1.code ll gen(E.addr '=' 'uminus' E_1.addr)
E —> (E_1)	E.addr = E_1.addr E. code = E_1.code
E —> id	E.addr = id.addr E.code = ''



Code Generation

- Example:
  - $a = 5 + 2 * b$



Code Generation

- Example:
  - a = 5 + 2 * b

$t_1 = 2 * b$

$t_2 = 5 + t_1$

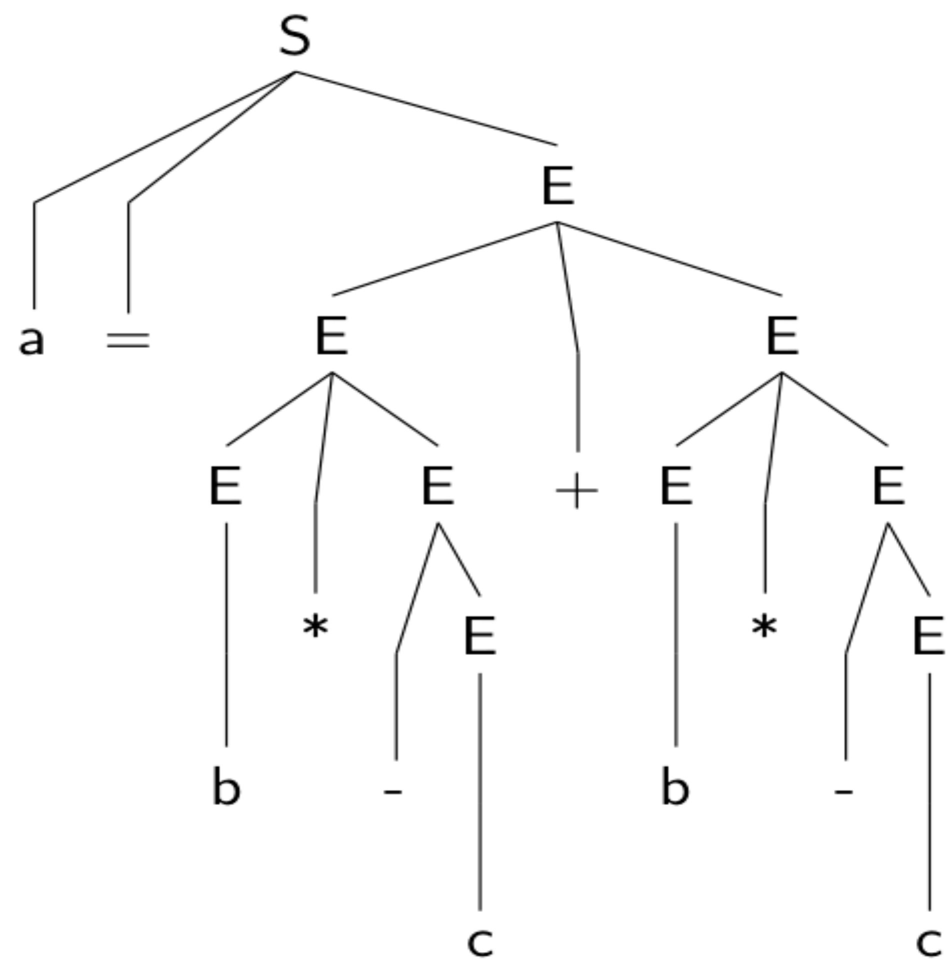
$a = t_2$

Code Generation

- Example:
  - a = b * -c + b * -c
  - Try syntax tree and abstract syntax tree

Code Generation

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



Code Generation

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

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

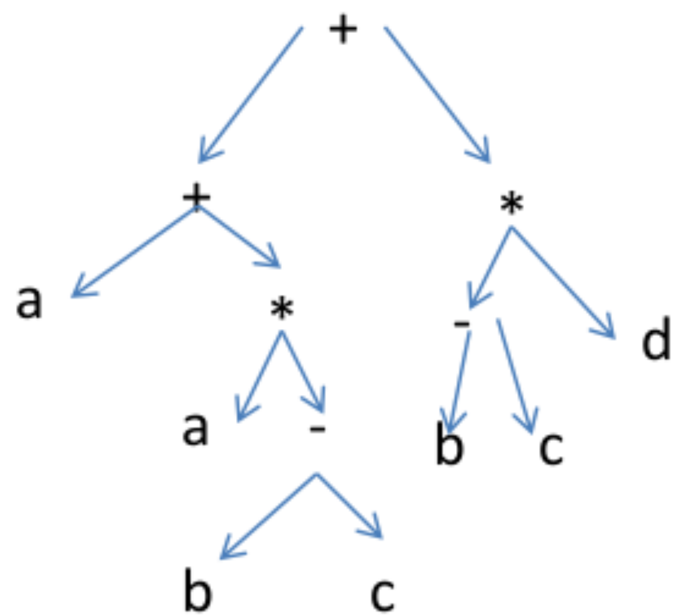
$a = t_5$

Code Generation

- Example:
  - a + a * (b – c) + (b – c) * d
  - Try syntax tree and abstract syntax tree and DAG

Code Generation

- Example:
  - a + a * (b – c) + (b – c) * d



Code Generation

- Example:
  - a + a * (b – c) + (b – c) * d

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = b - c$

$t_5 = t_4 * d$

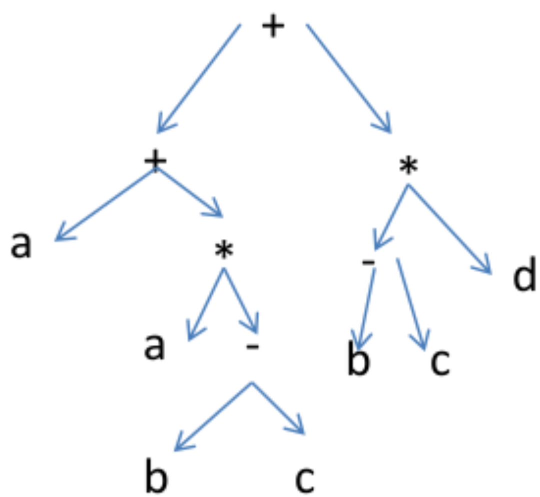
$t_6 = t_3 + t_5$

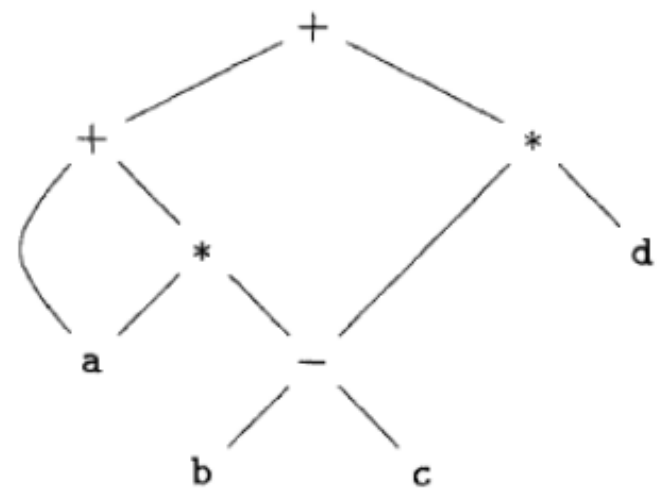
Directed Acyclic Graph (DAG)

- Node can have more than one parent
- Advantages:
  - More compact representation
  - Gives clues regarding generation of efficient code
- All what is needed is that check whether a node already exists. If such a node exists, a pointer is returned to that node.

Directed Acyclic Graph (DAG):

- DAG for the previous Example





Code Generation

- Example:
  - a + a * (b – c) + (b – c) * d

$t_1 = b - c$	$t_1 = b - c$
$t_2 = a * t_1$	$t_2 = a * t_1$
$t_3 = a + t_2$	$t_3 = a + t_2$
$t_4 = b - c$	$t_4 = t_1 * d$
$t_5 = t_4 * d$	$t_5 = t_3 + t_4$
$t_6 = t_3 + t_5$	

Directed Acyclic Graph (DAG):

- Example
  - $((x+y)-((x+y)*(x-y)))+(z*(x-y))$

Outline

- Introduction
- Syntax-Directed Translation
- Code Generation
- **Representations**
- More Structures of Code Generation

Representations

- The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure.
- In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands.
- Three such representations are called
  - “quadruples”
  - “triples”
  - “indirect triples”

Quadruples

- A quadruple (or just "quad!") has four fields, which we call *op*, arg_1, arg_2, and *result*.
- The *op* field contains an internal code for the operator.
- Example: x = y + z is represented by placing
  - \+ in *op*
  - *y* in arg_1
  - *z* in arg_2
  - *x* in *result*

Quadruples

- The following are some exceptions to this rule:
  - Instructions with unary operators like $x = minus\ y$ or $x = y$ do not use arg_2.
    - Note that for a copy statement like $x = y$, op is $=$, while for most other operations, the assignment operator is implied.
  - Operators like param use neither arg_2 nor *result*.
  - Conditional and unconditional jumps put the target label in *result*.

Quadruples

t1 = - c

t2 = b * t1

t3 = - c

t4 = b * t3

t5 = t2 + t4

a = t5

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

- A triple has only three fields, called *op, arg$_1$*, and *arg$_2$*
  - Note that the *result* field in quadruples is used primarily for temporary names.
- Using triples, we refer to the *result* of an operation *x op y* by its position, rather than by an explicit temporary name.

  => Thus, instead of the temporary t_i in quadruples, a triple representation would refer to position (0).
- Parenthesized numbers represent pointers into the triple structure itself.
- Positions or pointers to positions are called value numbers.

Triples

1. - c
2. b * (1)
3. - c
4. b * (3)
5. (2) + (4)
6. a = (5)

	op	arg1	arg2
(1)	minus	c	
(2)	*	b	(1)
(3)	minus	c	
(4)	*	b	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Quadruples vs Triples

- A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around.
  - With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change.
  - With triples, the *result* of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.
- This problem does not occur with indirect triples.

Indirect Triples

- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.
- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.
- When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

Indirect Triples

1. - c
2. b * (1)
3. - c
4. b * (3)
5. (2) + (4)
6. a = (5)

Instruction List:

35	(1)
36	(2)
37	(3)
38	(4)
39	(5)
40	(6)

	op	arg1	arg2
1	minus	c	
2	*	b	(1)
3	minus	c	
4	*	b	(3)
5	+	(2)	(4)
6	=	a	(5)

Example

- Example:
  - a + a * (b – c) + (b – c) * d

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

Example

- Example:
  - a + a * (b – c) + (b – c) * d

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

op	arg1	arg2	result
-	b	c	t1
*	a	t1	t2
+	a	t2	t3
*	t1	d	t4
+	t3	t4	t5

Example

- Example:
  - a + a * (b – c) + (b – c) * d

$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

op	arg1	arg2
-	b	c
*	a	(1)
+	a	(2)
*	(1)	d
+	(3)	(4)

Outline

- Introduction
- Syntax-Directed Translation
- Code Generation
- Representations
- **More Structures of Code Generation**