Syntax directed translation, Intermediate code generation, Code generation

Module 5

Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2

Content

- Syntax Directed Definition
- Evaluation orders for SDD s
- Applications of syntax directed translations
- Variants of syntax trees
- Three address code
- Issues in the design of code generators
- The target language

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A **syntax directed translation** is done by attaching rules or program fragments to production in a grammar.
- A **syntax directed definition** specifies the values of attributes by associating semantic rules with the grammar productions.

Production	**Semantic Rule**
E->E_1+T	E.code=E1.code||T.code||'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E_1+T {print '+'}

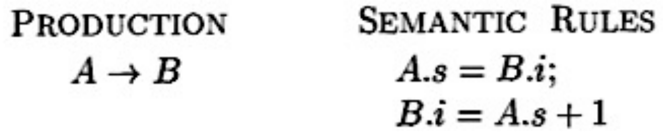
- The general approach to SDT is to construct the parse tree, and then compute the values of attributes at the nodes of the tree by visiting that nodes of the tree.
- 2 class of syntax directed translations
  - **L-Attributed translation**: encompass all translations that can be performed during parsing .
  - **S-Attributed translation** : performed easily in connection with bottom up parsers.

Syntax Directed Definitions

- A **SDD** is a context free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- 2 kinds of attributes for nonterminals.
- **Synthesized attributes**
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N itself.
- **Inherited attributes**
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Evaluating an SDD at the nodes of a Parse Tree

- A parse tree that shows the values of attributes is called ***annotated parse tree.***
- With synthesized attributes we can evaluate attributes in any bottom up order , post order traversal of the parse tree; The evaluation of S attributed definition.
- For SDDs with both inherited and synthesized attributes , there is no guarantee that there is even one order in which to evaluate attributes at nodes.



PRODUCTION	SEMANTIC RULES
$A \to B$	$A.s = B.i;$ $B.i = A.s + 1$

Evaluating an SDD at the nodes of a Parse Tree

Production	**Semantic Rules**
L->E n	L.val=E.val
E->E_1+T	E.val=E_1.val+T.val
E->T	E.val=T.val
T->T_1*F	T.val=T_1.val*F.val
T->F	T.val=F.val
F->(E)	F.val=E.val
F->digit	F.val=digit.lexval

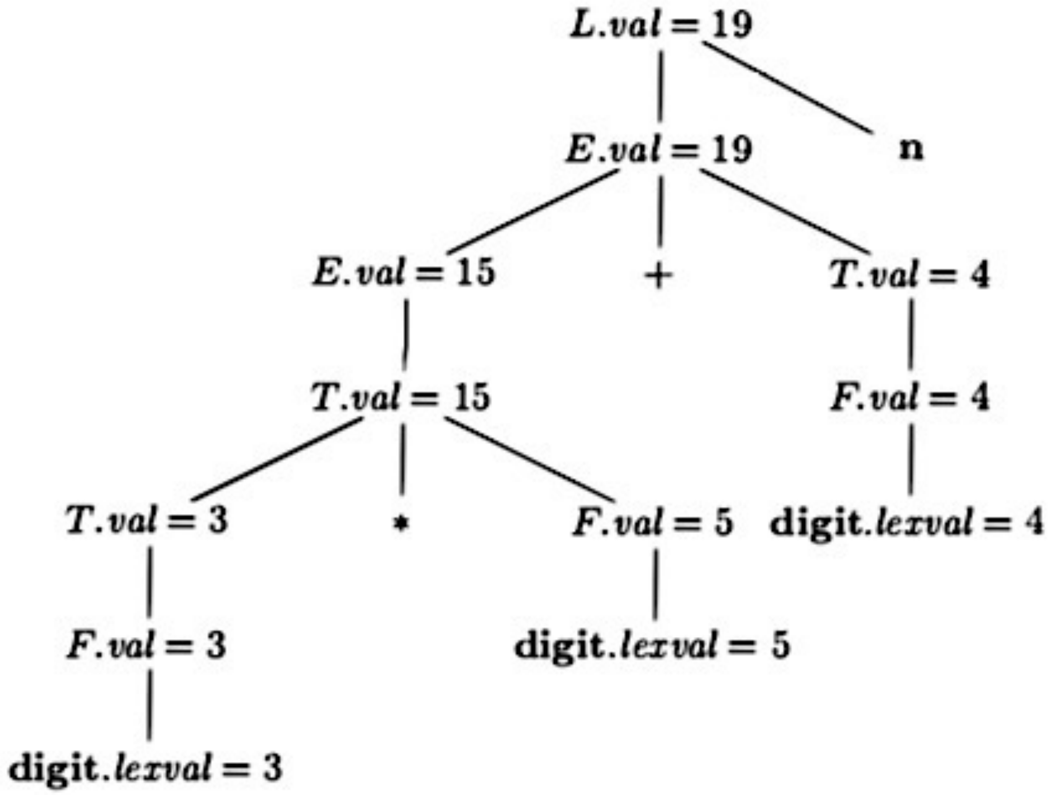


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

	Production	Semantic Rules
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * \, F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

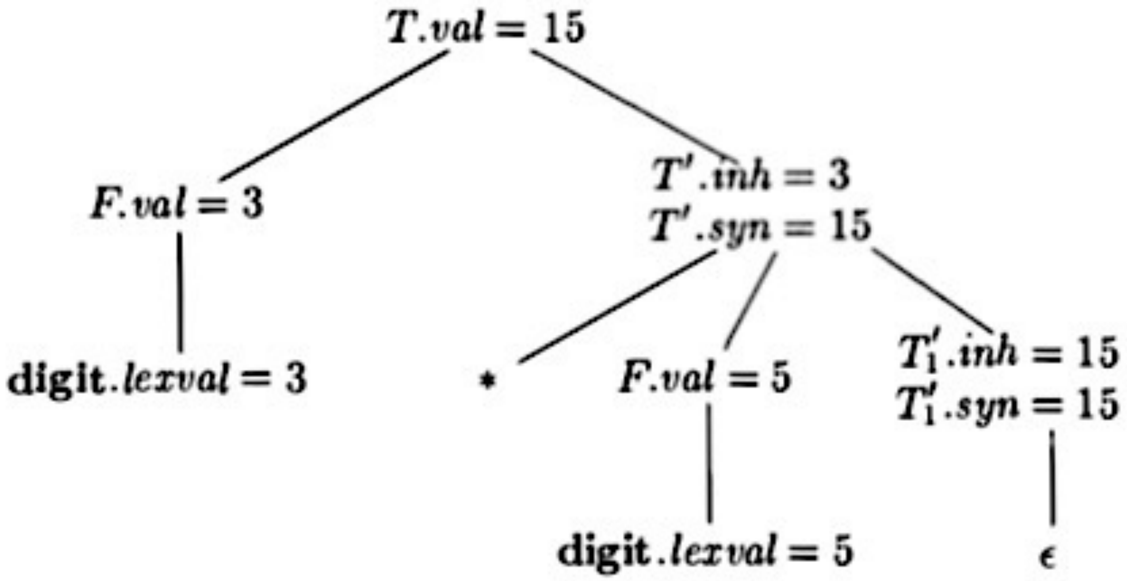


Figure 5.5: Annotated parse tree for $3 * 5$

Evaluation orders for SDD's

- A **dependency graph** is used to determine the order of computation of attributes.
- Dependency graph depicts the flow of information among the attribute instances in a particular parse tree.
- An edge from one attribute instance to another means that the value of the first is needed to compute the second.
- Dependency graph
  - For each parse tree node, a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.a to B.c

PRODUCTION	**SEMANTIC RULE**
E-> E_1 + T	E.val= E_1.val + T.val

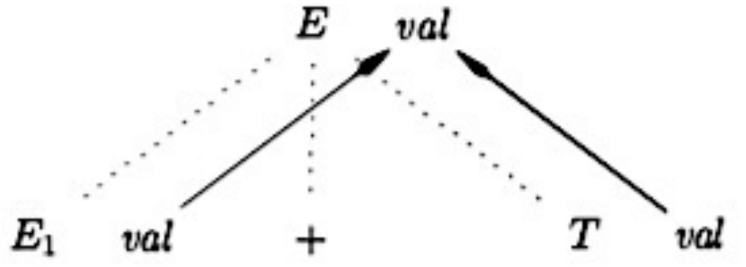


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

	Production	Semantic Rules
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to * \, F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

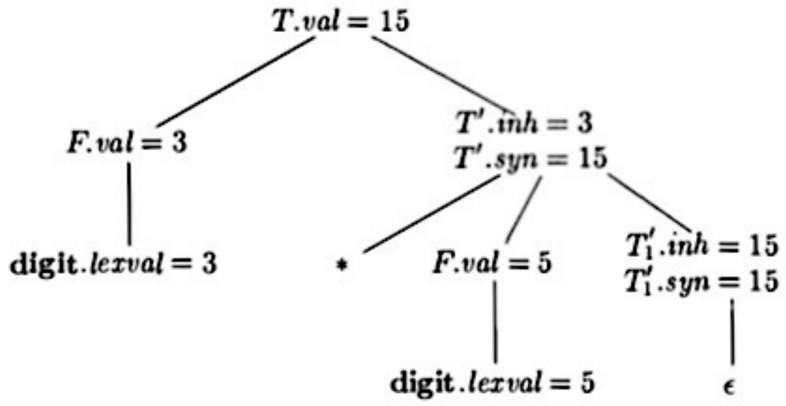
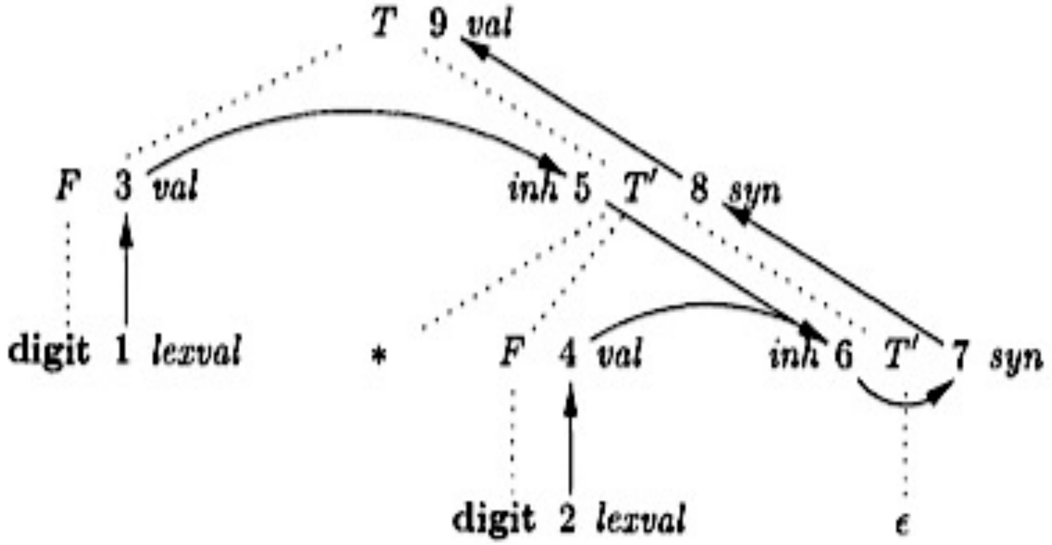


Figure 5.5: Annotated parse tree for $3 * 5$



Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes $N_1,N_2,\ldots,N_k$ such that if there is an edge from N_i to N_j then $i<j$
- Such an ordering is called a topological sort of a graph.
- One topological sort is the order in which the nodes have been numbered: 1, 2, 3….9
- Other topological sort is 1,3,5,2,4,6,7,8,9

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$
		$addType($**id**$.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType($**id**$.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declara

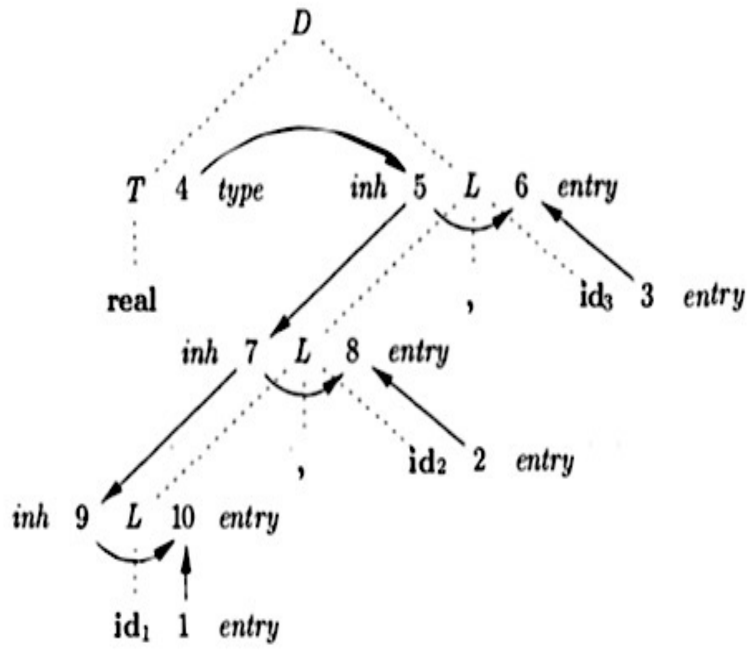


Figure 5.9: Dependency graph for a declaration **float id**$_1$, **id**$_2$, **id**$_3$

S-Attributed definitions

- An SDD is S-attributed if every attribute is synthesized.
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
    for (each child C of N, from the left) postorder(C);
    evaluate the attributes associated with node N;
}
```

- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there is a production A->$X_1X_2…X_n$ and there is an inherited attribute X_i.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Type checking and intermediate code generation.

Construction of syntax trees

- The nodes of a syntax tree are implemented as objects with a suitable number of fields.
- Each object will have an *op* field that is the label of the node.
- The additional fields as follows
  - Leaf nodes: *Leaf(op,val)*
  - Interior node: *Node(op,c1,c2,…,ck)*

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow$ **id**	$T.node$ = **new** $Leaf$(**id**, **id**.$entry$)
6) $T \rightarrow$ **num**	$T.node$ = **new** $Leaf$(**num**, **num**.val)

Figure 5.10: Constructing syntax trees for simple expressions

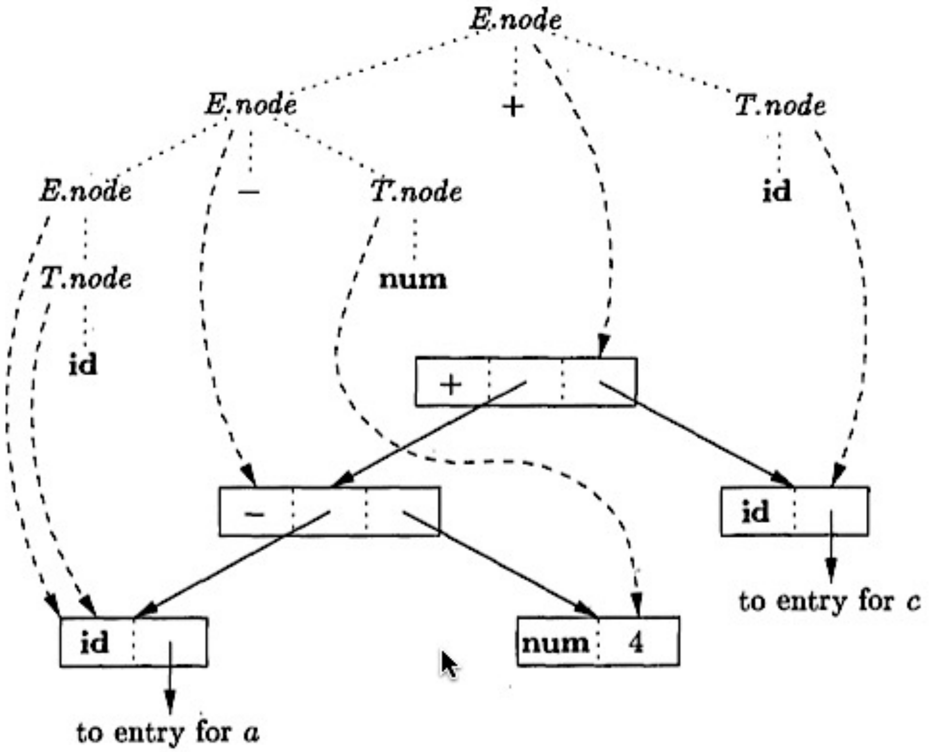
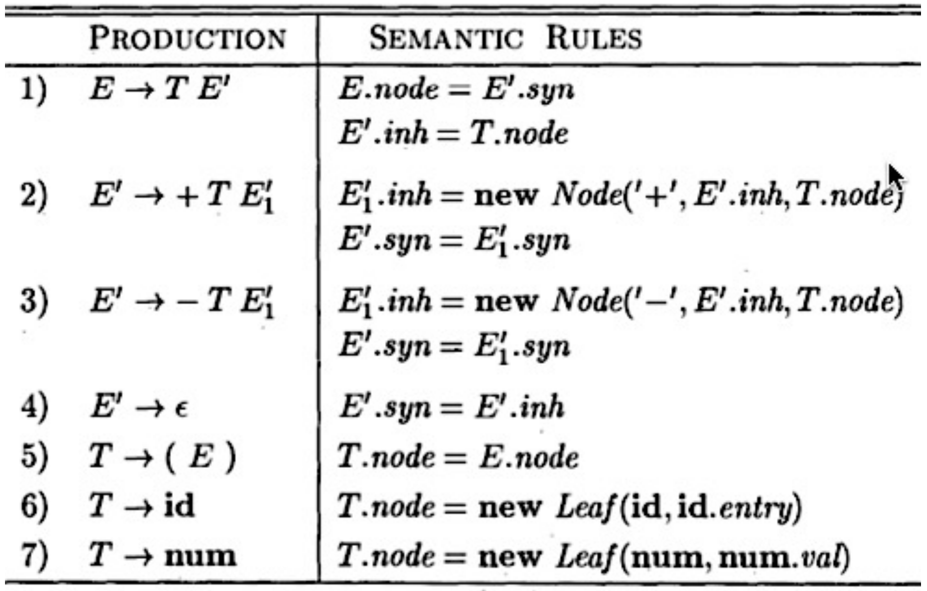


Figure 5.11: Syntax tree for $a - 4 + c$

1) p_1 = **new** $Leaf$(**id**, $entry\text{-}a$);
2) p_2 = **new** $Leaf$(**num**, 4);
3) p_3 = **new** $Node('-', p_1, p_2)$;
4) p_4 = **new** $Leaf$(**id**, $entry\text{-}c$);
5) p_5 = **new** $Node('+', p_3, p_4)$;

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

Syntax tree for L-attributed definition



	PRODUCTION	SEMANTIC RULES
1)	$E \to T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \to +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \to -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \to \epsilon$	$E'.syn = E'.inh$
5)	$T \to (\ E\ )$	$T.node = E.node$
6)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

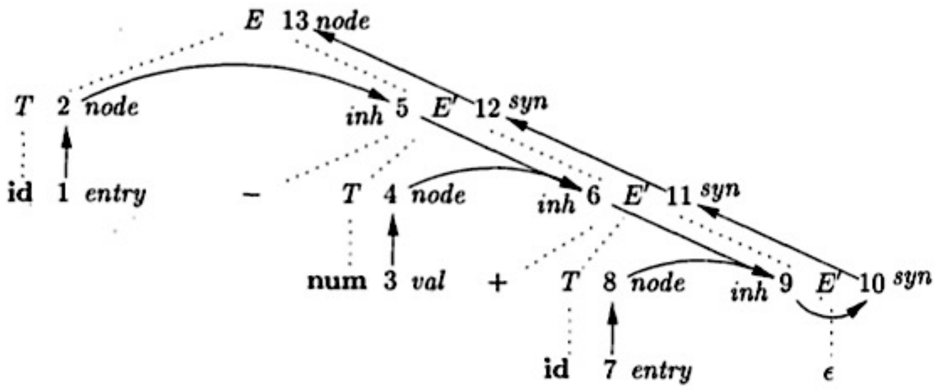


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

The structure of the type

- In c the type int [2][3] can be read as “array of 2 arrays of 3 integers”.
- Corresponding type expression array(2, array(3, integer))

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

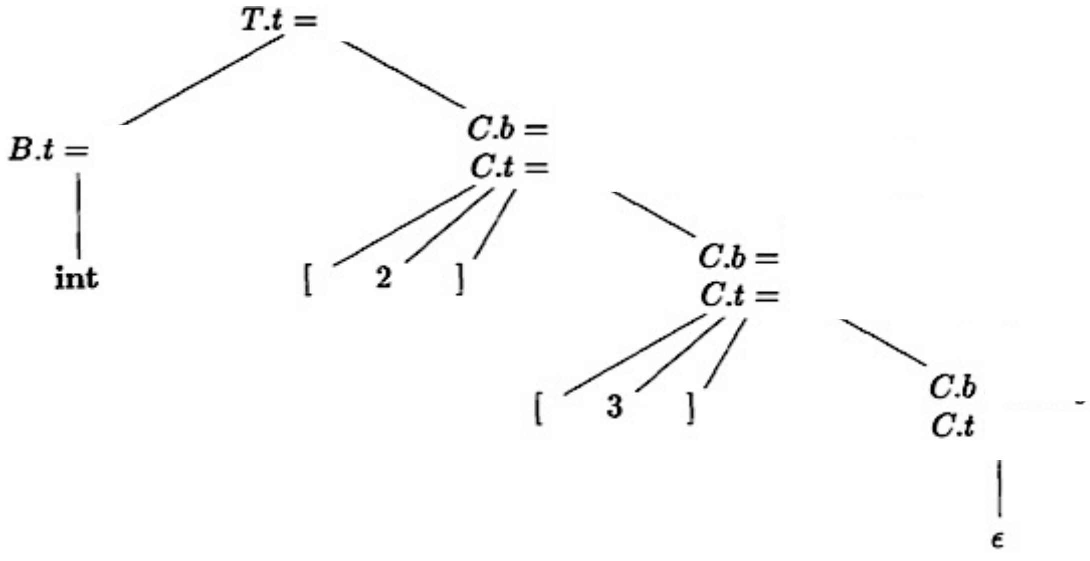
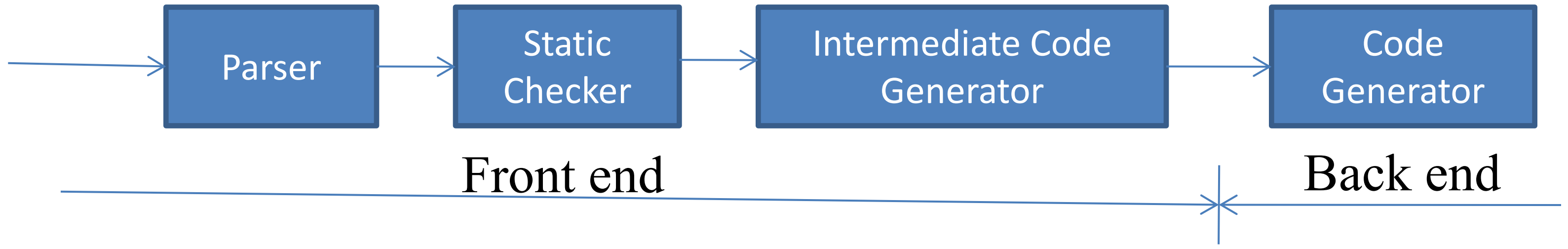


Figure 5.17: Syntax-directed translation of array types

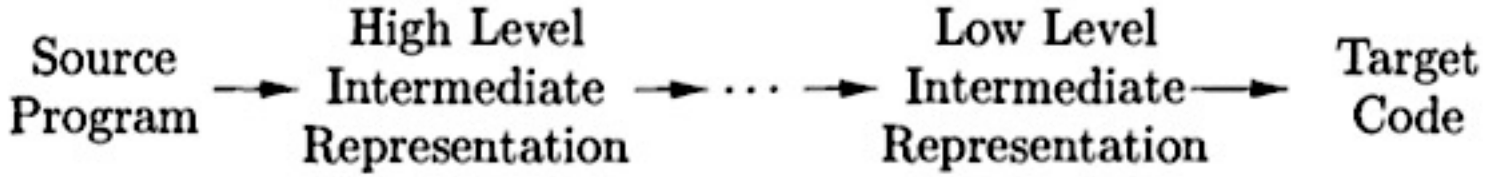
Intermediate code generation :Introduction

- Intermediate code is the interface between front end and back end in a compiler.
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (m*n compilers built by writing m front ends and n back ends).
- Intermediate representations : syntax trees and three address code
- General form *x = y op z*



Intermediate code generation : Introduction

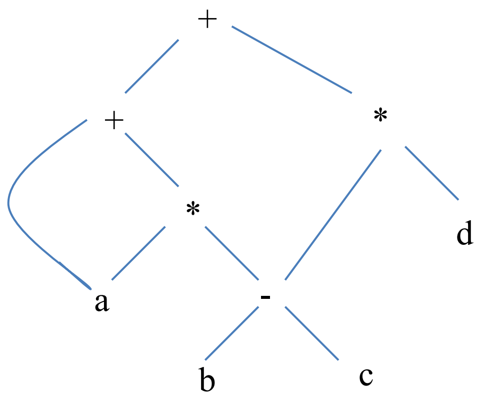
- In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations.


Source Program
High Level Intermediate Representation
···
Low Level Intermediate Representation
Target Code

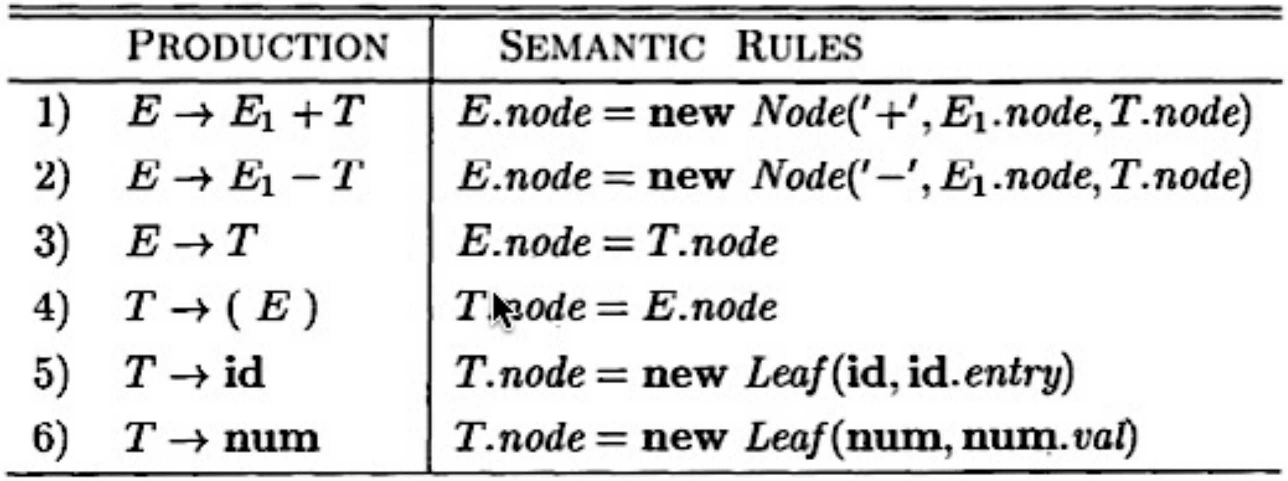
- High level representations are close to the source language.
- Low level representations are close to the target machine.

Variants of syntax trees

- A **Directed Acyclic Graph (DAG)** for an expression identifies the common subexpressions of the expression .
- Node N in a DAG has more than one parent if N represents a common subexpression in a syntax tree.
- DAG gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.
- Example: a+a*(b-c)+(b-c)*d
- ((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))



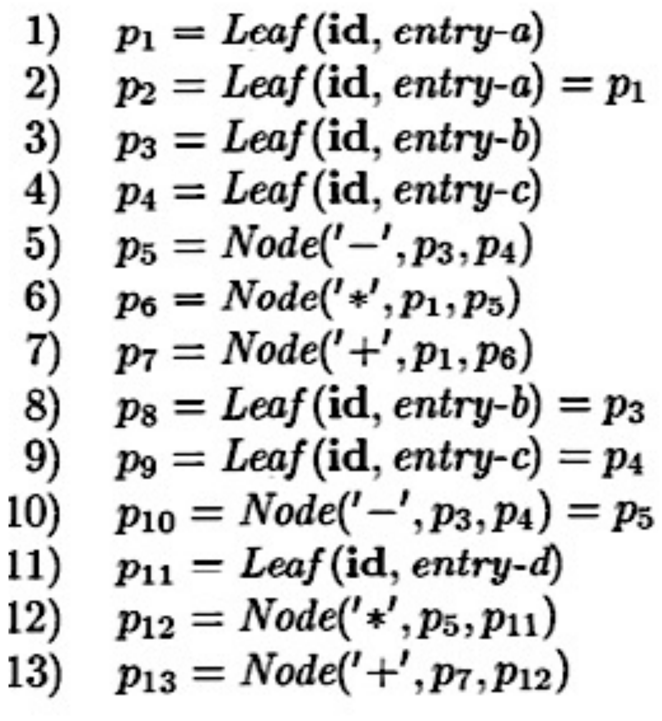
Directed Acyclic Graphs for Expressions



	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

Directed Acyclic Graphs for Expressions

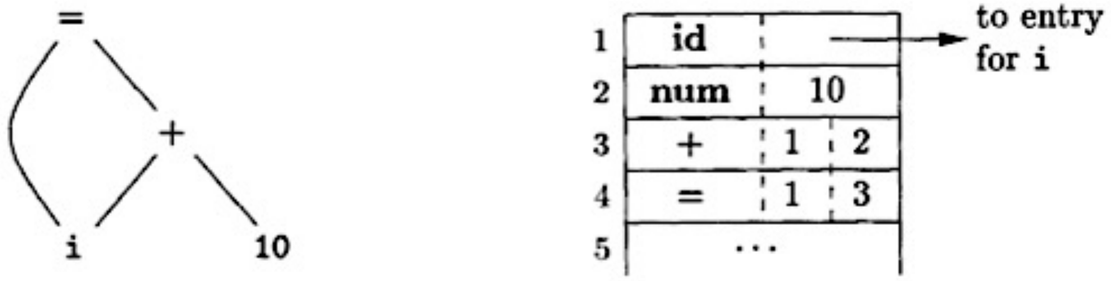


1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a})$
2) $p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a}) = p_1$
3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b})$
4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c})$
5) $p_5 = \mathit{Node}('-', p_3, p_4)$
6) $p_6 = \mathit{Node}('*', p_1, p_5)$
7) $p_7 = \mathit{Node}('+', p_1, p_6)$
8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b}) = p_3$
9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c}) = p_4$
10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}d})$
12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

Steps for constructing the DAG : a+a*(b-c)+(b-c)*d

The *Value-Number* method for constructing a DAG

- The nodes of a syntax tree or DAG are stored in an array of record .
- Each row represents one record and therefore one node.
- The first field is an operation code indicating the label of the node .
- Leaves have one additional field, which holds a lexical value .
- Interior nodes have two additional fields indicating the left and right children.
- Each node is referred by its *value number*.
- Signature of the interior node is the triple <op, l, r>



(a) DAG

(b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label *op*, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: Search the array for a node M with label *op*, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label *op*, left child l, and right child r, and return its value number. □

- Given the input op, l, r compute the bucket index h(op,l,r) and search the list of cells in the bucket for the given input node.

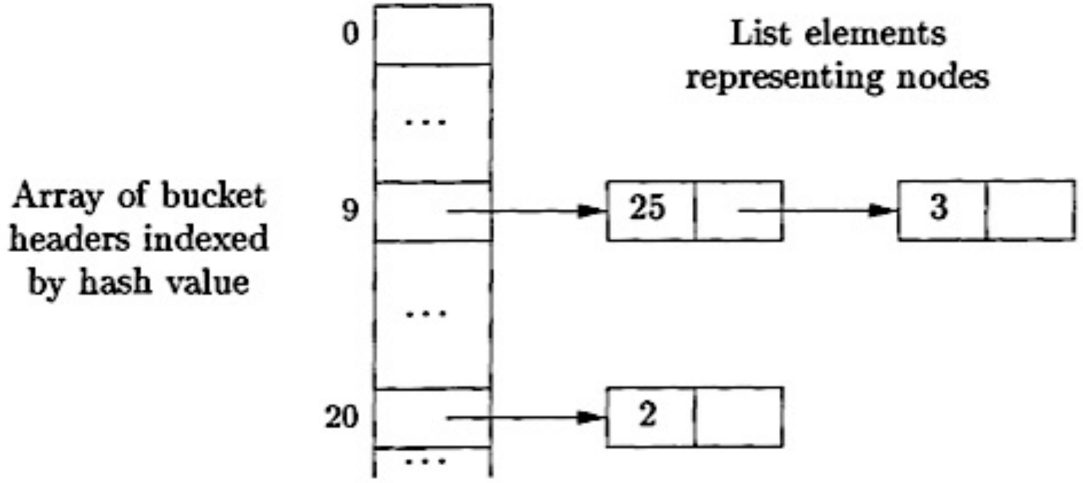

0
9
20
...
Array of bucket headers indexed by hash value
List elements representing nodes
25
3
2

Figure 6.7: Data structure for searching buckets

- Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

i. a+b+(a+b)
ii. a+b+a+b
iii. a+a+(a+a+a+(a+a+a+a))

Three address code

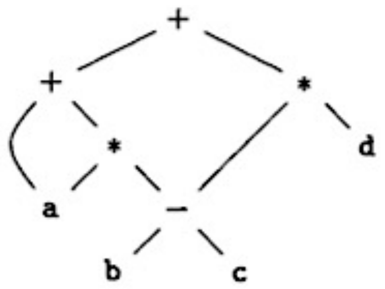
- At the most one operator on the right hand side of the instruction.
- Three address code is a statement containing at most 3 address or operand for each statement.
- The general form is **x = y op z** , with three address: two for the operands y and z, one for the result x.
- x, y, z are variables , constants or temporaries.
- Ex : Source language instruction x+y*z is translated into 3 address code as

t_1=y*z

t_2=x+t_1

where t_1 and t_2 are compiler generated names.

- Three address code is a linearized representation of a syntax tree or DAG.
- Example: a+a*(b-c)+(b-c)*d



(a) DAG

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

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

Addresses and Instructions

- Three address code is built from 2 concepts: addresses and instructions .
- An address can be one of the following:
  - A name
  - A constant
  - A compiler generated temporary

Common Tac instruction forms

- Assignment instructions: x=y op z
- Unary assignments : x=op y
- Copy instruction : x=y
- Unconditional jump : goto L
- Conditional jumps :if x goto L

  if x relop y goto L
- Procedural calls : p (x1,x2,…..xn)

```
param x1
....
param xn
call p,n
```

- Indexed copy instructions : x=y[i] and x[i]=y
- Address and pointer assignments: x=&y, x=*y, *x=y

Consider the statement

```
do i = i+1; while (a[i] < v);
```

L: t_1 = i + 1
$\quad$ i = t_1
$\quad$ t_2 = i * 8
$\quad$ t_3 = a [t_2]
$\quad$ if t_3 < v goto L

(a) Symbolic labels.

100: t_1 = i + 1
101: i = t_1
102: t_2 = i * 8
103: t_3 = a [t_2]
104: if t_3 < v goto 100

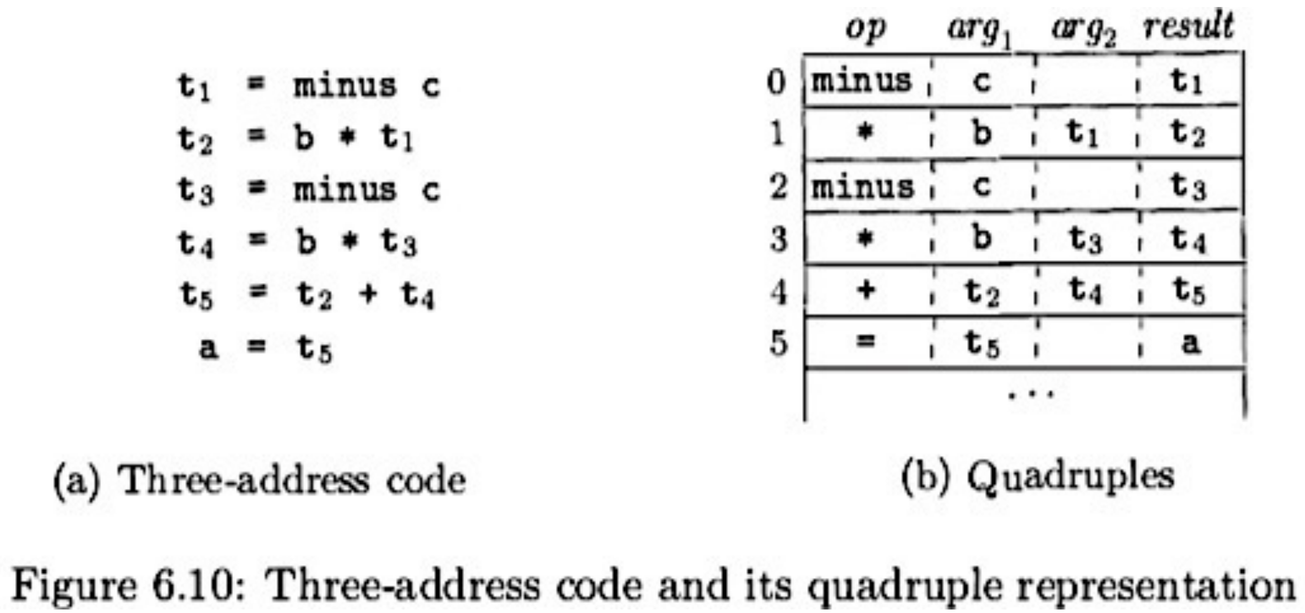
(b) Position numbers.

Three address code representation

Quadruples

- Four fields for instructions: op , arg1,arg2,result
- Instructions with unary operators like x= minus y or x=y do not use arg2
- Operators like param don’t use either arg2 or result
- Conditional and unconditional Jumps put the target label into result

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



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

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t$_1$
1	*	b	t$_1$	t$_2$
2	minus	c		t$_3$
3	*	b	t$_3$	t$_4$
4	+	t$_2$	t$_4$	t$_5$
5	=	t$_5$		a
		. . .		

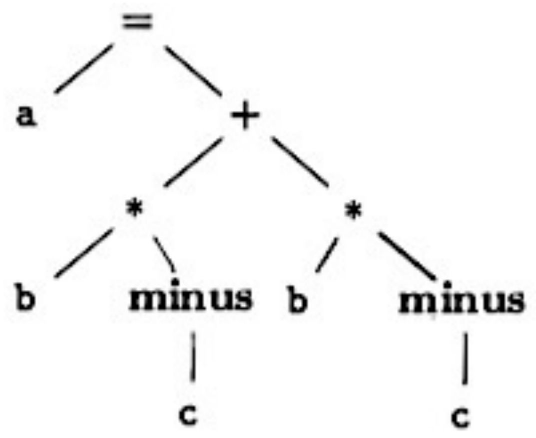
(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

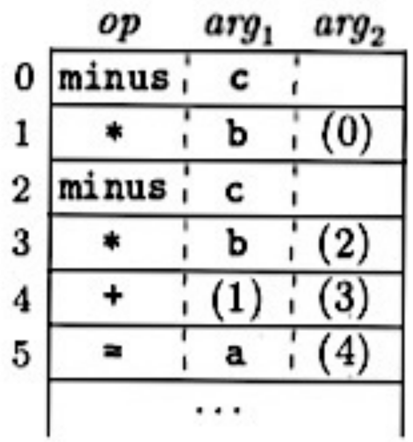
Triples

- Three fields for each instructions: op, arg1,arg2
- The result of an operation x op y is referred to by its position.

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



(a) Syntax tree

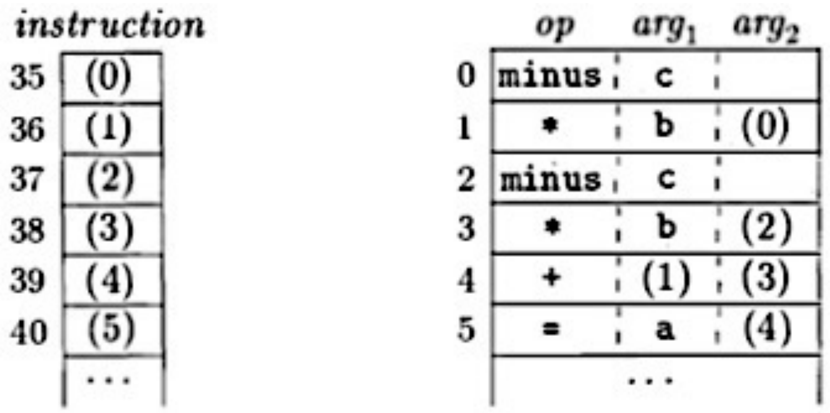


	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Indirect Triples

- Consist of listing of pointers to triples , rather than a listing of the triples



	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Figure 6.12: Indirect triples representation of three-address code

Static single assignment form

- Intermediate representation that facilitates certain code optimizations.
- All assignments in SSA are to variables with distinct names; hence the term static single assignments.

p = a + b	$p_1 = a + b$
q = p - c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e - p	$p_3 = e - p_2$
q = p + q	$q_2 = p_3 + q_1$

(a) Three-address code. (b) Static single-assignment form.

- The same variable may be defined in two different control flow paths in a program. EX:

if(flag) x=-1; else x=1;

y=x*a;

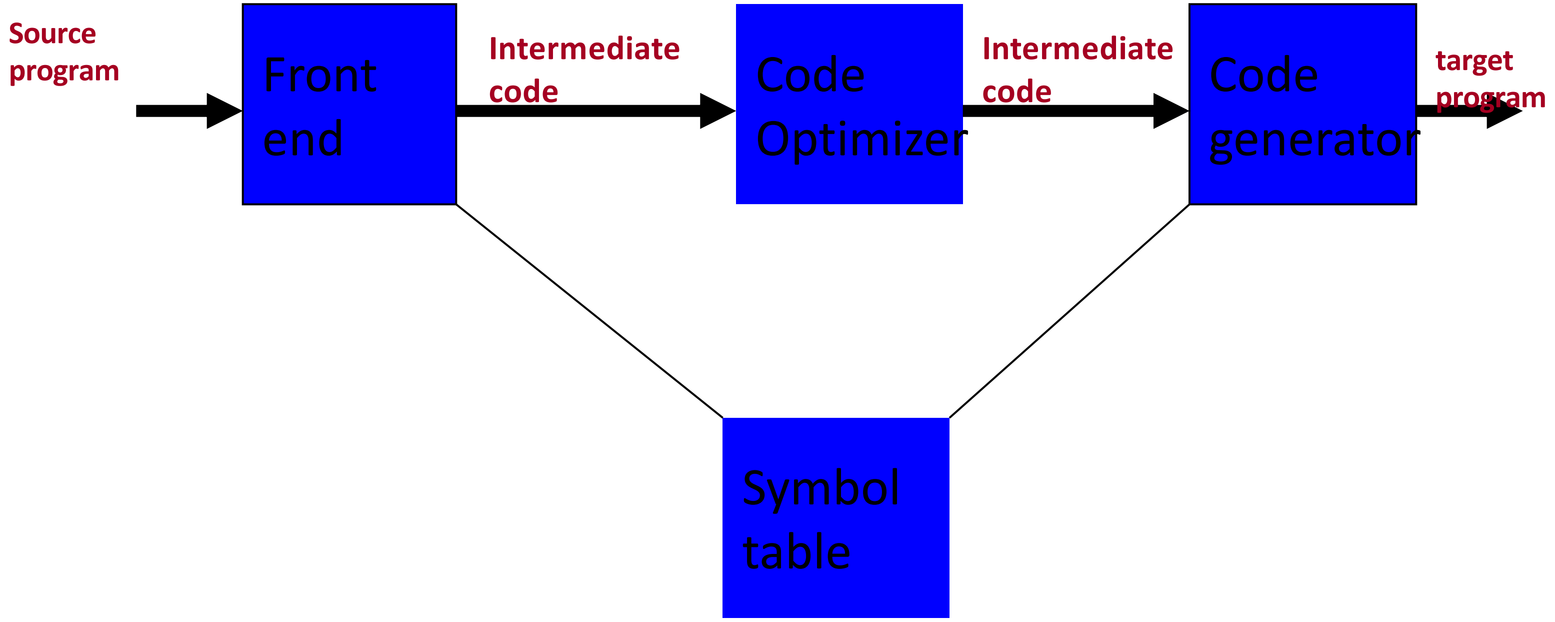
- SSA uses notational convention called the Ø-function to combine two definition of x.

if(flag) x_1=-1; else x_2=1;

x_3= Ø(x_1,x_2);

y=x_3*a;

Code generation :Introduction



Position of code generator

- The target program generated must preserve the semantic meaning of the source program and be of high quality.
- It must make effective use of the available resources of the target machine.
- The code generator itself must run efficiently.

A code generator has 3 primary tasks

- Instruction Selection
- Register Allocation and Assignment
- Instruction Ordering

Issues in the Design of a Code Generator

- The most important criteria for the code gen is that it produces correct codes.

- Depend on
  - Input to the code gen(IR)
  - Target program(language)
  - Operating System
  - Memory management
  - Instruction Selection
  - Register allocation and assignment
  - Evaluation order

Issues in the Design of a Code Generator
1)Input to the Code Generator

- We assume, front end has
  - Scanned, parsed and translate the source program into a reasonably detailed intermediate representations(IR)
  - Type checking, type conversion and obvious semantic errors have already been detected
  - Symbol table is able to provide run-time address of the data objects
  - Intermediate representations may be
    - **Three address representation –quadruples, triples, indirect triples.**
    - **Linear representation -Postfix notations**
    - **Virtual machine representation – bytecode, Stack machine code**
    - **Graphical representation - Syntax tree, DAG**

Issues in the Design of a Code Generator
2)Target Programs

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architectures are
  - RISC
  - CISC
  - Stack based.

Issues in the Design of a Code Generator 2)Target Programs(3)CISC machine

- few registers,
- two-address instructions,
- a variety of addressing modes,
- several register classes,
- variable-length instructions,
- and instructions with side effects.

Issues in the Design of a Code Generator 2)Target Programs(4)Stack-based machine

- Operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.
- To achieve high performance the top of the stack is typically kept in registers.
- Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

Issues in the Design of a Code Generator
2)Target Programs(5)Stack-based machine

- Stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM)
- The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers.
- The interpreter provides software compatibility across multiple platforms.
- *(just-in-time)* JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine.

Issues in the Design of a Code Generator
2)Target Programs(6)

- The output of the code generator is the target program.
- Target program may be
  - Absolute machine language
    - It can be placed in a fixed location of memory and immediately executed
  - Re-locatable machine language
    - Subprograms to be compiled separately
    - A set of re-locatable object modules can be linked together and loaded for execution by a linker
  - Assembly language
    - Easier

Issues in the Design of a Code Generator 2)Target Programs(7)Assumptions in this chapter

- very simple RISC machine
- CISC-like addressing modes(few)
- For readability, assembly code as the target language

Issues in the Design of a Code Generator
3) Instruction Selection(1)

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as:-
  - the level of the IR
  - the nature of the instruction-set architecture
  - the desired quality of the generated code.

Issues in the Design of a Code Generator
3) Instruction Selection(2)
1) Level of the IR

- If the IR is high level:-
  - often produces poor code that needs further optimization.
- If the IR is low level:-
  - generate more efficient code sequences.

Issues in the Design of a Code Generator
3) Instruction Selection(3)
2) Nature of the instruction-set architecture

- For example, the uniformity and completeness of the instruction set are important factors.
- If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.
- On some machines, for example, floating-point operations are done using separate registers.

Issues in the Design of a Code Generator
3) Instruction Selection(4)
3)The quality of the generated code:

- is determined by its speed and size.
  - Say for 3-addr stat **a = a + 1** the translated sequence is

```
LD R0,a
Add R0,R0,#1
ST a,R0
```

  - Instead ,if the target machine has increment instruction (INC), then it would be more efficient.
  - we can write **inc a**
  - We need to know instruction costs in order to design good code sequences
  - But ,accurate cost information is often difficult to obtain.

Issues in the Design of a Code Generator
3) Instruction Selection(5)
4) Instruction speeds and machine idioms

- For example, every three-address statement of the form **x = y + z**, where x, y, and z are statically allocated, can be translated as

```
LD RO, y
ADD RO, RO, z
ST x, RO
```

- This strategy often produces redundant loads and stores.

Issues in the Design of a Code Generator
3) Instruction Selection(6)
4) Instruction speeds and machine idioms(2)

- For example, the sequence of three-address statements

  a = b + c

  d = a + e

  can be translated as

```
LD RO, b
ADD RO, RO, c
ST a, RO
LD RO, a
ADD RO, RO, e
ST d, RO
```

- Here, the fourth statement is redundant since it loads a value that has just been stored, and
- so is the third if **a is not subsequently used.**

Issues in the Design of a Code Generator
4) Register allocation (1)

- A key problem in code generation is deciding what values to hold in what registers.
- Instructions involving
  - register operands :- are usually shorter and faster
  - Memory operands :-larger and comparatively slow.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
  - register allocation:- during which we select the set of variables that will reside in register at a point in the program.
  - register assignment:- during which we pick the specific register that a variable will reside in.

Issues in the Design of a Code Generator
4) Register allocation (2)

- For example certain machines require *register-pairs* for some operands and results.
  - M x, y   multiplication instruction
  - where x, the multiplicand, is the even register of an even/odd register pair and
  - y, the multiplier, is the odd register.
  - The product occupies the entire even/odd register pair.

Issues in the Design of a Code Generator
4) Register allocation (3)

- **D x, y**		the division instruction
- where the dividend occupies an even/odd register pair whose even register is x;
- the divisor is y.
- After division, the even register holds the remainder and the odd register the quotient.

Issues in the Design of a Code Generator
4) Register allocation (4)

- Now, consider the two three-address code sequences in which the only difference in the second statement

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
(a)              (b)
```

Issues in the Design of a Code Generator
4) Register allocation (5)

- The shortest assembly-code sequences for (a) and (b) are

```
L R1,a                 L R0, a
A R1,b                 A R0, b
M R0,c                     A R0, c
D R0,d                 SRDA R0, 32
ST R1,t                    D R0, d
                       ST R1, t
```

- (a) (b)
- **Where SRDA stands for Shift-Right-Double-Arithmetic and**
- **SRDA RO, 32 shifts the dividend into Rl and clears RO so all bits equal its sign bit.**

Issues in the Design of a Code Generator
5) Evaluation order

- It affects the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- Picking a best order in the general case is a difficult NP-complete problem.
- Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

The Target Language

- The target machine and its instruction set is a prerequisite for designing a good code generator.
- In this chapter, we shall use as a target language assembly code for a simple computer that is representative of many register machines.

A Simple Target Machine Model

- It is a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- Is a byte-addressable machine with *n general-purpose registers,* R0,R1,... ,Rn - 1.
- Very limited set of instructions
- Assume that all operands are integers.
- A label may precede an instruction.
- Most instructions consists of an operator, followed by a target, followed by a list of source operands.

- We assume the following kinds of instructions are available:
- *Load operations:* assignment *dst = addr*

  LD *dst, addr*

  LD *r, x*

  LD *r1,r2*

- *Store operations:*assignment *x = r*

  ST *x, r*

- *Computation operations:OP dst, src1,src2,*
    - **SUB *r1,r2,r3*** *r1 = r2 - r3*
- *Unconditional jumps:*
    - BR *L*
- *Conditional jumps:*
    - *Bcond r, L,*
  - where *r is a register, L is a label*
    - BLTZ r, *L*

Assume target machine has a variety of addressing modes:

1. Memory –to-memory:-
2. Indexed addressing :- useful in accessing arrays

   of the form *a(r), where a is* a variable and r is a register.

– For example, the instruction LD Rl, a**(R2)**

– Rl = *contents (a + contents **(R2))***

3. Register indexed addressing :-useful for pointers
   - For example,   LD Rl, **100(R2)**
   - **Rl = *contents(100 +*_contents_*(R2))***

4. Indirect addressing :-

- **r means the memory location* found in the location represented by the contents of register r and
- *100(r) means the memory location found in the location obtained by adding 100 to the contents of r.
- For example, LD Rl, *100(R2)
- Rl = *contents(contents(100 + contents(R2)))*

5. Immediate addressing
    - The constant is prefixed by #.

- LD Rl, **#100 loads the integer 100 into** register Rl,
- ADD Rl, Rl, **#100 adds the integer 100 into register Rl. '**

- The three-address statement x = y - z can be implemented by the machine instructions:
- **LD Rl, y**
- **LD R2, z**
- **SUB Rl, Rl, R2**
- **ST x, Rl**

- Suppose a is an array whose elements are 8-byte values, perhaps real numbers.
- Also assume elements of a are indexed starting at 0.
- three-address instruction b = a [i] by the machine instructions:
- **LD Rl, i // Rl = i**
- **MUL Rl, Rl, 8 // Rl = Rl * 8**
- **LD R2, a(Rl) // R2 = contents(a + contents(Rl))**
- **ST b, R2 // b = R2**

- the assignment into the array a represented by three-address instruction
- a [j] = **c is implemented by:**
- **LD Rl, c // Rl = c**
- **LD R2, j // R2 = j**
- **MUL R2, R2, 8 // R2 = R2 * 8**
- **ST a(R2), Rl // contents(a + contents(R2)) = Rl**

- the three-address statement
- x = *p, we can use machine instructions like:
- **LD Rl, p   // Rl = p**
- **LD R2, 0(R1)   // R2 = contents(0 + contents(Rl))**
- **ST x, R2   // x = R2**

- The assignment through a pointer *p = y is similarly implemented in machine code by:
- **LD Rl, p   // Rl = p**
- **LD R2, y   // R2 = y**
- **ST 0(R1), R2   // contents(0 + contents(Rl)) = R2**

- a conditional-jump three-address instruction like
- **if x < y goto L**
- **The machine-code equivalent would be something like:**
- **LD Rl, x**
- **LD R2, y**
- **SUB Rl, R l , R2**
- **BLTZ R l , M**

Generate code for the following three-address statements assuming *a and b are arrays whose elements are **4-byte values.***

- x = a [i]
- y = b [j]
- a [i] = y
- b [j] = x

Program and Instruction Costs

- A cost with compiling and running a program.
- some common cost measures are
  - the length of compilation time and the size,
  - running time and power consumption of the target program.
- Determining the actual cost of compiling and running a program is a complex problem.

- Finding an optimal target program for a given source program is an undecidable problem
- Many of the subproblems involved are NP-hard.
- In code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

- Assume each target-language instruction has an associated cost.
- For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.
- This cost corresponds to the length in words of the instruction.

- Addressing modes involving
  - registers have zero additional cost,
  - memory location or constant in them have an additional cost of one,
  - Some examples:
- LD RO, Rl cost=1
- LD RO, M cost=2
- LD Rl, *100(R2) *cost =3*

- the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input.

- Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs.

- Determine the costs of the following instruction sequence
- LD RO, y
- LD Rl, z
- ADD RO, RO, Rl
- ST x, RO