Module -5

Syntax Directed Translation, Intermediate code generation, Code generation

Syntax Directed Translation

Outline

- Syntax Directed Definitions
- Evaluation Orders of SDD’s
- Applications of Syntax Directed Translation

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Production — Semantic Rule

E->E1+T — E.code=E1.code||T.code||'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions(SDD)

- A SDD is a context free grammar together with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- If *X* is a symbol and *a* is one of its attributes, X.a denote the value of *a* at a particular parse tree node *X*
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- 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

Example of S-attributed SDD

	Production	Semantic Rules
1)	L -> E n	L.val = E.val
2)	E -> E1 + T	E.val = E1.val + T.val
3)	E -> T	E.val = T.val
4)	T -> T1 * F	T.val = T1.val * F.val
5)	T -> F	T.val = F.val
6)	F -> (E)	F.val = E.val
7)	F -> digit	F.val = digit.lexval

6+4*3

SDD contd…

- An SDD involves only synthesized attributes is called S-attributed; each rule computes an attribute for the non terminal at the head of a production from attributes taken from the body of the production.
- An S-attributed SDD implemented in conjunction with an LR parser.
- A SDD is sometimes called as an attribute grammar. The rules in attribute grammar define the value of an attribute in terms of values of other attributes and constants.

Evaluating an SDD at the Nodes of Parse Tree

- 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*.
- 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 in any bottom-up order, like postorder traversal of the parse tree

Evaluation contd…

- Consider nonterminals A and B, with synthesized and inherited attributes A.s and B.i respectively , along with the production and rules
- PRODUCTION SEMANTIC RULES

PRODUCTION	SEMANTIC RULES
A->B	A.s=B.i;
	B.i=A.s+1

These rules are circular ; Not possible to evaluate either A.s and B.i at some pair of nodes in a parse tree

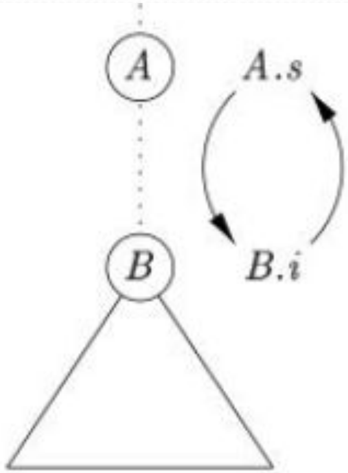


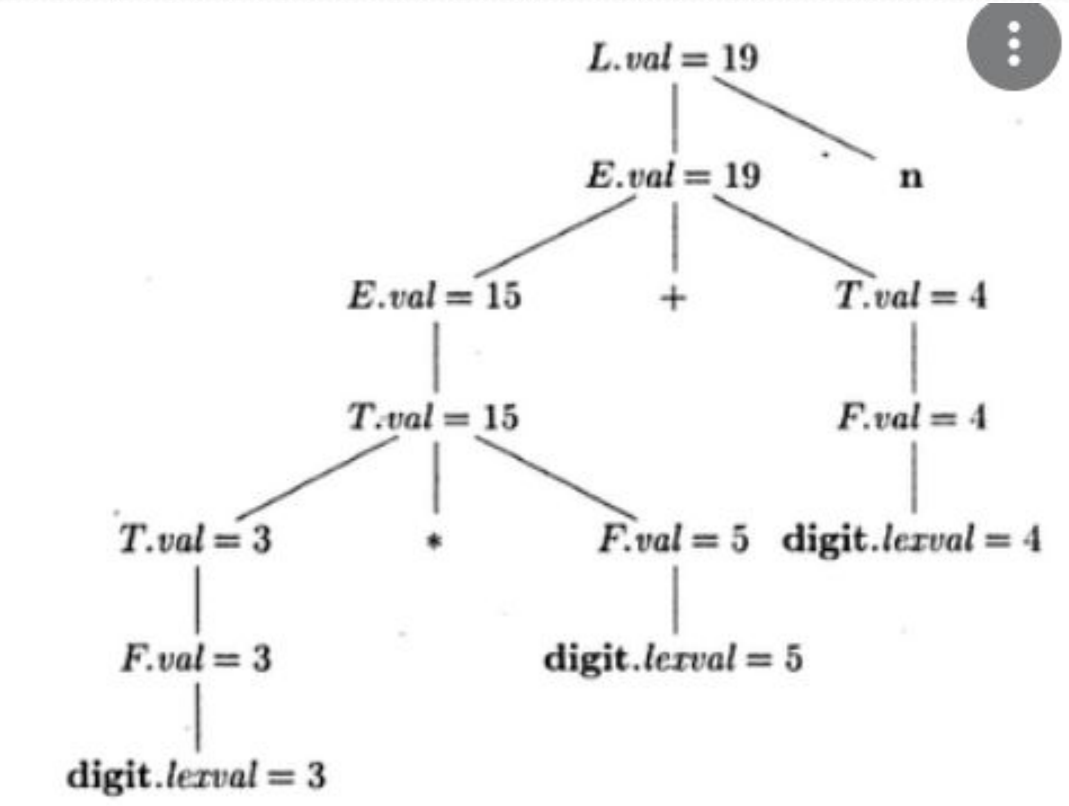
Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

Annotated parse tree

- The values of lexical are presumed supplied by the lexical analyzer
- Each of the nodes for the nonterminals has attribute val computed in a bottom-up order.
- Inherited attributes are useful when the structure of a parse tree does not “match” the abstract syntax of the source code

6+4*3

Annotated parse tree 3*5+4 n



Example of mixed attributes

Production	Semantic Rules
1) T -> FT'	T'.inh = F.val T.val = T'.syn
2) T' -> *FT'$_1$	T'$_1$.inh = T'.inh*F.val T'.syn = T'$_1$.syn
3) T' -> ε	T'.syn = T'.inh
1) F -> digit	F.val = F.val = digit.lexval

Annotated parse tree for 3*5

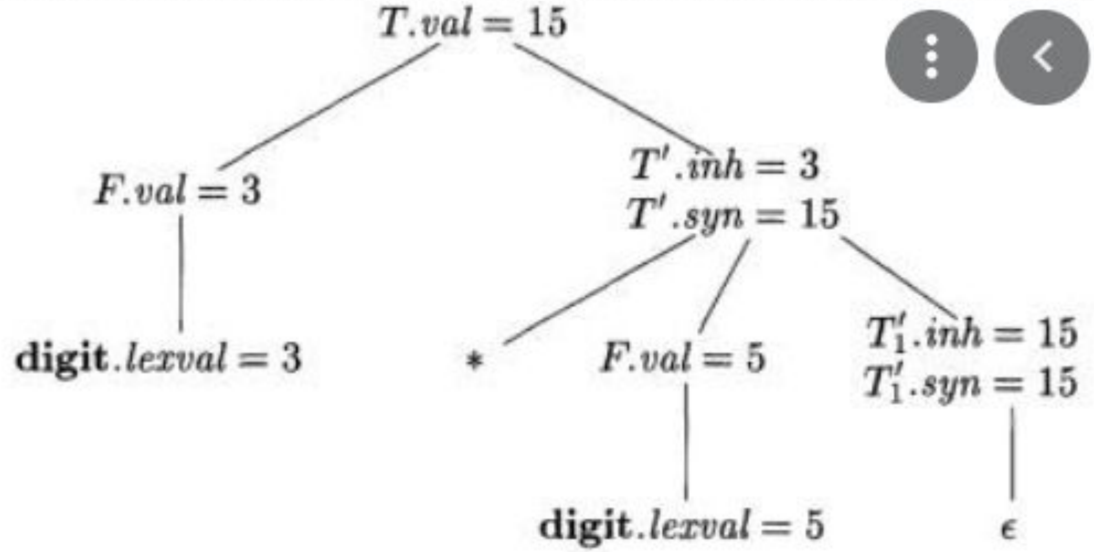


Figure 5.5: Annotated parse tree for $3 * 5$

Evaluation orders for SDD’s

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - 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.c to B.c

Example

- Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
E->E1 + T	E.val=E1.val + T.val

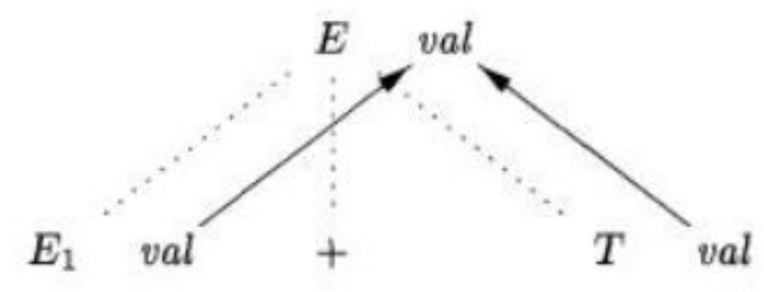
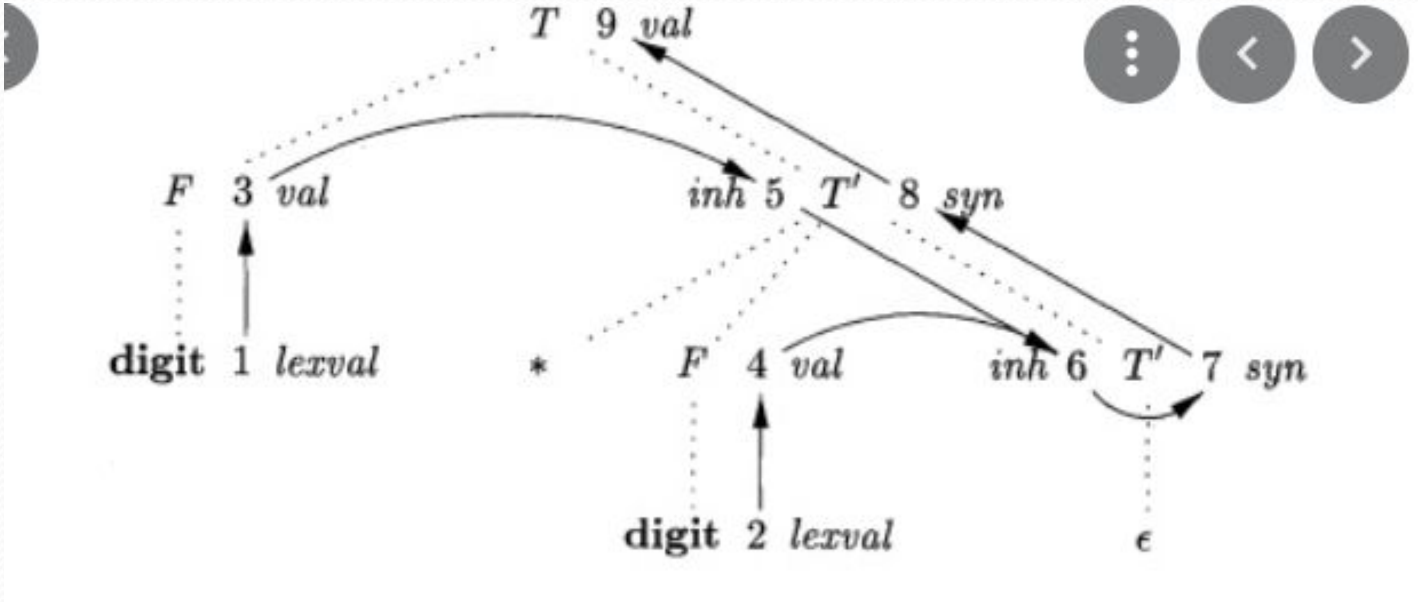


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Dependency graph for the 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 N1,N2,….,Nk such that if there is an edge from Ni to Nj then i<j
- Such an ordering embeds a directed graph into a linear order and is called a topological sort of the graph
- If there is a cycle in the graph, then there are no topological sorts means no way to evaluate the SDD on this parse tree

Ordering the evaluation of attributes contd…

- If there are no cycles, then there is always at least one topological sort
- Since there are no cycles , find a node with no edge entering, if there is no such node , proceed from predecessor to predecessor until get the node that already seen, yielding a cycle
- Make this node the first in the topological order, remove it from the dependency graph and repeat the process on the remaining nodes
- Another topological sort is 1,3,5,2,4,6,7,8,9

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->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may use only :
    - 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 formed by the attributes of this Xi

Example

- PRODUCTION SEMANTIC RULE

  T->FT' T'.inh=F.val

  T'->*FT'1 T'1.inh=T'.inh*F.val
- The first rule defines the inherited attribute T'.inh using only F.val and F appears to the left of T' in the production body as required
- The second rule defines T'.inh using the inherited attribute T'.inh associated with the head and F.val, where F appears to the left of T'1 in the production body.

Example

- Any SDD containing the following production and rules cannot be L-attributed:
- PRODUCTION SEMANTIC RULE

  A->B C A.s=B.b;

  B.i=f(C,c,A.s)
- The first rule A.s=B.b, is a legitimate rule in either as S-attributed or L-attributed SDD. It defines a synthesized attribute A.s in terms of an attribute at a child
- The second rule defines an inherited B.i, so the entire SDD cannot be S-attributed.
- The SDD cannot be L-attributed, because the attribute C.c is used to help define B.i, and C is to the right of B in the production body.

Semantic Rules with Controlled Side Effects

- Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph.
- Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment
- Controlling the side effects in SDD's in following the ways

Ways to control side effects

- Permit incidental side effects that do not constrain attribute evaluation.
- Permit the side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation, where correct depends on the application
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order.
- The constraints can be as implicit edges added to the dependency graph

Example

- The rule L.val=E.val-saves the result in the synthesized attribute
- Consider: PRODUCTION SEMANTIC RULE

  1) L->E n print(E.val)
- Example
- A simple declaration D consisting of a basic type T followed by a list L of identifiers T can be ***int*** or ***float***
- This SDD does not check whether an identifier is declared more than once

SDD for simple type declarations

	Production	Semantic Rules
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to \textbf{int}$	$T.type = \text{integer}$
3)	$T \to \textbf{float}$	$T.type = \text{float}$
4)	$L \to L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \to \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

Example

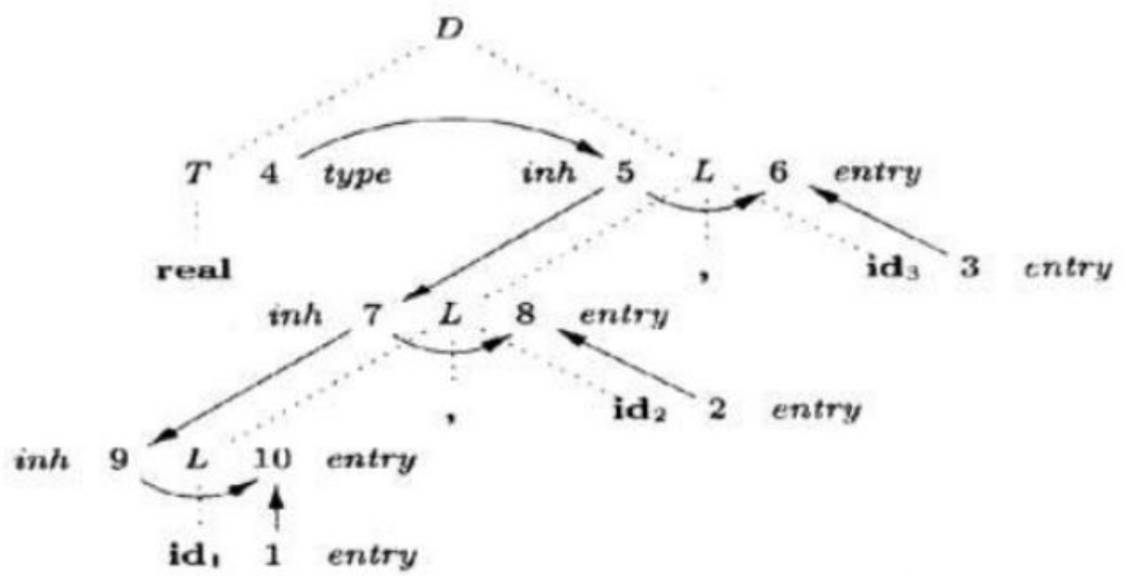
- Production 1 has nonterminal D represents a declaration which consists of type T followed by a list L of identifiers. T has one attribute, T.type- the type in declaration D. Nonterminal L also has one attribute as inherited attribute .
- Production 2 and 3 each evaluate the synthesized attribute T. type as integer or float. This type is passed to the attribute L.inh in the rule for production 1.
- Production 4 passes L.inh down the parse tree. The value L1.inh is computed at a parse tree node by copying the value of L.inh from the parent of that node; the parent corresponds to the head of the production.

Example

- Production 4 and 5 also a rule in which function addType is called with arguments:
  - id.entry , a lexical value that points to a symbol-table object, and
  - L.inh, the type being assigned to every identifier on the list

A dependency graph for the input string

float id1 , id 2, id3



Application of Syntax Directed Translation

- Type checking and intermediate code generation (chapter 6)
- Construction of syntax trees
  - Leaf nodes: Leaf(op,val)
  - Interior node: Node(op,c1,c2,…,ck)
- Example:

	Production	Semantic Rules
1)	E -> E1 + T	E.node=new node('+', E1.node,T.node)
2)	E -> E1 - T	E.node=new node('-', E1.node,T.node)
3)	E -> T	E.node = T.node
4)	T -> (E)	T.node = E.node
5)	T -> id	T.node = new Leaf(id,id.entry)
6)	T -> num	T.node = new Leaf(num,num.val)

Example- Syntax tree for a-4+c

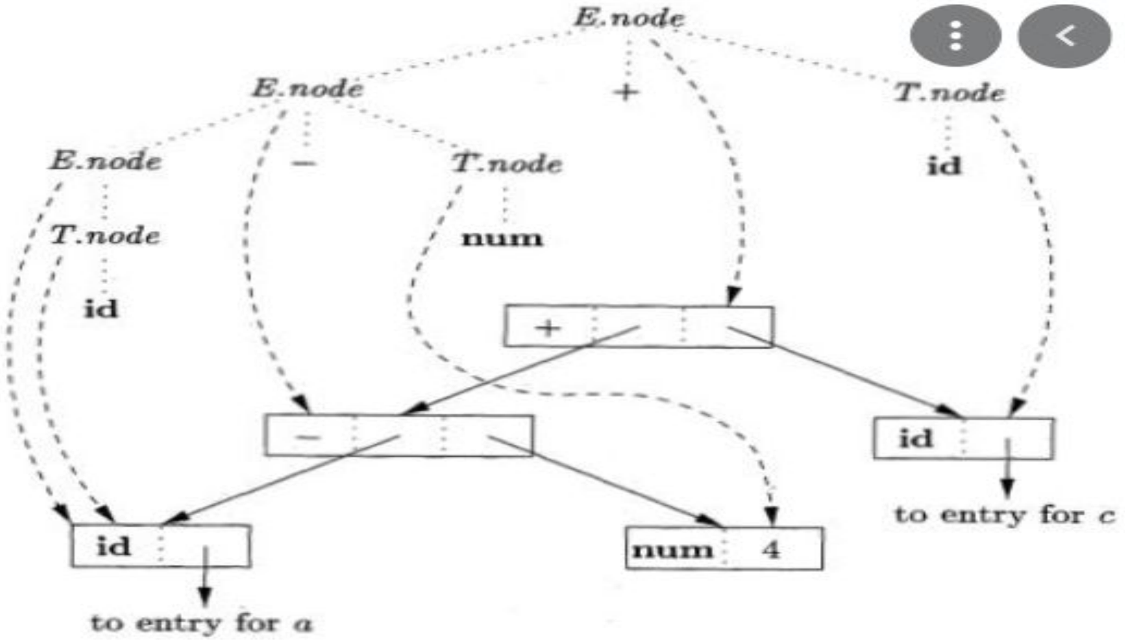


Figure 5.11: Syntax tree for $a - 4 + c$

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

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

Syntax tree for L-attributed definition – Top down parsing

	Production	Semantic Rules
1)	E -> TE’	E.node=E’.syn E’.inh=T.node
2)	E’ -> + TE1’	E1’.inh=new node(‘+’, E’.inh,T.node) E’.syn=E1’.syn
3)	E’ -> -TE1’	E1’.inh=new node(‘+’, E’.inh,T.node) E’.syn=E1’.syn
4)	E’ -> ∈	E’.syn = E’.inh
5)	T -> (E)	T.node = E.node
6)	T -> id	T.node=new Leaf(id,id.entry)
7)	T -> num	T.node = new Leaf(num,num.val)

Dependency graph for a-4+c

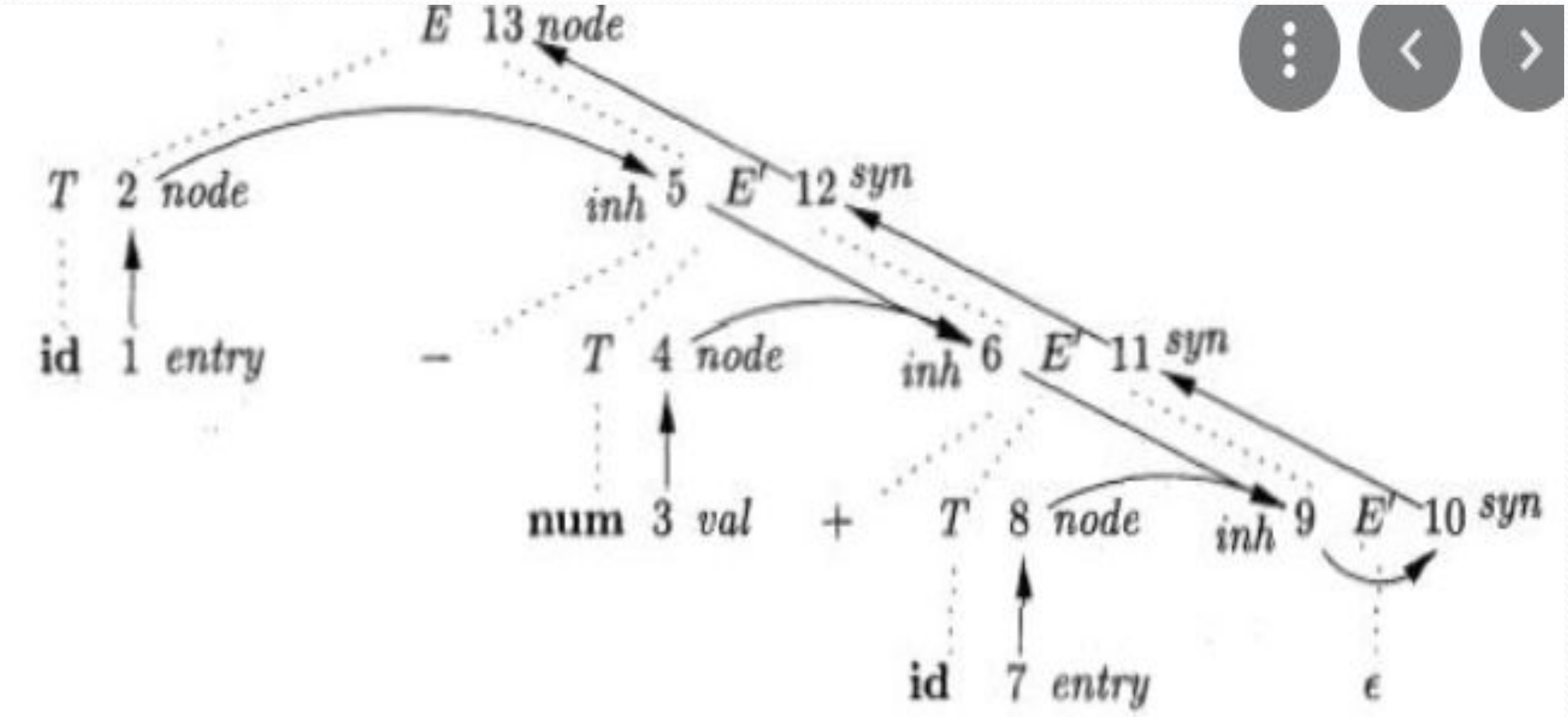


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

The structure of a Type

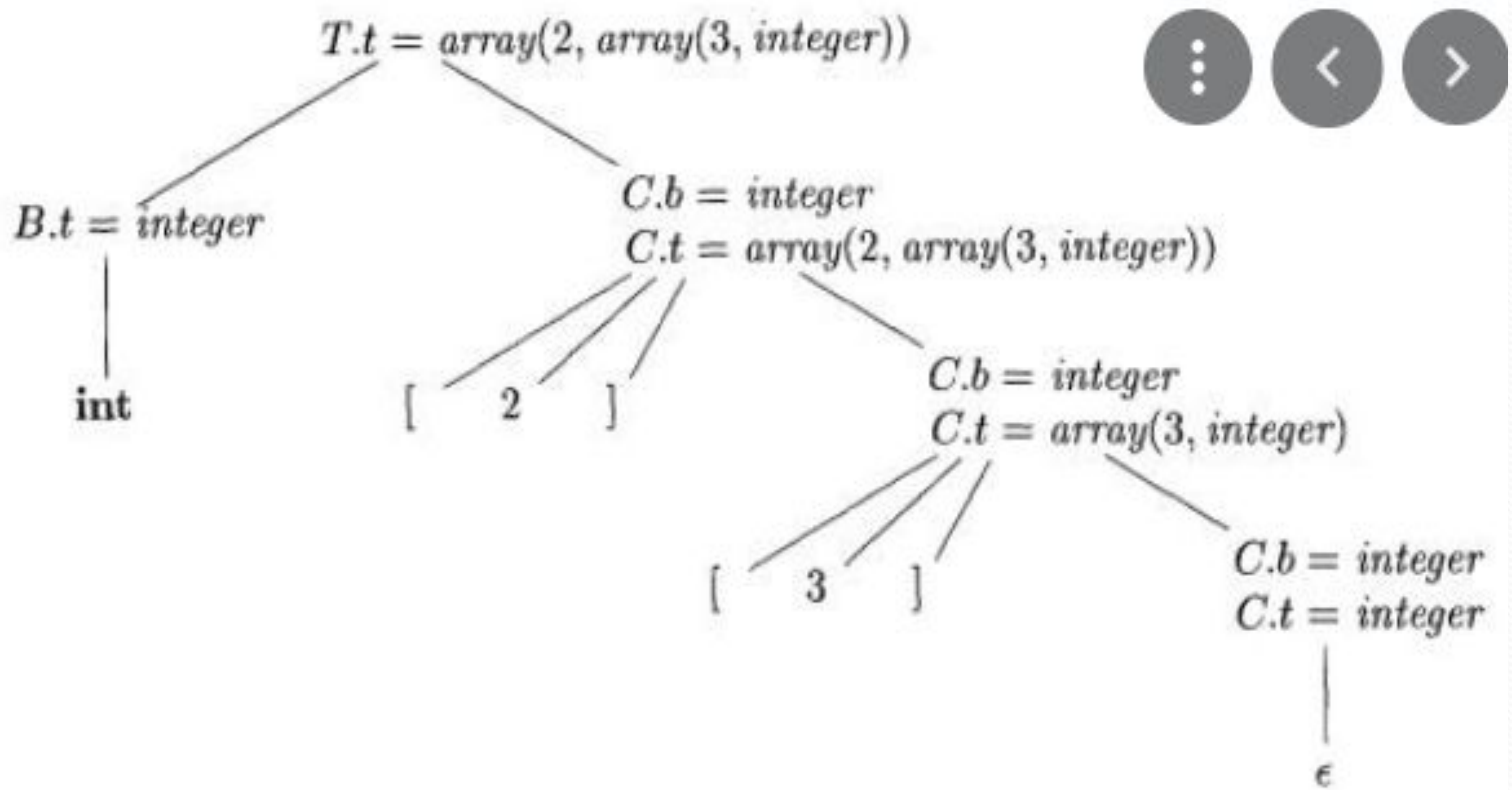
- Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax input; attributes can then be carry information from one part of the parse tree to another.
- Example shows how a mismatch in structure can be due to the design of the language and not due to constraints imposed by the parsing method
- The nonterminals B and T have a synthesized attribute t representing a type. The nonterminal C has two attributes: an inherited attribute b pass a basic type down the tree and the synthesized t attributes accumulate the result

T generates a basic type or an array Type

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

In C, the type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below.

Syntax-directed translation of array types
Annotated parse tree



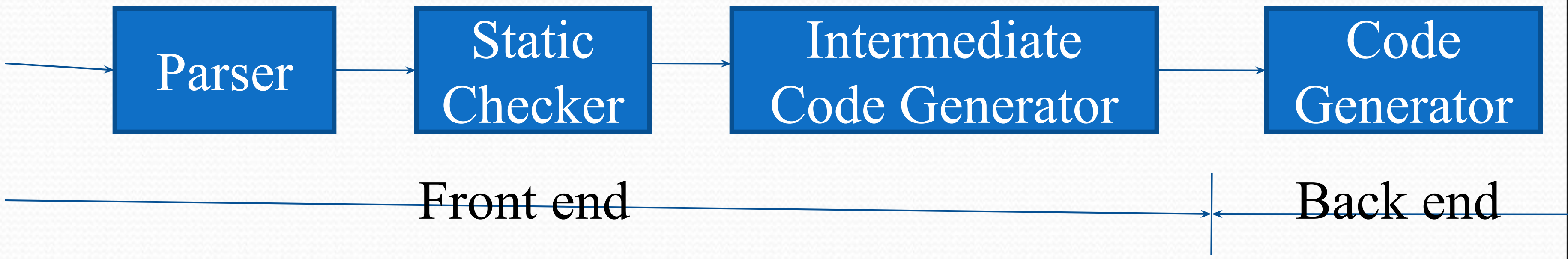
Intermediate Code Generation

Outline

- Variants of Syntax Trees
- Three-address code

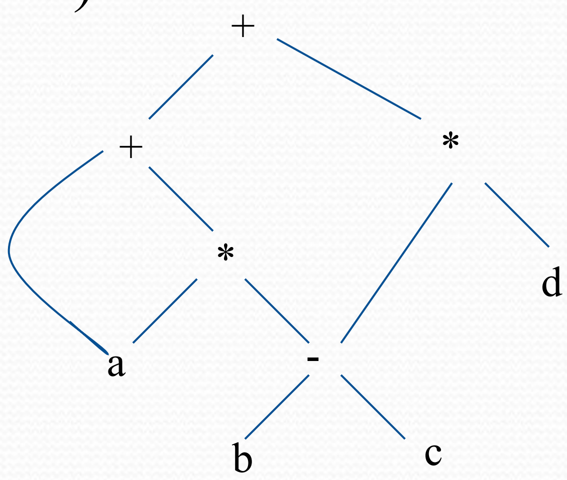
Introduction

- 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 (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation



Variants of syntax trees

- It is sometimes beneficial to crate a Directed Acyclic Graph(DAG) instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- A DAG has leaves as atomic operands and interior nodes as operators
- The difference is that a node N in a DAG has more than one parent of N represents a common sub expression; in syntax tree sub expressions are replicated.
- Example: a+a*(b-c)+(b-c)*d



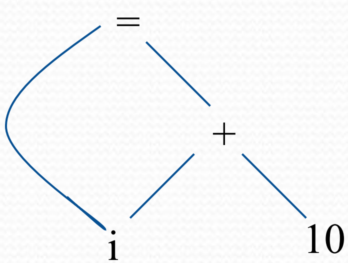
SDD for creating DAG's

Production	Semantic Rules
1) E -> E1+T	E.node= new Node('+', E1.node,T.node)
2) E -> E1-T	E.node= new Node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = new Leaf(num, num.val)

Example:

1)p1=Leaf(id, entry-a)
2)P2=Leaf(id, entry-a)=p1
3)p3=Leaf(id, entry-b)
4)p4=Leaf(id, entry-c)
5)p5=Node('-',p3,p4)
6)p6=Node('*',p1,p5)
7)p7=Node('+',p1,p6)
8) p8=Leaf(id,entry-b)=p3
9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p3,p4)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p5,p11)
13) p13=Node('+',p7,p12)

Value-number method for constructing DAG's

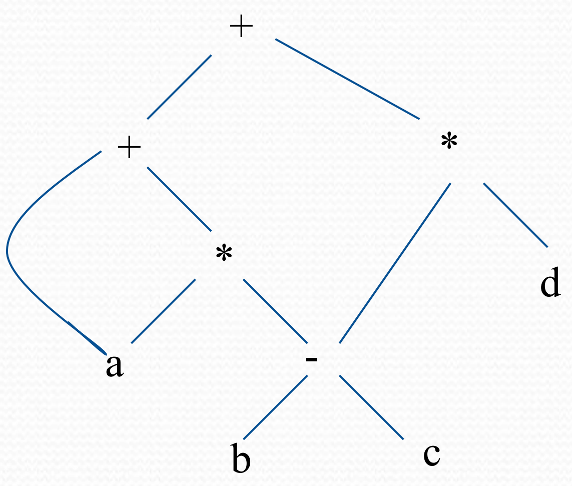


id	→ To entry for i	
num	10	
+	1	2
3	1	3

- Algorithm
  - 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 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
- We may use a hash table

Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example:



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

Forms of three address instructions

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and ifFalse x goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

Example

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

Symbolic labels	Position numbers
L: t1 = i + 1	100: t1 = i + 1
i = t1	101: i = t1
t2 = i * 8	102: t2 = i * 8
t3 = a[t2]	103: t3 = a[t2]
if t3 < v goto L	104: if t3 < v goto 100

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

- b * minus c + b * minus c

Three address code

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

Quadruples

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

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples

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

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Static Single-Assignment Form(SSA)

- SSA is an intermediate representation that facilitates certain code optimizations.
- All assignments in SSA are to variables with distinct names;
- Subscripts distinguish each definition of variables p and q in the SSA representation
- The source program: if(flag) x=-1; else x=1;
- y=x*a
- SSA: if(flag) x1=-1; else x2=1;

  x3=φ (x1,x2)

Intermediate program in three-address code and SSA

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$
\begin{aligned}
p_1 &= a + b\\
q_1 &= p_1 - c\\
p_2 &= q_1 * d\\
p_3 &= e - p_2\\
q_2 &= p_3 + q_1
\end{aligned}
$$

(b) Static single-assignment form.

Code Generation

Outline

- Code Generation Issues
- Target language Issues

Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Insrtuction ordering

Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
  - IR + Symbol table
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected
- The target program
  - Common target architectures are: RISC, CISC and Stack based machines
  - In this chapter we use a very simple RISC-like computer with addition of some CISC-like addressing modes

complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD   R0, y
ADD  R0, R0, z
ST   x, R0
```

a=b+c
d=a+e

```
LD   R0, b
ADD  R0, R0, c
ST   a, R0
LD   R0, a
ADD  R0, R0, e
ST   d, R0
```

Register allocation

- Two subproblems
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Resister assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

LD   R1, a
A    R1, b
M    R0, c
D    R0, d
ST   R1, t
```

```
t=a+b
t=t+c
T=t/d

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

```
a=a+1

INC a

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

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means R1=contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: *r and *100(r)
- immediate constant addressing mode: like LD R1, #100

Example

- x=y-z
- LD R1,y //R1=y
- LD R2,z //R2=z
- SUB R1,R1,R2 //R1=R1-R2
- ST x, R1 //R1=x

b = a [i]

LD R1, i **//R1 = i**

MUL R1, R1, 8 **//R1 = Rl * 8**

LD R2, a(R1) **//R2=contents(a**+**contents(R1))**

ST b, R2 **//b = R2**

a[j] = c

```
LD R1, c            //R1 = c
LD R2, j            // R2 = j
MUL R2, R2, 8       //R2 = R2 * 8
ST  a(R2), R1       //contents(a+contents(R2))=R1
```

x=*p

LD R1, p //R1 = p

LD R2, 0(R1) // R2 = contents(0+contents(R1))

ST x, R2 // x=R2

*P=y

LD R1,p //R1=p

LD R2,y //R2=y

ST 0(R1),R2 // contents(0+contents(R1))=R2

conditional-jump three-address instruction

If x<y goto L

```
LD R1, x            // R1 = x
LD R2, y            // R2 = y
SUB R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M          // i f R1 < 0 jump t o M
```

Program and Instruction costs

- Optimizing the program in terms of cost requires
  - Length of compilation time
  - The Size
  - The running time
  - Power consumption of the target program
- Determining the actual cost of compiling and running a program is complex problem.
- Finding an optimal target program for a given source program is undecidable problem and other prbolems are NP hard

Cost allocation

- We shall assume each target-language instruction has an associated cost.
- We take cost of one instruction to be one plus the costs associated with the addressing modes of the operands.
- Cost corresponds to the length in words of the instruction.
- Addressing modes involving registers have zero additional cost , involving memory location and constants in them have an additional cost of one

costs associated with the addressing modes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 3