COMPILER DESIGN

UNIT-3

Syntax-Directed Translation

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Semantic Analysis

- Additional info related to the meaning of the program once syntactic structure known

- In C: semantic analysis - adding additional info to symbol table, perform type checking

- Semantic analysis needs
  (i) Representation Formalism
  (ii) Implementation Mechanism

Semantic Rules

- Two notations for attaching semantic rules
  1. Syntax directed Definitions - high level
  2. Translation schemes - indicates order of semantic rules

SYNTAX-DIRECTED TRANSLATION

- Illustration of Representational Formalism

- Meaning of input sentence related to its syntactic structure (parse tree)

- Syntax-directed definition associates
  1. A set of attributes with every grammar symbol (NT & T)
  2. Set of semantic rules with each production (compute the values of the attributes associated with the grammar symbols in the production)

Evaluating attributes

1. For input string x, construct a parse tree

2. Apply semantic rules to evaluate attributes at each node in the parse tree (as follows)

Suppose a node N in a parse tree is labeled by the grammar symbol X. We write $X.a$ to denote the value of attribute a of X at that node. A parse tree showing the attribute values at each node is called an *annotated* parse tree. For example, Fig. 2.9 shows an annotated parse tree for 9-5+2 with an attribute t associated with the nonterminals *expr* and *term*. The value 95-2+ of the attribute at the root is the postfix notation for 9-5+2. We shall see shortly how these expressions are computed.

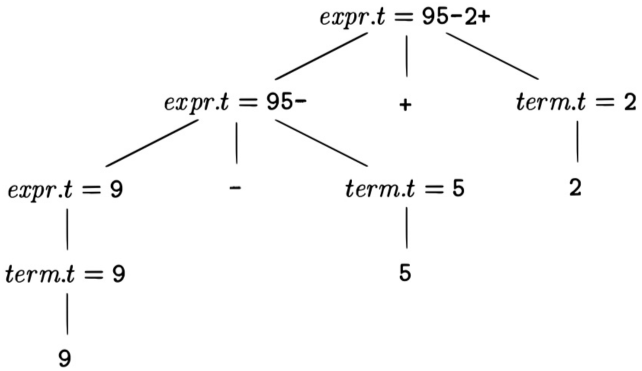


Figure 2.9: Attribute values at nodes in a parse tree

- **Synthesized attribute:** if attribute's value at a parse-tree node N is determined from attribute values at the children of N and at N itself
  - can be evaluated during single bottom-up traversal

- Inherited attribute: if attribute's value at a parse-tree node N is determined from attribute values at the parent of N, N itself and N's siblings

Example 2.10: The annotated parse tree in Fig. 2.9 is based on the syntax-directed definition in Fig. 2.10 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued attribute t that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol $\|$ in the semantic rule is the operator for string concatenation.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \;\|\; term.t \;\|\; '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \;\|\; term.t \;\|\; '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Figure 2.10: Syntax-directed definition for infix to postfix translation

[3] In this and many other rules, the same nonterminal (*expr*, here) appears several times. The purpose of the subscript 1 in $expr_1$ is to distinguish the two occurrences of *expr* in the production; the "1" is not part of the nonterminal. See the box on "Convention Distinguishing Uses of a Nonterminal" for more details.

SDDs

- Each production $A \to \alpha$ is associated with a set of semantic rules

$$b := f(c_1, c_2, \ldots, c_k)$$

- f : function

- b: either (i) or (ii)

  (i) Synthesized attribute of A, and $c_1, c_2, \ldots, c_k$ are attributes of grammar symbols of prod $A \to \alpha$

  (ii) Inherited attribute of a grammar symbol in α, are $c_1, c_2, \ldots, c_k$ are attributes of grammar symbols in α or attributes of A

Production	Semantic Rule
$E \to E_1 + T$	{ $E.val = E_1.val + T.val$ }
$E \to T$	{ $E.val = T.val$ }
$T \to T_1 * F$	{ $T.val = T_1.val * F.val$ }
$T \to F$	{ $T.val = F.val$ }
$F \to$ num	{ $F.val$ = num.lexval }
$F \to$ id	{ $F.val$ = id.lexval }

- Each non-terminal associated with a synthesized attribute val

- Terminals assumed to have synthesized attributes supplied by the lexical analyzer

Example 5.1: The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators + and $*$. It evaluates expressions terminated by an endmarker **n**. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

- S-attributed SDD: SDD that involves only synthesized attributes
  - each rule computes an attribute for NT at prod head from attributes of body of prod

- Evaluation order: semantic rules in S-Attributed Definition can be evaluated by bottom-up or Post-order traversal of the parse tree

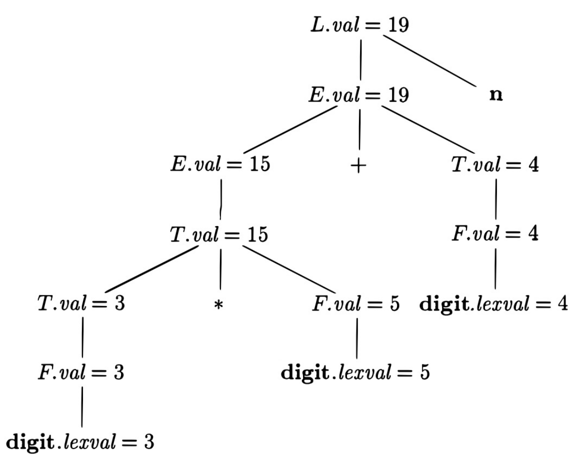


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

Inherited Attributes

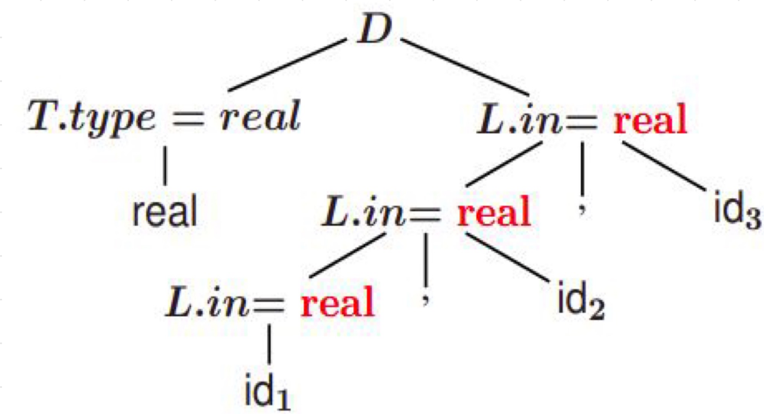
- Useful for expressing dependence of a construct on the context in which it appears
- Order in which inherited attributes of children are computed is important
- Evaluation order for inherited SDDs: cannot perform a simple preorder traversal of parse tree
- Inherited attributes that do not depend on right children: can be evaluated by classical preorder

Example

- SDD for type declarations
- Non terminal T: synth attr type determined by keyword in the declaration
- Production D → T L : associated with semantic rule L.in = T.type; which sets inherited attr L.in

Production	Semantic Rule
D -> T L	*{ L.in = T.type;}*
T -> int	*{ T.type = integer; }*
T -> real	*{ T.type = float; }*
L -> L_1 , id	*{ L_1.in= L.in; addType(id.entry, L.in); }*
L -> id	*{ addType(id.entry, L.in); }*

- Annotated parse tree for input real id1, id2, id3



- L.in inherited top-down
- At each L node, addtype inserts the type of the identifier into the symbol table

EVALUATING SDDs

1. Construct parse tree for given input
2. Construct dependency graph
3. Topologically sort nodes of the dependency graph
4. Produce as output annotated parse tree

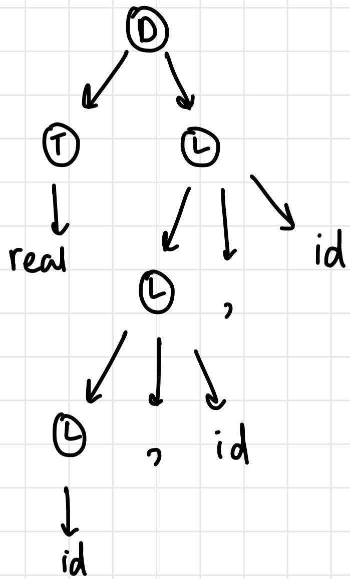
Dependency Graph

- Most general technique used to evaluate SDDs with both synthesized and inherited attributes
- Shows interdependencies among attrs of various nodes of parse tree
  - There is a node for every attribute
  - If attr b depends on attr c, there is a link from node c to node b (b ⟵ c)
- Dependency rule: if b ⟵ c, need to fire semantic rule for c and then b

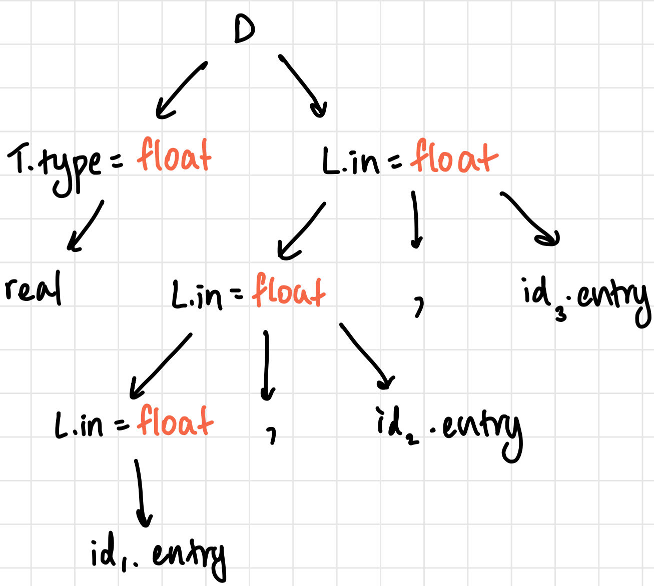
Q: Consider example of SDD for type declarations. Construct annotated parse tree and dependency graph for input real id1, id2, id3.

Production	Semantic Rule
D → T L	{ L.in = T.type;}
T → int	{ T.type = integer; }
T → real	{ T.type = float; }
L → L_1 , id	{ L_1.in= L.in; addType(id.entry, L.in); }
L → id	{ addType(id.entry, L.in); }

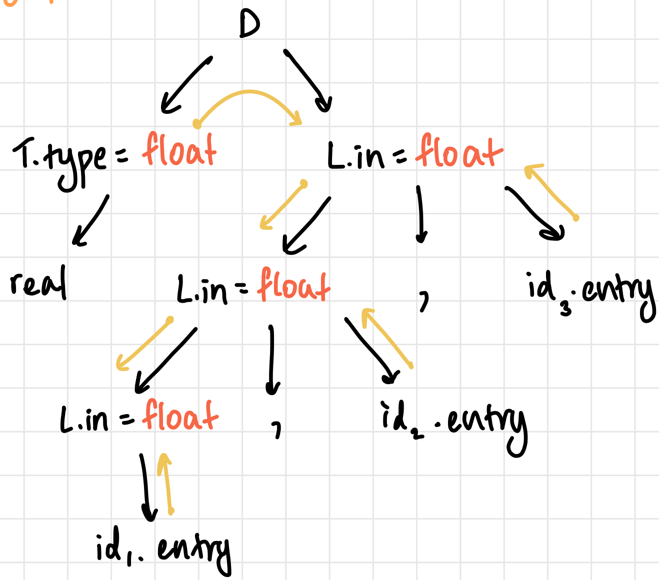
1. Parse tree



2. Annotated parse tree

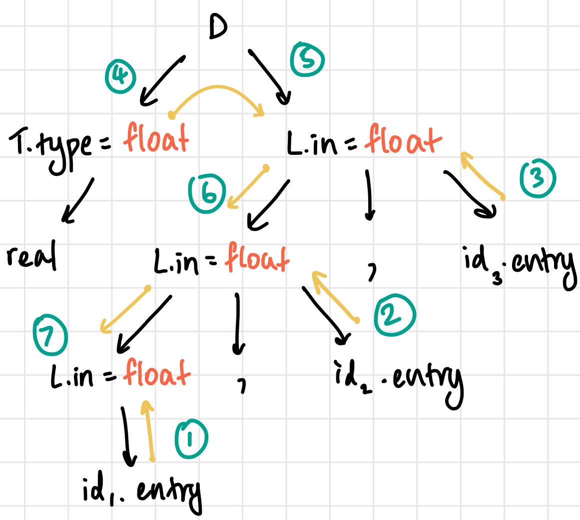


3. Dependency graph



TOPOLOGICAL SORT

- Evaluation order of semantic rules derived from topological sort derived from dependency graph
- Topological sort: ordering $m_1, m_2, \dots, m_k$ of nodes in a directed graph such that if $m_i \to m_j$, then m_i happens before m_j



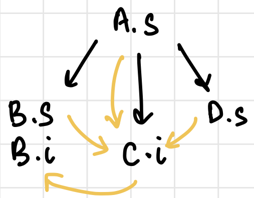
Problems

1. Fails if DG has cycles (need test for non-circularity)

2. Time-consuming

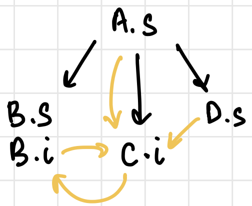
Q:- Will the following SDDs work with DG?

(i) A → BCD { C.i = A.s * B.s + D.s ;
B.i = C.i * 2 ; }



yes ∵ no cycle

(ii) A → BCD { C.i = A.s * B.i + D.s ;
B.i = C.i * 2 ; }



no ∵ cycle

Solutions

- Design SDD as S-Attributed SDD

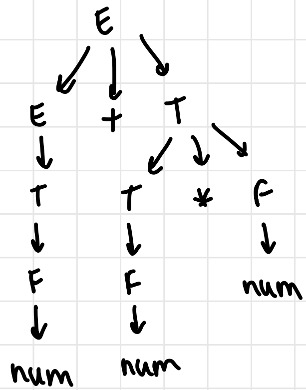
S-Attributed Definitions

- Can be evaluated by a bottom-up parser (avoiding construction of dependency graph)
- Parser keeps values of synthesized attributes in stack
- Whenever a reduction $A \rightarrow \alpha$ is made, attr for A computed from attr of α (which appear on stack)
- $\therefore$ translator for S-Attributed Definition can be implemented by extending stack of LR parser

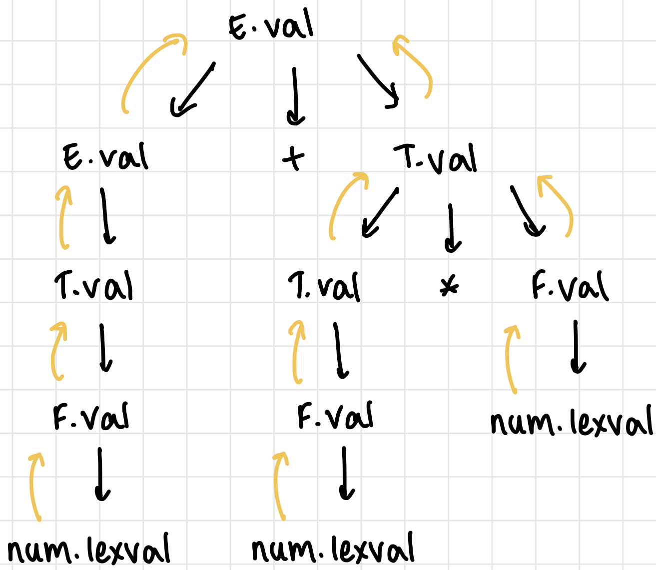
Q: Evaluate the following SDD for input 3+4*5

Production	Semantic Rule
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow num$	$\{ F.val = num.lexval \}$

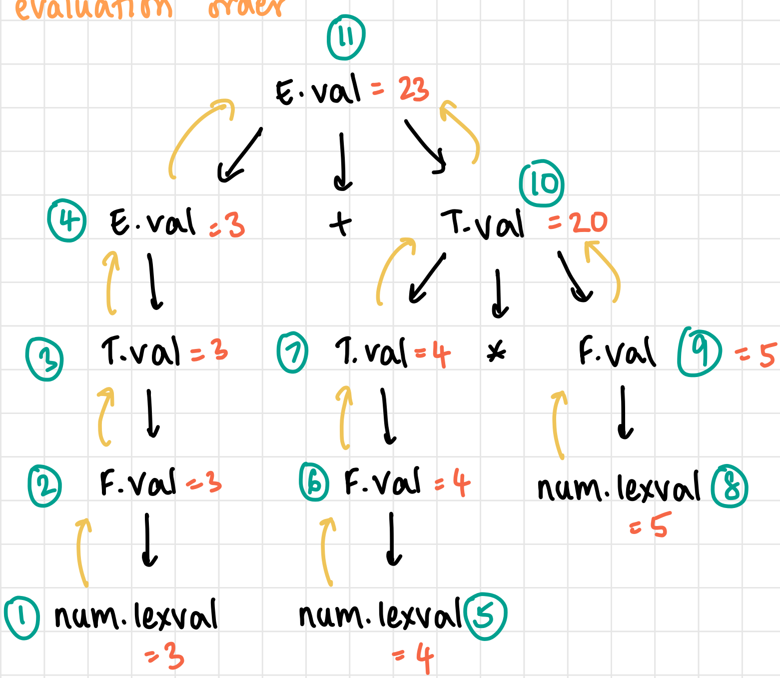
1. Parse Tree



2. Annotated Parse Tree + DG



3. Decide evaluation order

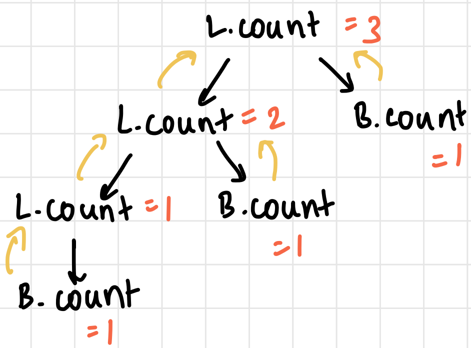


Q: Write SDDs to count no. of 1's in a Binary number

$L \rightarrow LB \mid B$
$B \rightarrow 0 \mid 1$

Production	Semantic Rule
$L \rightarrow L_1 B$	{L.count = L_1.count + B.count;}
$L \rightarrow B$	{L.count = B.count;}
$B \rightarrow 0$	{ B.count = 0; }
$B \rightarrow 1$	{ B. count = 1;}

101



Q: Write Syntax Directed Definitions to calculate no. of 0's in a binary no

$L \rightarrow LB \mid B$
$B \rightarrow 0 \mid 1$

Production	Semantic Rule
$L \rightarrow L_1 B$	{L.count = L_1.count + B.count;}
$L \rightarrow B$	{L.count = B.count;}
$B \rightarrow 0$	{ B.count = 1; }
$B \rightarrow 1$	{ B. count = 0;}

Q: Write SDDs to calculate no. of bits in a binary no.

$L \rightarrow LB \mid B$
$B \rightarrow 0 \mid 1$

Production	Semantic Rule
$L \rightarrow L_1 B$	{L.count = L_1.count + B.count;}
$L \rightarrow B$	{L.count = B.count;}
$B \rightarrow 0$	{ B.count = 1; }
$B \rightarrow 1$	{ B.count = 1;}

Q: Write SDD to convert binary to decimal

$L \rightarrow LB \mid B$
$B \rightarrow 0 \mid 1$

Production	Semantic Rule
$L \rightarrow L_1 B$	{L.val = L_1.val * 2 + B.val}
$L \rightarrow B$	{L.val = B.val;}
$B \rightarrow 0$	{ B.val = 0;}
$B \rightarrow 1$	{ B.val = 1;}

Q: Write SDDs to convert binary fraction to decimal fractions.

$L \rightarrow D.D$
$D \rightarrow DB \mid B$
$B \rightarrow 0 \mid 1$

Production	Semantic Rule
$L \rightarrow D_1 . D_2$	{L.val = D_1.val + D_2.val / (2^ D_2.count);}
$D \rightarrow D_1 B$	{D.val = D_1.val * 2 + B.val; D.count = D_1.count + 1; }
$D \rightarrow B$	{D.val = B.val; D.count = 1;}
$B \rightarrow 0$	{B.val = 0;}
$B \rightarrow 1$	{B.val = 1;}

Q: Write an SDD to count the no. of balanced parentheses

$S \rightarrow (S) \mid a$

Production	Sematic Rule
$S \rightarrow (S_1)$	{S.count = S_1.count + 1;}
$S \rightarrow a$	{S.count = 0;}

Q: Write SDD to convert infix to postfix

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

Production	Sematic Rule
$E \rightarrow E_1 + T$	{ printf("+");}

$E \to T$

$T \to T*F$ {printf("*");}

$T \to F$

$F \to num$ {printf("%d", num.lexval);}

Q: SDD for expr involving + and int or float operands.
(Determine type)

Grammar given

$E \to E+T \mid T$
$T \to num.num \mid num$

Production	Sematic Rule
$E \to E_1 + T$	{ if (E_1.type == float || T.type == float) { E.type = float; } else { E.type = int; } }
$E \to T$	{ E.type = T.type ; }

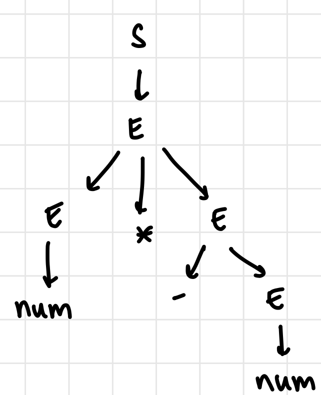
$T \rightarrow num.num$ $\{T.type = float;\}$

$T \rightarrow num$ $\{T.type = int;\}$

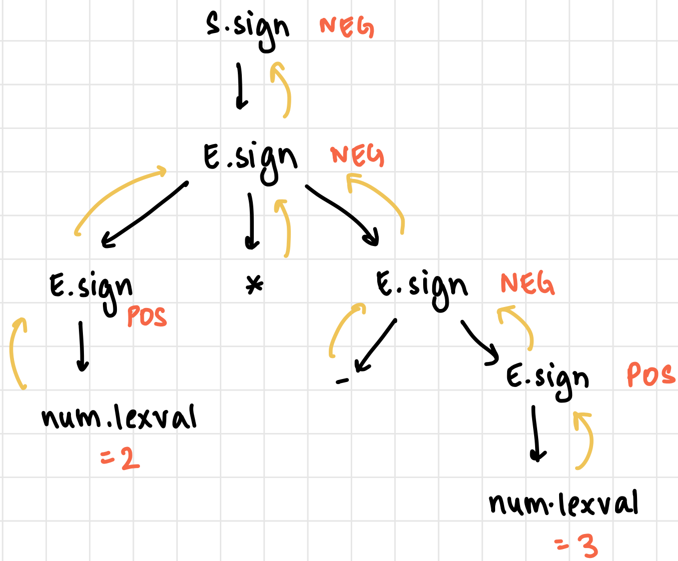
Q:. SDD to identify sign of evaluated expression (complete the table). Show parse tree for input 2*-3 (and annotated)

Production	Semantic Rule
$S \rightarrow E$	{S.sign = E.sign;}
$E \rightarrow num$	not necessary if num is unsigned { if (num.lexval < 0) E.sign = POS ; else E.sign = NEG; }
$E \rightarrow +E_1$	{E.sign = E_1.sign;}
$E \rightarrow -E_1$	{ if (E_1.sign == POS) E.sign = NEG; else E.sign = POS; }
$E \rightarrow E_1 * E_2$	{ if (E_1.sign == E_2.sign) E.sign = POS; else E.sign = NEG; }

Parse Tree



Annotated PT



L-Attributed SDD

- SDD is L-attributed if all attributes are either

1. Synthesized

2. Extended synthesized
   - Suppose prod $A \rightarrow X_1 X_2 \dots X_n$
   - Suppose inherited attribute $X_i.a$ computed from prod rule
   - Rule can only use
     (a) Inherited attrs of A
     (b) Inherited or synthesized attrs of $X_1, X_2, \dots, X_{i-1}$ (left)

3. Inherited or synthesized attr of X_i ST no cycles in DG are formed

- Formal definition: SDD is L-attributed if each inherited attr of X_i in $A \rightarrow X_1 X_2 \ldots X_i \ldots X_n$ depends only on

  1. Attrs of symbols to the left of X_i $(X_1, X_2, \ldots, X_{i-1})$

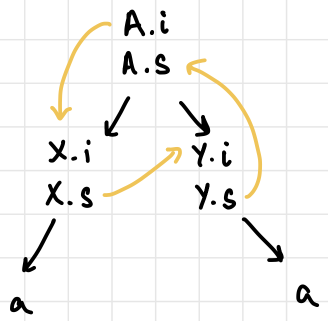
  2. Inherited attrs of A

- Inherited attributes of L-attributed definitions can be computed by a preorder traversal of parse tree

Q: Draw DG for the grammar

$$A \rightarrow XY$$
$$X \rightarrow a$$
$$Y \rightarrow a$$

$X.i := A.i$
$Y.i := X.s$
$A.s := Y.s$

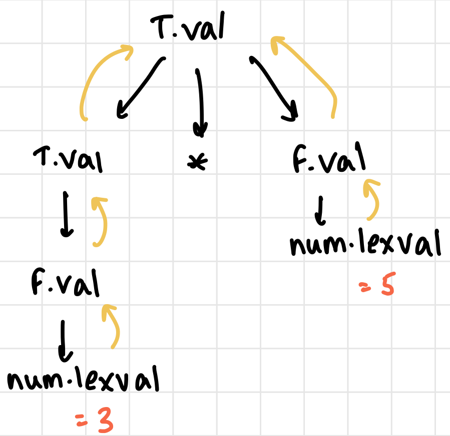


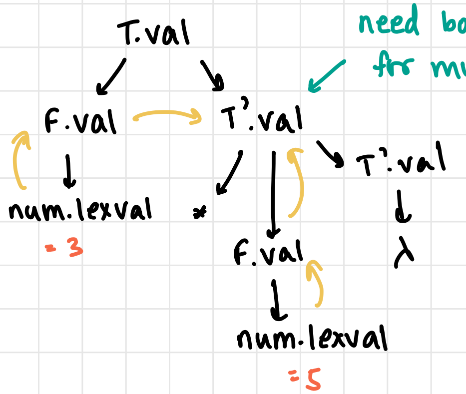
Grammars Suitable for TDP

input = $3*5$

$T \to T * F$
$T \to F$
$F \to num$
} left recursive

$T \to F\,T'$
$T' \to * F\,T' \mid \lambda$
$F \to num$
} suitable for TDP

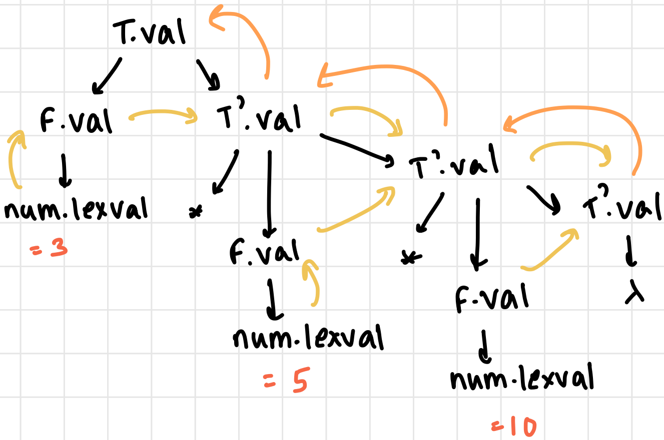




need both for mult

won't work for a*b*c?

suppose input = $3*5*10$ (left-associative)



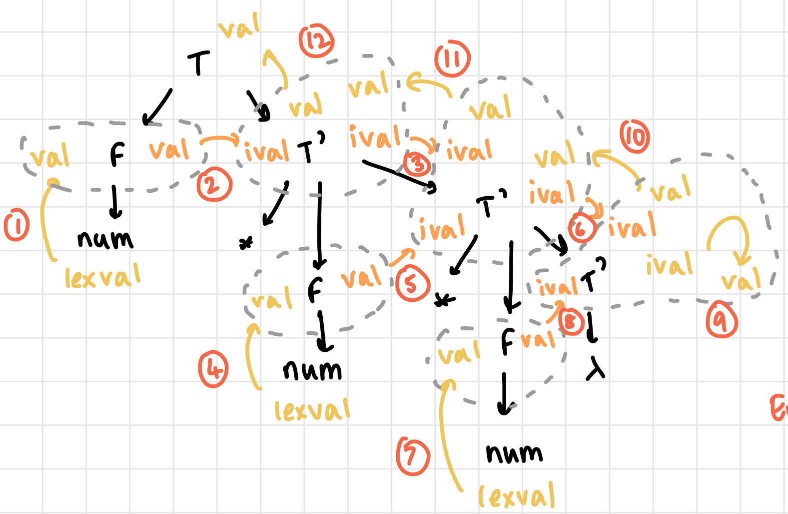
Q: Write semantic rules for L-attributed SDD.

synth : val
inherited : ival

$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to num$

$w = 3*5*10$

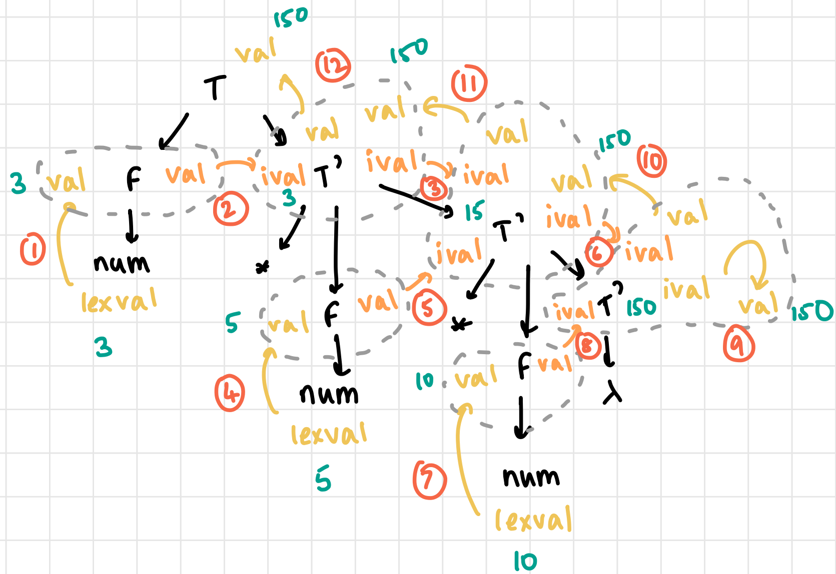
Production	Semantic Rules
$T \to FT'$	$\{T'.ival = F.val;$ $T.val = T'.val;\}$
$T' \to *FT_1'$	$\{T_1'.ival = T'.ival * F.val;$ $T'.val = T_1'.val;\}$
$T' \to \lambda$	$\{T_1'.val = T_1'.ival;\}$
$F \to num$	$\{F.val = num.lexval;\}$



Evaluation order

Annotated Parse Tree

$3 * 5 * 10$



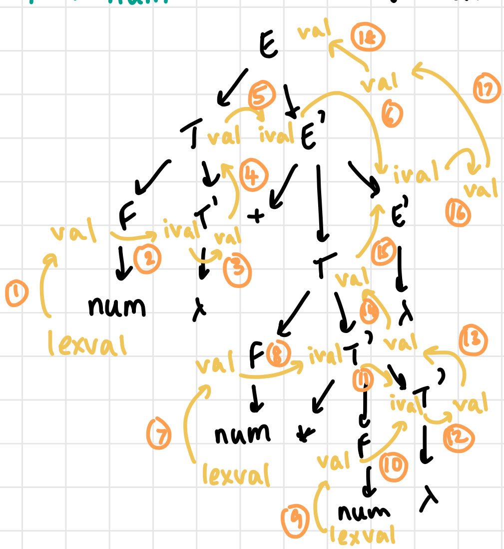
Q: Remove left factoring

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow num$

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \lambda$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \lambda$
$F \rightarrow num$

Q: Complete the semantic rules for the L-attributed SDD
Evaluate for the input $3+5*4$

Production	Semantic Rule
$E \rightarrow TE'$	{ $E'.ival = T.val;$ $E.val = E'.val;$ }
$E' \rightarrow +TE_1'$	{ $E_1'.ival = E'.ival + T.val;$ $E'.val = E_1'.val;$ }
$E' \rightarrow \lambda$	{ $E'.val = E'.ival;$ }
$T \rightarrow FT'$	{ $T'.ival = F.val;$ $T'.val = T'.val;$ }
$T' \rightarrow * FT_1'$	{ $T_1'.ival = T'.ival * F.val;$ $T'.val = T_1'.val;$ }
$T' \rightarrow \lambda$	{ $T'.val = T'.inval;$ }
$F \rightarrow num$	{ $F.val = num.lexval;$ }

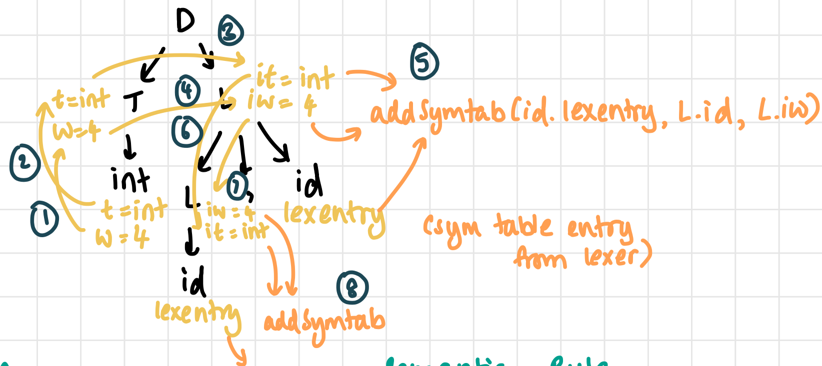


Q: Type declaration - add to symbol table

int a;
int a, b;

Attributes: type — t
storage — w

$D \to TL$
$L \to L, id \mid id$
$T \to int \mid float$



Production	Semantic Rule
$D \to TL$	{ L.it = T.it; L.iW = T.iw; }
$L \to L_1, id$	{ addSymtab(id.lexentry, L.id, L.iw); L_1.it = L.it; L_1.iw = L.iw; }
$L \to id$	{ addSymtab(id.lexentry, L.id, L.iw); }
$T \to int$	{ T.t = int; T.w = 4; }
$T \to float$	{ T.t = float; T.w = 8; }

Q: Type declaration with arrays

int $\longrightarrow$ t = int , w = 4

int[2] $\longrightarrow$ t = array(2, int) , w = 2*4

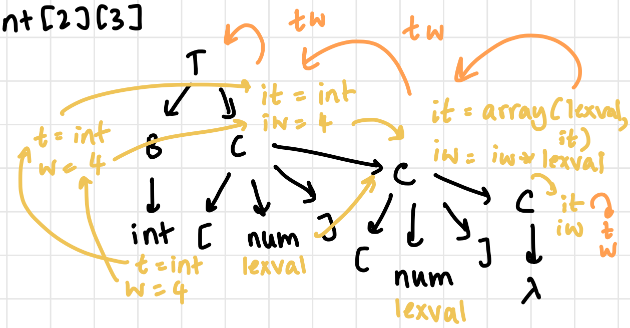
int[2][3] $\longrightarrow$ t = array(2, array(3, int)) , w = 2*3*4

input = int[2][3]

T → BC

B → int | float

C → [num] C | λ

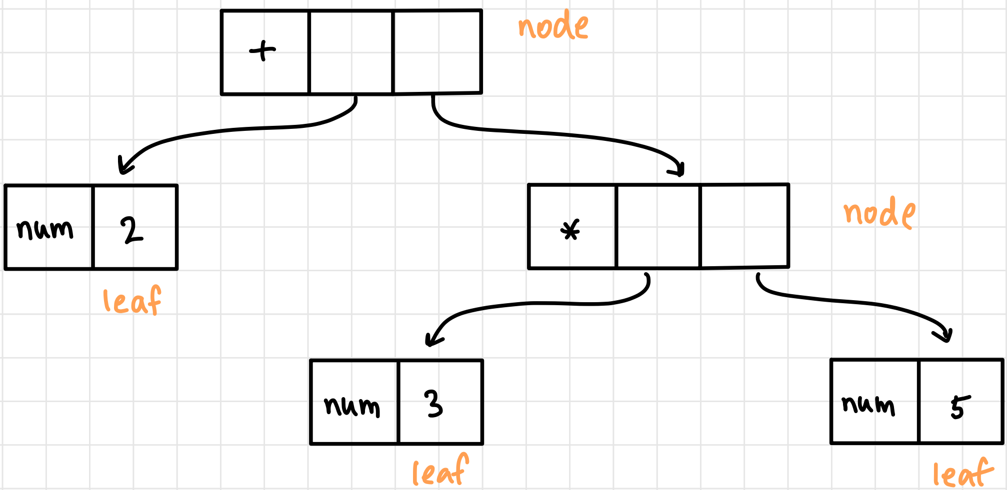


Production	Semantic Rules
T → BC	{ C.it = B.t ; C.iw = B.w ; }
B → int	{ B.t = int; B.w = 4; }
B → float	{ B.t = float; B.w = 8; }
C → [num] C_1	{ C_1.it = array(num.lexval, C.it); C_1.iw = num.lexval * C.iw; C.t = C_1.t ; C.w = C_1.w; }
C → λ	{ C.t = C.it ; C.w = C.iw ; }

ABSTRACT SYNTAX TREE GENERATION

2+3*5

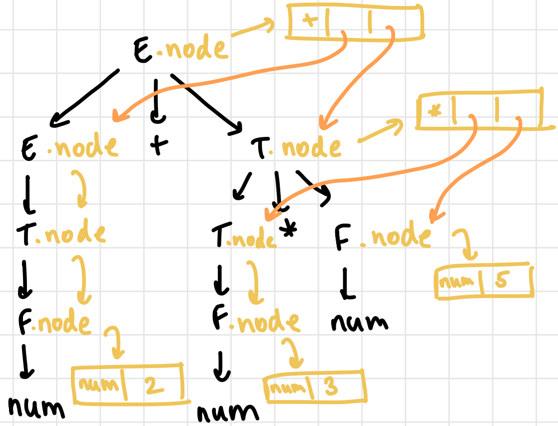
Linked list



Q: convert grammar to AST

2 + 3 * 5

E → E+T | T
T → T*F | F
F → num



Production	Semantic Rules
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node);
$E \rightarrow T$	E.node = T.node;
$T \rightarrow T_1 * F$	T.node = new Node('*', T_1.node, F.node);
$T \rightarrow F$	T.node = F.node;
$F \rightarrow$ num	F.node = new Leaf(num, num.lexval);

Three Address Code Generation

- Intermediate (machine-independent) code generation
- Attributes: addr, code
- Method to create temp variables: new Temp() = t1, next t2, so on

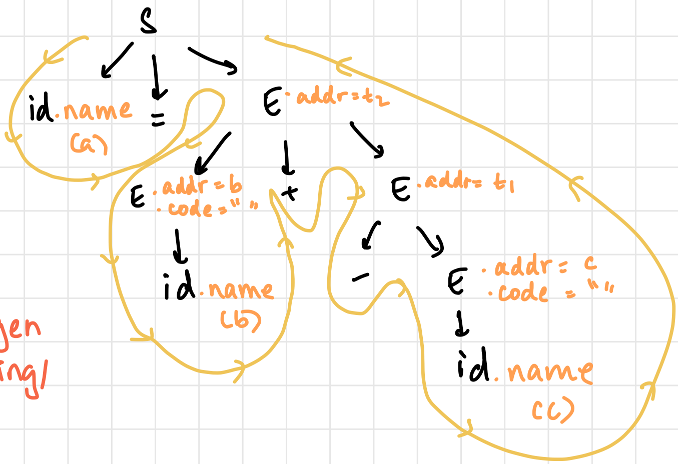
a = b + − c

t1 = minus c
t2 = b + t1
a = t2

Q: SDD to generate intermediate code for expressions

$S \to id = E$
$E \to E + E$
$E \to -E$
$E \to (E)$
$E \to id$

input = a = b + -c



on-the-fly code generation: print / gen code without storing / carrying over

Production	Semantic Rules
$S \to id = E$	S.code = gen(id.name "=" E.addr);
$E \to E_1 + E_2$	E.addr = new Temp(); E.code = gen(E.addr "=" E_1.addr "+" E_2.addr);
$E \to -E_1$	E.addr = new Temp(); E.code = gen(E.addr "=" "-" E_1.addr);
$E \to (E_1)$	E.code = E_1.code; E.addr = E_1.addr;
$E \to id$	E.addr = id.name; E.code = " ";

Q: Using concat operator – ||

Production	Semantic Rules
$S \to id = E$	S.code = E.code || gen(id.name "=" E.addr);
$E \to E_1 + E_2$	E.addr = new Temp(); E.code = E_1.code || E_2.code || gen(E.addr "=" E_1.addr "+" E_2.addr);
$E \to -E_1$	E.addr = new Temp(); E.code = E_1.code || gen(E.addr "=" "-" E_1.addr);
$E \to (E_1)$	E.code = E_1 code; E.addr = E_1.addr;
$E \to id$	E.addr = id.name; E.code = "";

Q: Conditional branching

$\left.\begin{array}{l}\text{C.true: where to go when C is true} \\ \text{C.false: where to go when C is false}\end{array}\right\}$ (inherited attrs

$\left.\begin{array}{l}\text{C.addr:} \\ \text{C.code:}\end{array}\right\}$ synth

$C \to E_1 \text{ rel } E_2$
rel $\to$ >
rel $\to$ <

Production	Semantic Rules
$C \rightarrow E_1$ rel E_2	{ C.addr = new Temp(), C.code = gen(C.addr "=" E_1.addr rel.op E_2.addr) || gen("if" C.addr "goto" C.true) || gen("goto" C.false); }
rel $\rightarrow$ >	{ rel.op = ">"; }
rel $\rightarrow$ <	{ rel.op = "<"; }

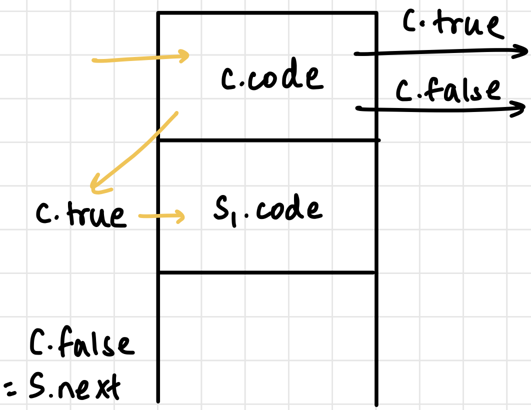
Inherited and Synthesized Attributes

	syn	inh
E	code addr	—
S	code addr	next
C	code addr	true false

Block Diagram

next: what follows

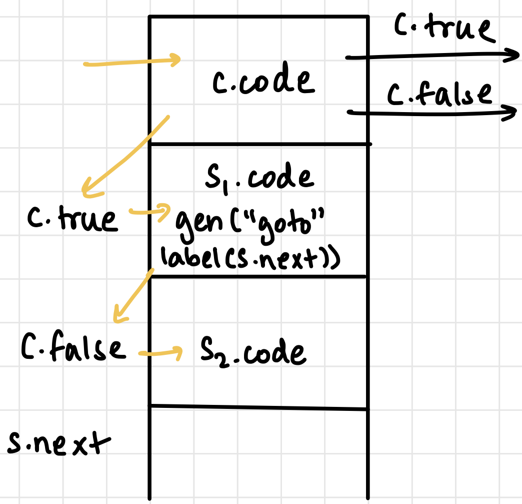
$S \rightarrow$ if (C) S_1



c.false = S.next;
c.true = new Label();
S_1.next = S.next;

S.code = C.code ||
label (C.true) ||
S_1.code

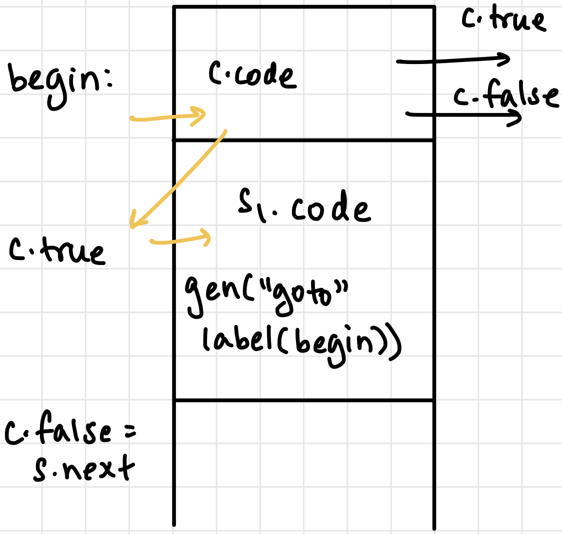
$S \rightarrow$ if (C) S_1 else S_2



c.true = new Label();
c.false = new Label();
S_1.next = S.next;
S_2.next = S.next;

S.code = C.code ||
label (C.true) ||
S_1.code ||
gen("goto" label (S.next))
|| label (C.false) ||
S_2.code

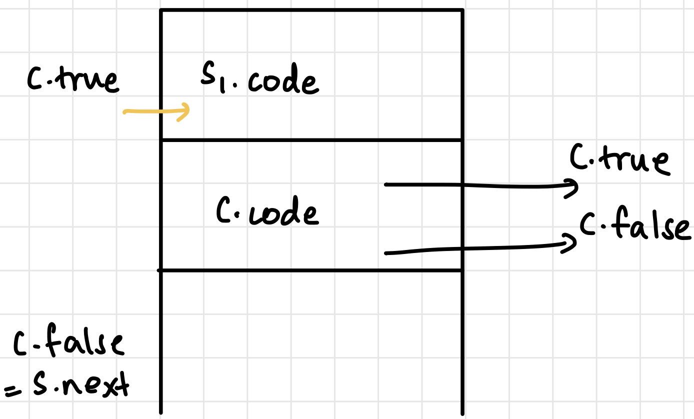
$S \rightarrow$ while (C) S_1



begin = new Label();
C.true = new Label();
C.false = S.next;
S_1.next = begin;

S.code =
label(begin) ||
C.code ||
label (C.true) ||
S_1.code ||
gen("goto" label(begin));

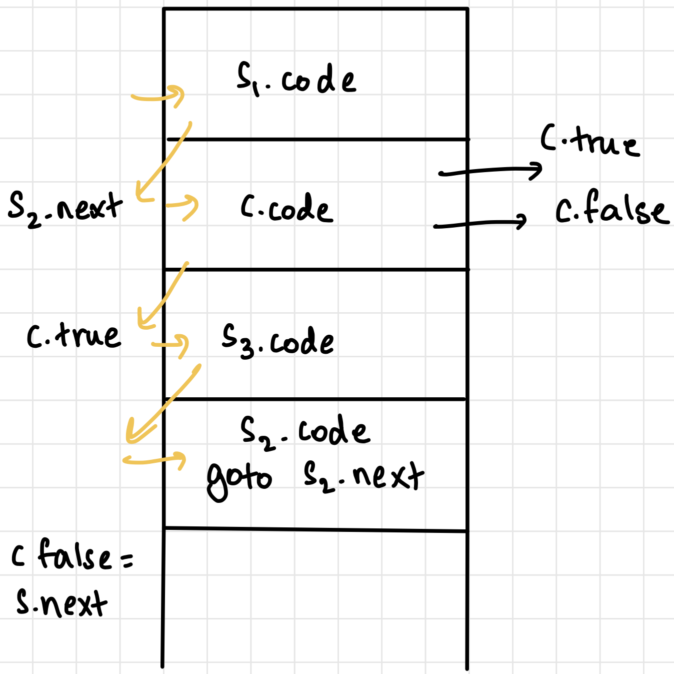
$S \rightarrow$ do (S_1) while (C)



C.true = new Label();
C.false = S.next

S.code = label (C.true) ||
S_1.code ||
C.code

$S \rightarrow$ for (S_1 ; C ; S_2) S_3



C.true = new Label();
C.false = S.next;
S_2.next = new Label();

S.code =
S_1.code ||
label(S_2.next) ||
C.code ||
label(C.true) ||
S_3.code ||
S_2.code ||
gen("goto" S_2.next);

Full program

S → id = E | if (C) S | if (C) S else S | while (C) S |
do (S) while (C) | for (S ; C ; S) S

E → E + T | T
T → T * F | F
F → id

C → E rel E
rel → > | < | <= | >= | == | !=

Syntax-Directed Translation

- Meaning of an input sentence is related to its syntactic structure (parse tree)
- Two notations for attaching semantic rules associated with grammar productions

  1. Syntax directed definitions: high-level (what we just did)
  2. Translation schemes: more implementation-oriented; order of evaluation of semantic rules

Translation Scheme

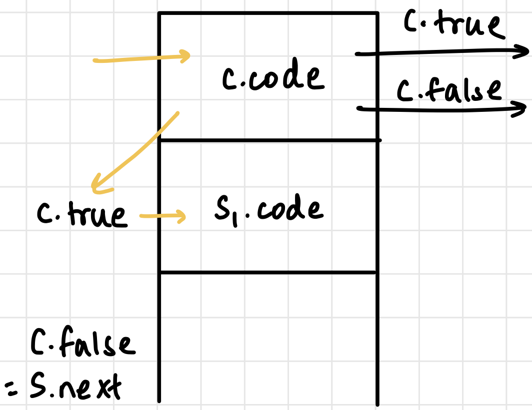
- Context-free grammar
  - Attrs associated with grammar symbols
  - Semantic actions within {} at RHS
- Yacc uses it

DESIGNING TRANSLATION SCHEME

- Make sure an attribute value is available when a semantic action is executed
- If S-attributed SDD, action can be directly put at the end of the production
- If L-attributed SDD,
  1. Inherited attr for a symbol in RHS must be computed in an action before the symbol
  2. Synthesized attr for a NT at the LHS can only be computed when all the attrs it references have been computed

Q: For the SDD, find the SDTs (Translation scheme)

$S \rightarrow$ if (C) S_1 $\begin{cases} C.true = new\ Label(); \\ C.false = S_1.next = S.next; \\ S.code = C.code \parallel Label(C.true) \parallel \\ S_1.code; \end{cases}$



inherited: C.true, c.false, S.next

$S \rightarrow$ if ($\underbrace{\begin{Bmatrix} C.true = new\ Label(); \\ C.false = S.next; \end{Bmatrix}}$ C) $\{S_1.next = S.next;\}$ S_1 $\{S.code = C.code \parallel Label(C.true) \parallel S_1.code\}$

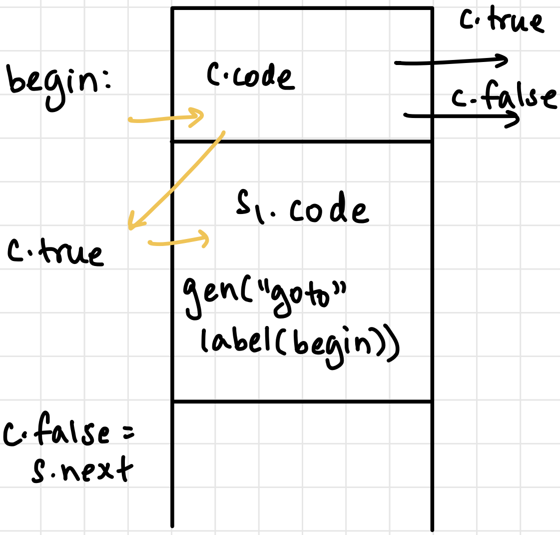
inherited — inherited — synthesized

Note: synth semantic rules need not appear just before they occur; can appear anywhere before

$S \rightarrow$ if ($\begin{Bmatrix} C.true = new\ Label(); \\ C.false = S.next; \\ S_1.next = S.next; \end{Bmatrix}$ C) S_1 $\begin{Bmatrix} S.code = C.code \parallel \\ Label(C.true) \parallel S_1.code \end{Bmatrix}$

Q: For the SDD, convert to SDTs

$S \rightarrow$ while $(C)\ S_1$



begin = new Label();
c.true = new label();
c.false = S.next;
S_1.next = begin;

S.code =
label(begin) ||
c.code ||
label (c.true) ||
S_1.code ||
gen("goto" label(begin));

$S \rightarrow$ while ({ begin = new Label(); c.true = new Label(); c.false = s.next(); S_1.next = begin; } C) S_1 { S.code = Label(begin) || C.code || Label (C.true) || S_1.code || gen("goto" Label((begin)); }

SDT Implementation

1. Ignore actions and produce parse tree of input
2. Add dummy nodes for all actions exactly how the appear in the production
3. Perform preorder traversal and evaluate actions in that order

Q: Consider the following SDTs

$S \rightarrow ER$

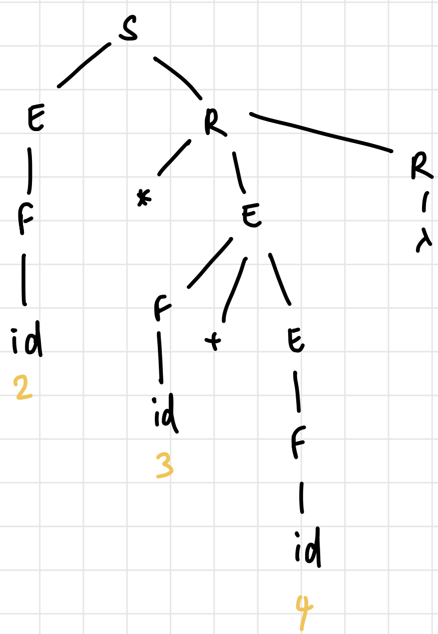
$R \rightarrow *E$ {printf("*");} R | λ

$E \rightarrow F+E$ {printf("+");} | F

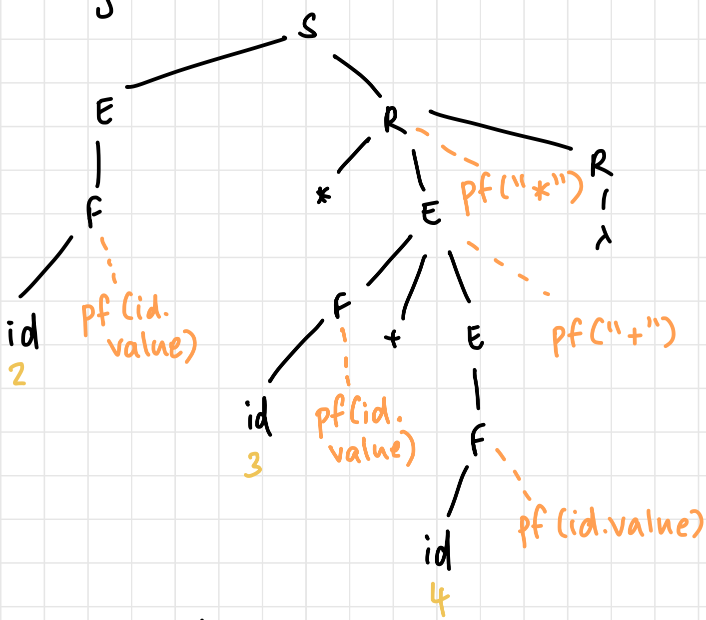
$F \rightarrow (S)$ | id {printf("%s", id.value);}

input = 2*3+4

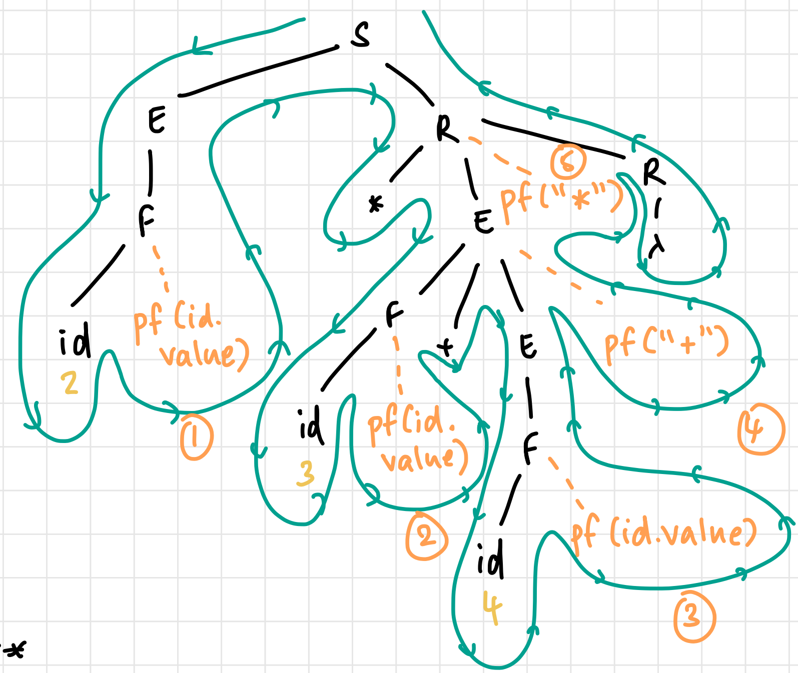
1. Parse tree without actions



2. With dummy nodes



3. Preorder traversal



2 3 4 + *

Q: SDTs for infix to prefix

$E \rightarrow$ {printf("+");} E_1 + T

$E \rightarrow T$

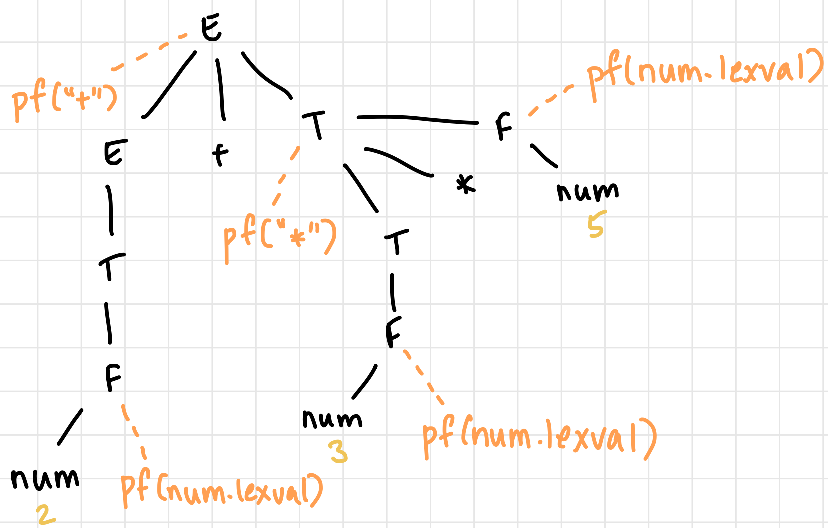
$T \rightarrow$ {printf("*");} T_1 * F

$T \rightarrow F$

$F \rightarrow$ num {printf("%d", num.lexval);}

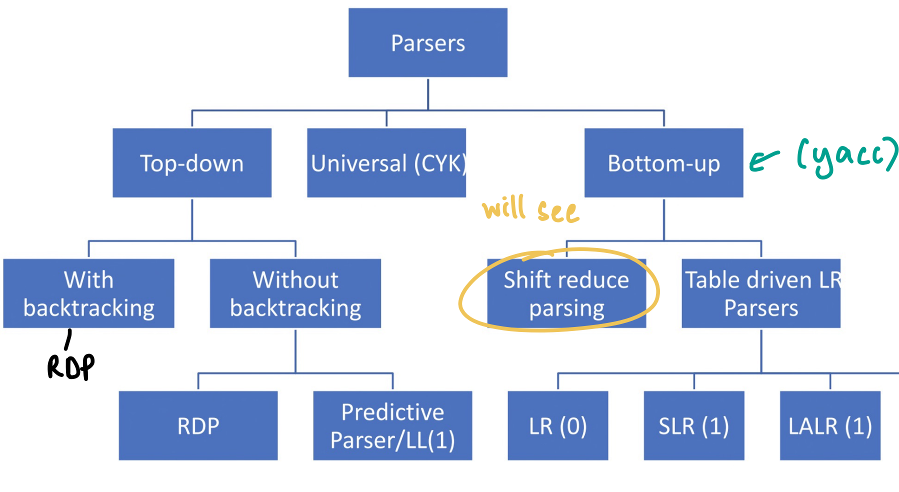
$F \rightarrow$ id {printf("%d", id.lexval);}

input = 2+3*5



+2*35

Translation During Parsing



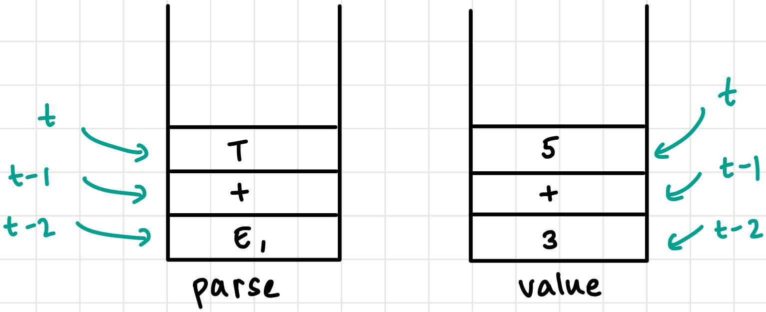
- Two types of SDTs

  (i) Synth attributes only — postfix SDTs
  (ii) Synth + inherited attributes — prefix SDTs

- 2 stacks: parse stack and value stack (or one stack with 2 fields per entry — state and val)

$$E \rightarrow E_1 + T \; \{ E.val = E_1.val + T.val; \}$$

input = 3+5



SDTs : $\left\{ \begin{array}{l} \underset{E}{S[top-2].val} \underset{=}{=} \underset{E}{S[top].val} \underset{+}{+} \underset{T}{S[top-2].val}; \\ \underset{\text{reassign top}}{top = top-2;} \end{array} \right\}$

{ \$\$ = \$1+\$2; } → yacc

Q: Postfix SDT for simple desk calculator

$E \rightarrow E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \rightarrow F$	
$F \rightarrow (\,E\,)$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \rightarrow \textbf{digit}$	

Bottom-Up Parser

- Perform action only during reduction
- Problem: actions in between RHS symbols

$$A \rightarrow X\ \{\ \}\ Y\ \{\ \}\ Z\ \{\ \}$$

- Yacc internally: for every embedded action, introduces a Marker non-terminal / dummy NT

$$A \rightarrow X \overset{①}{\{\,\}}\ Y \overset{②}{\{\,\}}\ Z \overset{③}{\{\,\}}$$

$$A \rightarrow X\ M\ Y\ N\ Z \overset{③}{\{\,\}}$$

marker NT's

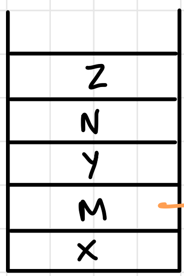
- Add λ-productions for the marker NT's with their actions

$$A \rightarrow X\ M\ Y\ N\ Z \overset{③}{\{\,\}}$$

$$M \rightarrow \lambda \overset{①}{\{\,\}}$$

$$N \rightarrow \lambda \overset{②}{\{\,\}}$$

- Stack sees entire RHS and can reduce



without seeing anything, reduce λ to M and perform action {2}

Parser Stack Structure

Stack record	***A***	***Synthesized attributes of A***
Record of Marker A	**Inherited attributes of A**	

C	C.addr	C.code
while		

Q: Parse using S/R parser

$S \to$ while (C) { S }

page 34

$S \to$ while (M C) N S_1 { S.code = . . . }

$M \to \lambda$ { begin = new Label(); C.true = new Label();
C.false = S.next(); }

$N \to \lambda$ { S_1.next = begin; }

input = while (C) S_1 \$

Stack	Input Buffer	Action
\$	while (c) S_1 \$	shift while
\$ while	(c) S_1 \$	shift (

while
\$

Stack	Input Buffer	Action
\$ while (	c) S_1 \$	Reduce $M \to \lambda$ and execute action

(
while
\$

{ begin = new Label(); c.true = new Label(); c.false = S.next(); }

Stack	Input Buffer	Action
\$ while (M	c) S_1 \$	shift c

M	begin	c.false	c.true
(			
while			
\$			

Stack	Input Buffer	Action
$ while (M C	) S_1 $	shift)

<table>
<tr><td colspan="2">C</td><td colspan="2">C.code</td></tr>
<tr><td>M</td><td>begin</td><td>C.false</td><td>C.true</td></tr>
<tr><td colspan="4">(</td></tr>
<tr><td colspan="4">while</td></tr>
<tr><td colspan="4">$</td></tr>
</table>

Stack	Input Buffer	Action
$ while (M C)	S_1 $	Reduce $N \to \lambda$ and execute action

{S_1.next = begin; }

<table>
<tr><td colspan="4">)</td></tr>
<tr><td colspan="2">C</td><td colspan="2">C.code</td></tr>
<tr><td>M</td><td>begin</td><td>C.false</td><td>C.true</td></tr>
<tr><td colspan="4">(</td></tr>
<tr><td colspan="4">while</td></tr>
<tr><td colspan="4">$</td></tr>
</table>

Stack	Input Buffer	Action
$ while (M C) N	S_1 $	shift S_1

<table>
<tr><td>N</td><td colspan="3">S_1.next = s[t-3].begin</td><td>← t</td></tr>
<tr><td colspan="4">)</td><td>← t-1</td></tr>
<tr><td colspan="2">C</td><td colspan="2">C.code</td><td>← t-2</td></tr>
<tr><td>M</td><td>begin</td><td>C.false</td><td>C.true</td><td>← t-3</td></tr>
<tr><td colspan="4">(</td><td></td></tr>
<tr><td colspan="4">while</td><td></td></tr>
<tr><td colspan="4">$</td><td></td></tr>
</table>

Stack	Input Buffer	Action
$ while (M C) N S_1	$	Reduce $S \rightarrow$ while(MC)NS and execute action

S_1	S_1.code			← t
N	S_1.next = s[t-3].begin			← t-1
)				← t-2
C	C.code			← t-3
M	begin	C.false	C.true	← t-4
(				← t-5
while				← t-6
$				

{ S.code = begin ||
C.code ||
C.true ||
S_1.code }

or

s[t-6].code = s[t-4].begin ||
s[t-3].code ||
s[t-4].true ||
s[t].code ;
t = t-6;

Stack	Input Buffer	Action
$ S	$	Accept

S	S.code
$	