University of Kansas | Drew Davidson

EECS 665
COMPILER CONSTRUCTION

Lecture 5
Syntax-Directed Definition

I *Think* We're Done with Review

Lecture 5 – Syntax-Directed Definition


COMPILER
Code Generation
Execution
Runtime Environment
Optimization
Intermediate Representation
SDD
Parsing
Semantics
Lexical Analysis
Syntactic Definiton
MOLASSES SWAMP
GLOPPY
CANDY CASTLE
KING KANDY
QUEEN FROSTINE
GRAMMA NUTT
PRINCESS LOLLY
LORD LICORICE
LICORICE CASTLE
JOLLY
PLUMPY

Lecture Outline

Syntax-Directed Definition

Recall Syntactic Ambiguity

Assigning Meaning to (Parse) Trees

- Tree translation intuition
- Introduce Syntax-Directed Definition

Tools for SDD

- Bison

Syntax-Directed Definition

Last Time

Review Lecture 4 –Syntactic Ambiguity

Recognizing Context-Free Grammars

- The parser wants a parse tree

Some Challenges in Syntactic Analysis

- Ambiguous Syntax
  - Precedence
  - Associativity

Syntactic Definition

Last Time

Review Lecture 4 –Syntactic Ambiguity

Force precedence constraints

Force associativity constraints

E := E **minus** E
E := E **times** E
E := E **pow** E

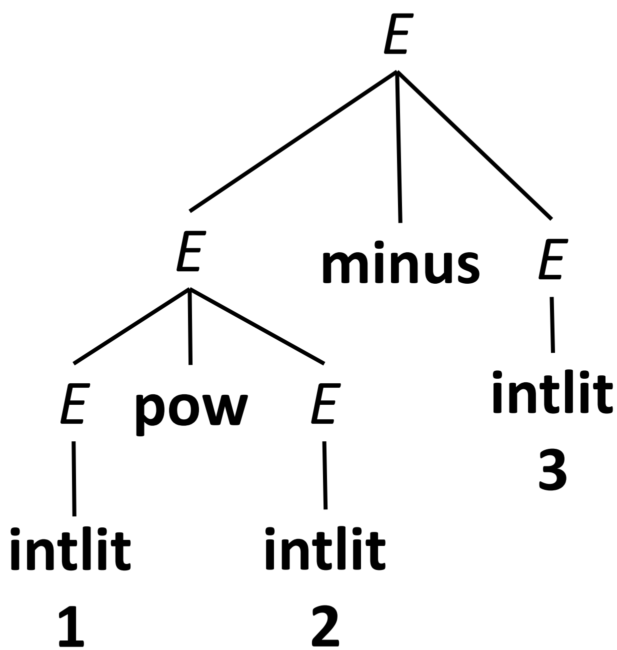
E := *E* **minus** *E*
| *T*
T := *T* **times** *T*
| *F*
F := *F* **pow** *F*
| *G*
G := **intlit**

E := *E* **minus** *T*
| *T*
T := *T* **times** *F*
| *F*
F := *G* **pow** *F*
| *G*
G := **intlit**

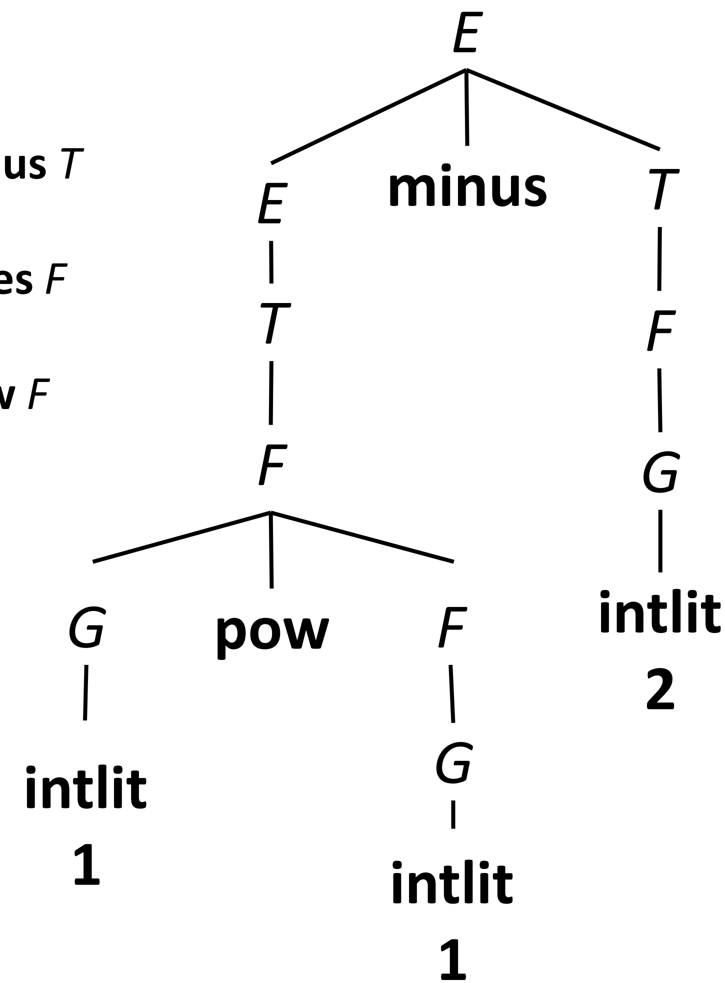
Syntactic Definition

Ensure Bad Exprs are Invalid Syntax

Working with Parse Trees



E := *E* **minus** *T*
    | *T*
T := *T* **times** *F*
    | *F*
F := *G* **pow** *F*
    | *G*
G := **intlit**



Lecture Outline

Preview Lecture 5 – Syntax-Directed Translation

Recall Syntactic Ambiguity

Assigning Meaning to (Parse) Trees

- Tree translation intuition
- Introduce Syntax-Directed Definition

Tools for SDD

- Bison

Syntax-Directed Definition

Benefits of Parse Trees

Assigning Meaning to Parse Trees

All of the known methods for defining the meaning of computer programs were based on rather intricate algorithms having roughly the same degree of complexity as compilers, or worse. This was in stark contrast to Chomsky's simple and elegant method of syntax definition via context-free grammars. As Dr. Caracciolo said, "How simple to realize [semantic correctness] if you write a procedure. The problem is, however, to find a metalanguage for doing that in a declarative way, not in an operational way" [3].

Benefits of Parse Trees

Assigning Meaning to Parse Trees

- Impose structure on tokens
- Easy to specify
- Easy to process

good algorithms
are known

More generally:
Trees are great data structures
for nesting relationships

Two Ways of Thinking About Trees

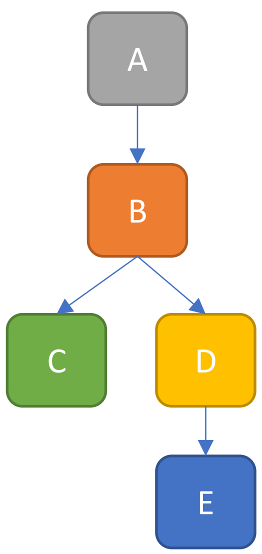
Working with Trees

Compilers use insights from both views

As a type of graph

Root is a node, children are successors

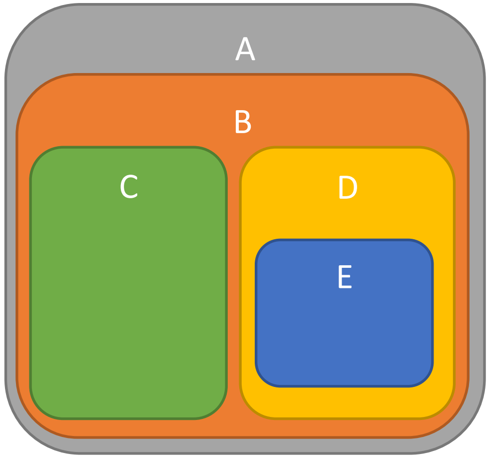
Work "root-down" vs "leaves-up"



As a type of nesting

Root is a tree, children are inside

Work "outside in" vs "inside out"

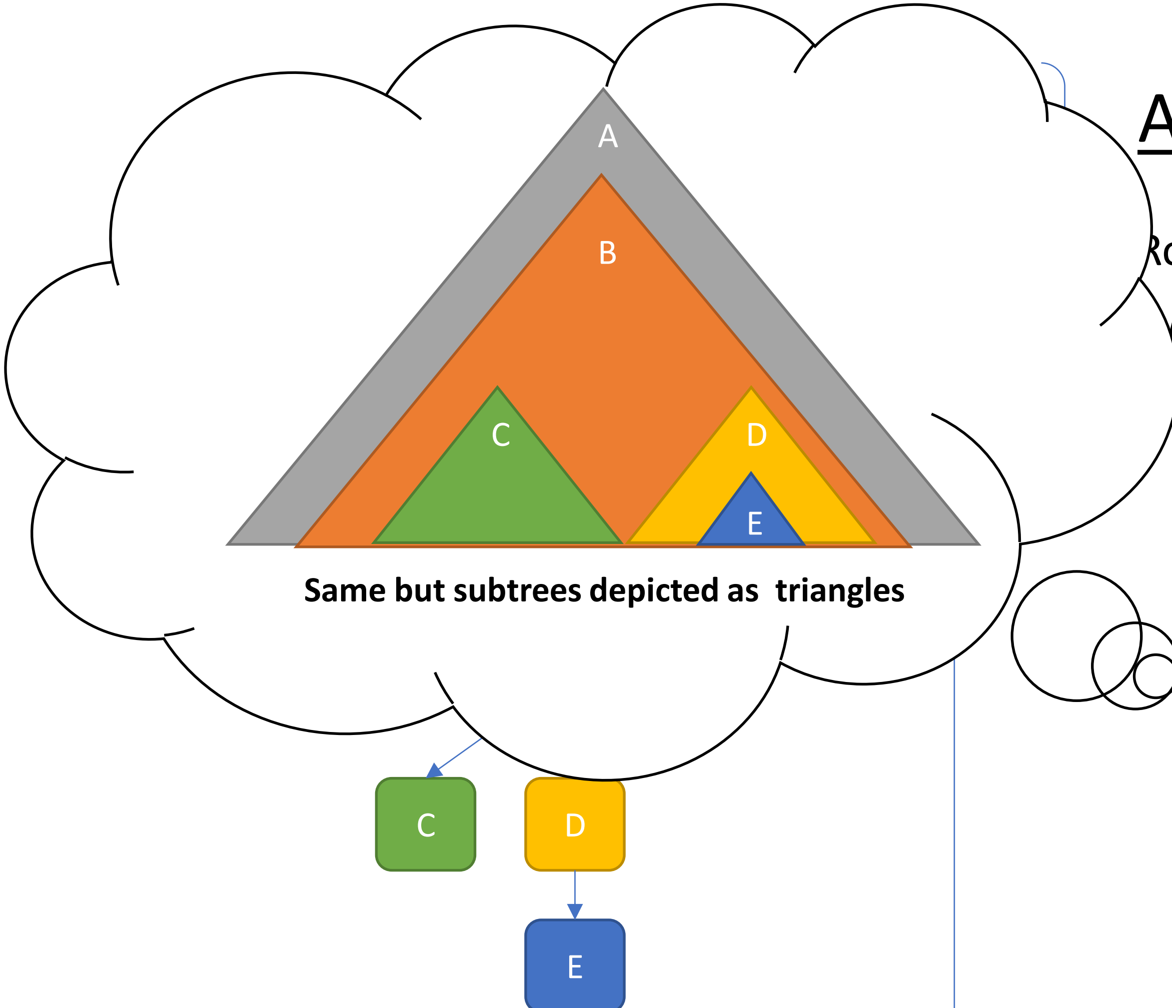


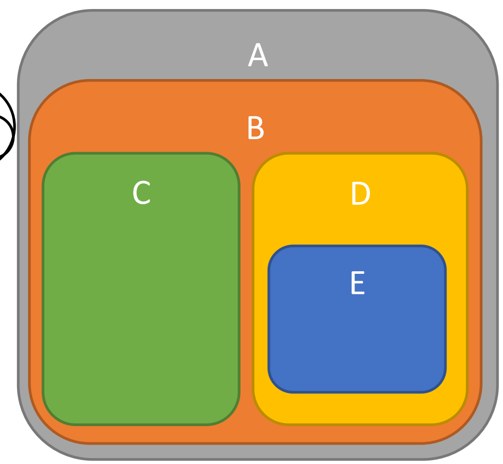
Two Ways of Thinking About Trees

Working with Trees

<u>As a type of nesting</u>

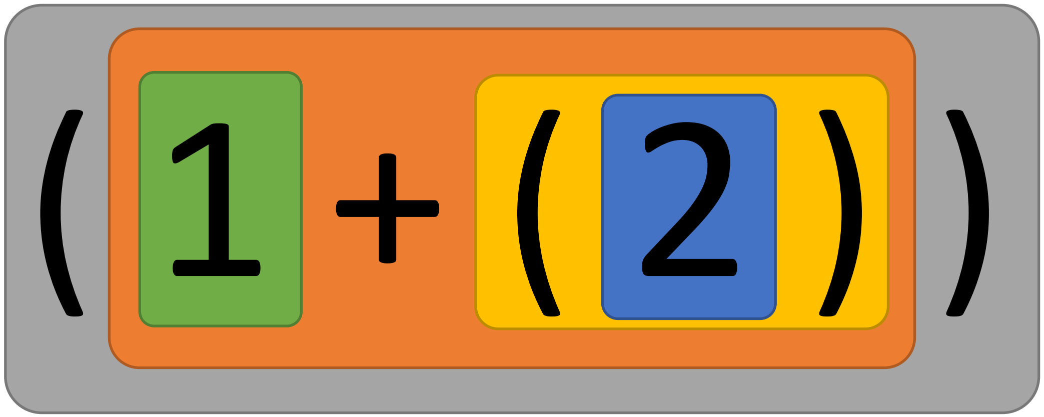
- Root is a tree, children are inside
- Work "outside in" vs "inside out"

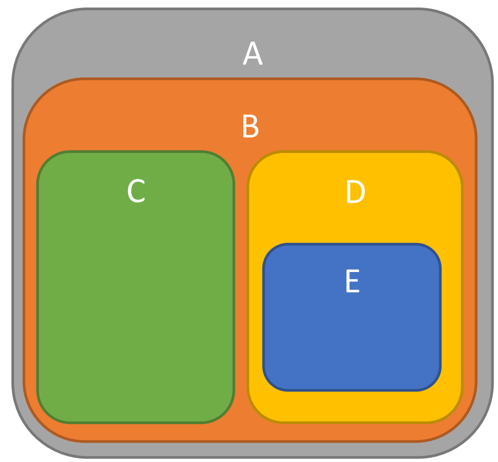




Trees as a Nesting

Working with Trees





A = (B)

B = C + D

C = 1

D = (E)

E = 2

Trees as a Nesting

Working with Trees

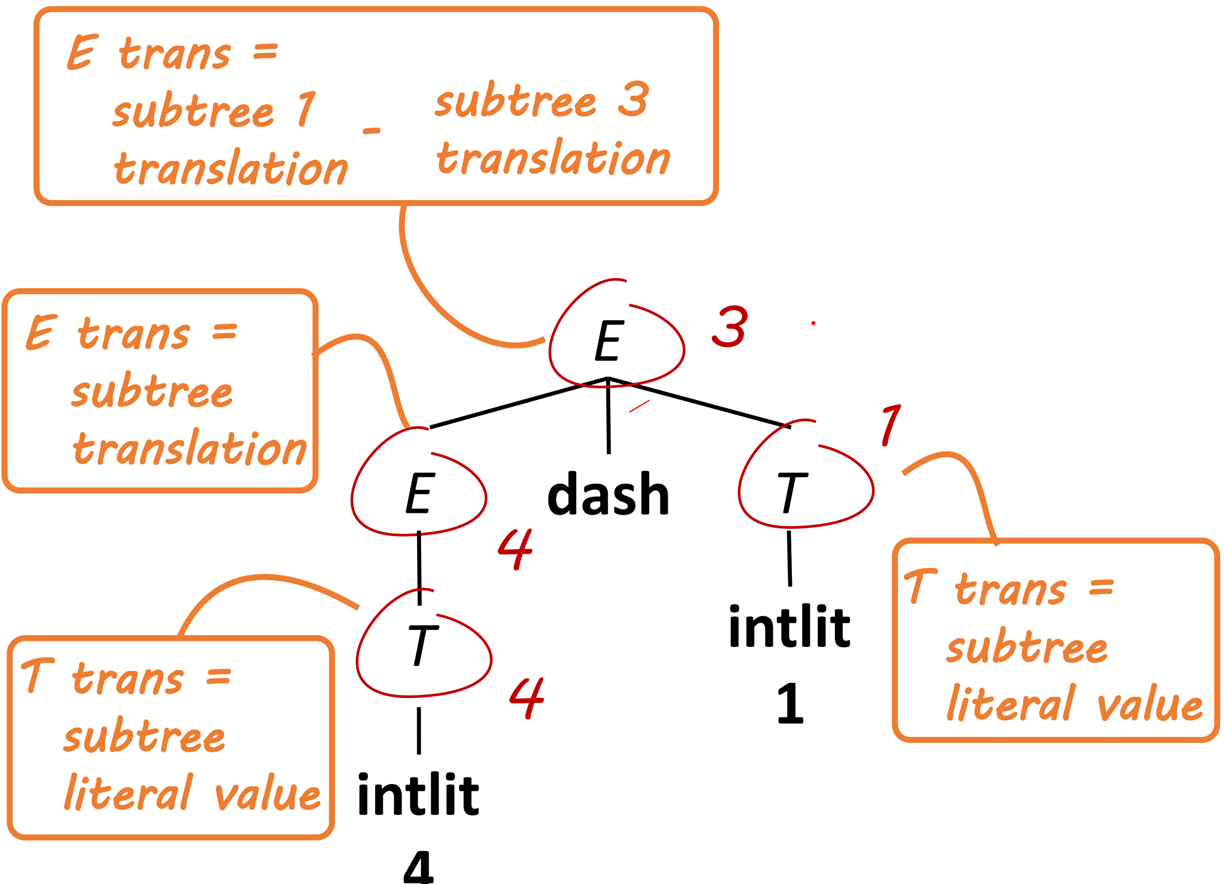
also 10, 11]. If we know the meaning of α and the meaning of β, then we know the meanings of such things as $\alpha + \beta$, $\alpha - \beta$, $\alpha \times \beta$, α/β, and (α). Thus the meaning of an arbitrarily large expression can be synthesized in a straightforward way by recursively building up the meanings of its subexpressions.

Assigning Meaning to Subtrees

Processing Parse Trees

Goal:
Translation is the value of the expression

Assign a ***translation*** for each node / subtree



Grammar:

E ::= *E* **dash** *T*

| *T*

T ::= **intlit**

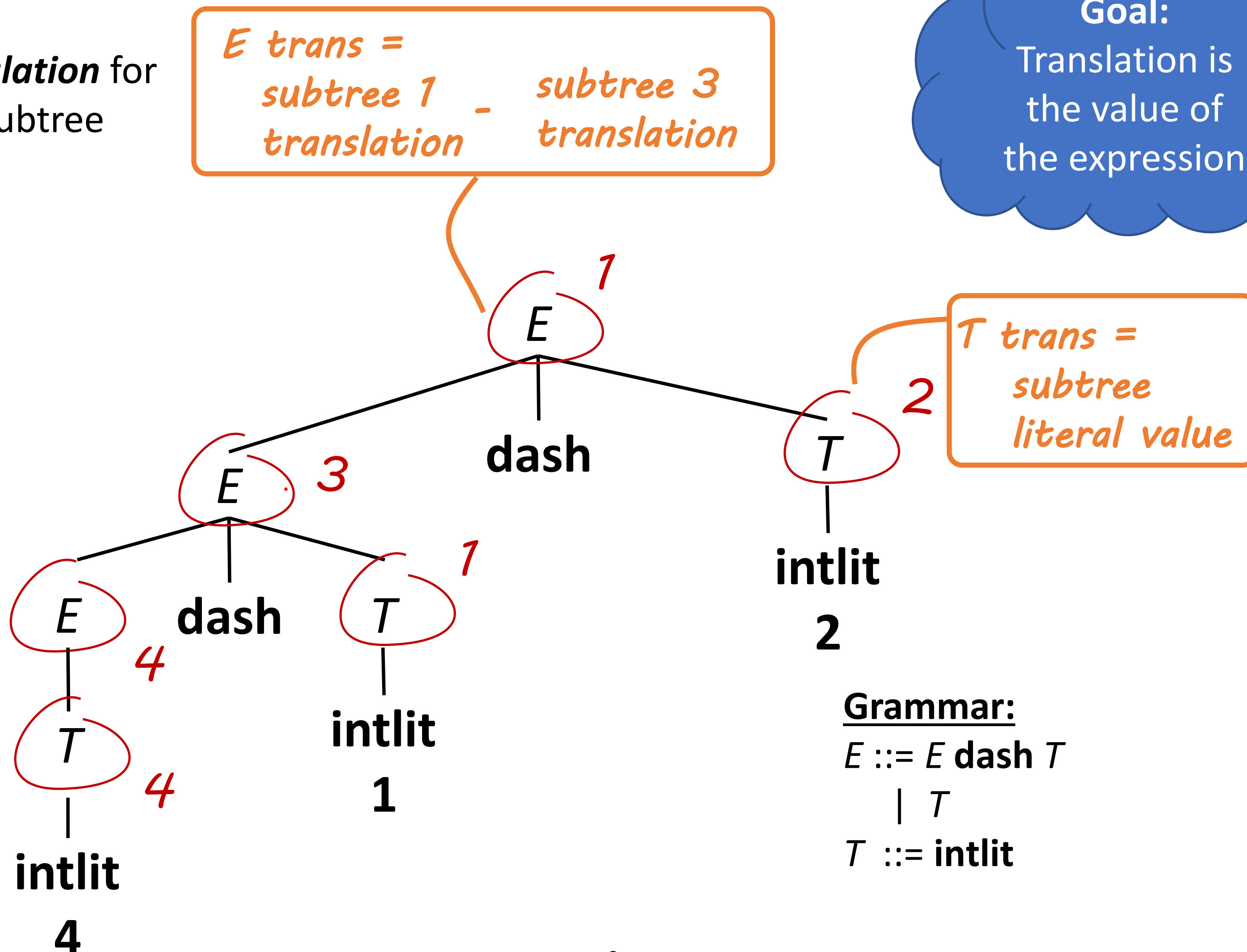
Input Expression: 4 – 1

Assigning Meaning to Subtrees

Processing Parse Trees

Assign a ***translation*** for each node / subtree

Goal:
Translation is the value of the expression



Grammar:

E ::= *E* **dash** *T*

| *T*

T ::= **intlit**

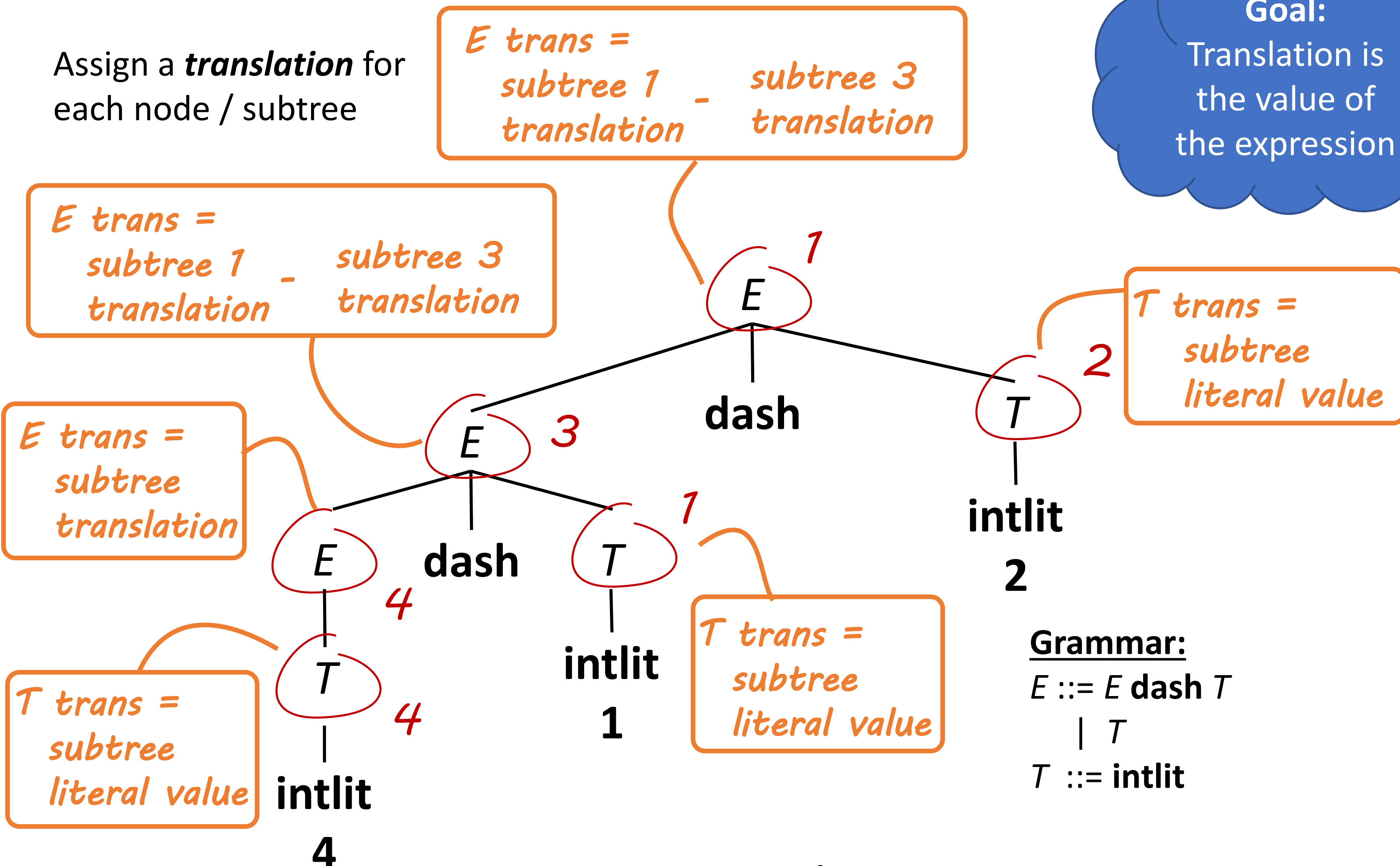
Input Expression: 4 – 1 – 2

Assigning Meaning to Subtrees

Processing Parse Trees

Assign a ***translation*** for each node / subtree

Goal:
Translation is the value of the expression



Grammar:
E ::= *E* **dash** *T*
| *T*
T ::= **intlit**

Input Expression: 4 – 1 – 2

In Summary

Processing Parse Trees

- Translation depends on the goal

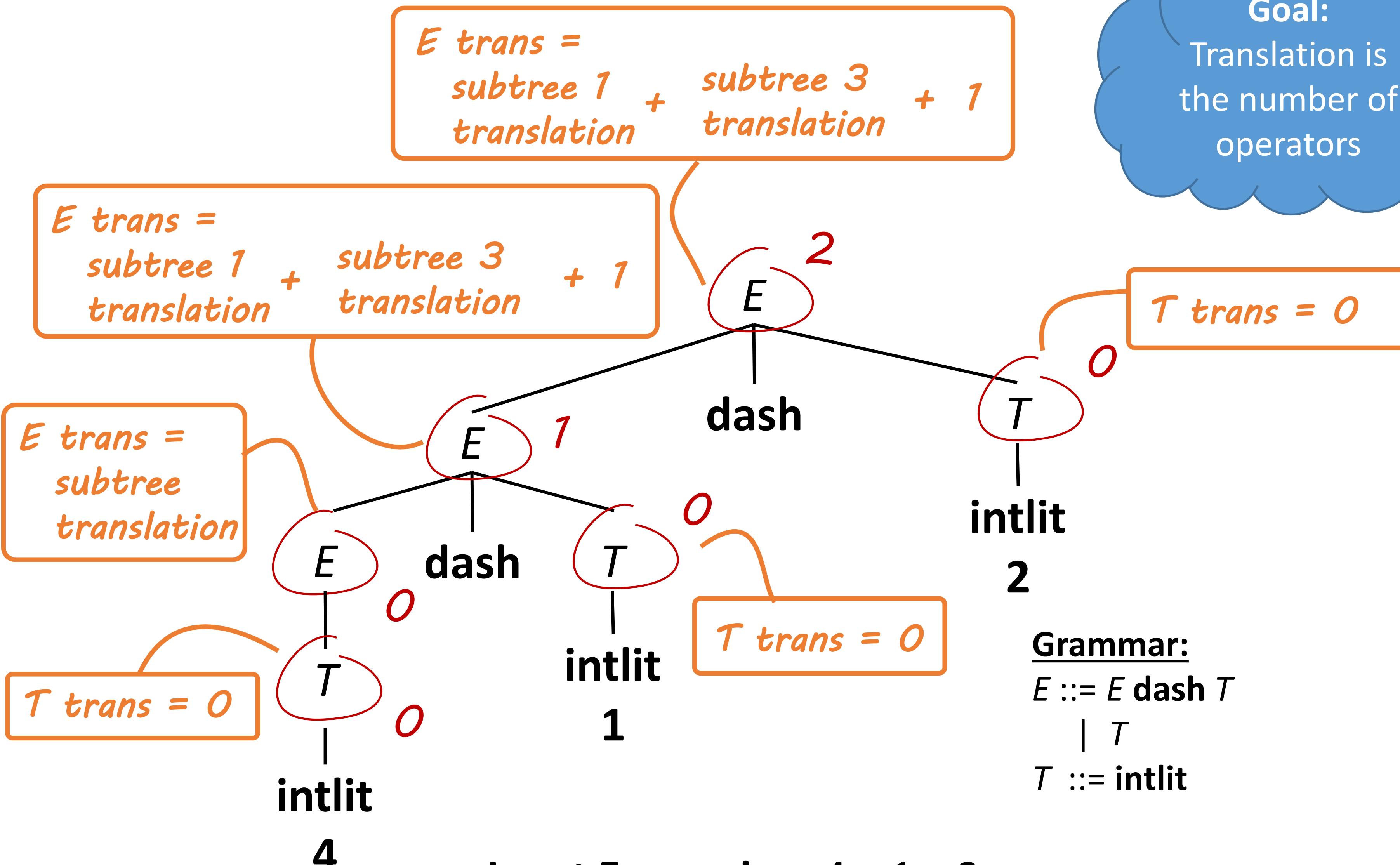
Goal:
Translation is the value of the expression

Goal:
Translation is the number of operands

Assigning Meaning to Subtrees

Processing Parse Trees

Goal: Translation is the number of operators



Grammar:

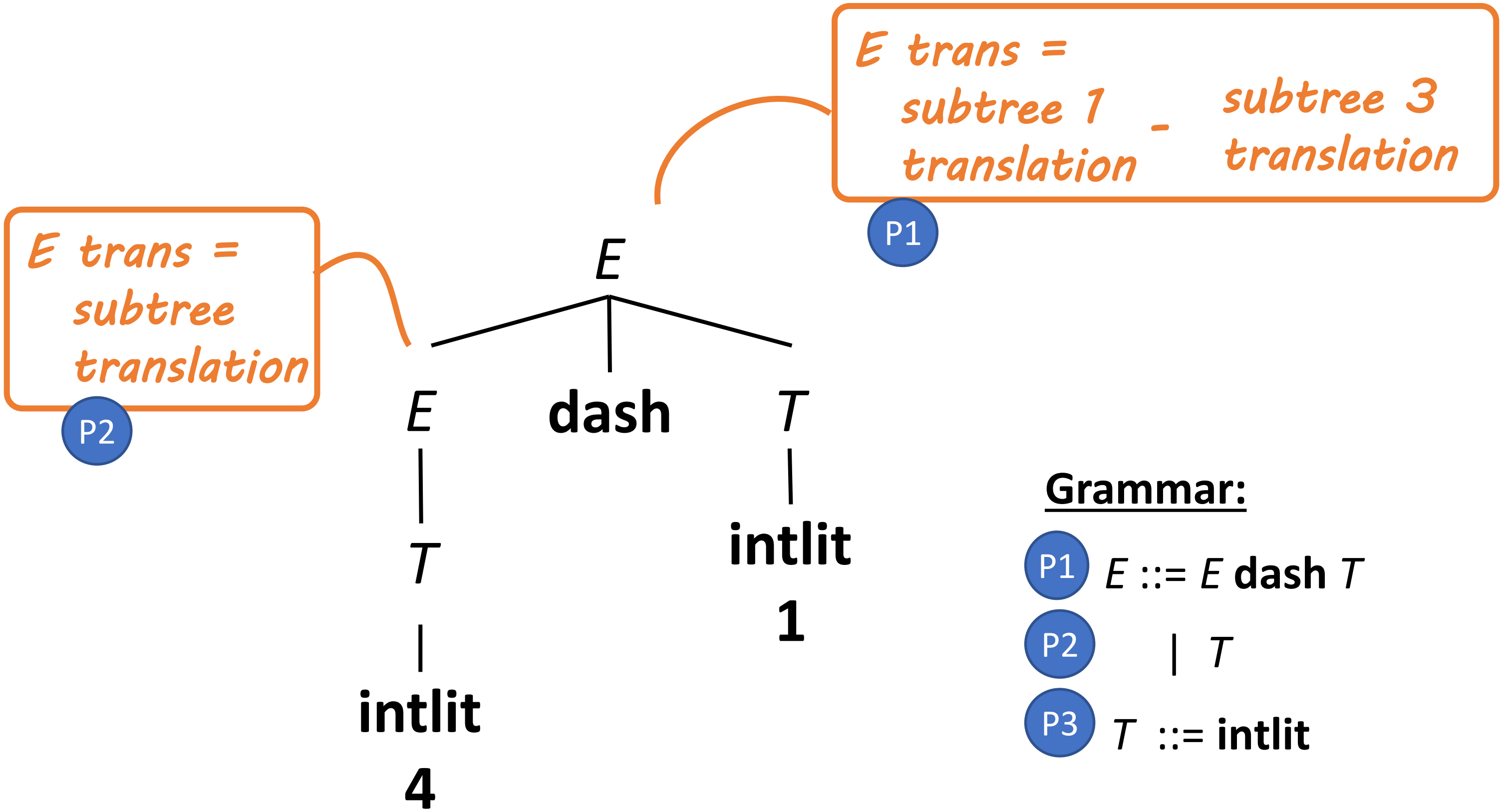
E ::= *E* **dash** *T*
| *T*
T ::= **intlit**

Input Expression: 4 – 1 – 2

Tree Translation Intuition: Summary

Processing Parse Trees

- Translation depends on the goal
- Translation is selected based on the production
- Conceptually, a post-pass over the complete parse tree



Grammar:

P1 *E* ::= *E* **dash** *T*

P2 | *T*

P3 *T* ::= **intlit**

Lecture Outline

Preview Lecture 5 – Syntax-Directed Translation

Recall Syntactic Ambiguity

Assigning Meaning to (Parse) Trees

- Tree translation intuition
- Introduce Syntax-Directed Definition

Tools for SDD

- Bison

Syntax-Directed Definition

Syntax-Directed Definitions

Processing Parse Trees

Semantics of Context-Free Languages

by

DONALD E. KNUTH
California Institute of Technology

ABSTRACT

"Meaning" may be assigned to a string in a context-free language by defining "attributes" of the symbols in a derivation tree for that string. The attributes can be defined by functions associated with each production in the grammar. This paper

Syntax-Directed Definitions

Processing Parse Trees

- Attach translation rules per-production

$X ::= \alpha_1 \ldots \alpha_n$ { *LHS*.trans = <*translation of X that can use translations of* $\alpha_1 \ldots \alpha_n$> }

| $\beta_1 \ldots \beta_n$ { *LHS*.trans = <*translation of X that can use translations of* $\beta_1 \ldots \beta_n$> }

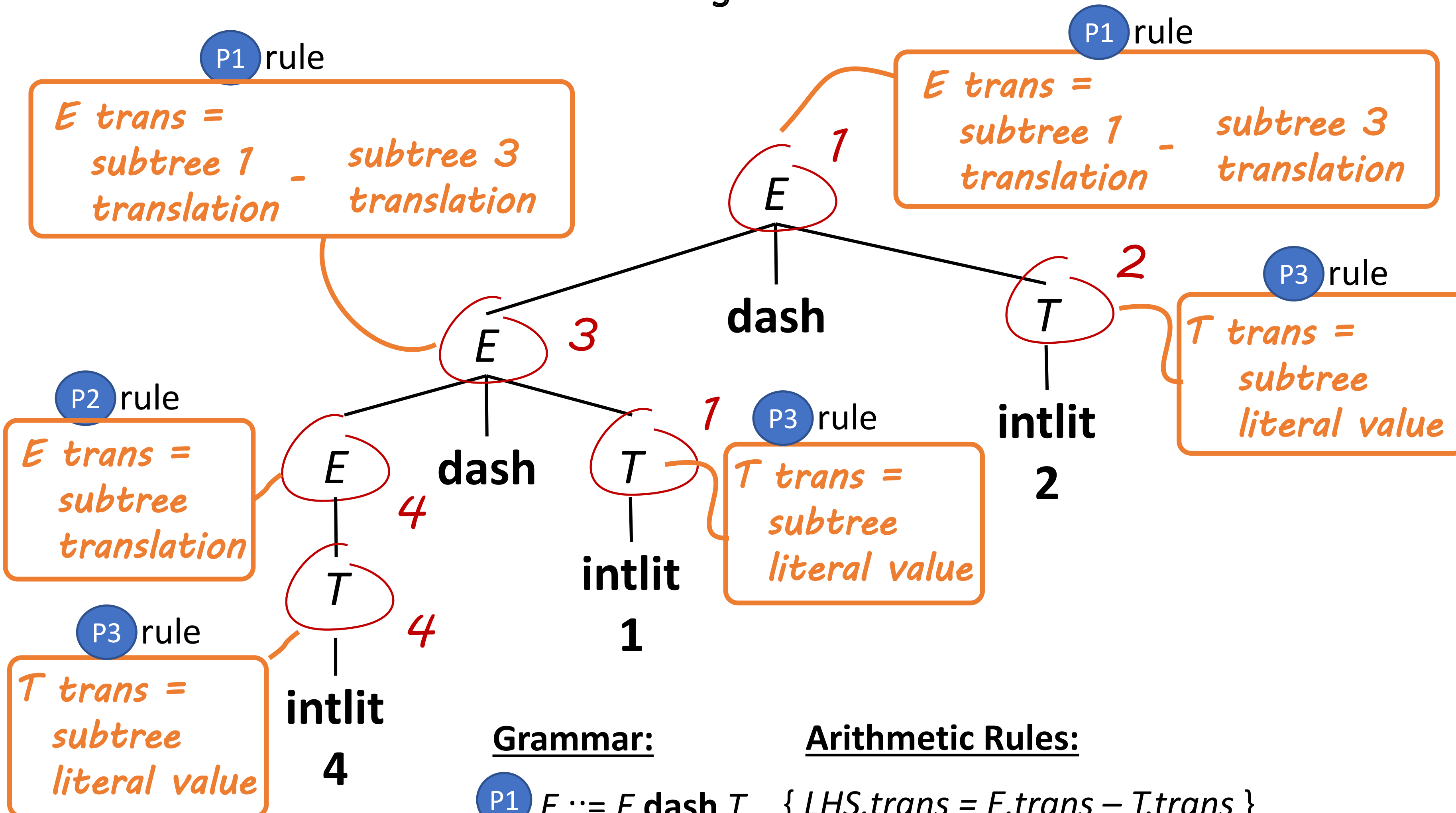
$Y ::= \gamma_1 \ldots \gamma_n$ { *LHS*.trans = <*translation of Y that can use translations of* $\gamma_1 \ldots \gamma_n$> }

Goal:
Translation is the value of the expression

Grammar:		**Arithmetic Rules:**
P1	*E* ::= *E* **dash** *T*	{ *LHS.trans = E.trans – T.trans* }
P2	| *T*	{ *LHS.trans = T.trans* }
P3	*T* ::= **intlit**	{ *LHS.trans =* **intlit**.value }

SDD: Example

Processing Parse Trees



Grammar:	**Arithmetic Rules:**
P1 *E* ::= *E* **dash** *T*	{ *LHS.trans* = *E.trans* – *T.trans* }
P2 | *T*	{ *LHS.trans* = *T.trans* }
P3 *T* ::= **intlit**	{ *LHS.trans* = **intlit**.value }

Lecture Outline

Syntax-Directed Definition

Working with Parse Trees

- Benefits of Parse Trees / Trees in general
- Tree translation intuition
- Syntax-Directed Definition
  - Finer points

Tools for SDD

- Bison

Syntax-Directed Definition

SDD: Parse Tree Processing Multitool

Processing Parse Trees

SDD is a flexible tool for assigning meaning to parse trees

- Useful beyond compilers
- We won't even use the full power of the technique

SDD: A Different Translation Scheme

Processing Parse Trees

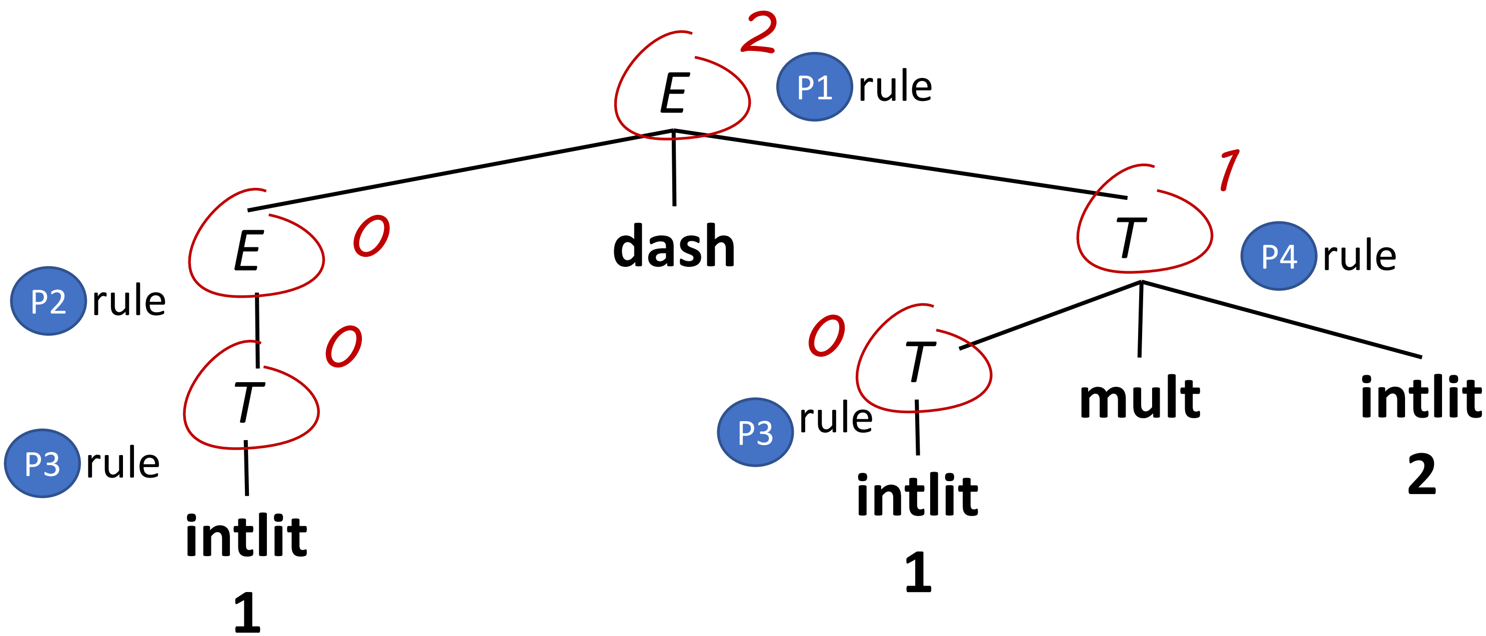
- SDD can do more than evaluate expressions

Goal:
Translate arithmetic expression to its number of operators

	Grammar:	Operator Count Rules:
P1	*E* ::= *E* **dash** *T*	{ *LHS.trans* = *E*.trans + *T*.trans + 1}
P2	| *T*	{ *LHS.trans* = *T*.trans }
P3	*T* ::= **intlit**	{ *LHS.trans* = 0 }
P4	*T* ::= T **mult intlit**	{ *LHS.trans* = *T*.trans + 1 }

SDD: A Different Translation Scheme

Processing Parse Trees



Grammar:	**Operator Count Rules:**
P1 *E* ::= *E* **dash** *T*	{ *LHS.trans* = *E*.trans + *T*.trans + 1}
P2 | *T*	{ *LHS.trans* = *T*.trans }
P3 *T* ::= **intlit**	{ *LHS.trans* = 0 }
P4 *T* ::= T **mult intlit**	{ *LHS.trans* = *T*.trans + 1 }

SDD: Yet *Another* Translation

Processing Parse Trees

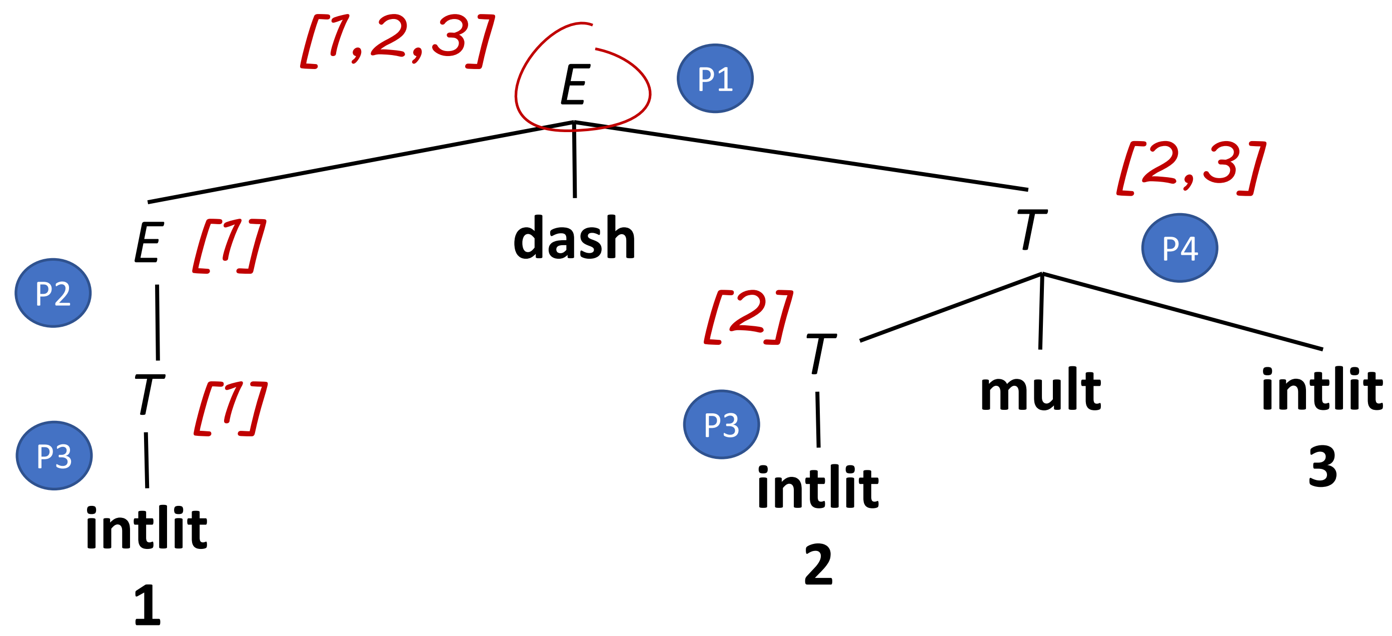
- Myth: SDD translations are always int values

Goal:
Translate arithmetic expression to a list of its int literals

Grammar:	**List of integers Rules:**
P1 *E* ::= *E* **dash** *T*	{ *LHS.trans* = *E*.trans.extend(*T*.trans) }
P2 | *T*	{ *LHS.trans* = *T*.trans }
P3 *T* ::= **intlit**	{ *LHS.trans* = [**intlit.value**] }
P4 *T* ::= T **mult intlit**	{ *LHS.trans* = *T*.trans.extend([**intlit.value**]) }

SDD: Yet *Another* Translation

Processing Parse Trees



Grammar:

P1	*E* ::= *E* **dash** *T*	{ *LHS.trans* = *E*.trans.extend(*T*.trans) }
P2	| *T*	{ *LHS.trans* = *T*.trans }
P3	*T* ::= **intlit**	{ *LHS.trans* = [**intlit.value**] }
P4	*T* ::= T **mult intlit**	{ *LHS.trans* = *T*.trans.extend([**intlit.value**]) }

SDD: Binary Numbers
Processing Parse Trees

CFG		Rules
P1: B ::=	**0**	*LHS*.trans = ???
P2:	| **1**	*LHS*.trans = ???
P3:	| *B* **0**	*LHS*.trans = ???
P4:	| *B* **1**	*LHS*.trans = ???

Goal:
Translation is the value of the input

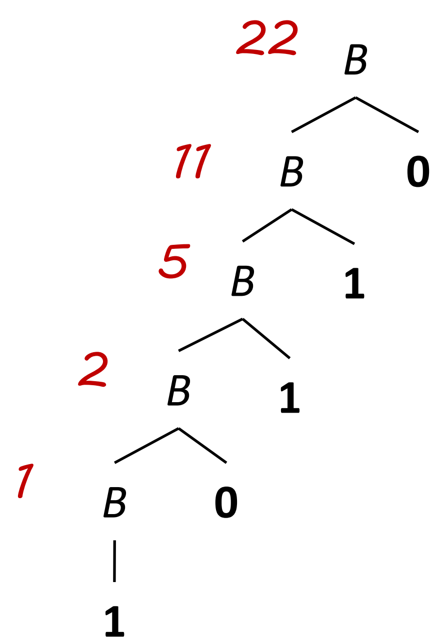
Example:
Input string 10110 should be 22

SDD: Binary Numbers

Processing Parse Trees

CFG	Rules
P1: B ::= **0**	*LHS*.trans = 0
P2: | **1**	*LHS*.trans = 1
P3: | *B* **0**	*LHS*.trans = B_1.trans * 2
P4: | *B* **1**	*LHS*.trans = B_1.trans * 2 + 1

Goal:
Translation is the value of the input



Example:
Input string 10110 should be 22

SDD: Int Declaration List

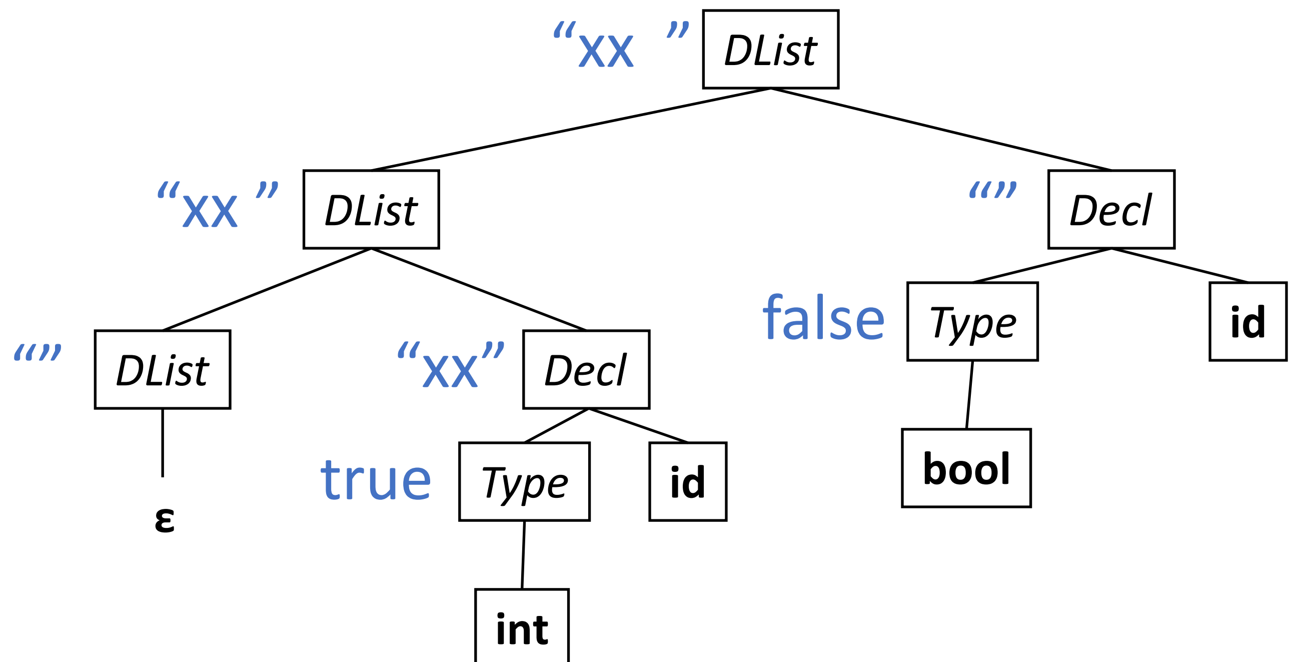
Processing Parse Trees

CFG			Rules
DList	::=	**ε**	*LHS*.trans = “”
	|	*DList Decl*	*LHS.trans* = *Decl.trans* + “ “ + DList_2.trans
Decl	::=	*Type* **id ;**	if (*Type*.trans) {*LHS*.trans = **id**.value} else {*Decl*.trans = “”}
Type	::=	**int**	*LHS*.trans = true
	|	**bool**	*LHS*.trans = false

Input string

int xx;
bool yy;

Translation is a String of **int** ids only



BISON: A tool for SDD

Assigning Meaning to Parse Trees

```
%union {
        int intval;
}

%token zero
%token one


%type <intval> B

%%

S : B      { printf("Value is %d\n", $1); }
B :  zero  { $$ = 0; }
  | one    { $$ = 1; }
  | B zero { $$ = $1 * 2 + 0; }
  | B one  { $$ = $1 * 2 + 1; }
;

%%
```

Next Time
Lecture Preview

Discuss a use of SDD of particular use to Compilers:

- Translating the parse tree into an *Abstract Syntax Tree*