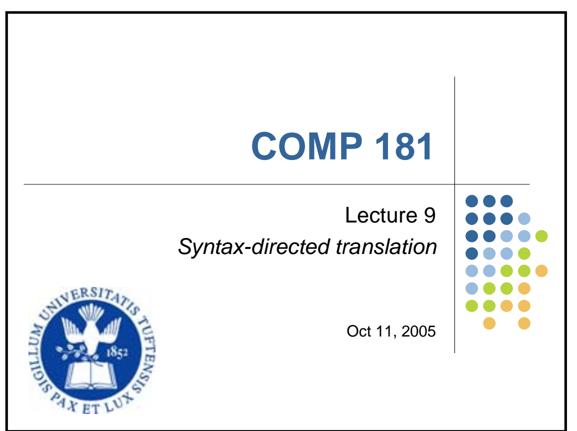
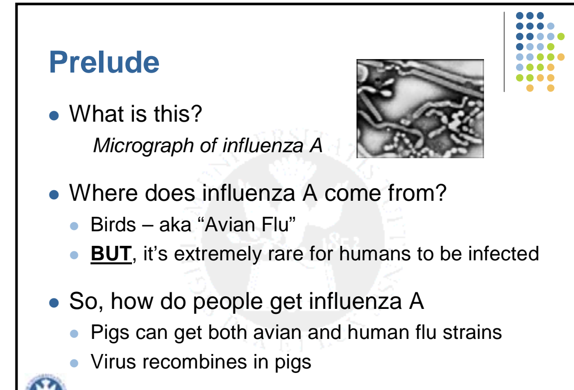
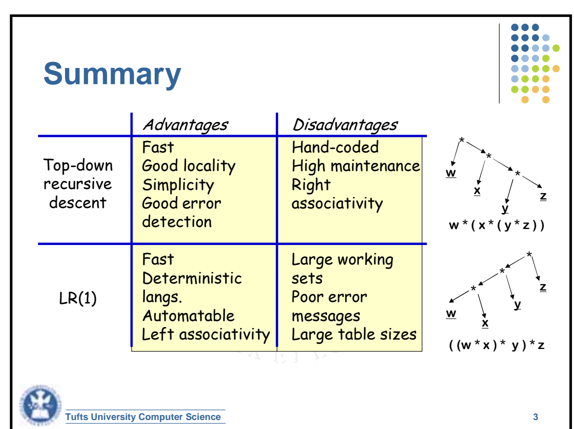
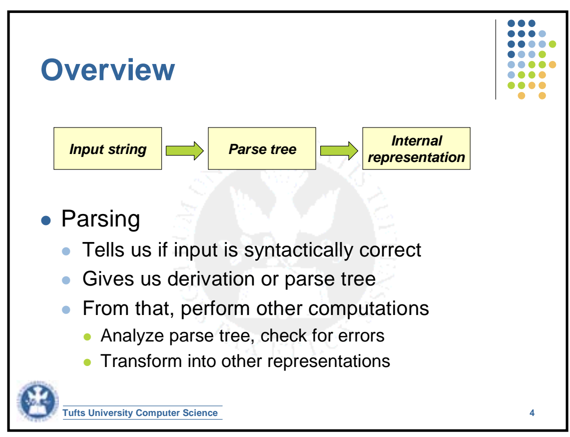
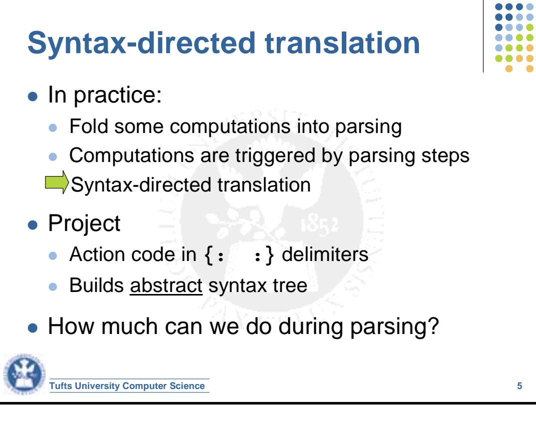


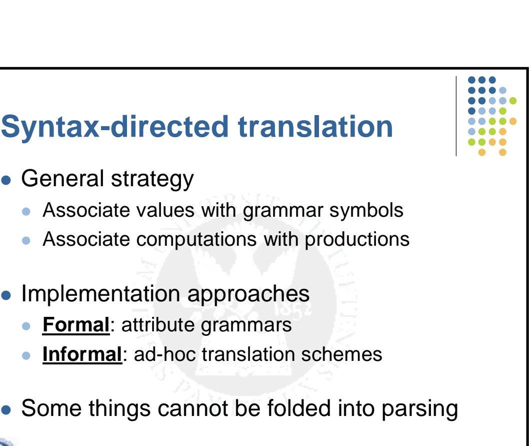


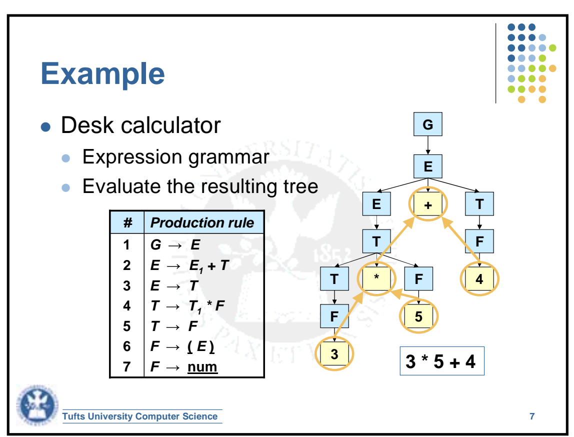


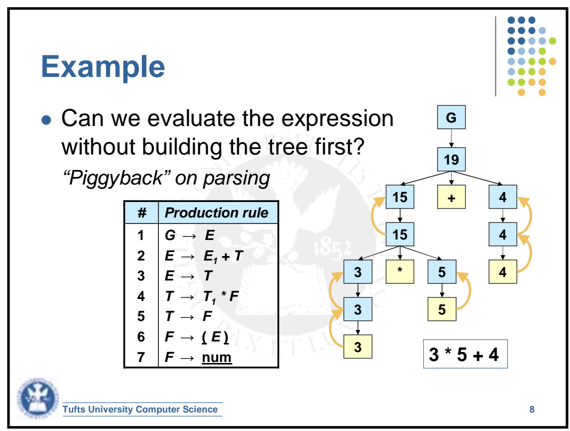
	Advantages	*Disadvantages*
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

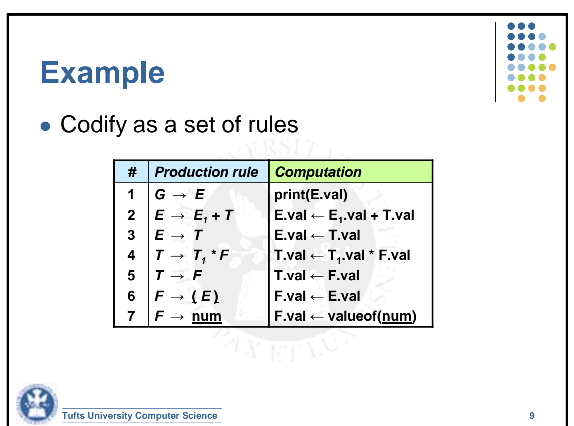


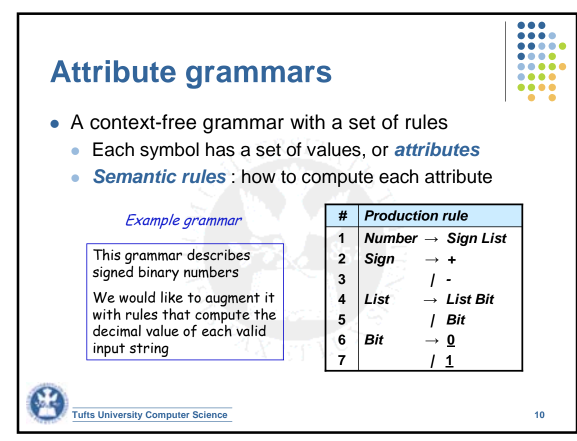


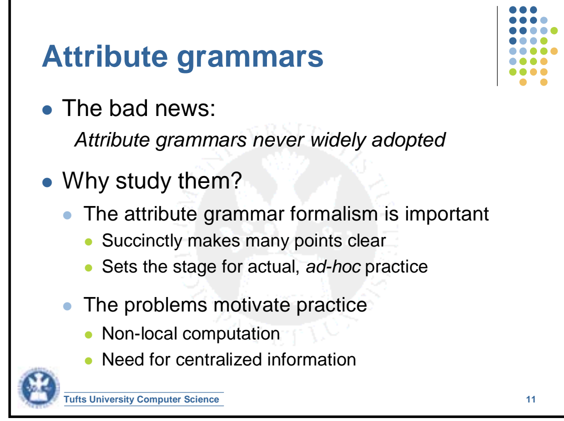


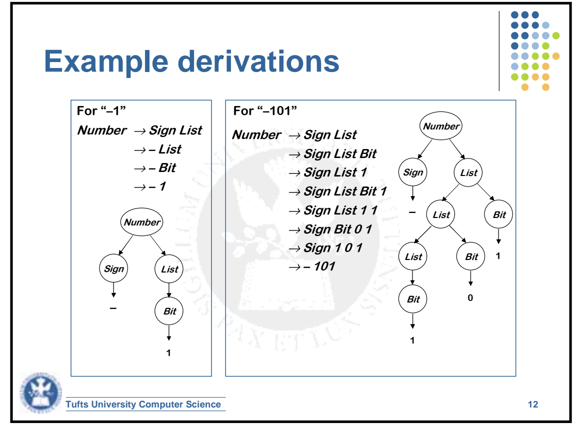

Example
Desk calculator
Expression grammar
Evaluate the resulting tree
| Production rule
1 G → E
2 E → E1 + T
3 E → T
4 T → T1 * F
5 T → F
6 F → (E)
7 F → num
G
E
E + T
T F
T * F 4
F 5
3
3 * 5 + 4
Tufts University Computer Science
7

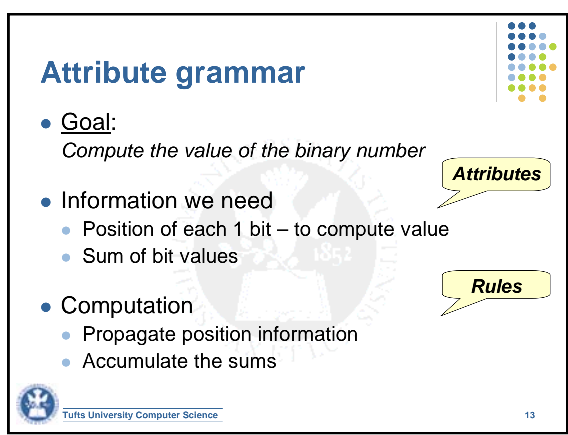

Example
Can we evaluate the expression without building the tree first?
"Piggyback" on parsing
| Production rule
1 G → E
2 E → E1 + T
3 E → T
4 T → T1 * F
5 T → F
6 F → (E)
7 F → num
G
19
15 + 4
15 4
3 * 5 4
3 5
3
3 * 5 + 4
Tufts University Computer Science
8

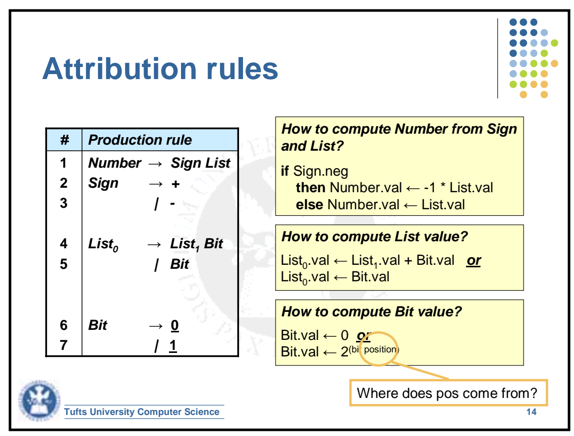

Example
Codify as a set of rules
| Production rule | Computation
1 G → E print(E.val)
2 E → E1 + T E.val ← E1.val + T.val
3 E → T E.val ← T.val
4 T → T1 * F T.val ← T1.val * F.val
5 T → F T.val ← F.val
6 F → (E) F.val ← E.val
7 F → num F.val ← valueof(num)
Tufts University Computer Science
9

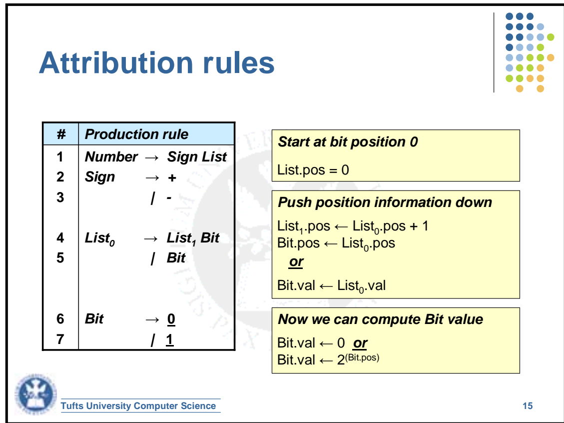

Attribute grammars
A context-free grammar with a set of rules
Each symbol has a set of values, or attributes
Semantic rules : how to compute each attribute
Example grammar
This grammar describes signed binary numbers
We would like to augment it with rules that compute the decimal value of each valid input string
| Production rule
1 Number → Sign List
2 Sign → +
3 | -
4 List → List Bit
5 | Bit
6 Bit → 0
7 | 1
Tufts University Computer Science
10

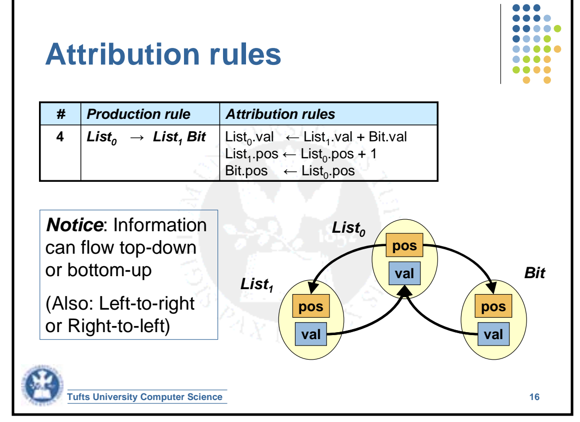

Attribute grammars
The bad news:
Attribute grammars never widely adopted
Why study them?
The attribute grammar formalism is important
Succinctly makes many points clear
Sets the stage for actual, ad-hoc practice
The problems motivate practice
Non-local computation
Need for centralized information
Tufts University Computer Science
11

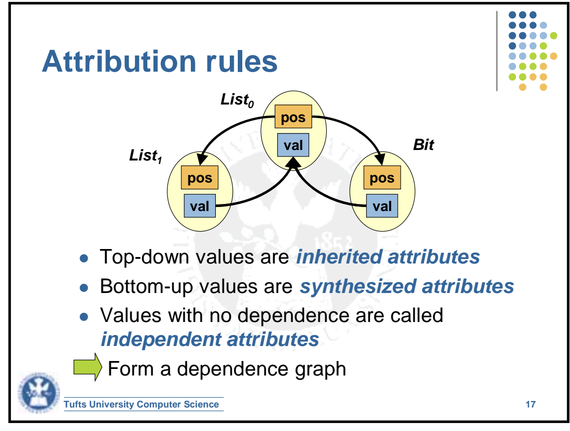

Example derivations
For "–1"
Number → Sign List
→ – List
→ – Bit
→ – 1
For "–101"
Number → Sign List
→ Sign List Bit
→ Sign List 1
→ Sign List Bit 1
→ Sign List 1 1
→ Sign Bit 0 1
→ Sign 1 0 1
→ – 101
Tufts University Computer Science
12

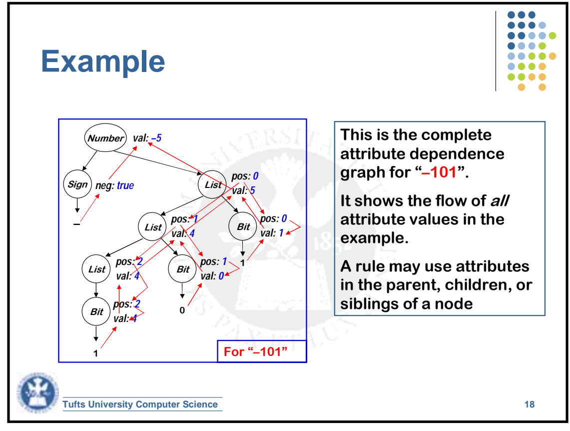

Attribute grammar
Goal:
Compute the value of the binary number
Attributes
Information we need
Position of each 1 bit – to compute value
Sum of bit values
Rules
Computation
Propagate position information
Accumulate the sums
Tufts University Computer Science
13

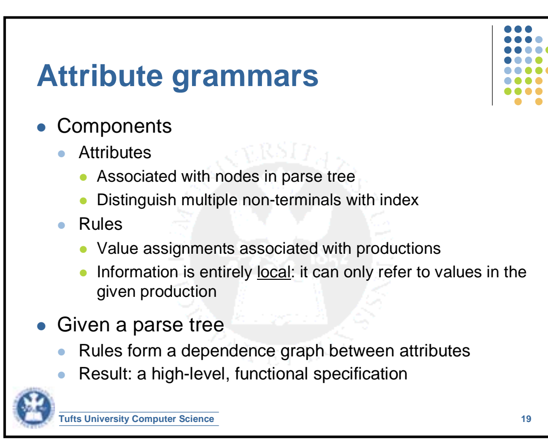

Attribution rules
Production rule
1 Number → Sign List
2 Sign → +
3 | -
4 List0 → List1 Bit
5 | Bit
6 Bit → 0
7 | 1
How to compute Number from Sign and List?
if Sign.neg
then Number.val ← -1 * List.val
else Number.val ← List.val
How to compute List value?
List0.val ← List1.val + Bit.val or
List0.val ← Bit.val
How to compute Bit value?
Bit.val ← 0 or
Bit.val ← 2(bit position)
Where does pos come from?
Tufts University Computer Science
14

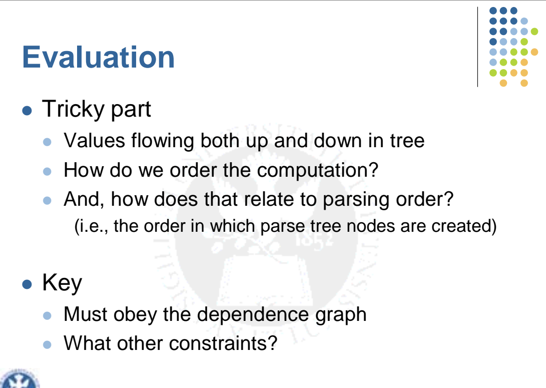

Attribution rules
Production rule
1 Number → Sign List
2 Sign → +
3 | -
4 List0 → List1 Bit
5 | Bit
6 Bit → 0
7 | 1
Start at bit position 0
List.pos = 0
Push position information down
List1.pos ← List0.pos + 1
Bit.pos ← List0.pos
or
Bit.val ← List0.val
Now we can compute Bit value
Bit.val ← 0 or
Bit.val ← 2(Bit.pos)
Tufts University Computer Science
15

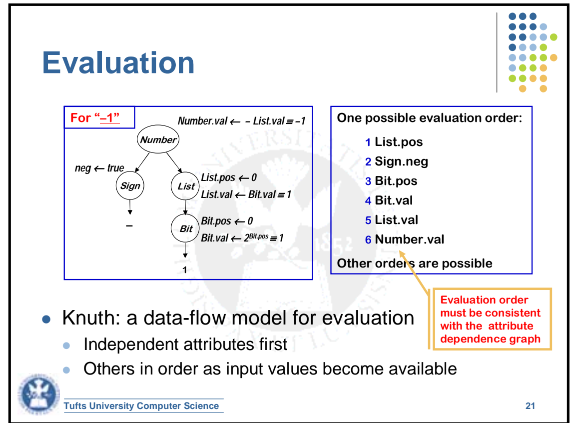

Attribution rules
Production rule Attribution rules
4 List0 → List1 Bit
List0.val ← List1.val + Bit.val
List1.pos ← List0.pos + 1
Bit.pos ← List0.pos
Notice: Information can flow top-down or bottom-up
(Also: Left-to-right or Right-to-left)
List0
pos
val
List1
pos
val
Bit
pos
val
Tufts University Computer Science
16

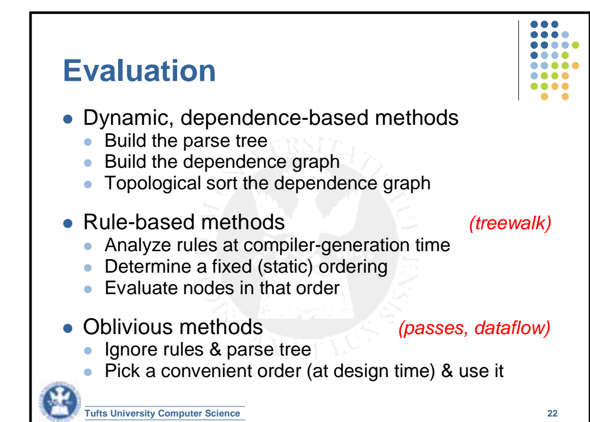

Attribution rules
List0
pos
val
List1
pos
val
Bit
pos
val
Top-down values are inherited attributes
Bottom-up values are synthesized attributes
Values with no dependence are called independent attributes
Form a dependence graph
Tufts University Computer Science
17

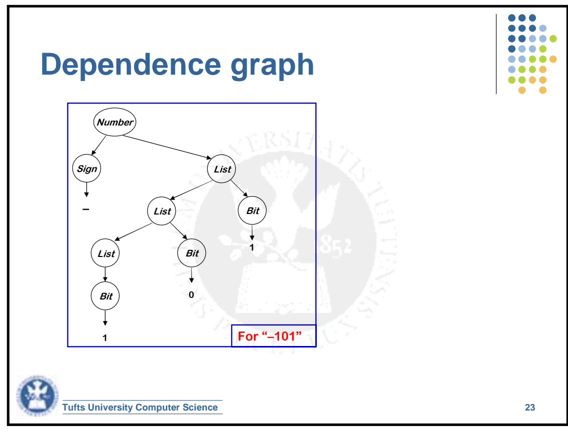

Example
Number val: –5
Sign neg: true
List pos: 0 val: 5
List pos: 1 val: 4
Bit pos: 0 val: 1
List pos: 2 val: 4
Bit pos: 1 val: 0
Bit pos: 2 val: 4
For "–101"
This is the complete attribute dependence graph for "–101".
It shows the flow of all attribute values in the example.
A rule may use attributes in the parent, children, or siblings of a node
Tufts University Computer Science
18

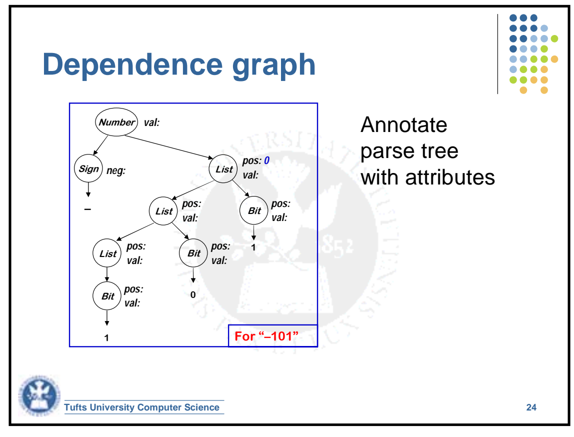

Attribute grammars
Components
Attributes
Associated with nodes in parse tree
Distinguish multiple non-terminals with index
Rules
Value assignments associated with productions
Information is entirely local: it can only refer to values in the given production
Given a parse tree
Rules form a dependence graph between attributes
Result: a high-level, functional specification
Tufts University Computer Science
19

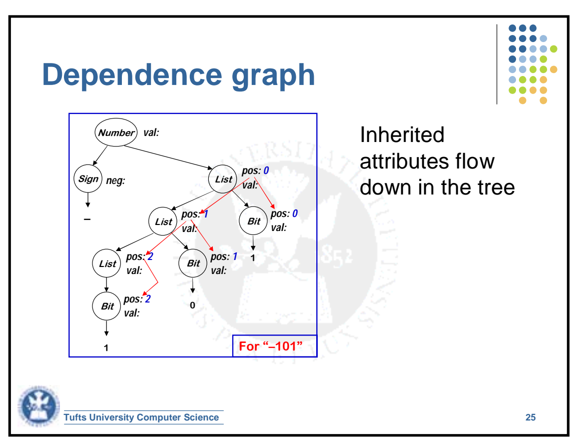

Evaluation
Tricky part
Values flowing both up and down in tree
How do we order the computation?
And, how does that relate to parsing order?
(i.e., the order in which parse tree nodes are created)
Key
Must obey the dependence graph
What other constraints?
Tufts University Computer Science
20

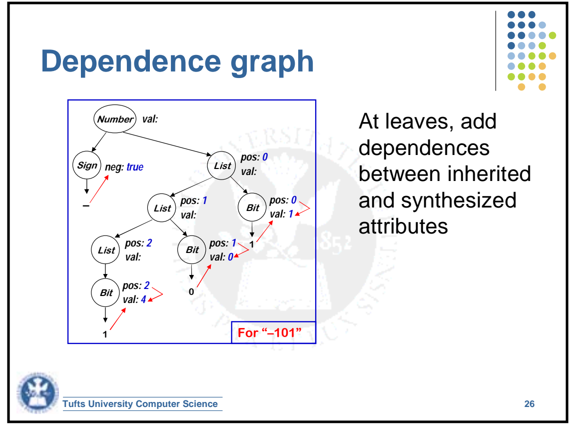

Evaluation
For "–1"
Number
Sign
List
Bit
One possible evaluation order:
1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val
Other orders are possible
Evaluation order must be consistent with the attribute dependence graph
Knuth: a data-flow model for evaluation
Independent attributes first
Others in order as input values become available
Tufts University Computer Science
21

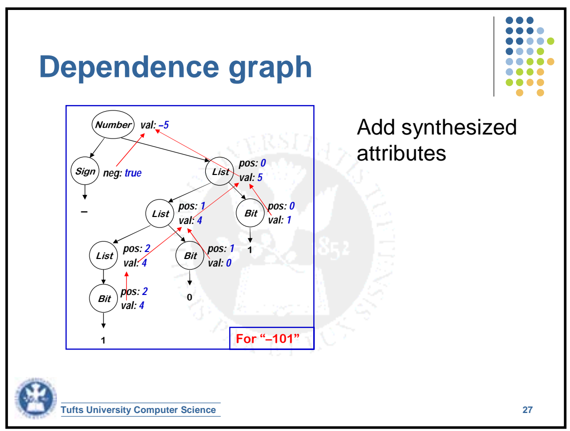

Evaluation
Dynamic, dependence-based methods
Build the parse tree
Build the dependence graph
Topological sort the dependence graph
Rule-based methods
(treewalk)
Analyze rules at compiler-generation time
Determine a fixed (static) ordering
Evaluate nodes in that order
Oblivious methods
(passes, dataflow)
Ignore rules & parse tree
Pick a convenient order (at design time) & use it
Tufts University Computer Science
22

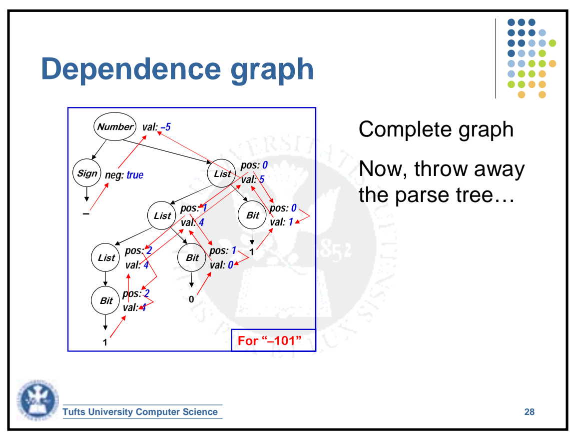

Dependence graph
Number
Sign
List
List
Bit
List
Bit
Bit
For "–101"
Tufts University Computer Science
23

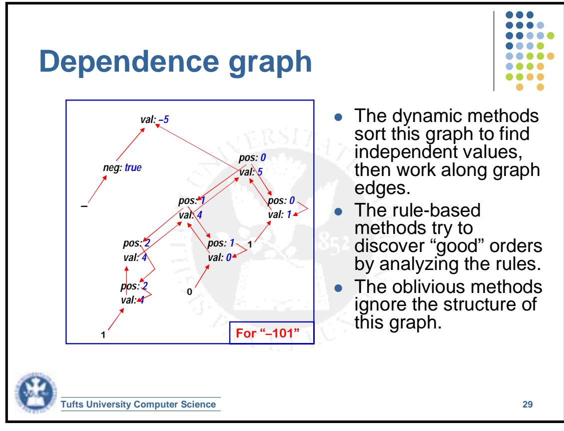

Dependence graph
Number val:
Sign neg:
List pos: 0 val:
List pos: val:
Bit pos: val:
For "–101"
Annotate parse tree with attributes
Tufts University Computer Science
24

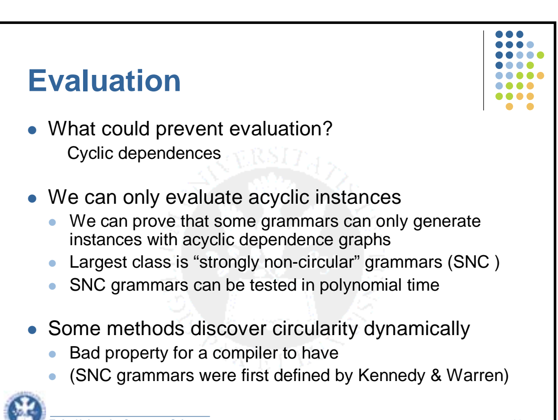

Dependence graph
Inherited attributes flow down in the tree
For "–101"
Tufts University Computer Science
25


Dependence graph
At leaves, add dependences between inherited and synthesized attributes
For "–101"
Tufts University Computer Science
26


Dependence graph
Add synthesized attributes
For "–101"
Tufts University Computer Science
27


Dependence graph
Complete graph
Now, throw away the parse tree…
For "–101"
Tufts University Computer Science
28


Dependence graph
The dynamic methods sort this graph to find independent values, then work along graph edges.
The rule-based methods try to discover "good" orders by analyzing the rules.
The oblivious methods ignore the structure of this graph.
For "–101"
Tufts University Computer Science
29


Evaluation
What could prevent evaluation?
Cyclic dependences
We can only evaluate acyclic instances
We can prove that some grammars can only generate instances with acyclic dependence graphs
Largest class is "strongly non-circular" grammars (SNC)
SNC grammars can be tested in polynomial time
Some methods discover circularity dynamically
Bad property for a compiler to have
(SNC grammars were first defined by Kennedy & Warren)
Tufts University Computer Science
30

Syntax-directed translation

- Attribute grammars
  - Clean, declarative
  - Handle a wide variety of problems
  - BUT, have limitations
  - ⇨ Never widely adopted
- Reality
  - LR parsers
  - Associate attributes with parser symbols
  - Apply arbitrary code actions on attributes
  - ⇨ This is what you're doing in the project

Realist's alternative

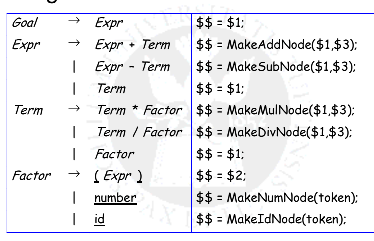
- *Ad-hoc* syntax-directed translation
  - Associate pieces of code with each production
  - At each reduction, the code is executed
  - Arbitrary code provides complete flexibility
    Includes ability to do tasteless & bad things
- To make this work:
  - Need names for attributes on *lhs* & *rhs*
  - Yacc introduced **$$, $1, $2, … $n**, left to right

Syntax-directed translation and parsing

- Classes of attributes that fit into parsing
- ***S-attributed*** definition
  - All attributes are synthesized
  - Use values from children & from constants
  - Can evaluate bottom-up in one pass
  - Fits cleanly into LR parsing

Example

- Building the AST



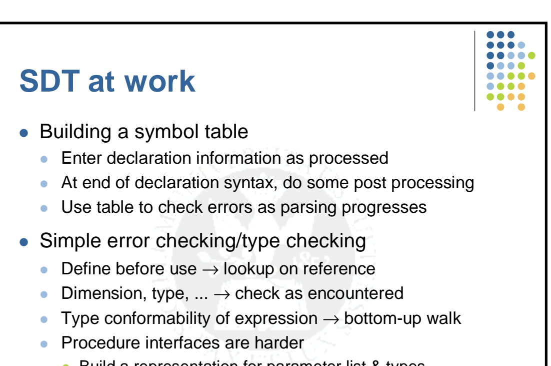
Goal	→	*Expr*	$$ = $1;
Expr	→	*Expr + Term*	$$ = MakeAddNode($1,$3);
	|	*Expr - Term*	$$ = MakeSubNode($1,$3);
	|	*Term*	$$ = $1;
Term	→	*Term * Factor*	$$ = MakeMulNode($1,$3);
	|	*Term / Factor*	$$ = MakeDivNode($1,$3);
	|	*Factor*	$$ = $1;
Factor	→	*(Expr)*	$$ = $2;
	|	number	$$ = MakeNumNode(token);
	|	id	$$ = MakeIdNode(token);

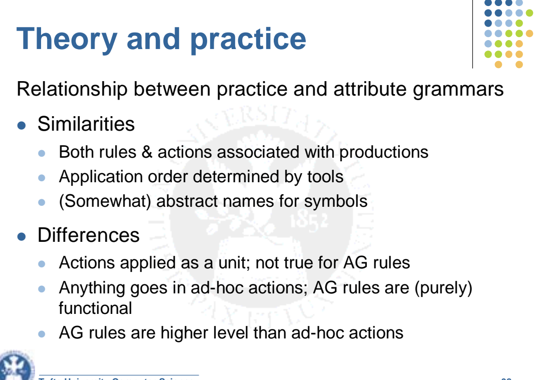
Syntax-directed translation and parsing

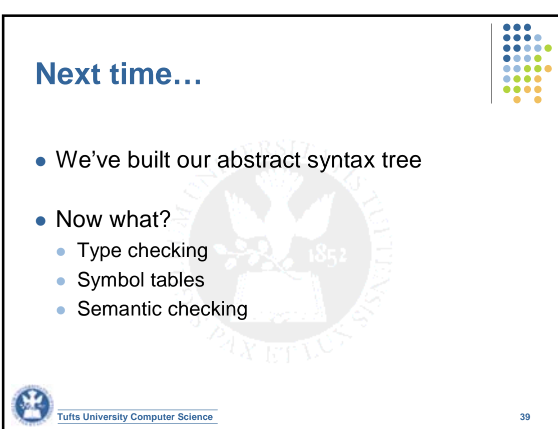
- ***L-attributed*** definition
  - Use values from parent, constants, & siblings
  - For production $A \rightarrow X_1\ X_2 \dots X_n$
  - Each attribute of X_i depends on
    - Attributes of $X_1\ X_2 \dots X_{i-1}$, and
    - Inherited attributes of A
  - Evaluate in a single top-down pass (left to right)
  - Good match for predictive parsing, but can also be adapted for LR parsing

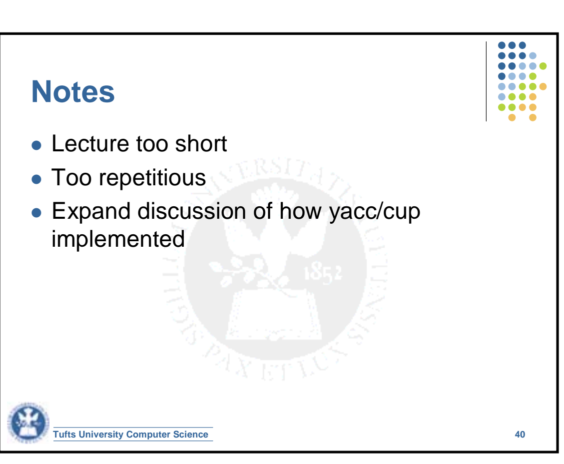
SDT and LR parsers

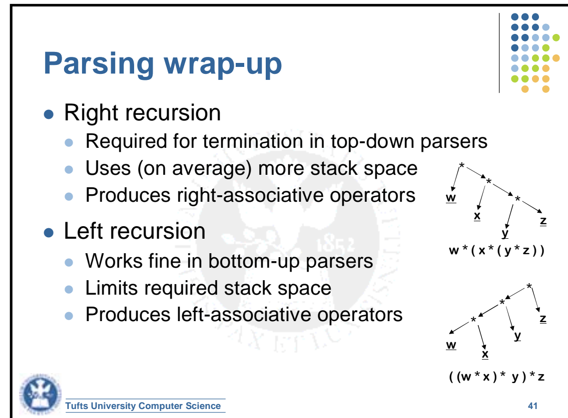
- Two issues
  - Where to store the attribute values
  - Integrating the action code
- Key: store attributes on stack
  At a reduction of $A \rightarrow \beta$
  - Pop 3 × |β| symbols – 1 symbol, 1 state, 1 value
  - Map values to **$1** … **$n**
  - Invoke action code on values, collect result in **$$**
  - Push **$$** back on stack with new symbol, state

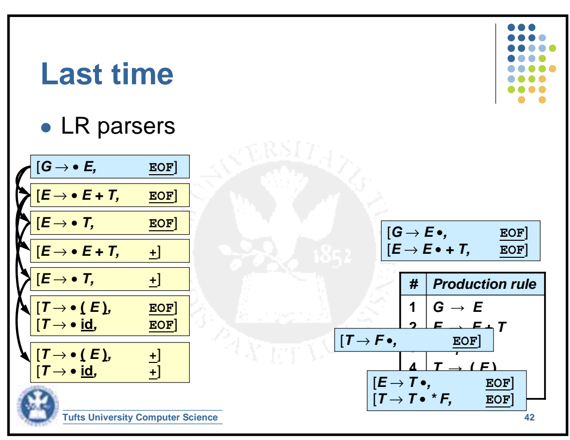

SDT at work
Building a symbol table
Enter declaration information as processed
At end of declaration syntax, do some post processing
Use table to check errors as parsing progresses
Simple error checking/type checking
Define before use → lookup on reference
Dimension, type, ... → check as encountered
Type conformability of expression → bottom-up walk
Procedure interfaces are harder
Build a representation for parameter list & types
Check actual vs. formal parameter list
Tufts University Computer Science
37


Theory and practice
Relationship between practice and attribute grammars
Similarities
Both rules & actions associated with productions
Application order determined by tools
(Somewhat) abstract names for symbols
Differences
Actions applied as a unit; not true for AG rules
Anything goes in ad-hoc actions; AG rules are (purely) functional
AG rules are higher level than ad-hoc actions
Tufts University Computer Science
38

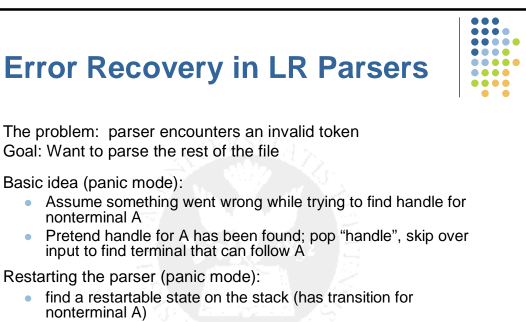

Next time…
We've built our abstract syntax tree
Now what?
Type checking
Symbol tables
Semantic checking
Tufts University Computer Science
39


Notes
Lecture too short
Too repetitious
Expand discussion of how yacc/cup implemented
Tufts University Computer Science
40


Parsing wrap-up
Right recursion
Required for termination in top-down parsers
Uses (on average) more stack space
Produces right-associative operators
w * (x * (y * z))
Left recursion
Works fine in bottom-up parsers
Limits required stack space
Produces left-associative operators
((w * x) * y) * z
Tufts University Computer Science
41


Last time
LR parsers
[G → • E, EOF]
[E → • E + T, EOF]
[E → • T, EOF]
[E → • E + T, +]
[E → • T, +]
[T → • (E), EOF]
[T → • id, EOF]
[T → • (E), +]
[T → • id, +]
[G → E •, EOF]
[E → E • + T, EOF]
Production rule
1 G → E
[T → F •, EOF]
[E → T •, EOF]
[T → T • * F, EOF]
Tufts University Computer Science
42

Error Recovery in LR Parsers



The problem: parser encounters an invalid token
Goal: Want to parse the rest of the file

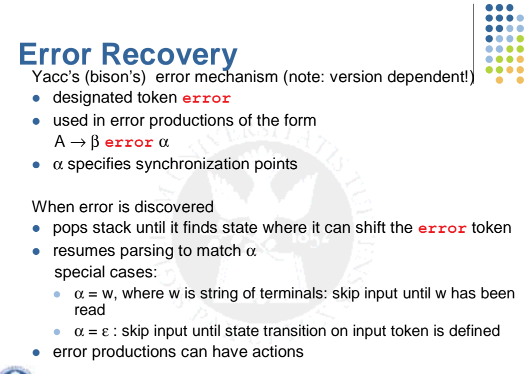
Basic idea (panic mode):

- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser (panic mode):

- find a restartable state on the stack (has transition for nonterminal A)
- move to a consistent place in the input (token that can follow A)
- perform (error) reduction (for nonterminal A)
- print an informative message

Error Recovery



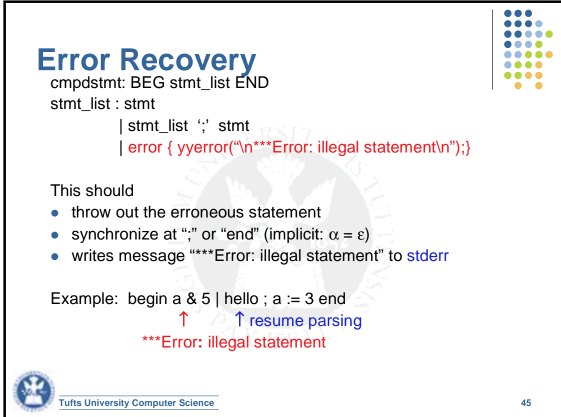
Yacc's (bison's) error mechanism (note: version dependent!)

- designated token `error`
- used in error productions of the form
  $A \rightarrow \beta$ `error` α
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the `error` token
- resumes parsing to match α
  special cases:
  - α = w, where w is string of terminals: skip input until w has been read
  - $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Error Recovery



```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list ';' stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "end" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end
↑ ↑ resume parsing
***Error: illegal statement