

# CMPT 379

## Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

11/26/10

1

## Syntax directed Translation

- Models for translation from parse trees into assembly/machine code
- Representation of translations
  - Attribute Grammars (semantic actions for CFGs)
  - Tree Matching Code Generators
  - Tree Parsing Code Generators

11/26/10

2

## Attribute Grammars

- Syntax-directed translation uses a grammar to produce code (or any other “semantics”)
- Consider this technique to be a generalization of a CFG definition
- Each grammar symbol is associated with an attribute
- An attribute can be anything: a string, a number, a tree, any kind of record or object

11/26/10

3

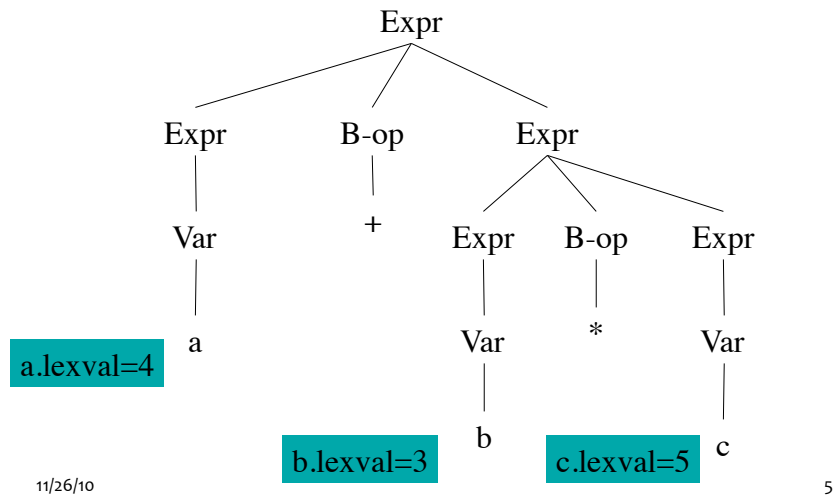
## Attribute Grammars

- A CFG can be viewed as a (finite) representation of a function that relates strings to parse trees
- Similarly, an attribute grammar is a way of relating strings with “meanings”
- Since this relation is syntax-directed, we associate each CFG rule with a semantics (rules to build an abstract syntax tree)
- In other words, attribute grammars are a method to *decorate* or *annotate* the parse tree

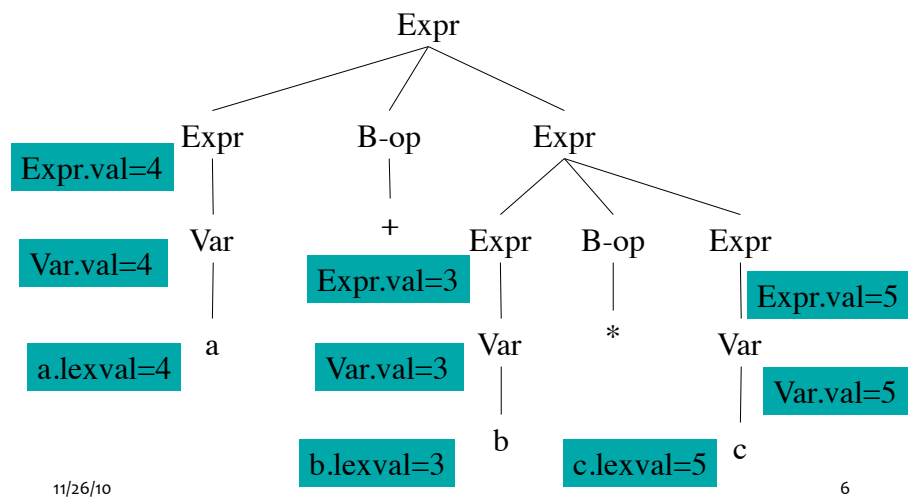
11/26/10

4

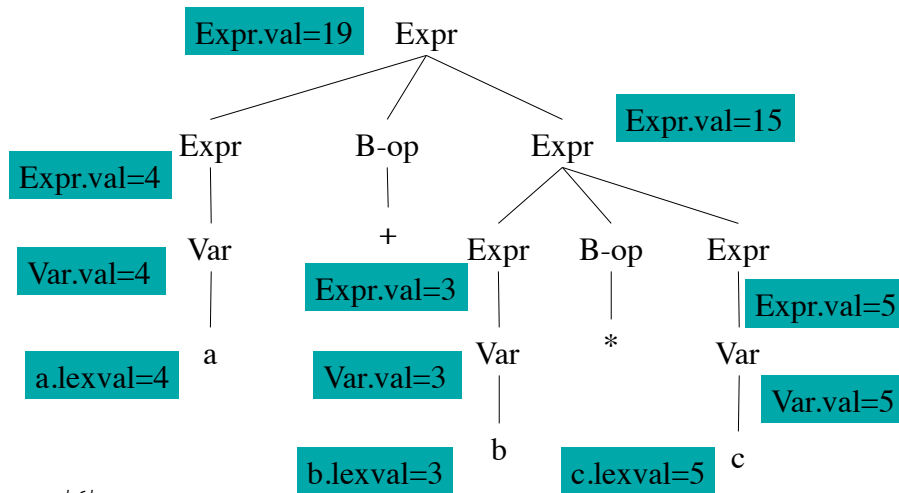
## Example



## Example



## Example



7

## Syntax directed definition

$\text{Var} \rightarrow \text{IntConstant}$

$\{ \$0.\text{val} = \$1.\text{lexval}; \}$   $\dashrightarrow$  In yacc:  $\{ \$\$ = \$1 \}$

$\text{Expr} \rightarrow \text{Var}$

$\{ \$0.\text{val} = \$1.\text{val}; \}$

$\text{Expr} \rightarrow \text{Expr B-op Expr}$

$\{ \$0.\text{val} = \$2.\text{val} (\$1.\text{val}, \$3.\text{val}); \}$

$\text{B-op} \rightarrow +$

$\{ \$0.\text{val} = \text{PLUS}; \}$

$\text{B-op} \rightarrow *$

$\{ \$0.\text{val} = \text{TIMES}; \}$

11/26/10

8

## Flow of Attributes in *Expr*

- Consider the flow of the attributes in the *Expr* syntax-directed defn
- The lhs attribute is computed using the rhs attributes
- Purely bottom-up: compute attribute values of all children (rhs) in the parse tree
- And then use them to compute the attribute value of the parent (lhs)

11/26/10

9

## Synthesized Attributes

- **Synthesized attributes** are attributes that are computed purely bottom-up
- A grammar with semantic actions (or syntax-directed definition) can choose to use *only* synthesized attributes
- Such a grammar plus semantic actions is called an **S-attributed definition**

11/26/10

10

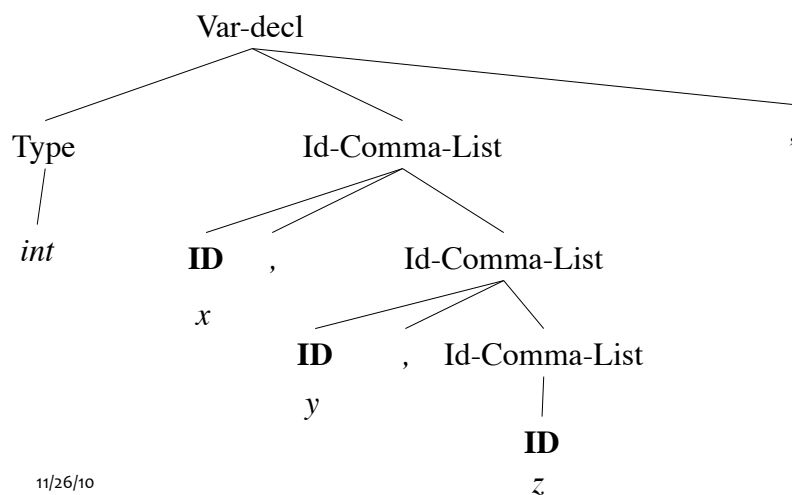
## Inherited Attributes

- Synthesized attributes may not be sufficient for all cases that might arise for semantic checking and code generation
- Consider the (sub)grammar:  
 Var-decl  $\rightarrow$  Type Id-comma-list ;  
 Type  $\rightarrow$  **int** | **bool**  
 Id-comma-list  $\rightarrow$  **ID**  
 Id-comma-list  $\rightarrow$  **ID** , Id-comma-list

11/26/10

11

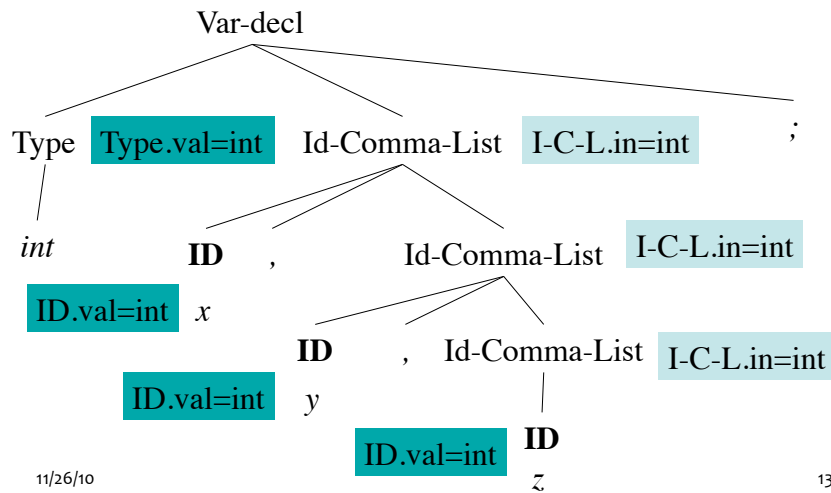
Example: *int x, y, z ;*



11/26/10

12

Example: *int x, y, z ;*



## Syntax-directed definition

$\text{Var-decl} \rightarrow \text{Type Id-comma-list ;}$

$\{ \$2.in = \$1.val; \}$

$\text{Type} \rightarrow \text{int} \mid \text{bool}$

$\{ \$0.val = \text{int}; \} \& \{ \$0.val = \text{bool}; \}$

$\text{Id-comma-list} \rightarrow \text{ID}$

$\{ \$1.val = \$0.in; \}$

$\text{Id-comma-list} \rightarrow \text{ID , Id-comma-list}$

$\{ \$1.val = \$0.in; \$3.in = \$0.in; \}$

## Syntax-directed definition

$\text{Var-decl} \rightarrow \text{Type Id-comma-list};$

In yacc:  $\text{Var-decl} \rightarrow \text{Type} \{ \$\langle \text{val} \rangle \$ = \$1 \} \text{Id-comma-list}$

$\text{Type} \rightarrow \text{int} \mid \text{bool}$

$\{ \$0.\text{val} = \text{int}; \} \& \{ \$0.\text{val} = \text{bool}; \}$

$\text{Id-comma-list} \rightarrow \text{ID}$

$\{ \$1.\text{val} = \$0.\text{in}; \} \quad \text{---}\rightarrow \quad \text{In yacc: } \{ \$1 = \$\langle \text{val} \rangle 0 \}$

$\text{Id-comma-list} \rightarrow \text{ID}, \text{Id-comma-list}$

$\{ \$1.\text{val} = \$0.\text{in}; \$3.\text{in} = \$0.\text{in}; \}$

11/26/10

15

## Flow of Attributes in *Var-decl*

- How do the attributes flow in the *Var-decl* grammar
- **ID** takes its attribute value from its parent node
- *Id-Comma-List* takes its attribute value from its left sibling *Type*
- Computing attributes purely bottom-up is not sufficient in this case
- Do we need synthesized attributes in this grammar?

11/26/10

16



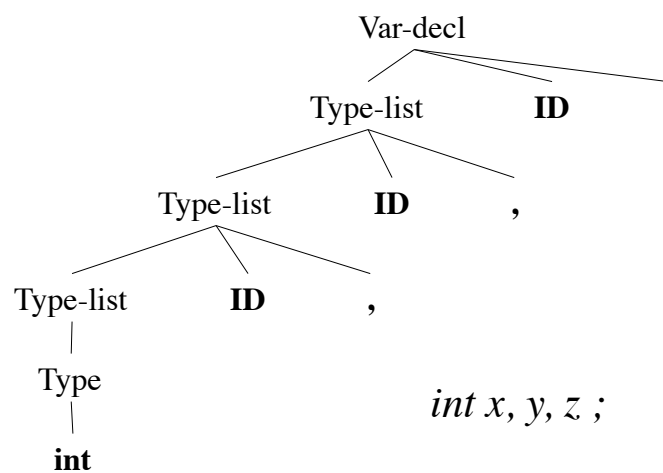
## Inherited Attributes

- **Inherited attributes** are attributes that are computed at a node based on attributes from siblings or the parent
- Typically we combine synthesized attributes and inherited attributes
- It is possible to convert the grammar into a form that *only* uses synthesized attributes

11/26/10

17

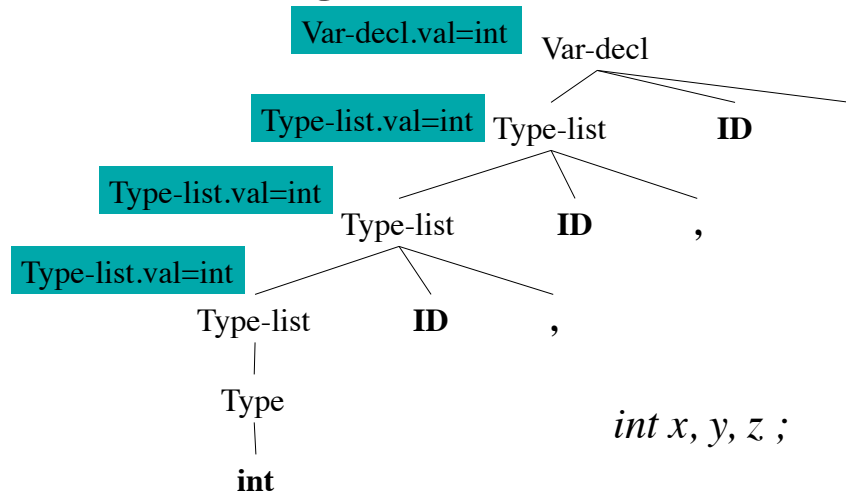
## Removing Inherited Attributes



11/26/10

18

## Removing Inherited Attributes



11/26/10

19

## Removing inherited attributes

Var-decl  $\rightarrow$  Type-List **ID** ;

{ \$0.val = \$1.val; }

Type-list  $\rightarrow$  Type-list **ID** ,

{ \$0.val = \$1.val; }

Type-list  $\rightarrow$  Type

{ \$0.val = \$1.val; }

Type  $\rightarrow$  **int** | **bool**

{ \$0.val = int; } & { \$0.val = bool; }

11/26/10

20

## Direction of inherited attributes

- Consider the syntax directed defns:

$A \rightarrow L M$

$\{ \$1.in = \$0.in; \$2.in = \$1.val; \$0.val = \$2.val; \}$

$A \rightarrow Q R$

$\{ \$2.in = \$0.in; \$1.in = \$2.val; \$0.val = \$1.val; \}$

- Problematic definition:  $\$1.in = \$2.val$
- Difference between incremental processing vs. using the completed parse tree

11/26/10

21

## Incremental Processing

- Incremental processing: constructing output as we are parsing
- Bottom-up or top-down parsing
- Both can be viewed as left-to-right and depth-first construction of the parse tree
- Some inherited attributes cannot be used in conjunction with incremental processing

11/26/10

22

## L-attributed Definitions

- A syntax-directed definition is **L-attributed** if for a CFG rule

$A \rightarrow X_1..X_{j-1}X_j..X_n$  two conditions hold:

- Each inherited attribute of  $X_j$  depends on  $X_1..X_{j-1}$
- Each inherited attribute of  $X_j$  depends on  $A$
- These two conditions ensure left to right and depth first parse tree construction
- Every S-attributed definition is L-attributed

11/26/10

23

## Syntax-directed defns

- Two important classes of SDTs:
  1. LR parser, syntax directed definition is S-attributed
  2. LL parser, syntax directed definition is L-attributed

11/26/10

24

## Syntax-directed defns

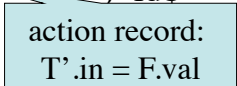
- LR parser, S-attributed definition
  - Implementing S-attributed definitions in LR parsing is easy: execute action on reduce, all necessary attributes have to be on the stack
- LL parser, L-attributed definition
  - Implementing L-attributed definitions in LL parsing is similarly easy: we use an additional action record for storing synthesized and inherited attributes on the parse stack

11/26/10

25

## Syntax-directed defns

- LR parser, S-attributed definition
  - more details later ...
- LL parser, L-attributed definition

Stack	Input	Output
\$T')T'F	id)*id\$	$T \rightarrow F T' \{ \$2.in = \$1.val \}$
\$T')T'id	id)*id\$	$F \rightarrow id \{ \$0.val = \$1.val \}$
\$T')T'  )*id\$		The action record stays on the stack when T' is replaced with rhs of rule

11/26/10

26

## Top-down translation

- Assume that we have a top-down predictive parser
- Typical strategy: take the CFG and eliminate left-recursion
- Suppose that we start with an attribute grammar
- Can we still eliminate left-recursion?

11/26/10

27

## Top-down translation

```

E → E + T
    { $0.val = $1.val + $3.val; }
E → E - T
    { $0.val = $1.val - $3.val; }
T → IntConstant
    { $0.val = $1.lexval; }
E → T
    { $0.val = $1.val; }
T → ( E )
    { $0.val = $2.val; }

```

11/26/10

28

## Top-down translation

$$E \rightarrow T R$$

$$\{ \$2.in = \$1.val; \$0.val = \$2.val; \}$$

$$R \rightarrow + T R$$

$$\{ \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; \}$$

$$R \rightarrow - T R$$

$$\{ \$3.in = \$0.in - \$2.val; \$0.val = \$3.val; \}$$

$$R \rightarrow \epsilon \{ \$0.val = \$0.in; \}$$

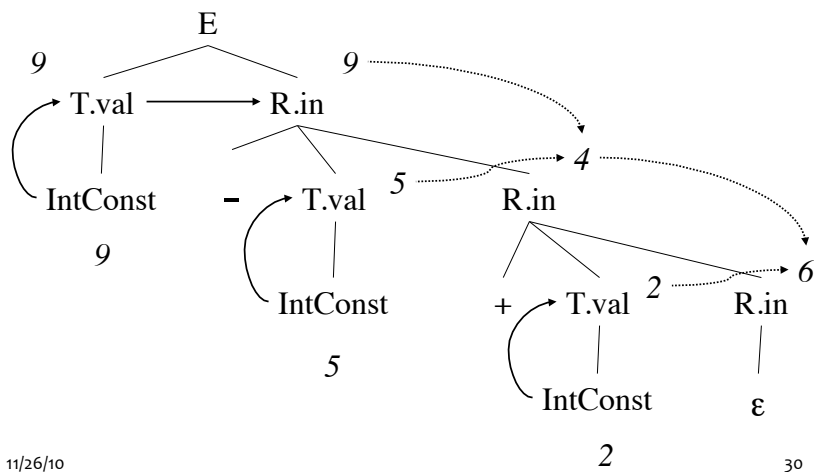
$$T \rightarrow ( E ) \{ \$0.val = \$2.val; \}$$

$$T \rightarrow \text{IntConstant} \{ \$0.val = \$1.lexval; \}$$

11/26/10

29

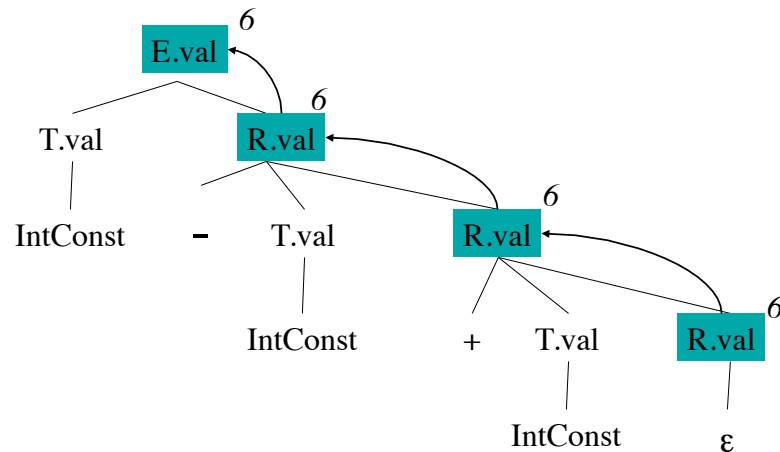
## Example: 9 - 5 + 2



11/26/10

30

## Example: $9 - 5 + 2$



11/26/10

31

## Dependencies and SDTs

- There can be circular definitions:

$A \rightarrow B \{ \$0.val = \$1.in; \$1.in = \$0.val + 1; \}$

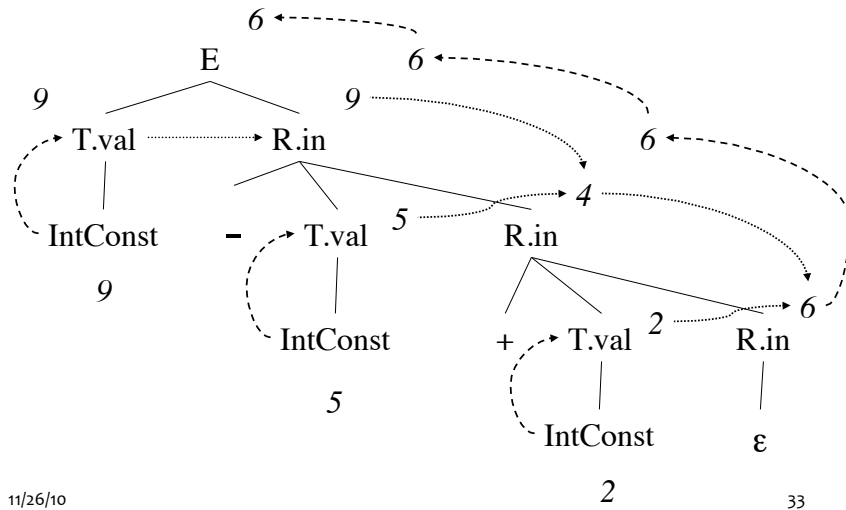
- It is impossible to evaluate either  $\$0.val$  or  $\$1.in$  first (each value depends on the other)
- We want to avoid circular dependencies
- Detecting such cases in all parse trees takes exponential time!
- S-attributed or L-attributed definitions cannot have cycles

11/26/10

32



## Dependency Graphs



11/26/10

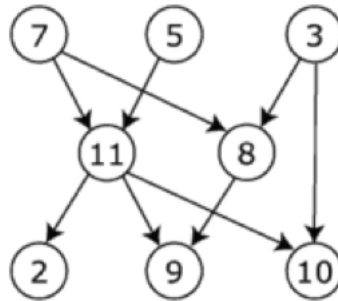
33

## Dependency Graphs

- A dependency graph is drawn based on the syntax directed definition
- Each dependency shows the flow of information in the parse tree
- There are many ways to order these dependencies
- Each ordering is called a **topological sort** of the dependency edges
- A graph with a cycle has no possible topological sorting

11/26/10

34



The graph shown to the left has many valid topological sorts, including:

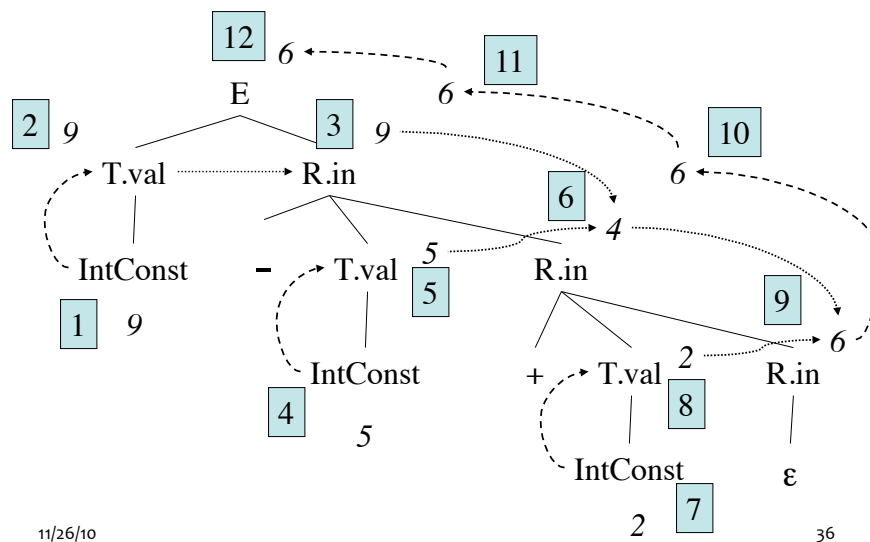
- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (least number of edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10

11/26/10

Source: Wikipedia

35

## Dependency Graphs



11/26/10

36

## Dependency Graphs

- A topological sort is defined on a set of nodes  $N_1, \dots, N_k$  such that if there is an edge in the graph from  $N_i$  to  $N_j$  then  $i < j$
- One possible topological sort for previous dependency graph is:
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
- Another possible sorting is:
  - 4, 5, 7, 8, 1, 2, 3, 6, 9, 10, 11, 12

11/26/10

37

## Syntax-directed definition with actions

- Some definitions can have side-effects:
- $E \rightarrow T R \{ \text{printf}("%s", \$2); \}$
- Can we predict when these side-effects will occur?
  - In general, we cannot and so the translation will depend on the parser

11/26/10

38

## Syntax-directed definition with actions

- A definition with side-effects:  

$$E \rightarrow T R \{ \text{printf}("%s", \$2); \}$$
- We can impose a condition: allow side-effects if the definition obeys a condition:
  - The same translation is produced for **any topological sort** of the dependency graph
- In the above example, this is true because the print statement is executed at the end

11/26/10

39

## SDTs with Actions

- A syntax directed definition that maps infix expressions to postfix:

$$E \rightarrow T R$$

$$R \rightarrow + T \{ \text{print}(' + '); \} R$$

$$R \rightarrow - T \{ \text{print}(' - '); \} R$$

$$R \rightarrow \varepsilon$$

$$T \rightarrow \text{id} \{ \text{print}(\text{id.lookup}); \}$$

11/26/10

40

## SDTs with Actions

- A buggy syntax directed definition that tries to map infix expressions to prefix:

$E \rightarrow T R$

$R \rightarrow \{ \text{print( '+' ); } \} + T R$

$R \rightarrow \{ \text{print( '-' ); } \} - T R$

$R \rightarrow \epsilon$

$T \rightarrow \text{id} \{ \text{print( id.lookup ); } \}$

Problematic for  
left to right  
processing.  
Translation on  
the parse tree is  
possible

11/26/10

41

## LR parsing and inherited attributes

- As we just saw, inherited attributes are possible when doing top-down parsing
- How can we compute inherited attributes in a bottom-up shift-reduce parser
- Problem: doing it incrementally (while parsing)
- Note that LR parsing implies depth-first visit which matches L-attributed definitions

11/26/10

42

## LR parsing and inherited attributes

- Attributes can be stored on the stack used by the shift-reduce parsing
- For synthesized attributes: when a reduce action is invoked, store the value on the stack based on value popped from stack
- For inherited attributes: transmit the attribute value when executing the **goto** function

11/26/10

43

## Example: Synthesized Attributes

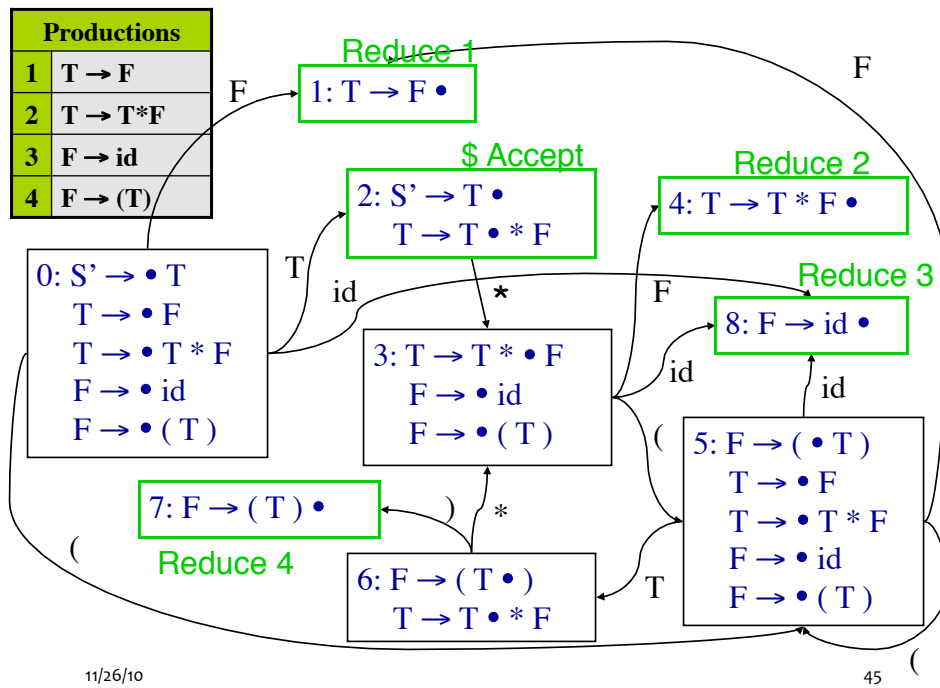
```

T → F    { $0.val = $1.val; }
T → T * F
    { $0.val = $1.val * $3.val; }
F → id
    { val := id.lookup();
      if (val) { $0.val = $1.val; }
      else { error; } }
F → ( T ) { $0.val = $2.val; }

```

11/26/10

44



Trace “(id<sub>val=3</sub>)\*id<sub>val=2</sub>”

Stack	Input	Action	Attributes
0	( id ) * id \$	Shift 5	
0 5	id ) * id \$	Shift 8	
0 5 8	) * id \$	Reduce 3 $F \rightarrow id$ , pop 8, goto [5,F]=1	<b>a.Push id.val=3;</b> { \$0.val = \$1.val }
0 5 1	) * id \$	Reduce 1 $T \rightarrow F$ , pop 1, goto [5,T]=6	<b>a.Pop; a.Push 3;</b> { \$0.val = \$1.val }
0 5 6	) * id \$	Shift 7	<b>a.Pop; a.Push 3;</b>
0 5 6 7	* id \$	Reduce 4 $F \rightarrow (T)$ , pop 7 6 5, goto [0,F]=1	{ \$0.val = \$2.val } <b>3 pops; a.Push 3</b>

Trace “(id<sub>val=3</sub>)\*id<sub>val=2</sub>”

Stack	Input	Action	Attributes
0 1	* id \$	<b>Reduce 1 T→F, pop 1, goto [0,T]=2</b>	{ \$0.val = \$1.val } <b>a.Pop; a.Push 3</b>
0 2	* id \$	<b>Shift 3</b>	<b>a.Push mul</b>
0 2 3	id \$	<b>Shift 8</b>	<b>a.Push id.val=2</b>
0 2 3 8	\$	<b>Reduce 3 F→id, pop 8, goto [3,F]=4</b>	<b>a.Pop a.Push 2</b>
0 2 3 4	\$	<b>Reduce 2 T→T * F pop 4 3 2, goto [0,T]=2</b>	{ \$0.val = \$1.val * \$3.val; }
0 2	\$	<b>Accept</b>	<b>3 pops; a.Push 3*2=6</b>

11/26/10

47

## Example: Inherited Attributes

$E \rightarrow T R$

{ \$2.in = \$1.val; \$0.val = \$2.val; }

$R \rightarrow + T R$

{ \$3.in = \$0.in + \$2.val; \$0.val = \$3.val; }

$R \rightarrow \epsilon$  { \$0.val = \$0.in; }

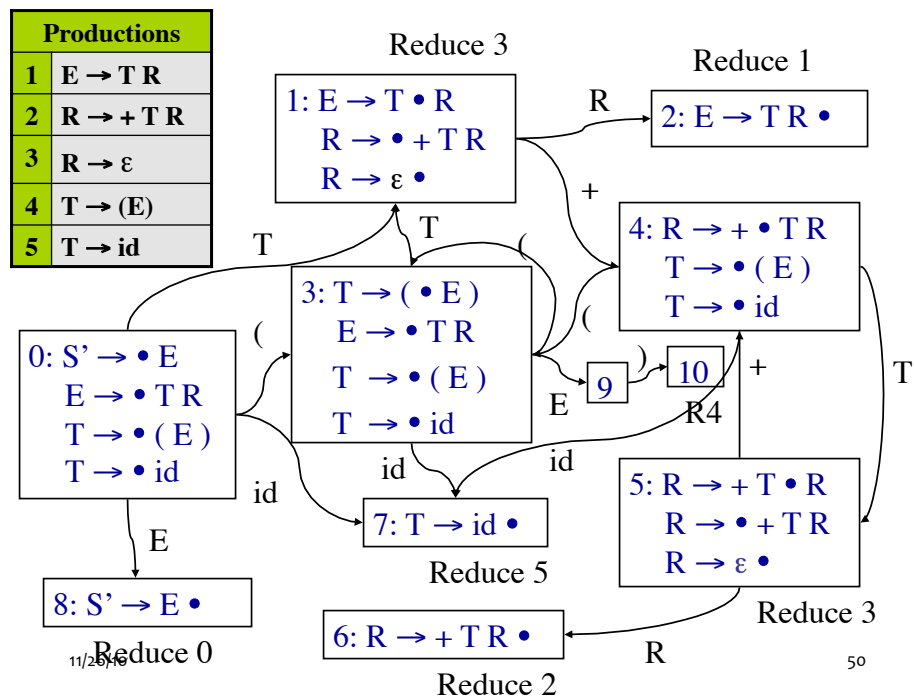
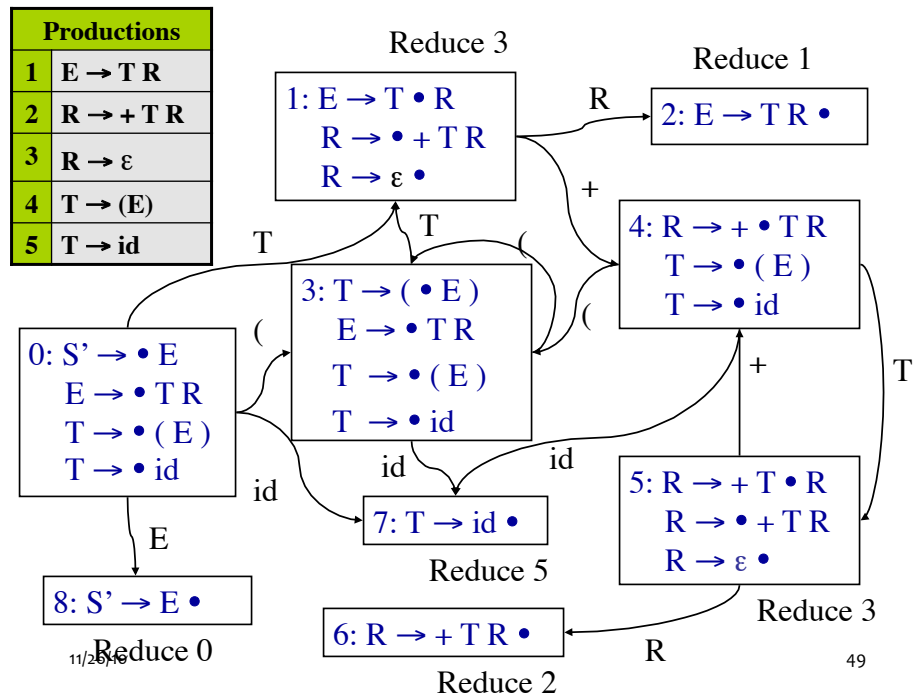
$T \rightarrow ( E )$  { \$0.val = \$1.val; }

$T \rightarrow \text{id}$  { \$0.val = id.lookup; }

11/26/10

48





Productions			
1	$E \rightarrow T R$ { $\$2.in = \$1.val$ ; $\$0.val = \$2.val$ ; }		
2	$R \rightarrow + T R$ { $\$3.in = \$0.in + \$2.val$ ; $\$0.val = \$3.val$ ; }		
3	$R \rightarrow \epsilon$ { $\$0.val = \$0.in$ ; }		
4	$T \rightarrow (E)$ { $\$0.val = \$1.val$ ; }		
5	$T \rightarrow id$ { $\$0.val = id.lookup$ ; }		

Stack	Input	Action	Attributes
0 7	$id + id \$$	Reduce 5 $T \rightarrow id$	{ $\$0.val = id.lookup$ }
0 1	$+ id \$$	pop 7, goto [0,T]=1	{ pop; attr.Push(3)
0 1 4	$id \$$	Shift 4	$\$2.in = \$1.val$
0 1 4 7	$\$$	Shift 7	$\$2.in := (1).attr$
0 1 4 5	$\$$	Reduce 5 $T \rightarrow id$	{ $\$0.val = id.lookup$ }
		pop 7, goto [4,T]=5	{ pop; attr.Push(2); }
		Reduce 3 $R \rightarrow \epsilon$	{ $\$3.in = \$0.in + \$1.val$
		goto [5,R]=6	$(5).attr := (1).attr + 2$
			$\$0.val = \$0.in$
			$\$0.val = (5).attr = 5$

11/26/10

Trace “ $id_{val=3} + id_{val=2}$ ”

Stack	Input	Action	Attributes
0	$id + id \$$	Shift 7	
0 7	$+ id \$$	Reduce 5 $T \rightarrow id$	{ $\$0.val = id.lookup$ }
0 1	$+ id \$$	pop 7, goto [0,T]=1	{ pop; attr.Push(3)
0 1 4	$id \$$	Shift 4	$\$2.in = \$1.val$
0 1 4 7	$\$$	Shift 7	$\$2.in := (1).attr$
0 1 4 5	$\$$	Reduce 5 $T \rightarrow id$	{ $\$0.val = id.lookup$ }
		pop 7, goto [4,T]=5	{ pop; attr.Push(2); }
		Reduce 3 $R \rightarrow \epsilon$	{ $\$3.in = \$0.in + \$1.val$
		goto [5,R]=6	$(5).attr := (1).attr + 2$
			$\$0.val = \$0.in$
			$\$0.val = (5).attr = 5$

11/26/10

Trace “ $\text{id}_{\text{val}=3} + \text{id}_{\text{val}=2}$ ”

Stack	Input	Action	Attributes
<b>0 1 4 5 6</b>		\$ <b>Reduce 2 <math>R \rightarrow + T R</math></b> <b>Pop 4 5 6, goto [1,R]=2</b>	{ \$0.val = \$3.val pop; attr.Push(5); }
<b>0 1 2</b>		\$ <b>Reduce 1 <math>E \rightarrow T R</math></b> <b>Pop 1 2, goto [0,E]=8</b>	{ \$0.val = \$3.val pop; attr.Push(5); }
<b>0 8</b>		\$ <b>Accept</b>	{ \$0.val = 5 attr.top = 5; }

11/26/10

53

## LR parsing with inherited attributes

Bottom-Up/rightmost	
ccbca $\Leftarrow$ Acbca	$A \rightarrow c$
$\Leftarrow$ AcbB	$B \rightarrow ca$
<b>line 3</b> $\Leftarrow$ AB	$B \rightarrow cbB$
$\Leftarrow$ S	$S \rightarrow AB$

Parse stack at line 3:

['x'] A ['x'] c b B

\$1.in = 'x'

\$2.in = \$1.val

11/26/10

Consider:

 $S \rightarrow AB$ 

{ \$1.in = 'x';  
\$2.in = \$1.val }

 $B \rightarrow cbB$ 

{ \$0.val = \$0.in + 'y'; }

Parse stack at line 4:

['x'] A B

['xy']

54

## Marker non-terminals

- Convert L-attributed into S-attributed definition
- Prerequisite: use embedded actions to compute inherited attributes, e.g.

$$R \rightarrow + T \{ \$3.in = \$0.in + \$2.val; \} R$$

- For each embedded action introduce a new marker non-terminal and replace action with the marker

$$R \rightarrow + T M R$$

$$M \rightarrow \epsilon \{ \$0.val = \$-1.val - \$-3.in; \}$$

11/26/10

note the use of  $-1$ ,  $-2$ , etc. to access attributes

55

## Marker Non-terminals

$$E \rightarrow T R$$

$$R \rightarrow + T \{ \text{print}(' + '); \} R$$

$$R \rightarrow - T \{ \text{print}(' - '); \} R$$

$$R \rightarrow \epsilon$$

$$T \rightarrow \text{id} \{ \text{print}(\text{id.lookup}); \}$$

Actions that should be done after recognizing T but before predicting R

11/26/10

56

## Marker Non-terminals

$$E \rightarrow T R$$

$$R \rightarrow + T M R$$

$$R \rightarrow - T N R$$

$$R \rightarrow \epsilon$$

$$T \rightarrow \text{id} \{ \text{print}(\text{id.lookup}); \}$$

$$M \rightarrow \epsilon \{ \text{print}(' '); \}$$

$$N \rightarrow \epsilon \{ \text{print}('- '); \}$$

Equivalent SDT using  
*marker non-terminals*

11/26/10

57

## Impossible Syntax-directed Definition

$$E \rightarrow \{ \text{print}(' '); \} E + T$$

$$E \rightarrow T$$

$$T \rightarrow \{ \text{print}('* '); \} T * R$$

$$T \rightarrow F$$

$$T \rightarrow \text{id} \{ \text{print} \$1.lexval; \}$$

Tries to convert  
infix to prefix

Impossible either top-down or  
bottom-up. Problematic only  
for left-to-right processing, ok  
for generation from parse tree.

11/26/10

58

## Tree Matching Code Generators

- Write tree patterns that match portions of the parse tree
- Each tree pattern can be associated with an action (just like attribute grammars)
- There can be multiple combinations of tree patterns that match the input parse tree

11/26/10

59

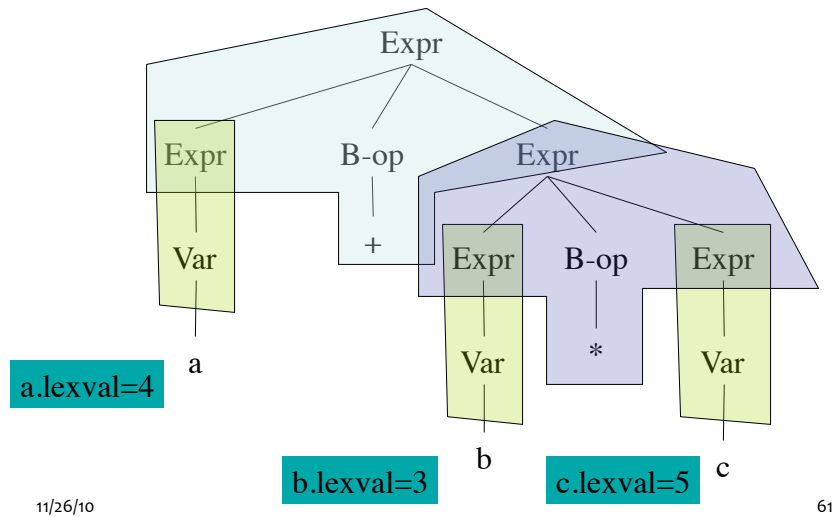
## Tree Matching Code Generators

- To provide a unique output, we assign costs to the use of each tree pattern
- E.g. assigning uniform costs leads to smaller code or instruction costs can be used for optimizing code generation
- Three algorithms: Maximal Munch, Dynamic Programming, Tree Grammars
- Section 8.9 (Purple Dragon book)

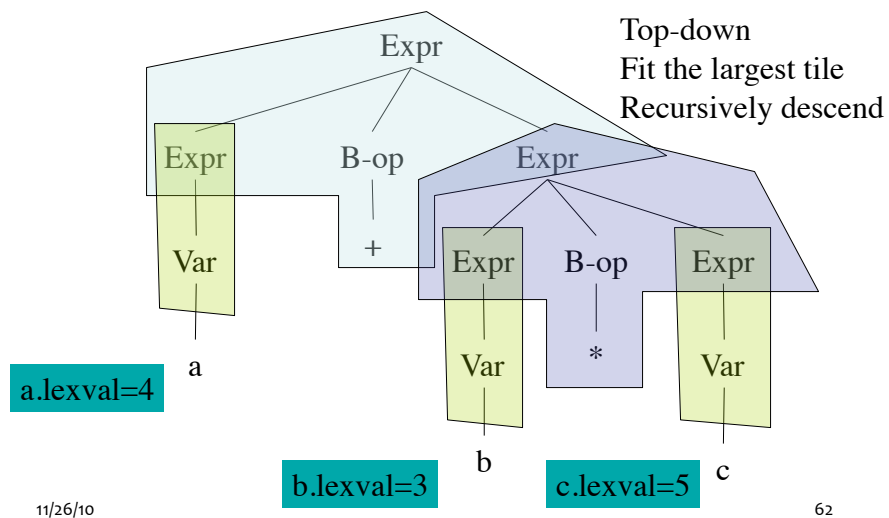
11/26/10

60

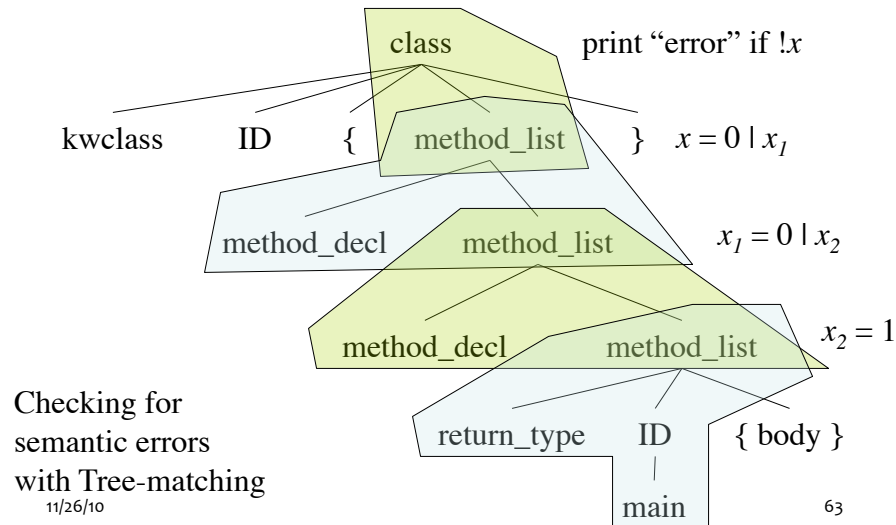
## Maximal Munch: Example 1



## Maximal Munch: Example 1



## Maximal Munch: Example 2



## Tree Parsing Code Generators

- Take the prefix representation of the syntax tree
  - E.g. (+ (\* c1 r1) (+ ma c2)) in prefix representation uses an inorder traversal to get + \* c1 r1 + ma c2
- Write CFG rules that match substrings of the above representation and non-terminals are registers or memory locations
- Each matching rule produces some predefined output

• Section 8.9.3 (Purple Dragon book)

64



## Code-generation Generators

- A CGG is like a compiler-compiler: write down a description and generate code for it
- Code generation by:
  - Adding semantic actions to the original CFG and each action is executed while parsing, e.g. yacc
  - Tree Rewriting: match a tree and commit an action, e.g. lcc
  - Tree Parsing: use a grammar that generates trees (not strings), e.g. twig, burs, iburg

11/26/10

65

## Summary

- The parser produces concrete syntax trees
- Abstract syntax trees: define semantic checks or a syntax-directed translation to the desired output
- Attribute grammars: static definition of syntax-directed translation
  - Synthesized and Inherited attributes
  - S-attribute grammars
  - L-attributed grammars
- Complex inherited attributes can be defined if the full parse tree is available

11/26/10

66