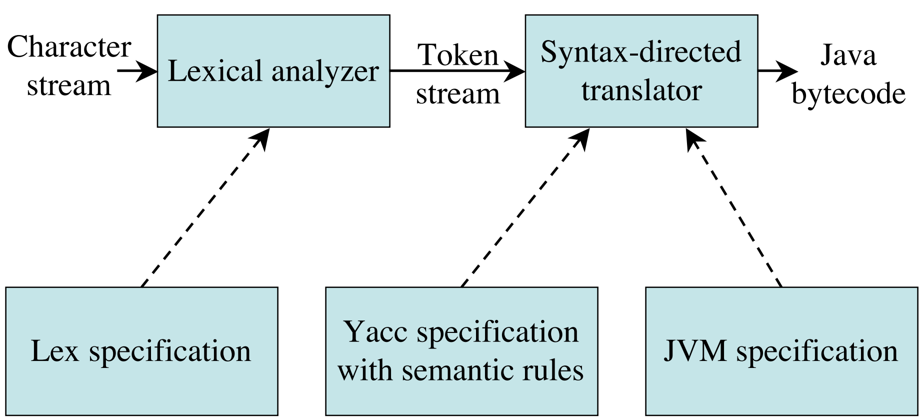
Syntax-Directed Translation
Part I

Chapter 5

COP5621 Compiler Construction

The Structure of our Compiler Revisited



Syntax-Directed Definitions

- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes*
- A *depth-first traversal* algorithm is used to compute the values of the attributes in the parse tree using the semantic rules
- After the traversal is completed, the attributes contain the translated form of the input

Example Attribute Grammar

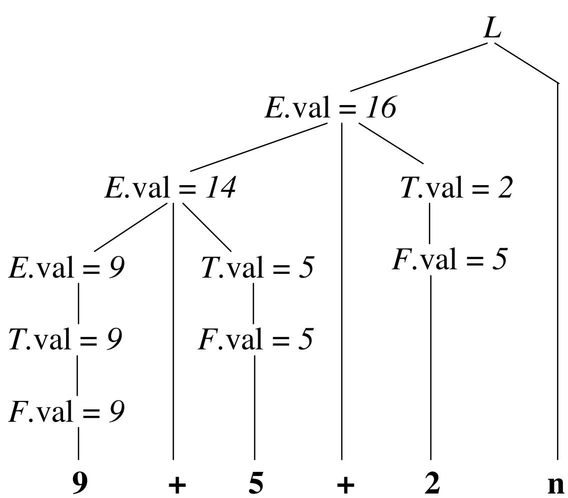
Production	Semantic Rule
$L \rightarrow E$ **n**	*print*(E.val)
$E \rightarrow E_1$ **+** T	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1$ ***** F	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow$ **(** E **)**	F.val := E.val
$F \rightarrow$ **digit**	F.val := **digit**.lexval

Note: all attributes in this example are of the synthesized type

Example Attribute Grammar in Yacc

```
%token DIGIT
%%
L : E '\n'          { printf("%d\n", $1); }
  ;
E : E '+' T         { $$ = $1 + $3; }
  | T               { $$ = $1; }
  ;
T : T '*' F         { $$ = $1 * $3; }
  | F               { $$ = $1; }
  ;
F : '(' E ')'       { $$ = $2; }
  | DIGIT           { $$ = $1; }
  ;
%%
```


Example Annotated Parse Tree



Note: all attributes in this example are of the synthesized type

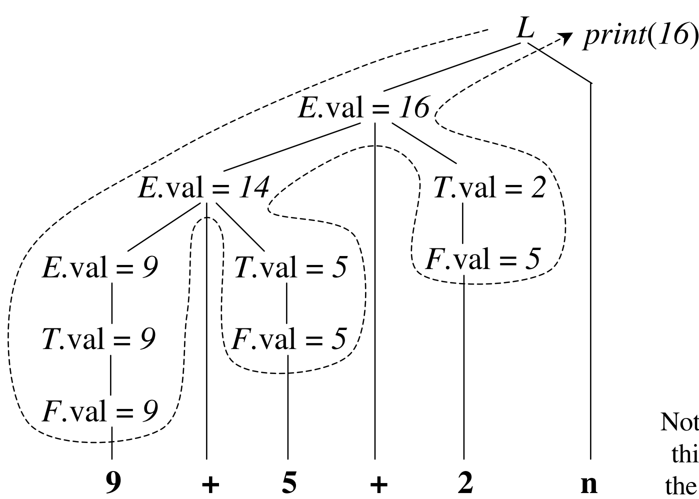
7

Annotating a Parse Tree With Depth-First Traversals

procedure *visit*(*n* : *node*);
begin
  for each child *m* of *n*, from left to right **do**
    visit(*m*);
  evaluate semantic rules at node *n*
end

8

Depth-First Traversals (Example)



Note: all attributes in this example are of the synthesized type

Attributes

- Attribute values can represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Synthesized Versus Inherited Attributes

- Given a production

  $A \to \alpha$

  then each semantic rule is of the form

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

  where f is a function and c_i are attributes of A and α, and either
  - b is a *synthesized* attribute of A
  - b is an *inherited* attribute of one of the grammar symbols in α

Synthesized Versus Inherited Attributes (cont’d)

Production	Semantic Rule
$D \to T\ L$	L.in := T.type
$T \to$ **int**	T.type := ‘integer’
…	…
$L \to$ **id**	… := L.in

inherited

synthesized

S-Attributed Definitions

- A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)
- A parse tree of an S-attributed definition can be annotated with a simple bottom-up traversal
- Yacc only supports S-attributed definitions

Bottom-up Evaluation of S-Attributed Definitions in Yacc

13

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \to$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \to F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *****	*3 _*	**5+4n$**	shift	
$ *T* *** 5**	*3 _ 5*	**+4n$**	reduce $F \to$ **digit**	$$ = $1
$ *T* ***** *F*	*3 _ 5*	**+4n$**	reduce $T \to T$ * F	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \to T$	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* **+**	*15 _*	**4n$**	shift	
$ *E* **+ 4**	*15 _ 4*	**n$**	reduce $F \to$ **digit**	$$ = $1
$ *E* **+** *F*	*15 _ 4*	**n$**	reduce $T \to F$	$$ = $1
$ *E* **+** *T*	*15 _ 4*	**n$**	reduce $E \to E$ **+** T	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19 _*	**$**	reduce $L \to E$ **n**	print $1
$ *L*	*19*	**$**	accept	

14

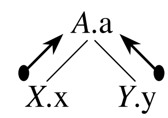
Example Attribute Grammar with Synthesized+Inherited Attributes

Production	Semantic Rule
$D \to T\ L$	*L*.in := *T*.type
$T \to$ **int**	*T*.type := 'integer'
$T \to$ **real**	*T*.type := 'real'
$L \to L_1$ **, id**	L_1.in := *L*.in; *addtype*(**id**.entry, *L*.in)
$L \to$ **id**	*addtype*(**id**.entry, *L*.in)

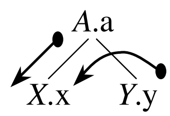
Synthesized: *T*.type, **id**.entry
Inherited: *L*.in

Acyclic Dependency Graphs for Parse Trees

$A \rightarrow X\ Y$

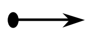


$A.\mathrm{a} := f(X.\mathrm{x},\ Y.\mathrm{y})$

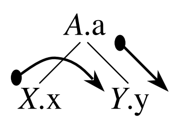


$X.\mathrm{x} := f(A.\mathrm{a},\ Y.\mathrm{y})$

Direction of



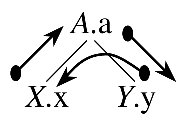
value dependence



$Y.\mathrm{y} := f(A.\mathrm{a},\ X.\mathrm{x})$

Dependency Graphs with Cycles?

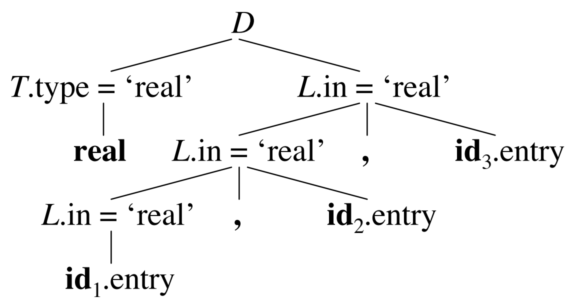
- Edges in the dependence graph show the evaluation order for attribute values
- Dependency graphs cannot be cyclic



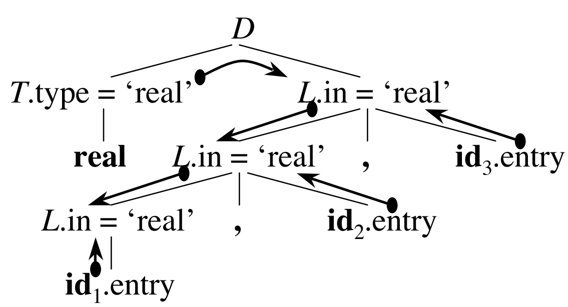
$A.\mathrm{a} := f(X.\mathrm{x})$
$X.\mathrm{x} := f(Y.\mathrm{y})$
$Y.\mathrm{y} := f(A.\mathrm{a})$

Error: cyclic dependence

Example Annotated Parse Tree



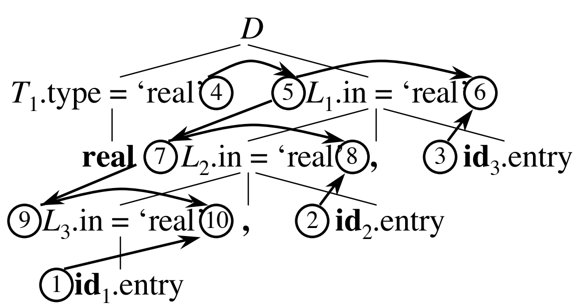
Example Annotated Parse Tree with Dependency Graph



Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge then m_i appears before m_j
- Any topological sort of a dependency graph gives a valid evaluation order for the semantic rules

Example Parse Tree with Topologically Sorted Actions



Topological sort:

1. Get **id**$_1$.entry
2. Get **id**$_2$.entry
3. Get **id**$_3$.entry
4. T_1.type='real'
5. L_1.in=T_1.type
6. *addtype*(**id**$_3$.entry, L_1.in)
7. L_2.in=L_1.in
8. *addtype*(**id**$_2$.entry, L_2.in)
9. L_3.in=L_2.in
10. *addtype*(**id**$_1$.entry, L_3.in)

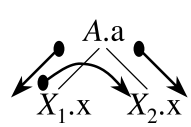
Evaluation Methods

- *Parse-tree methods* determine an evaluation order from a topological sort of the dependence graph constructed from the parse tree for each input
- *Rule-base methods* the evaluation order is pre-determined from the semantic rules
- *Oblivious methods* the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)

L-Attributed Definitions

- The example parse tree on slide 18 is traversed "in order", because the direction of the edges of inherited attributes in the dependence graph point top-down and from left to right
- More precisely, a syntax-directed definition is *L-attributed* if each inherited attribute of X_j on the right side of $A \rightarrow X_1\ X_2\ \ldots\ X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

Shown: dependences of inherited attributes

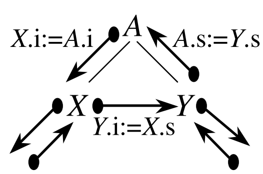


23

L-Attributed Definitions (cont'd)

- L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

$A \rightarrow X\ Y$



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

- Note: every S-attributed syntax-directed definition is also L-attributed

24

Using Translation Schemes for L-Attributed Definitions

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Translation Scheme

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id** { *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

25

Implementing L-Attributed Definitions in Top-Down Parsers

L-attributed definitions are implemented in translation schemes first:

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }

```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute

Input: inherited attribute

26

Implementing L-Attributed Definitions in Bottom-Up Parsers

- More difficult and also requires rewriting L-attributed definitions into translation schemes
- Insert marker nonterminals to remove embedded actions from translation schemes, that is
  $A \rightarrow X$ { actions } Y
  is rewritten with marker nonterminal N into
  $A \rightarrow X\, N\, Y$
  $N \rightarrow \varepsilon$ { actions }
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

Emulating the Evaluation of L-Attributed Definitions in Yacc

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id**
{ *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

```
%{
Type Lin; /* global variable */
%}
%%
D   : Ts L
    ;
Ts  : T         { Lin = $1; }
    ;
T   : INT       { $$ = TYPE_INT; }
    | REAL      { $$ = TYPE_REAL; }
    ;
L   : L ',' ID { addtype($3, Lin);}
    | ID        { addtype($1, Lin);}
    ;
%%
```


Rewriting a Grammar to Avoid Inherited Attributes

Production	Production	Semantic Rule
$D \rightarrow L$ **:** T	$D \rightarrow$ **id** L	*addtype*(**id**.entry, L.type)
$T \rightarrow$ **int**	$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	$L \rightarrow$ **, id** L_1	*addtype*(**id**.entry, L.type)
$L \rightarrow$ **id**	$L \rightarrow$ **:** T	L.type := T.type