Syntactic Directed Translation

Translation Schemes

Syntax-Directed Translations

- Syntax-Directed Translations
  - Semantic Actions are Embedded in Productions
- Single Pass Translation Schemes
  - Faster!
  - No need to Construct the Parse Tree and then do a Topological Sorting to Find out Feasible Order for Evaluation of
- Issues:
  - Dealing with Embedded Actions
    - May Require Inserting Additional Symbols, Markers
  - Dealing with Inherited Attributes
    - Reach into the Stack for Value of Attribute
    - Position Independence in the Stack
  - Synthesized Attributes are trivially Handled

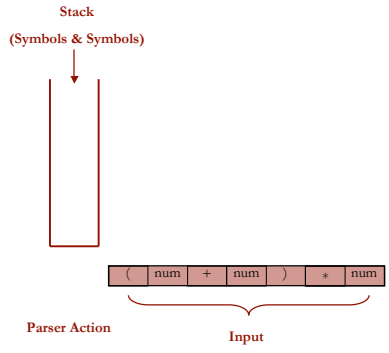
Example: Building an AST (*Abstract Syntax Tree*)

- Assume Constructors for Each Node (*e.g.,* malloc in C)
- Assume Stack holds Pointers to Nodes & Symbols
- Assume YACC Syntax with Refs to Stack Depth

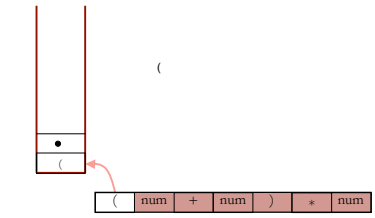
Goal	→	*Expr*	$$ = $1;
Expr	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
	|	*Expr* – *Term*	$$ = MakeSubNode($1,$3);
	|	*Term*	$$ = $1;
Term	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
	|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
	|	*Factor*	$$ = $1;
Factor	→	(*Expr*)	$$ = $2;
	|	number	$$ = MakeNumNode($1.token);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

Semantic Rules Executed on Reduction of corresponding Production During Parsing

Example — Building an AST

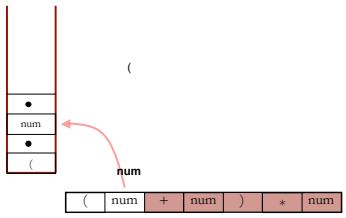


Example — Building an AST



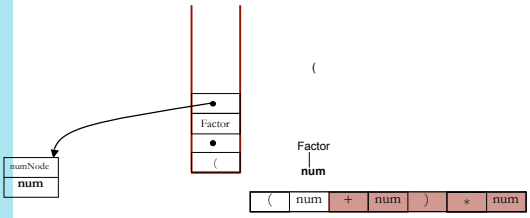
Shift: push '(' onto stack

Example — Building an AST



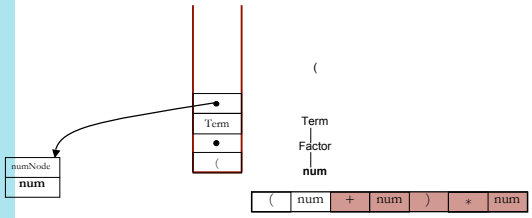
Shift: push 'num' onto stack

Example — Building an AST



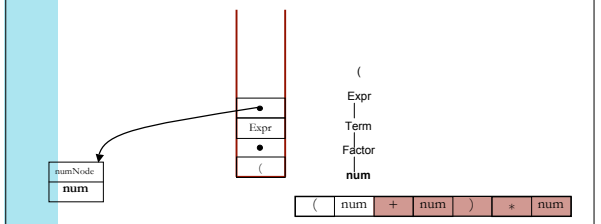
Reduce: Factor → num $$=MakeNumNode($1.token)

Example — Building an AST



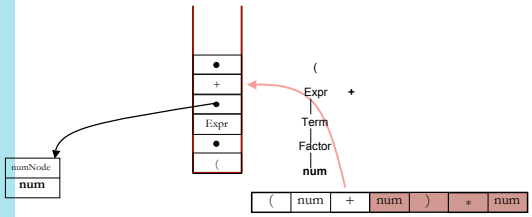
Reduce: Term → Factor $$ = $1

Example — Building a AST



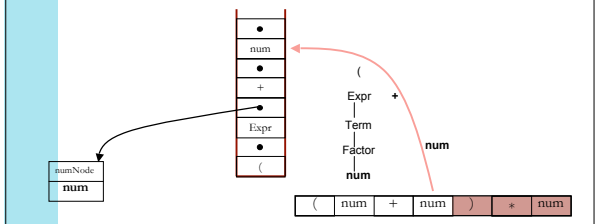
Reduce: Expr → Term $$ = $1

Example — Building an AST



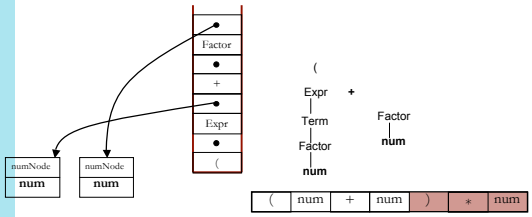
Shift: push '+' onto stack

Example — Building an AST



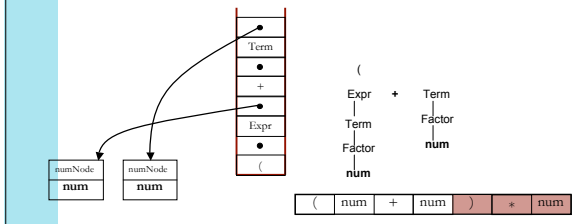
Shift: push 'num' onto stack

Example — Building a AST



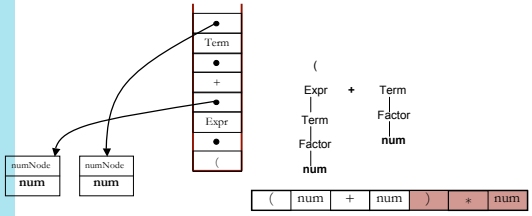
Reduce: Factor → num $$ = MakeNumNode($1.token)

Example — Building a AST

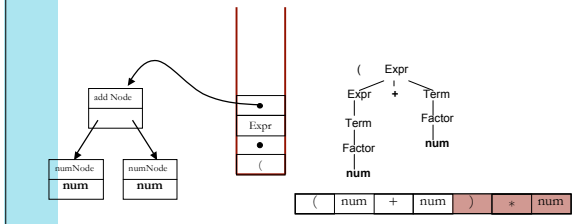


Reduce: Term → Factor $$ = $1

Example — Building a AST

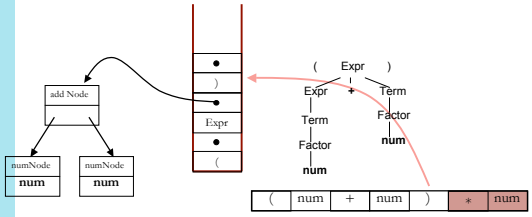


Example — Building an AST



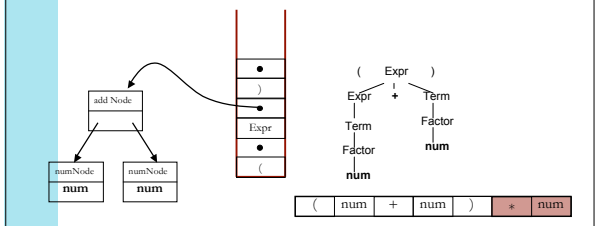
Reduce: Expr → Expr + Term $$ = MakeAddNode($1,$3)

Example — Building an AST

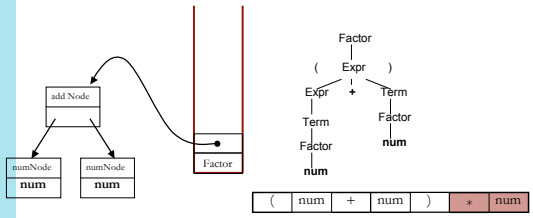


Shift: push ')' onto stack

Example — Building an AST

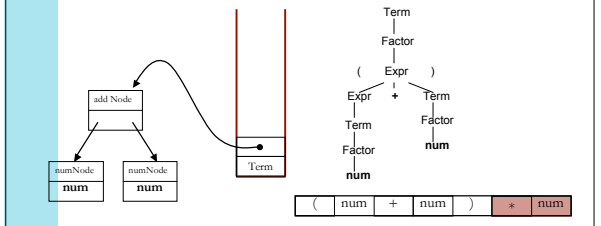


Example — Building an AST



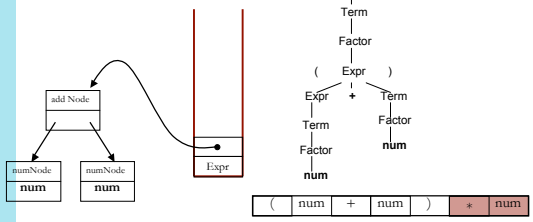
Reduce: Expr → (Expr) $$ = $2

Example — Building an AST



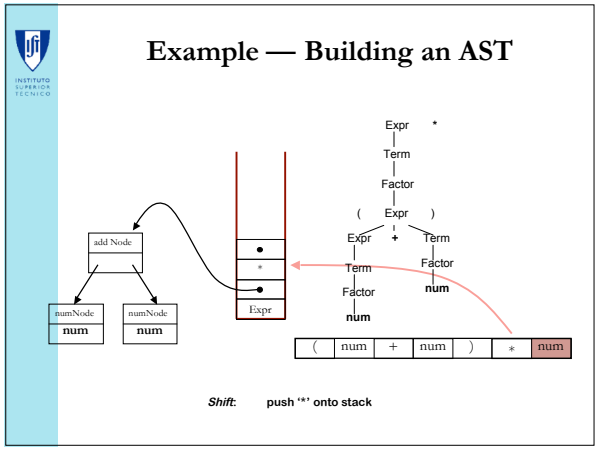
Reduce: Term → Factor $$ = $1

Example — Building an AST



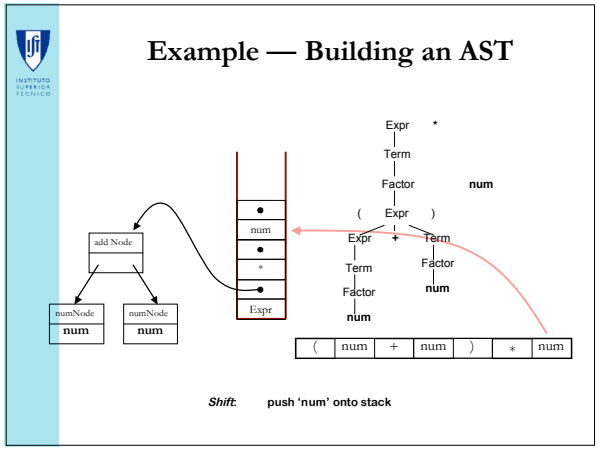
Reduce: Reduce → Term $$ = $1

Example — Building an AST



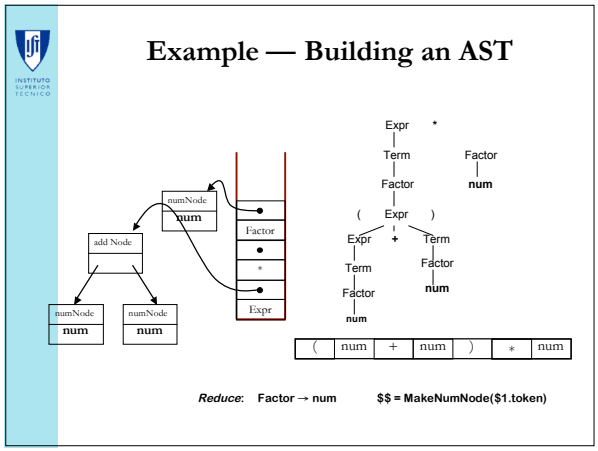
Shift: push '*' onto stack

Example — Building an AST



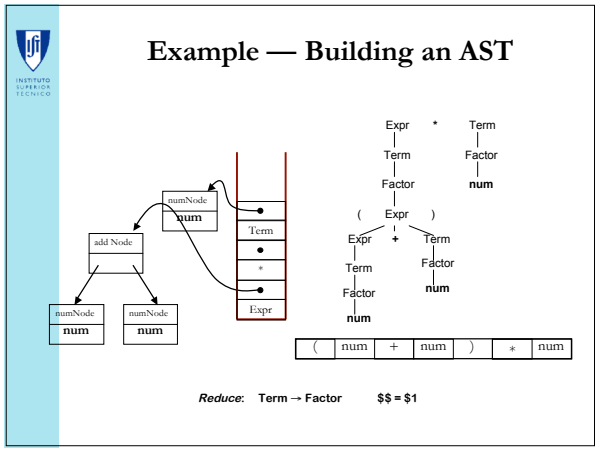
Shift: push 'num' onto stack

Example — Building an AST



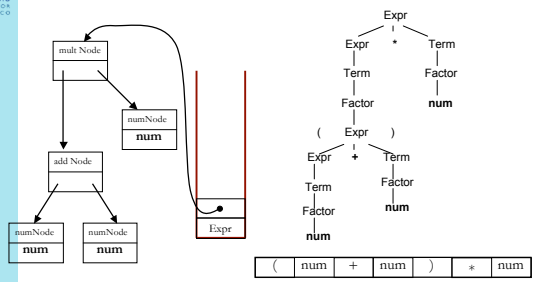
Reduce: Factor → num $$ = MakeNumNode($1.token)

Example — Building an AST



Reduce: Term → Factor $$ = $1

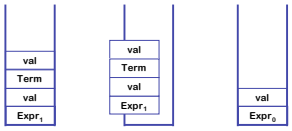
Example — Building an AST



Reduce: Expr → Expr * Term $$ = MakeMultNode($1,$3)

Example: Calculator Revisited

- Use Parser Stack for Symbols and Attribute Values
  - As Parser Algorithm Places Symbols
  - Semantic Actions use Stack locations for Storing Values
    - Reach Into Stack for Values of Symbols
    - When Reduction, Evaluate Semantic Action and Store Value of Production LHS



$Expr_0 \rightarrow Expr_1 * Term$ { $Expr_0.val = Expr_1.val * Term.val$ }

Example: Calculator Revisited

Production			Actions
Line	→	*Expr*	{ print(Expr.val); }
$Expr_0$	→	$Expr_0$ *+ Term*	{ $Expr_0.val = Expr_1.val + Term.val$; }
	|	*Term*	{ $Expr_0.val = Term.val$; }
$Term_0$	→	$Term_1$ ** Factor*	{ $Term_0.val = Term_1.val * Factor.val$; }
	|	*Factor*	{ $Term_0.val = Factor.val$; }
Factor	→	*(Expr)*	{ Factor.val = Expr.val; }
	|	digit	{ Factor.val = digit.lexval; }

Example: Calculator Revisited

Production			Actions
Line	→	*Expr*	{ print(Expr.val); }
$Expr_0$	→	$Expr_0$ *+ Term*	{ $Expr_0.val = Expr_1.val + Term.val$; }
	|	*Term*	{ $Expr_0.val = Term.val$; }
$Term_0$	→	$Term_1$ ** Factor*	{ $Term_0.val = Term_1.val * Factor.val$; }
	|	*Factor*	{ $Term_0.val = Factor.val$; }
Factor	→	*(Expr)*	{ Factor.val = Expr.val; }
	|	digit	{ Factor.val = digit.lexval; }

Line	→	*Expr*	{ print(stack[top].val); }
$Expr_0$	→	$Expr_0$ *+ Term*	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
	|	*Term*	{ }
$Term_0$	→	$Term_1$ ** Factor*	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; }
	|	*Factor*	{ }
Factor	→	*(Expr)*	{ stack[top-2].val = stack[top-1].val; top = top – 2;}
	|	digit	{ stack[top].val = digit.lexval; }

Translation Schemes

- L-Attributed Syntax-Directed Definition where:
  - Embed Semantic Actions in { }
  - Positioned Between Symbols of Production
  - Useful for Specifying Translation During Parsing
- Attribute Values Must be Available when Actions Refer to it.
  1. Inherited attributes for a symbol on the RHS of a production must be computed in an action before that Symbol.
  2. An action must not refer to a synthesized attribute of a symbol to the right of the action.
  3. A synthesized attribute for the symbol on the RHS of the production can only be computed after all attributes it references have been computed.
- Easily Implemented in Bottom-Up Parsers

L-Attributed Definitions

- A Syntax-Directed Definition is L-Attributed if
  - Each inherited Attribute $X_j, 1 \le j \le n$, for $A \rightarrow X_1 X_2 ,..., X_n$ depends only on:
    - The Attributes of the Symbols $X_1, X_2 ,..., X_{j-1}$ to the left of X_j
    - The Inherited Attributes of A
- Values Flow from Left-to-Right in the Parse Tree.
- Still an S-Attributed Definition
  - Restrictions are for Inherited Attributes Only.
- Can be Evaluated in a Depth-first Traversal
- In Many Cases even in a Single Pass

L-Attributed Def.: Evaluation Order

```
procedure dfvisit(n:node)
begin
  foreach child m of n from left to right do
    evaluate inherited attributes of m;
    dfvisit(m);
  end
  evaluate synthesized attributes of n
end
```

- What to do When?
  - Embedded Actions
  - Inherited Attributes
  - Replacing Inherited Attributes by Synthesized Attributes

Embedded Actions

- Actions are Executed when Parser Reduces a Production
  - After reductions for the RHS have occurred
  - Values for the Symbols available on the Stack
- What to do with Embedded Actions?
  - A → X { action } Y Z

Embedded Actions

- Actions are Executed when Parser Reduces a Production
  - After reductions for the RHS have occurred
  - Values for the Symbols available on the stack
- What to do with Embedded Actions?
  - A → X { action } Y Z
  - The action should execute before the actions for Y and Z
- Transform the Grammar adding a *Marker* Symbol using an empty RHS production for *Marker*
  - A → X *M* Y Z
  - *M* → ε { action }

Inherited Attributes

- For the production A → X Y when the parser reduces X's production, it's attributes will be on the top of the stack
  - If Y uses synthesized attributes X.s from X just needs to copy value from the top of the stack into computation of Y's attributes
- Observation: Reaching into the Stack works…
  - If you know the position of the symbol's inherited attribute
  - By looking at the corresponding grammar's production

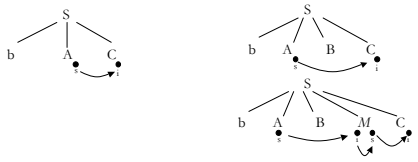
Production	Semantic Rule
S → aAC	C.i =A.s
S → bABC	C.i =A.s
C → c	C.s =func(C.i)

- Problem:
  - There maybe a B symbol between A and C and thus the relative position of the synthesized attribute A.s on the stack is not known to compute C.i.

Inherited Attributes

- Insert a Marker Symbol just before C

Production	Semantic Rule
S → aAC	C.i =A.s
S → bAB*M*C	*M*.i =A.s; C.i = *M*.s
C → c	C.s =func(C.i)
M → ε	*M*.s =*M*.i



Replacing Inherited with Synthesized Attributes

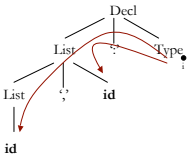
- Inherited Attributes are hard to handle
- Alternative: Modify Grammar (if possible)
- Example:
  - List of Declaration in Pascal.
  - Type as an inherited attribute
  - Change Grammar and Type is Synthesized

Decl → List ':' Type
Type → **integer** | **real**
List → List ',' **id** | **id**

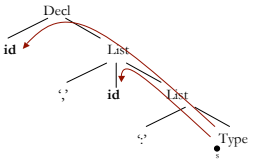
Decl → **id** List
List → ',' **id** List | ':' Type
Type → **integer** | **real**

Replacing Inherited with Synthesized Attributes

Decl → List ':' Type
Type → **integer** | **real**
List → List ',' **id** | **id**



Decl → **id** List
List → ',' **id** List | ':' Type
Type → **integer** | **real**

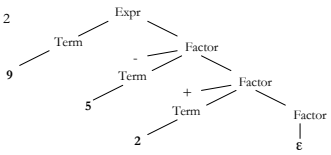


Example

- What Does This Scheme Do?

```
Expr    → Term Factor1
Factor0 → addOp Term { print(addOp.lexeme)} Factor1
        | ε
Term    → num { print(num.val)}
```

Input: 9 - 5 + 2

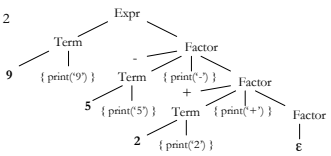


Example

- What Does This Scheme Do?

```
Expr    → Term Factor1
Factor0 → addOp Term { print(addOp.lexeme)} Factor1
        | ε
Term    → num { print(num.val)}
```

Input: 9 - 5 + 2

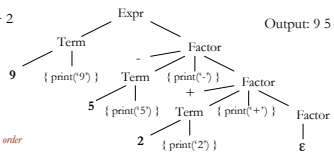


Example

- What Does This Scheme Do?

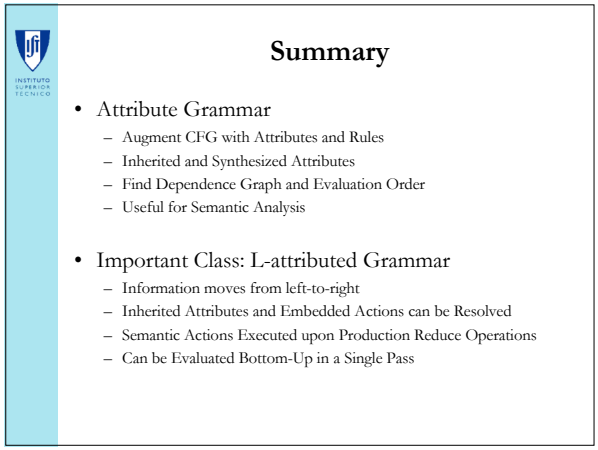
```
Expr    → Term Factor1
Factor0 → addOp Term { print(addOp.lexeme)} Factor1
        | ε
Term    → num { print(num.val)}
```

Input: 9 - 5 + 2



Output: 9 5 - 2 +

depth-first evaluation order



Summary

- Attribute Grammar
  - Augment CFG with Attributes and Rules
  - Inherited and Synthesized Attributes
  - Find Dependence Graph and Evaluation Order
  - Useful for Semantic Analysis
- Important Class: L-attributed Grammar
  - Information moves from left-to-right
  - Inherited Attributes and Embedded Actions can be Resolved
  - Semantic Actions Executed upon Production Reduce Operations
  - Can be Evaluated Bottom-Up in a Single Pass