Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Attribute Grammars

Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of *semantic rules*
- Each symbol in the derivation has a set of values, or *attributes*
- The *semantic rules* specify how to compute a value for each attribute

Example grammar:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T \,/\, F \\
 & | & F \\
F & \rightarrow & \textbf{num}
\end{array}
$$

We want to write an expression interpreter

One way to do this is to augment the expression grammar with semantic rules that compute the value of each valid expression

Expression Interpreter

Productions			Semantic Rules
S	$\rightarrow$	E	$S.val \leftarrow E.val$
E_0	$\rightarrow$	$E_1 + T$	$E_0.val \leftarrow E_1.val + T.val$
	|	$E_1 - T$	$E_0.val \leftarrow E_1.val - T.val$
	|	T	$E_0.val \leftarrow T.val$
T_0	$\rightarrow$	$T_1 * F$	$T_0.val \leftarrow T_1.val * F.val$
	|	T_1 / F	$T_0.val \leftarrow T_1.val / F.val$
	|	F	$T_0.val \leftarrow F.val$
F	$\rightarrow$	**num**	$F.val \leftarrow$ **num**.*val*

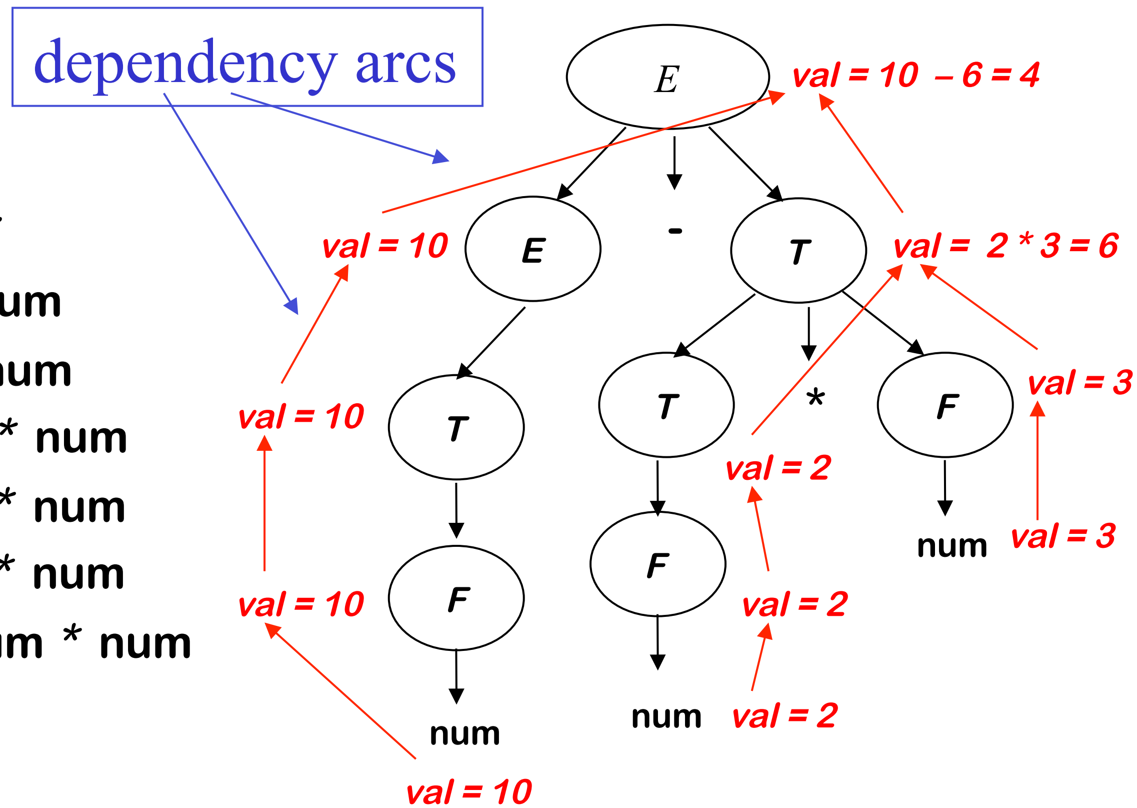
Notice that:

- Semantic rules use context information
- In this attribute grammar, attributes of grammar symbols on the lhs are computed using the attributes of grammar symbols on the rhs (such attributes are called *synthesized attributes*)
- The token **num** has a value attribute returned by the lexer

To Evaluate an Expression

For "10 – 2 * 3"

$$
\begin{aligned}
S &\Rightarrow E \\
&\Rightarrow E - T \\
&\Rightarrow E - T * F \\
&\Rightarrow E - T * \text{num} \\
&\Rightarrow E - F * \text{num} \\
&\Rightarrow E - \text{num} * \text{num} \\
&\Rightarrow T - \text{num} * \text{num} \\
&\Rightarrow F - \text{num} * \text{num} \\
&\Rightarrow \text{num} - \text{num} * \text{num}
\end{aligned}
$$



Annotated parse tree
(values of the attributes are evaluated)

Attribute Grammars

- Attributes are associated with nodes in parse tree (terminals and non-terminals)
- Productions are associated with semantic rules which define how to assign values to attributes
- An attribute is defined (computed) once, using local information
- Identical terms in a production are labeled for uniqueness
  - $E \rightarrow E + T$ becomes $E_0 \rightarrow E_1 + T$
- Rules and parse tree define an attribute dependence graph
  - Dependence graph must be non-circular, otherwise it has no meaning

This produces a high-level, functional specification (no side-effects)

Synthesized attribute
  - Depends on values from children

Inherited attribute
  - Depends on values from siblings and parent

Attribute Grammars

$A \rightarrow X_1 X_2 \ldots X_n$

- **Synthesized attributes**: An attribute of A computed using attributes of X_1 , X_2 , ... , X_n
  - Example: $E_0 \rightarrow E_1 + T$ $\{ E_0.val \leftarrow E_1.val + T.val \}$

    production semantic rule

- **Inherited attributes**: An attribute of a symbol on the rhs computed using attributes of A , X_1 , X_2 , ... , X_n
  - Example: $Decl \rightarrow Type\ L\ ;$ $\{ L.type \leftarrow Type.type \}$

    production semantic rule

  - Example: $L_0 \rightarrow L_1 , id$ $\{ L_1.type \leftarrow L_0.type \}$

    production semantic rule

Another Example Grammar

Number	→	***Sign List***
Sign	→	**+**
	|	**-**
List	→	***List Bit***
	|	***Bit***
Bit	→	**0**
	|	**1**

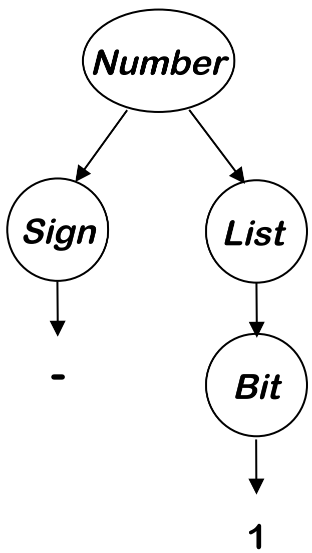
This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

Example Parse Trees

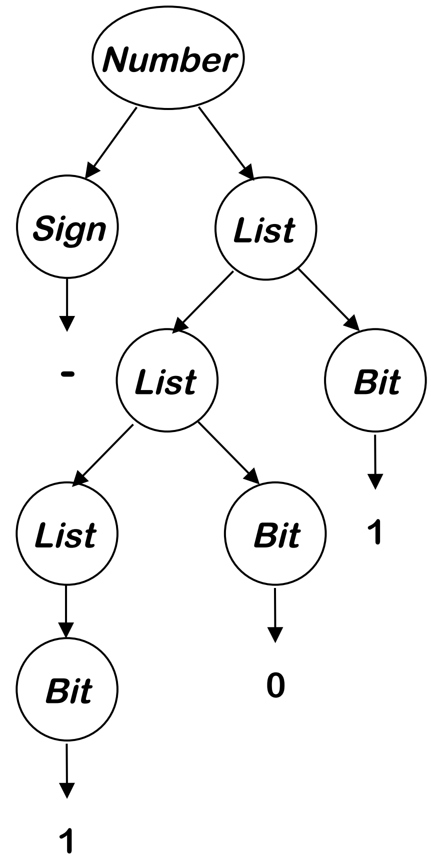
For "-1"

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*



Attribute Grammar

Add rules to compute the decimal value of a signed binary number

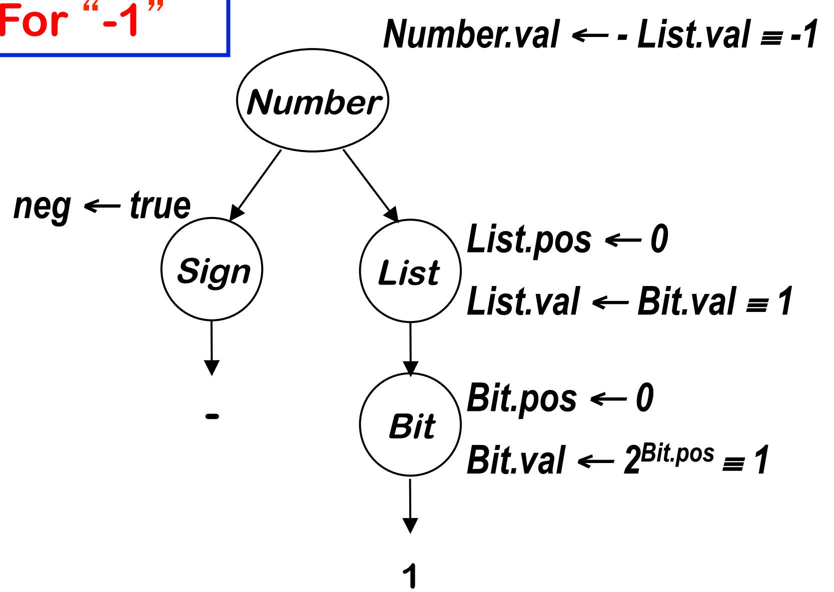
Productions			Attribute Rules
Number	→	*Sign List*	***List.pos*** $\leftarrow$ ***0*** ***If Sign.neg*** ***then Number.val*** $\leftarrow$ ***- List.val*** ***else Number.val*** $\leftarrow$ ***List.val***
Sign	→	+	***Sign.neg*** $\leftarrow$ ***false***
	|	-	***Sign.neg*** $\leftarrow$ ***true***
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	***Bit.pos*** $\leftarrow$ ***List.pos*** ***List.val*** $\leftarrow$ ***Bit.val***
Bit	→	0	***Bit.val*** $\leftarrow$ ***0***
	|	1	$Bit.val \leftarrow 2^{Bit.ps}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Rules and parse tree together imply an attribute dependence graph

Example

For "-1"



One possible evaluation order:

1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

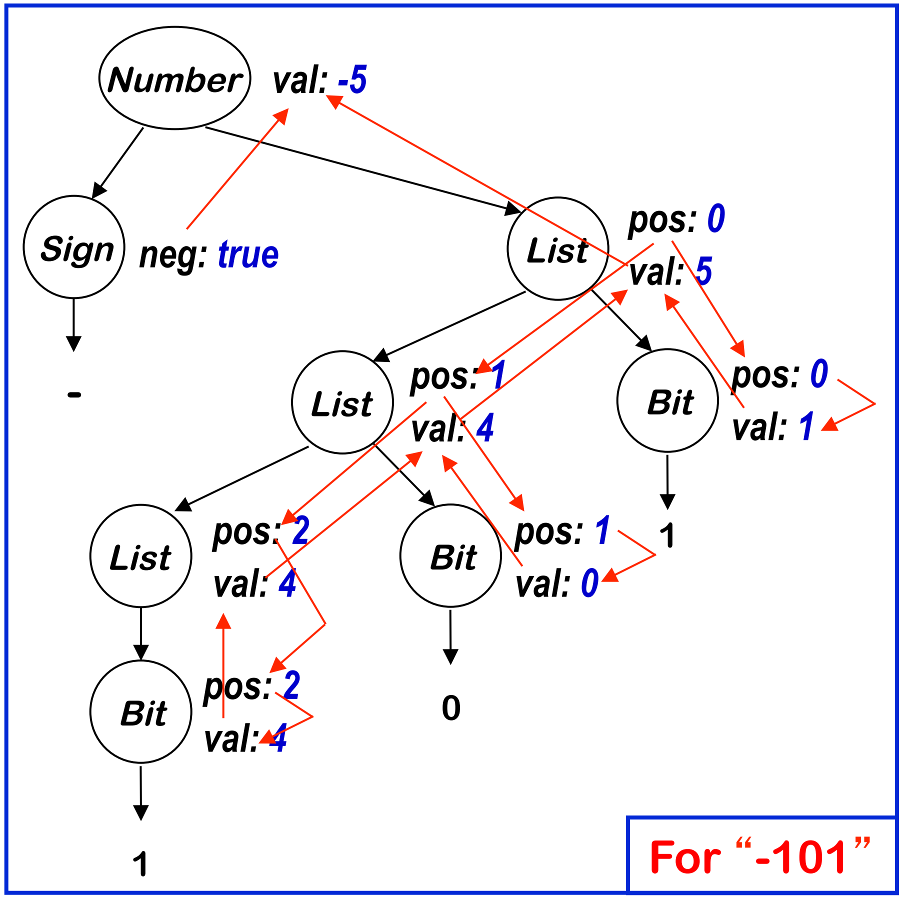
Other orders are possible

Evaluation order for attributes:

- Independent attributes first
- Others in order as input values become available

Evaluation order must be consistent with the attribute dependence graph

Example: Dependency Graph For Attributes



This is the complete attribute dependence graph for "-101".

It shows the flow of *all* attribute values in the example.

Some flow downward (or sideways)

→ inherited attributes

Some flow upward

→ synthesized attributes

A rule may use attributes in the parent, children, or siblings of a node

Evaluation Methods

Dynamic, dependence-based methods

- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Compute the attributes in topological order

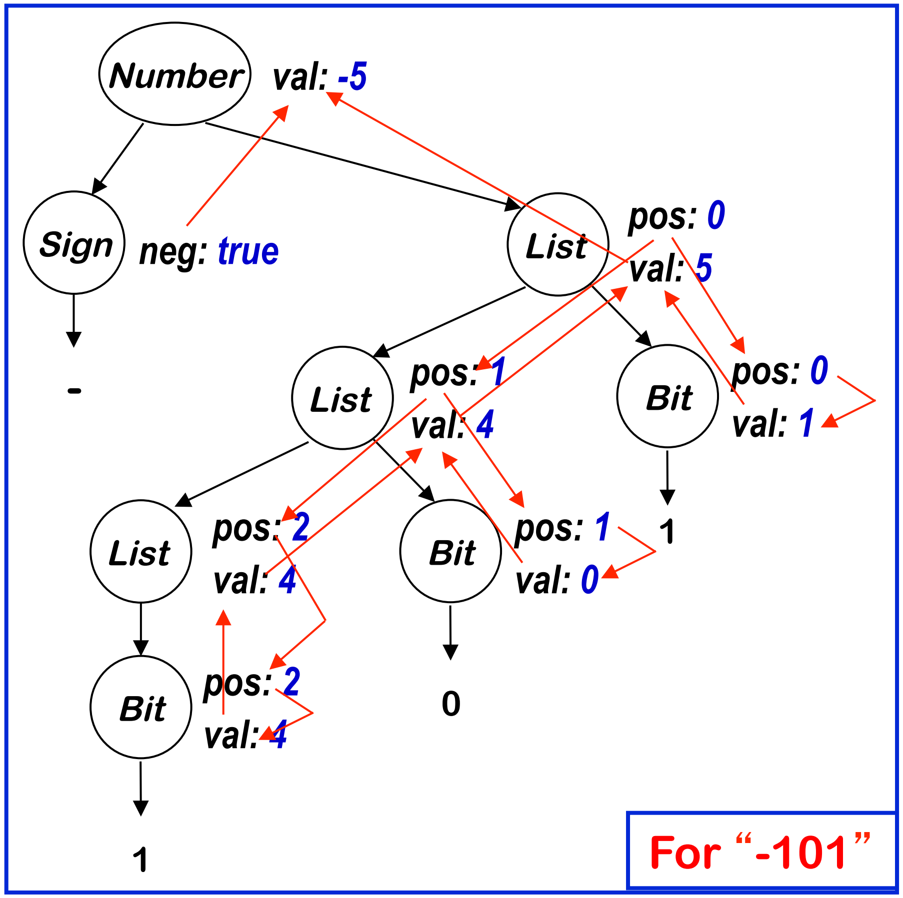
Rule-based methods

- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

Oblivious methods

- Ignore rules and parse tree
- Pick a convenient order (at design time) and use it
- This is what JavaCUP and Yacc do

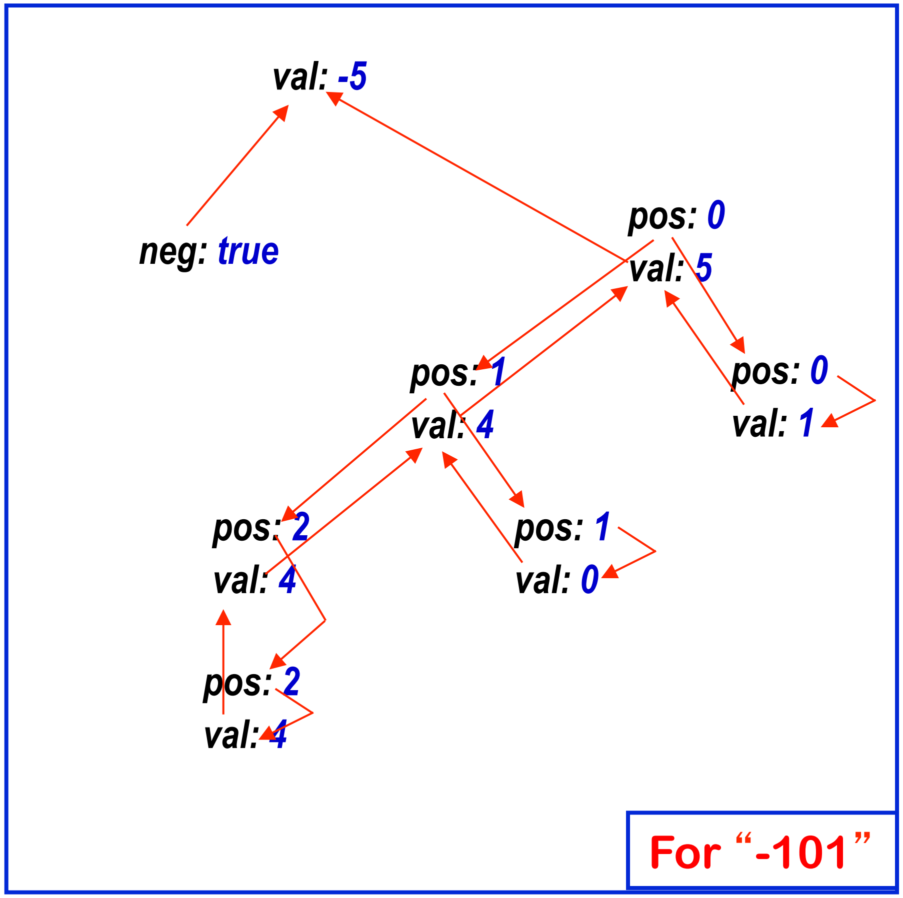
Back to the Example



If we show the computation ...

and then peel away the parse tree ...

Back to the Example



All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The dynamic methods start with the independent values, and then follow the graph edges.

The rule-based methods try to discover "good" orders by analyzing the rules.

The oblivious methods ignore the structure of this graph.

The dependence graph must be acyclic

S-Attributed Grammars

- A grammar that uses only synthesized attributes is called an:

  S-attributed grammar

  – S-attributed grammars can be evaluated in a single bottom-up pass

- LR parsers can easily deal with S-attributed grammars
  – Store the attributes of the symbols in the parser stack
  – When a reduce action is taken
    - Symbols in the rhs of the production and their attributes are already in the stack
    - Compute the synthesized attributes of the symbol in the lhs of the production using the attributes of the symbols on the rhs

Synthesized Attributes on the Parser Stack

Production	Semantic Rule
$E_0 \rightarrow E_1 + T$	$E_0.val \leftarrow E_1.val + T.val$

top of the parser stack

T	$T.val$
$+$	
E_1	$E_1.val$

after the reduction

E_0	$E_0.val$

top of the parser stack

L-Attributed Grammars

- If inherited attribute of a symbol is computed using the inherited attributes of its parent and attributes of symbols on its left in the production, then the grammar is called an: *L-attributed grammar*

- Given a symbol X_i on the rhs of production $A \rightarrow X_1 X_2 \ldots X_n$, each inherited attribute of X_i depends only on:
  - Inherited attributes of A
  - The attributes of $X_1, X_2, \ldots, X_{i-1}$ to the left of X_i on the rhs of the production

L-Attributed Grammars

- Attributed grammars can be evaluated using a depth-first traversal of the parse tree:

The nodes in this algorithm are the nodes of the parse tree

Start the depth-first traversal by calling the **dfsvisit** on the root of the parse tree

```
procedure dfsvisit(n: node)
begin
   for each child m of n from left to right do
      evaluate inherited attributes of m;
      dfsvisit(m);
   endfor
   evaluate synthesized attributes of n
end
```

An Extended Attribute-Grammar Example

Grammar for a block of assignments

$Block_0$	$\rightarrow$	$Block_1$ $Assign$
	|	$Assign$
$Assign$	$\rightarrow$	$Ident$ = $Expr$;
$Expr_0$	$\rightarrow$	$Expr_1$ + $Term$
	|	$Expr_1$ – $Term$
	|	$Term$
$Term_0$	$\rightarrow$	$Term_1$ * $Factor$
	|	$Term_1$ / $Factor$
	|	$Factor$
$Factor$	$\rightarrow$	($Expr$)
	|	$Number$
	|	$Identifier$

Estimate execution time

- **Each operation has a COST**
- **Add them, bottom up**
- **Assume a load per value**
- **Assume no reuse**

Can be solved using an attribute grammar

An Extended Example (continued)

$Block_0$	→	*$Block_1$ Assign*	$Block_0$.cost ← $Block_1$.cost + Assign.cost
	|	*Assign*	$Block_0$.cost ← Assign.cost
Assign	→	*Ident = Expr ;*	Assign.cost ← COST(store) + Expr.cost
$Expr_0$	→	*$Expr_1$ + Term*	$Expr_0$.cost ← $Expr_1$.cost + COST(add) + Term.cost
	|	*$Expr_1$ – Term*	$Expr_0$.cost ← $Expr_1$.cost + COST(add) + Term.cost
	|	*Term*	$Expr_0$.cost ← Term.cost
$Term_0$	→	*$Term_1$ * Factor*	$Term_0$.cost ← $Term_1$.cost + COST(mult) + Factor.cost
	|	*$Term_1$ / Factor*	$Term_0$.cost ← $Term_1$.cost + COST(div) + Factor.cost
	|	*Factor*	$Term_0$.cost ← Factor.cost
Factor	→	*(Expr)*	Factor.cost ← Expr.cost
	|	*Number*	Factor.cost ← COST(loadi)
	|	*Identifier*	Factor.cost ← COST(load)

All the attributes are synthesized !

An Extended Example

Properties of the example grammar

- All attributes are synthesized $\Rightarrow$ S-attributed grammar
- Rules can be evaluated bottom-up in a single pass
  – Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement?

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

A Better Execution Model

Adding load tracking

- Need sets *Before* and *After* for each production
- Must be initialized, updated, and passed around the tree

Factor → (Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After
| Number	Factor.cost ← COST(loadi) ; Factor.After ← Factor.Before

| Identifier	If (Identifier.name ∉ Factor.Before)
	then
	Factor.cost ← COST(load);
	Factor.After ← Factor.Before ∪ Identifier.name
	else
	Factor.cost ← 0
	Factor.After ← Factor.Before

A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the “copy rules”
- Every production needs rules to copy *Before* and *After*

A sample production

$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost + \text{COST}(add) + Term.cost\ ;$ $Expr_1.Before \leftarrow Expr_0.Before\ ;$ $Term.Before \leftarrow Expr_1.After;$ $Expr_0.After \leftarrow Term.After$

These copy rules multiply rapidly

Each creates an instance of the set

Lots of work, lots of space, lots of rules to write

An Even Better Model

What about accounting for finite register sets?

- *Before* and *After* must be of limited size
- Adds more complexity to *Factor* → *Identifier*

Jump from tracking loads to tracking registers is small

- Copy rules are already in place
- Some local code to perform the allocation

Tracking loads introduced *Before* and *After* sets and caused significant change in the attribute grammar

Attribute Grammars

- Non-local computation needed lots of supporting rules
- Complex local computation is relatively easy

The Problems

- Copy rules increase complexity
- Copy rules increase space requirements
  - Need copies of attributes
- After we write the attribute grammar, to evaluate the attributes
  - Must build the parse tree
  - Must traverse tree to evaluate the attributes

Addressing the Problem

- Use rules with side effects, store the results in global variables
- For the example, use a table of names (symbol table)
  - Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation and storage headaches
- All inter-assignment attribute flow is through table
  - Clean, efficient implementation
  - Good techniques for implementing the table
  - When its done, information is in the table!
  - Cures most of the problems
- This design violates the functional paradigm

Reworking the Example (with load tracking)

Block₀	→	*Block₁ Assign*	
	|	*Assign*	*cost ← 0;*
Assign	→	*Ident = Expr ;*	*cost ← cost + COST*(store)*;*
Expr₀	→	*Expr₁ + Term*	*cost ← cost + COST*(add)*;*
	|	*Expr₁ – Term*	*cost ← cost + COST*(sub)*;*
	|	*Term*	
Term₀	→	*Term₁ * Factor*	*cost ← cost + COST*(mult)*;*
	|	*Term₁ / Factor*	*cost ← cost + COST*(div)*;*
	|	*Factor*	
Factor	→	*(Expr)*	
	|	Number	*cost ← cost + COST*(loadI)*;*
	|	Identifier	*i ← hash*(Identifier)*;* *if (Table[i].loaded* = false*)* *then* *cost ← cost + COST*(load)*;* *Table[I].loaded ← true;*

Syntax-Directed Translation

Ad-hoc syntax-directed translation

- Associate a snippet of code with each production
- Evaluation method: At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
  - We can easily implement S-attributed grammars with it
  - Gives ability to do tasteless and bad things too

To make this work

- Need names for attributes of each symbol on *lhs* & *rhs*
  - Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, …)
  - Yacc introduced $$, $1, $2, … $n, left to right

    Expr := Expr + Term { $$ = $1 + $3; }
- Evaluation method fits nicely into LR(1) parsing algorithm

Example — Building a Parse Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume Yacc syntax

S	→	*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*	**$$ = $1;**
Factor	→	(*Expr*)	**$$ = $1;**
	|	num	**$$ = MakeNumNode(token);**
	|	id	**$$ = MakeIdNode(token);**

Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses the shortcomings of the attribute grammars
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Typical Uses

- 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 check of expression → bottom-up walk
  - Procedure interfaces are harder
    - Build a representation for parameter list and types

Is This Really “Ad-hoc” ?

Relationship between practice and attribute grammars

Similarities

- Both associate rules with productions
- Application order determined by tools, not author
- Abstract names for symbols

Differences

- Actions in ad-hoc method are applied as a unit;
  not true for attribute grammar rules
- Anything goes in *ad-hoc* actions;
  attribute grammar rules are functional