Context-sensitive Analysis
or
Semantic Elaboration

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(int a, int b,int c,int d) {
    …
}
fee() {
    int f[3],g[0], h, i, j, k;
    char *p;
    fie(h,i,“ab”,j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”,p,q);
    p = 10;
}
```

What is wrong with this program?

(let me count the ways …)

- number of args to fie()
- declared g[0], used g[17]
- “ab” is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are

“deeper than syntax”

To generate code, we need to understand its meaning !

Beyond Syntax

These are beyond the expressive power of a CFG

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does a given use reference?
- Is the expression "x * y + z" type-consistent?
- In "a[i,j,k]", does a have three dimensions?
- Where can "z" be stored? (register, local, global, heap, static)
- In "f ← 15", how should 15 be represented?
- How many arguments does "fie()" take? What about "printf ()" ?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?
- Is "x" defined before it is used?

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars?
  - Attribute grammars
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code (action routines)

In context-sensitive analysis, ad-hoc techniques dominate in practice.

Beyond Syntax

Telling the story

- We will study the formalism — an attribute grammar
  - Clarify many issues in a succinct and immediate way
  - Separate analysis problems from their implementations

- We will see that the problems with attribute grammars motivate actual, ad-hoc practice
  - Non-local computation
  - Need for centralised information

We will cover attribute grammars, then move on to ad-hoc ideas

When?

- These kind of analyses are either performed together with parsing or in a post-pass that traverses the IR produced by the parser

Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of rules computing values
- Each symbol in the derivation (or parse tree) has a set of named values, or attributes
- The rules specify how to compute a value for each attribute

  - Attribution rules are functional; they uniquely define the value
  - Each attribute is defined by rules that can refer to the values of all the other attributes in the production (local information)

Example

1	*Number*	→	*Sign List*
2	*Sign*	→	+
3		|	-
4	*List*	→	*List Bit*
5		|	*Bit*
6	*Bit*	→	0
7		|	1

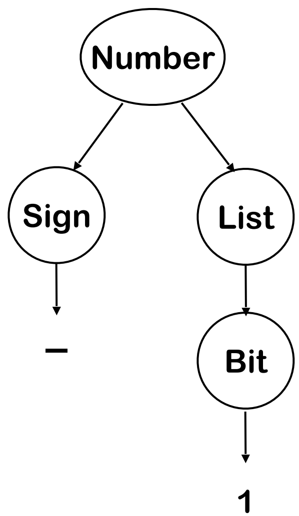
This grammar describes signed binary numbers

e.g., -10010 or +00101

Examples

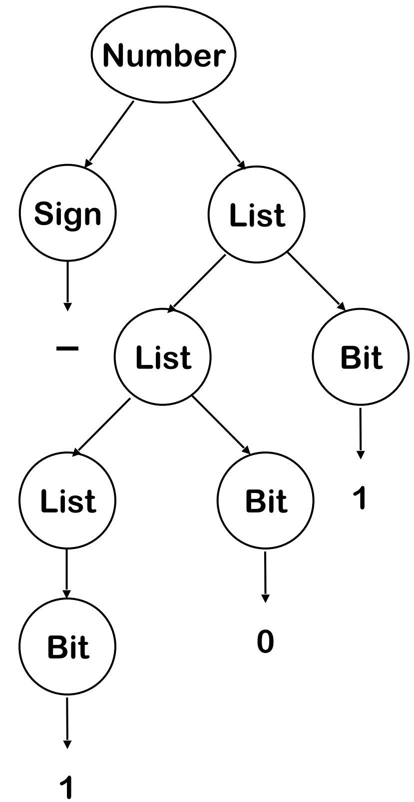
For "–1"

Number → Sign List
→ Sign Bit
→ Sign 1
→ – 1



For "–101"

Number → Sign List
→ Sign List Bit
→ Sign List 1
→ Sign List Bit 1
→ Sign List 0 1
→ Sign Bit 0 1
→ Sign 1 0 1
→ – 101



We will use these two examples throughout the lecture

Attribute Grammars

1	*Number*	→	*Sign List*
2	*Sign*	→	+
3		|	-
4	*List*	→	*List Bit*
5		|	*Bit*
6	*Bit*	→	0
7		|	1

- We would like to augment it with rules that compute the decimal value of each valid input string: e.g. -10010 -> -18 +00101 -> +5

- For this we consider the following attributes

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

Attribute Grammars

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

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

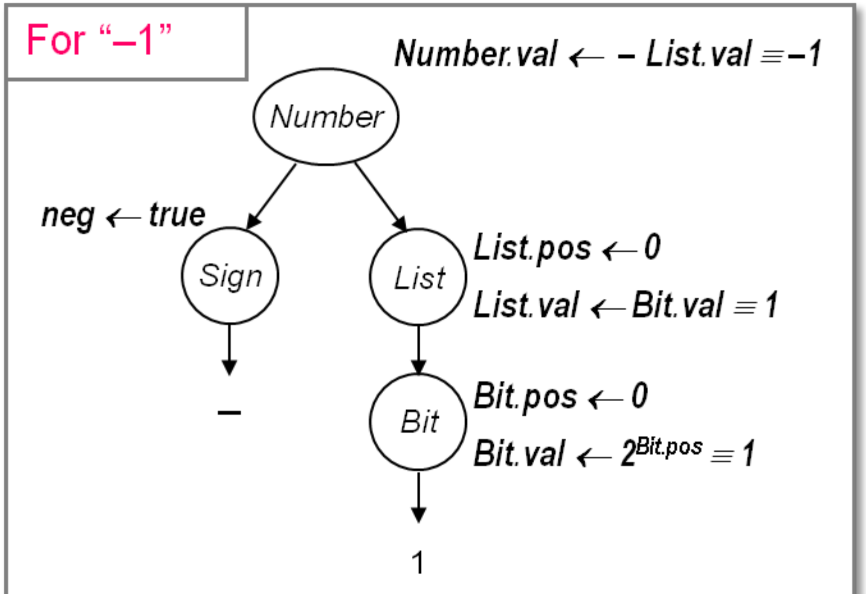
Productions			Attribution Rules
Number	$\rightarrow$	*Sign List*	$List.pos \leftarrow 0$ *if Sign.neg* *then* $Number.val \leftarrow -\ List.val$ *else* $Number.val \leftarrow List.val$
Sign	$\rightarrow$	+	$Sign.neg \leftarrow false$
	|	-	$Sign.neg \leftarrow true$
$List_0$	$\rightarrow$	$List_1$ *Bit*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

Note: for some rules the information flows from left to right

for some rules the information flows from right to left

Back to the Examples

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

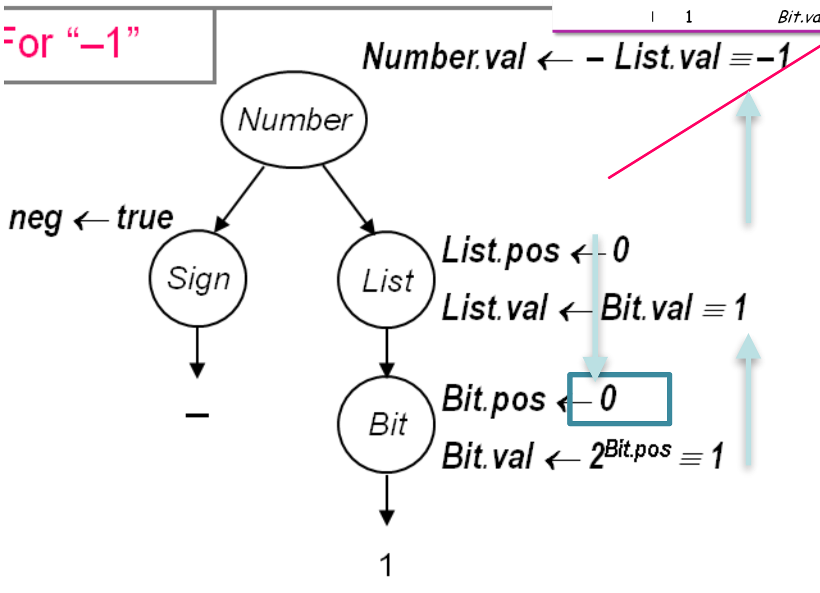


Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← 0
			if Sign.neg
			then Number.val ← - *List*
			else Number.val ← *List.*
Sign	→	+	*Sign.neg* ← *false*
	|	-	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$
			$Bit.pos \leftarrow List_0.pos$
			$List_0.val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

Evaluation order

Productions			Attribution Rules
Number	→	Sign List	List.pos ← 0 if Sign.neg then Number.val ← - List.val else Number.val ← List.val
Sign	→	+	Sign.neg ← false
	|	-	Sign.neg ← true
$List_0$	→	$List_1$ Bit	$List_1$.pos ← $List_0$.pos + 1 Bit.pos ← $List_0$.pos $List_0$.val ← $List_1$.val + Bit.val
	|	Bit	Bit.pos ← List.pos List.val ← Bit.val
Bit	→	0	Bit.val ← 0
	|	1	Bit.val ← $2^{Bit.pos}$

Rules + parse tree imply an attribute dependence graph



One possible evaluation order:

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

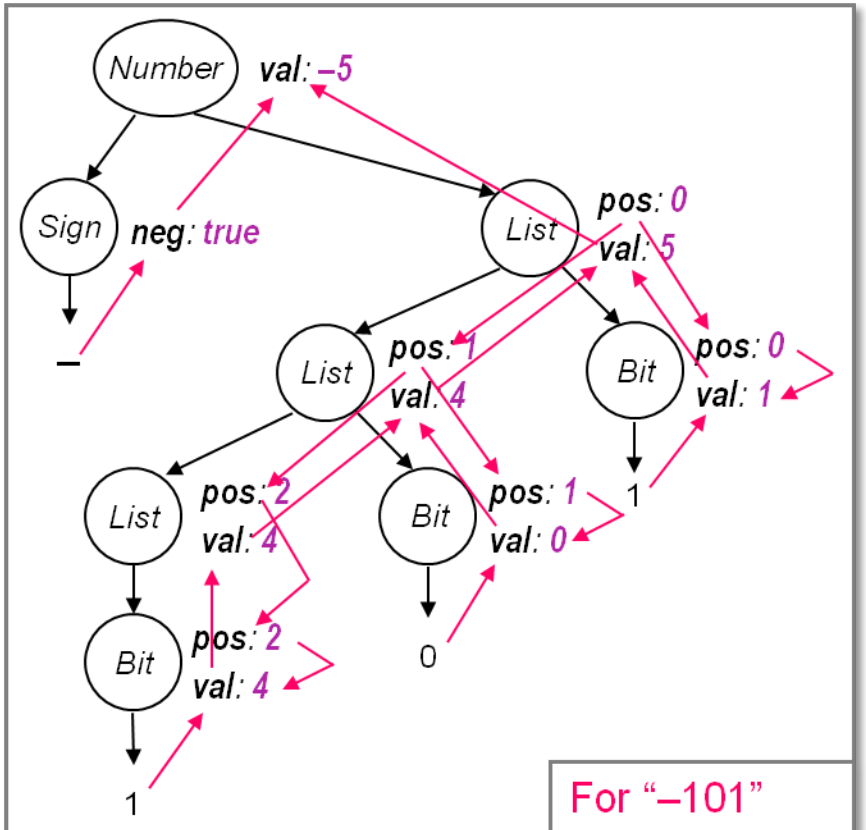
Other orders are possible

Evaluation order must be consistent with the attribute dependence graph

Knuth suggested a data-flow model for evaluation

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

Back to the Examples



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

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

Some flow downward

→ inherited attributes

Some flow upward

→ synthesized attributes

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

The Rules of the Game

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Rules & parse tree define an attribute dependence graph
  - Graph must be non-circular

This produces a high-level, functional specification

We need a attributed grammar evaluator

N.B.: *AG* is a specification for the computation, not an algorithm

Using Attribute Grammars

Attribute grammars can specify context-sensitive actions

- Take values from syntax
- Perform computations with values
- Insert tests, logic, …

Synthesized Attributes

- Use values from children & from constants
- S-attributed grammars
- Evaluate in a single bottom-up pass

Good match to LR parsing

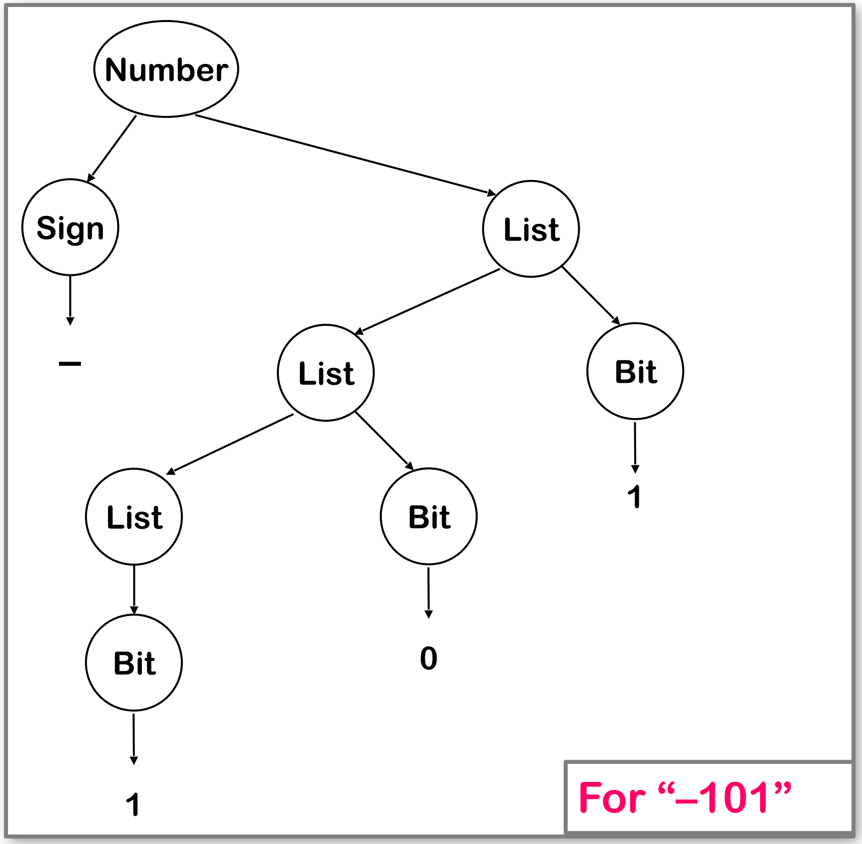
Inherited Attributes

- Use values from parent, constants, & siblings
- Thought to be more natural

Not easily done at parse time

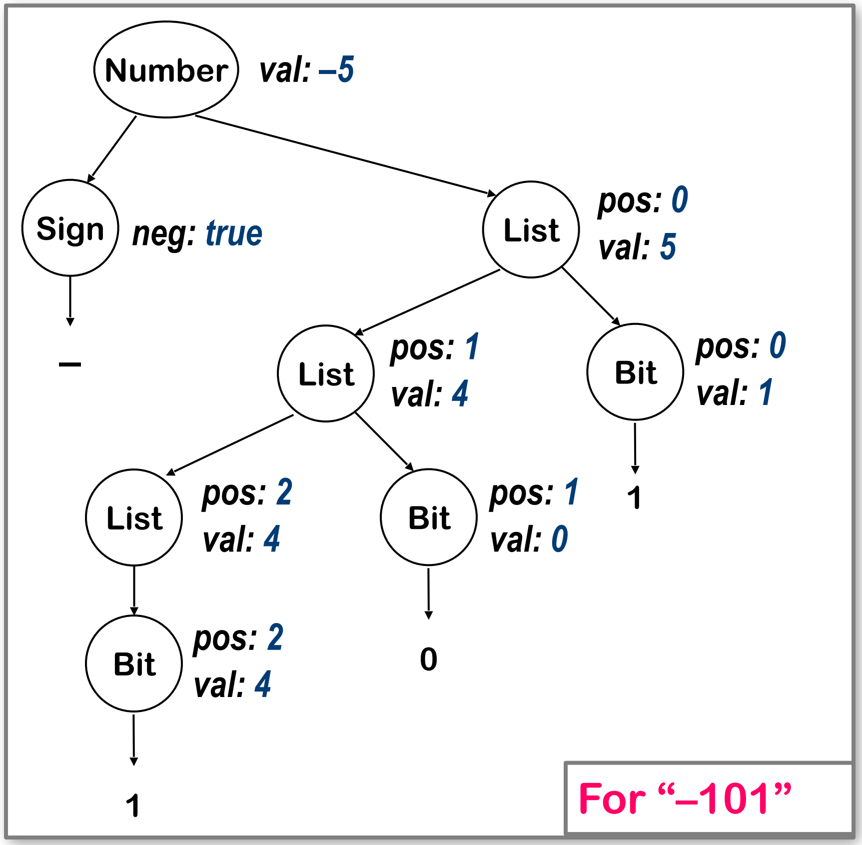
We want to use both kinds of attributes

Back to the Example



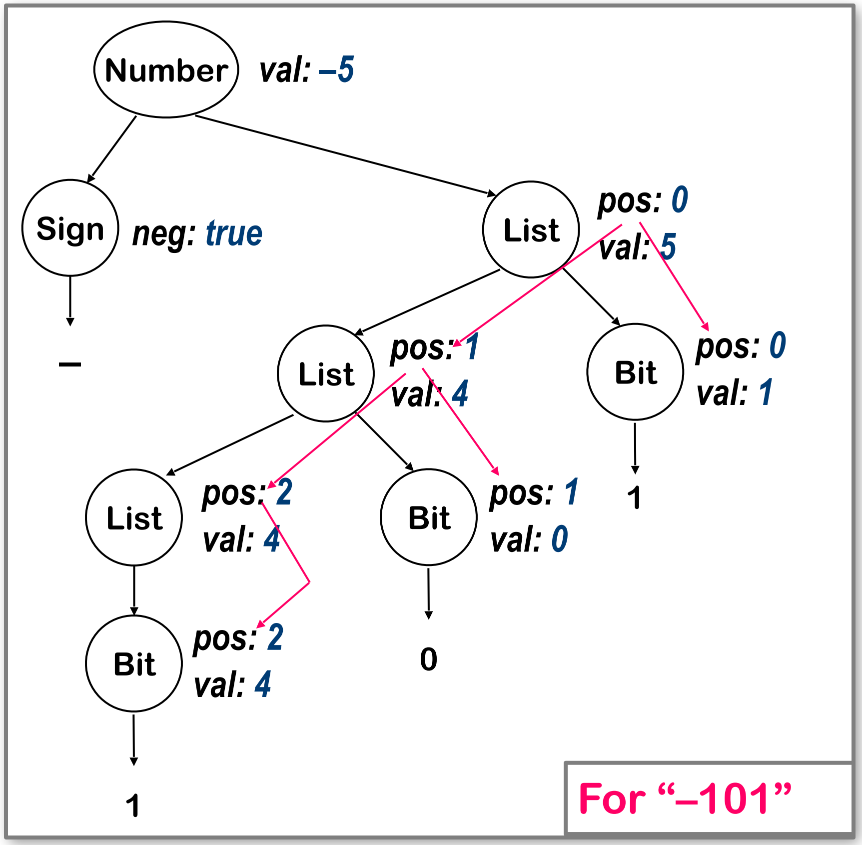
Syntax Tree

Back to the Example



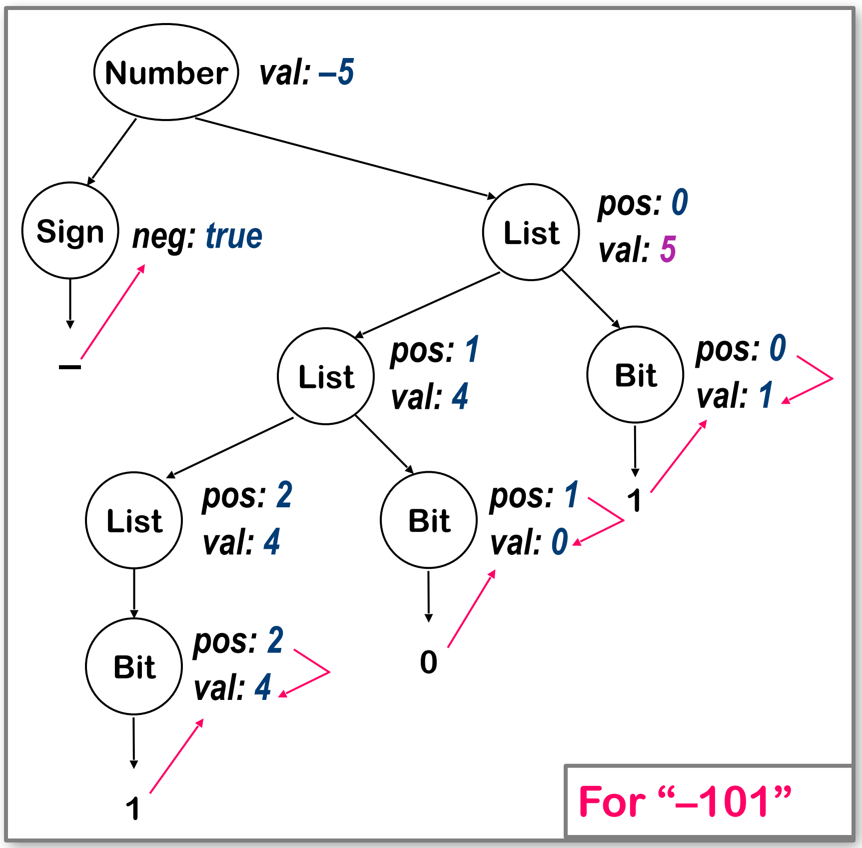
Attributed Syntax Tree

Back to the Example



Inherited Attributes

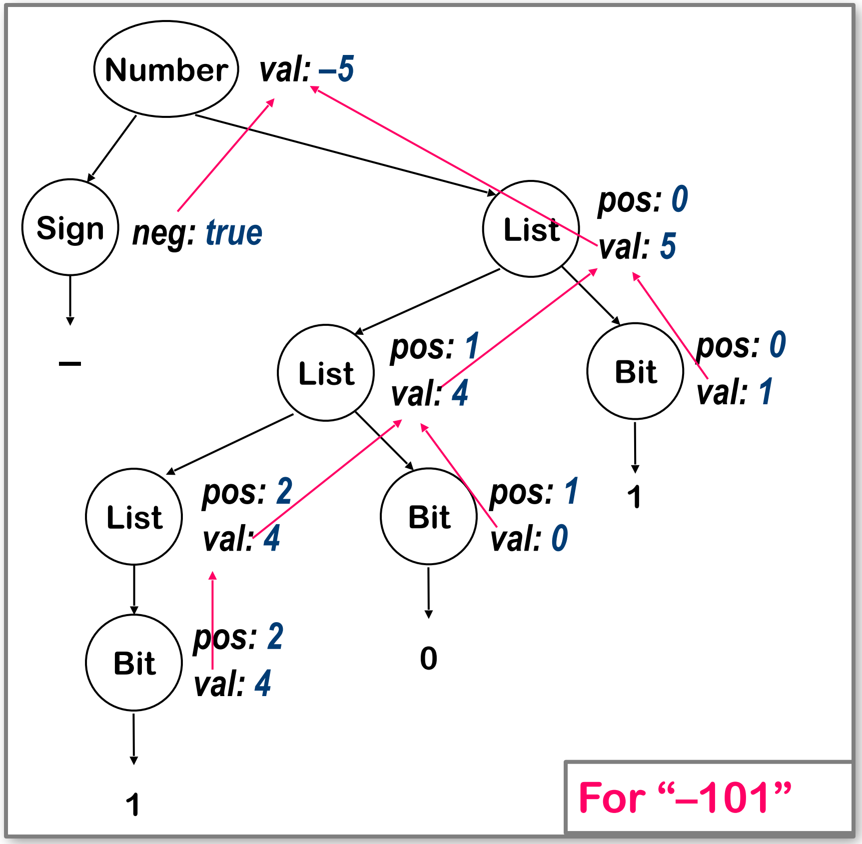
Back to the Example



Synthesized attributes

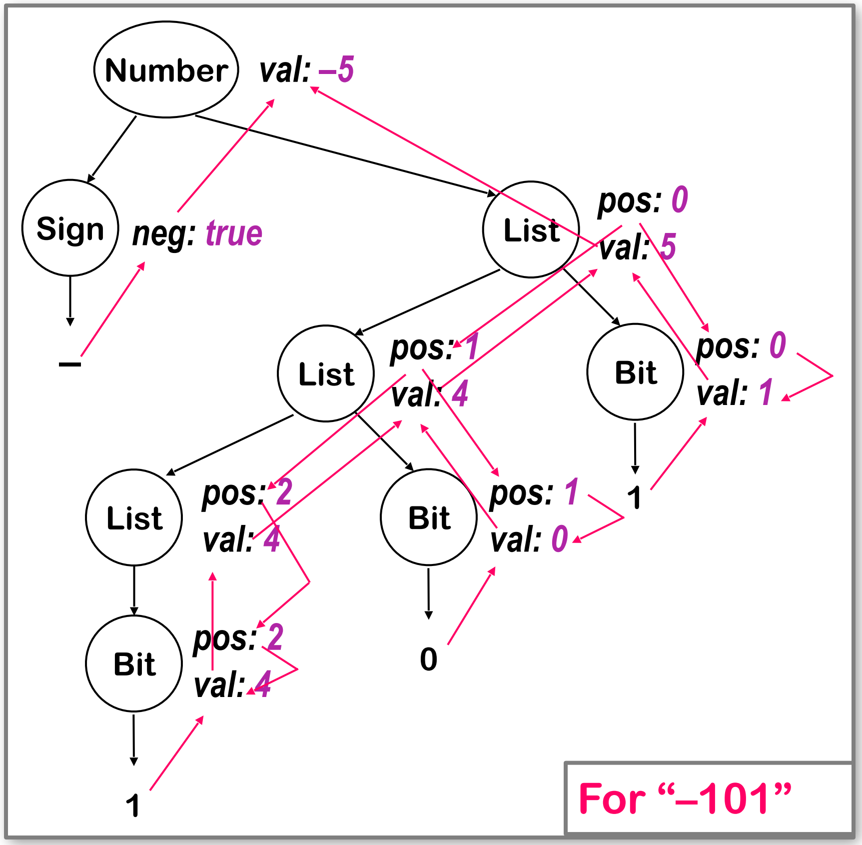
Val draws from children & the same node.

Back to the Example



More Synthesized attributes

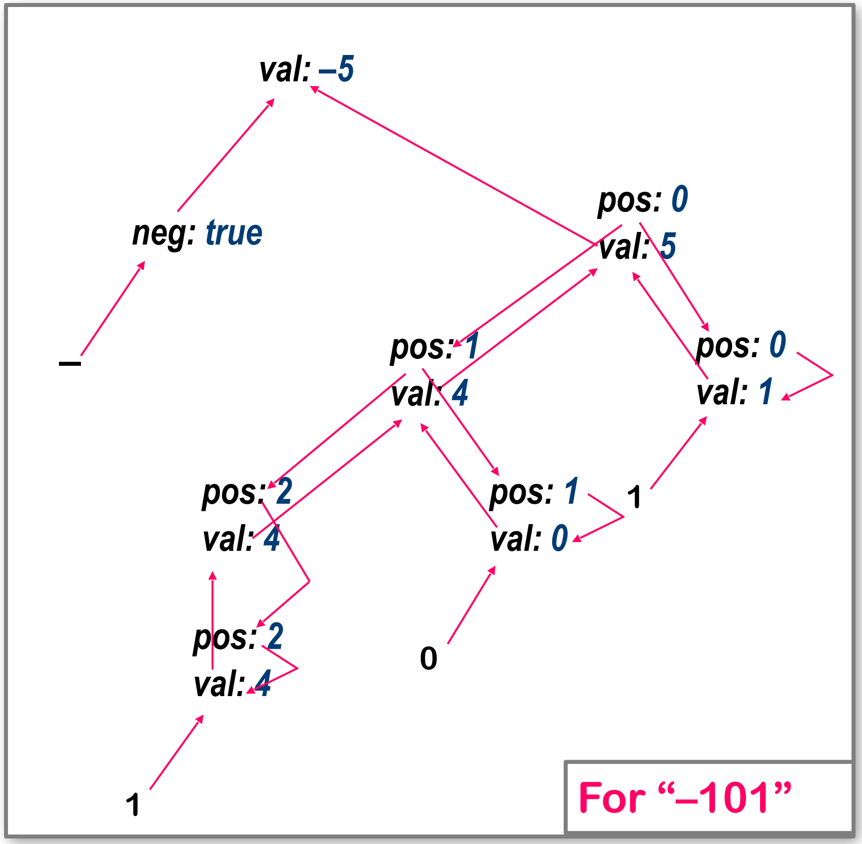
Back to the Example



If we show the computation ...

& 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 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.

The dependence graph must be acyclic

Circularity

We can only evaluate acyclic instances

- General circularity testing problem is inherently exponential!
- We can prove that some grammars can only generate instances with acyclic dependence graphs
  - Largest such class is "strongly non-circular" grammars (SNC)
  - SNC grammars can be tested in polynomial time
  - Failing the SNC test is <u>not</u> conclusive (sufficient conditions)
  - Many evaluation methods discover circularity dynamically

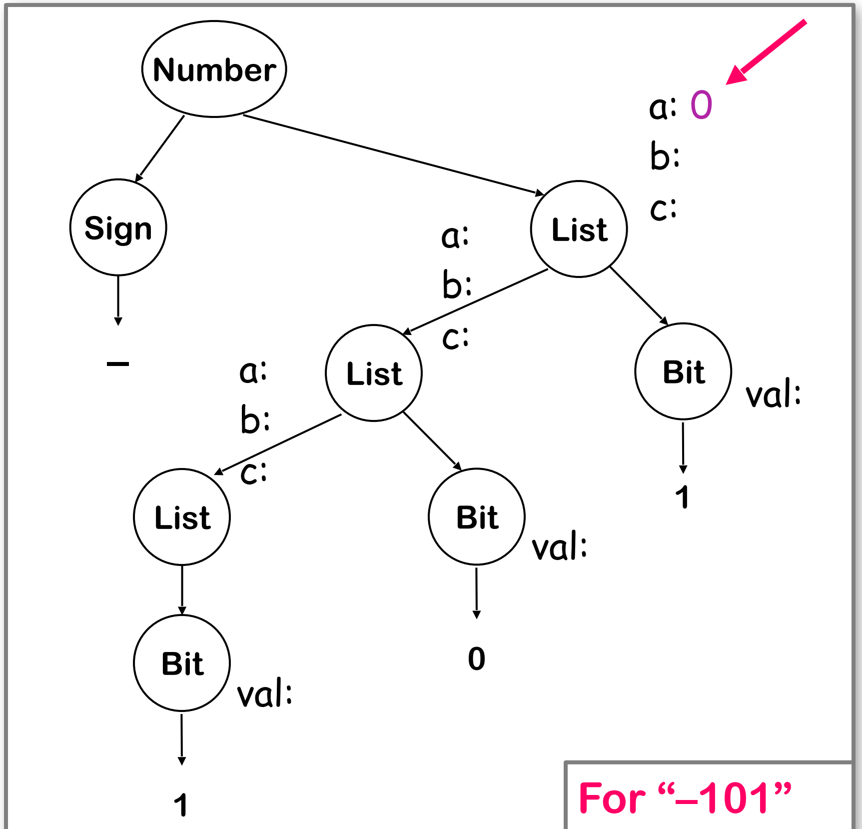
$\Rightarrow$ Bad property for a compiler to have

A Circular Attribute Grammar

Productions			Attribution Rules
Number	$\rightarrow$	List	$List.a \leftarrow 0$
$List_0$	$\rightarrow$	$List_1$ Bit	$List_1.a \leftarrow List_0.a + 1$
			$List_0.b \leftarrow List_1.b$
			$List_1.c \leftarrow List_1.b + Bit.val$
	|	Bit	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	$\rightarrow$	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 1$

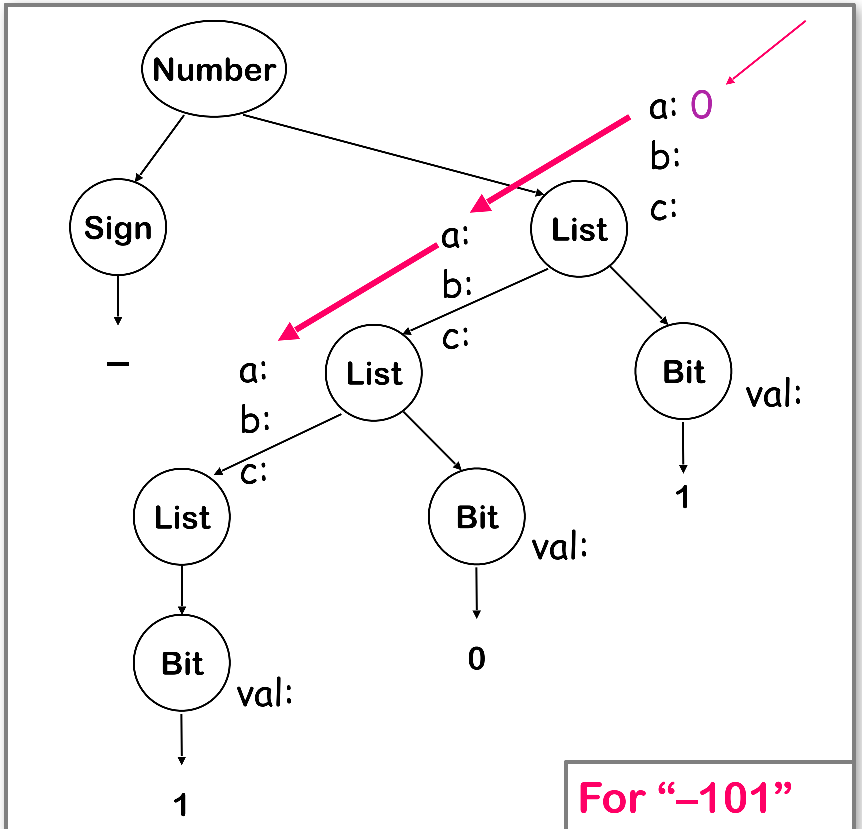
Remember, the circularity is in the attribution rules, not the underlying CFG

Circular Grammar Example



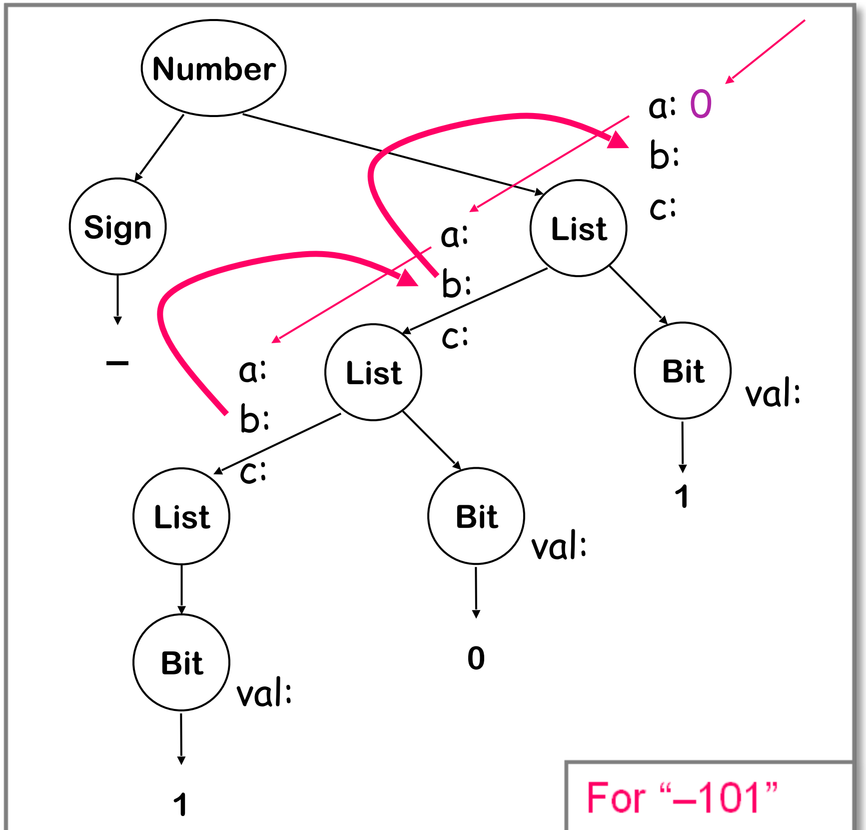
Productions			Attribution Rules
Number	→	*List*	$List.a \leftarrow 0$
$List_0$	→	$List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	→	*0*	$Bit.val \leftarrow 0$
	|	*1*	$Bit.val \leftarrow 1$

Circular Grammar Example



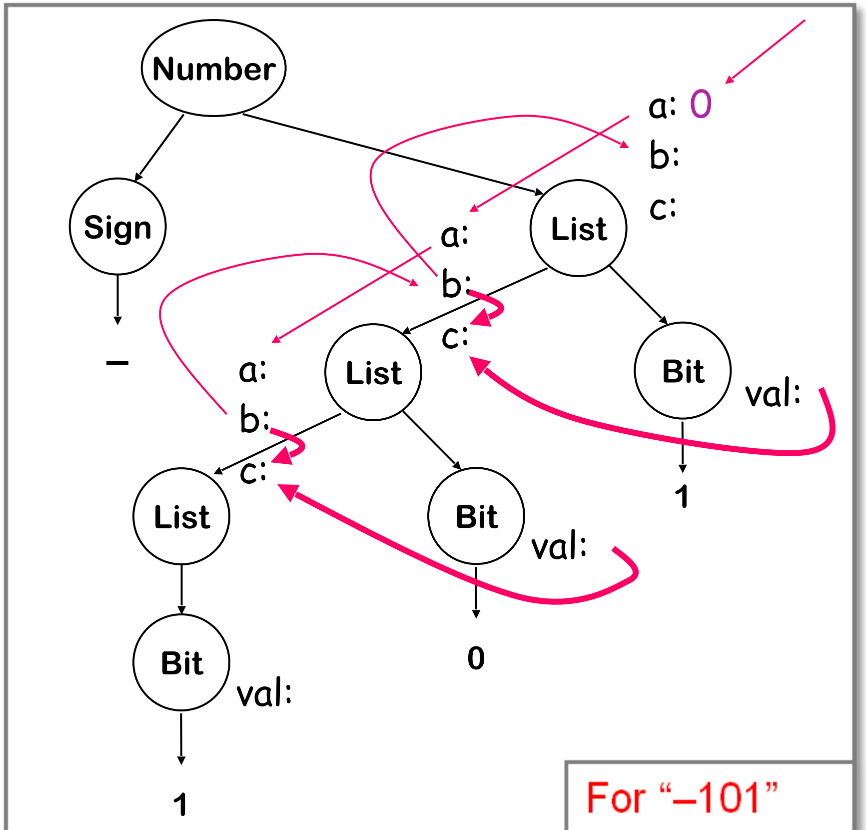
Productions			Attribution Rules
Number	→	*List*	$List.a \leftarrow 0$
$List_0$	→	$List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	→	*0*	$Bit.val \leftarrow 0$
	|	*1*	$Bit.val \leftarrow 1$

Circular Grammar Example



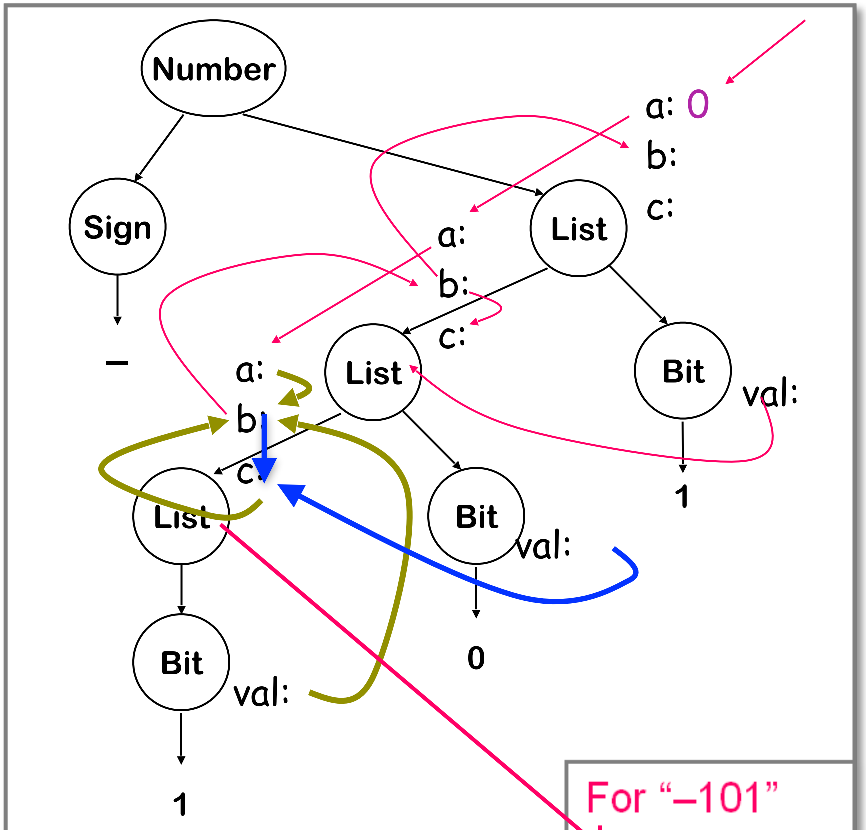
Productions			Attribution Rules
Number	→	*List*	$List.a \leftarrow 0$
$List_0$	→	$List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$
			$List_0.b \leftarrow List_1.b$
			$List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	→	*0*	$Bit.val \leftarrow 0$
	|	*1*	$Bit.val \leftarrow 1$

Circular Grammar Example



Productions			Attribution Rules
Number	→	*List*	*List.a* ← 0
List$_0$	→	*List*$_1$ *Bit*	*List*$_1$.*a* ← *List*$_0$.*a* + 1 *List*$_0$.*b* ← *List*$_1$.*b* *List*$_1$.*c* ← *List*$_1$.*b* + *Bit.val*
	|	*Bit*	*List*$_0$.*b* ← *List*$_0$.*a* + *List*$_0$.*c* + *Bit.val*
Bit	→	*0*	*Bit.val* ← 0
	|	*1*	*Bit.val* ← 1

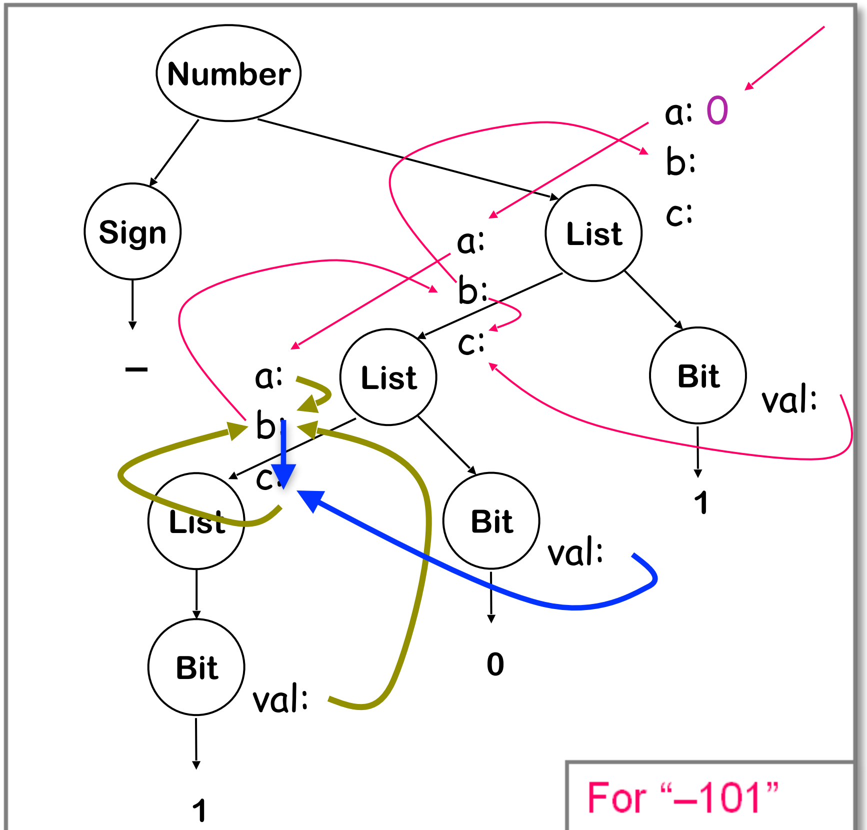
Circular Grammar Example



Productions			Attribution Rules
Number	$\rightarrow$	*List*	$List.a \leftarrow 0$
$List_0$	$\rightarrow$	$List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	$\rightarrow$	*0*	$Bit.val \leftarrow 0$
	|	*1*	$Bit.val \leftarrow 1$

Here is the circularity …

Circular Grammar Example



Productions			Attribution Rules
Number	→	*List*	$List.a \leftarrow 0$
$List_0$	→	$List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	→	*0*	$Bit.val \leftarrow 0$
	|	*1*	$Bit.val \leftarrow 1$

Here is the circularity …

Circularity — The Point

- Circular grammars have indeterminate values
  - Algorithmic evaluators will fail
- Noncircular grammars evaluate to a unique set of values

⇒ Should (undoubtedly) use provably noncircular grammars

Remember, we are studying AGs to gain insight

- We should avoid circular, indeterminate computations
- If we stick to provably noncircular schemes, evaluation should be easier

Another Example on Attribute Grammar

Grammar for a basic block

1	$Block_0$	$\rightarrow$	$Block_1$ Assign
2		|	Assign
3	$Assign_0$	$\rightarrow$	Ident = Expr ;
4	$Expr_0$	$\rightarrow$	$Expr_1$ + Term
5		|	$Expr_1$ - Term
6		|	Term
7	$Term_0$	$\rightarrow$	$Term_1$ * Factor
8		|	$Term_1$ / Factor
9		|	Factor
10	Factor	$\rightarrow$	(Expr)
11		|	Number
12		|	Ident

Let's estimate cycle counts

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

Simple problem for an AG

Hey, that is a practical application!

An Extended Example (continued)

1	$Block_0$	$\rightarrow$	$Block_1$ $Assign$	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
2		|	$Assign$	$Block_0.cost \leftarrow Assign.cost$
3	$Assign_0$	$\rightarrow$	$Ident = Expr\ ;$	$Assign.cost \leftarrow$ COST(store) + $Expr.cost$
4	$Expr_0$	$\rightarrow$	$Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost$ + COST(add) + $Term.cost$
5		|	$Expr_1 - Term$	$Expr_0.cost \leftarrow Expr_1.cost$ + COST(sub) + $Term.cost$
6		|	$Term$	$Expr_0.cost \leftarrow Term.cost$
7	$Term_0$	$\rightarrow$	$Term_1 * Factor$	$Term_0.cost \leftarrow Term_1.cost$ + COST(mult) + $Factor.cost$
8		|	$Term_1 / Factor$	$Term_0.cost \leftarrow Term_1.cost$ + COST(div) + $Factor.cost$
9		|	$Factor$	$Term_0.cost \leftarrow Factor.cost$
10	$Factor$	$\rightarrow$	$(\ Expr\ )$	$Factor.cost \leftarrow Expr.cost$
11		|	$Number$	$Factor.cost \leftarrow$ COST(loadI)
12		|	$Ident$	$Factor.cost \leftarrow$ COST(load)

These are all synthesized attributes !

Values flow from rhs to lhs in prod'ns

An Extended Example (continued)

Properties of the example grammar

- All attributes are synthesized ⇒ 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? x=y+y

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

An Extended Example

- We would like something like

```
if ( name has not been loaded )
    then Factor.cost ← Cost(load);
    else Factor.cost ← 0;
```

Non local information!

- to realize it we consider two attributes before and after that contains set of names
    - before contains the set of all names that occur earlier in the block
    - after contain all names in before plus any name that was loaded in the subtree rooted at that node

A Better Execution Model

Adding load tracking

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

10	*Factor*	→	(*Expr*)	*Factor.cost* ← *Expr.cost* *Expr.before* ← *Factor.before* *Factor.after* ← *Expr.after*
11		|	*Number*	*Factor.cost* ← COST(loadI) *Factor.after* ← *Factor.before*
12		|	*Ident*	*If (Ident.name* ∉ *Factor.before)* *then* *Factor.cost* ← COST(load) *Factor.after* ← *Factor.before* ∪ { *Ident.name* } *else* *Factor.cost* ← 0 *Factor.after* ← *Factor.before*

This version is much more complex

A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the "copy rules"
- Every production needs rules to copy Before & After

A sample production

4	$Expr_0$	$\rightarrow Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost +$ COST(add) $+ Term.cost$ $Expr_1.before \leftarrow Expr_0.before$ $Term.before \leftarrow Expr_1.before$ $Expr_0.after \leftarrow Term.after$

These copy rules multiply rapidly

Each creates an instance of the set

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

A second example: inferring expression types

- Any compiler that tries to generate efficient code for a typed language must confront the problem of inferring types for every expression in the program

- This relies on context-sensitive information: the type of name or of a num depends on its identity rather than its syntactic category

Type inference for expressions

Assume

- name and num that appear in the parse tree has already an attribute type
- $\mathcal{F}_{+}$ $\mathcal{F}_{-}$ $\mathcal{F}_{\times}$ $\mathcal{F}_{\div}$ encode information as the one for + in this table

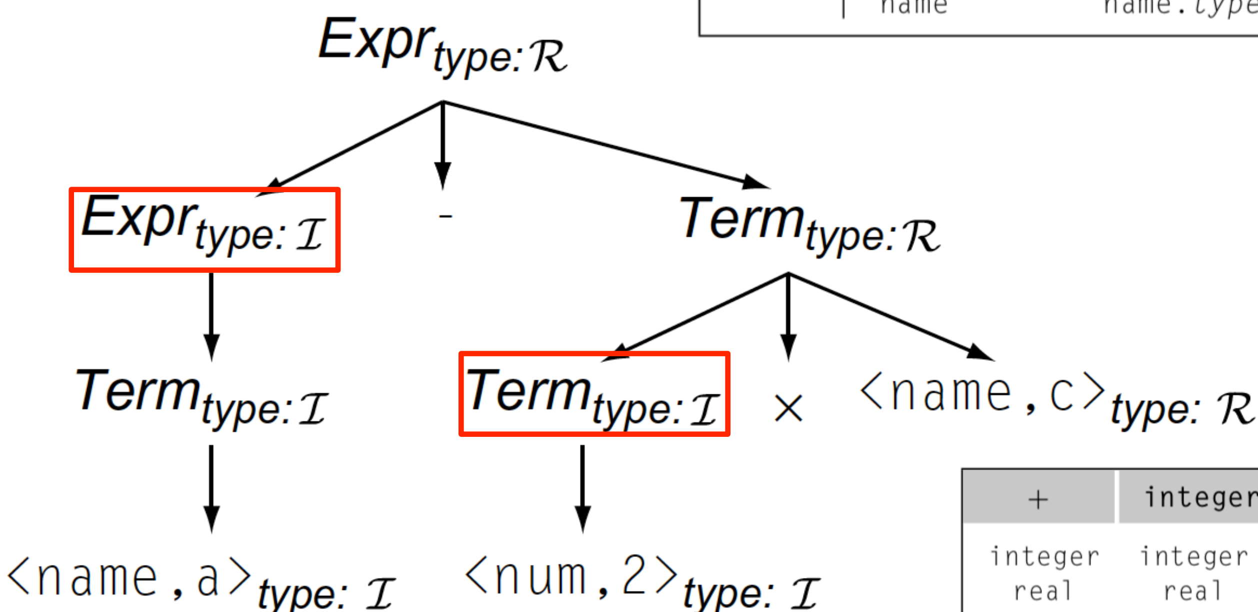
+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

The attribute Grammar

Production	Attribution Rules
$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.type \leftarrow \mathcal{F}_{+}(Expr_1.type, Term.type)$
$\quad \mid \; Expr_1 - Term$	$Expr_0.type \leftarrow \mathcal{F}_{-}(Expr_1.type, Term.type)$
$\quad \mid \; Term$	$Expr_0.type \leftarrow Term.type$
$Term_0 \rightarrow Term_1 \; Factor$	$Term_0.type \leftarrow \mathcal{F}_{\times}(Term_1.type, Factor.type)$
$\quad \mid \; Term_1 \; Factor$	$Term_0.type \leftarrow \mathcal{F}_{\div}(Term_1.type, Factor.type)$
$\quad \mid \; Factor$	$Term_0.type \leftarrow Factor.type$
$Factor \rightarrow (Expr)$	$Factor.type \leftarrow Expr.type$
$\quad \mid \; \texttt{num}$	$\texttt{num}.type$ *is already defined*
$\quad \mid \; \texttt{name}$	$\texttt{name}.type$ *is already defined*

a - 2 x c

Production			Attribution Rules
$Expr_0$	$\rightarrow$	$Expr_1 + Term$	$Expr_0.type \leftarrow \mathcal{F}_{+}(Expr_1.type, Term.type)$
	|	$Expr_1 - Term$	$Expr_0.type \leftarrow \mathcal{F}_{-}(Expr_1.type, Term.type)$
	|	$Term$	$Expr_0.type \leftarrow Term.type$
$Term_0$	$\rightarrow$	$Term_1\ Factor$	$Term_0.type \leftarrow \mathcal{F}_{\times}(Term_1.type, Factor.type)$
	|	$Term_1\ Factor$	$Term_0.type \leftarrow \mathcal{F}_{\div}(Term_1.type, Factor.type)$
	|	$Factor$	$Term_0.type \leftarrow Factor.type$
$Factor$	$\rightarrow$	($Expr$)	$Factor.type \leftarrow Expr.type$
	|	num	num.type *is already defined*
	|	name	name.type *is already defined*



+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

For each case the operand will have a different type from the type of the other operand the compiler need to add a conversion

Type inference for expressions

- We have assumed that name.type and num.type were already defined

- but to fill those values using an attribute grammar the compiler writer would need to develop a set of rules for the portion of the grammar that handle declarations, to collect this information and to add attributes for propagate that information on all variables: many copy rules!

- at the leaf node the rules need to extract the appropriate facts

The result set of rules would be similar the one of the previous example

Problems with Attribute-Grammar Approach

- Attribute grammars handle well problems where all information flows in the same direction and is local
- There is a problem in handling non local information
- Non-local computation need a lots of supporting rules
  - Copy rules increase cognitive overhead
  - Copy rules increase space requirements
    - Need copies of attributes

- Result is an attributed tree
  - Must build the parse tree
  - All the answer are in the values of the attributed tree. To find them later phases has either visit the tree for answers or copy relevant information in the root (more copy rules)

To solve the Problems

- Drop the functional approach of the rules
- Add a central repository for attributes
- An attribute rule can write or read from a global table: it can access to non-local information

The Realist's Alternative

Ad-hoc syntax-directed translation

- Build on the grammar as attribute grammar
- Associate a snippet (action) of code with each production
- If you have a descendent parser call a procedure at each parsing routine
- In the bottom up parser, for each reduction, the corresponding snippet runs (in the next slides assume a bottom up parser!)

Reworking the Example

The variable cost is global!

1	$Block_0$	→	$Block_1$ *Assign*	
2		|	*Assign*	
3	$Assign_0$	→	*Ident = Expr ;*	cost ← cost + COST(store)
4	$Expr_0$	→	$Expr_1$ + *Term*	cost ← cost + COST(add)
5		|	$Expr_1$ - *Term*	cost ← cost + COST(sub)
6		|	*Term*	
7	$Term_0$	→	$Term_1$ * *Factor*	cost ← cost + COST(mult)
8		|	$Term_1$ / *Factor*	cost ← cost + COST(div)
9		|	*Factor*	
10	*Factor*	→	*(Expr)*	
11		|	*Number*	cost ← cost + COST(loadI)

12		|	*Ident*	i ← hash(Ident);
				if (Table[i].loaded = false)
				then {
				cost ← cost + COST(load)
				Table[i].loaded ← true
				}

This looks cleaner & simpler than the AG !

One missing detail: initializing cost

Reworking the Example (with load tracking)

0	*Start*		*Init Block*	
.5	*Init*		ε	cost ← 0
1	$Block_0$	→	$Block_1$ *Assign*	
2		|	*Assign*	
3	$Assign_0$	→	*Ident = Expr ;*	cost ← cost + COST(store)

and so on as shown on previous slide…

- Before parser can reach Block, it must reduce Init
- Reduction by Init sets cost to zero

We split the production to create a reduction in the middle — for the sole purpose of hanging an action there. This trick has many uses.

To make this work

- Need names for attributes of each symbol on lhs & rhs
  - Yacc introduced **$$**, **$1**, **$2**, … **$n**, left to right
- Need an evaluation scheme
  - Fits nicely into **LR(1)** parsing algorithm

Example — Assigning Types in Expression Nodes

- Assume typing functions or tables F_+, F_-, $F_\times$, and $F_\div$

$F_\times$	Int 16	Int 32	Float	Double
Int 16	Int 16	Int 32	Float	Double
Int 32	Int 32	Int 32	Float	Double
Float	Float	Float	Float	Double
Double	Double	Double	Double	Double

1	*Goal*	→	*Expr*	\$\$ = \$1;
2	*Expr*	→	*Expr* + *Term*	\$\$ = F_+(\$1,\$3);
3		|	*Expr* - *Term*	\$\$ = F_-(\$1,\$3);
4		|	*Term*	\$\$ = \$1;
5	*Term*	→	*Term* * *Factor*	\$\$ = $F_\times$(\$1,\$3);
6		|	*Term* / *Factor*	\$\$ = $F_\div$(\$1,\$3);
7		|	*Factor*	\$\$ = \$1;
8	*Factor*	→	(*Expr*)	\$\$ = \$2;
9		|	number	\$\$ = type of num;
10		|	ident	\$\$ = type of ident;

Assuming leaf nodes already have typed information!

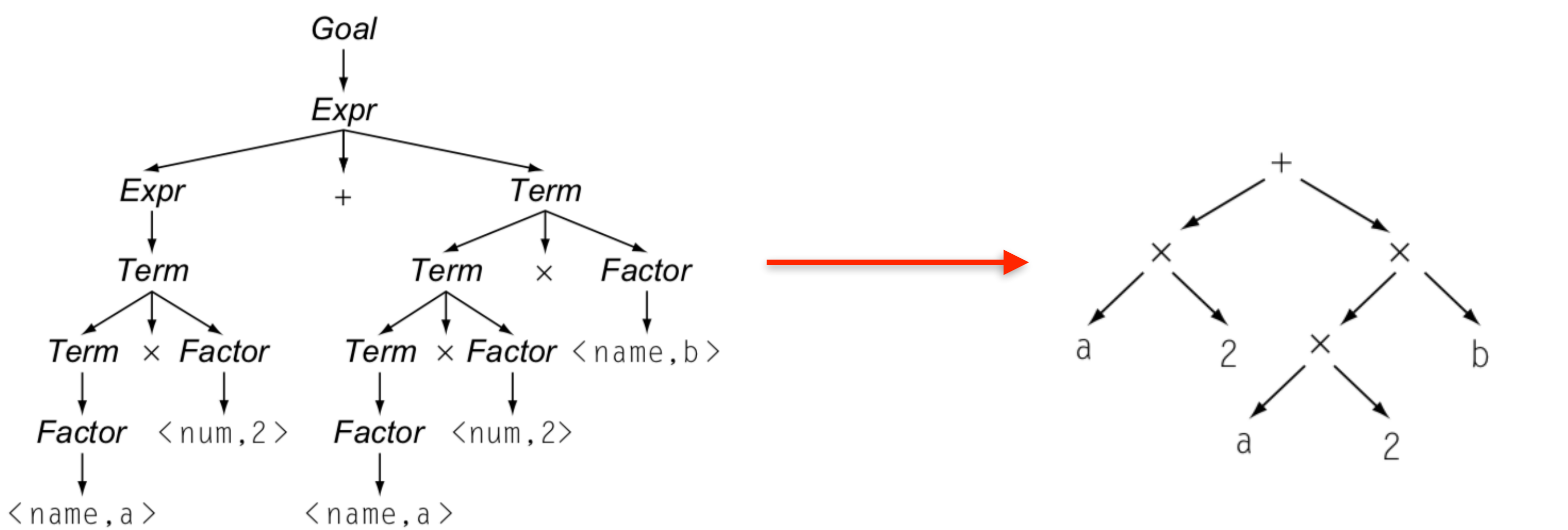
Types of Intermediate Representations

Three major categories

- Structural
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large
- Linear
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange
- Hybrid
  - Combination of graphs and linear code

Intermediate reppresentations: Abstract syntax tree

- Abstract syntax tree: retains the essential strutture of the parse tree but eliminates the non-terminal nodes



Intermediate reppresentations: Linear IR

- Linear code: sequence of instructions that execute in their order of appearance

```
push  2
push  b
multiply
push  a
subtract
```

Stack-Machine Code

$t_1 \leftarrow 2$
$t_2 \leftarrow b$
$t_3 \leftarrow t_1 \times t_2$
$t_4 \leftarrow a$
$t_5 \leftarrow t_4 - t_3$

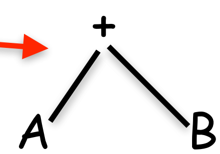
Three-Address Code

- In your book ILOC is an example of three-address code

Building an Abstract Syntax Tree

Assume the following 4 routines :

- MakeAddNode (A, B)
- MakeSubNode (A, B)
- MakeDivNode (A, B)
- MakeMulNode (A, B)



and

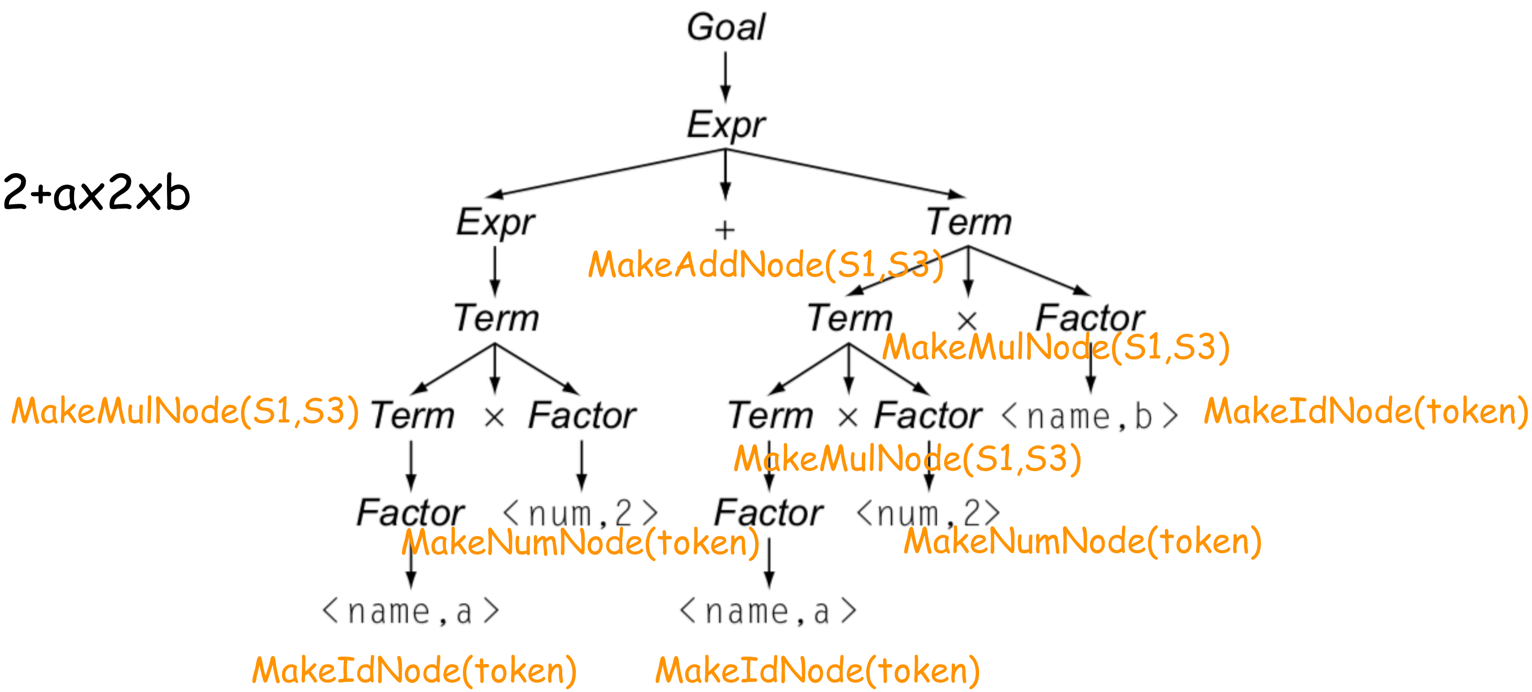
- MakeNumNode(<num,val>) → val
- MakeIdNode(<name,x>) → x

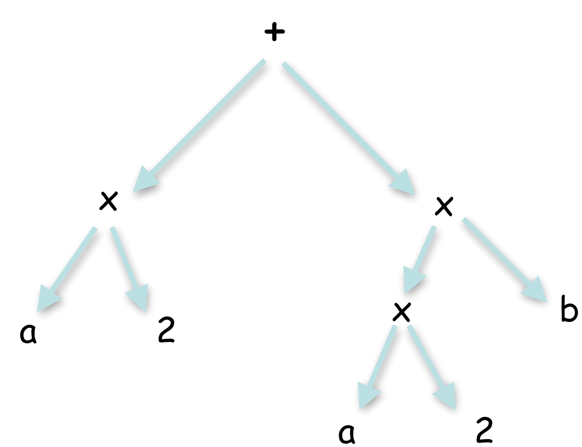
Example — Building an Abstract Syntax Tree

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

1	*Goal*	→	*Expr*	$$ = $1;
2	*Expr*	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
3		|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
6		|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	→	(*Expr*)	$$ = $2;
9		|	<u>number</u>	$$ = MakeNumNode(token);
10		|	<u>ident</u>	$$ = MakeIdNode(token);

ax2+ax2xb


Goal
Expr
Expr
+
Term
MakeAddNode(S1,S3)
Term
Term
×
Factor
MakeMulNode(S1,S3)
MakeMulNode(S1,S3)
Term
×
Factor
Term
×
Factor
<name,b>
MakeIdNode(token)
MakeMulNode(S1,S3)
Factor
<num,2>
Factor
<num,2>
MakeNumNode(token)
MakeNumNode(token)
<name,a>
<name,a>
MakeIdNode(token)
MakeIdNode(token)


+
x
x
a
2
x
b
a
2

Emitting ILOC

Assume

- NextRegister() returns a new register name
- 4 routines
  - Emit(sub, r1,r2,r3) → sub r1, r2, r3 (r1-r2->r3)
  - Emit(mult, r1,r2,r3) → mult r1, r2, r3 (r1xr2->r3)
  - Emit(add, r1,r2,r3) → add r1, r2, r3 (r1+r2->r3)
  - Emit(div, r1,r2,r3) → div r1, r2, r3 (r1/r2->r3)

activationrecordpointer

- EmitLoad(iden, r) → loadAI(rarp,@iden,r)

  Memory(rarp + c)->r

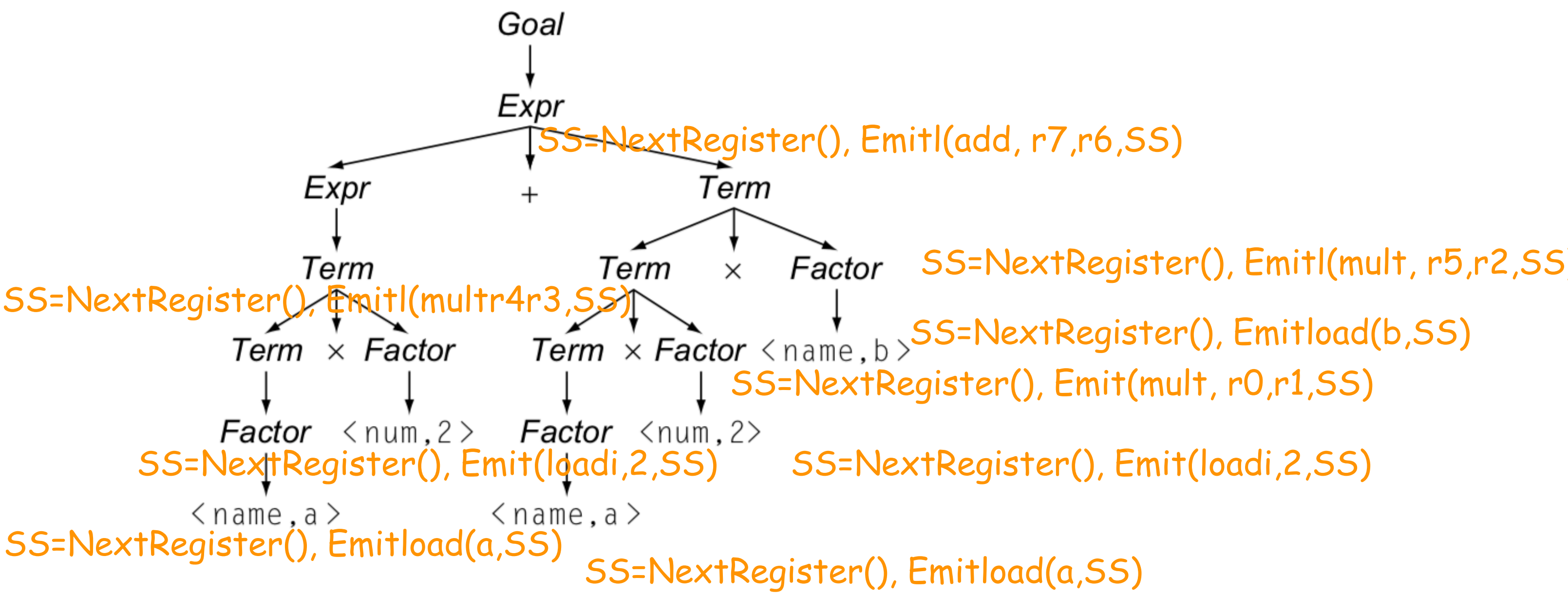
- Emit(loadi,n,r) → loadI(n,r) n->r

Example — Emitting ILOC

1	*Goal*	→	*Expr*	
2	*Expr*	→	*Expr* + *Term*	$$ = *NextRegister*(); *Emit*(add, $1,$3, $$);
3		|	*Expr* - *Term*	$$ = *NextRegister()*; *Emit*(sub, $1, $3, $$);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = *NextRegister()*; *Emit*(mult,$1, $3, $$)
6		|	*Term* / *Factor*	$$ = *NextRegister()*' *Emit*(div, $1, $3, $$);
7		|	*Factor*	$$ = $1;

Example — Emitting ILOC

8	*Factor*	→	(*Expr*)	$$ = $2;
9		|	number	$$ = *NextRegister()*; *Emit*(loadi,Value(lexeme),$$);
10		|	ident	$$ = *NextRegister()*; *EmitLoad*(ident,$$);




```
LoadAI rarp, @a, r0
LoadI   2 r1
LoadAI rarp, @b, r2
LoadI   2 r3
LoadAI rarp, @a, r4
Mult      ro,  r1, r5
Mult      r5,  r2, r6
Mult      r4,  r3, r7
Add       r7,  r6, r8
```

Reality

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

Advantages

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

Disadvantages

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

Making Ad-hoc SDT Work

How do we fit this into an LR(1) parser?

```
stack.push(INVALID);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
      s = stack.top();
      if ( ACTION[s,token] == "reduce A→β" ) then {
         stack.popnum(2*|β|);      // pop 2*|β| symbols
         s = stack.top();
         stack.push(A);            // push A
         stack.push(GOTO[s,A]);    // push next state
      }
      else if ( ACTION[s,token] == "shift si" ) then {
            stack.push(token); stack.push(si);
            token ← scanner.next_token();
      }
      else if ( ACTION[s,token] == "accept"
                        & token == EOF )
            then break;
      else throw a syntax error;
}
report success;
```

From previous lectures

Augmented LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(NULL);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
      s = stack.top();
      if ( ACTION[s,token] == "reduce A→β" ) then {

         /* insert case statement here */

         stack.popnum(3*|β|);     // pop 3*|β| symbols
         s = stack.top();
         stack.push(A);           // push A
         stack.push(GOTO[s,A]);   // push next state
      }
      else if ( ACTION[s,token] == "shift si" ) then {
            stack.push(token); stack.push(si);
            token ← scanner.next_token();
      }
      else if ( ACTION[s,token] == "accept"
                        & token == EOF )
            then break;
      else throw a syntax error;
}
report success;
```

To add yacc-like actions

- Stack 3 items per symbol rather than 2 (3rd is $$)
- Add case statement to the reduction processing section
  - → Case switches on production number
  - → Each case clause holds the code snippet for that production
  - → Substitute appropriate names for $$, $1, $2, …
- Slight increase in parse time
- increase in stack space

How do we fit this into an LR(1) parser?

- Need a place to store the attributes
  - Stash them in the stack, along with state and symbol
  - Push three items each time, pop 3 x $|\beta|$ symbols
- Need a naming scheme to access them
  - $n translates into stack location (top - 3n)
- Need to sequence rule applications
  - On every reduce action, perform the action rule
  - Add a giant case statement to the parser