Content Sensitive Analysis

Roadmap (Where are we?)

Last lecture

- LR(1) parsing
  - → Building ACTION / GOTO tables
  - → Shift / reduce and reduce / reduce conflicts
  - → SLR(1), LALR(1) parsers

This lecture

- Context-sensitive analysis
  - → Motivation
  - → Attribute grammars
    - Attributes
    - Evaluation order
  - → Ad hoc Syntax-directed translation

Context-Sensitive Analysis: Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, 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 …)

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

All these errors are “deeper than syntax”

To generate code, we need to understand its meaning !

Beyond Syntax

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 each 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?

These are beyond a CFG

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? ***(attributed grammars?)***
- Use *ad-hoc* techniques
  - → Symbol tables
  - → *Ad-hoc* code ***(action routines)***

In scanning & parsing, formalism won; different story here.

Beyond Syntax

Telling the story

- The attribute grammar formalism is important
  - → Succinctly makes many points clear
  - → Sets the stage for actual, *ad-hoc* practice
- The problems with attribute grammars motivate practice
  - → Non-local computation
  - → Need for centralized information

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

Attribute Grammars

What is an attribute grammar?

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

Example grammar

Number	→	Sign List
Sign	→	+
	|	–
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

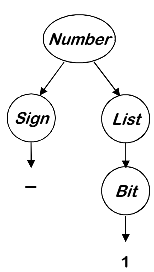
This grammar describes signed binary numbers

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

Examples

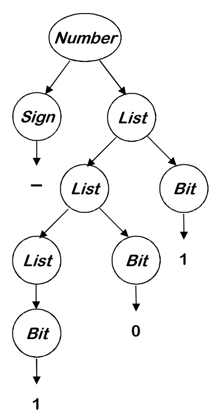
For "–1"

Number ⇒ ***Sign List***
⇒ ***– List***
⇒ ***– Bit***
⇒ ***– 1***



For "–101"

Number ⇒ ***Sign List***
⇒ ***Sign List Bit***
⇒ ***Sign List 1***
⇒ ***Sign List Bit 1***
⇒ ***Sign List 0 1***
⇒ ***Sign Bit 0 1***
⇒ ***Sign 1 0 1***
⇒ ***– 101***



We will use these two throughout the lecture

Attribute Grammars

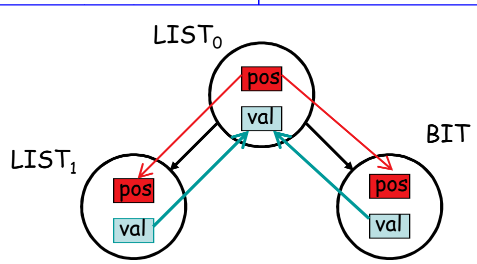
Add rules to compute the decimal value of a signed binary number

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* ← $2^{Bit.pos}$

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

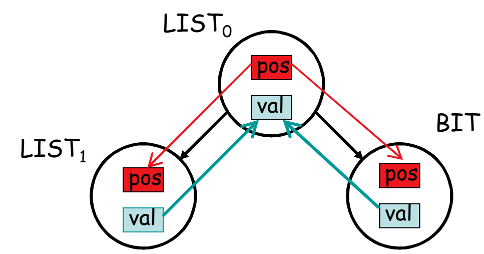
Attribute Grammars

Productions			**Attribution Rules**
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*



- Semantic rules define partial dependency graph
- Value flow top down or across: inherited attributes
- Value flow bottom-up: synthesized attributes

Attribute Grammars

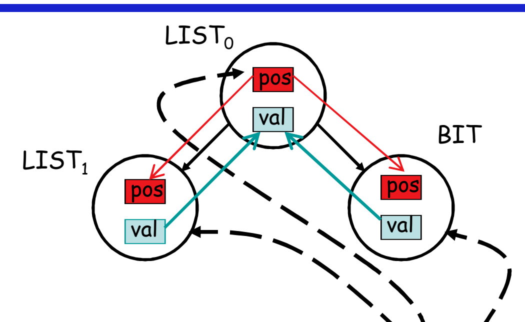


Note:
- Semantic rules associated with production $A \rightarrow \alpha$ have to specify the values for all
  - synthesized attributes for A (root)
  - inherited attributes for grammar symbols in α (children)

  $\Rightarrow$ rules must specify local value flow!

- Terminals can be associated with values returned by the scanner. These input values are associated with a synthesized attribute.
- Starting symbol cannot have inherited attributes.

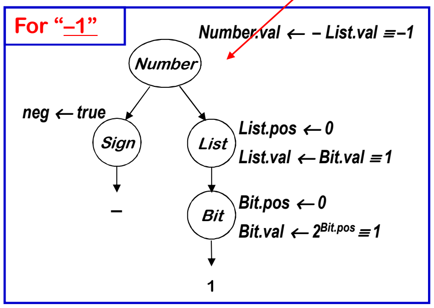
Attribute Grammars



Question: What rules specify values for , and ?

Back to the Examples

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

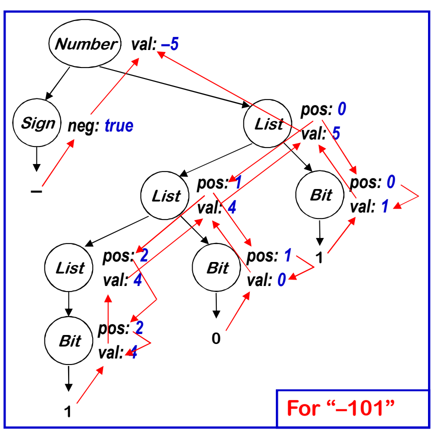
Other orders are possible

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
- Label identical terms in production for uniqueness
- Rules & parse tree define an attribute dependence graph
  - → Graph must be non-circular

This produces a high-level, functional specification

Synthesized attribute

- → Depends on values from children

Inherited attribute

- → Depends on values from siblings & parent

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: synthesized attributes only
- Evaluate in a single bottom-up pass

Good match to LR parsing

Inherited Attributes

- Use values from parent, constants, & siblings
- L-attributed grammars:

$A \rightarrow X_1 X_2 \ldots X_n$ and each inherited attribute of X_i depends on
- attributes of $X_1 X_2 \ldots X_{i-1}$, and
- inherited attributes of A

- Evaluate in a single top-down pass (left to right)

Good match for LL parsing

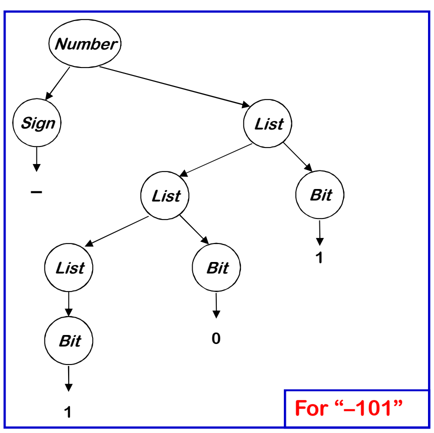
Evaluation Methods

Dynamic, dependence-based methods
- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Define attributes in topological order

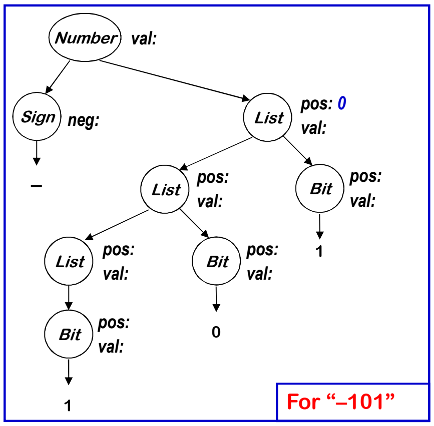
Rule-based methods *(treewalk)*
- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

Oblivious methods *(passes, dataflow)*
- Ignore rules & parse tree
- Pick a convenient order (at design time) & use it

Back to the Example

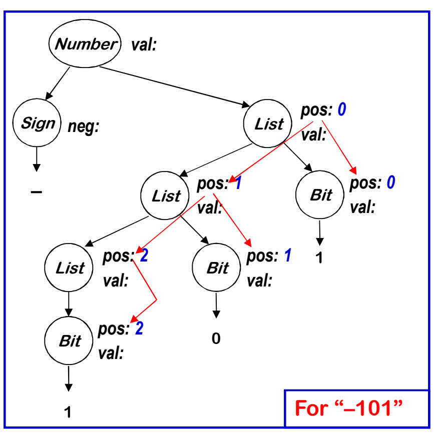


Back to the Example



CS430

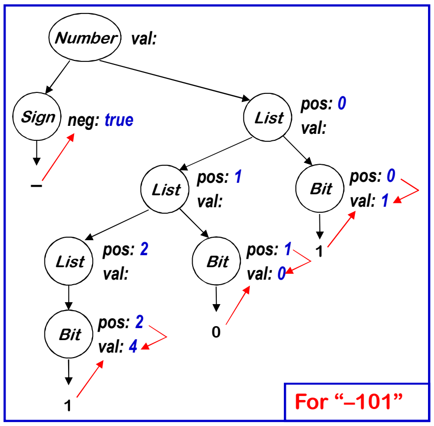
Back to the Example



Inherited Attributes

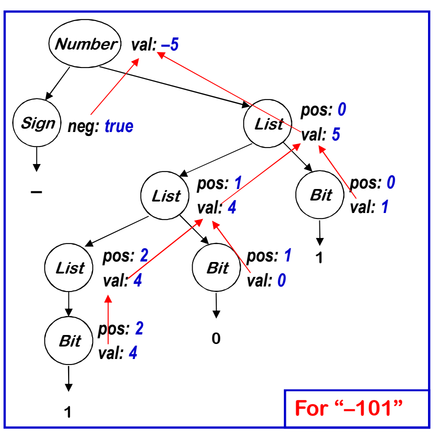
CS430

Back to the Example



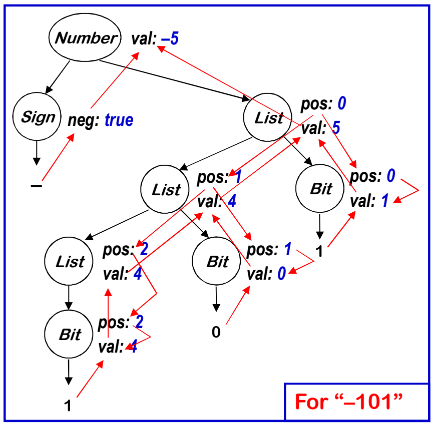
Synthesized attributes

Back to the Example



Synthesized attributes

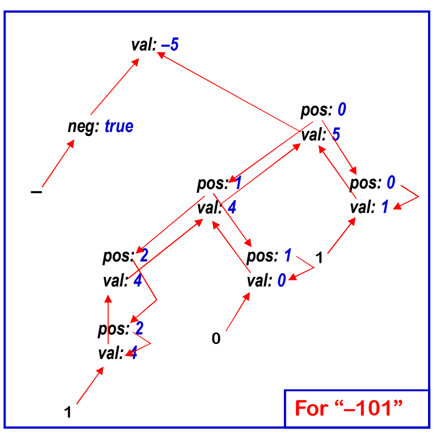
Back to the Example



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

- 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

Many evaluation methods discover circularity dynamically
⇒ Bad property for a compiler to have

SNC grammars were first defined by Kennedy & Warren

An Extended Example

Grammar for a basic block (§ 4.3.3)

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

Let's estimate cycle counts

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

Simple problem for an *AG*

Hey, this looks useful !

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)

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

Things will get more complicated.

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

This looks 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

$Expr_0 \rightarrow Expr_1$ + Term	$Expr_0$.cost ← $Expr_1$.cost + COST(add) + Term.cost ; $Expr_1$.Before ← $Expr_0$.Before ; Term.Before ← $Expr_1$.After; $Expr_0$.After ← 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

The Moral of the Story

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

The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - → Need copies of attributes
- Result is an attributed tree
  - → Must build the parse tree
  - → Either search tree for answers or copy them to the root

Addressing the Problem

What would a good programmer do?

- Introduce a central repository for facts
- Table of names
  - → Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
  - → Clean, efficient implementation
  - → Good techniques for implementing the table *(hashing, § B.4)*
  - → When its done, information is in the table !
  - → Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - → Do we care?

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate pieces of code with each production
- At each reduction, the corresponding code is executed
- Allowing arbitrary code provides complete flexibility
  - → Includes ability to do tasteless & bad things

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
- Need an evaluation scheme
  - → Fits nicely into **LR(1)** parsing algorithm

Reworking the Example (*with load tracking*)

$Block_0$	→	$Block_1$ Assign	
	|	Assign	
Assign	→	Ident = Expr ;	cost← cost + *COST*(store);
$Expr_0$	→	$Expr_1$ + Term	cost← cost + *COST*(add);
	|	$Expr_1$ - Term	cost← cost + *COST*(sub);
	|	Term	
$Term_0$	→	$Term_1$ * Factor	cost← cost + *COST*(mult);
	|	$Term_1$ / 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; } }

This looks cleaner & simpler than the AG sol'n !

One missing detail: initializing "cost"; (we ignore "Table[ ] for now)

Reworking the Example (*with load tracking*)

Start	→	Init Block	
Init	→	ε	cost ← 0;
$Block_0$	→	$Block_1$ Assign	
	|	Assign	
Assign	→	Ident = Expr ;	cost← cost + COST(store);

… and so on as in the previous version of the example …

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

This is an example of splitting a production to create a reduction in the middle — for the sole purpose of hanging an action routine there (marker production)!

Reworking the Example (*with load tracking*)

$Block_0$	→	$Block_1$ Assign	$$ ← $1 + $2 ;
	|	Assign	$$ ← $1 ;
Assign	→	Ident = Expr ;	$$← COST(store) + $3;
$Expr_0$	→	$Expr_1$ + Term	$$← $1 + COST(add) + $3;
	|	$Expr_1$ - Term	$$← $1 + COST(sub) + $3;
	|	Term	$$ ← $1;
$Term_0$	→	$Term_1$ * Factor	$$ ← $1 + COST(mult) + $3;
	|	$Term_1$ / Factor	$$ ← $1 + COST(div) + $3;
	|	Factor	$$ ← $1;
Factor	→	(Expr)	$$ ← $2;
	|	Number	$$ ← COST(loadi);
	|	Identifier	{ i← hash(Identifier); if (Table[i].loaded = false) then { $$ ← COST(load); Table[i].loaded ← true; } else $$ ← 0 }

This version passes the values through attributes. It avoids the need for initializing "cost"

However, Table[ ] still needs to be initialized

Example — Building an Abstract Syntax Tree

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

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	$$ = MakeNumNode(token);
	|	id	$$ = MakeIdNode(token);

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

Most parser generators support a yacc-like notation

Typical Uses (Semantic Analysis)

- 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 conformability of expression → bottom-up walk
  - → Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Check actual vs. formal parameter list
    - Positional or keyword associations

assumes table is global

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools
- (Somewhat) abstract names for symbols

Differences

- Actions applied as a unit; not true for AG rules
- Anything goes in *ad-hoc* actions; AG rules are (purely) functional
- AG rules are higher level than *ad-hoc* actions

Making Ad-hoc Syntax Directed Translation Work

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 - 3 x ($|\beta|$ - n)
- Need to sequence rule applications
  - → On every reduce action, perform the action rule

What about a rule that must work in mid-production?

- Can transform the grammar
  - → Split it into two parts at the point where rule must go and apply the rule on reduction to the appropriate part
  - → Introduce marker productions $M \rightarrow \varepsilon$ with appropriate action