RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Attribute Grammar and Syntax Directed Translation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Homework Problem Set 7 will be posted this weekend
- Second project will come out tomorrow
- 4/7 Class will be covered by Professor Uli Kremer

Review - 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

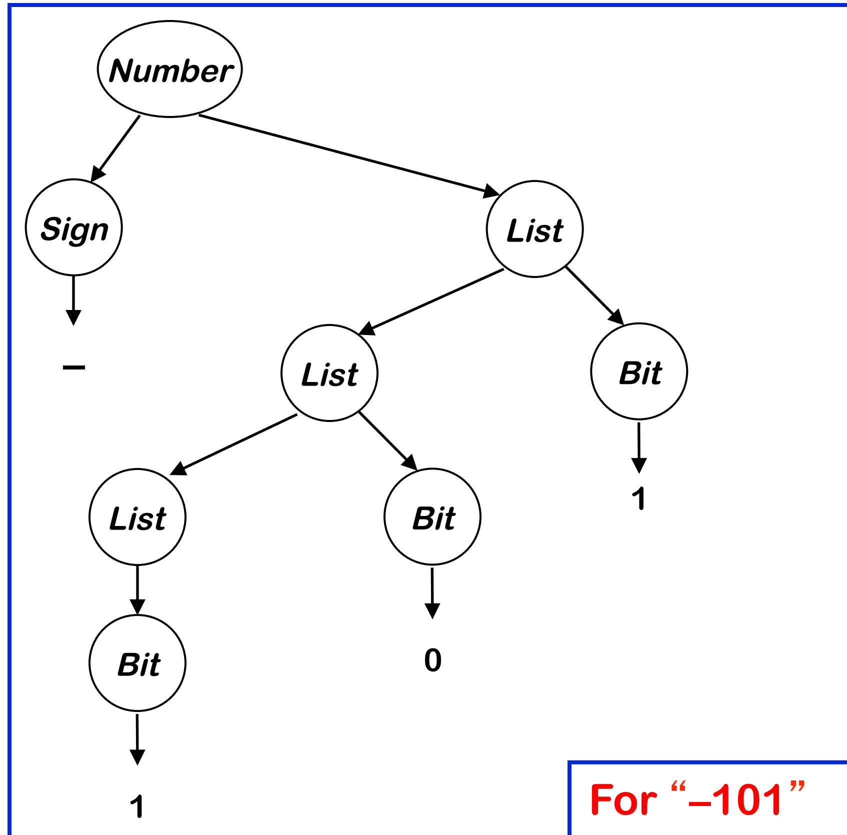
Number	→	Sign List
Sign	→	+
	|	-
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

This grammar describes signed binary numbers

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

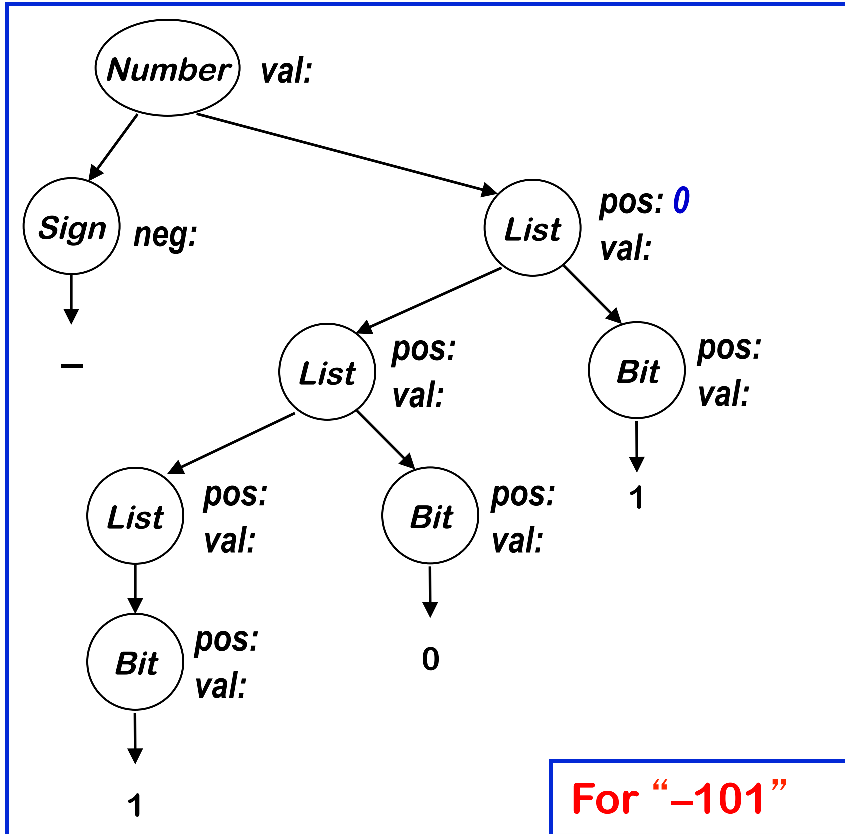
Example

compute the decimal value of a signed binary number



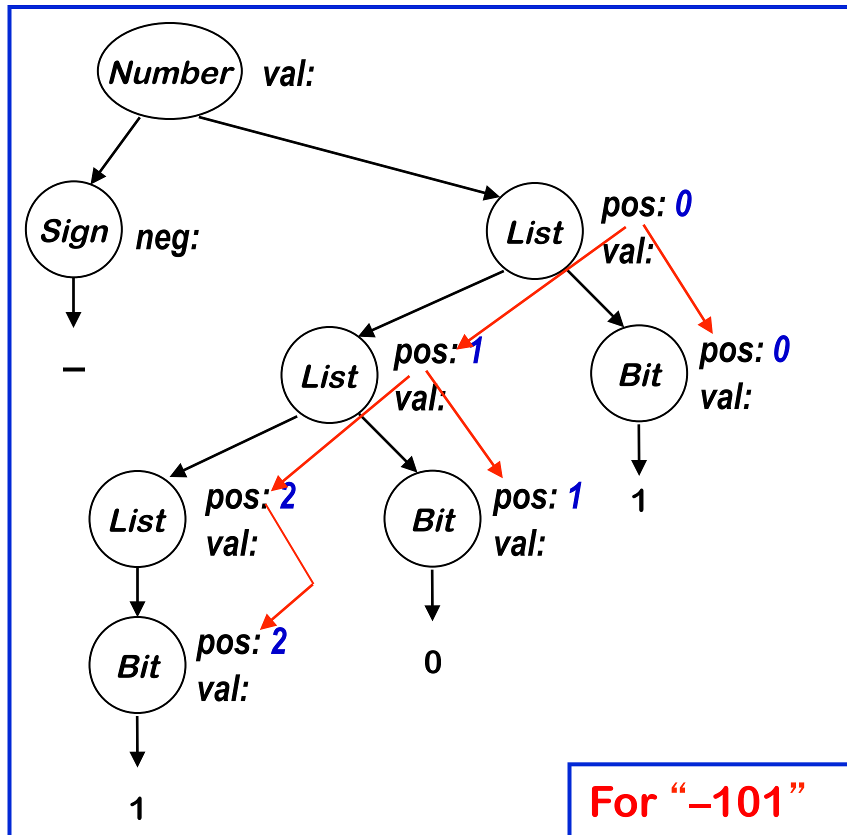
Example

compute the decimal value of a signed binary number



Example

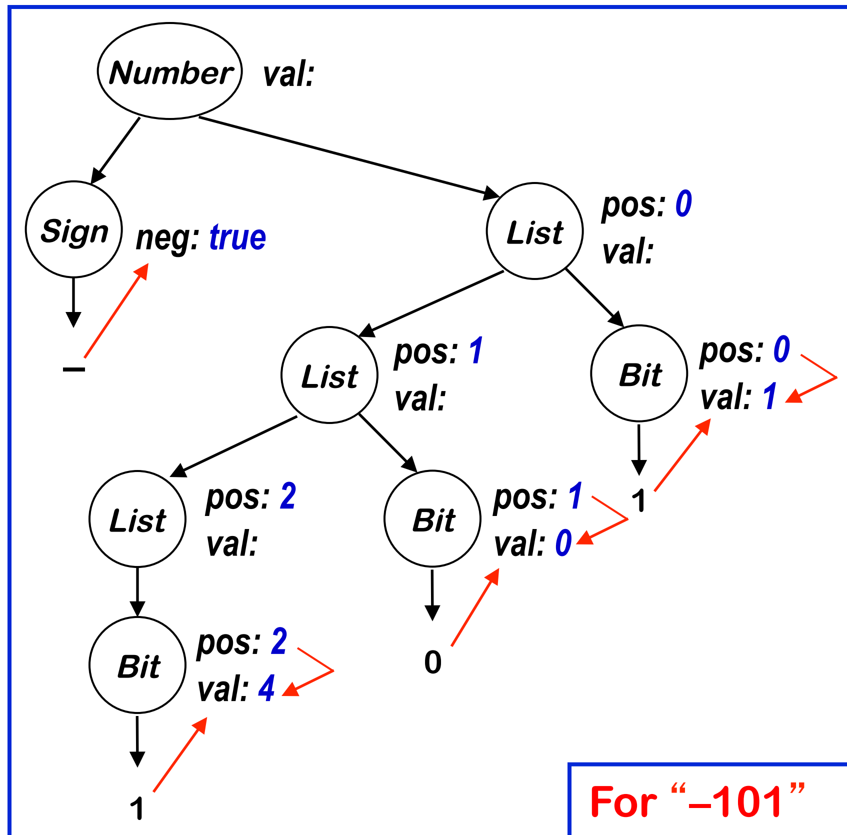
compute the decimal value of a signed binary number



Inherited Attributes

Example

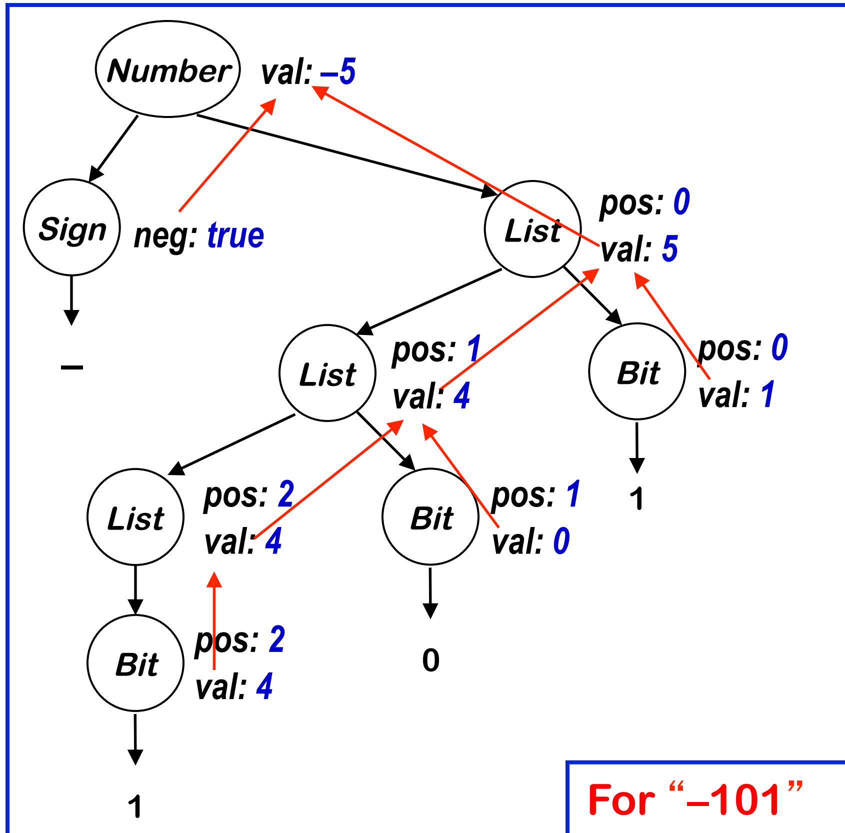
compute the decimal value of a signed binary number



Synthesized attributes

Example

compute the decimal value of a signed binary number



Synthesized attributes

Attribute Grammars

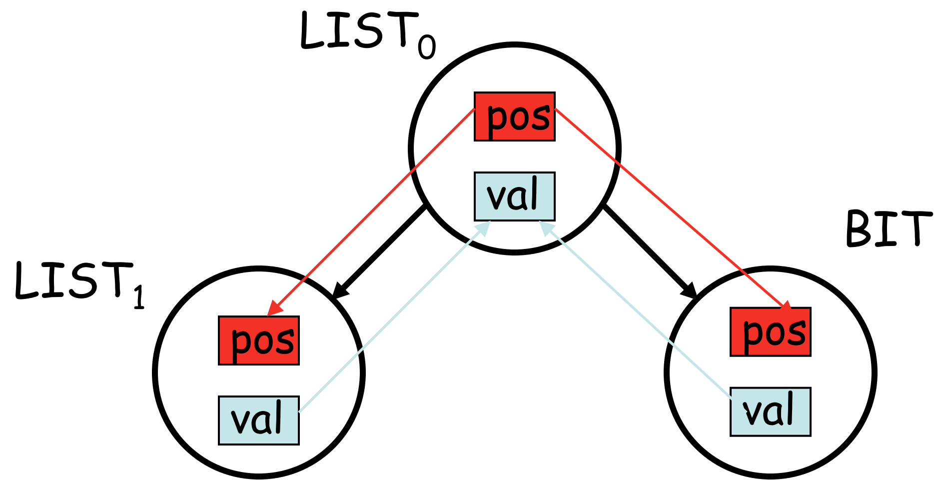
Add rules to compute the decimal value of a signed binary number

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*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

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

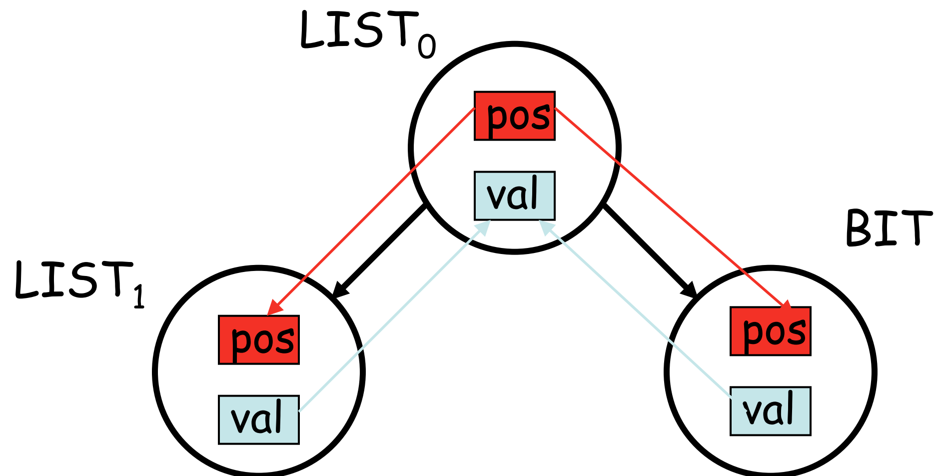
Attribute Grammars

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:

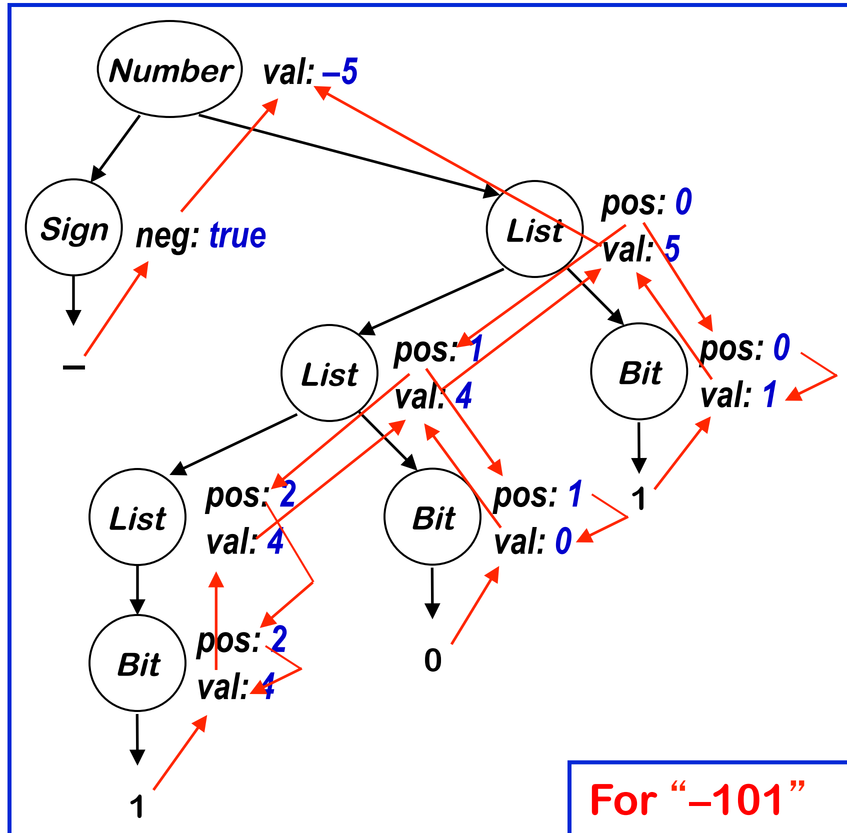
- semantic rules associated with production $A \rightarrow \alpha$ have to specify the values for all
  - synthesized attributes for A (root)
  - inherited attributes for grammar symbols in α (children)

  $\Rightarrow$ 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.

Example revisited

compute the decimal value of a signed binary number



If we show the computation ...

& then peel away the parse tree ...

Example revisited

compute the decimal value of a signed binary number

val:
neg:
–
pos: 0
val:
pos:
val:
pos:
val:
pos:
val:
pos:
val:
pos:
val:
1
0
1
1

For "–101"

All that is left is the attribute dependence graph.

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

The dependence graph must be acyclic

Example revisited

compute the decimal value of a signed binary number

val: –5
neg: true
pos: 0
val: 5
–
pos: 1
val: 4
pos: 0
val: 1
pos: 2
val: 4
pos: 1
val: 0
1
pos: 2
val: 4
0
1

For "–101"

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

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

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*

An Extended Example (*continued*)

Block₀	→	*Block₁ Assign*	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
	|	*Assign*	$Block_0.cost \leftarrow Assign.cost$
Assign	→	*Ident = Expr ;*	$Assign.cost \leftarrow$ **COST**(store) + Expr.cost
Expr₀	→	*Expr₁ + Term*	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(add) + Term.cost
	|	*Expr₁ - Term*	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(sub) + Term.cost
	|	*Term*	$Expr_0.cost \leftarrow Term.cost$
Term₀	→	*Term₁ * Factor*	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(mult) + Factor.cost
	|	*Term₁ / Factor*	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(div) + Factor.cost
	|	*Factor*	$Term_0.cost \leftarrow Factor.cost$
Factor	→	*(Expr)*	$Factor.cost \leftarrow Expr.cost$
	|	*Number*	$Factor.cost \leftarrow$ **COST**(loadI)
	|	*Identifier*	$Factor.cost \leftarrow$ **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 $\Rightarrow$ 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

Things will get more complicated.

A Better Execution Model

Adding load tracking

- Need sets *Before* and *After* for each production

  Question: synthesized or inherited?
- Must be initialized, updated, and passed around the tree

Production	Actions
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

$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost +$ 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

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, AG 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, …)
  - → 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 ! "cost" and Table[ ] are global variables

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

YACC : parse.y

parse.y :

```
%{
#include <stdio.h>
#include "attr.h"
int yylex();
void yyerror(char * s);
#include "symtab.h"
%}
```

Will be included verbatim in parse.tab.c

List and assign attributes

```
%union {tokentype token; }

%token PROG PERIOD PROC VAR ARRAY RANGE OF
%token INT REAL DOUBLE WRITELN THEN ELSE IF
%token BEG END ASG NOT
%token EQ NEQ LT LEQ GEQ GT OR EXOR AND DIV NOT
%token <token> ID CCONST ICONST RCONST

%start program
```

```
%%
program : PROG ID ';' block PERIOD { }
    ;
block       : BEG ID ASG ICONST END { }
    ;
```

Rules with semantic actions

```
%%

void yyerror(char* s) {
        fprintf(stderr,"%s\n",s);
        }
int
main() {
    printf("1\t");
    yyparse();
    return 1;
    }
```

Main program and "helper" functions; may contain initialization code of global structures. Will be included verbatim in parse.tab.c

Project #2 : Things to do

- Learn/Review the C programming language
- Add error productions (syntax errors)
- Define and assign attributes to non-terminals
- Implement single-level symbol table
- Perform type checking and produce required error messages; note: actions may occur at "any" location on right-hand side (implicit use of marker productions)

RUTGERS

Project #2 (see "lex & yacc", Levine et al., O'Reilly)

→ You do have to (slightly) change the scanner (scan.l)

→ How to specify and use attributes in YACC?

- Define attributes as types in attr.h

  typedef struct info_node {int a; int b} infonode;

- Include type attribute name in %union in parse.y

  %union {tokentype token; infonode myinfo; ... }

- Assign attributes in parse.y to
  - Terminals: *%token <token> ID ICONST*
  - Non-terminals: *%type <myinfo> block variables procdecls cmpdstmt*

- Accessing attribute values in parse.y
  - use $$, $1, $2 ... etc. notation:

    block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt

    { $$.a = $1.a + $2.a + $4.b;}

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);

Making Ad-hoc SDT Work

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

- Need a place to store the attribute and their values
  - → 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

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

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

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

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea (panic mode):

- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser (panic mode):

- find a restartable state on the stack (has transition for nonterminal A)
- move to a consistent place in the input (token that can follow A)
- perform (error) reduction (for nonterminal A)
- print an informative message

Error Recovery in YACC

Yacc's (bison's) error mechanism (note: version dependent!)

- designated token **error**
- used in error productions of the form

  $A \rightarrow$ **error** α // basic case
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the **error** token
- resumes parsing to match α

  special cases:
  - α = w, where w is string of terminals: skip input until w has been read
  - $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list ';' stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "end" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end

↑ ↑ resume parsing

***Error: illegal statement

Next Class

Intermediate representations

Read EaC: Chapters 5.1 – 5.3