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 is often used.

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!

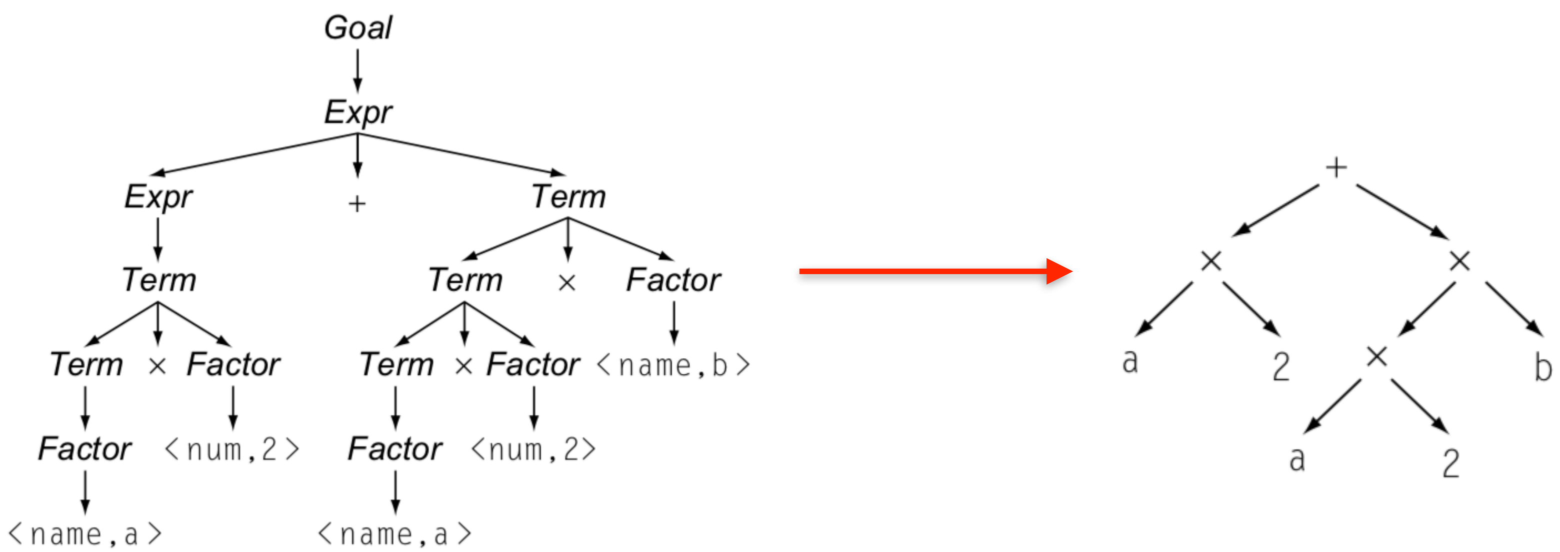
Different kinds of Intermediate Representations

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 representations: Abstract syntax tree

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



Intermediate representations: 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

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

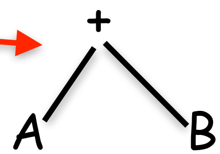
Three-Address Code

- 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

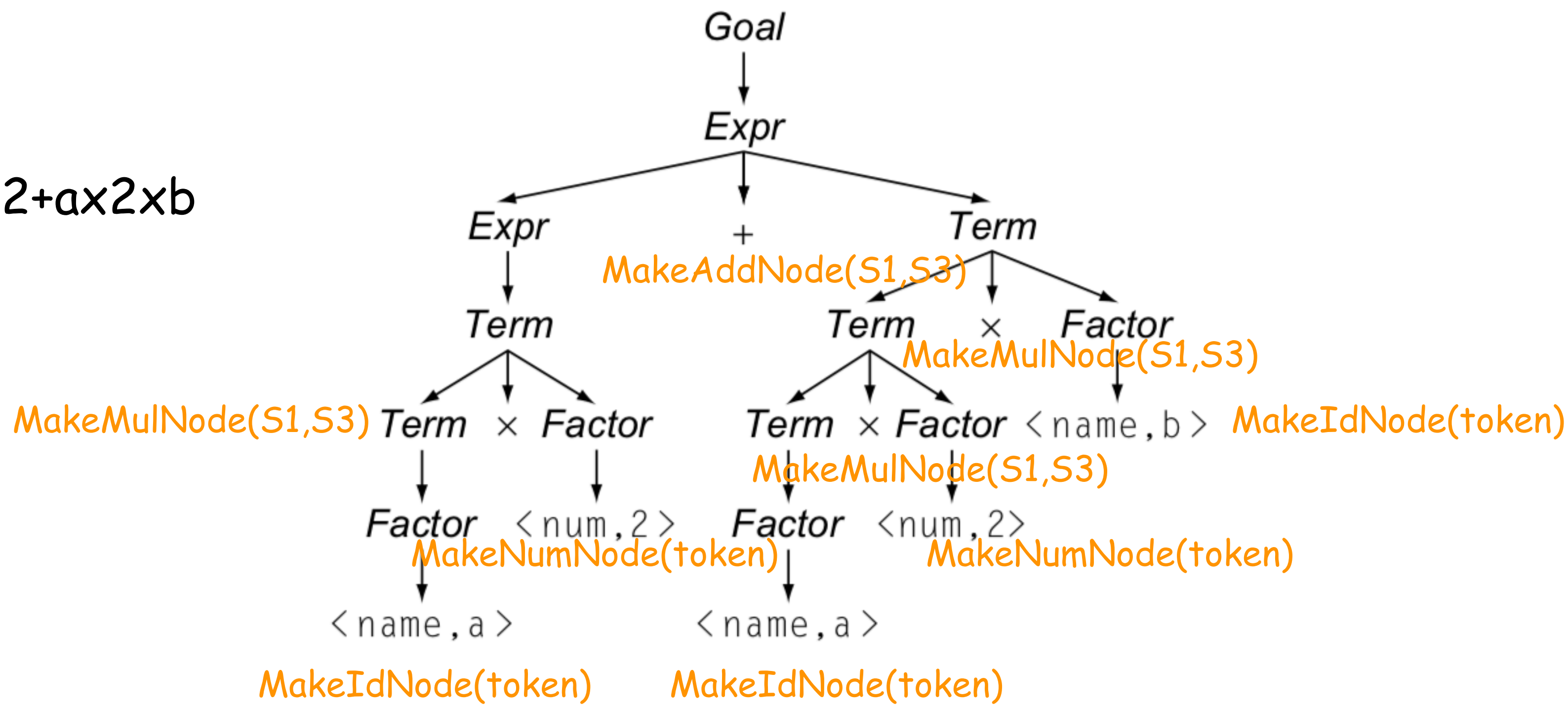
- MakeNumNode(<num,val>) → val
- MakeIdNode(<name,x>) → x

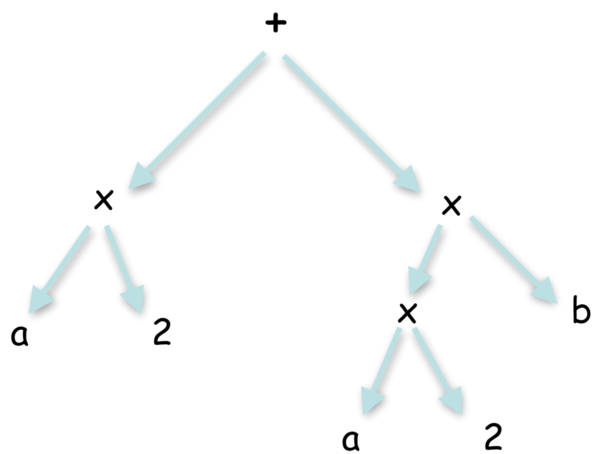
Example — Building an Abstract Syntax Tree

- 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		|	number	$$ = MakeNumNode(token);
10		|	ident	$$ = 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

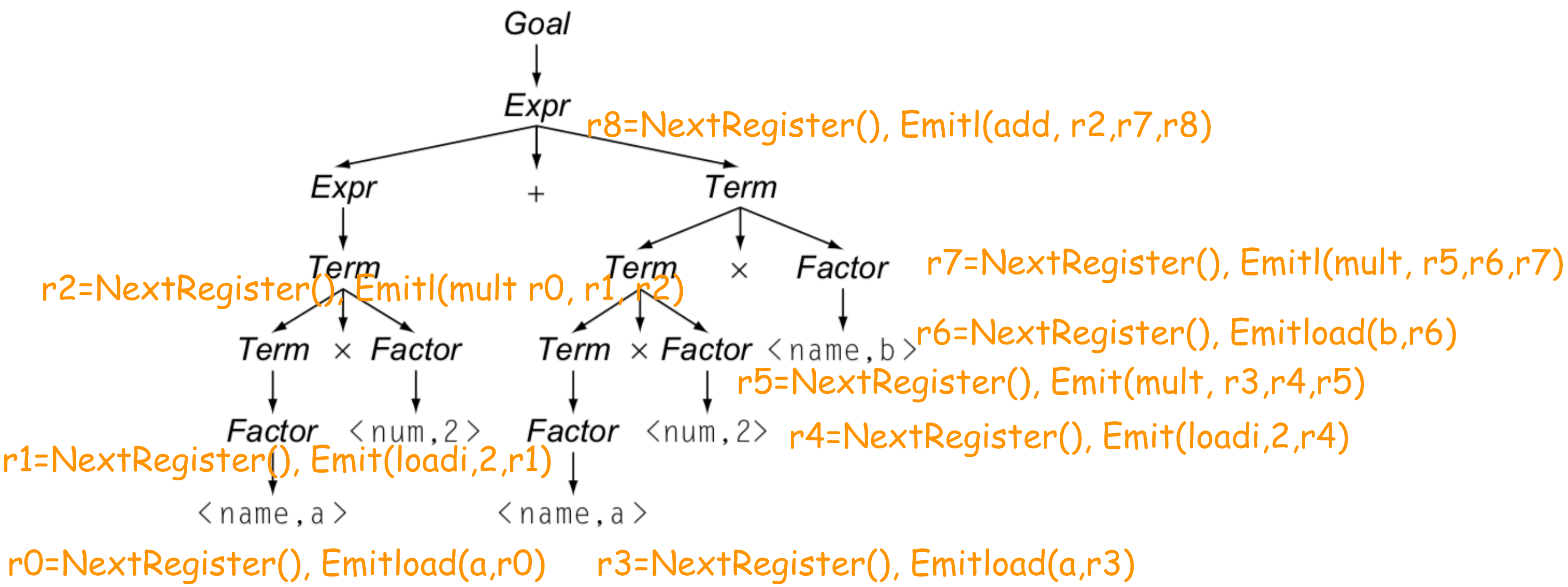
- Emit(loadi,n,r) → loadI(n,r) n->r

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*)	$$ = $2;
9		|	<u>number</u>	$$ = *NextRegister()*; *Emit*(loadi,Value(lexeme),$$);
10		|	<u>ident</u>	$$ = *NextRegister()*; *EmitLoad*(ident,$$);




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

From previous lectures

```
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;
```

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 {
          stack.popnum(3*|β|);      // pop 3*|β| symbols
/* insert case statement here computing $$ */
          s = stack.top();
          stack.push(A);             // push A
          stack.push($$);             // push $$
        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   (2rd 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 - 3(n-1)-1)
- Need to sequence rule applications
  - On every reduce action, perform the action rule
  - Add a giant case statement to the parser