Context-sensitive Analysis
Part III

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

$Block_0$	→	$Block_1$ *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)

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

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

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

$Block_0$	→	$Block_1$ *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)

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

$Block_0$	→	$Block_1$ *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)

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**

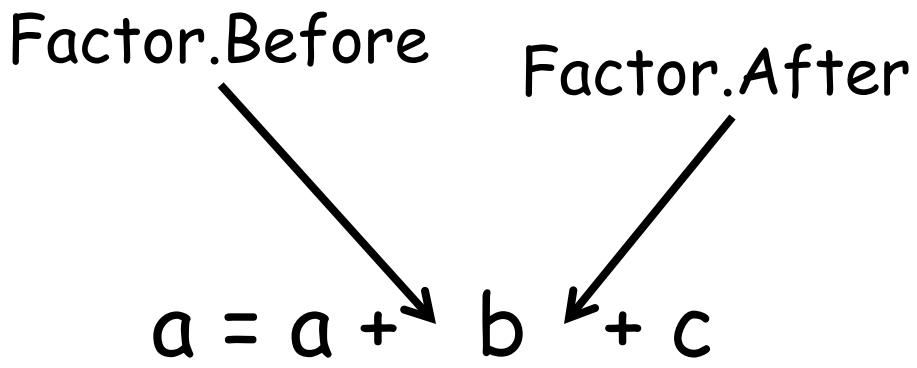
This looks more complex!

A Better Execution Model

Adding load tracking

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

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



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

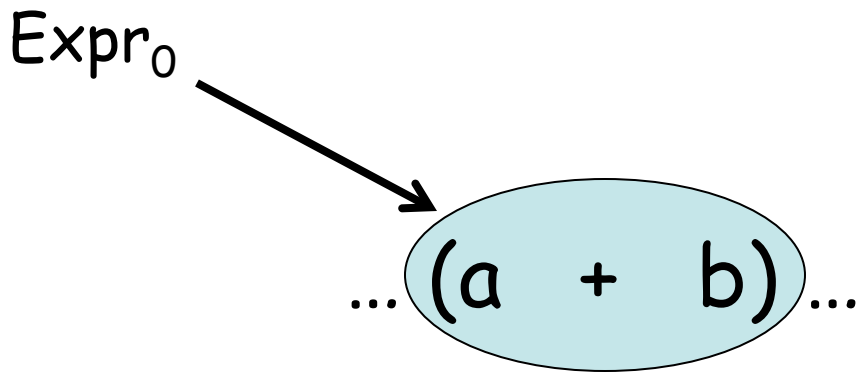
This looks more complex!

A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

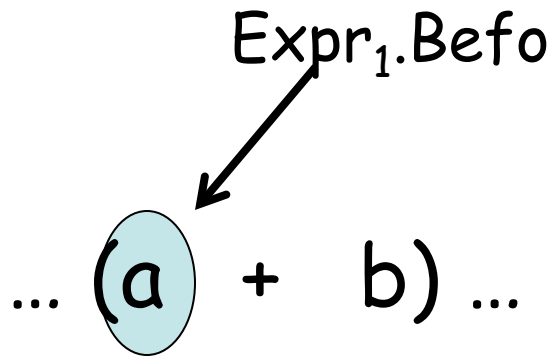


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

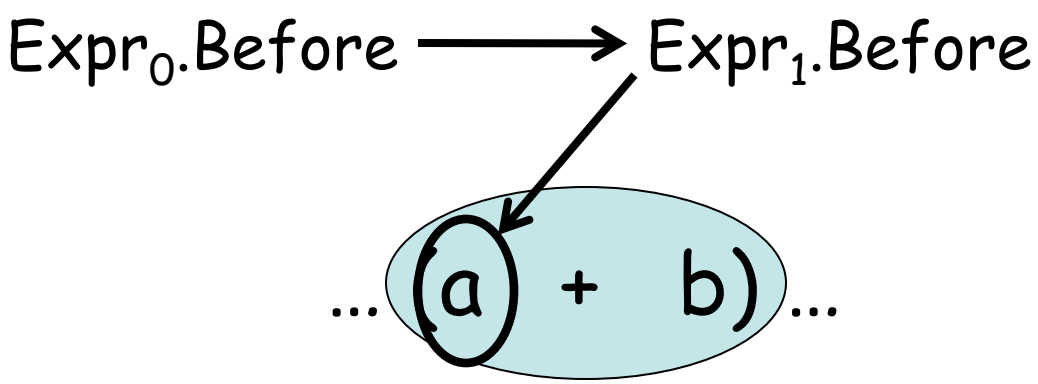


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

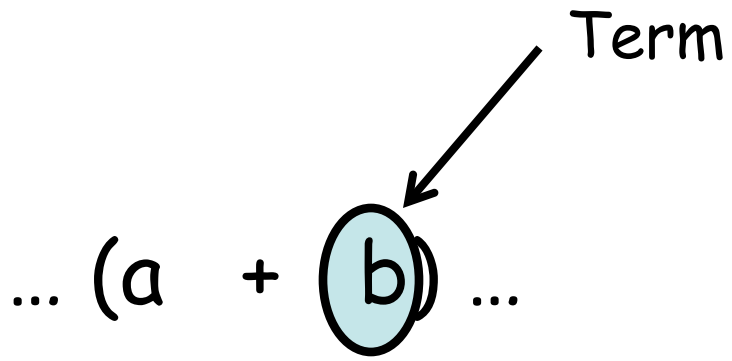


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

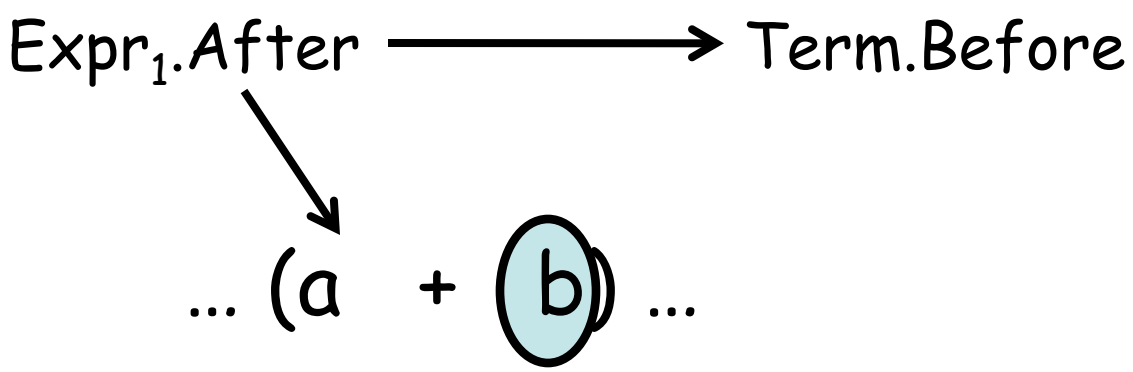


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

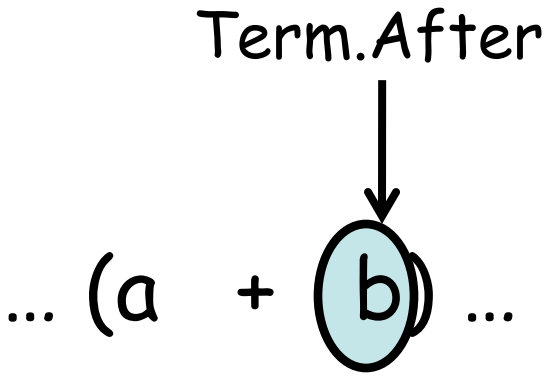


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

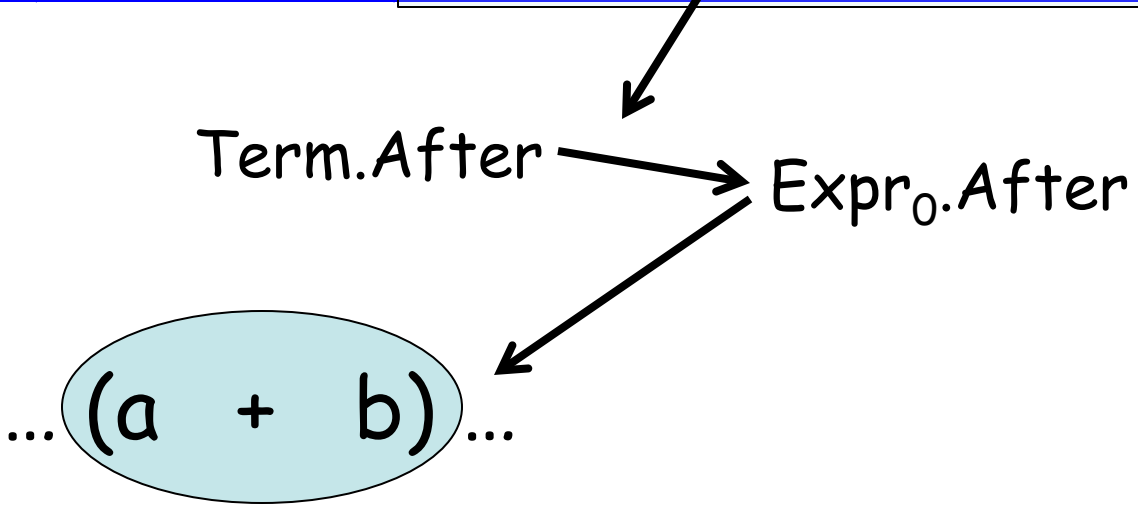


A Better Execution Model

- Load tracking adds complexity
- 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 ; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$



An Even Better Model

What about accounting for finite register sets?

- *Before* & *After* must be of limited size
- Adds complexity to *Factor→Identifier*
  - → *Needs to track size of Before/After sets*

The Moral of the Story

- Non-local computation needs lots of supporting rules
- Complex local computation was relatively easy

The Problems

- Copy rules increase complexity
  - → difficult to debug, maintain
- Copy rules increase space requirements
  - → Need copies of attributes

Context-sensitive Analysis

Part III

Ad-hoc syntax-directed translation, Symbol Tables, and Types

Addressing the Problem

If you gave the problem of estimating cycle counts to a competent junior or senior CS major, …

- Introduce a central repository for information
- Table of identifiers
  - → Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches

Addressing the Problem (cont'd)

- All inter-assignment attribute flow is through table
  - → Clean, efficient implementation
  - → Good techniques for implementing the table
  - → When it is done, information is in the table !
  - → Cures most of the problems

Unfortunately, this design violates the functional paradigm of an AG.

The Realist's Alternative

Ad-hoc syntax-directed translation

- Build on bottom-up, shift-reduce parser
- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allow arbitrary code provides complete flexibility

The Realist's Alternative (cont'd)

To make this work

- Need names for attributes of each symbol on *lhs* & *rhs*
  - → Typically, one attribute passed through parser + arbitrary code
  - → Yacc introduced **$$**, **$1**, **$2**, … **$n**, left to right
- Need an evaluation scheme
  - → Bottom-up evaluation works much of the time
  - → Fits nicely into **LR(1)** parsing algorithm

Reworking the Example (*with load tracking*)

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
				}

One missing detail: initializing cost

Reworking the Example (*with load tracking*)

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
				}

Reworking the Example (*with load tracking*)

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
			}

Reworking the Example (*with load tracking*)

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
			}

Much cleaner than the AG approach.
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…

As mentioned previously,
Yacc introduced **$$**, **$1**, **$2**, … **$n**, left to right
We can rewrite grammar using Yacc notation

Reworking the Example (*with load tracking*)

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

12		|	*Ident*	i ← hash(Ident);
				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 to initialize "cost"

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

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;

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

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

assumes table is global

Symbol Table: Typical Uses

- 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
    - Create list of sites to check

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (Somewhat) abstract names for symbols

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Differences

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