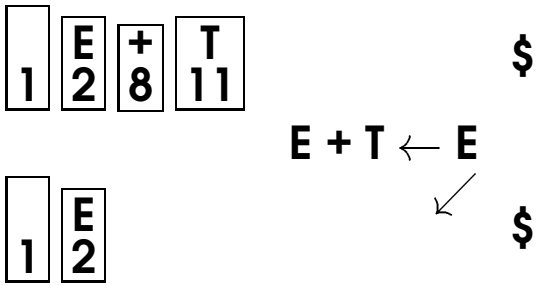
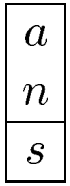
Semantic processing at parse time

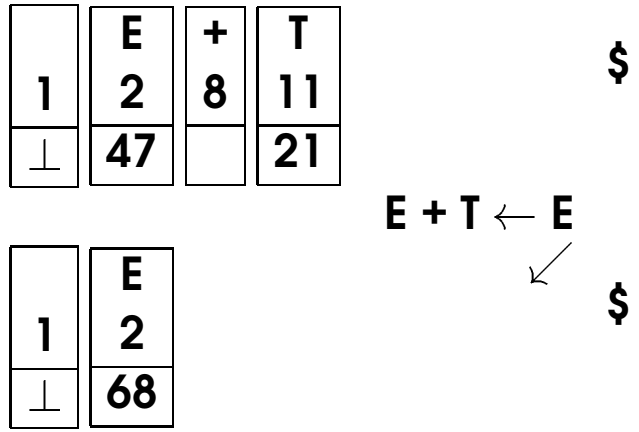
Recall how an LR parser uses a stack to apply reductions:



Our parse stack previously consisted of $\boxed{\begin{matrix}a\\n\end{matrix}}$, where a is a grammar symbol and n is a parse state. We now augment the stack to contain semantic information:



We can use this semantic stack to *synthesize* information during the parse. In our example, when E+T is reduced, we could add together the values associated with E and T, pushing the sum on the stack along with the E replacing the E+T:



In YACC, the grammar file can specify a union of types for the semantic stack, so that information of any form can be synthesized during the parse.

Copyright ©1994 Ron K. Cytron. All rights reserved

Synthesized attributes

With YACC, a segment of C code can be associated with each production. Given a rule

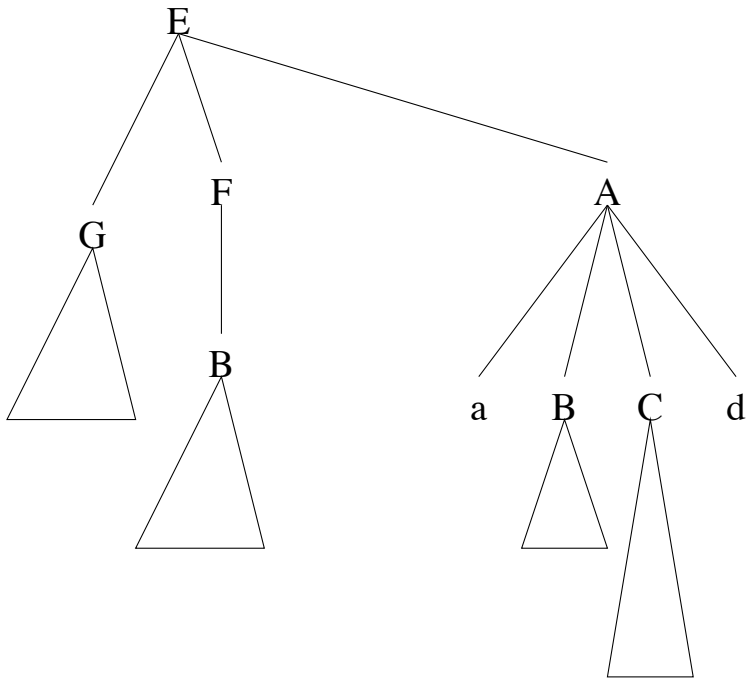
$$\mathbf{A} \rightarrow a_1\ a_2 \ldots a_k$$

the segment of C code can refer to the semantic stack values of symbols as follows:

Rule Symbol	Semantic Value
a_1	**$1**
a_2	**$2**
⋮	
a_k	**$k**
A	**$$**

so that a typical rule looks like

$$\mathbf{A} \rightarrow a_1\ a_2 \ldots a_k$$
`{$$ = $3 + f($2);}`



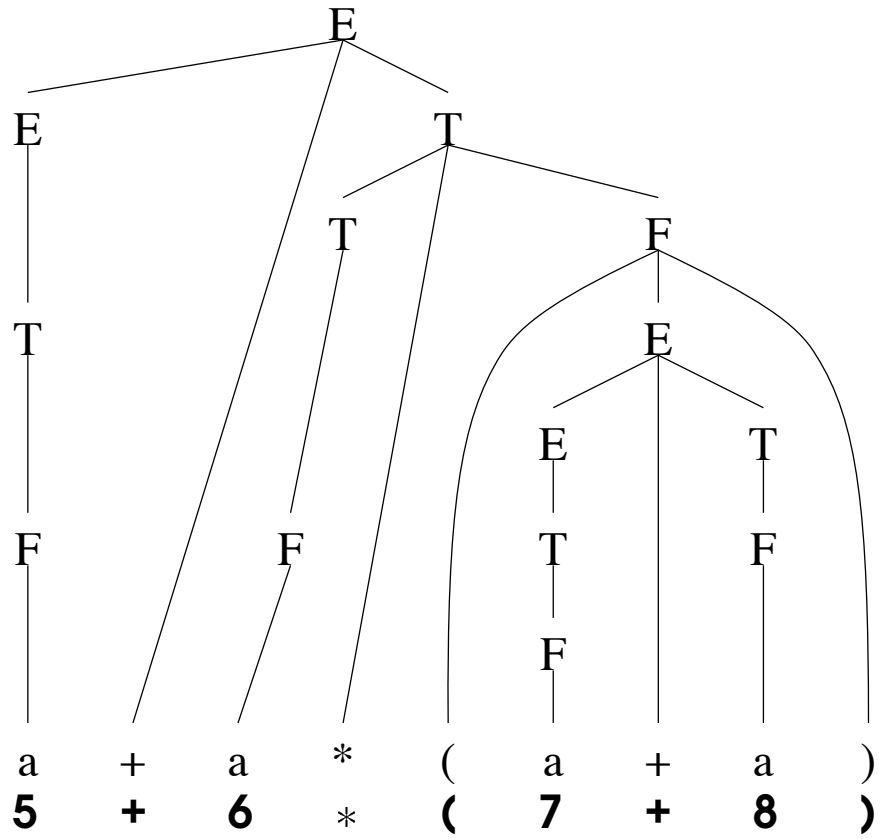
When "a B C d" is reduced to A, information previously contributed to B and C can be incorporated into the information synthesized for A. Without global storage, information computed at a tree node X is a pure function of information computed at X's descendents.

More generally, one can reference any value still on the stack. In our example,this include information associated with F and G. Such grammars are called *L-attributed*.

Copyright ©1994 Ron K. Cytron. All rights reserved

Evaluating infix expressions

```
S → E $
    {printf("Answer is %d\n",$1);}
E → E + T
    {$$ = $1 + $3;}
  | T
    {$$ = $1;}
T → T * F
    {$$ = $1 * $3;}
  | F
    {$$ = $1;}
F → ( E )
    {$$ = $2;}
  | const
    {$$ = $1;}
```



Notice how the unit productions ($A \rightarrow B$) lead to simple copying of values up the parse tree. While these rules participate in disambiguating the grammar, they are not always conducive to semantic processing.

Copyright ©1994 Ron K. Cytron. All rights reserved

Another example

Num → **D $**

```
{printf("Answer:  %d\n",$1);}
```

D → **D d**

```
{$$ = (10 * $1) + $2;}
```

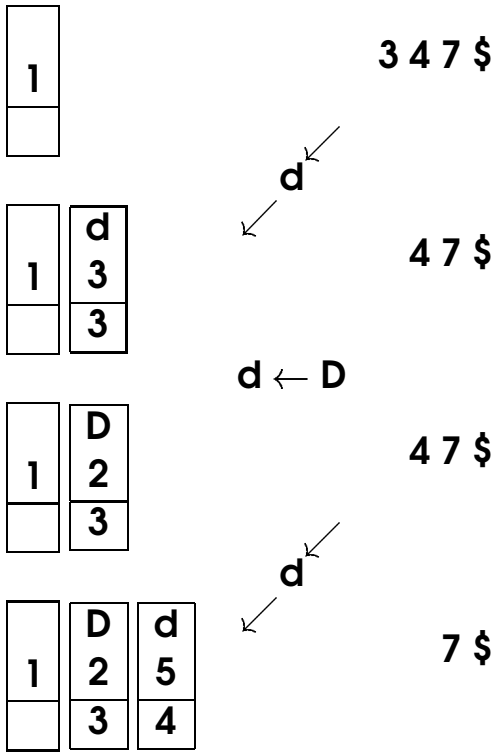
| **d**

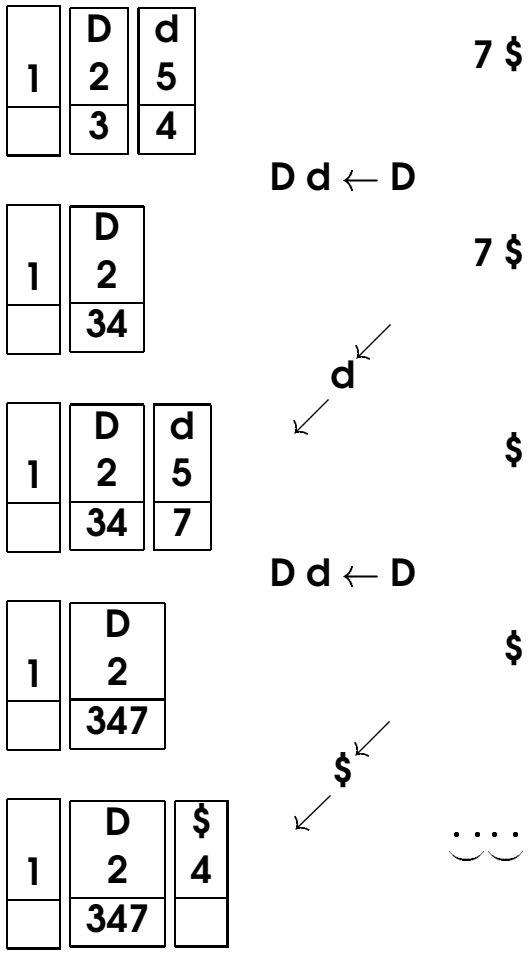
```
{$$ = $1;}
```

State	d	$	D
1	Goto State 3		Goto State 2
2	Goto State 5	Goto State 4	
3	Reduce by rule 1	·	Reduce by rule 1
4	☹	·	☹
5	Reduce by rule 2	·	Reduce by rule 2

Note: grammar is LR(0)

Applied to the string "347$"





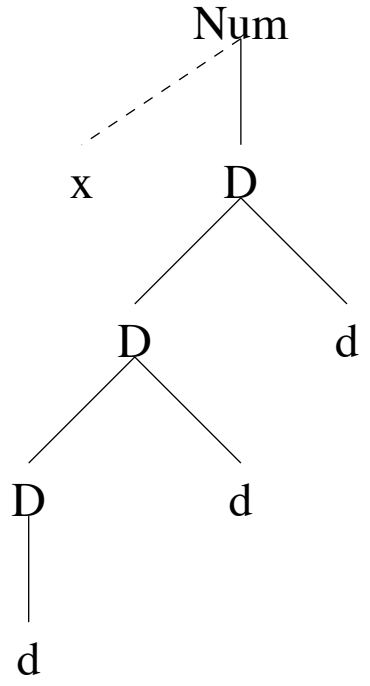
Copyright ©1994 Ron K. Cytron. All rights reserved

Grammars and semantic processing

There are usually many unambiguous grammars that generate a given programming language. In planning for semantic processing, it is often convenient to rewrite the grammar so that reductions and stack activity are conducive to the required actions.

Consider the grammar:

$$
\begin{array}{lcl}
\textbf{Num} & \rightarrow & \textbf{x D} \\
 & | & \textbf{D} \\
\textbf{D} & \rightarrow & \textbf{D d} \\
 & | & \textbf{d}
\end{array}
$$



Interpretation: a string of digits represents a base-10 number, unless the string is preceded by an 'x', in which case the string represents a base-8 number.

String	Number
3 4 7	347
x 3 4 7	231

We could compute the number by passing the list of digits up the tree, forming the answer at Num. We would prefer to compute the number *as we reduce the digits,* but this grammar's parse trees have the base information in the wrong place.

Copyright ©1994 Ron K. Cytron. All rights reserved

Rewriting the grammar

Num → **x OctD $**

```
{printf("Answer:  %d\n",$2)}
```

| **DecD $**

```
{printf("Answer:  %d\n",$1)}
```

DecD → **DecD d**

```
{$$ = (10 * $1) + $2;}
```

| **d**

```
{$$ = $1;}
```

OctD → **OctD d**

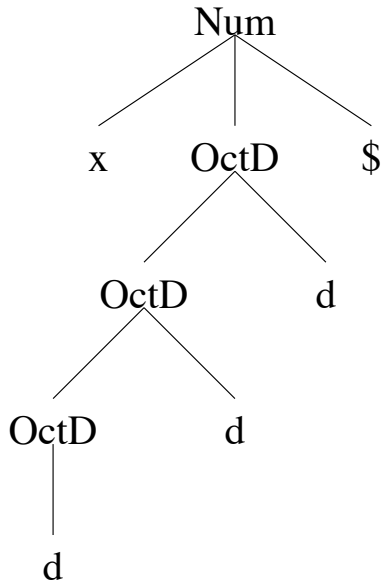
```
{$$ = (8 * $1) + $2;}
```

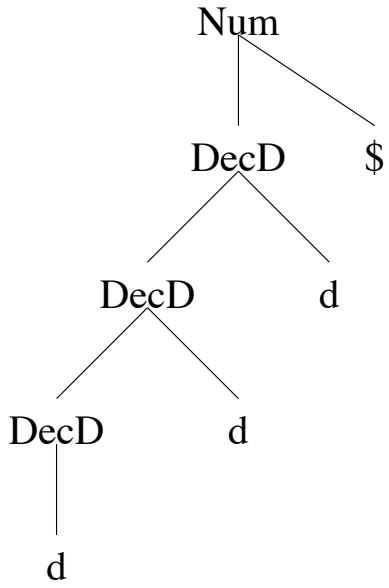
| **d**

```
{$$ = $1;}
```

State	d	x	$	DecD	OctD
1	Goto State 4	Goto State 2			
2	Goto State 6				Goto State 5
3	Goto State 8		Goto State 7		
4	Reduce by rule 4	. . . . .			Reduce by rule 4
5	Goto State 10		Goto State 9		
6	Reduce by rule 6	. . . . .			Reduce by rule 6
7	:)	. . . . .			:)
8	Reduce by rule 3	. . . . .			Reduce by rule 3
9	:)	. . . . .			:)
10	Reduce by rule 5	. . . . .			Reduce by rule 5

Note: grammar is LR(0)





Copyright ©1994 Ron K. Cytron. All rights reserved

Another change!

Suppose we want the base itself to be part of the input:

String	Number
3 4 7	347
x 8 3 4 7	231
x 9 3 4 7	286

One possibility is to use a global variable:

Num $\rightarrow$ **x B D $**

```
{printf("Answer:  %d\n",$3);}
```

| **Skip D $**

```
{printf("Answer:  %d\n",$2);}
```

B $\rightarrow$ **d**

```
{Base = $1;}
```

Skip $\rightarrow \lambda$

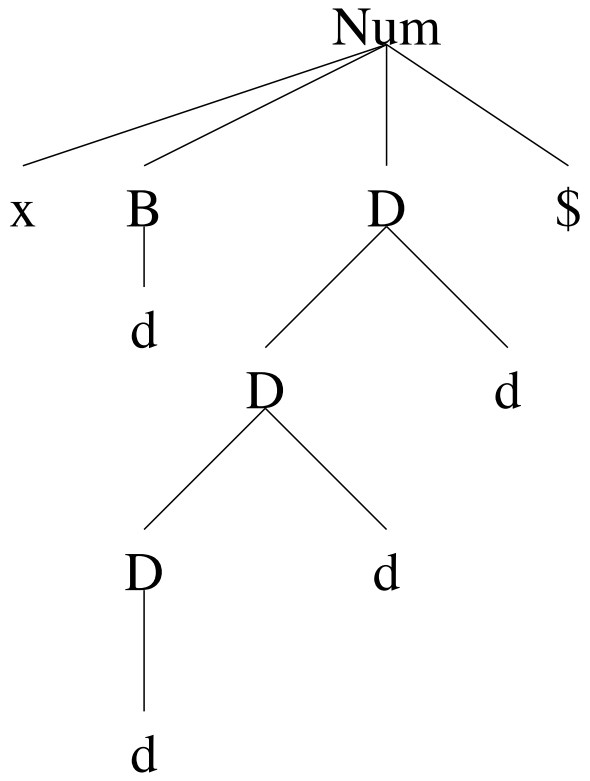
```
{Base = 10;}
```

D $\rightarrow$ **D d**

```
{$$ = (Base * $1) + $2;}
```

| **d**

```
{$$ = $1;}
```



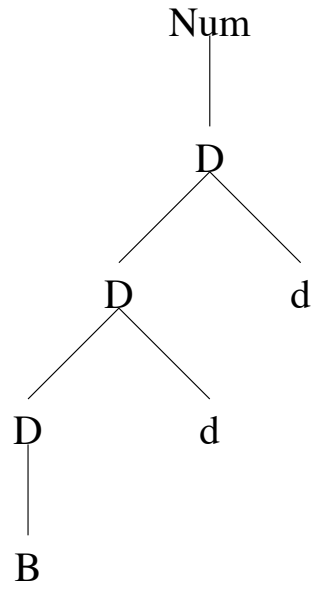
Note that the reduction B $\rightarrow$ d is necessary to set the global variable. In the LR parse, this is the first reduction, so the base will indeed be set when the first D $\rightarrow$ D d rule is applied. But global variables are not very clean, especially if constructs could be nested so that global variables get overwritten.

Copyright ©1994 Ron K. Cytron. All rights reserved

Arriving at a good grammar

Let's engineer the tree we would like to see, and then construct the appropriate grammar. In the tree shown to the right, it's possible to synthesize the base up the tree.

At each reduction, we could know how to compute the new value to pass up the tree.



Num $\rightarrow$ D \$

```
{printf("Answer:  %d\n",$1.value);}
```

D $\rightarrow$ D d

```
{$$.value = ($1.base × $1.value) + $2;
$$.base = $1.base;}
```

| B

```
{$$.base = $1;}
```

B $\rightarrow$ x d

```
{$$ = $2;}
```

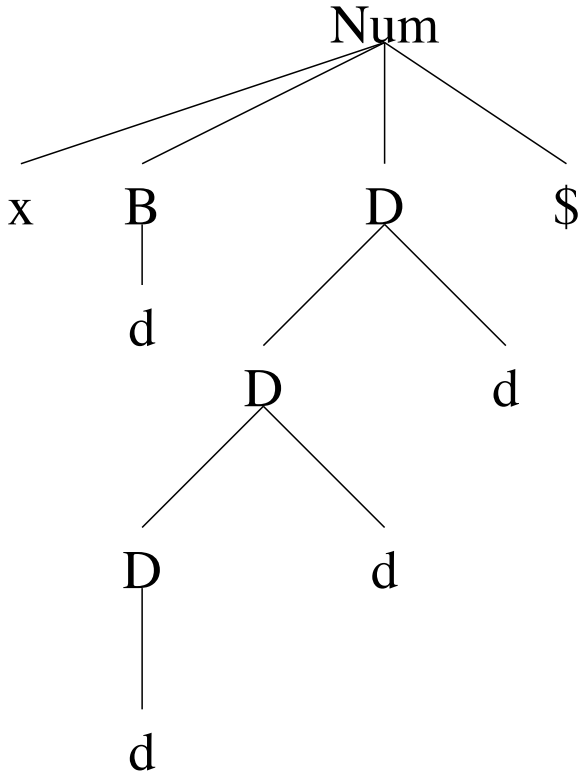
| λ

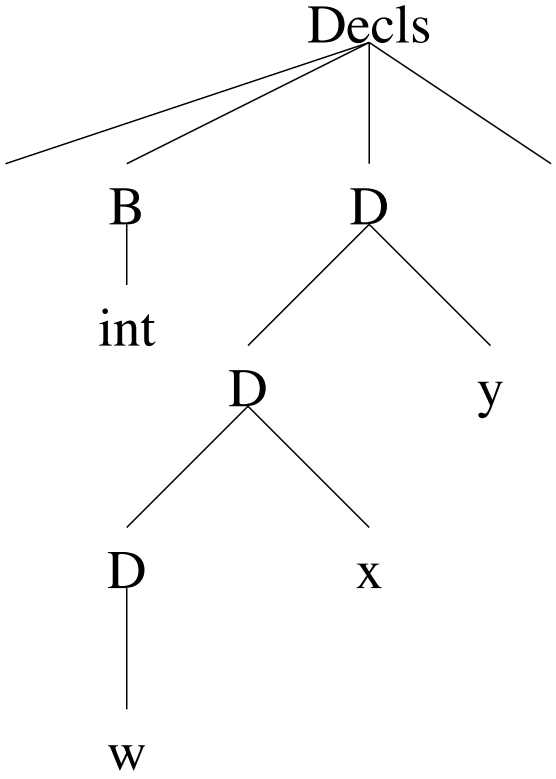
```
{$$ = 10;}
```

Copyright ©1994 Ron K. Cytron. All rights reserved

Back to declarations

The running example of converting a string of digits to a number is actually an abstraction of processing variable declarations in C**.**





We would like to enter the variables in the symbol table, along with their types, as we parse the input. Rewriting the ANSI C grammar to accomplish this is a good exercise.

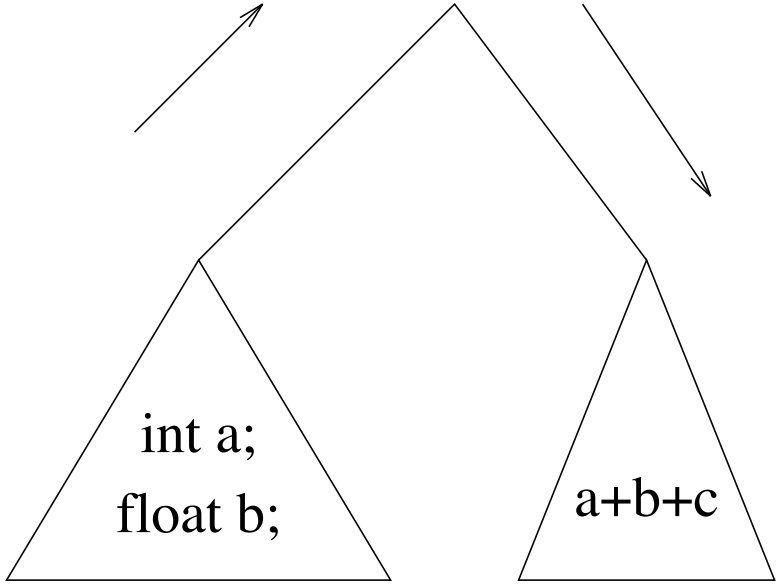
Note that PASCAL has its type information at the end, and so a right recursive rule can similarly accommodate that form of syntax.

Copyright ©1994 Ron K. Cytron. All rights reserved

Back to symbol tables

An *attributed grammar* allows semantic equations whose terms depend on synthesized and *inherited* attributes. A classical use of attribute grammar systems is for the synthesis and use of type information.

As the declarations are parsed, a "symbol table" is synthesized up the parse tree. While processing the code of a procedure, this symbol and those from outer scopes are available as an inherited attribute.



While attribute grammars offer a clean mechanism for expressing semantics, such systems are usually slower than those involving only synthesized attributes, and one must still get the equations "right". The Cornell Program Synthesizer is a popular and robust system for developing compilers based on attribute grammars (41, 31, 32).

Copyright ©1994 Ron K. Cytron. All rights reserved