

# Chapter 7 Syntax-Directed Translation

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: March 25, 2010

current version: January 25, 2011

©January 25, 2011 by Wuu Yang. All rights reserved.

## Chapter outline: Syntax-Directed Translation

1. Overview
2. Bottom-up syntax-directed translation
3. Top-down syntax-directed translation
4. Abstract syntax trees
5. AST design and construction
6. AST structures for left and right values
7. Design patterns for AST

### References:

- Chris Lattner and Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004.
- Gal, Franz, and Probst, Java Bytecode Verification via Static

Single Assignment Form, ACM Transactions on Programming Languages and Systems, October 2007.

- dynamic dispatch in Wikipedia.
- double dispatch in Wikipedia.<sup>a</sup>

---

<sup>a</sup>Double dispatch and reflection are not related to compiler construction. However, the textbook uses them in their algorithm.

## §7.0 double dispatch in Java

```
class foo {
    public static void main(String[] args) {
        Pvisitor v1;          node n1;
        v1 = new Rvisitor();  n1 = new Anode();
        System.out.println("1st try"); v1.visit(n1);
        System.out.println("2nd try"); n1.accept(v1);
    }
}
```

```
class node {
    void accept(Pvisitor p1) { p1.visit(this); }
}
class Anode extends node {
    void accept(Pvisitor p1) { p1.visit(this); }
}
```

```
class visitor {
    void visit(node nn) { System.out.println(10); }
}
class Pvisitor extends visitor {
    void visit(Anode an)    { System.out.println(1); }
    // void visit(node an) { System.out.println(3); }
}
class Rvisitor extends Pvisitor {
    void visit(Anode an)    { System.out.println(5); }
    // void visit(node an)  { System.out.println(7); }
}
```

Output is “1st try; 10; 2nd try; 5”.

## §7.1 Overview

The parser identifies the syntactic structure of a program (parsing). The compiler needs to perform *translation*. There are two approaches of translation:

1. Perform translation immediately after identifying each individual syntactic structure. That is, translation actions are attached to the production rules. This approach is called *syntax-directed translation*.
2. Build an internal representation of the program during parsing. Later the compiler examines the internal representation to generate target code. The internal representation is usually the *abstract syntax tree* (AST).

Actually building the AST is done in a syntax-directed fashion.

### §7.1.1 Semantic actions and values

When a production rule is applied (the reduce action in the LR parser and the expand action in the LL parser) during parsing, a syntactic structure in the input program is recognized. At this moment, the parser could perform some actions (which are called *semantic actions*). That is, a semantic action is attached to each production rule. A semantic action could be a null action (doing nothing) or it could perform a lengthy sequence of code. In yacc, we could do as follows:

```
PackageDeclaration : RW_PACKAGE PackageSpecOrBody SEMI
                    { $$ = mkUnyTree(Package_Declaration, $2); }
                    ;
```

Here `mkUnyTree` is a user-defined function.

There is a *semantic value* associated with each (terminal and nonterminal) symbol. Sometimes, a semantic value is trivial. For instance, the ; symbol has no serious meaning. Its semantic value can be ignored. Sometimes, a semantic value could be very complex, such as a large record consisting of many fields or pointers to other data structures. For the sake of uniformity (and static checking in the language for writing the compiler), we may simply assume every symbol has a uniform semantic value.

The semantic actions compute the new semantic values from known semantic values. In yacc, the \$\$ symbol denotes the semantic value of the left-hand-side nonterminal and the \$2 symbol denotes the semantic value of the 2nd symbol on the right-hand side of the production rule.

In order to compute the semantic values, a compiler writer frequently re-structures the grammar to aid the computation.

## §7.1.2 Synthesized and inherited attributes

For the sake of simplicity, we may break a complex semantic value into several, smaller pieces. We may imagine a semantic value as a structure (or an object) and each piece as a field in the structure.

Each piece is called an *attribute* of the symbol. There are two kinds of attributes:

- inherited attributes
- synthesized attributes

Semantic actions are used to specify the computation of the attributes. More general actions, such as `print`, `search`, etc., can also be specified in the semantic actions. The following table shows an example of attributes. In the example, each (terminal and nonterminal) symbol has one attribute (or semantic value) *val*. We use the notation  $E.val$  to denote the *val* attribute of the  $E$  symbol. The attributes of  $+$ ,  $*$ ,  $($ , and  $)$  are useless in the above example

and are omitted.

A grammar together with a set of semantic actions is called an *attribute grammar*.

	Production	Semantic actions
$E$	$\rightarrow E_1 + T$	$\{E.val := E_1.val + T.val\}$
$E$	$\rightarrow T$	$\{E.val := T.val\}$
$T$	$\rightarrow T_1 * F$	$\{T.val := T_1.val * F.val\}$
$T$	$\rightarrow F$	$\{T.val := F.val\}$
$F$	$\rightarrow ( E )$	$\{F.val := E.val\}$
$F$	$\rightarrow intliteral$	$\{F.val := intliteral.val\}$

A closer look at the above semantic actions reveals that, in each production, we use attributes of the right-hand-side symbols to compute the attributes of the left-hand-side nonterminal. Such

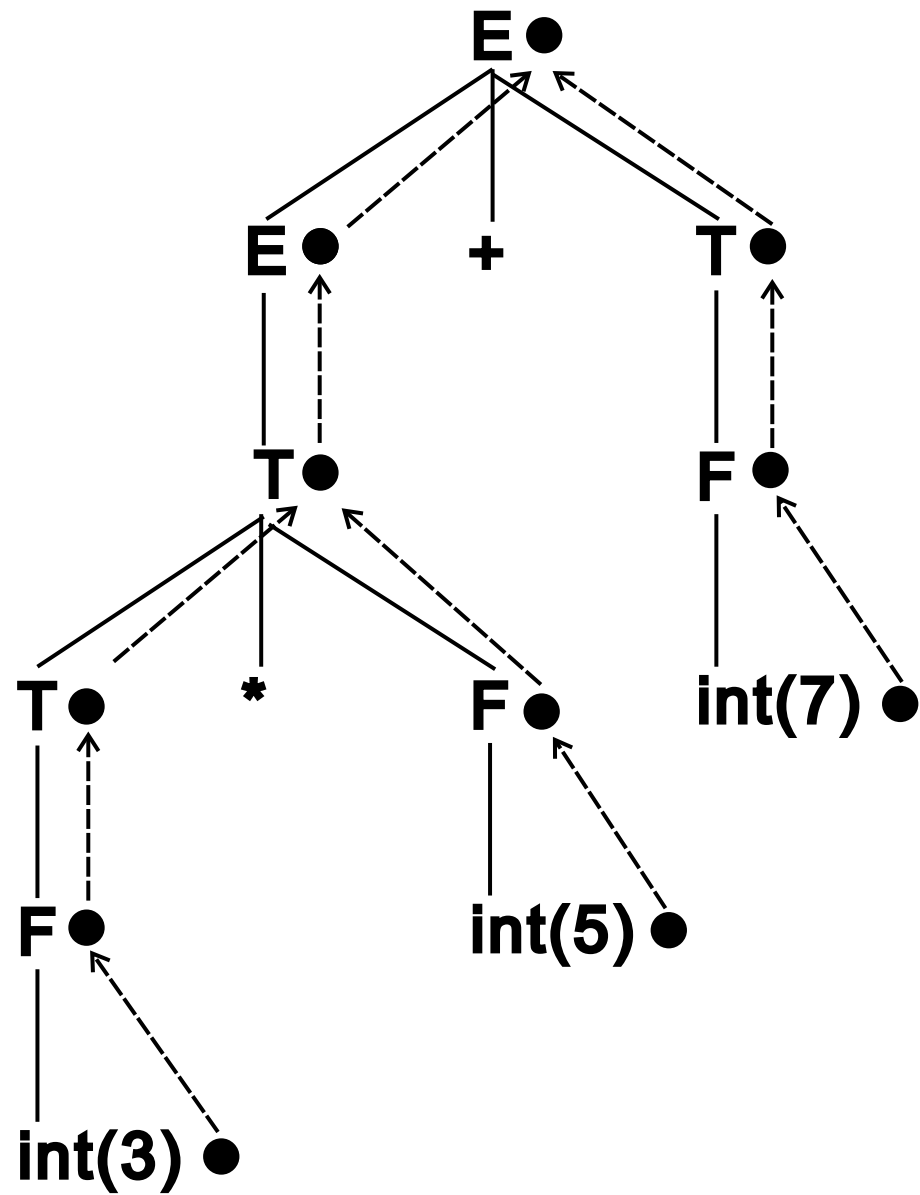
attributes are classified as *synthesized* attributes.

Conversely, we may use attributes of the left-hand-side nonterminal to compute the attributes of the right-hand-side symbols. Such attributes are classified as *inherited* attributes.

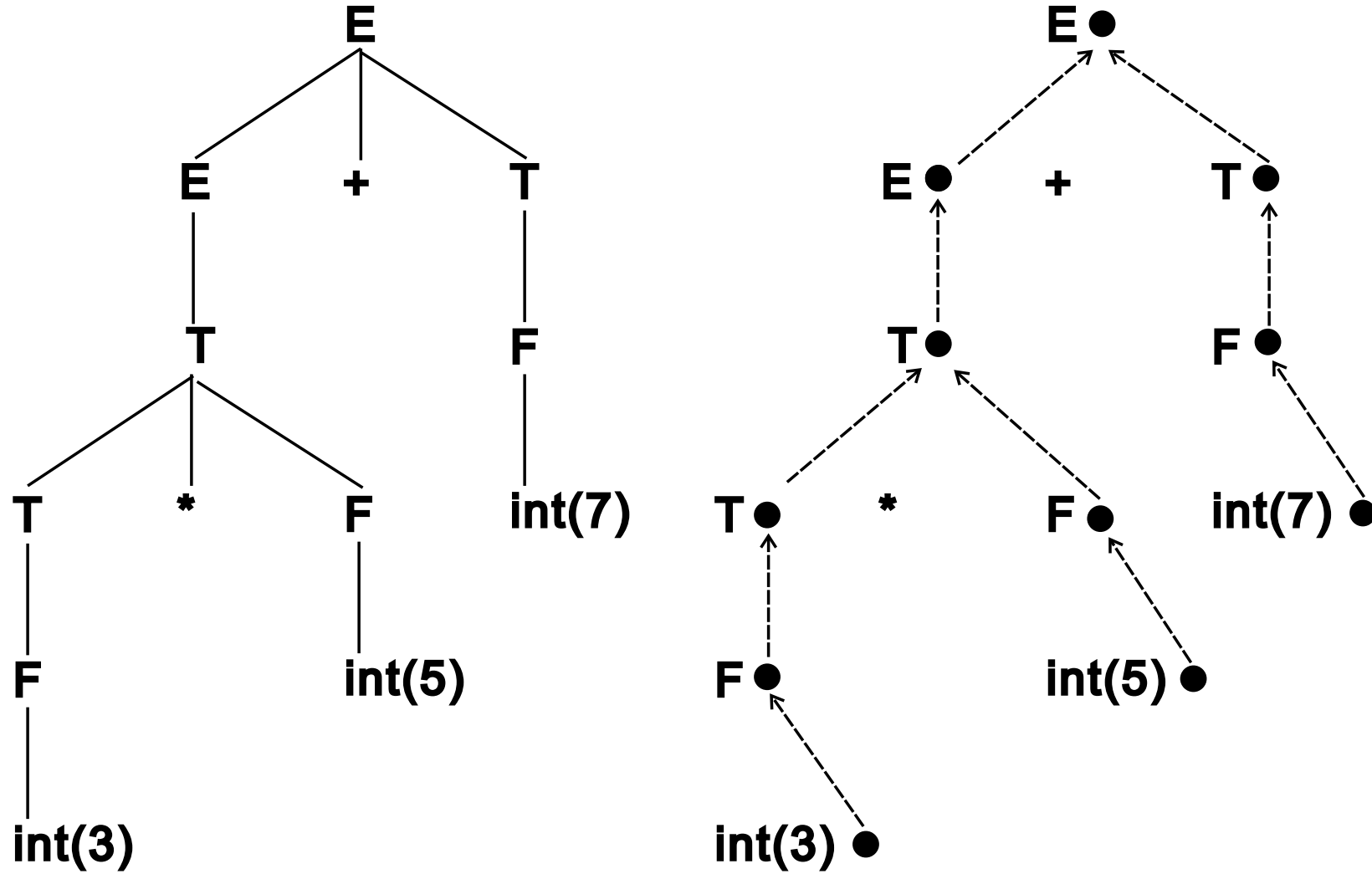
If an attribute grammar contains only synthesized attributes, it is called an *S-attributed grammar*. For ASTs derived from an S-attributed grammar, the attributes can always be evaluated in a bottom-up traversal. Due to this bottom-up nature, the attributes can be evaluated during an LR parsing.

If we draw the graph for the sentence

$$3 * 5 + 7$$



We may separate the the parse tree and attribute graph.





We will see the direction of computation is all upward, that is, from leaves to the root. The “direction of computation” shows the *dependence* among the attributes. When a compiler evaluates the attributes, it should follow an order that conforms to the direction of computation.

In other cases, the dependence could be downward, that is, from the root to the leaves. In even more general cases, a syntax tree could contain a mixture of upward and downward dependences.

Syntax-directed translation of common programming languages usually requires both downward and upward dependences. (See an example later.) If an LR parser is used, we can handle the downward dependences with additional techniques, say with a global symbol table. The upward dependences are handled naturally by the parser.

Conversely, if an LL parser is used, the upward dependences can be

handled with additional techniques.

In the more general cases, the parser can build the (abstract) syntax tree. Later a separate evaluator program will traverse the syntax tree and evaluate the attributes.

**Example.** Figure 7.1 shows another example of synthesized attributes.

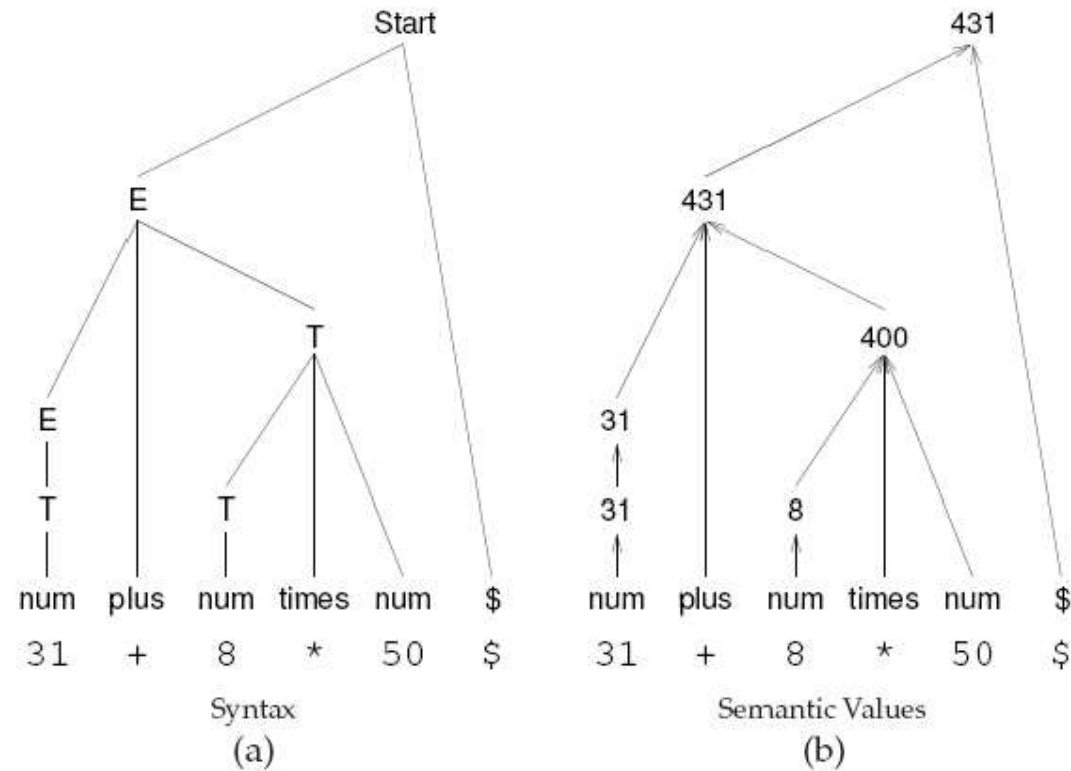


Figure 7.1: (a) Parse tree for the displayed expression;  
 (b) Synthesized attributes transmit values up the parse tree toward the root.

**Example.** Figure 7.2 shows an example of inherited attributes. This example counts the number of A's.

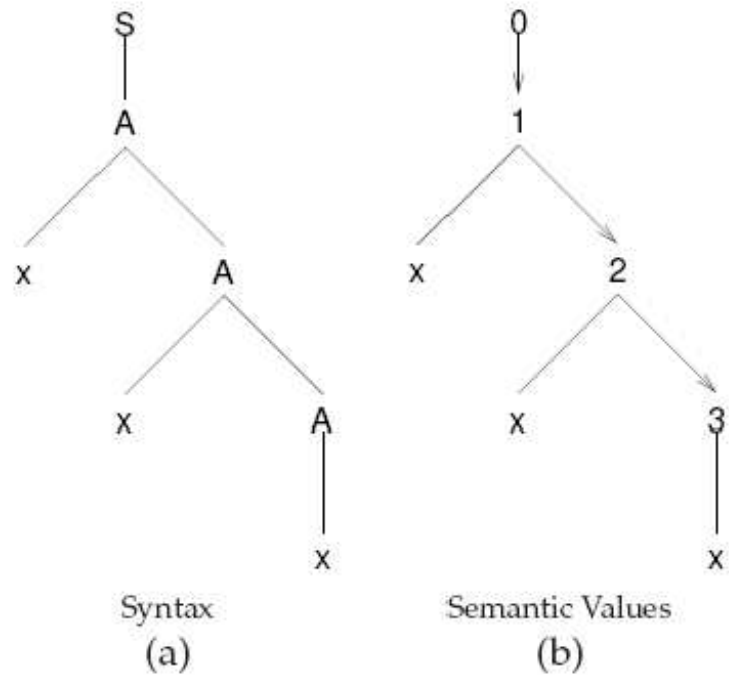


Figure 7.2: (a) Parse tree for the displayed input string; (b) Inherited attributes pass from parent to child.

---

## §7.2 Bottom-up syntax-directed translation

Consider an LR parser. When the parser is about to reduce with a production, say

$$A \rightarrow X_1 X_2 \dots X_n$$

The top of the parse stack contains  $n$  symbols (i.e.,  $X_1, X_2, \dots, X_n$ ) together with their (already evaluated) attributes. The parser will compute the attributes of  $A$  from those of  $X_1, X_2, \dots, X_n$  and then pops off the  $n$  symbols and finally pushes the nonterminal  $A$  (with its evaluated attributes) onto the run-time stack.

In a bottom-up translation, the semantic values of the leaves are established by the scanner.

We may imagine there are two stacks: the parse stack (for terminals and nonterminals) and the semantic stack (for semantic values).

The two stacks can actually be combined and managed together.

## §7.2.1 Example

	Production	Semantic actions
$Start$	$\rightarrow Digs \$$	$\{ print(Digs.val) \}$
$Digs$	$\rightarrow Digs_1 d$	$\{ Digs.val := Digs_1.val * 10 + d.val \}$
$Digs$	$\rightarrow d$	$\{ Digs.val := d.val \}$

Figure 7.3 is the syntax tree for the sentence.

4 3 1 \$

In Figure 7.3, all semantic values have the integer type. A parser generator, such as yacc, allows the compiler writer to declare the types of semantic values.

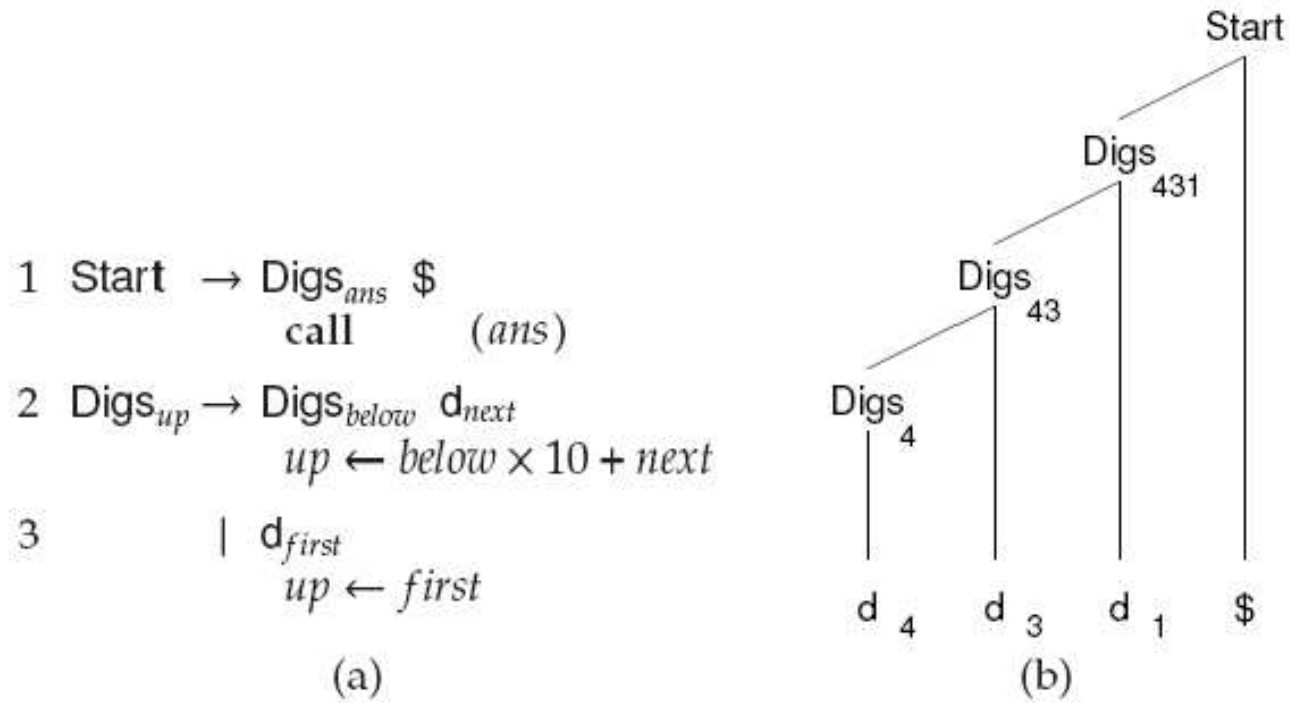


Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input 4 3 1 \$.

---

*Example.* Suppose we want to extend the above grammar to handle both decimal and octal numbers. An octal number starts with the character *o*.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow$ <i>Num</i> \$	{ <i>print</i> ( <i>Num.val</i> ) }
2. <i>Num</i>	$\rightarrow$ <i>o Digs</i>	{ <i>Num.val</i> := <i>Digs.val</i> }
3. <i>Num</i>	$\rightarrow$ <i>Digs</i>	{ <i>Num.val</i> := <i>Digs.val</i> }
4. <i>Digs</i>	$\rightarrow$ <i>Digs</i> <sub>1</sub> <i>d</i>	{ <i>Digs.val</i> := <i>Digs</i> <sub>1</sub> . <i>val</i> * 10 + <i>d.val</i> }
5. <i>Digs</i>	$\rightarrow$ <i>d</i>	{ <i>Digs.val</i> := <i>d.val</i> }

Figure 7.4 shows an octal number “o 4 3 1 \$”.

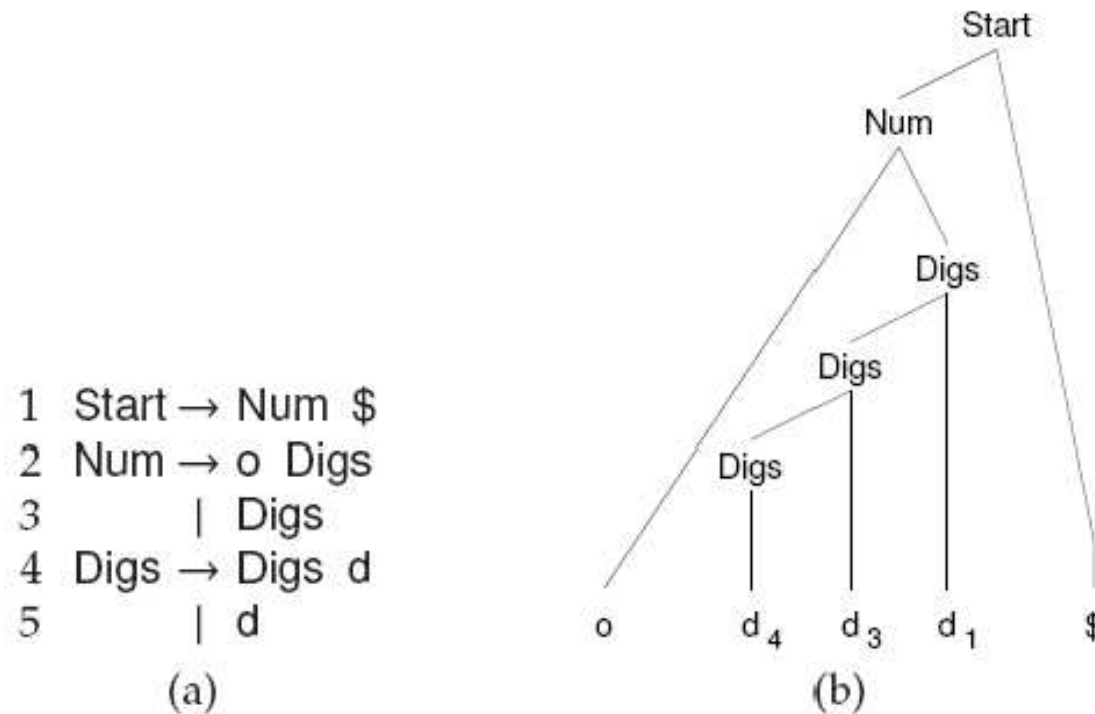


Figure 7.4: (a) Grammar and (b) parse tree for the input o 4 3 1 \$.

---

There is a problem with the semantic action in rule 4. We should use the constant 10 for decimal numbers and 8 for octal numbers.

The problem is because rules 4 and 5 are shared for both decimal and octal numbers. We can modify the grammar and/or the semantic actions for this kind of problem.

## §7.2.2 Rule cloning

One solution is to clone rules 4 and 5 for octal numbers.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow$ <i>Num</i> \$	{ <i>print</i> ( <i>Num.val</i> )}
2. <i>Num</i>	$\rightarrow$ <i>o OctDigs</i>	{ <i>Num.val</i> := <i>OctDigs.val</i> }
3. <i>Num</i>	$\rightarrow$ <i>DecDigs</i>	{ <i>Num.val</i> := <i>DecDigs.val</i> }
4. <i>DecDigs</i>	$\rightarrow$ <i>DecDigs</i> <sub>1</sub> <i>d</i>	{ <i>DecDigs.val</i> := <i>DecDigs</i> <sub>1</sub> . <i>val</i> * 10 + <i>d.val</i> }
5. <i>DecDigs</i>	$\rightarrow$ <i>d</i>	{ <i>DecDigs.val</i> := <i>d.val</i> }
6. <i>OctDigs</i>	$\rightarrow$ <i>OctDigs</i> <sub>1</sub> <i>d</i>	{ <b>if</b> <i>d</i> ≥ 8 <b>then</b> <i>error</i> () <b>else</b> <i>OctDigs.val</i> := <i>OctDigs</i> <sub>1</sub> . <i>val</i> * 8 + <i>d.val</i> }
7. <i>OctDigs</i>	$\rightarrow$ <i>d</i>	{ <b>if</b> <i>d</i> ≥ 8 <b>then</b> <i>error</i> () <b>else</b> <i>OctDigs.val</i> := <i>d.val</i> }

### §7.2.3 Forcing semantic actions

An alternative is to use a **global variable** *base* in the computation. *base* is set properly when seeing/skipping the *o* marker. We need to introduce a rule in order to add a semantic action to set up the *base* variable.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow$ <i>Num</i> \$	$\{print(Num.val)\}$
2. <i>Num</i>	$\rightarrow$ <i>SignalOct Digs</i>	$\{Num.val := Digs.val\}$
3. <i>Num</i>	$\rightarrow$ <i>SignalDec Digs</i>	$\{Num.val := Digs.val\}$
4. <i>SignalOct</i>	$\rightarrow$ <i>o</i>	$\{base := 8\}$
5. <i>SignalDec</i>	$\rightarrow$	$\{base := 10\}$
6. <i>Digs</i>	$\rightarrow$ <i>Digs</i> <sub>1</sub> <i>d</i>	$\{Digs.val := Digs_1.val * base + d.val\}$
7. <i>Digs</i>	$\rightarrow$ <i>d</i>	$\{Digs.val := d.val\}$

This method avoids cloning production rules. We can extend the above grammar even further to accommodate other bases. See Figure 7.7 below.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow$ <i>Num</i> \$	{ <i>print(Num.val)</i> }
2. <i>Num</i>	$\rightarrow$ <i>x SetBase Digs</i>	{ <i>Num.val := Digs.val</i> }
3. <i>Num</i>	$\rightarrow$ <i>BaseTen Digs</i>	{ <i>Num.val := Digs.val</i> }
4. <i>SetBase</i>	$\rightarrow$ <i>d</i>	{ <i>base := d.val</i> }
5. <i>BaseTen</i>	$\rightarrow$	{ <i>base := 10</i> }
6. <i>Digs</i>	$\rightarrow$ <i>Digs<sub>1</sub> d</i>	{ <b>if</b> <i>d.val</i> $\geq$ <i>base</i> <b>then</b> <i>error()</i> <b>else</b> <i>Digs.val := Digs<sub>1</sub>.val * base + d.val</i> }
7. <i>Digs</i>	$\rightarrow$ <i>d</i>	{ <b>if</b> <i>d.val</i> $\geq$ <i>base</i> <b>then</b> <i>error()</i> <b>else</b> <i>Digs.val := d.val</i> }

With the grammar in Figure 7.7, we have the following numbers:

input	meaning	value (base 10)
4 3 1 \$	$431_{10}$	431
x 8 4 3 1 \$	$431_8$	281
x 5 4 3 1 \$	$431_5$	116

```

1 Start      → Numans $
               call      (ans)

2 Numans    → x SetBase Digsbaseans
               ans ← baseans

3           | SetBaseTen Digsdecans
               ans ← decans

4 SetBase    → dval
               base ← val

5 SetBaseTen → λ
               base ← 10

6 Digsup    → Digsbelow dnext
               if next ≥ base
               then      ("Digit outside allowable range")
               up ← below × base + next
                                                    ③

7           | dfirst
               if first ≥ base
               then      ("Digit outside allowable range")
               up ← first
                                                    ④

```

Figure 7.7: Strings with an optionally specified base.

## §7.2.4 Aggressive grammar restructuring

In the above attribute grammar, we used a global variable *base*. We can replace global variables with appropriate attributes. See Figure 7.8. In this example, the semantic value of *Digs* consists of two attributes: *b* and *val*. The two attributes can be implemented as a single value with a `struct` in the C language.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow Digs \$$	$\{print(Digs.val)\}$
2. <i>Digs</i>	$\rightarrow Digs_1 d$	$\{Digs.b := Digs_1.b$ $Digs.val := Digs_1.val * Digs_1.b + d.val\}$
3. <i>Digs</i>	$\rightarrow SetBase$	$\{Digs.b := SetBase.b$ $Digs.val := 0\}$
4. <i>SetBase</i>	$\rightarrow$	$\{SetBase.b := 10\}$
5. <i>SetBase</i>	$\rightarrow x d$	$\{SetBase.b := d.val\}$

Rule 3 should be changed to

$$Digs \rightarrow SetBase\ d$$

Otherwise we will have strings such as “x 5 \$”.

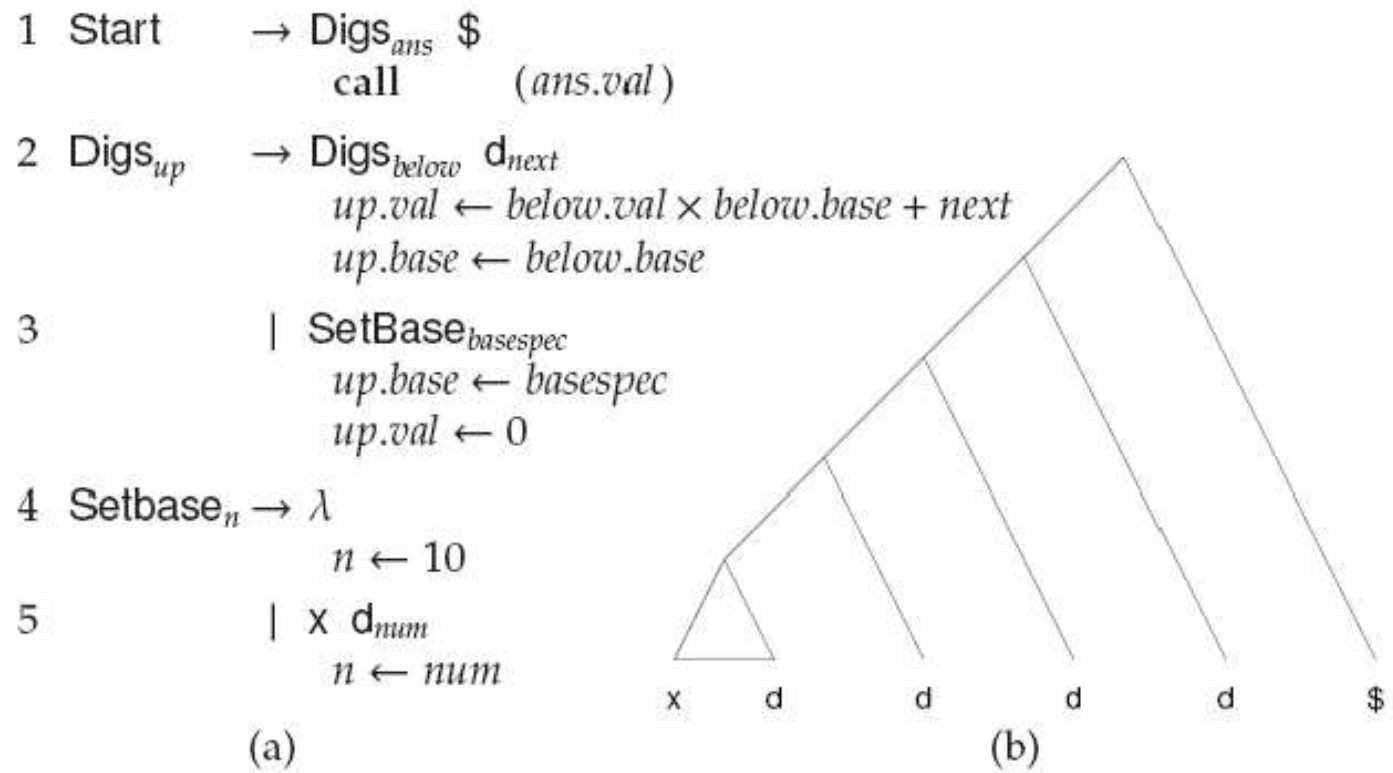


Figure 7.8: (a) Grammar that avoids global variables; (b) Parse tree reorganized to facilitate bottom-up attribute propagation.

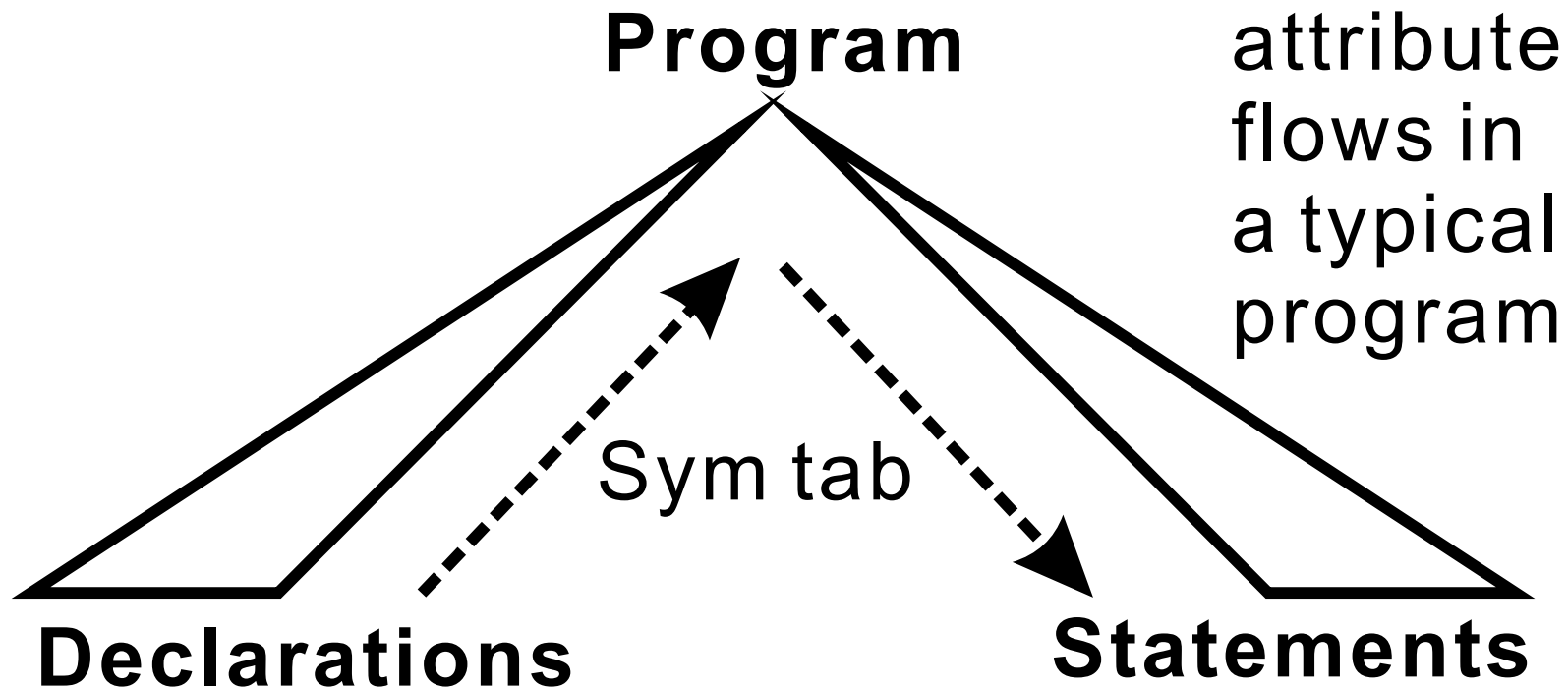
## Use inherited attributes

It is more natural to use an *inherited* attribute in this example.

Here  $b$  is an inherited attribute.

	Production	Semantic actions
1. <i>Start</i>	$\rightarrow$ <i>SetBase Digs</i> \$	$\{Digs.b := SetBase.b\}$ $\{print(Digs.val)\}$
2. <i>Digs</i>	$\rightarrow$ <i>Digs</i> <sub>1</sub> $d$	$\{Digs_1.b := Digs.b\}$ $\{\mathbf{if} \ d.val \geq Digs.b \ \mathbf{then} \ error()\}$ $\mathbf{else} \ Digs.val := Digs_1.val * Digs.b + d.val\}$
3. <i>Digs</i>	$\rightarrow$ $d$	$\{\mathbf{if} \ d.val \geq Digs.b \ \mathbf{then} \ error()\}$ $\mathbf{else} \ Digs.val := d.val\}$
4. <i>SetBase</i>	$\rightarrow$	$\{SetBase.b := 10\}$
5. <i>SetBase</i>	$\rightarrow$ $x \ d$	$\{SetBase.b := d.val\}$

This grammar includes both synthesized and inherited attributes. It is typical in a conventional program, which includes declarations and statements.



## §7.3 Top-down syntax-directed translation

```
1 Start  → Value $
2 Value  → num
3         | lparen Expr rparen
4 Expr   → plus Value Value
5         | prod Values
6 Values → Value Values
7         | λ
```

Figure 7.9: Grammar for Lisp-like expressions.

---

Consider the Lisp-style expression grammar in Figure 7.9. Rule 6 is a right-recursive rule. An example program looks as follows:

```
( plus 31 ( prod 20 2 20 ) )
```

We will use a usual recursive descent parser. In the recursive descent parser, we will add *semantic actions*.

```
procedure Start()
  switch (...)
  case ts.peek() in { num, lparen }:
    answer := Value(); // semantic value of Value
    call match($)      // Value is nonterminal.
    call print(answer)
  end

procedure Value() returns int
  switch (...)
  case ts.peek() == num:
    call match(num)
    answer := num.valueof() // semantic value of num
    return(answer)         // num is a terminal.
```

```
    case ts.peek() == lparen:
        call match(lparen)
        answer := Expr()    // semantic value of Expr
        call match(rparen)
        return(answer)
end
```

```
function Expr() returns int
switch (...)
case ts.peek() == plus:
    call match(plus)
    op1 := Value()    // from bottom up
    op2 := Value()    // from bottom up
    return(op1 + op2)
case ts.peek() == prod:
    call match(prod)
```

```

        answer := Values(1) // semantic value of Value
        return(answer)
end

function Values(int thusfar) returns int
    switch (...)
    case ts.peek() in { num, lparen }:
        next      := Value() // from bottom up
        answer    := Values(thusfar * next)
        return(answer)
    case ts.peek() == rparen:
        return(thusfar)
end

```

Figure 7.10 Recursive descent parser with semantic actions.  
Ts means the input token stream.

Because a recursive descent parser takes a top-down approach, the parameters of the parsing routines (say *Values*) are inherited attributes of the nonterminal (which are passed and used during the construction of the *Values* subtrees) and the return values of the parsing routines are synthesized attributes (which are computed from the subtree).

For the **Values** nonterminal, the **thusfar** attributes propagate from top down. For the return values of the parsing procedures, the attributes propagate from bottom up.

Try the above recursive descent parser with the following input:

( plus 31 ( prod 20 7 50 ) )

Draw the syntax tree and show the propagation of attributes.

## Attributes in yacc

In yacc, we use `$$`, `$1`, `$2`, `...`, to denote the semantic values (as semantic records) of the symbols in a production rule.

The type of the attribute values in yacc is `YYSTYPE`, which is a union type. Its declaration looks like

```
%union {
    int ival;
    char *name;
    double dval;
}
```

We may declare a terminal or nonterminal with type information, as follows,

```
%type<ival> intToken
```

An example.

```
%union {  
    int    value;  
    char   *symbol;  
}
```

```
%type<value> exp term factor
```

```
%type<symbol> ident
```

```
. . .
```

```
exp : exp '+' term    { $$ = $1 + $3; };  
    /* Note $1 and $3 are ints here. */
```

```
factor : ident        { $$ = lookup(symbolTable, $1); };  
    /* Note $1 is a char* here. */
```

Note that there is a type for each attribute. There would be a type error in the following code:

```
exp : exp '+' ident    { $$ = $1 + $3; };
```

*Example.*

```
exp : exp op term    { if $2 == 1 then $$ = $1 + $3;  
                      else if $2 == 2 then $$ = $1 - $3;};  
op:  '+'             { $$ = 1; };  
op:  '-'             { $$ = 2; };
```

## §7.4 Abstract syntax trees

Simple compilers can be implemented in one pass. More powerful and complex compilers require more passes, such as semantic analysis, symbol table construction, program optimization, and code generation, etc. For multi-pass compilation, there is usually an internal representation of the program. Abstract syntax trees (AST) is such an internal representation. The parser simply builds the AST in a syntax-directed way.

### §7.4.1 Concrete and abstract syntax trees

Ex. Nonterminals for operator precedence and associativity are not included in AST.

Ex. Nonterminals used for ease of parsing are omitted in AST.

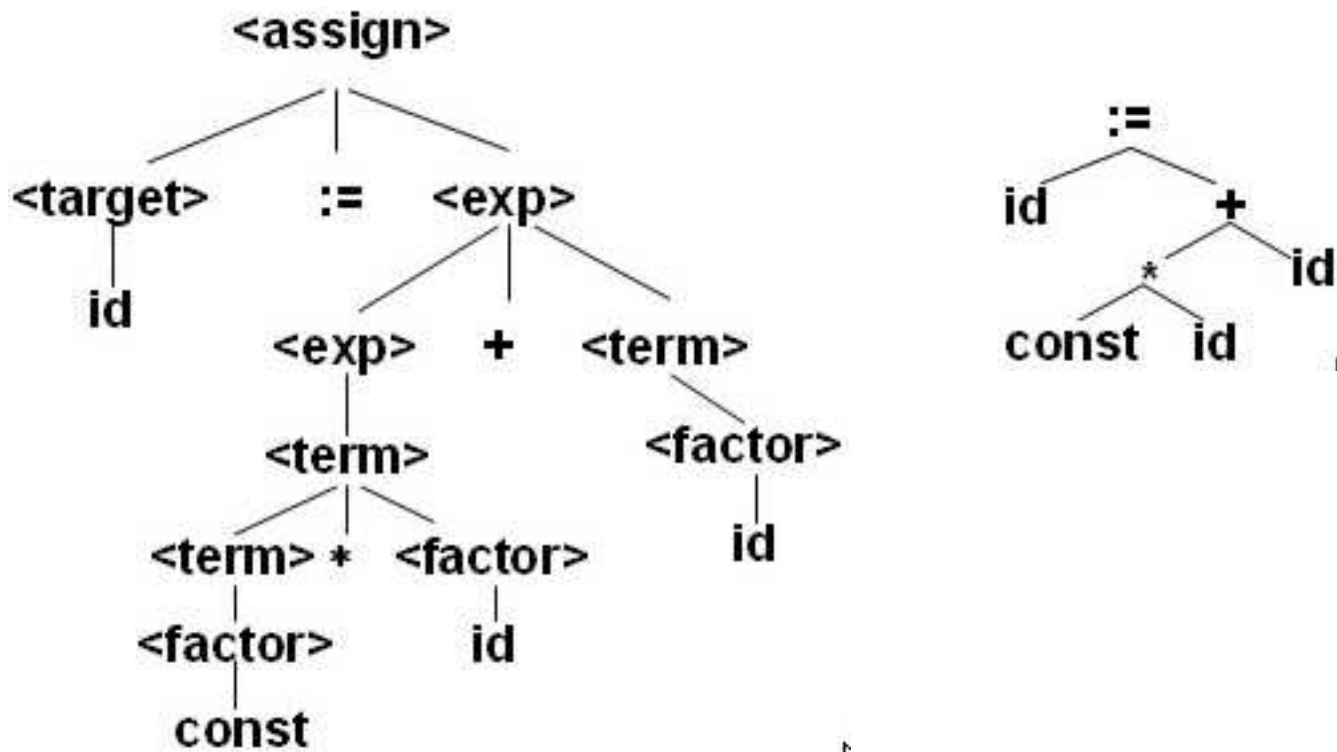


Figure 1: Parse tree vs. AST

**Example.** Figure 7.11 is an abstract syntax tree for *Digs*. The nonterminal *Digs* essentially represents an ordered list of digits.

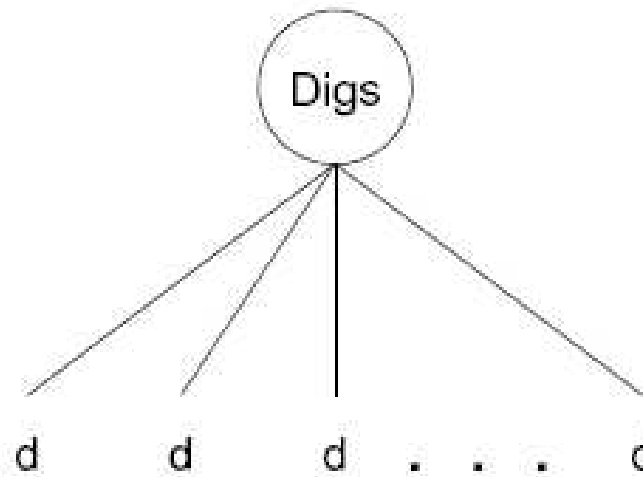


Figure 7.11: Abstract syntax tree for *Digs*.

---

Some nodes have a fixed number of children, say `+` and `*`. Other nodes may have an arbitrary number of children, such as `stmt-list`, `parameter-list`, etc. (However, the nodes themselves have a fixed size.)

Figure 7.12 is a sample node for AST. Each node has four pointers: each for parent, leftmost sibling, right sibling, and leftmost child, respectively.

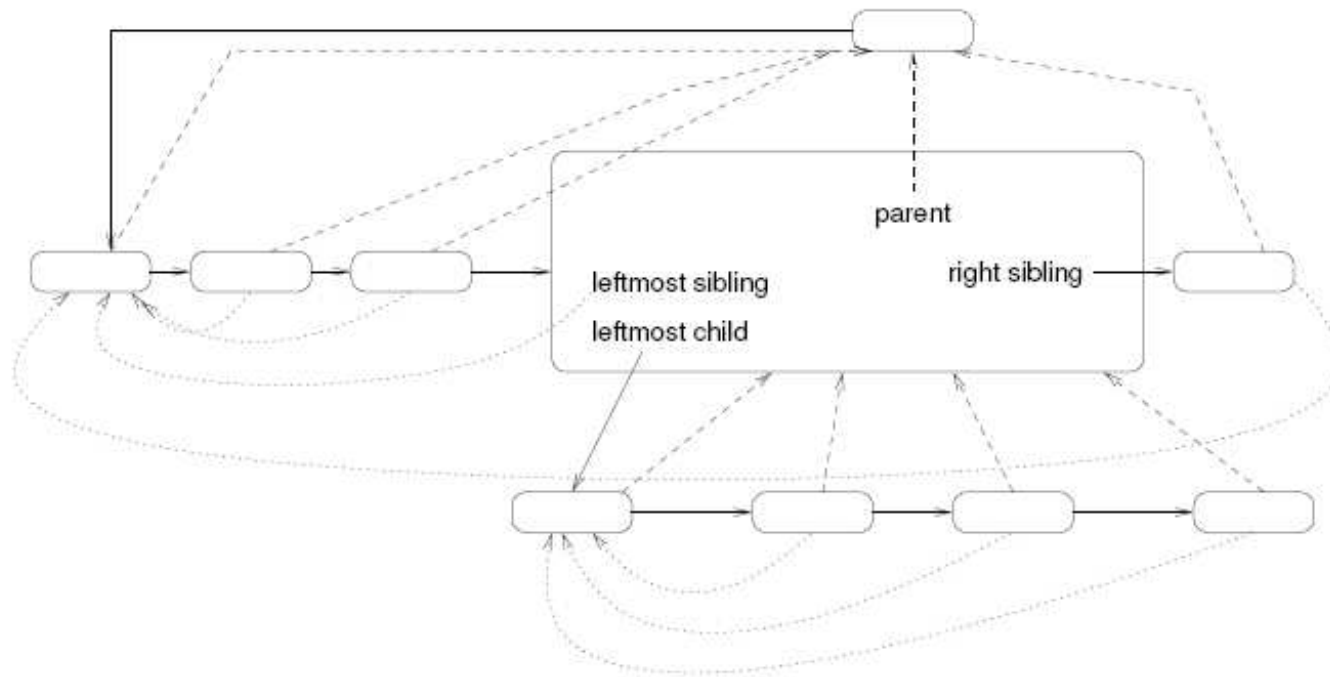


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

### §7.4.3 Infrastructure for creating ASTs

We need four methods for creating ASTs:

1. `MakeNode(t)`: It can make an integer node, a symbol node, an operator node, or a null node (not a null pointer), etc.
2. `x.MakeSiblings(y)`: This causes node  $y$  to be  $x$ 's rightmost sibling.
3. `x.AdoptChildren(y)`: This makes node  $x$  the parent of  $y$  and  $y$ 's siblings.
4. `MakeFamily(op, kid1, kid2, ..., kidn)`: This creates a family with  $op$  as the parent and others as children of  $op$ . For `MakeFamily(op, kid1, kid2)`, we can use

*MakeNode(op).AdoptChildren(kid1.MakeSiblings(kid2))*

Figure 7.13 shows two methods for building ASTs.

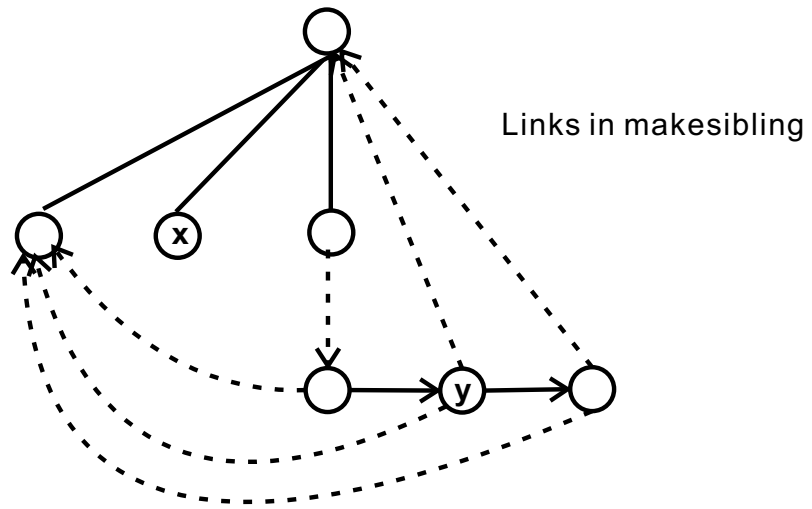
```

function MakeSiblings(y) returns Node
  /* find the rightmost node in "this" list */
  xsibs          := this
  while xsibs.rightSib != null do xsibs := xsibs.rightSib
  /* join the list */
  ysibs          := y.leftmostSib
  xsibs.rightSib := ysibs
  /* set pointers for the new siblings */
  ysibs.leftmostSib := xsibs.leftmostSib
  ysibs.parent      := xsibs.parent
  while ysibs.rightSib != null do
    ysibs          := ysibs.rightSib
    ysibs.leftmostSib := xsibs.leftmostSib
    ysibs.parent    := xsibs.parent
  return(ysibs)
end

```

```
function AdoptChildren(y) returns Node
  if this.leftmostChild != null
  then this.leftChild.MakeSiblings(y)
  else ysibs          := y.leftmostSib
       this.leftmostChild := ysibs
       while ysibs != null do
         ysibs.parent      := this
         ysibs              := ysibs.rightSib
       end
end
```

Figure 7.13 Methods for building an AST



```

/* Assert: y ≠ null */
function S (y) returns Node
/* Find the rightmost node in this list */
xsibs ← this
while xsibs.rightSib ≠ null do xsibs ← xsibs.rightSib
/* Join the lists */
ysibs ← y.leftmostSib
xsibs.rightSib ← ysibs
/* Set pointers for the new siblings */
ysibs.leftmostSib ← xsibs.leftmostSib
ysibs.parent ← xsibs.parent
while ysibs.rightSib ≠ null do
ysibs ← ysibs.rightSib
ysibs.leftmostSib ← xsibs.leftmostSib
ysibs.parent ← xsibs.parent
return (ysibs)
end

/* Assert: y ≠ null */
function C (y) returns Node
if this.leftmostChild ≠ null
then this.leftmostChild.S (y)
else
ysibs ← y.leftmostSib
this.leftmostChild ← ysibs
while ysibs ≠ null do
ysibs.parent ← this
ysibs ← ysibs.rightSib
end

```

Figure 7.13: Methods for building an AST.

---

## §7.5 AST design and construction

The designer of AST should consider the following issues:

1. It should be possible to unparse (i.e., convert) an AST into a form for execution.
2. There is no single view of the AST that will fit all phases of a compiler. Thus, we design a class structure that is suitable for the construction of AST. Then we design an *interface* of the class structure for each phase.

In general there are four steps in the development of a compiler:

1. Define an unambiguous grammar for the programming language.
2. Design an AST for the compiler.
3. Add semantic actions to the grammar to construct the AST.
4. Develop the phases of the compiler.

**Example.** Consider the grammar in Figure 7.14. The AST for the major structures are shown in Figure 7.15. For an `if` statement without the `else` part (Figure 7.15 (c)), use a *null* node.

```
1 Start → Stmt $
2 Stmt → id assign E
3       | if lparen E rparen Stmt else Stmt fi
4       | if lparen E rparen Stmt fi
5       | while lparen E rparen do Stmt od
6       | begin Stmts end
7 Stmts → Stmts semi Stmt
8       | Stmt
9 E     → E plus T
10      | T
11 T    → id
12      | num
```

Figure 7.14: Grammar for a simple language.

---

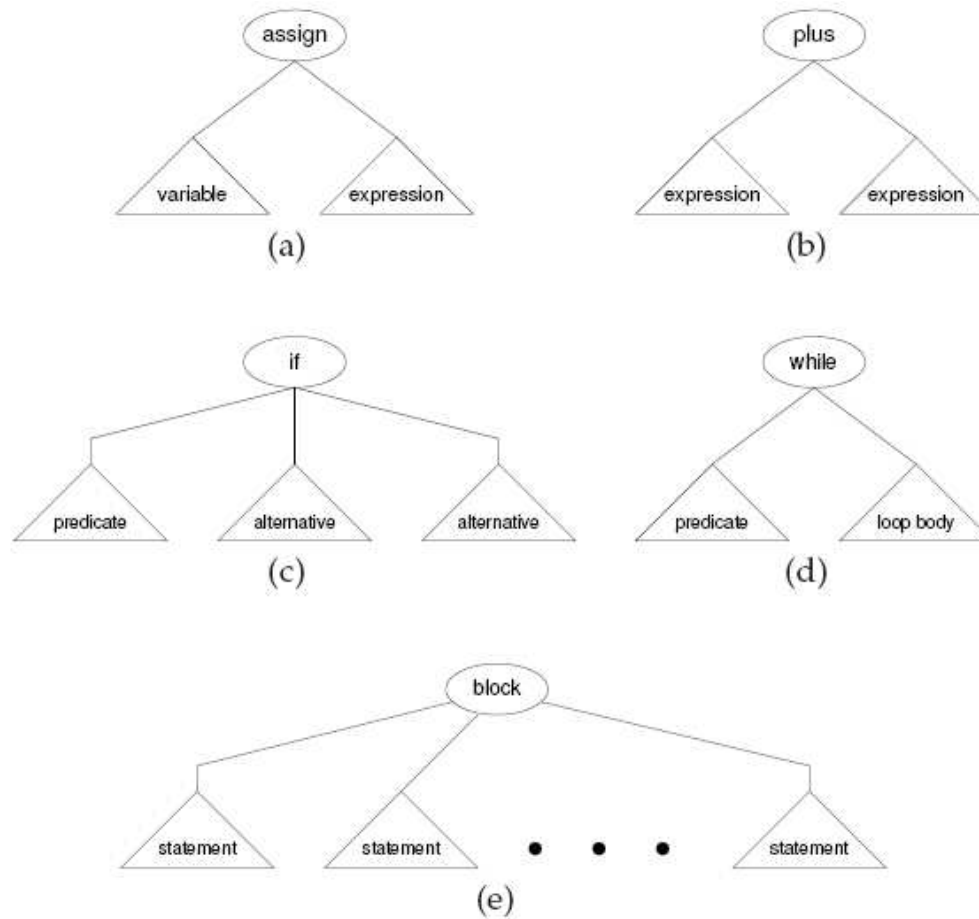


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

---

Figure 7.16 shows a concrete and an abstract syntax trees.

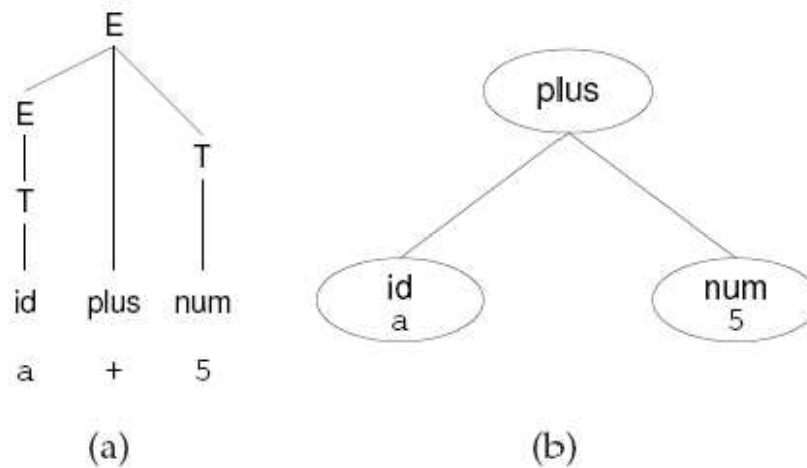


Figure 7.16: (a) Derivation of a + 5 from E;  
(b) Abstract representation of a + 5.

---

Figure 7.17 shows the semantic actions for building the AST.

1.  $Start \rightarrow Stmt \$$   
. { return(Stmt.result) }
2.  $Stmt \rightarrow id\ assign\ E$   
. { Stmt.result := MakeFmaily(assign, id.var, E.expr) }
3.  $Stmt \rightarrow if\ lparen\ E\ rparen\ Stmt\ fi$   
. { Stmt.result := MakeFmaily(if, E.expr, Stmt\_2.result, MakeNode()) }
4.  $Stmt \rightarrow if\ lparen\ E\ rparen\ Stmt\ else\ Stmt\ fi$   
. { Stmt.result := MakeFmaily(if, E.expr, Stmt\_2.result, Stmt\_3.result) }
5.  $Stmt \rightarrow while\ lparen\ E\ rparen\ do\ Stmt\ od$   
. { Stmt.result := MakeFmaily(while, E.expr, Stmt\_2.result) }
6.  $Stmt \rightarrow begin\ Stmts\ end$   
. { Stmt.result := MakeFmaily(block, Stmts.list) }
7.  $Stmts \rightarrow Stmts\ semi\ Stmt$

. {  $\text{Stmts.list} := \text{Stmts\_2.list.MakeSiblings}(\text{Stmt.result})$  }

8.  $\text{Stmts} \rightarrow \text{Stmt}$

. {  $\text{Stmts.list} := \text{Stmt.result}$  }

9.  $E \rightarrow E \text{ plus } T$

. {  $E.\text{expr} := \text{MakeFamily}(\text{plus}, E\_2.\text{expr}, T.\text{expr})$  }

10.  $E \rightarrow T$

. {  $E.\text{expr} := T.\text{expr}$  }

11.  $T \rightarrow id$

. {  $T.\text{expr} := \text{MakeNode}(id.\text{var})$  }

12.  $T \rightarrow num$

. {  $T.\text{expr} := \text{MakeNode}(num.\text{value})$  }

Figure 7.17 Semantic actions for building the ast for the grammar in Figure 7.14

1	Start	→ Stmt <sub>ast</sub> \$ return (ast)	(13)
2	Stmt <sub>result</sub>	→ id <sub>var</sub> assign E <sub>expr</sub> result ← F (assign, var, expr)	(14)
3		if lparen E <sub>p</sub> rparen Stmt <sub>s</sub> fi result ← F (if, p, s, N ( ))	(15)
4		if lparen E <sub>p</sub> rparen Stmt <sub>s1</sub> else Stmt <sub>s2</sub> fi result ← F (if, p, s1, s2)	(16)
5		while lparen E <sub>p</sub> rparen do Stmt <sub>s</sub> od result ← F (while, p, s)	(17)
6		begin Stmt <sub>s</sub> <sub>list</sub> end result ← F (block, list)	(18)
7	Stmts <sub>result</sub>	→ Stmt <sub>s</sub> <sub>so far</sub> semi Stmt <sub>s</sub> <sub>next</sub> result ← so far. S (next)	(19)
8		Stmt <sub>s</sub> <sub>first</sub> result ← first	(20)
9	E <sub>result</sub>	→ E <sub>e1</sub> plus T <sub>e2</sub> result ← F (plus, e1, e2)	(21)
10		T <sub>e</sub> result ← e	(22)
11	T <sub>result</sub>	→ id <sub>var</sub> result ← N (var)	(23)
12		num <sub>val</sub> result ← N (val)	(24)

Figure 7.17: Semantic actions for grammar in Figure 7.14.

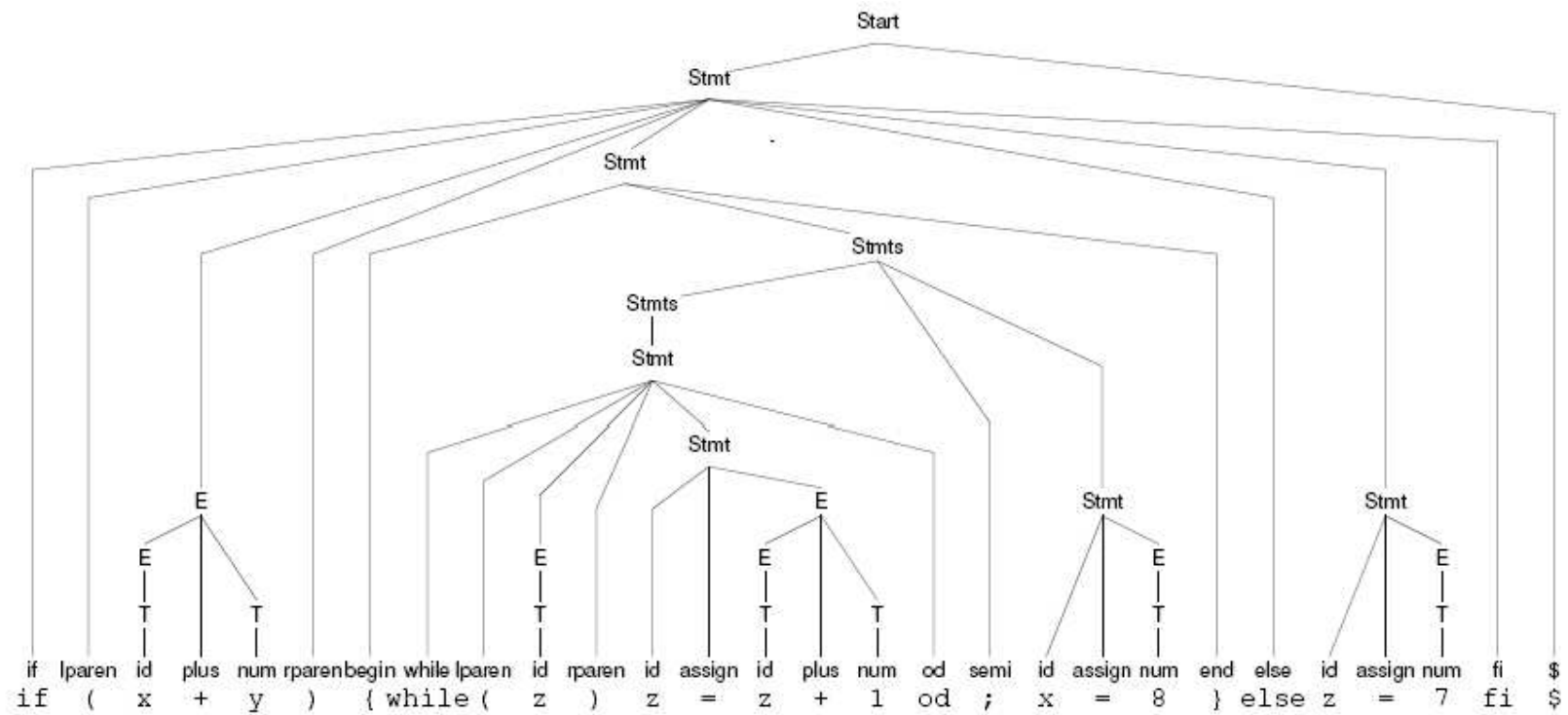


Figure 7.18: Concrete syntax tree.

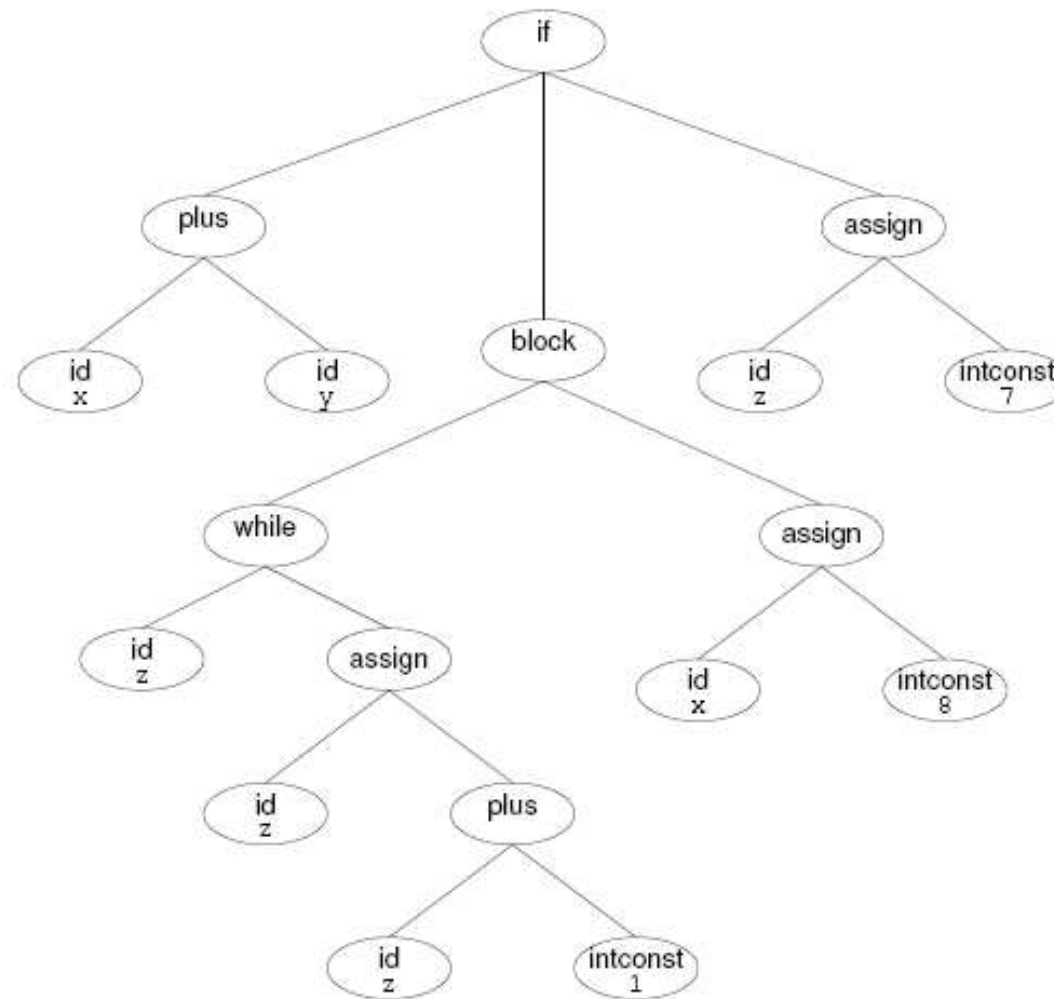


Figure 7.19: AST for the parse tree in Figure 7.18.

## §7.6 AST structures for left and right values

Consider the following assignment statement:

$$x := y;$$

A variable in a program may denote

1. the value stored in that variable. This is called the *right value*.

A constant typically has only the right value. Its address cannot be referenced and hence its value cannot be changed.

An object's self reference `this` is similar.

2. the address of that variable. This is called the *left value*.

The left value of a variable usually cannot be changed by the program though the run-time system (such as garbage collection) may relocate the variable and hence change its left value.

Some programming languages provide a *dereferenec* operator,

which can convert a right value into a left value. An example is C:

*\*exp*

Java does not provide such a dangerous operation.

For the grammar in Figure 7.14, only the variable in the left-hand side of an assignment statement (production 2) generates a left value. All other usages of a variable generate a right value.

In a language such as C that provides a dereference operator, things are a little more complex:

1.  $p = 0$  Here  $p$  generates a left value.
2.  $*p = 0$  Here  $p$  generates a right value.
3.  $x = p$  Here  $p$  generates a right value.
4.  $x = *p$  Here  $p$  generates a right value.
5.  $x = \&p$  Here  $\&p$  generates a left value.

In the AST, an identifier will always denote its location. We will add an explicit `deref` operator to the AST in order to dereference an identifier's left value. The grammar in Figure 7.20 models the C language:

```
1 Start → Stmt $
2 Stmt → L assign R
3 L     → id
4       | deref R
5 R     → L
6       | num
7       | addr L
```

Figure 7.20: Grammar for left and right values.

---

1.  $Start \rightarrow Stmt \$$ 
  - . return(Stmt.ast)
2.  $Stmt \rightarrow L assign R$ 
  - . Stmt.ast := MakeFamily(assign, L.target, R.value)
3.  $L \rightarrow id$ 
  - . L.target := MakeNode(id.name)
4.  $L \rightarrow deref R$ 
  - . L.target := R.value
5.  $R \rightarrow L$ 
  - . R.value := MakeFamily(deref, L.target)
6.  $R \rightarrow num$ 
  - . R.value := MakeNode(num.value)
7.  $R \rightarrow addr L$ 
  - . R.value := L.target // do not de-reference L

Figure 7.21 Semantic actions for the grammar in Figure 7.20

- 1 Start  $\rightarrow$  Stmt<sub>ast</sub> \$  
return (*ast*)
- 2 Stmt<sub>result</sub>  $\rightarrow$  L<sub>target</sub> assign R<sub>source</sub>  
result  $\leftarrow$  F (*assign, target, source*)
- 3 L<sub>result</sub>  $\rightarrow$  id<sub>name</sub>  
result  $\leftarrow$  N (*name*)
- 4 | deref R<sub>val</sub>  
result  $\leftarrow$  val (25)
- 5 R<sub>result</sub>  $\rightarrow$  L<sub>val</sub>  
result  $\leftarrow$  F (*deref, val*) (26)
- 6 | num<sub>val</sub>  
result  $\leftarrow$  N (*val*)
- 7 | addr L<sub>val</sub>  
result  $\leftarrow$  val (27)

Figure 7.21: Semantic actions to create ASTs for the grammar in Figure 7.20.

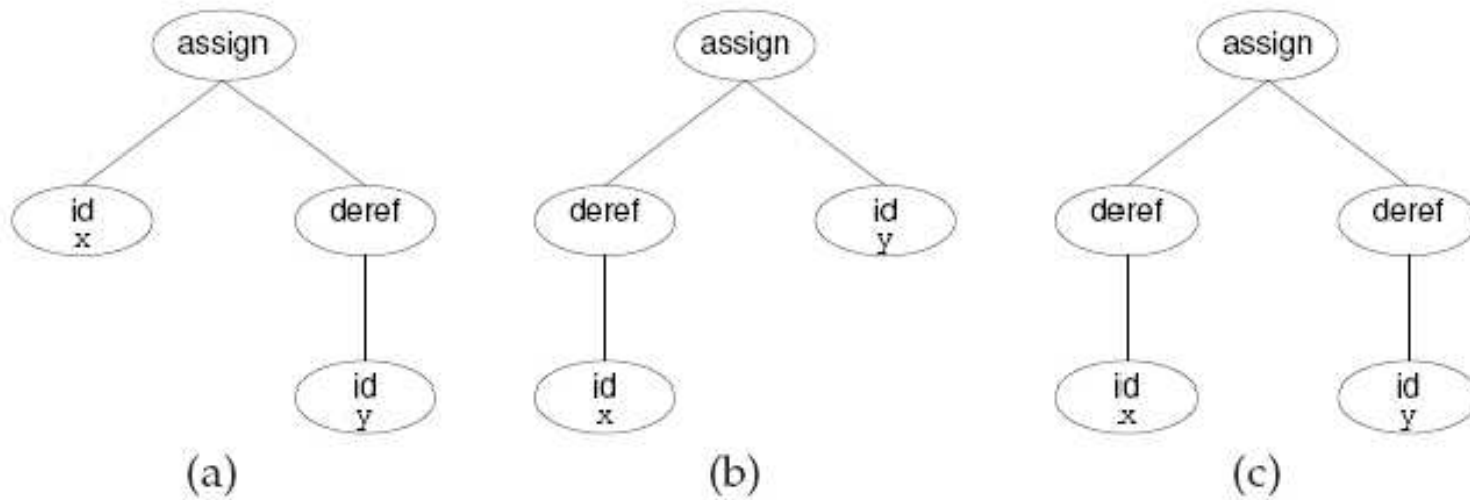


Figure 7.22: ASTs illustrating left and right values for the assignments:

- (a)  $x = y$
- (b)  $x = \&y$
- (c)  $\star x = y$

Note that (b) should be “ $\star x = \&y$ ”.

## §7.7 Design patterns for ASTs

Patterns are meant to save time and effort in developing and maintaining software.

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Not all software patterns are design patterns. For instance, algorithms solve computational problems rather than software design problems.

### §7.7.1 Node class hierarchy

The node management issues in §7.4 (including node data structure with 4 pointers, the `MakeNode`, `MakeSiblings`, `AdoptChildren`, and `MakeFamily` routines) are common to AST construction in a compiler. We can design a base node class (*AbstractNode*) with these features. Other node types (such as *if*, *plus*, etc.) are its extensions (subclasses).

Different phases in a compiler view the AST very differently. It is not easy to design a class hierarchy that fits all phases. So we will use a quite flat class hierarchy.

## §7.7.2 Visitor patterns

(Note. You need to be familiar with an OO language (e.g., Java or C++) to understand the example in this subsection.)

In an OO program, a variable, such as `foo` below, has a *declared type* (which is A) and an *actual type* (which is B). The actual type is either the declared type or a subclass of the declared type.

```
class A { ... }  
class B extends A { ... }  
A foo;  
foo := new B();
```

AST for languages like Java contains around 50 node types.

Compilers like GCC have around 200 phases.

It is recommended that code for a single phase should be written in a single class, and not distributed among the various node types. A phase is crafted by writing a *Visit* method in the phase's class. A phase  $f$  performs its work for a particular node  $n$ :

$$f.Visit(AbstractNode\ n)$$

Common phases include semantic analysis, reachability analysis, type checking, type inference, code generation, etc. Consider the following program fragment:

```
class Visitor
  procedure Visit(AbstractNode n)
    print(1)
  end
end
class TypeChecking extends Visitor
  procedure Visit(IfNode i)
    print(2)
  end
  procedure Visit(PlusNode p)
    print(3)
  end
  procedure Visit(MinusNode m)
    print(4)
  end
end
```

```
Visitor f;  
f := new TypeChecking();  
AbstractNode n;  
n := new IfNode();  
f.Visit(n);    // find Visit(AbstractNode) in TypeChecking class
```

For the call `f.Visit(n)`, the `Visit(AbstractNode n)` method within the `Visitor` class is invoked. That is, 1 is printed.

But we actually want to invoke the `Visit(IfNode i)` method within the `TypeChecking` class. How should we implement the `Visit(AbstractNode n)` method in the `Visitor` class so that the `Visit(IfNode i)` method is invoked?

Common object-oriented languages use *single dispatch*. What we hope is *double dispatch*. We will use the following code to achieve double dispatch:

```

class Visitor
  procedure Visit(AbstractNode n)
    n.accept(this) // find accept(Visitor) in IfNode class, etc.
  end
end
class TypeChecking extends Visitor
  procedure Visit(IfNode i)      . . .      end
  procedure Visit(PlusNode p)   . . .      end
  procedure Visit(MinusNode m)  . . .      end
end

class AbstractNode { . . . }

class IfNode extends AbstractNode
  procedure accept(Visitor v)
    v.Visit(this) // find Visit(IfNode) in TypeChecking class
  end
end

```

```
        end
    end

class PlusNode extends AbstractNode
    procedure accept(Visitor v)
        v.Visit(this)
    end
end

class MinusNode extends AbstractNode
    procedure accept(Visitor v)
        v.Visit(this)
    end
end

Visitor f;
```

```
f := new TypeChecking();  
AbstractNode n;  
n := new IfNode();  
f.Visit(n);    // find Visit(IfNode) in TypeChecking class
```

### §7.7.3 Reflective visitor pattern

There are a few disadvantages in the solution in §7.7.2:

1. Every concrete node class, such as `IfNode` and `PlusNode`, must include the `Accept(Visitor)` method in order to achieve double dispatch.
2. Every visitor, such as `TypeChecking`, must be prepared to visit any concrete node type.

For this problem, we may create an `EmptyVisitor` class between the generic `Visitor` class and the `TypeChecking` class. All `Visit` methods in the `EmptyVisitor` class are dummy.

By using *reflection*, we can view node types on a per-visitor basis and avoid the need to specify a `Visit` method for every node type.

*Reflection* is a programming language's ability to inspect, reason about, manipulate, and act upon elements of the language, such as object types.

```

class ReflectiveVisitor
  procedure Visit(AbstractNode n)
    this.Dispatch(n)
  end
  procedure Dispatch(object ob)
    /* invoke the Visit(n) method whose declared parameter n
       is the closest match for the actual type of ob. */
  end
  procedure DefaultVisit(AbstractNode n)
    foreach AbstractNode c in Children(n) do
      this.Visit(c)
    end
  end
end

class TypeChecking extends ReflectiveVisitor
  procedure Visit(NeedsBooleanPredicate nbp)
    /* check the type of nbp.GetPredicate() */

```

```

        /* nbp could be an IfNode or a WhileNode. */
    end
    procedure Visit(NeedsCompatibleTypes nct)
        /* nct could be a PlusNode. */
    end
    procedure Visit(NeedsLeftChildType nlct)
    end
end

class IfNode extends AbstractNode
    implements {NeedsBooleanPredicate}
end

class WhileNode extends AbstractNode
    implements {NeedsBooleanPredicate}
end

```

```
class PlusNode extends AbstractNode
  implements {NeedsCompatibleTypes}
end
```

```
TypeChecking v;
v.Visit(root);
```

Figure 7.24 ReflectiveVisitor

The `v.Visit(root)` call invokes the `this.Dispatch(root)` method.

`this.Dispatch(root)` examines all `Visit` methods in the actual visitor (i.e., the `TypeChecking` class). The `Visit` method that accepts a node type *most closely matched* to the supplied node's actual type is invoked. If no suitable `Visit` method is found, the

`DefaultVisit` method is invoked. Note that the `DefaultVisit` method can be redefined in a `ReflectiveVisitor` subclass.

The meaning of *most closely matched*:

In the call `v.Dispatch(n)`, if `n`'s type `t`, then the exact match `Visit(t)` method in `v`'s class is preferred.

On the other hand, if no exact match is found, the search will widen to find a `Visit` method that can handle a superclass of `t`.



```

class ReflectiveVisitor
  /* Generic visit */
  procedure (AbstractNode n)
    this. (n)
  end
  procedure (Object o)
    /* Find and invoke the (n) method
    /* whose declared parameter n is the closest match
    /* for the actual type of o.
  end
  procedure V (AbstractNode n)
    foreach AbstractNode c ∈ Children(n) do this. (c)
  end
end
class IfNode
  extends AbstractNode
  implements { NeedsBooleanPredicate }
end
class WhileNode
  extends AbstractNode
  implements { NeedsBooleanPredicate }
end
class PlusNode
  extends AbstractNode
  implements { NeedsCompatibleTypes }
end
class TypeChecking extends ReflectiveVisitor
  procedure (NeedsBooleanPredicate nbp)
    /* Check the type of nbp. P ()
  end
  procedure (NeedCompatibleTypes nct)
end
  procedure (NeedsLeftChildType nlct)

```

Appendix.

*Example.* We may use attributes to construct an abstract syntax tree for a program.

	Production	Semantic rules
$E$	$\rightarrow E_1 + T$	$E.ptr = makenode(+, E_1.ptr, T.ptr)$
$E$	$\rightarrow T$	$E.ptr = T.ptr$
$T$	$\rightarrow T_1 * F$	$T.ptr = makenode(*, T_1.ptr, F.ptr)$
$T$	$\rightarrow F$	$T.ptr = F.ptr$
$T$	$\rightarrow (E)$	$T.ptr = E.ptr$
$T$	$\rightarrow int$	$T.ptr = makenode(int.val)$

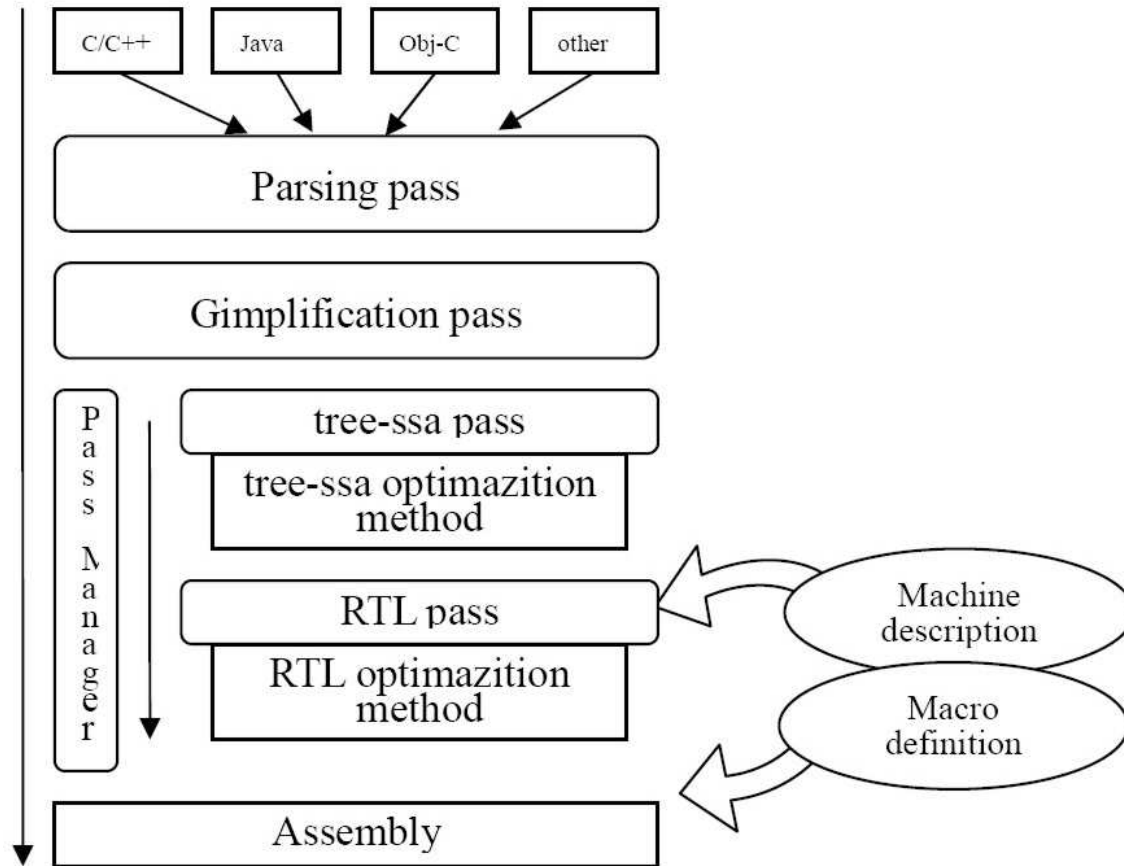
## Common semantic checks

1. Variables must be defined before being used.
2. The types of the expression and the target in an assignment statement must be compatible.
3. Are variables declared with the appropriate types and scopes?
4. Are there duplicated declarations of a name?
5. Are the number and types of the arguments to an operator (including functions and procedures) agree with those of the declared parameters?
6. Are the visibility rules (`public`, `protected`, `private`, `import`, `export`, etc.) observed?
7. Can a variable, a function, and a procedure have identical names (overloading)?
8. Can a variable, a function, and a procedure be redefined

(overriding)?

There are three major categories of semantic rules:

1. semantic rules related to *types*
2. semantic rules related to *scopes*
  - variable not defined.
  - multiple declarations of the same name
  - visibility rules not observed
3. semantic rules related to *control flow*
  - `break` and `continue` are in a `while/for` loop.
  - The label of a `goto` statement must exist.
  - visibility rules not observed.



圖五 gcc passes

Figure 2: GCC Compilers



## Compiler Structure

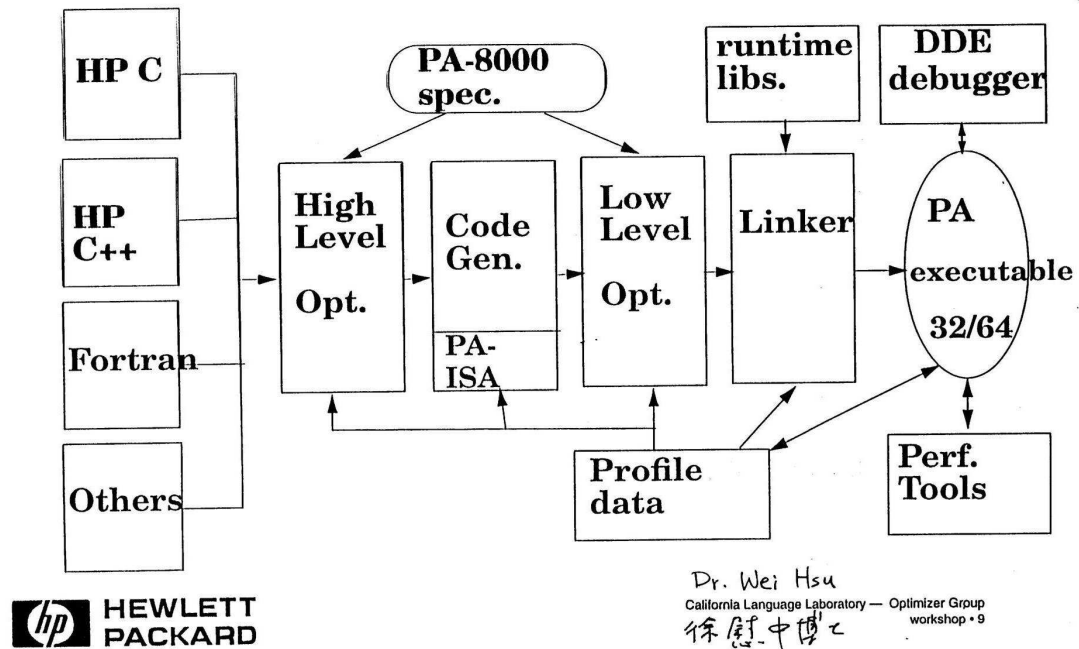


Figure 3: HP Compilers

CPU characteristics.

Processor and architecture	CPU characteristics
DEC AXP 21064	3 operations per clock: 1 int, 1 float, 1 memory.
TI superSPARC superscalar	3 operations per clock: 2 int, 1 float. int ops may include 1 branch and 1 memory ref
Motorola 88110 superscalar	2 operations per clock: 1/2 int, float mul or add memory, divide, bit field manipulations, 1/2 graphics ops
IBM RIOS (R/S 6000) superscalar	4 ops per clock: 1 float, 1 int or memory, 1 branch, 1 condition code
Intel i860XP long instruction word	2 ops per clock: 1 float, 1 int or memory can operate at single instruction/clock

Intel Pentium	2 80X86 ops per clock. Pipelined float (1 per clock). Can overlap memory ops.
HP PA 7100 superscaler	2 ops per clock: 1 float, 1 int or memory
MPIS (SGI) R4000 superpipelined	1 op per clock. Any combination.

### Cache and branch architecture.

Processor	cache	branch prediction
DEC AXP 21064	Icache: 8 KB, 32b lines, direct mapped.  Dcache: 8 KB, 32b lines, direct mapped	early resolution or static based on forward/backward branch displacement.
TI superSPARC	Icache: 20 KB, 8b lines, 5-way set associative  Dcache: 16 KB, 4b lines, 4-way set associative	static with ability to cancel instructions if mispredicted.

Motorola 88110	Icache: 8 KB, 32b lines, 2-way set associative. Dcache: 8 KB, 32b lines, 2-way set associative.	static forward/backward prediction backed by target instruction code. Inst can be cancelled.
IBM RIOS (R/S 6000)	Icache: 8 KB, 64b lines, 2-way set associative. Dcache: 64 KB, 128b lines, 4-way set associative.	Early branch resolution or static forward predicated with ability to cancel.
Intel i860XP	Icache: 16 KB, 32b lines, 2-way set associative. Dcache: 16 KB, 32b lines, 2-way set associative.	static with ability cancel instructions in branch delay slot.

Intel Pentium	Icache: 8 KB, 2-way set associative. Dcache: 8 KB dual access, 2-way set associative.	Dynamic with support of a branch target buffer.
HP PA 7100	Icache: external, 4K-1M, 8b lines, direct-mapped Dcache: external, 4K-2M, 8b lines, direct-mapped	static prediction based on forward/backward dis- placement. Instructions can be canceled.
MPIS (SGI) R4000	Icache: 8 KB, 32/64b lines, direct-mapped Dcache: 8 KB, 32/64b lines,	static with branch delay slot.

In yacc, we use `$$`, `$1`, `$2`, ..., to denote the attribute values (semantic records) of the symbols in a production rule.

The type of the attribute values in yacc is `YYSTYPE`. Its declaration looks like

```
%union {  
    int ival;  
    char *name;  
    double dval;  
}
```

We may declare a terminal or nonterminal with type information, as follows,

```
%type<ival>intToken
```

An example.

```
%union {  
    int    value;  
    char   *symbol;  
}
```

```
%type<value> exp term factor
```

```
%type<symbol> ident
```

```
. . .
```

```
exp : exp '+' term    { $$ = $1 + $3; };  
    /* Note, $1 and $3 are ints here */
```

```
factor : ident        { $$ = lookup(symbolTable, $1); };  
    /* Note, $1 is a char* here */
```

## Attribute propagation in syntax directed translation

1st Example. (LR(1))

expr ::= NUM

expr ::= expr PLUS expr

expr ::= expr MULT expr

expr ::= LPAR expr RPAR

```
abstract class Expr { }

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) { left = L; right = R; }
}

class Mul extends Expr {
    Expr left, right;
    Mul(Expr L, Expr R) { left = L; right = R; }
}

class Num extends Expr {
    int value;
    Num(int v) { value = v; }
}
```

The attribute of every terminal and nonterminal is an Expr object.  
Use yacc for building AST as follows (LR):

```
expr ::= NUM           { $$ = new Num($1.val); }
expr ::= expr PLUS expr { $$ = new Add($1, $3); }
expr ::= expr MULT expr { $$ = new Mul($1, $3); }
expr ::= LPAR expr RPAR { $$ = $2; }
```

**2nd Example.** (LR(1)) Show the upward propagation of attributes.

```
expr    ::= expr PLUS term
expr    ::= term
term    ::= term MULT factor
term    ::= factor
factor  ::= num
factor  ::= LPAR expr RPAR
```

```
abstract class Expr { }

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) { left = L; right = R; }
}

class Mul extends Expr {
    Expr left, right;
    Mul(Expr L, Expr R) { left = L; right = R; }
}

class Num extends Expr {
    int value;
    Num(int v) { value = v; }
}
```

The attribute of every terminal and nonterminal is an Expr object.

Use yacc for building AST as follows (LR):

```
expr ::= expr PLUS term    { $$ = new Add($1, $3); }
expr ::= term              { $$ = $1; }
term  ::= term MULT factor { $$ = new Mult($1, $3); }
term  ::= factor          { $$ = $1; }
factor ::= num            { $$ = new Num($1.val); }
factor ::= LPAR expr RPAR { $$ = $2; }
```

**3rd Example.** (LL(1)) Show the upward propagation of attributes.

```
expr      ::= term etail
etail     ::= PLUS term etail
etail     ::=
term      ::= factor ttail
ttail     ::= MULT factor ttail
ttail     ::=
factor    ::= num
factor    ::= LPAR expr RPAR
```

```

abstract class Expr { }

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) { left = L; right = R; }
    Expr appendLeft(Expr L, Expr R) {
        if R.left != NULL then appendLeft(L, R.left);
        else R.left = L;
    }
}

class Mul extends Expr {
    Expr left, right;
    Mul(Expr L, Expr R) { left = L; right = R; }
    Expr appendLeft(Expr L, Expr R) {
        if R.left != NULL then appendLeft(L, R.left);

```

```
        else R.left = L;
    }
}

class Num extends Expr {
    int value;
    Num(int v) { value = v; }
}
```

The attribute of every terminal and nonterminal is an Expr object.

```
expr ::= term etail
      { $$ = if $2 == NULL then $1 else appendLeft($1, $2); }
etail ::= PLUS term etail
      { $$ = if $3 == NULL then new Add(NULL, $2)
            else appendLeft( new Add(NULL, $2), $3); }
etail ::= { $$ = NULL; }
term ::= factor ttail
      { $$ = if $2 == NULL then $1 else appendLeft($1, $2); }
ttail ::= MULT factor ttail
      { $$ = if $3 == NULL then new Mult(NULL, $2)
            else appendLeft( new Mult(NULL, $2), $3); }
ttail ::= { $$ = NULL; }
factor ::= num { $$ = new Num($1.val); }
factor ::= LPAR expr RPAR { $$ = $2; }
```

From the attribution equations, we can see the attributes propagate from leaves to the root.

We will need to examine the semantic stack management for LL(1) parsers at this point.

We can also use this method (syntax-directed translation) to perform type checking, as follows:

```
D ::= T id      { AddType(id, T.type);  
                  D.type = T.type; }  
D ::= D, id     { AddType(id, D0.type);  
                  D0.type = D1.type; }  
T ::= int       { T.type = intType; }  
T ::= float     { T.type = floatType; }
```