Chapter 7 Syntax-Directed Translation

Wuu Yang

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

first draft: March 25, 2010

current version: August 13, 2023

Chapter outline: Syntax-Directed Translation

1. Overview
2. Bottom-up syntax-directed translation
3. Top-down syntax-directed translation
4. Abstrtact 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.[a]

[a]Double dispatch and reflection are not related to compiler construction. However, the textbook uses them in their algorithms.

```
main() {
  AbstractSyntaxTree ast := parser(inputfile);

  SymbolTable symtab := buildSymbolTable(ast);

  SemanticsVisitor sv = new SemanticsVisitor();
  sv.Visit(ast);

  ReachabilityVisitor rv = new ReachabilityVisitor();
  rv.Visit(ast);

  ThrowsVisitor tv = new ThrowsVisitor();
  tv.Visit(ast);

  TopVisitor topv  = new TopVisitor();
  topv.Visit(ast);
}
```

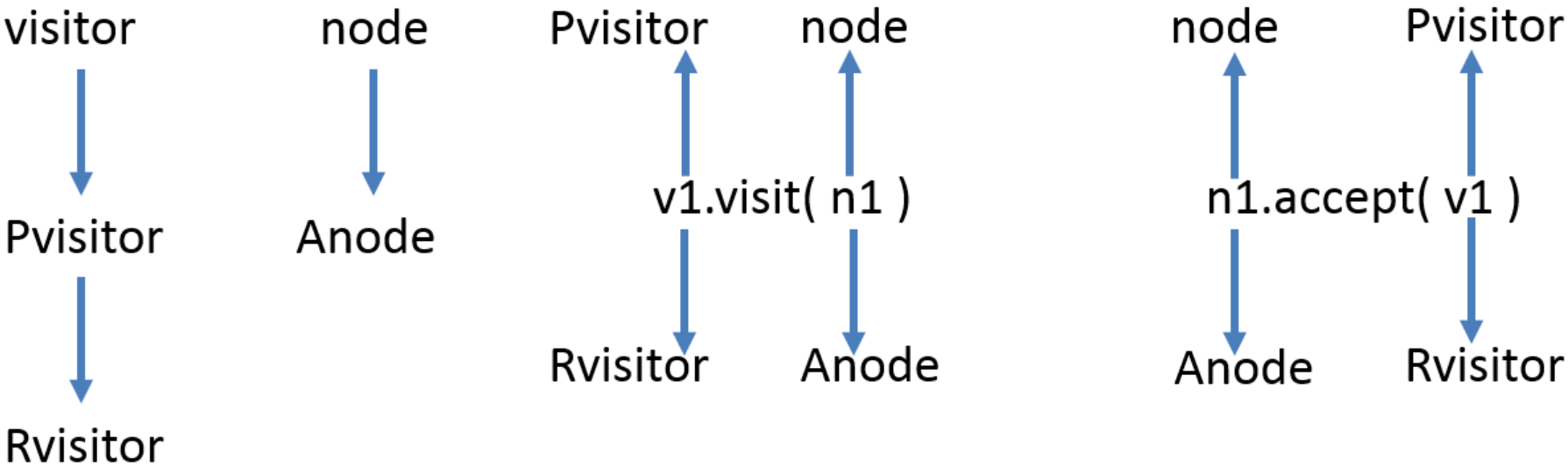
Figure 7.0 Review: the structure of a compiler.

§7.0 double dispatch in Java

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

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

Output is “1st try; 10; 2nd try; 5”.


visitor
Pvisitor
Rvisitor
node
Anode
Pvisitor
node
v1.visit(n1)
Rvisitor
Anode
node
Pvisitor
n1.accept(v1)
Anode
Rvisitor

§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.

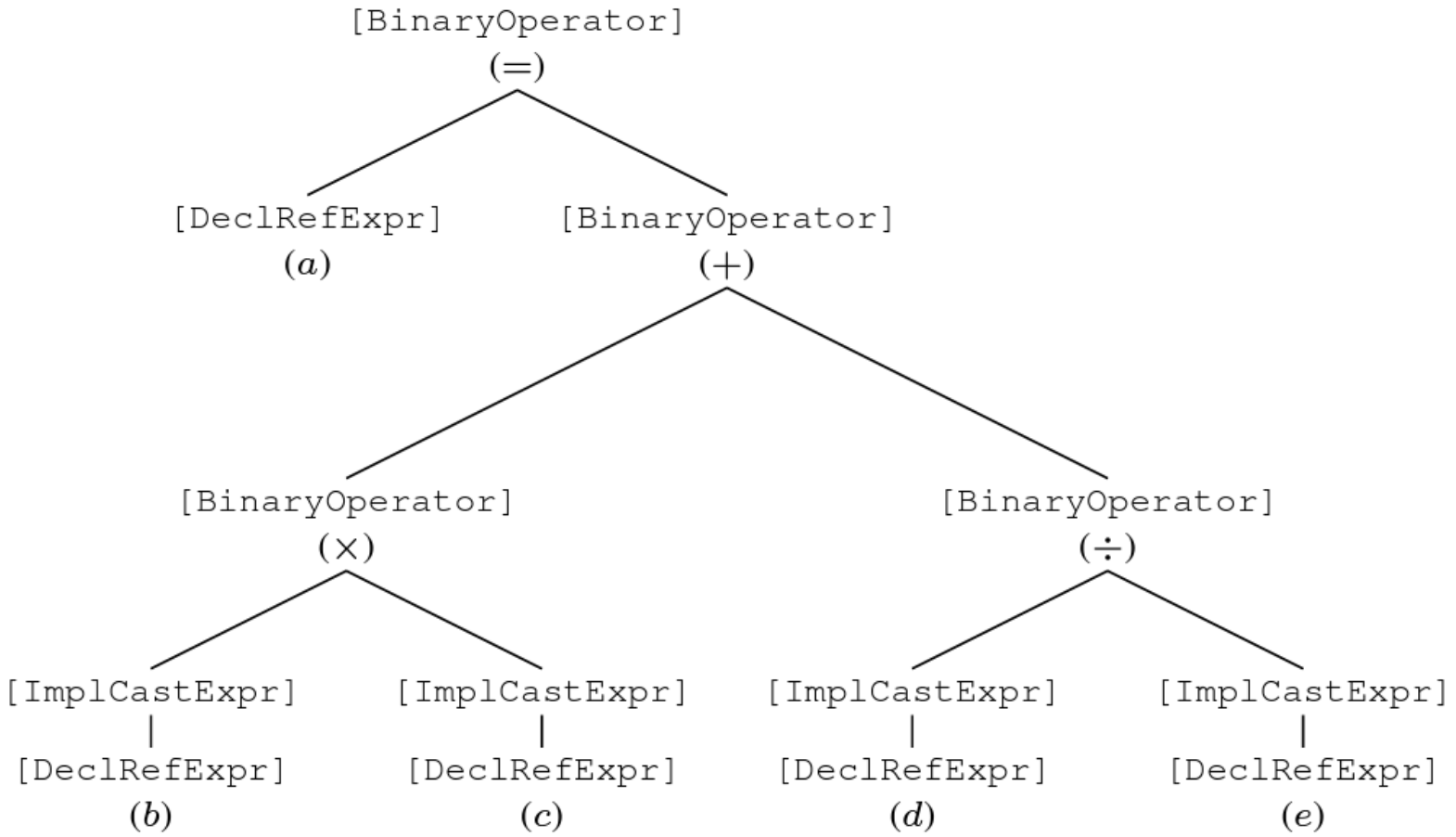


Fig. 1. Example of abbreviated Clang AST for the expression $a = b \times c + d \div e$.

Figure 1: clang-AST.

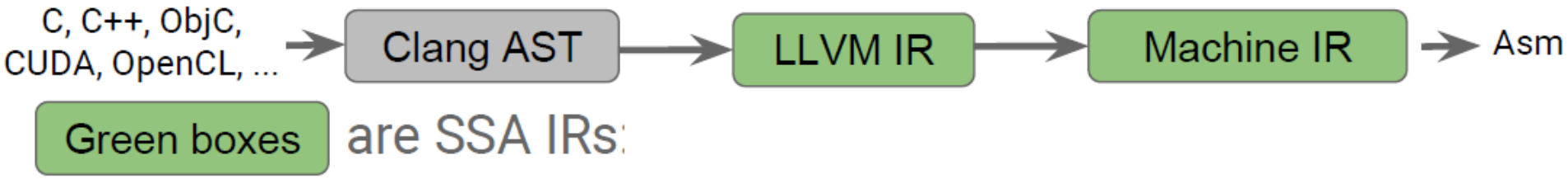


Figure 2: CLANG-AST2.

§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, respectively) during parsing, a syntactic structure in the input program is recognized/predicted. 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
        { $$ = makeUnaryTree(Package_Declaration, $2); }
        ;
```

Here `makeUnaryTree` is a user-defined function.

There is a *semantic value* associated with each (terminal and nonterminal) symbol.

A `num` token needs a semantic value denoting its value. Different `num` tokens have different semantic values.

An `ID` token needs a semantic value denoting the real identifier it represents. Consider "aaa := bbb + 111".

Sometimes, a semantic value is trivial. For instance, the `;` symbol has no serious meaning. Its semantic value can be ignored.

Sometimes, a semantic value (say, of an identifier) 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 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.

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 could be a union type. Its declaration looks like

```
typedef node {
  int a;
  float b;
  node *next;} node;


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

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

```
%type<ival> intToken
%type<expname> Exp Factor Term
%%
Exp ::= Exp pls Term  { $$->a = $1->a + $3->a;  }
```

§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. Examples of attributes are (integer or float) values, scopes, sizes, data types, addresses, access priviliedges, symbol tables, etc.

There are two kinds of attributes:

- inherited attributes
- synthesized attributes

Semantic actions (also called attribute equations) 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 *val*. We use the notation $E.val$ to denote the *val* attribute of the E symbol. The

attributes of +, *, (, and) are useless in the 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 (as well as the right-hand-side terminals and nonterminals) to compute the attributes of the right-hand-side symbols. Such attributes are classified as *inherited* attributes.

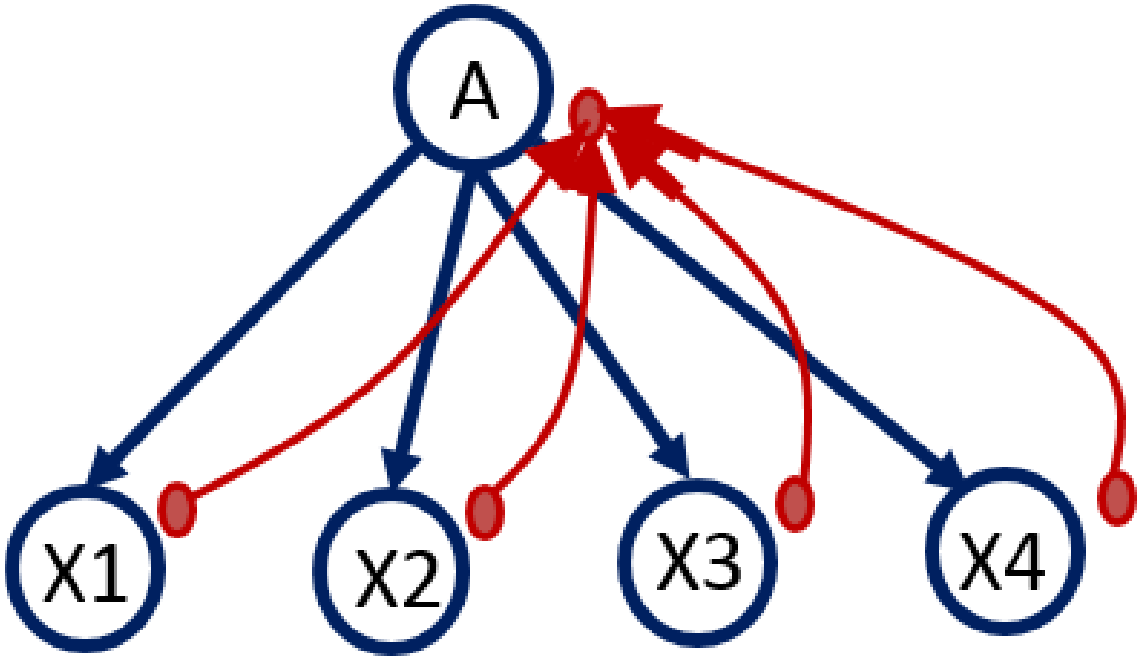


Figure 3: A synthesized attribute in $A \to X_1 X_2 X_3 X_4$.

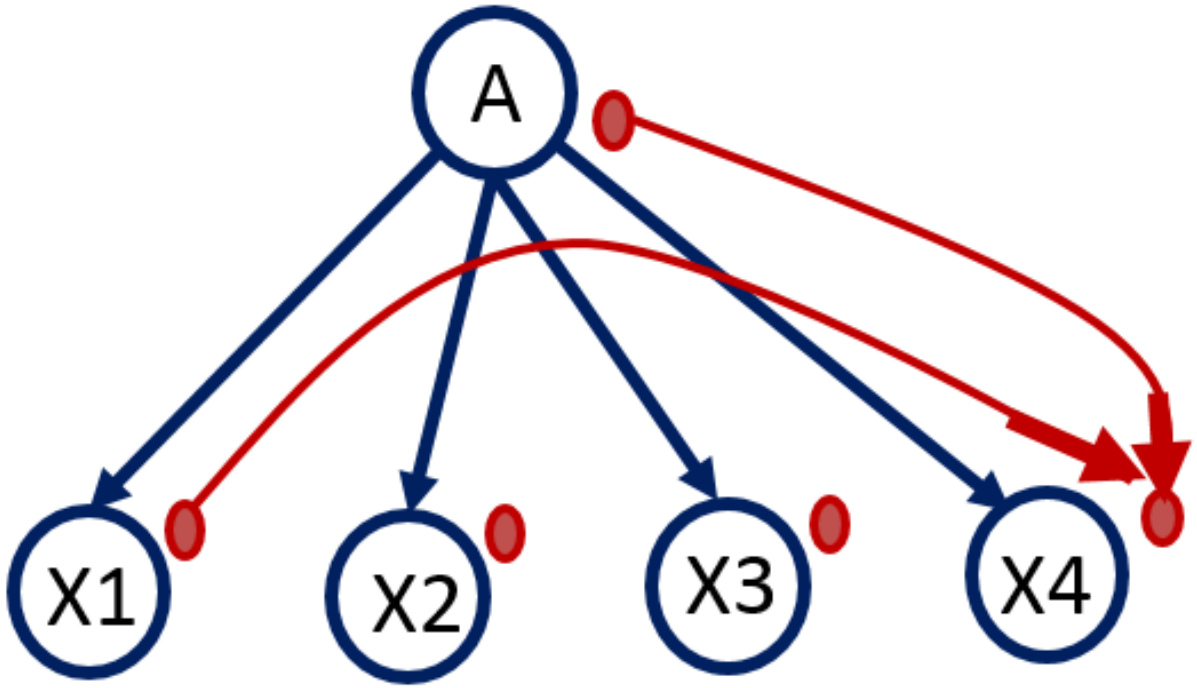


Figure 4: An inherited attribute in $A \to X_1 X_2 X_3 X_4$.

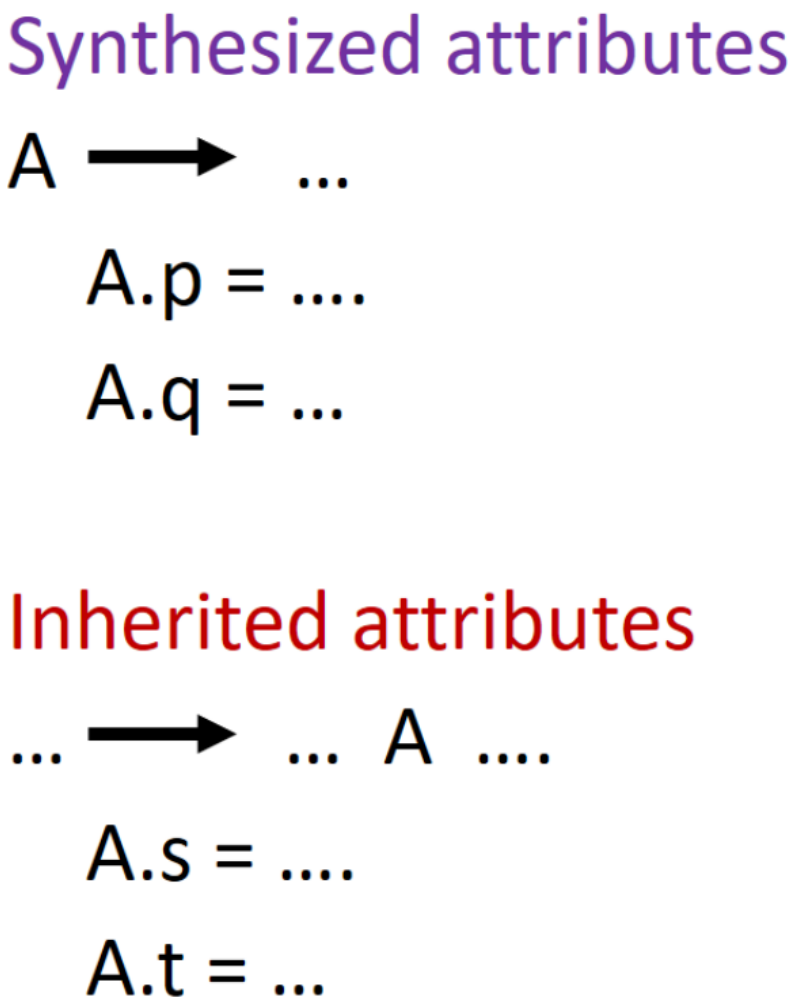
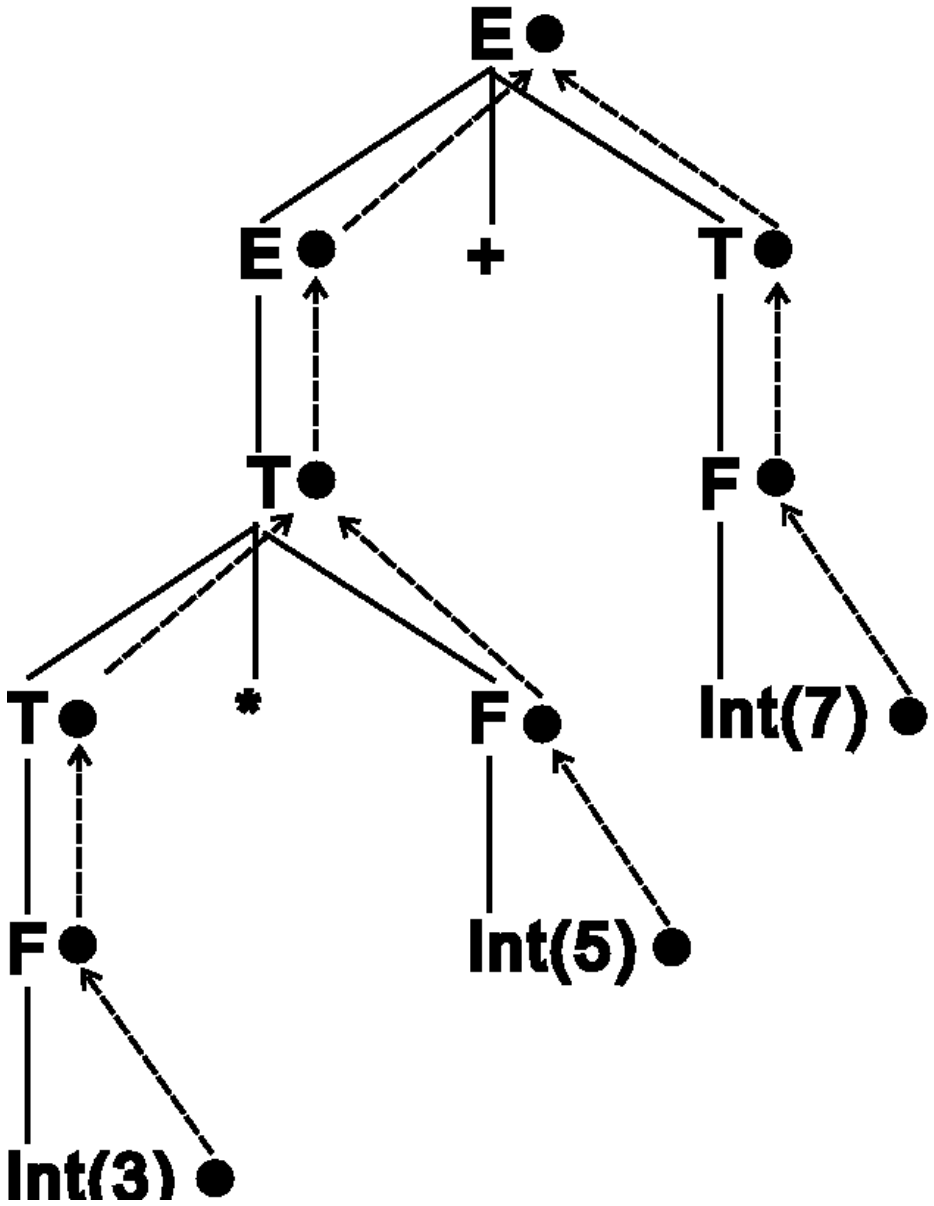


Figure 5: synthesized and inheritated attributes.

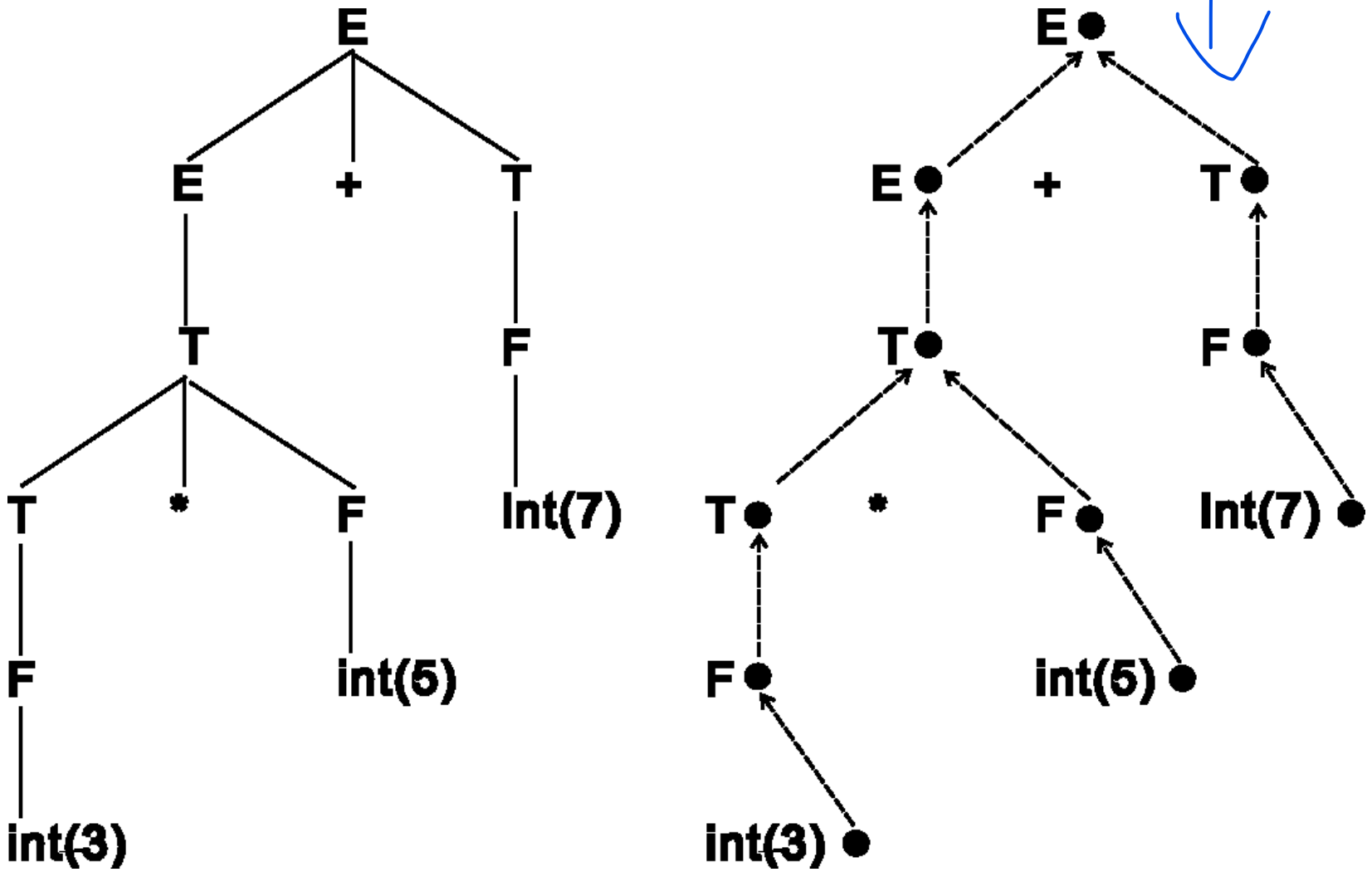
If an attribute grammar contains only synthesized attributes, it is called an *S-attributed grammar*. For ASTs derived from an S-attributed grammar, the attributes can always be evaluated in a bottom-up traversal. Due to this bottom-up nature, the attributes can be evaluated during an LR parsing.

If we draw the graph for the sentence

```
3 * 5 + 7
```


E
E
+
T
T
F
T
*
F
Int(7)
F
Int(5)
Int(3)

We may separate the the parse tree and the attribute (dependency) 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 downard dependences with additional techniques, say with a global symbol table. The upward dependences are handled naturally by the LR 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.

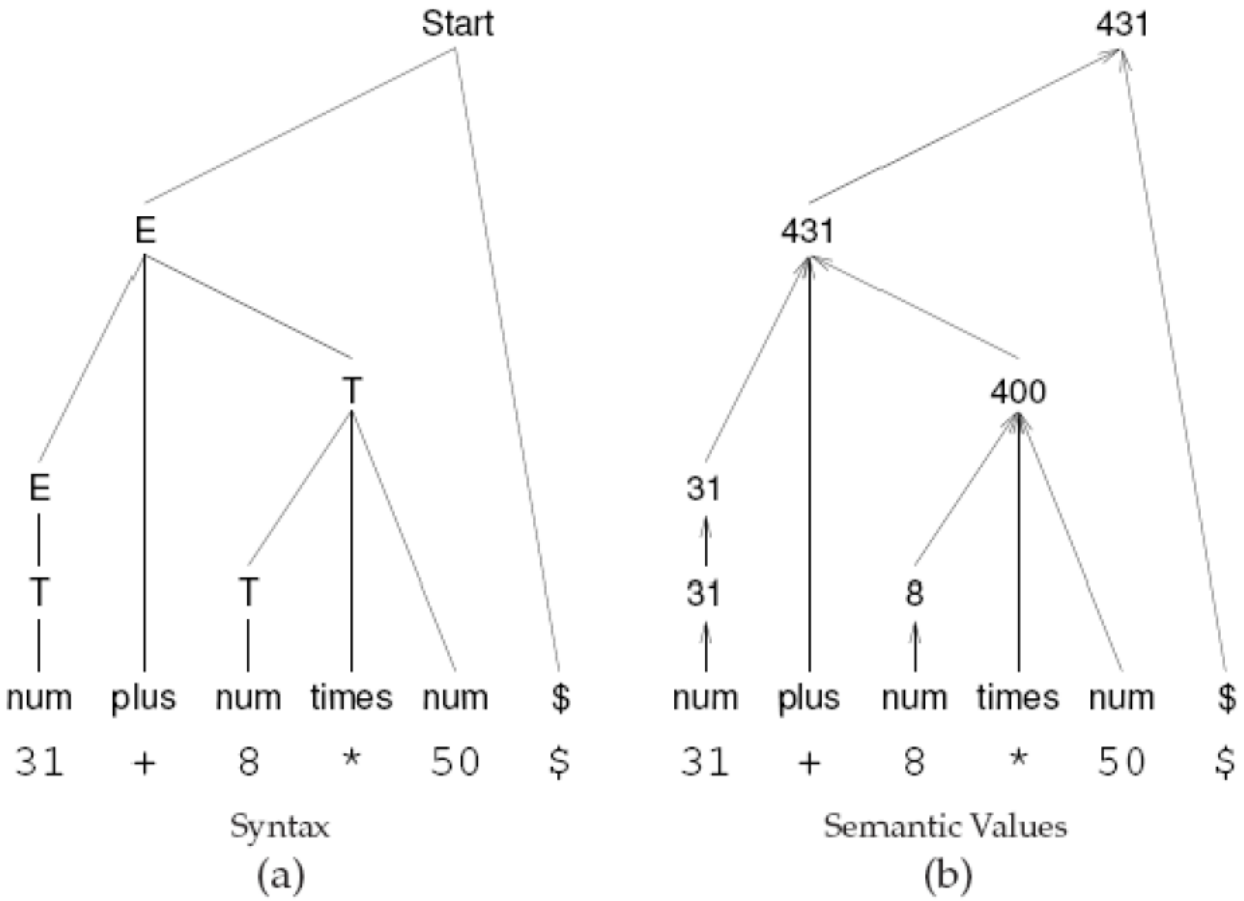


Figure 7.1: (a) Parse tree for the displayed expression;
(b) Synthesized attributes transmit values up the parse tree toward the root.

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

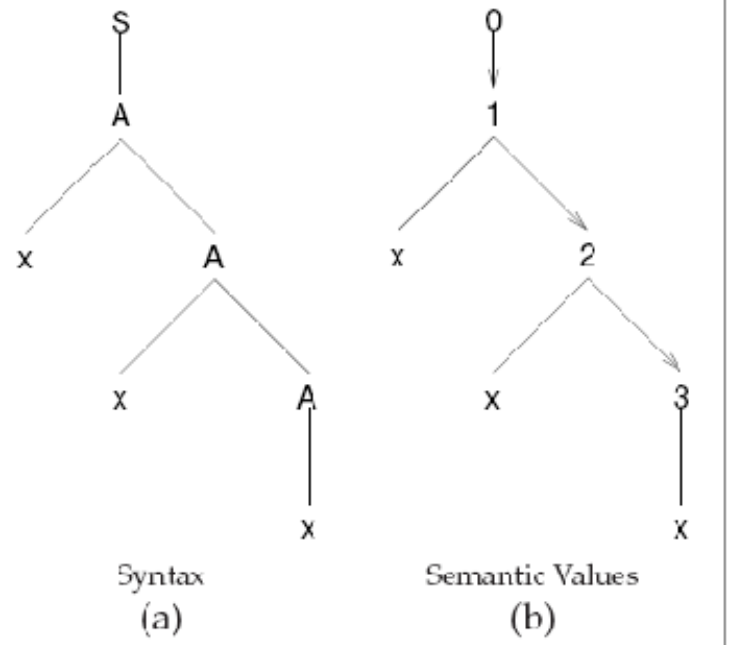


Figure 7.2: (a) Parse tree for the displayed input string; (b) attributes pass from parent to child.

Fig. 1(a). Syntactic rules and semantic functions of a simple grammar, discussed in Section 2.

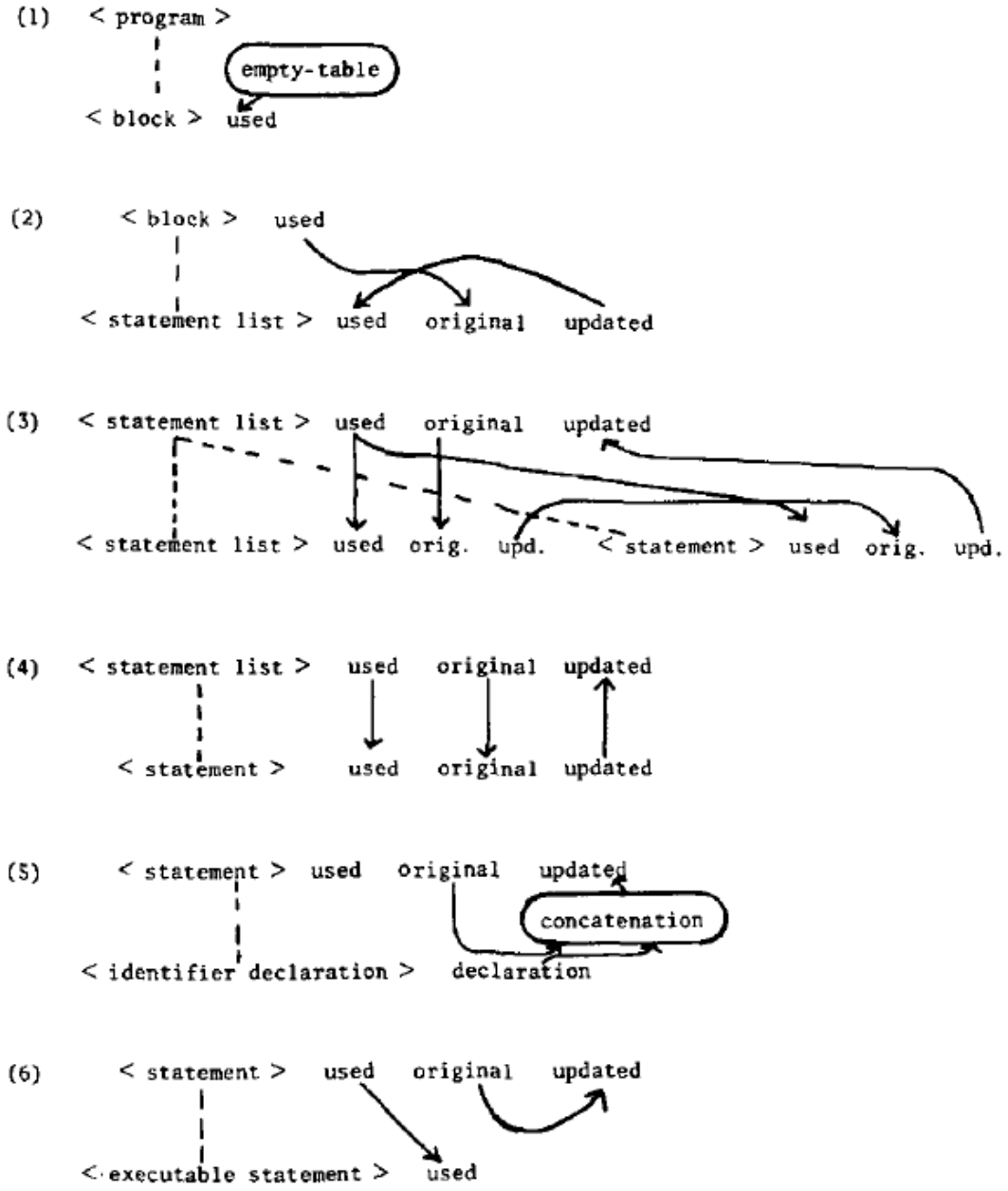
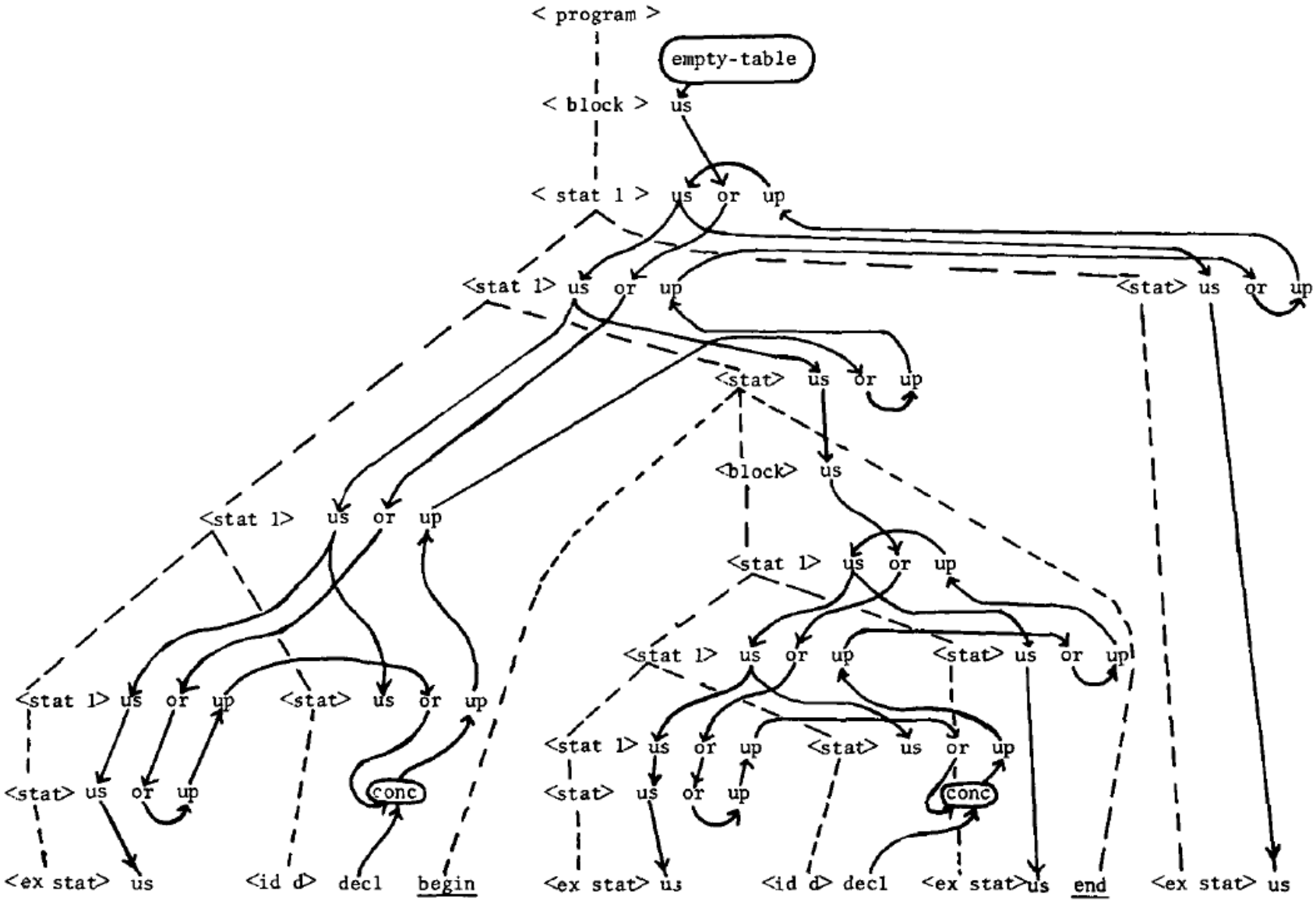


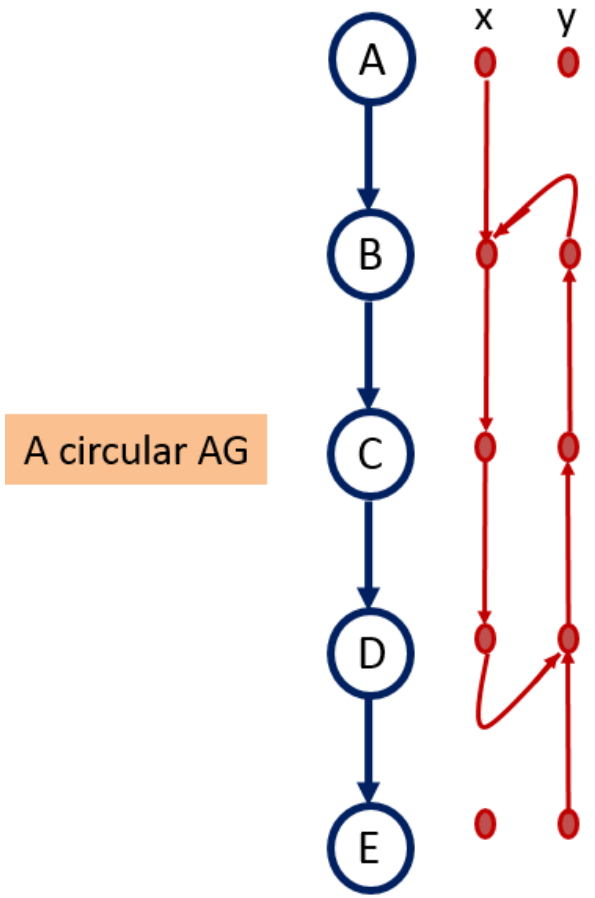
Fig. 2. The derivation tree of a sample program showing the evaluation of the attributes. The names of the syntactic symbols and the attributes are abbreviated.



Issues in attribute grammars:

1. Kastens, ordered attribute grammar.
2. Circularity problem in attribute grammars.
3. Jazayeri, Ogden, and Rounds, "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars," Communications of ACM, Vol. 18, No. 12, Dec. 1975, pp. 697-706.

Example. Here is a circular AG.



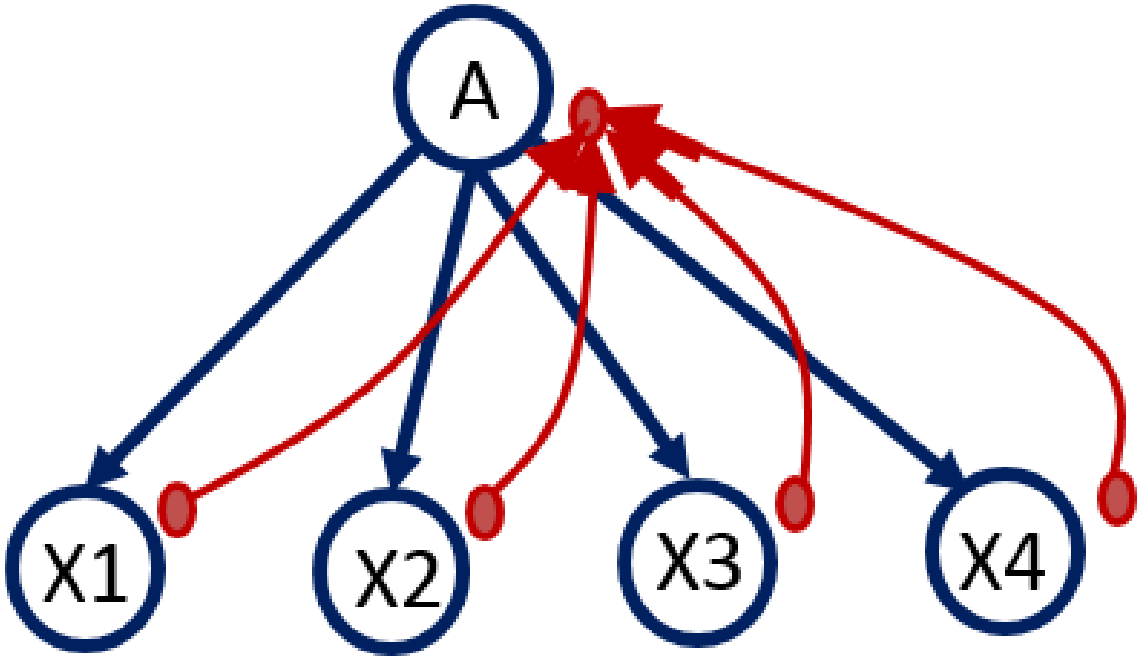
§7.2 Bottom-up syntax-directed translation

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

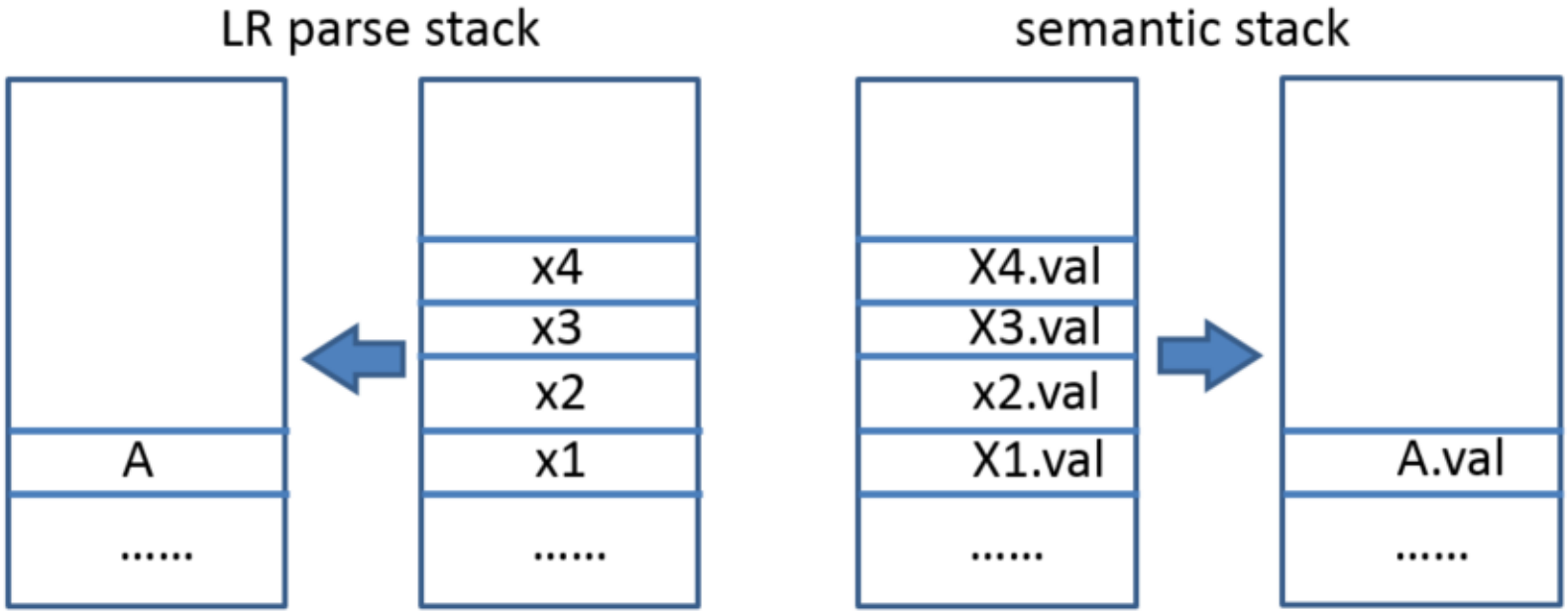
$$A \to X_1 X_2 \ldots X_n$$

The top of the parse stack contains n symbols (i.e., $X_1, X_2, \ldots, X_n$) together with their (already evaluated) attributes. The parser will compute the attributes of A from those of $X_1, X_2, \ldots, X_n$ 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.


A
X1
X2
X3
X4

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(b) 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.

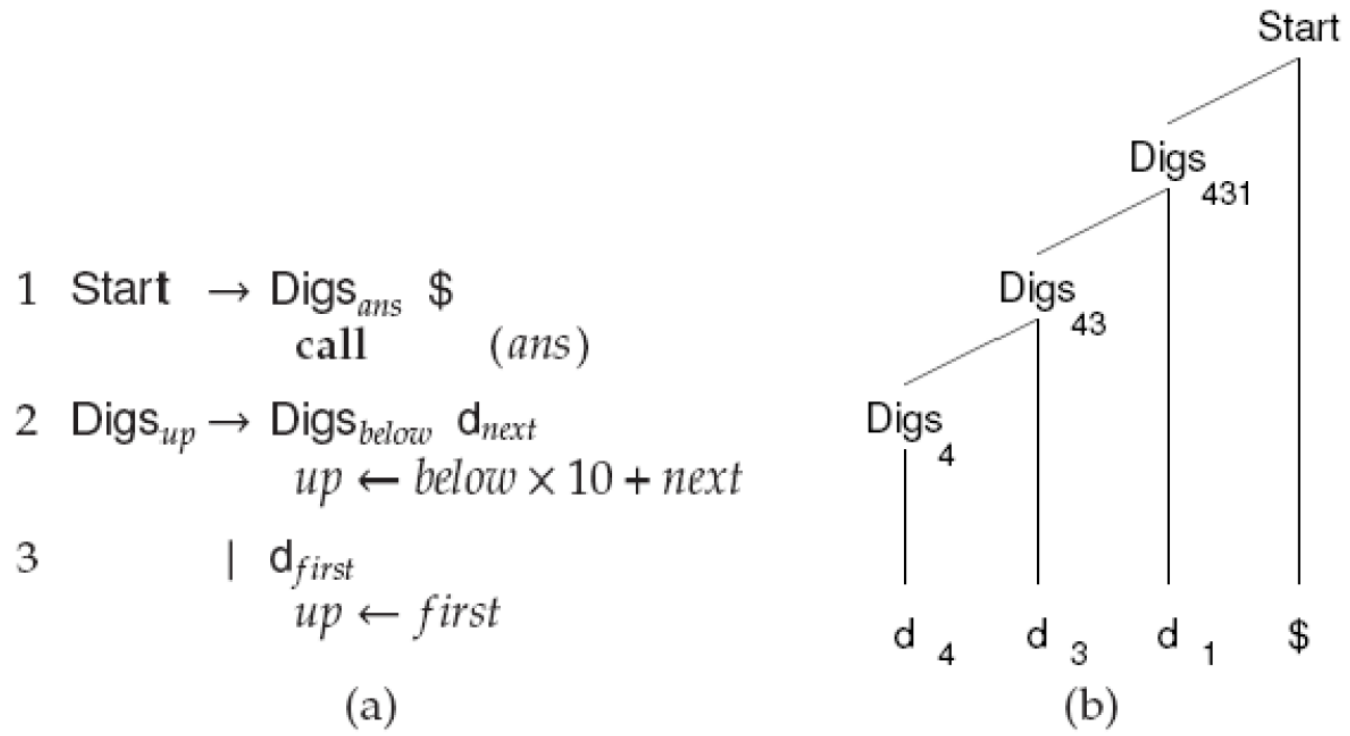


Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input `4 3 1 $`.

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. $Start \rightarrow$	Num \$	$\{ print(Num.val) \}$
2. $Num \rightarrow$	$o\ Digs$	$\{ Num.val := Digs.val \}$
3. $Num \rightarrow$	$Digs$	$\{ Num.val := Digs.val \}$
4. $Digs \rightarrow$	$Digs_1\ d$	$\{ Digs.val := Digs_1.val * 10 + d.val \}$
5. $Digs \rightarrow$	d	$\{ Digs.val := d.val \}$

Figure 7.4(b) shows an octal number "`o 4 3 1 $`".

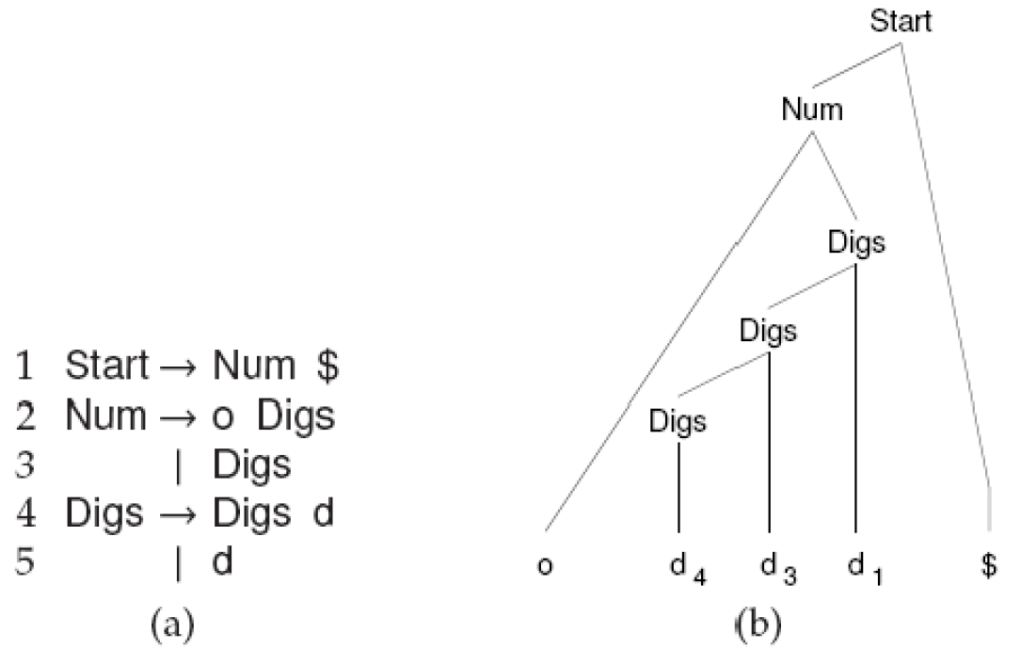


Figure 7.4: (a) Grammar and (b) parse tree for the input o 4 3 1 $.

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 problem.

§7.2.2 Rule cloning

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

		Production	Semantic actions
1. $Start$	$\rightarrow$	Num \$	$\{print(Num.val)\}$
2. Num	$\rightarrow$	$o\ OctDigs$	$\{Num.val := OctDigs.val\}$
3. Num	$\rightarrow$	$DecDigs$	$\{Num.val := DecDigs.val\}$
4. $DecDigs$	$\rightarrow$	$DecDigs_1\ d$	$\{DecDigs.val := DecDigs_1.val * 10 + d.val\}$
5. $DecDigs$	$\rightarrow$	d	$\{DecDigs.val := d.val\}$
6. $OctDigs$	$\rightarrow$	$OctDigs_1\ d$	$\{$**if** $d.val \geq 8$ **then** $error()$
			else $OctDigs.val := OctDigs_1.val * 8 + d.val\}$
7. $OctDigs$	$\rightarrow$	d	$\{$**if** $d.val \geq 8$ **then** $error()$
			else $OctDigs.val := d.val\}$

§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. $Start$	$\rightarrow$	$Num\ \$$	$\{print(Num.val)\}$
2. Num	$\rightarrow$	$SignalOct\ Digs$	$\{Num.val := Digs.val\}$
3. Num	$\rightarrow$	$SignalDec\ Digs$	$\{Num.val := Digs.val\}$
4. $SignalOct$	$\rightarrow$	o	$\{base := 8\}$
5. $SignalDec$	$\rightarrow$		$\{base := 10\}$
6. $Digs$	$\rightarrow$	$Digs_1\ d$	$\{Digs.val := Digs_1.val * base + d.val\}$
7. $Digs$	$\rightarrow$	d	$\{Digs.val := d.val\}$

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. $Start$	$\rightarrow$	Num \$	$\{print(Num.val)\}$
2. Num	$\rightarrow$	$x\ SetBase\ Digs$	$\{Num.val := Digs.val\}$
3. Num	$\rightarrow$	$BaseTen\ Digs$	$\{Num.val := Digs.val\}$
4. $SetBase$	$\rightarrow$	d	$\{base := d.val\}$
5. $BaseTen$	$\rightarrow$		$\{base := 10\}$
6. $Digs$	$\rightarrow$	$Digs_1\ d$	$\{$**if** $d.val \geq base$ **then** $error()$
			else $Digs.val := Digs_1.val * base + d.val\}$
7. $Digs$	$\rightarrow$	d	$\{$**if** $d.val \geq base$ **then** $error()$
			else $Digs.val := d.val\}$

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          → Num_ans $
                     call      (ans)
2 Num_ans        → x SetBase Digs_baseans
                     ans ← baseans
3                  | SetBaseTen Digs_decans
                     ans ← decans
4 SetBase        → d_val
                     base ← val
5 SetBaseTen     → λ
                     base ← 10
6 Digs_up        → Digs_below d_next
                     if next ≥ base
                     then      ("Digit outside allowable ran
                     up ← below × base + next
7                  | d_first
                     if first ≥ base
                     then      ("Digit outside allowable ran
                     up ← first
```

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. $Start$	$\rightarrow$	$Digs$ \$	$\{print(Digs.val)\}$
2. $Digs$	$\rightarrow$	$Digs_1\ d$	$\{Digs.b := Digs_1.b$
			$Digs.val := Digs_1.val * Digs_1.b + d.val\}$
3. $Digs$	$\rightarrow$	$SetBase$	$\{Digs.b := SetBase.b$
			$Digs.val := 0\}$
4. $SetBase$	$\rightarrow$		$\{SetBase.b := 10\}$
5. $SetBase$	$\rightarrow$	$x\ d$	$\{SetBase.b := d.val\}$

1 Start $\rightarrow$ Digs_{ans} \$
  call $(ans.val)$

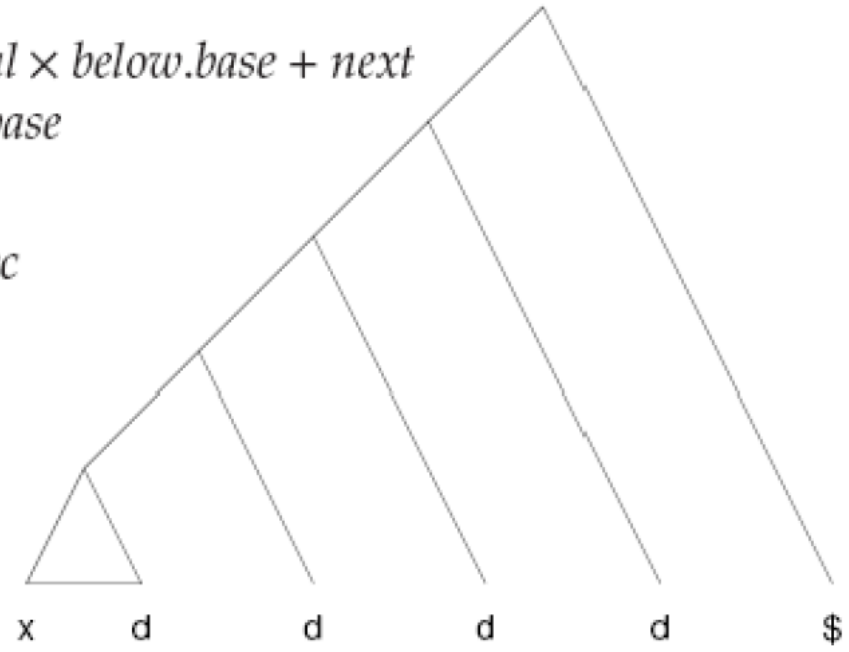
2 Digs_{up} $\rightarrow$ Digs_{below} d_{next}
  $up.val \leftarrow below.val \times below.base + next$
  $up.base \leftarrow below.base$

3 | $\text{SetBase}_{basespec}$
  $up.base \leftarrow basespec$
  $up.val \leftarrow 0$

4 $\text{Setbase}_{n} \rightarrow \lambda$
  $n \leftarrow 10$

5 | x d_{num}
  $n \leftarrow num$

(a)



(b)

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

Rule 3 should be changed to

$$Digs \rightarrow SetBase\ d$$

Otherwise we will have strings such as "`x 5 $`".

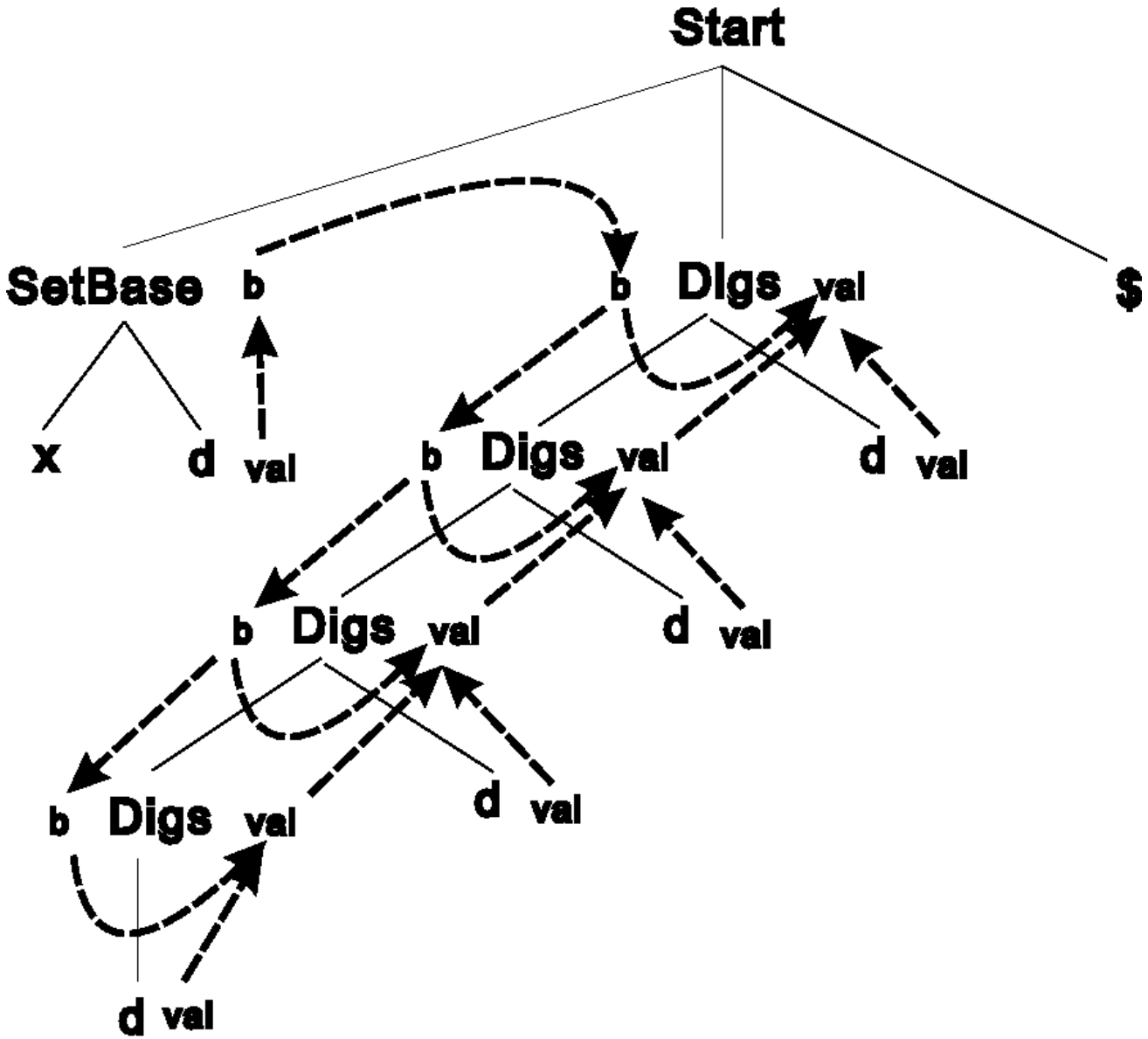
How to modify the attribute equations?

Use inherited attributes

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

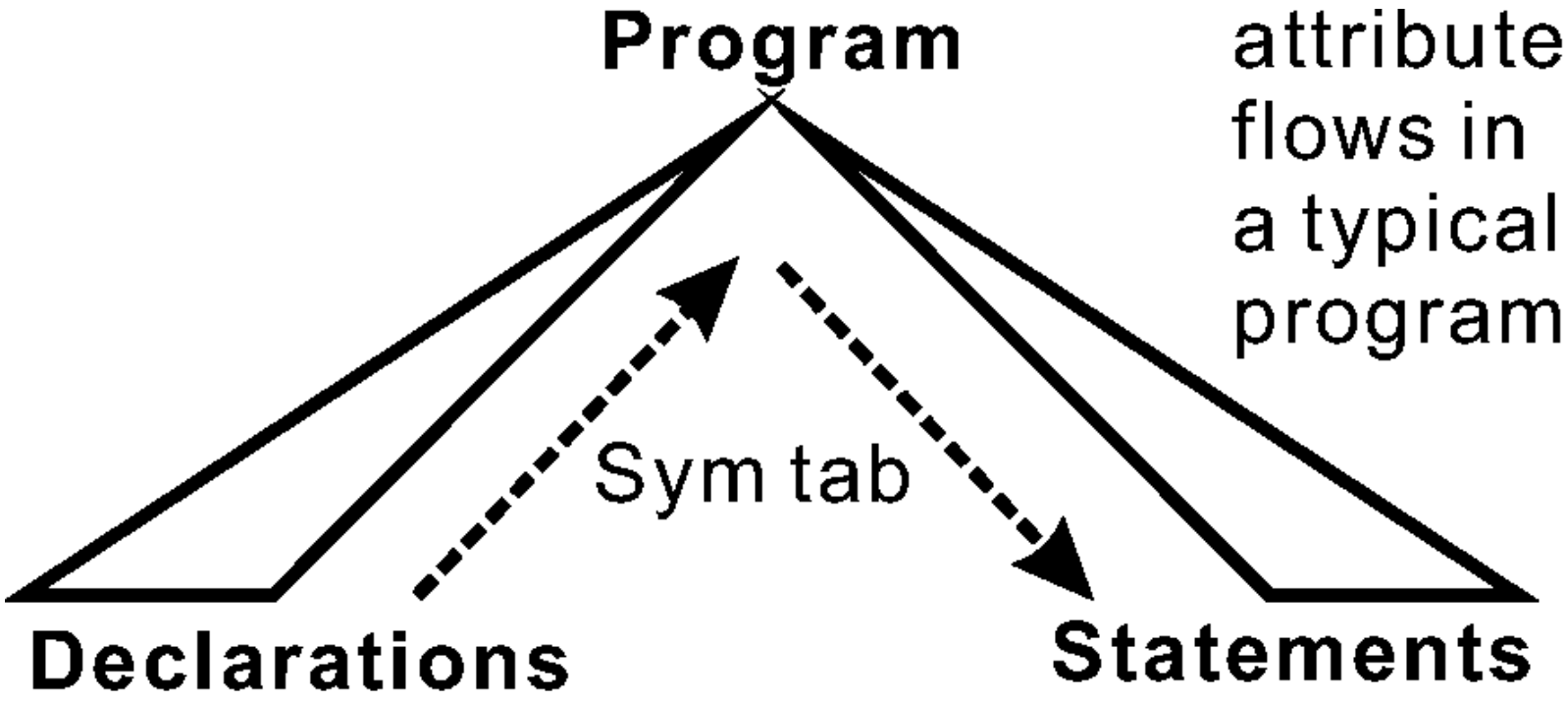
		Production	Semantic actions
1. $Start$	$\rightarrow$	$SetBase\ Digs\ \$$	$\{Digs.b := SetBase.b\}$
			$\{print(Digs.val)\}$
2. $Digs$	$\rightarrow$	$Digs_1\ d$	$\{Digs_1.b := Digs.b\}$
			$\{\textbf{if}\ d.val \geq Digs.b\ \textbf{then}\ error()$
			$\textbf{else}\ Digs.val := Digs_1.val * Digs.b + d.val\}$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

Example. Consider input *xdddddd*\$.



Exercise. Suppose we change the 2nd rule from "$Digs \to Digs_1\ d$" to "$Digs \to d\ Digs_1$". Please modify the previous attribute grammar appropriately.

This grammar includes both synthesized and inherited attributes. It is typical in a conventional programming language, which includes both 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 //return value=synth. attr
    switch (...)
    case ts.peek() == num:
        call match(num)
        answer := num.valueof() // semantic value of num
        return(answer)          // synthesized attribute
    case ts.peek() == lparen:
```

```
            call match(lparen)
            answer := Expr()     // semantic value of Expr
            call match(rparen)
            return(answer)       // synthesized attribute
    end

    function Expr() returns int
        switch (...)
        case ts.peek() == plus:
            call match(plus)
            op1 := Value()    // from bottom up
            op2 := Value()    // from bottom up
            answer := op1 + op2   // from bottom up
            return(answer)    // synthesized attribute
        case ts.peek() == prod:
            call match(prod)
            answer := Values(1) // semantic value of Values
            return(answer)       // synthesized attribute
```

```
    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)     // synthesized attribute
        case ts.peek() == rparen:
            answer := thusfar  // semantic value of Values
            return(answer)     // synthesized attribute
    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 *thusfar* in the *Values* procedure) 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. It should computes $31 + (((1 \times 20) \times 7) \times 50)$ during evaluation.

Exercise. Which formula did the above algorithm evaluate?
(1) $31 + (((1 \times 20) \times 7) \times 50)$
(2) $31 + (20 \times (\times 7 \times (50 \times 1)))$

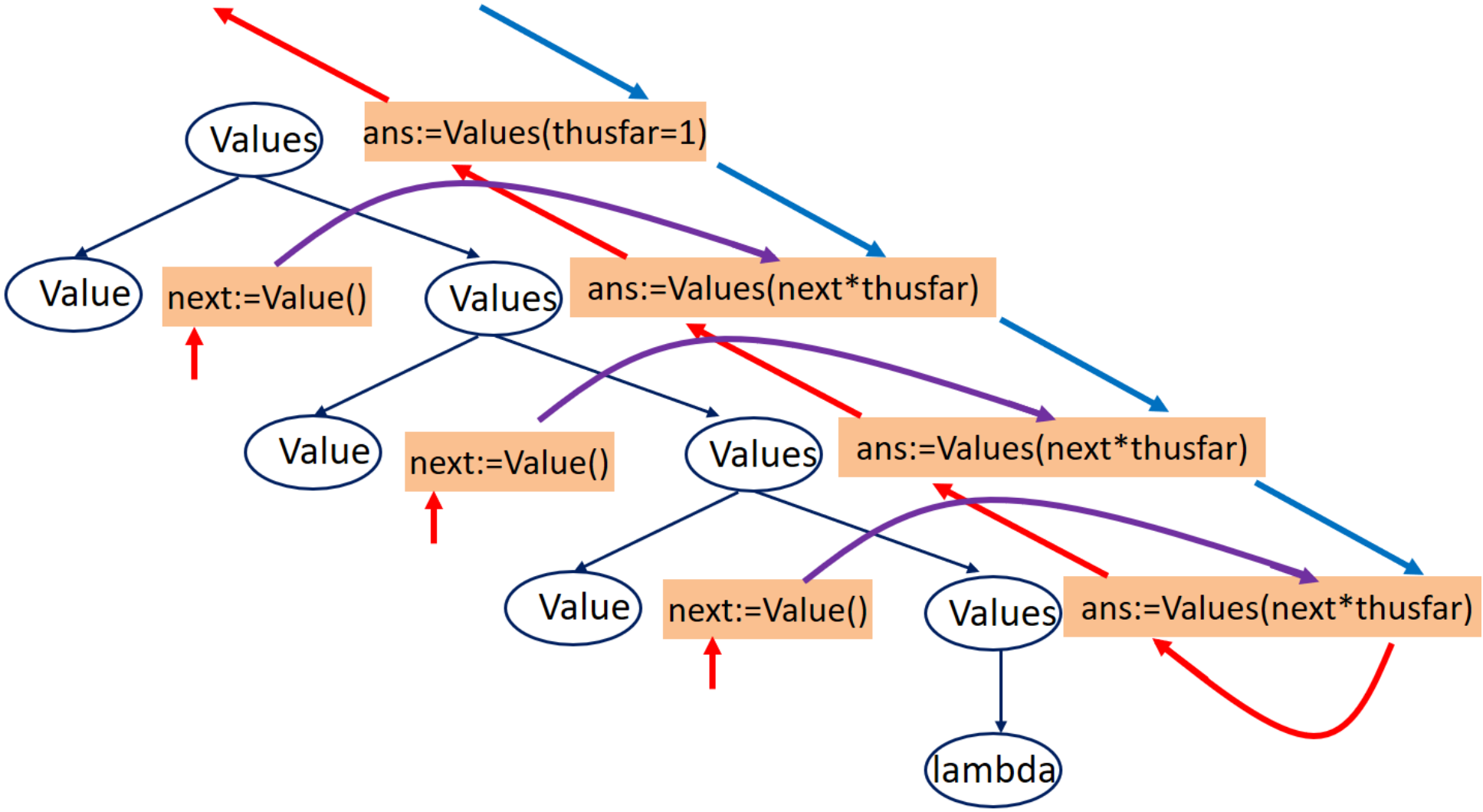


Figure 6: An abstract synatx tree for the *Values* nonterminal.

Exercises. Consider the following grammar:

```
Start ::= Value $
Expr ::= plus Value Value
Expr ::= prod Values
Value ::= num
Value ::= lparen Expr rparen
Values ::= Value Values
Values ::=  \lambda
```

Add attributes and their equations to calculate (1) the number of nodes; (2) the number of leaf nodes; (3) the number of internal nodes in the syntax tree.

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 could be 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  { $$ = lookup(symbolTable, $1); };
   /* Note $1 is a char* and $$ is an int. */
```

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; };
```

2nd example.

```
%{
typedef struct VStype {
    int ans;
    int thus;
} VStypetype;
%}

%union {
    int an;
    struct VStype vsn;
}
%type<an>  V NUM
%type<vsn>  VS
%%
VS : V VS
    { $2.thus = $$.thus * $1; $$.ans = $2.ans; }  ;
```

Connection between lex and yacc

lex passes two pieces of information to yacc:

- the token number: use `return`
- the string (or value) of the token: use `yylval`.

```
[0-9][0-9]* {
   fprintf(stderr, "num...%d.%s.\n", lineCount,yytext);
   yylval.an = atoi(yytext);
   return(NUM);
   }
-------------------------------------------------
V : NUM    { $$ = $1;    }
```

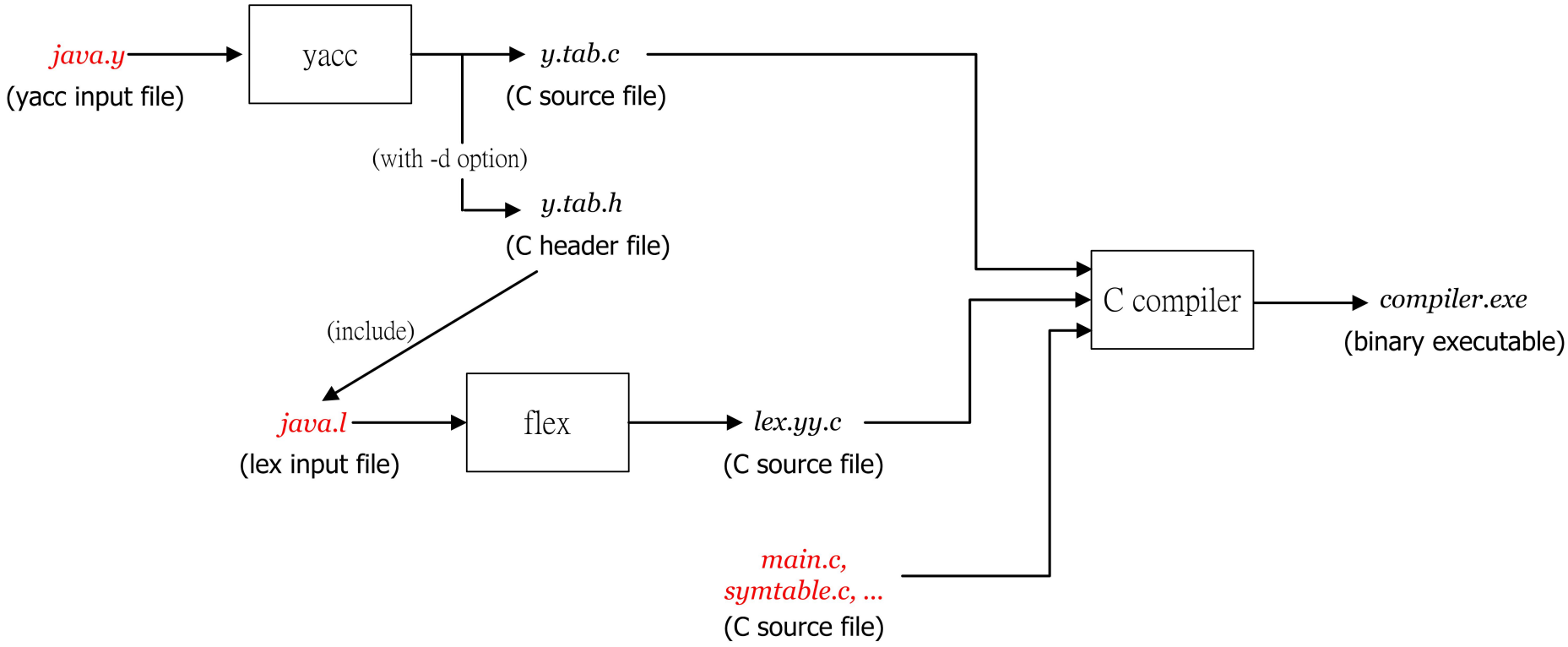


Figure 7: Connection between lex and yacc.

§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.

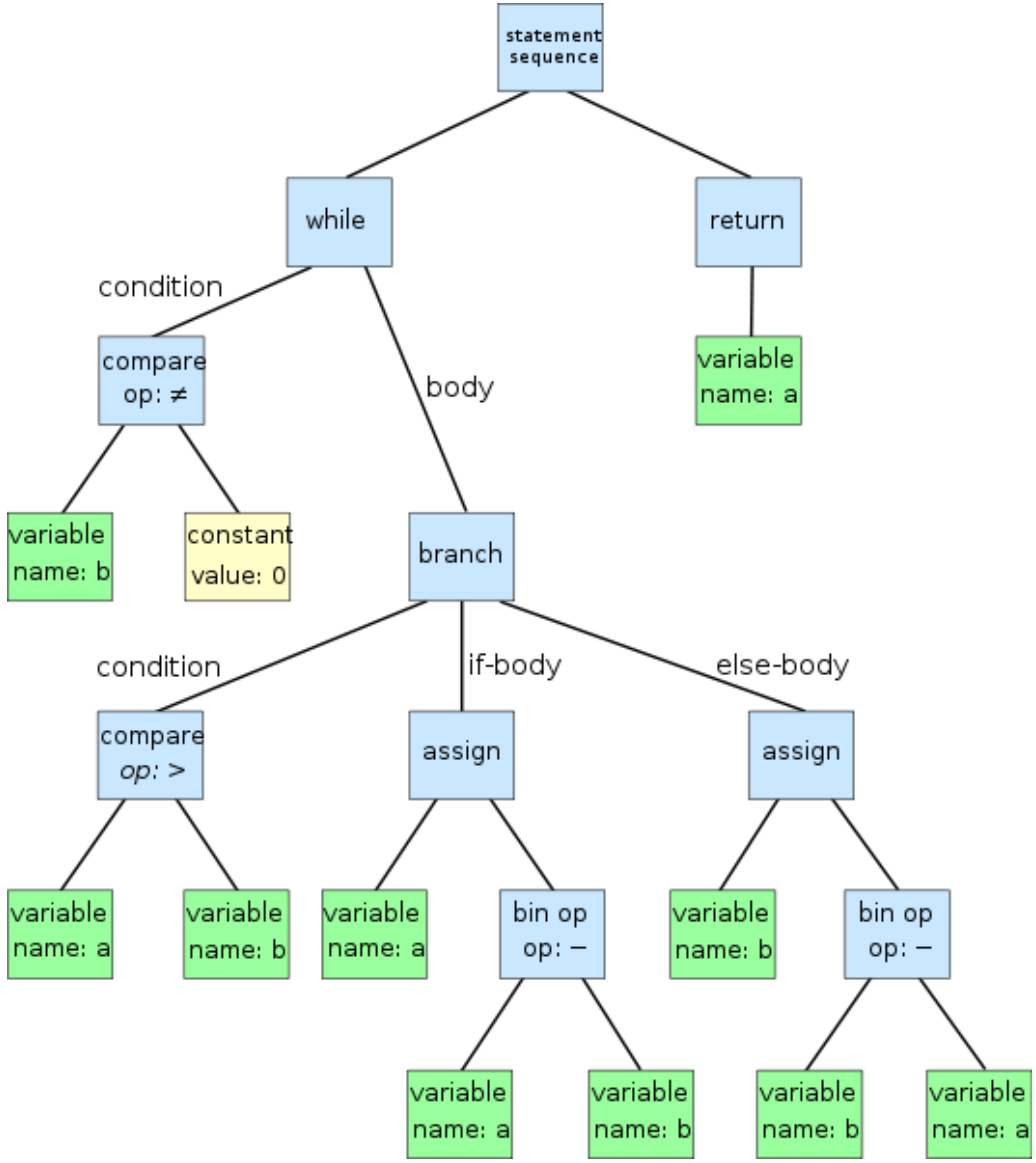


Figure 8: An abstract synatx tree.

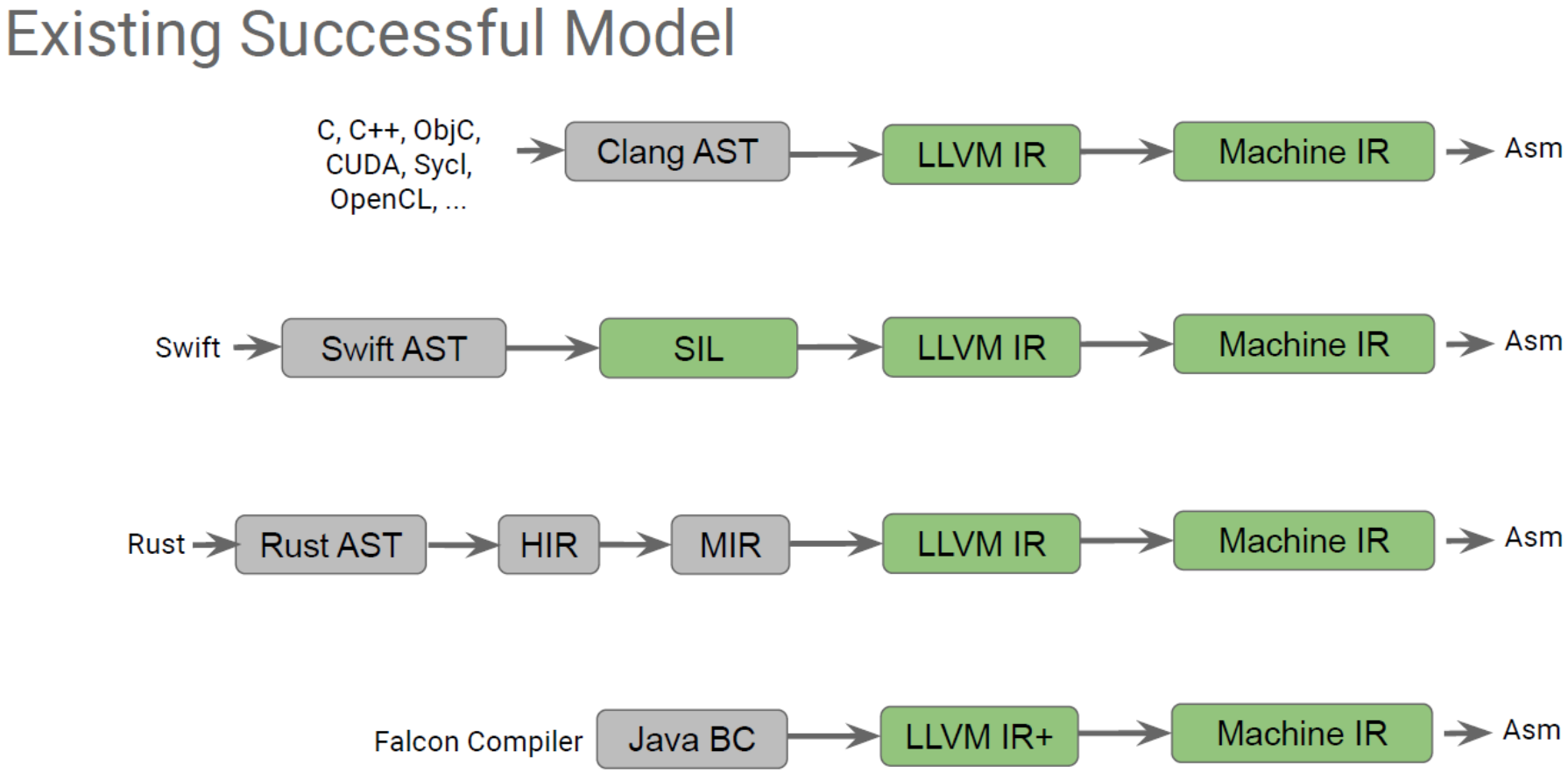


Figure 9: Intermediate Representations.

§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.

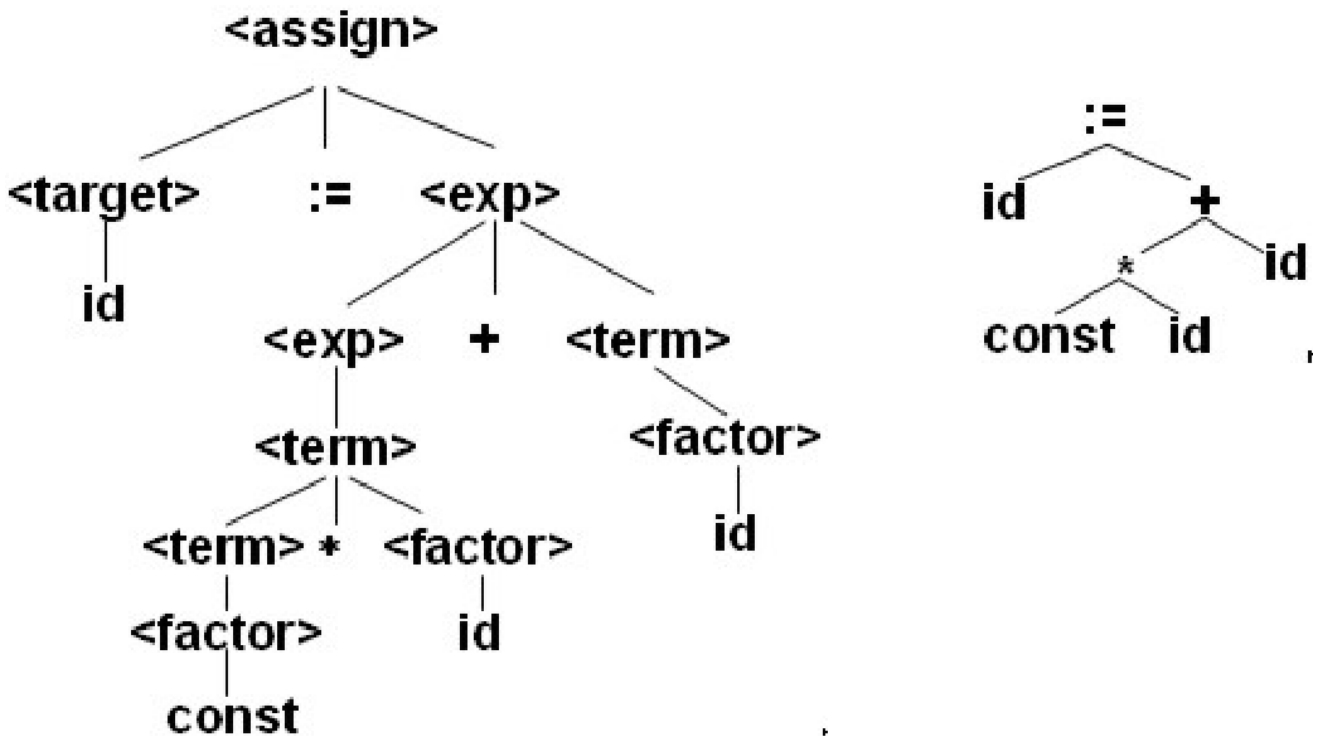


Figure 10: Parse tree vs. AST

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

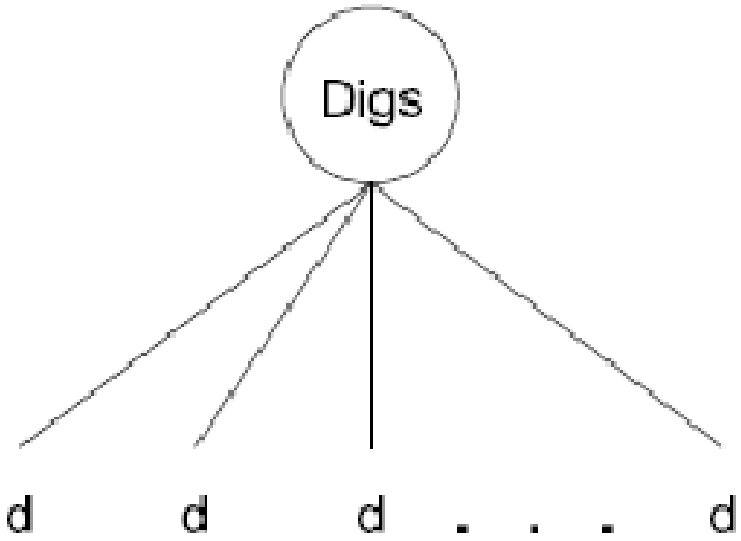
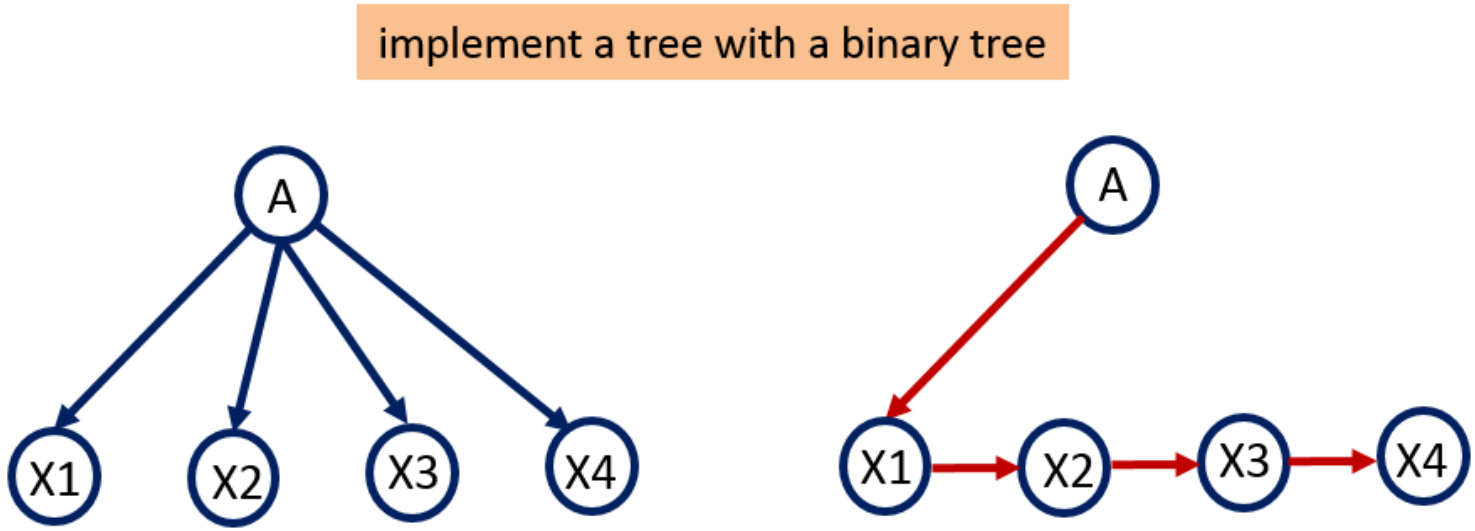


Figure 7.11: Abstract syntax tree for Digs.

Implement a tree with a binary tree. Every node includes exactly two pointers.



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.

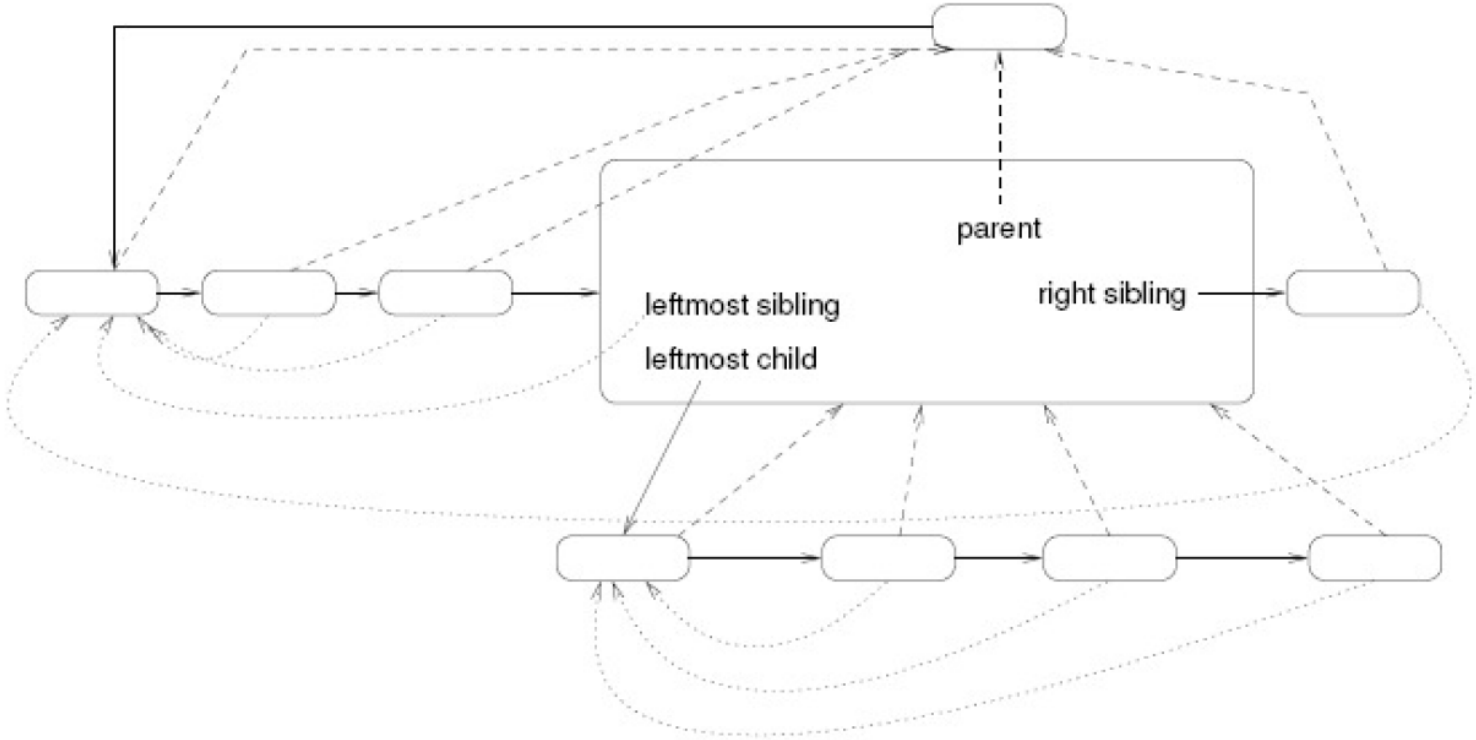


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

§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

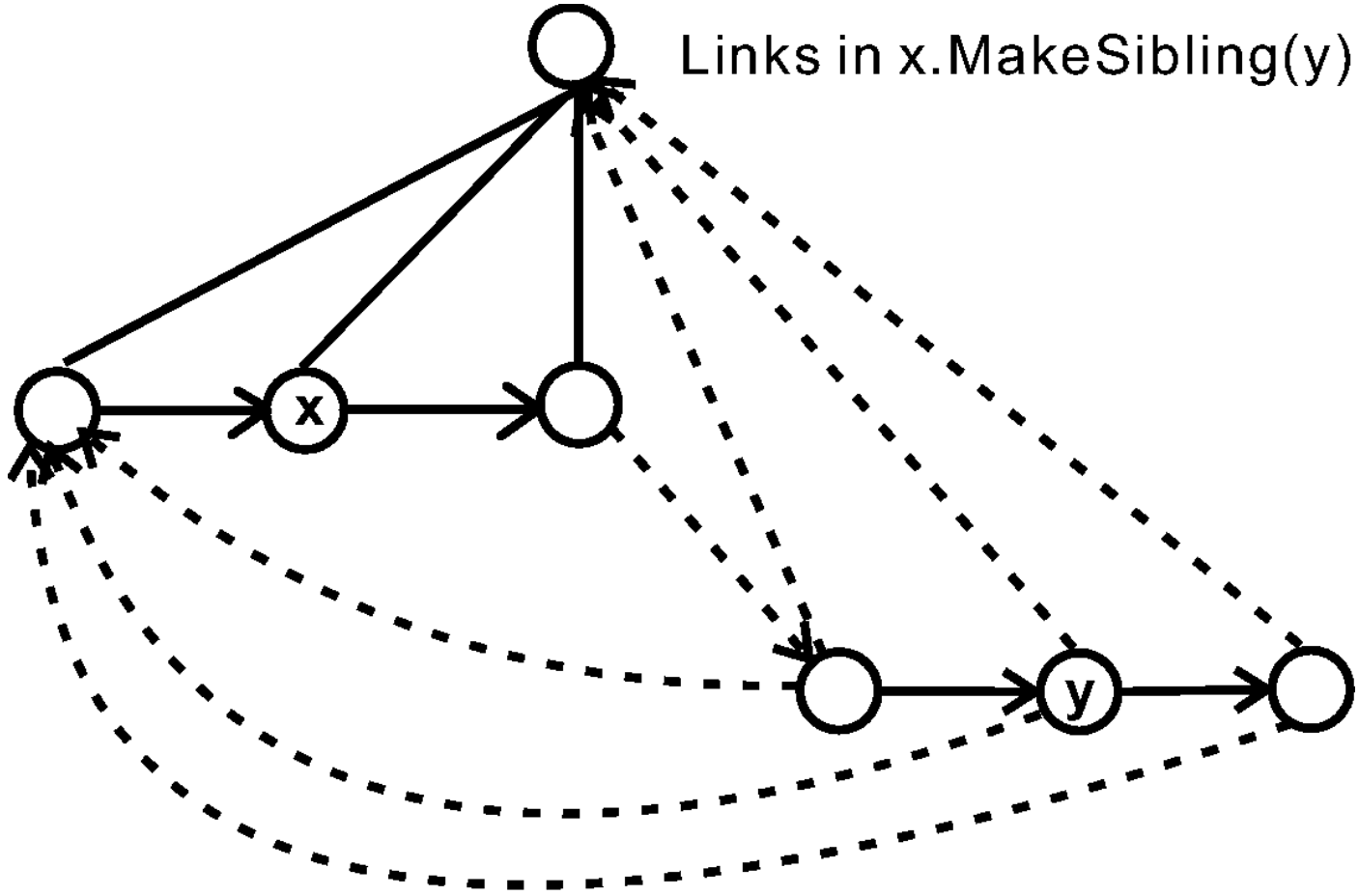
$$MakeNode(op).AdoptChildren(kid1.MakeSiblings(kid2))$$

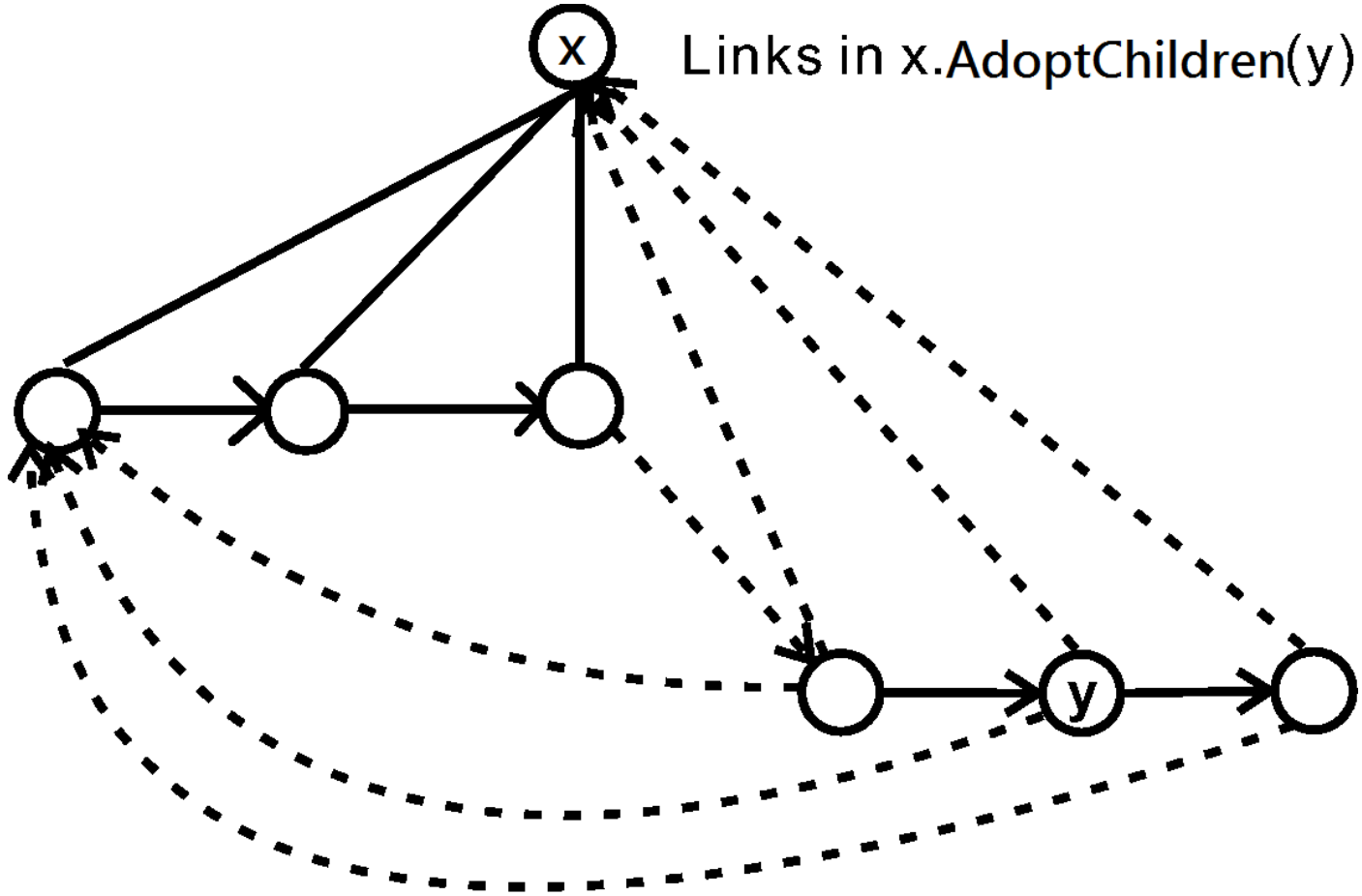
Figure 7.13 shows two methods for buidling ASTs.

```
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
```

Figure 7.13 Methods for building an AST


Links in x.MakeSibling(y)
x
y


x
Links in x.AdoptChildren(y)
y

```
/*   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.

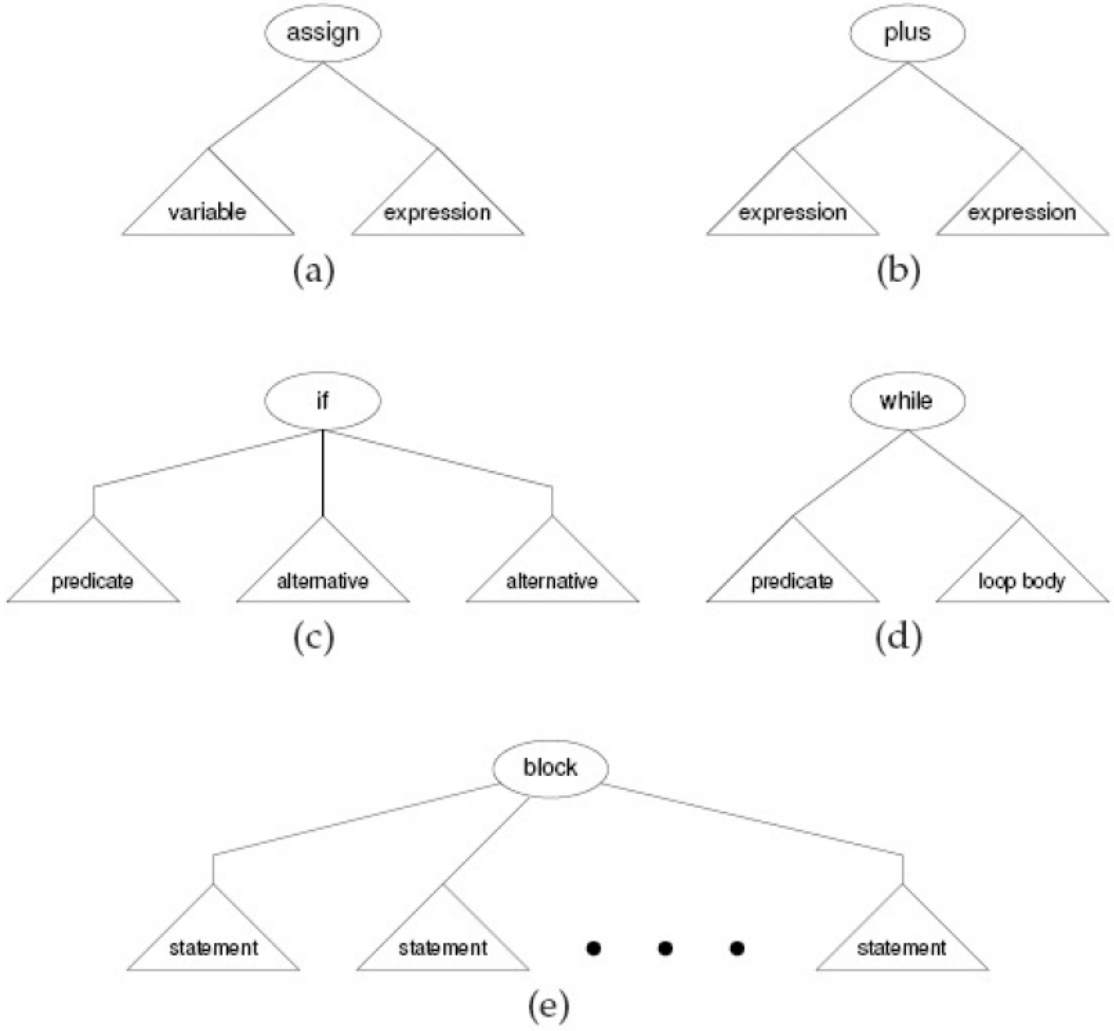


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

Examples of assignment statements:

```
x := y
```

```
a[b+c*d/3].fooptr->m = x+y[3*z+w]->qoo/3.14;
```

Figure 7.16 shows a concrete and an abstract syntax trees for $a + 5$.

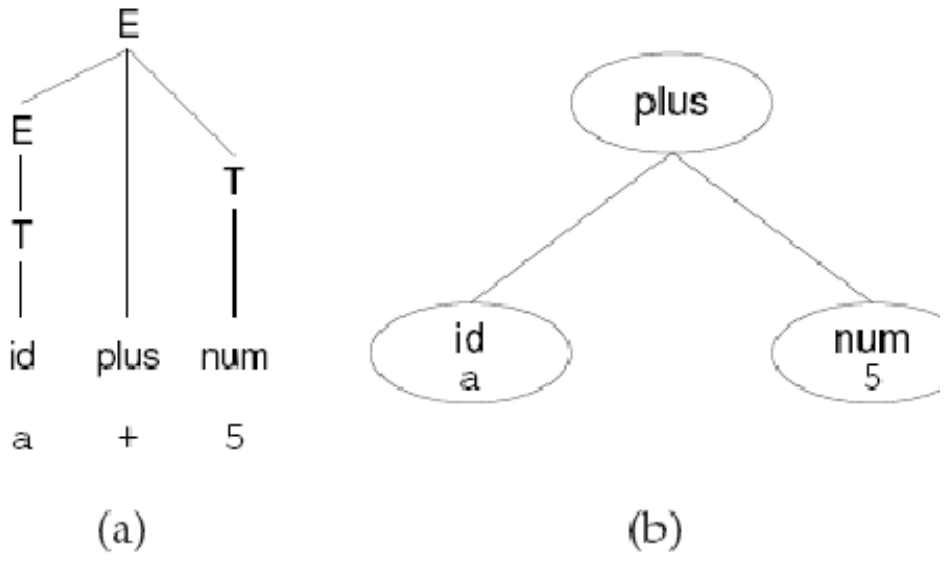


Figure 7.16: (a) Derivation of a + 5 from E;
(b) Abstract representation of a + 5.

Figure 7.17 shows the semantic actions for building the AST.

1. *Start* → *Stmt* $
   . { return(Stmt.result) }
2. *Stmt* → *id assign E*
   . { Stmt.result := MakeFmaily(assign, MakeNode(id.name), E.expr)
   }
3. *Stmt* → *if lparen E rparen Stmt fi*
   . { Stmt.result := MakeFmaily(if, E.expr, Stmt_2.result,
   MakeNode()) }
4. *Stmt* → *if lparen E rparen Stmt else Stmt fi*
   . { Stmt.result := MakeFmaily(if, E.expr, Stmt_2.result,
   Stmt_3.result) }
5. *Stmt* → *while lparen E rparen do Stmt od*
   . { Stmt.result := MakeFmaily(while, E.expr, Stmt_2.result) }
6. *Stmt* → *begin Stmts end*
   . { Stmt.result := MakeFmaily(block, Stmts.list) }
7. *Stmts* → *Stmts semi Stmt*

. { Stmts.list := Stmts_2.list.MakeSiblings(Stmt.result) }

8. $Stmts \rightarrow Stmt$

. { Stmts.list := Stmt.result }

9. $E \rightarrow E\ plus\ T$

. { E.expr := MakeFamily(plus, E_2.expr, T.expr) }

10. $E \rightarrow T$

. { E.expr := T.expr }

11. $T \rightarrow id$

. { T.expr := MakeNode(id.name) }

12. $T \rightarrow num$

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

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

```
1  Start          → Stmt_ast $
                    return (ast)
2  Stmt_result    → id_var assign E_expr
                    result ← F (assign, var, expr)
3                 | if lparen E_p rparen Stmt_s fi
                    result ← F (if, p, s, N ())
4                 | if lparen E_p rparen Stmt_s1 else Stmt_s2 fi
                    result ← F (if, p, s1, s2)
5                 | while lparen E_p rparen do Stmt_s od
                    result ← F (while, p, s)
6                 | begin Stmts_list end
                    result ← F (block, list)
7  Stmts_result   → Stmts_sofar semi Stmt_next
                    result ← sofar. S (next)
8                 | Stmt_first
                    result ← first
9  E_result       → E_e1 plus T_e2
                    result ← F (plus, e1, e2)
10                | T_e
                    result ← e
11 T_result       → id_var
                    result ← N (var)
12                | num_val
                    result ← N (val)
```

Figure 7.17: Semantic actions for grammar in Figure 7.14.

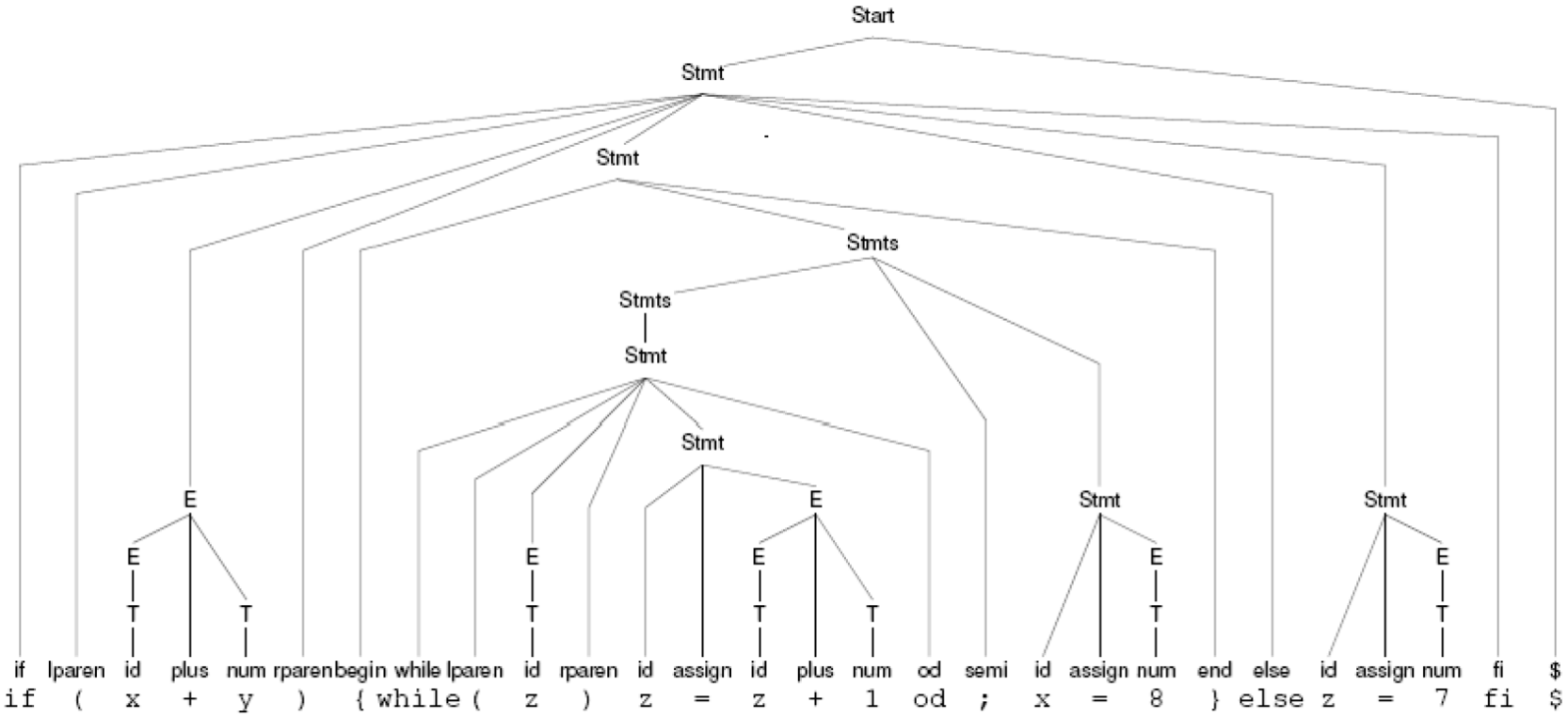


Figure 7.18: Concrete syntax tree.

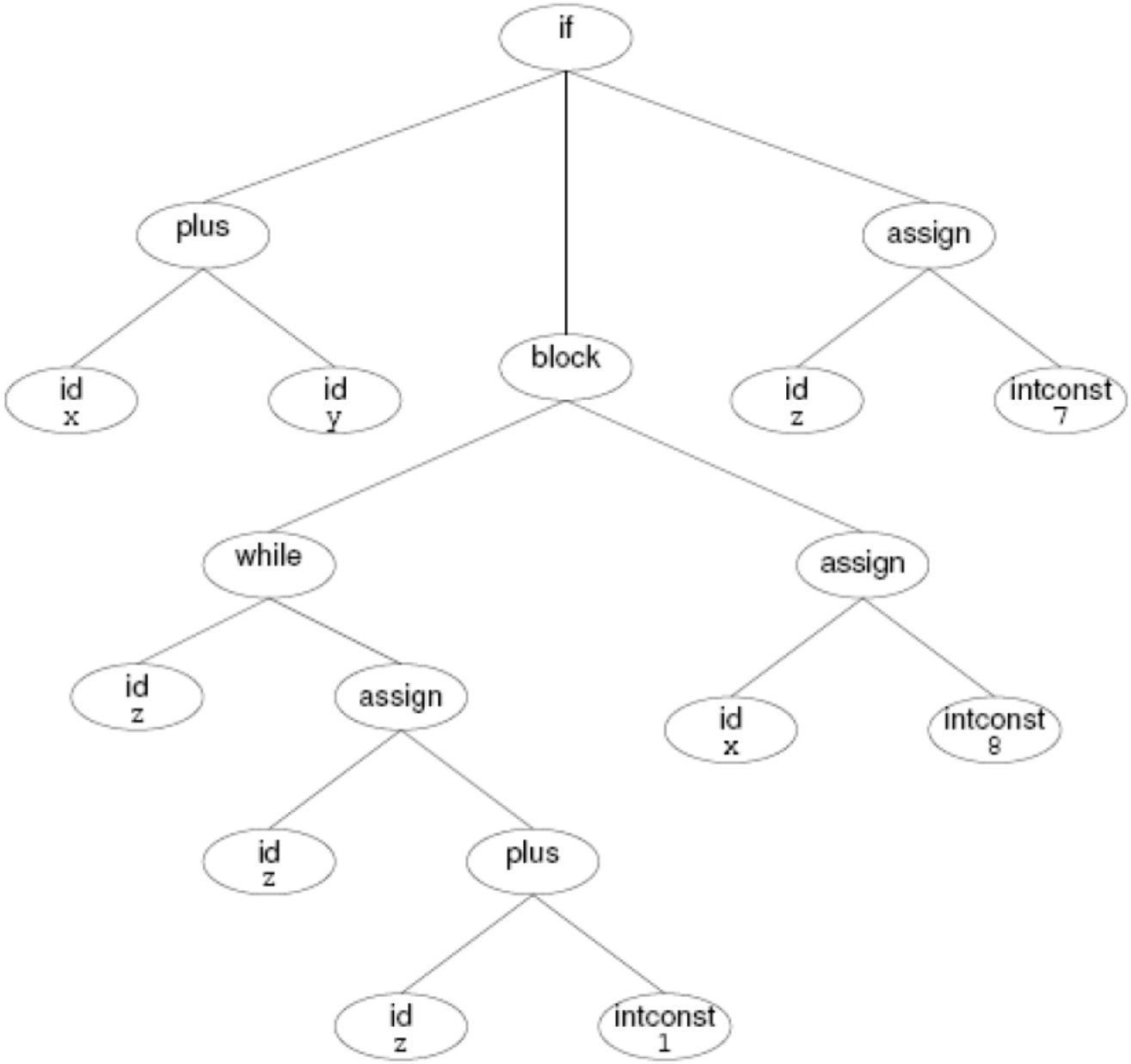


Figure 7.19: AST for the parse tree in Figure 7.18.

Additional fragments of abstract syntax trees can be found in Chapters 8 and 9. They are for variable and type declarations, class and method declarations, expressions and statements, etc.

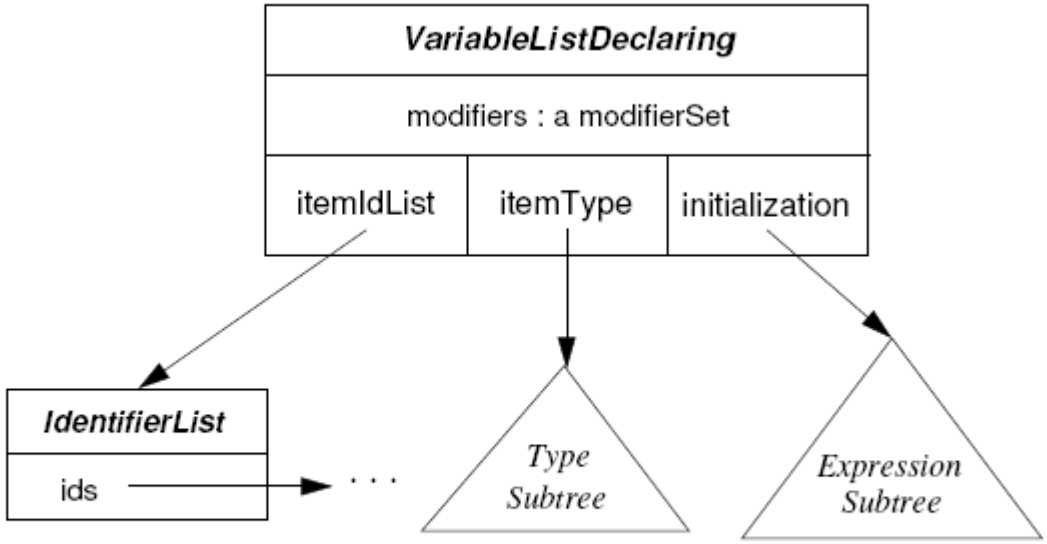


Figure 8.17: AST for Generalized Variable Declarations

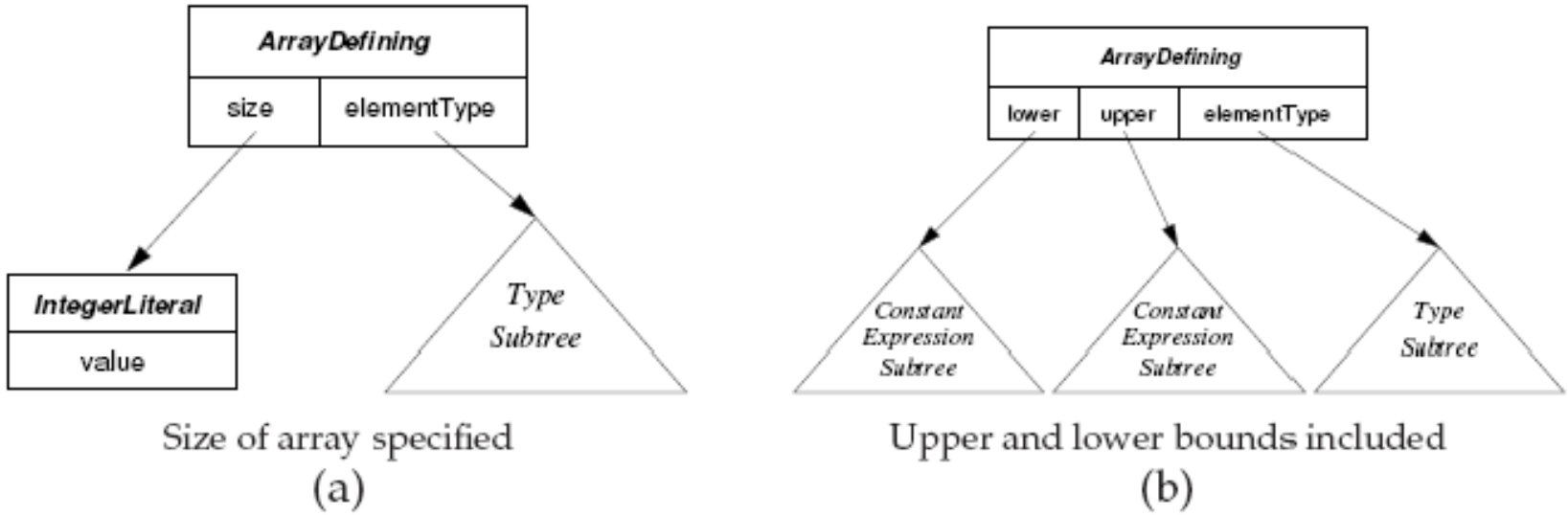


Figure 8.19: Abstract Syntax Trees for Array Definitions

§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.
   Relocation is usually invisible to the program.
   Some programming languages provide a *dereference* 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. (`store`)
2. `*p = 0` Here `p` generates a right value. (`load/store indirect`)
3. `x = p` Here `p` generates a right value. (`load/store`)
4. `x = *p` Here `p` generates a right value. (`load/load indirect/store`)
5. `x = &p` Here `&p` generates a left value. (`store`)

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. $Start \to Stmt\ \$$
2. $Stmt \to L\ assign\ R$
3. $L \to id$
4. $L \to *\ R$
5. $R \to L$
6. $R \to num$
7. $R \to \&\ L$

Figure 7.20 Grammar for the left and right values

We may write `***p`, `&p`, `*&p`, etc.

Question. How to modify the AG on page 97 so that it will allows statemetns such as "$x = (*p)(y)$"?

Note that it is wrong to assume all variables on the left of the assignment operator provide left values. For example, in the following expression:

```
A[p+q] := ...
```

Both `p` and `q` provide right values while `A` and `A[p+q]` provide left values.

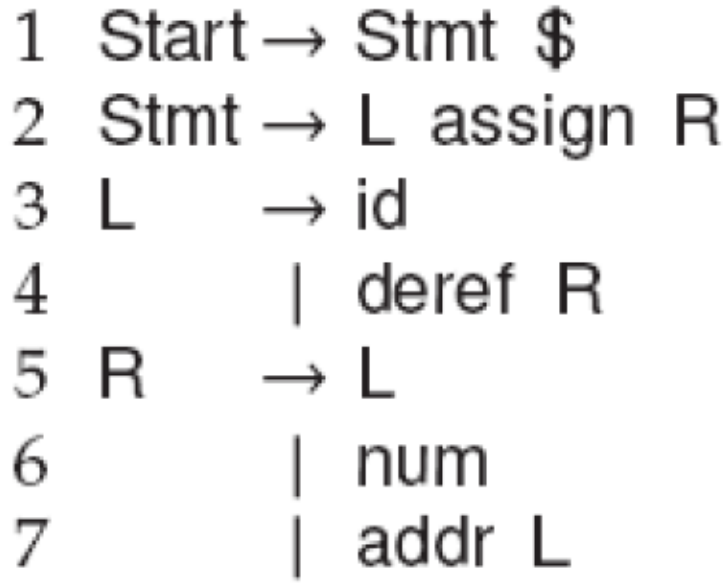


Figure 7.20: Grammar for left and right values.

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. $L \rightarrow * R$
. L.target := R.value
5. $R \rightarrow L$
. R.value := MakeFamily(deref, L.target)
6. $R \rightarrow num$
. R.value := MakeNode(num.value) // do not de-reference L
7. $R \rightarrow \& L$
. R.value := L.target // do not de-reference L

Figure 7.21 Semantic actions for the grammar in Figure 7.20

```
1 Start      → Stmt_ast $
                 return (ast)

2 Stmt_result → L_target assign R_source
                 result ←        F       (assign, target, source)

3 L_result   → id_name
                 result ←        N       (name)

4            | deref R_val
                 result ← val                                        (25)

5 R_result   → L_val
                 result ←        F       (deref, val)                (26)

6            | num_val
                 result ←        N       (val)

7            | addr L_val
                 result ← val                                        (27)
```

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

Exercise. Add production rules and attribute equations to the attribute grammar in Fig 7.21. These production rules are used for invoking a procedure with a parameter. Procedure calls may appear on both sides of an assignment sign. For example,

```
x = foo(23);
x = *foo(y);
*qoo(z) = 23;
```

8. $R \rightarrow\ CALL$

. R.value := CALL.value

9. $CALL \rightarrow id\ (\ R\ )$

. CALL.value := MakeFamily(calling, MakeNode(id.name), R.value)

Figure 7.21-2 Add procedure invocations to the grammar in Figure 7.21

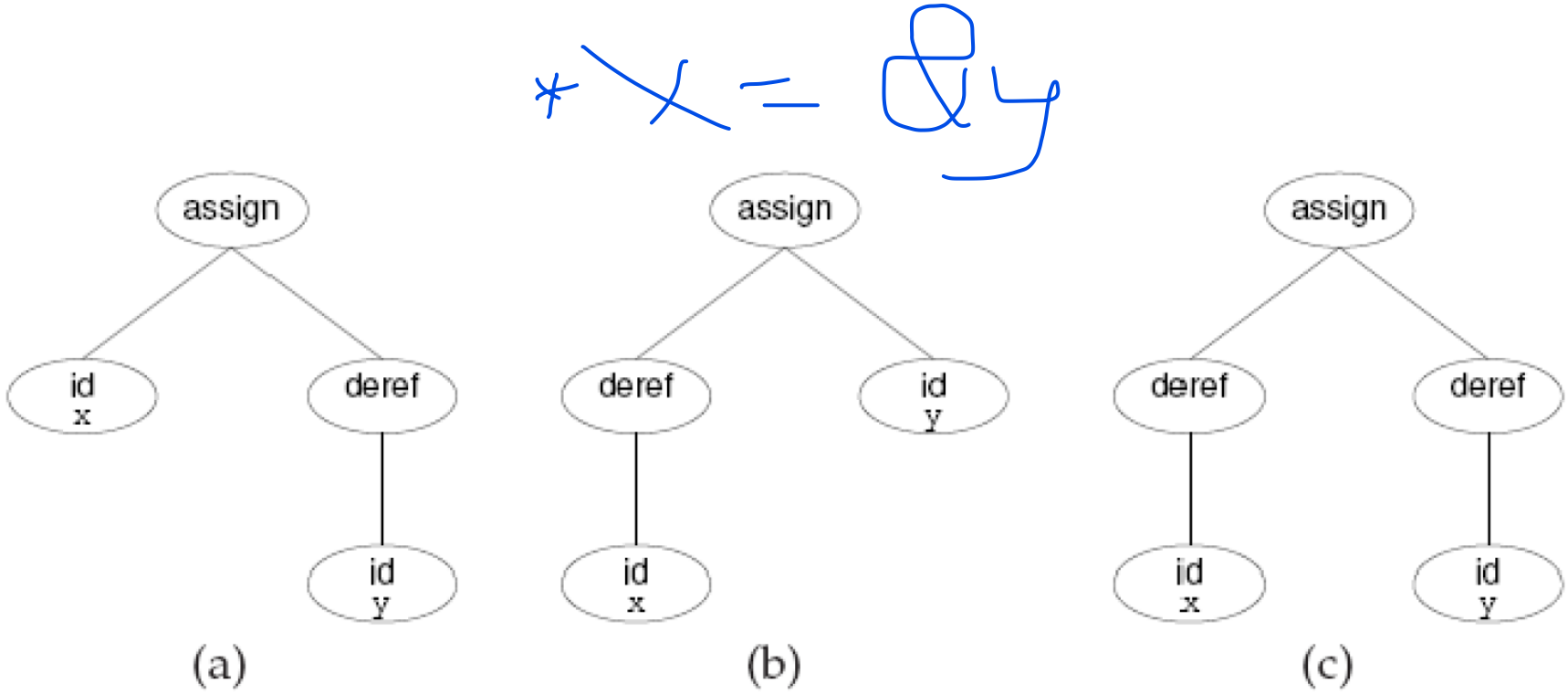


Figure 7.22: ASTs illustrating left and right values for the assignments:
(a) x = y
(b) x = &y
(c) ⋆x = y

Note that (b) should be "`*x = &y`".

Example 1. Consider the statement:

$$^*\mathrm{x} = \mathrm{y}$$

The parsing steps are

$Stmt \Rightarrow^{s1} L_1\ assign\ R_1 \Rightarrow^{s2} *R_2\ assign\ R_1 \Rightarrow^{s3} *L_2\ assign\ R_1 \Rightarrow^{s4}$
$*idx\ assign\ R_1 \Rightarrow^{s5} *idx\ assign\ L_3 \Rightarrow^{s6} *idx\ assign\ idy$

Attribute computation is

$s1 : stmt.ast = MakeFamily(assign, L_1.target, R_1.value)$

$s2 : L_1.target = R_2.value$

$s3 : R_2.value = MakeFamily(deref, L_2.target)$

$s4 : L_2.target = MakeNode(idx)$

$s5 : R_1.value = MakeFamily(deref, L_3.target)$

$s6 : L_3.target = MakeNode(idy)$

Draw the ast.

Example 2. Consider the statement:

$$\text{x} = {}^*\text{y}$$

The parsing steps are

$Stmt \Rightarrow^{s1} L_1\ assign\ R_1 \Rightarrow^{s2} idx\ assign\ R_1 \Rightarrow^{s3} idx\ assign\ L_2 \Rightarrow^{s4}$
$idx\ assign\ *R_2 \Rightarrow^{s5} idx\ assign\ *L_3 \Rightarrow^{s6} idx\ assign\ *idy$

Attribute computation is

$s1 : stmt.ast = MakeFamily(assign, L_1.target, R_1.value)$

$s2 : L_1.target = MakeNode(idx)$

$s3 : R_1.value = MakeFamily(deref, L_2.target)$

$s4 : L_2.target = R_2.value$

$s5 : R_2.value = MakeFamily(deref, L_3.target)$

$s6 : L_3.target = MakeNode(idy)$

Draw the ast.

Example 3. Consider the statement:

$$^*\mathrm{x} = {}^*\mathrm{y}$$

The parsing steps are

$Stmt \Rightarrow^{s1} L_1\ assign\ R_1 \Rightarrow^{s2} *R_2\ assign\ R_1 \Rightarrow^{s3} *L_2\ assign\ R_1 \Rightarrow^{s4}$
$*idx\ assign\ R_1 \Rightarrow^{s5} *idx\ assign\ L_3 \Rightarrow^{s6} *idx\ assign\ *R_4 \Rightarrow^{s7}$
$*idx\ assign\ *L_4 \Rightarrow^{s8} *idx\ assign\ *idy$

Attribute computation is

$s1 : stmt.ast = MakeFamily(assign, L_1.target, R_1.value)$
$s2 : L_1.target = R_2.value$
$s3 : R_2.value = MakeFamily(deref, L_2.target)$
$s4 : L_2.target = MakeNode(idx)$
$s5 : R_1.value = MakeFamily(deref, L_3.target)$
$s6 : L_3.target = R_4.value$
$s7 : R_4.value = MakeFamily(deref, L_4.target)$
$s8 : L_4.target = MakeNode(idy)$

Draw the ast.

Example. The following statements cannot be parsed.

(1) &x = ...

(2) ... = &&y

(3) ... = (*p)(123)

(3) (*p)(123) = ...

Example 4. Consider the statement:

$$\text{x} = \text{*\&y}$$

The parsing steps are

$Stmt \Rightarrow^{s1} L_1\ assign\ R_1 \Rightarrow^{s2} idx\ assign\ R_1 \Rightarrow^{s3} idx\ assign\ L_2 \Rightarrow^{s4}$
$idx\ assign\ *R_2 \Rightarrow^{s5} idx\ assign\ *\&L_3 \Rightarrow^{s6} idx\ assign\ *\&idy$

Attribute computation is

$s1 : stmt.ast = MakeFamily(assign, L_1.target, R_1.value)$

$s2 : L_1.target = MakeNode(idx)$

$s3 : R_1.value = MakeFamily(deref, L_2.target)$

$s4 : L_2.target = R_2.value$

$s5 : R_2.value = L_3.target$

$s6 : L_3.target = MakeNode(idy)$

This statement is exactly the same as "$x = y$".

Example 5. Consider the statement:

$$\text{x} = \&^*\text{y}$$

The parsing steps are

$Stmt \Rightarrow^{s1} L_1\ assign\ R_1 \Rightarrow^{s2} idx\ assign\ R_1 \Rightarrow^{s3} idx\ assign\ \&L_2 \Rightarrow^{s4}$
$idx\ assign\ \& * R_2 \Rightarrow^{s5} idx\ assign\ \& * L_3 \Rightarrow^{s6} idx\ assign\ \& * idy$

Attribute computation is

$s1 : stmt.ast = MakeFamily(assign, L_1.target, R_1.value)$

$s2 : L_1.target = MakeNode(idx)$

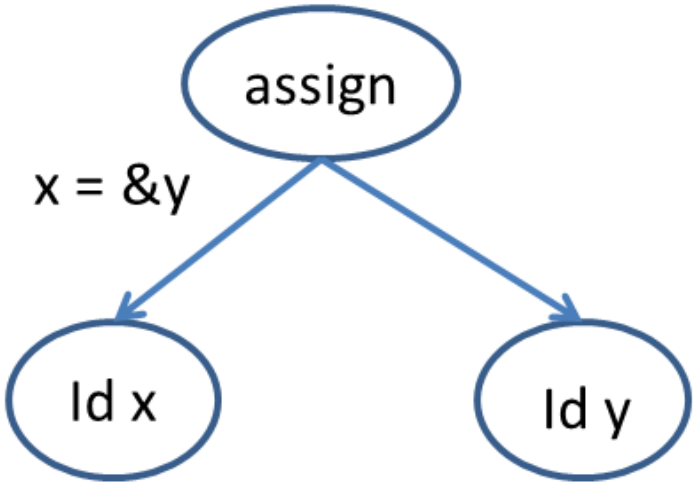
$s3 : R_1.value = L_2.target$

$s4 : L_2.target = R_2.value$

$s5 : R_2.value = MakeFamily(deref, L_3.target)$

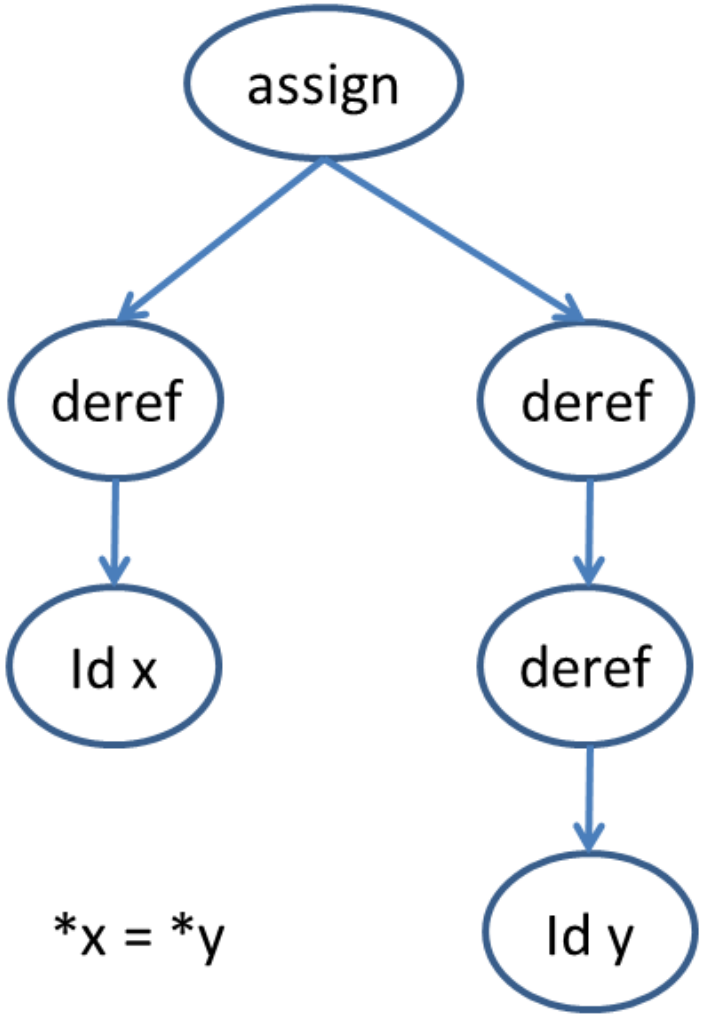
$s6 : L_3.target = MakeNode(idy)$

This statement is exactly the same as "$x = y$".



Instruction Sequence:

push x
push y
store



Instruction Sequence:

push x
fetch
push y
fetch
fetch
store

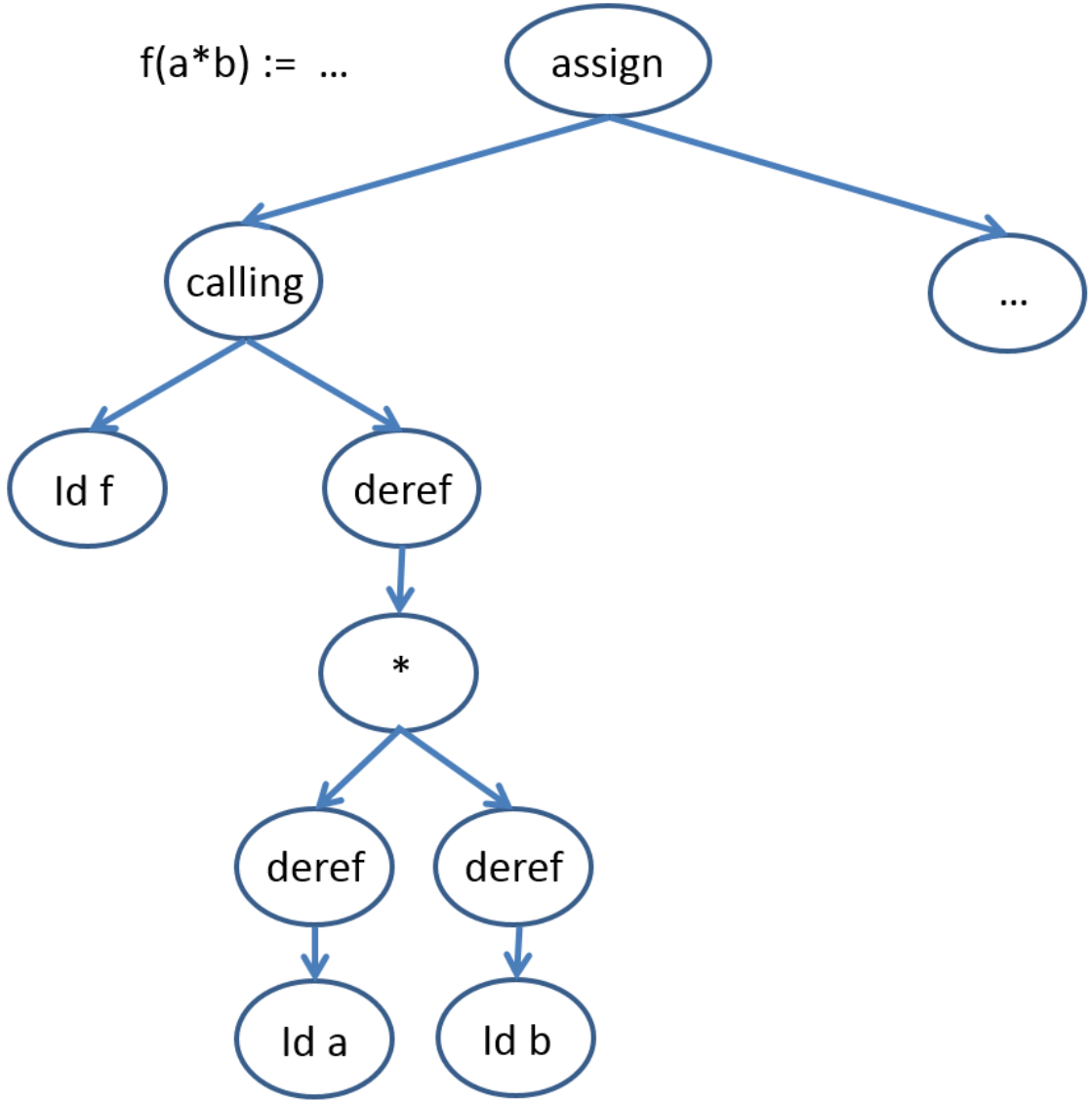


Figure 11: A syntax error. It is not possible to generate "`f(a*b) := ...`" with the given grammar.

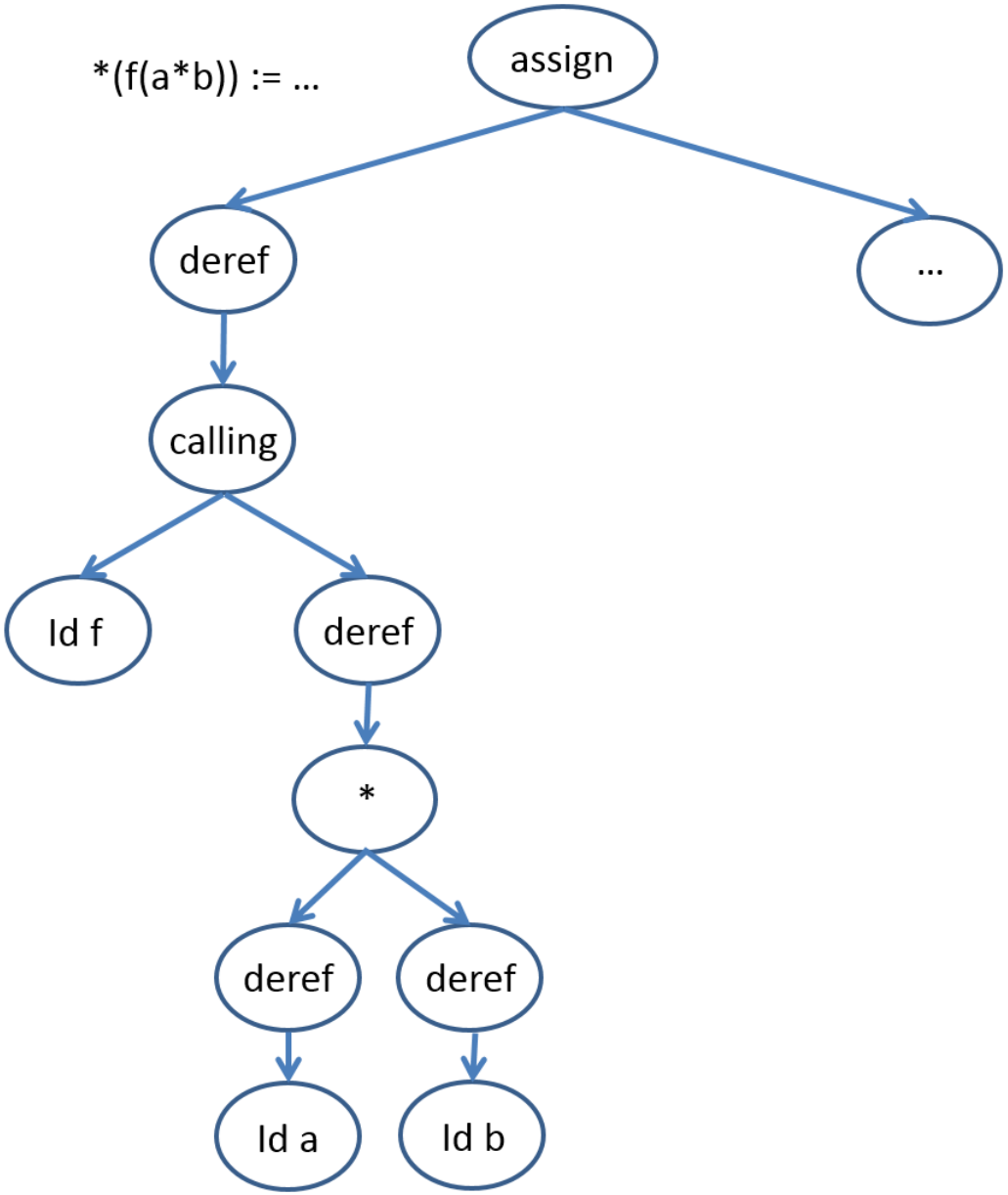


Figure 12: A wrong AST. There should not be the two `deref` nodes.

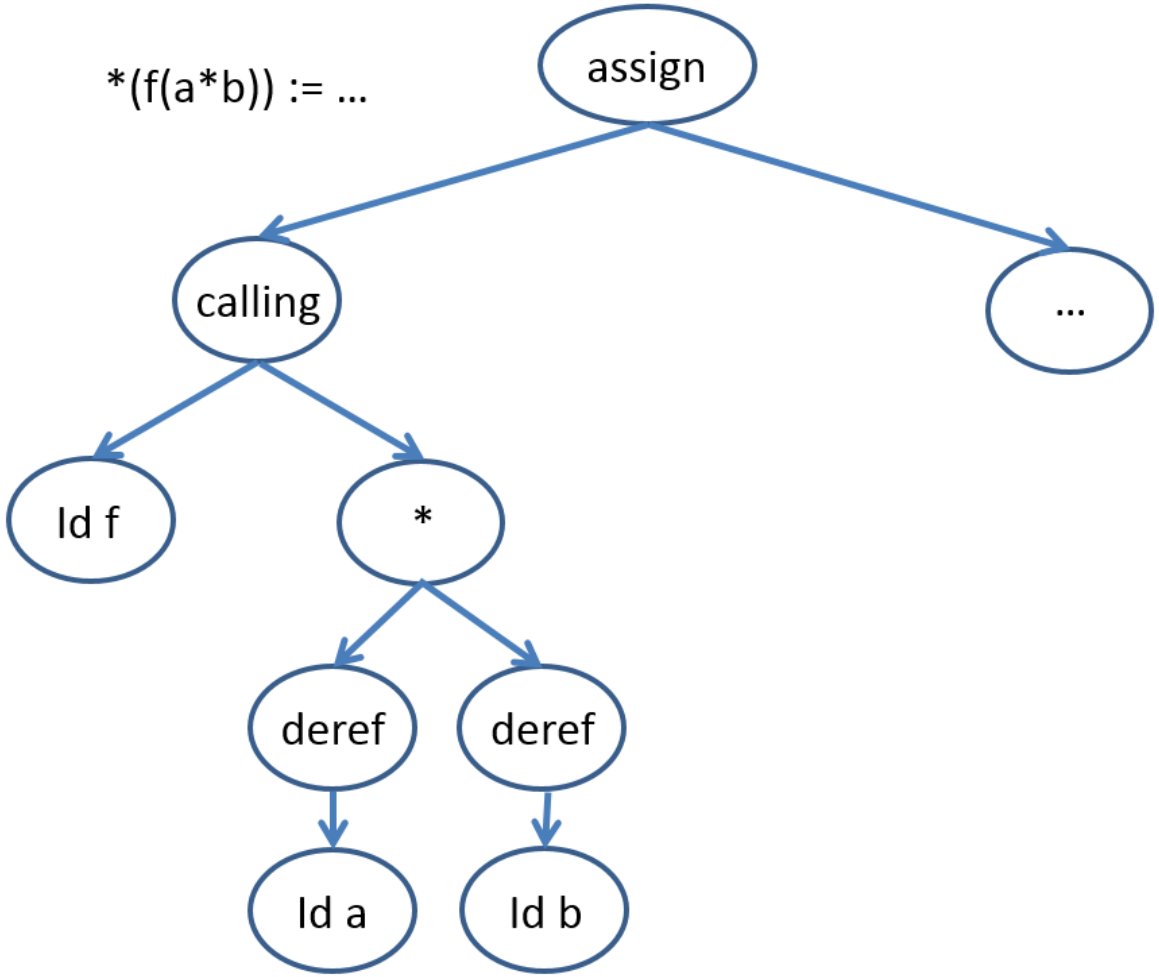


Figure 13: The correct AST though we did not define the * operator.

Exercise If we use Figure 14 to represent the AST of the statement, *(f(a*b)) = *(g(d)) + (*h)(m), which parts are wrong? (Strictly speaking, "(*h)(m)" causes a syntax error.)

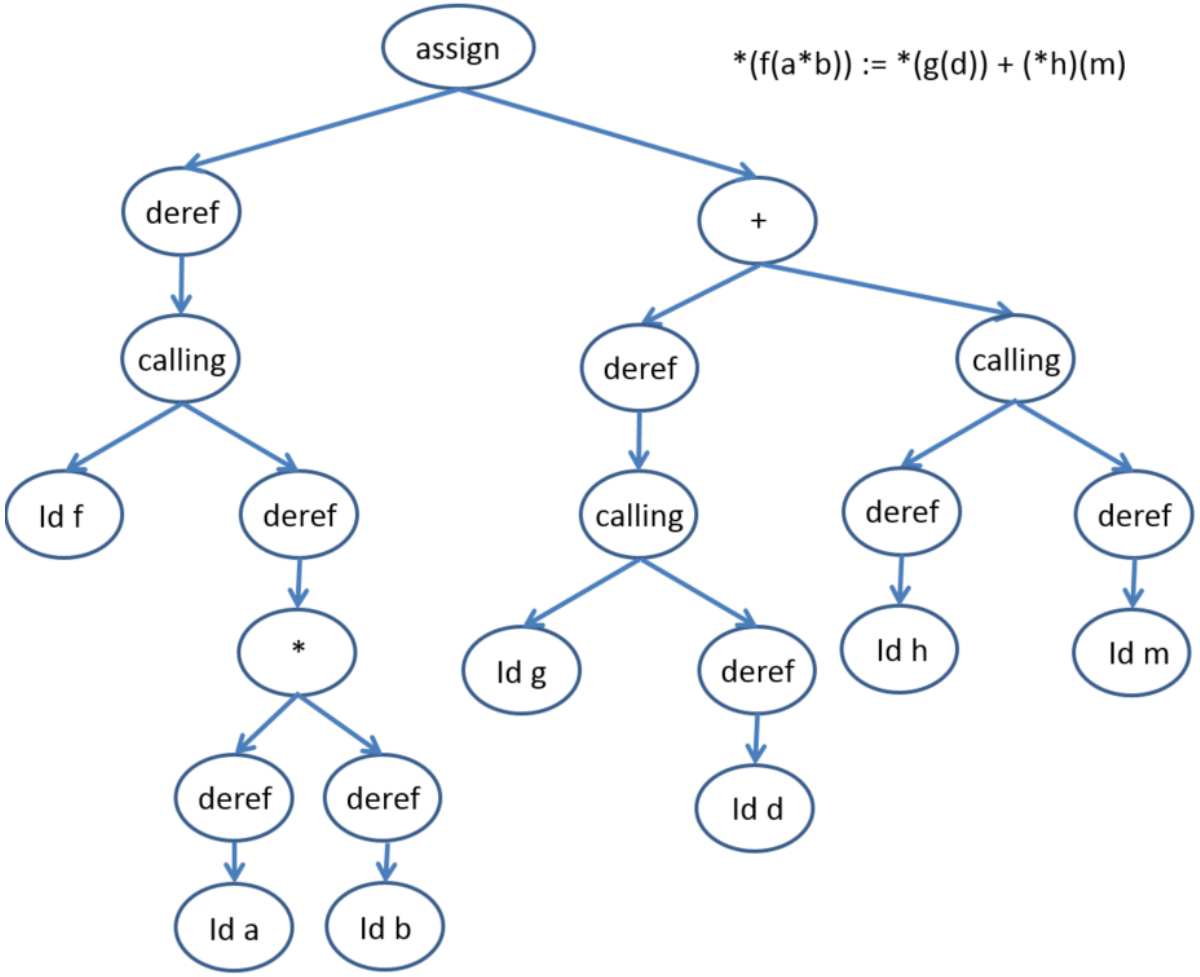


Figure 14: A wrong AST

*(f(a*b)) = *(g(d)) + (*h)(m)

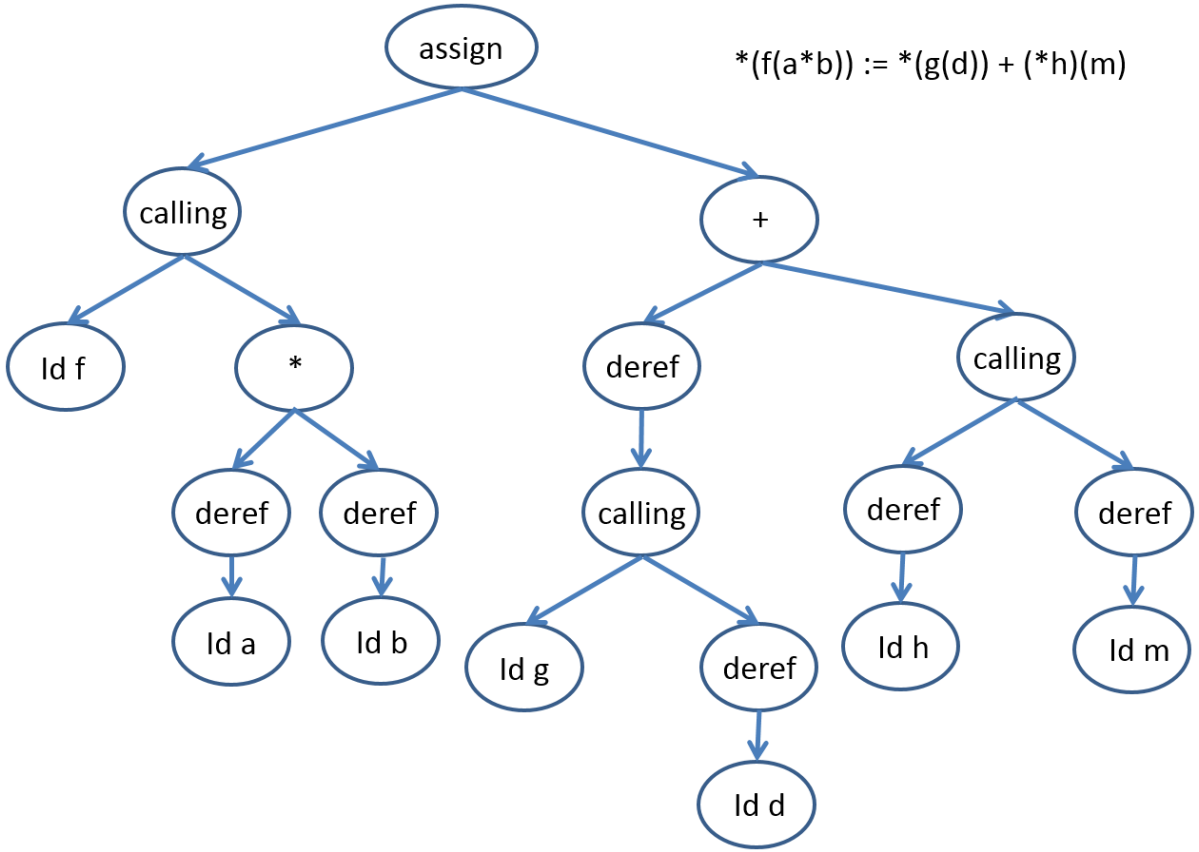


Figure 15: This is considered a correct AST though the grammar did not define expression and and * and + operators.

Example Draw the AST for the expression: *(f(a*b)) = *(g(d+e)) + (*h)(m/n).

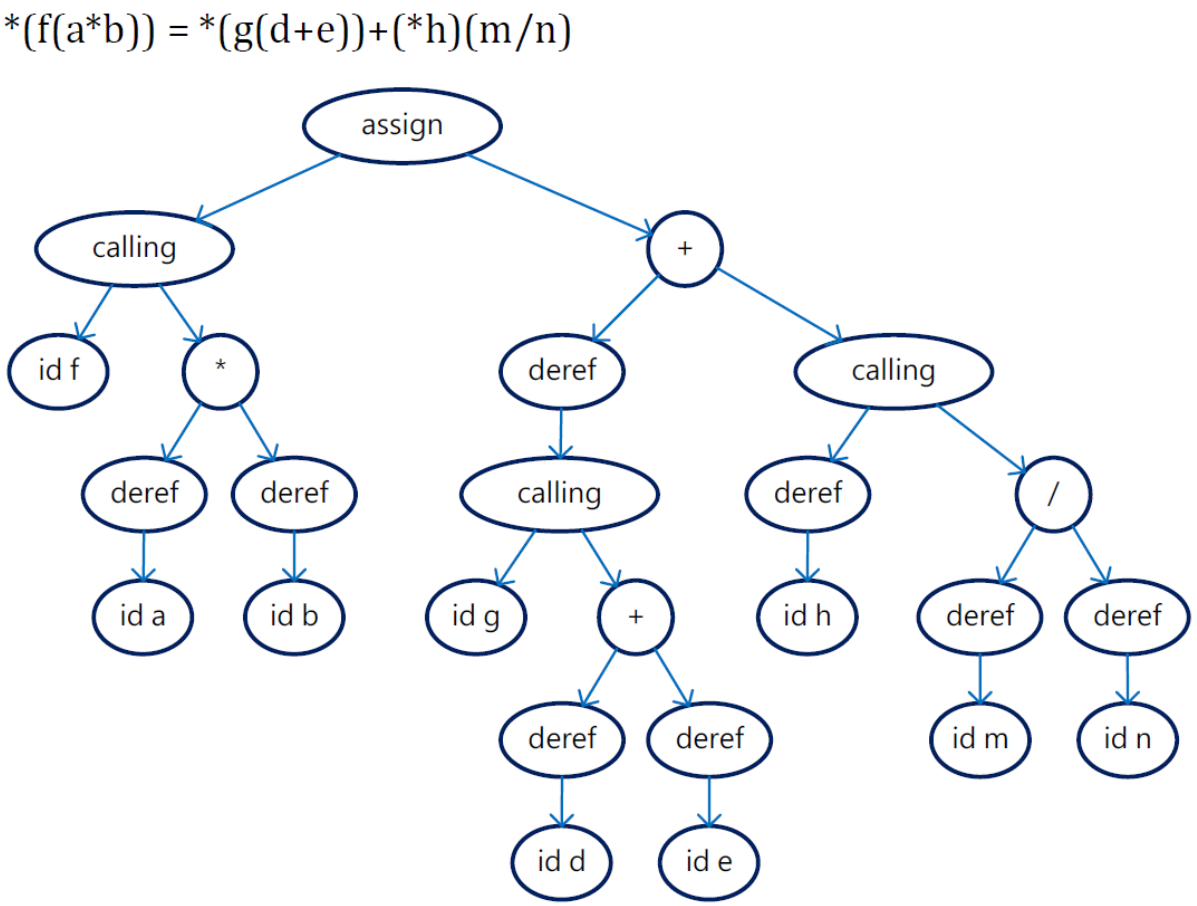


Figure 16: This is considered a correct AST though the grammar did not define expression and and * and + operators.

IGNORE THIS PAGE

Case Study: Webassembly

Text format

```
(module
  (func
    (i32.load (i32.const 0))
    (i32.load8_s (i32.const 0))
    (i32.load16_s (i32.const 0))
    (i32.load8_u (i32.const 0))
    (i32.load16_u (i32.const 0))
    (i64.load (i32.const 0))
    (i64.load8_s (i32.const 0))
    (i64.load16_s (i32.const 0))
    (i64.load32_s (i32.const 0))
    (i64.load8_u (i32.const 0))
    (i64.load16_u (i32.const 0))
    (i64.load32_u (i32.const 0))
    (f32.load (i32.const 0))
    (f64.load (i32.const 0))))
```

Binary format

```
0000000: 00                                       ; mem size log 2
0000001: 01                                       ; export mem
0000002: 0000                                     ; num globals
0000004: 0100                                     ; num funcs
0000006: 0000                                     ; num data segments
; function header 0
0000008: 00                                       ; func num args
0000009: 00                                       ; func result type
000000a: 0000 0000                                ; func name offset
000000e: 2000 0000                                ; func code start offset
0000012: 5800 0000                                ; func code end offset
0000016: 0000                                     ; num local i32
0000018: 0000                                     ; num local i64
000001a: 0000                                     ; num local f32
000001c: 0000                                     ; num local f64
000001e: 00                                       ; export func
000001f: 00                                       ; func external
; function data 0
0000020: 20                                       ; OPCODE_I32_LOAD_I32
0000021: 06                                       ; load access byte
0000022: 10                                       ; OPCODE_I8_CONST
0000023: 00                                       ; u8 literal
0000024: 20                                       ; OPCODE_I32_LOAD_I32
```

§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 in OO languages

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 examples 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, such as ifnode, whilenode, opnode, etc..

```
class AbstractNode ... end
class IfNode    extends AbstractNode ... end
class WhileNode extends AbstractNode ... end
class PlusNode  extends AbstractNode ... end
class MinusNode extends AbstractNode ... end
```

Compilers like GCC have around 200 phases. Common phases include semantic analysis, reachability analysis, type checking, type inference, code generation, etc.

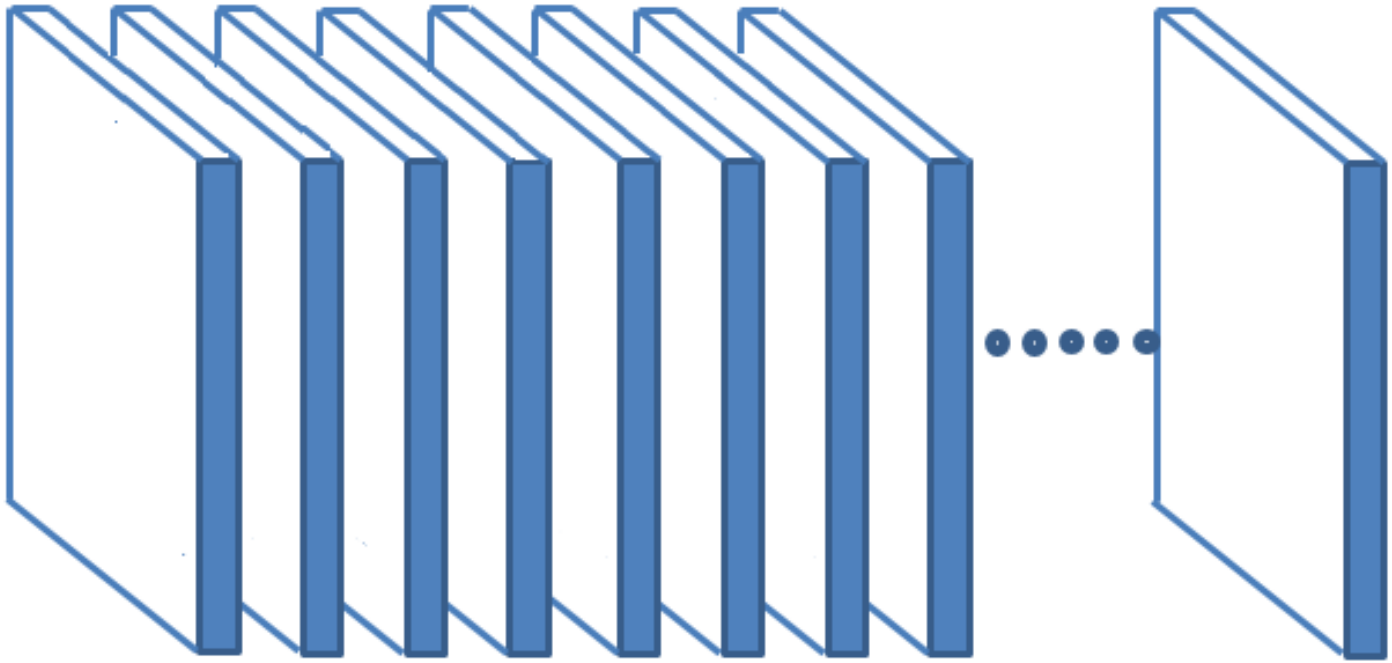


Figure 17: A compiler backend is made up of many phases. A new phase should not disturb existing phases.

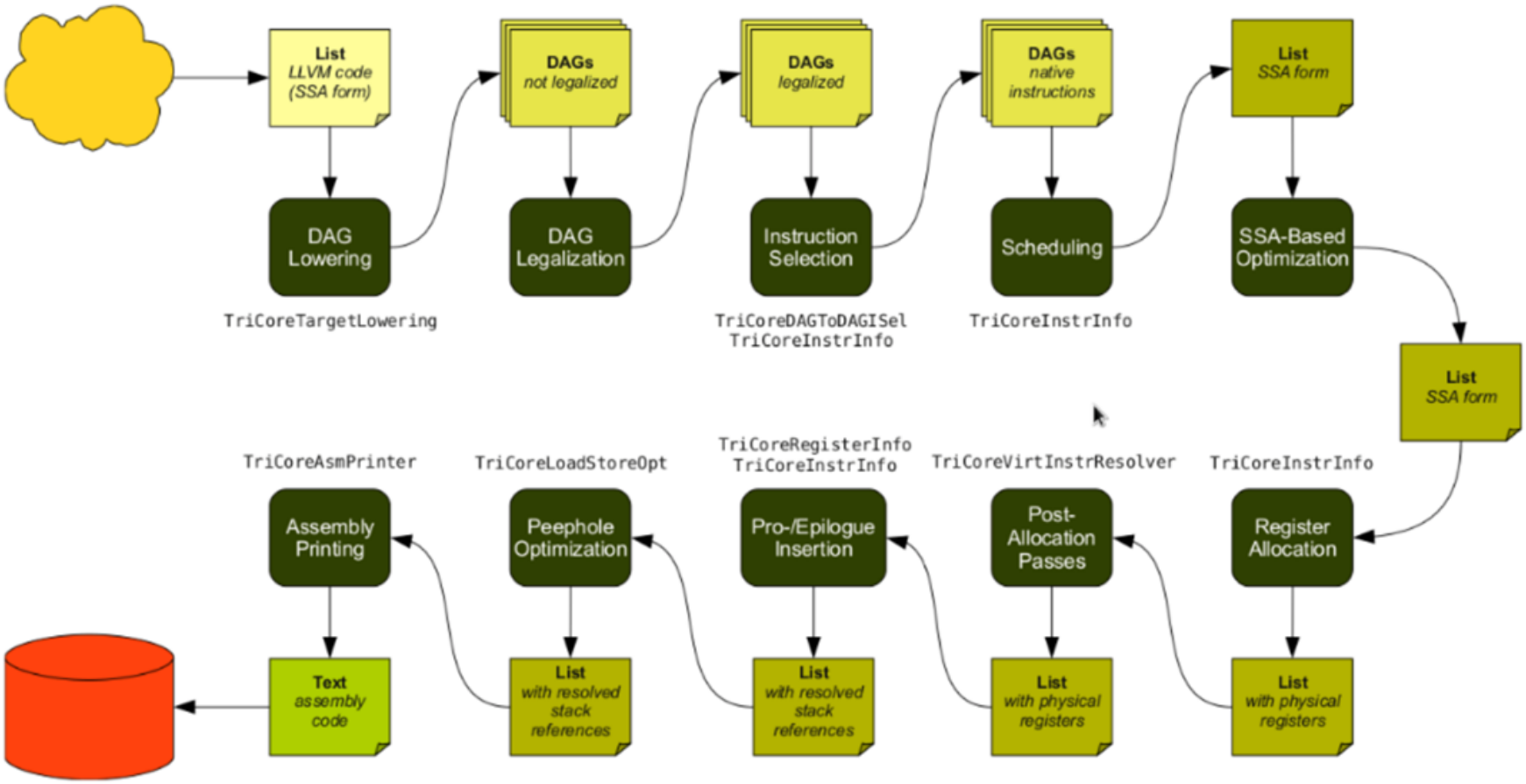


Figure 18: LLVM backend.

It is recommended that code for a single phase should be written in a single class, and not distributed among the various node classes. A phase is crafted by writing a *Visit* method in the phase's class.

```
class TypeChecking extends Visitor
   procedure Visit(IfNode i)    ... end
   procedure Visit(WhileNode w) ... end
   procedure Visit(PlusNode p)  ... end
   procedure Visit(MinusNode m) ... end
end
```

A Visitor f performs its work for a particular node type n as follows:

```
class Visitor    ... end
class TypeChecking extends Visitor    ... end

class AbstractNode    ... end
class IfNode extends AbstractNode    ... end

Visitor f;
f := new TypeChecking();
AbstractNode n;
n := new IfNode();
f.Visit(n);
```

The last call `f.Visit(n)` is intended to invoke the `Visit(IfNode)` method in the `TypeChecking` class. However, this is NOT the case in most OO languages. 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
```

```
class AbstractNode ... end
class IfNode    extends AbstractNode ... end
class PlusNode  extends AbstractNode ... end
class MinusNode extends AbstractNode ... end

Visitor f;
f := new TypeChecking();
AbstractNode n;
n := new IfNode();
f.Visit(n);  // 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 base class `Visitor` so that the `Visit(IfNode i)` method is invoked?

Common object-oriented languages use *single dispatch*. What we hope is *double dispatch*.

Though double dispatch is more intuitive, a technical difficulty in OO languages forces us to use single dispatch.

For double dispatch, we need to use single dispatch twice.

We will use the following code to achieve double dispatch:

Footnote. Suppose we have the following classes:

```
class A0 ... end;
class A1 extends A0 ... end;
class A2 extends A1 ... end;

class B0 ... end;
class B1 extends B0 ... end;
class B2 extends B1 ... end;

class C    foo(A0 x, B0 y) { print(111); }
           foo(A1 x, B2 y) { print(222); }
           foo(A2 x, B1 y) { print(333); }
end

A0 v1 = new A2;
B0 v2 = new B2;
C  v3 = new C0;
```

```
... v3.foo(v1, v2) ...
```

Consider the last call `v3.foo(v1, v2)`.

Which of the three `foo` functions in the `C` class should be invoked?

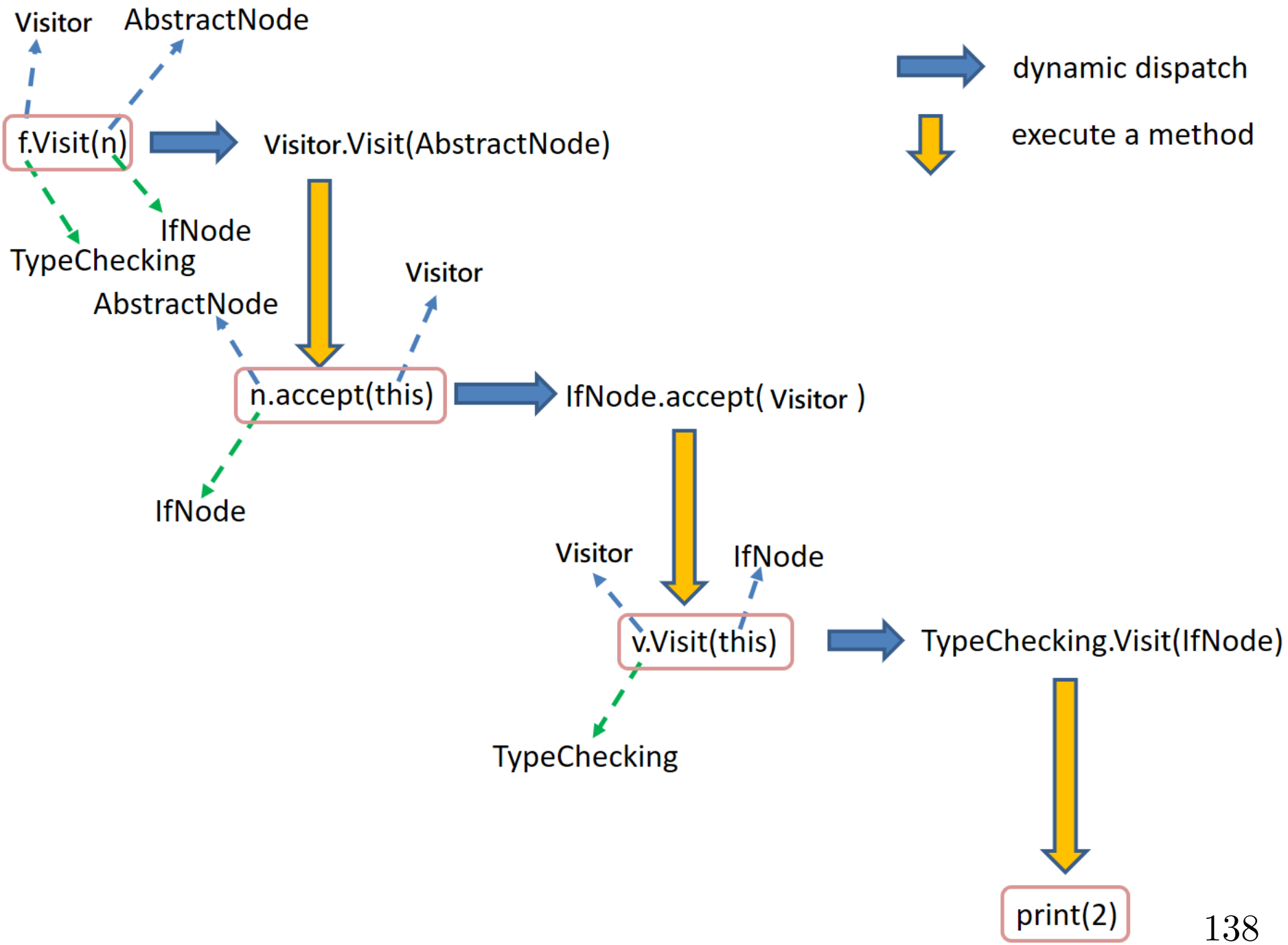
Answer: It is `foo(A0, B0)` that is invoked, which will print 111.

For this reason, double dispatch is NOT supported directly in the OO language. Double dispatch has to be achieved with two separate single dispatches.

end of footnote

```
class Visitor
   procedure Visit(AbstractNode n)
    { n.accept(this); } //accept(Visitor) in IfNode, etc.
   procedure Visit(IfNode n)    { /* empty method */ }
   procedure Visit(PlusNode p)  { /* empty method */ }
   procedure Visit(MinusNode m) { /* empty method */ }
end
class TypeChecking extends Visitor
   procedure Visit(IfNode i)     { print(2); }
   procedure Visit(PlusNode p)   { print(3); }
   procedure Visit(MinusNode m)  { print(4); }
end

class AbstractNode { accept(Visitor v) {/*empty method*/} }

class IfNode extends AbstractNode
   procedure accept(Visitor v) { v.Visit(this); }
     // Visit(IfNode) in TypeChecking
```

```
end

class PlusNode extends AbstractNode
   procedure accept(Visitor v) { v.Visit(this); }
end
class MinusNode extends AbstractNode
   procedure accept(Visitor v) { v.Visit(this); }
end

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


Visitor
AbstractNode
f.Visit(n)
Visitor.Visit(AbstractNode)
IfNode
TypeChecking
AbstractNode
Visitor
n.accept(this)
IfNode.accept(Visitor)
IfNode
Visitor
IfNode
v.Visit(this)
TypeChecking.Visit(IfNode)
TypeChecking
print(2)
dynamic dispatch
execute a method

What if we want to add a new Visitor (phase), say `CommonSubExpression`? Modify a Visitor (phase)?

Example

```
class CommonSubExpression extends Visitor
   procedure Visit(IfNode i)     { . . . }
   procedure Visit(PlusNode p)   { . . . }
   procedure Visit(MinusNode m)  { . . . }
end

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

§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` clss 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
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 is `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`.

```
    procedure      (AbstractNode n)
        this.          (n)
    end
    procedure          (Object o)
        /*   Find and invoke the      (n) method
        /*   whose declared parameter n is the closest matc
        /*   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)
```

Appendix.

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

		Production	Semantic rules
E	$\rightarrow$	$E_1 + T$	$E.ptr = MakeFamily(+, E_1.ptr, T.ptr)$
E	$\rightarrow$	T	$E.ptr = T.ptr$
T	$\rightarrow$	$T_1 * F$	$T.ptr = MakeFamily(*, T_1.ptr, F.ptr)$
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 in the scope
   - 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

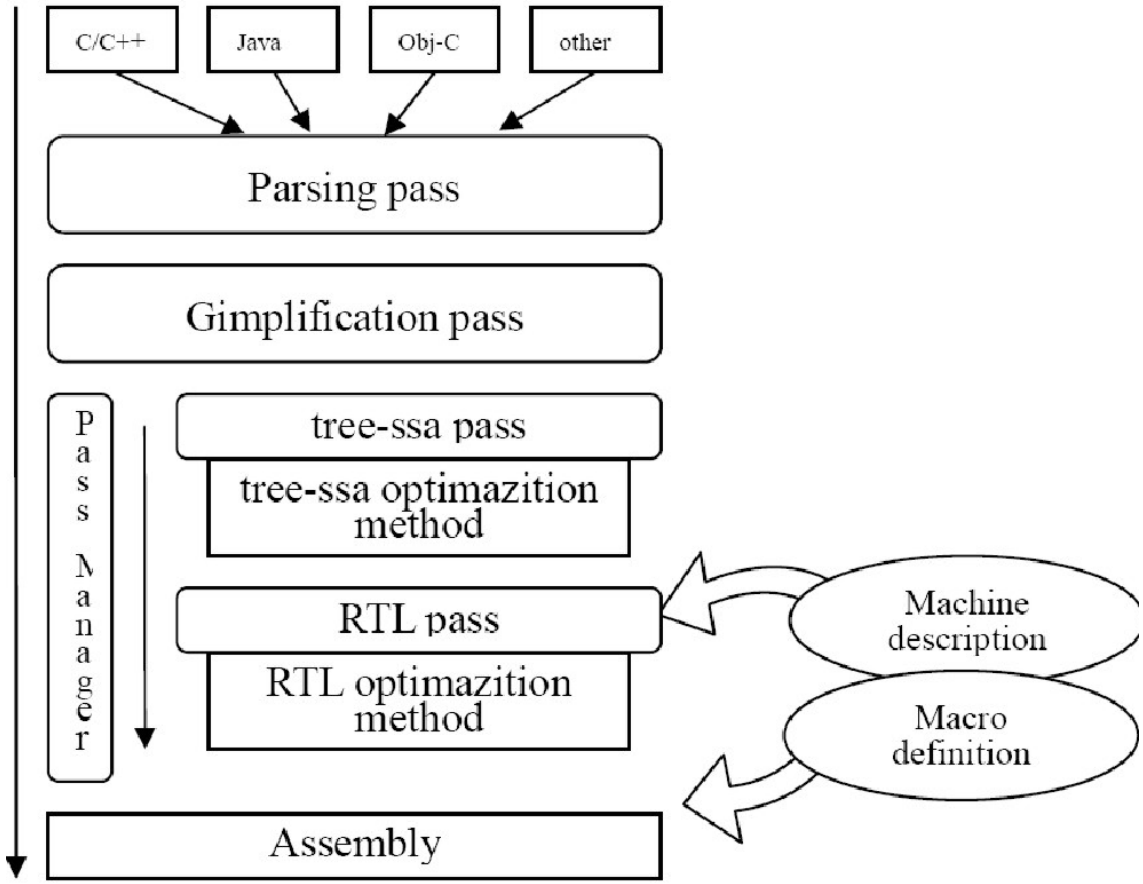


Figure 20: GCC Compilers

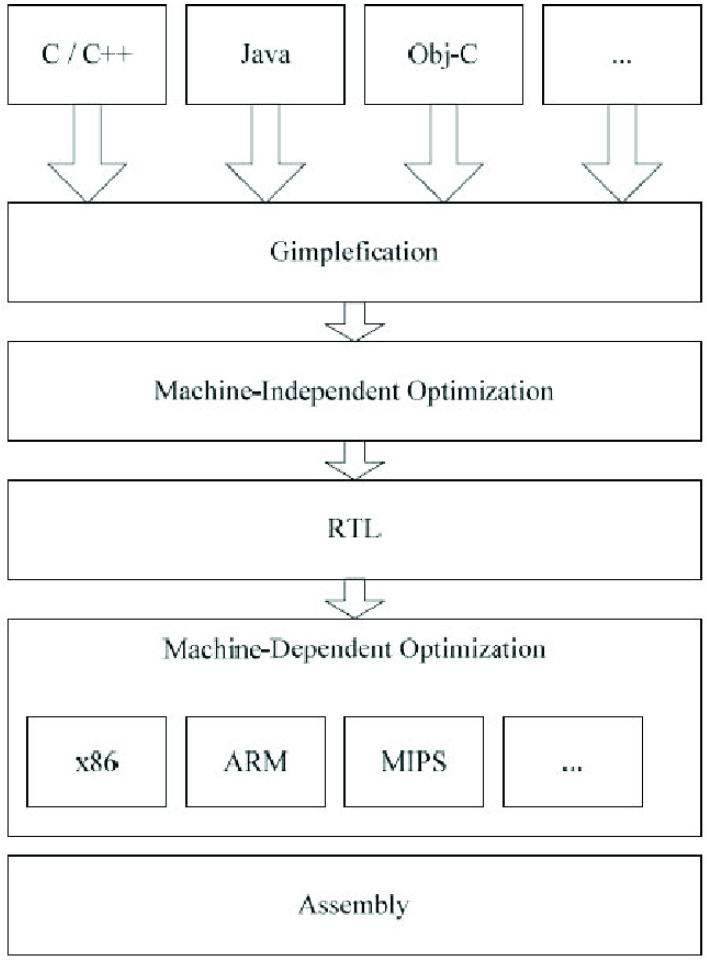


Figure 21: GCC Compilers (2)

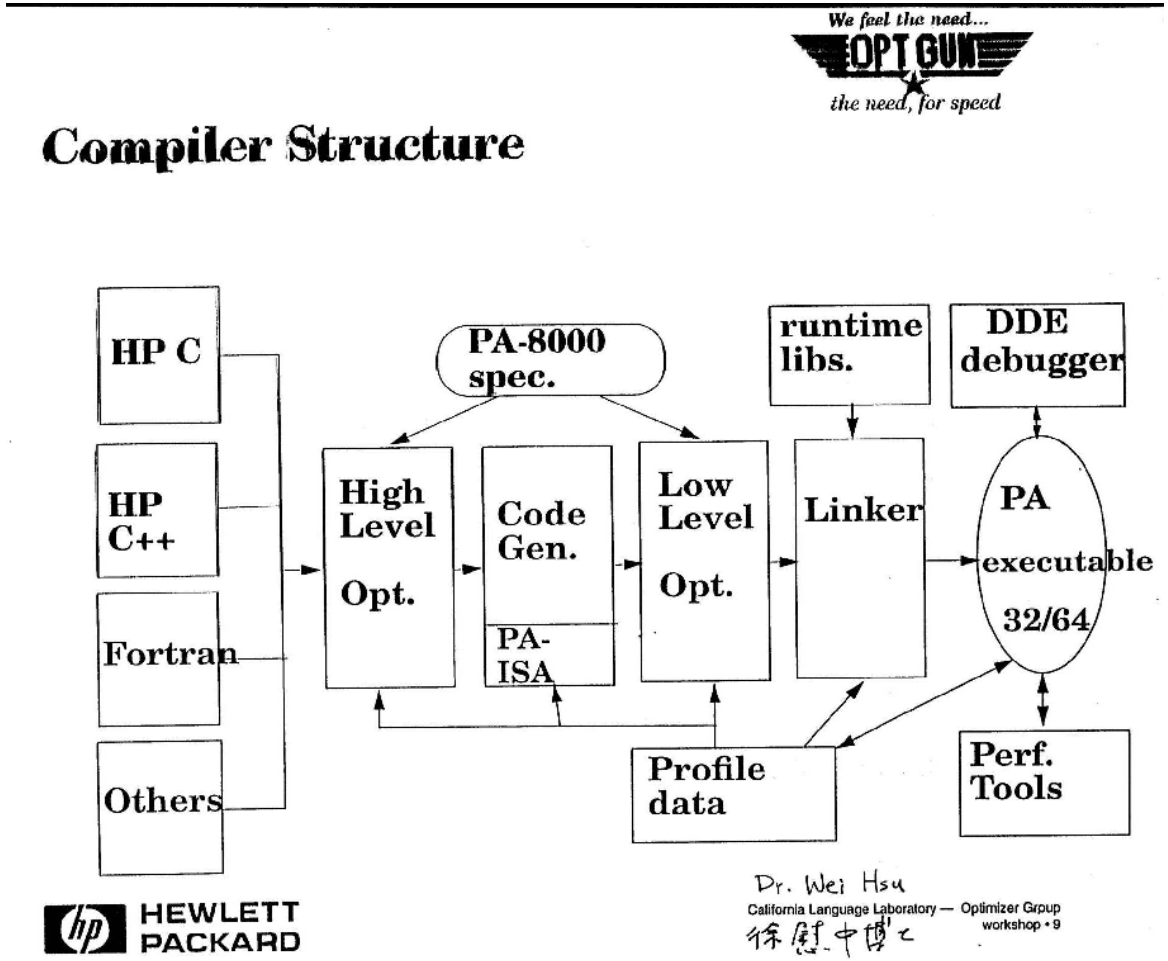


Figure 22: HP Compilers

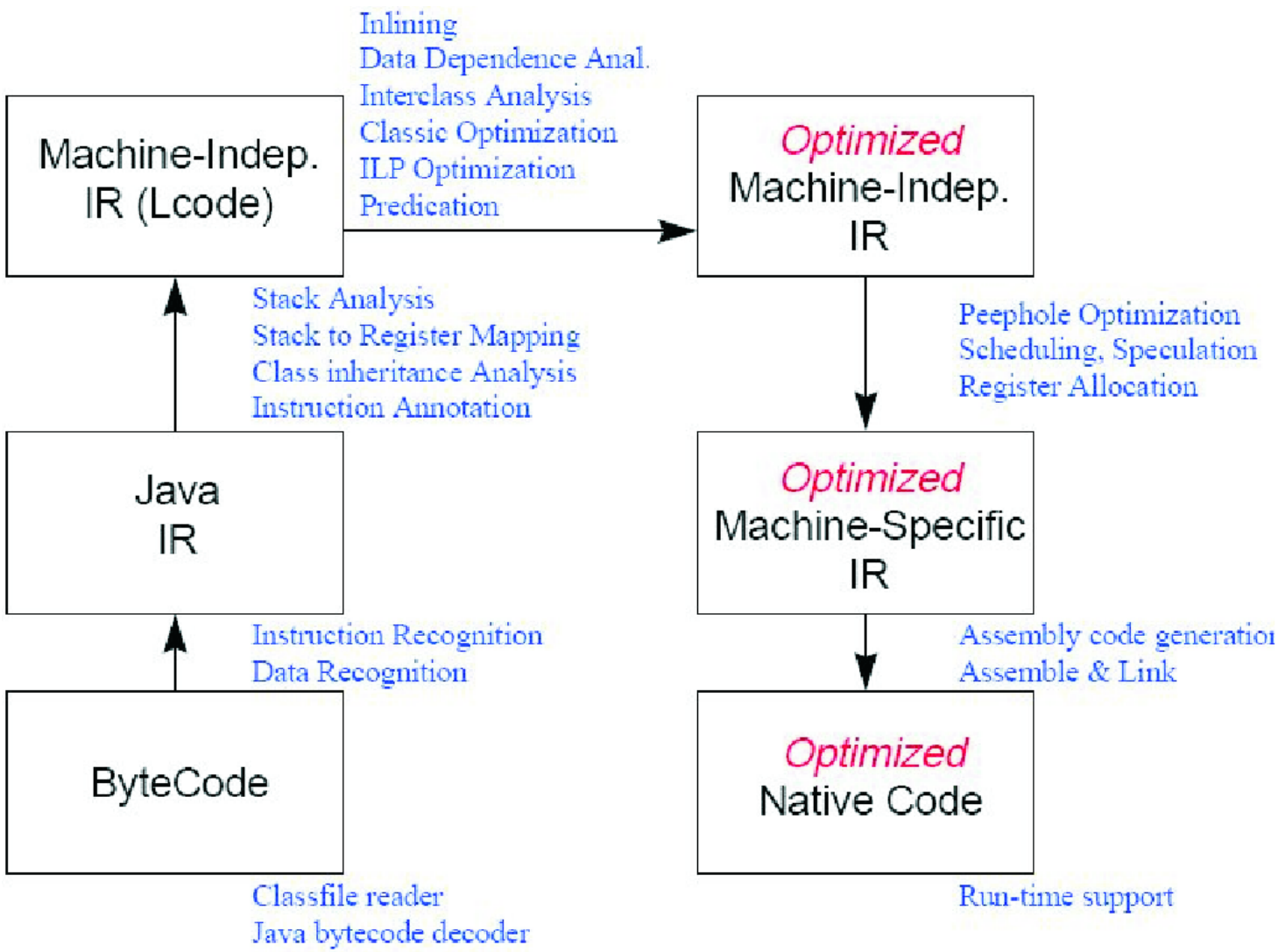


Figure 1. Java bytecode to native code translation steps.

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 mispredicated.

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.

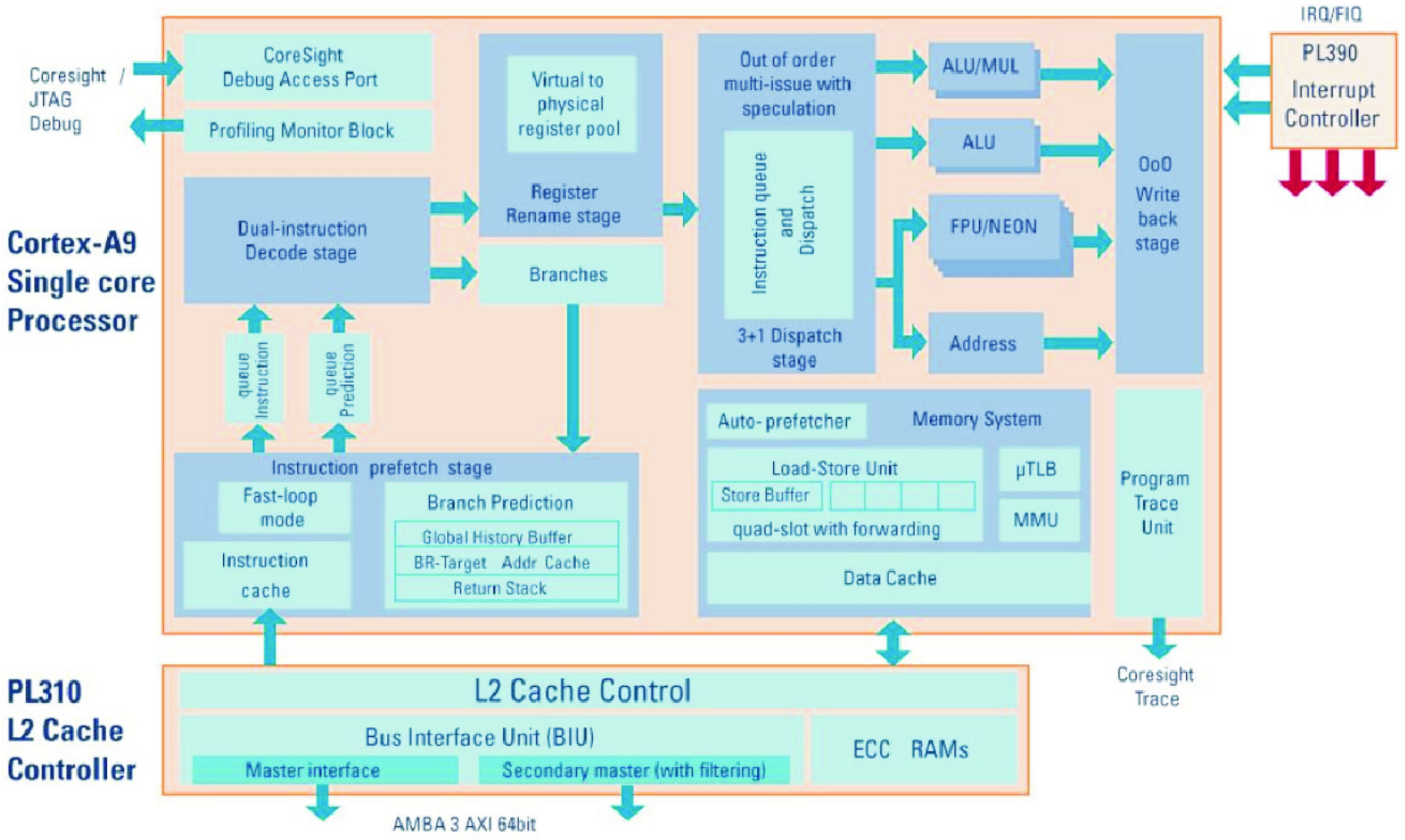


Figure 23: ARM Cortex A9 Architecture

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

```
%code requires {
     struct node {
        char * val;
        struct node *next;
     };
}
%union {
   int ival;
   char *name;
   double dval;
   struct node args;
}
```

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

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

An example. Interpret an expression.

```
%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. Build an abstract syntax tree in an OO language.

```
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; }
```

Show the upward propagation of attributes.

2nd Example. (LR(1) grammar) Build an abstract syntax tree for LR(1) grammar.

```
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; }
```

Show the upward propagation of attributes.

3rd Example. (LL(1) Grammar) Build an abstract syntax tree for an LR(1) grammar.

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

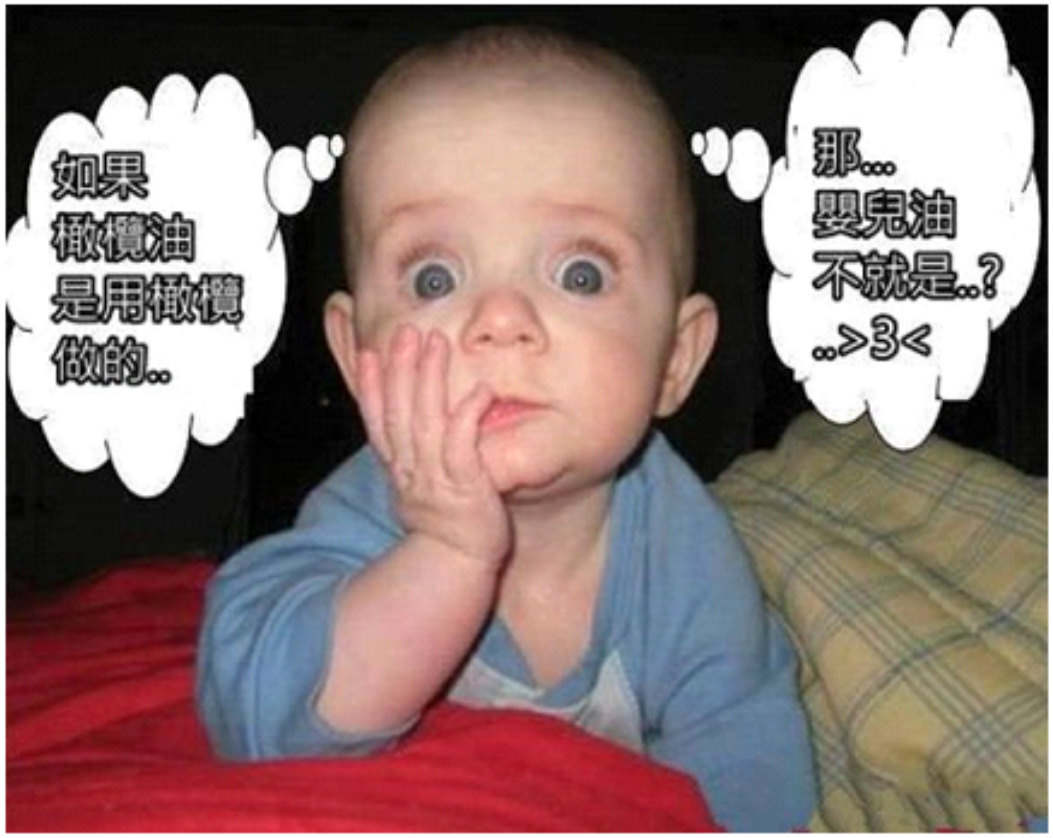
End of Chapter 7.

Syntax-directed translation: similar structures carry similar semantics.

train station vs workstation

鳳梨酥、*太陽餅*、魚刺肉羹

孩子，你想太多了!



appendix 1

(See lvalue-rvalue.c.) Try the following C++ code:

```
++ x = p + q;
x ++ = p + q;
pi = &++x;
pi = &x++;
```

Appendix. Example 1.

```
 public class foo {
  public static void main(String[] args) {
    Qvisitor v;            Bnode n;
    v = new Rvisitor();    n = new Cnode();
    System.out.println("1st try"); v.visit(n);
    System.out.println("2nd try"); n.accept(v);
  }
}

class Pvisitor {
  void visit(Anode aa) { System.out.println(0); }
}
class Qvisitor extends Pvisitor {
  void visit(Cnode cc)    { System.out.println(1); }
  // void visit(node nn)  { System.out.println(2); }
}
```

```
class Rvisitor extends Qvisitor {
  void visit(Anode aa)    { System.out.println(3); }
  void visit(Cnode cc)    { System.out.println(4); }
  // void visit(node nn)  { System.out.println(5); }
}

class Anode {
  void accept(Pvisitor pv) { pv.visit(this); }
}
class Bnode extends Anode {
  void accept(Pvisitor pv) { pv.visit(this); }
}
class Cnode extends Bnode {
  void accept(Pvisitor pv) { pv.visit(this); }
}
```

Answer. 1st try 3 2nd try 3

Example 2. (This is a good example.)

```
public class foo {
  public static void main(String[] args) {
    Qvisitor v;            Bnode n;
    v = new Rvisitor();    n = new Cnode();
    System.out.println("1st try"); v.visit(n);
    System.out.println("2nd try"); n.accept(v);
  }
}


class Pvisitor {
  void visit(Anode aa) { System.out.println(0); }
}
class Qvisitor extends Pvisitor {
  void visit(Cnode cc)    { System.out.println(1); }
  // void visit(node nn)  { System.out.println(2); }
```

```
}
class Rvisitor extends Qvisitor {
  void visit(Anode aa)    { System.out.println(3); }
  void visit(Cnode cc)    { System.out.println(4); }
  // void visit(node nn)  { System.out.println(5); }
}


class Anode {
  void accept(Qvisitor qv) { qv.visit(this); }
}
class Bnode extends Anode {
  void accept(Pvisitor pv) { pv.visit(this); }
}
class Cnode extends Bnode {
  void accept(Qvisitor qv) { qv.visit(this); }
}
```

Answer. 1st try 3 2nd try 4