BITS Pilani
Hyderabad Campus

Principles of Programming Languages(CS F301)

Prof.R.Gururaj
CS&IS Dept.

BITS Pilani
Hyderabad Campus

Description of Syntax and Semantics (Ch.3 of T1)

Prof R Gururaj

Introduction

The Formal description of Programming Languages first started with ALGOL.

But due to new notations used, it was not easily understandable.

For any description, the questions like–

For whom (target)? And Who are trying to understand it?

are the important.

For Programming languages descriptions are meant for three sections of audience.

1. Initial evaluators: (designers)
2. Implementers: (they need to understand how expressions, statements and program units are formed, and what is the effect of their execution.
3. Users of the language: who use the language to write programs to solve problems ; they use user manuals.

Syntax and semantics of a PL

Syntax of a programming language:

Is the form of its expressions, statements, and program units.

Semantics of a programming language:

Is the meaning of those expressions, statements, and program units.

Ex: *while (bool_expression)*

{statement} - is the *syntax*.

The *semantics* says that- when the current value of the Boolean expression is true, the embedded statement is executed.

Note:

Describing the syntax

Languages- Natural and Artificial.

What is language? – set of strings formed over an alphabet

What is a lexeme? - lower level of syntactic units.

What is a token? -Category of its lexemes.

Example

```
index = 2 * count + 17;
```

The lexemes and tokens of this statement are

Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

Language Recognizers

A language is formally defined in two ways:

- Language recognizer
- Language generator

Language *L*; Alphabet is $\sum$

The language Recognizer ***R*** is a device which tells a string/sentence is in *L* or not.

It is not meant for enumerating strings of that Language.

The syntax analysis step in compilation has a recognizer for the language it is translating.

Hence a syntax analyzer determines whether the given programs are syntactically correct or not.

Language Generators

A language generator is a device that can be used to generate a sentence of a language.

We use language generator mechanism called "Grammar" that are commonly used to describe the syntax of a programming language.

BNF (Backus-Naur Form)

In the middle of 1950s by

Noam Chomsky & John Backus developed the similar syntax description formalism that became popular approach to describe languages.

Chomsky in 1950 proposed four categories of Grammars.

1. Regular Grammars
2. Context Free Grammars
3. Context Sensitive Grammars
4. Recursively Enumerable Grammars

Popularly used to describe languages

BNF

In 1958 John Backus proposed formalism for describing languages.

Peter Naur modified it. It became BNF used first for describing ALGOL.

BNF is *metalanguage*.

BNF uses abstract syntax structures.

Uses Rules or Productions.

Ex:

<assign> → <var> = <expr>

Abstractions are called *non-terminals*.

BNF

Abstractions are called *non-terminals*.

Lexemes and tokens are called *terminals*.

A grammar is a collection of rules.

Grammar Derivations.

A Grammar is a generative device for defining languages.

Sentences of a language are generator through a sequence of applications of these rules beginning with a special NT called as Start symbol.

For programming languages, usually the start symbol is <program>

Example

A Grammar for a Small Language

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> - <var>
| <var>

Sample derivation

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

A Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Derivation:

Sentential Form:

Sentence:

LMD:

RMD:

LMD Ex:

A = B * (A + C)

is generated by the leftmost derivation:

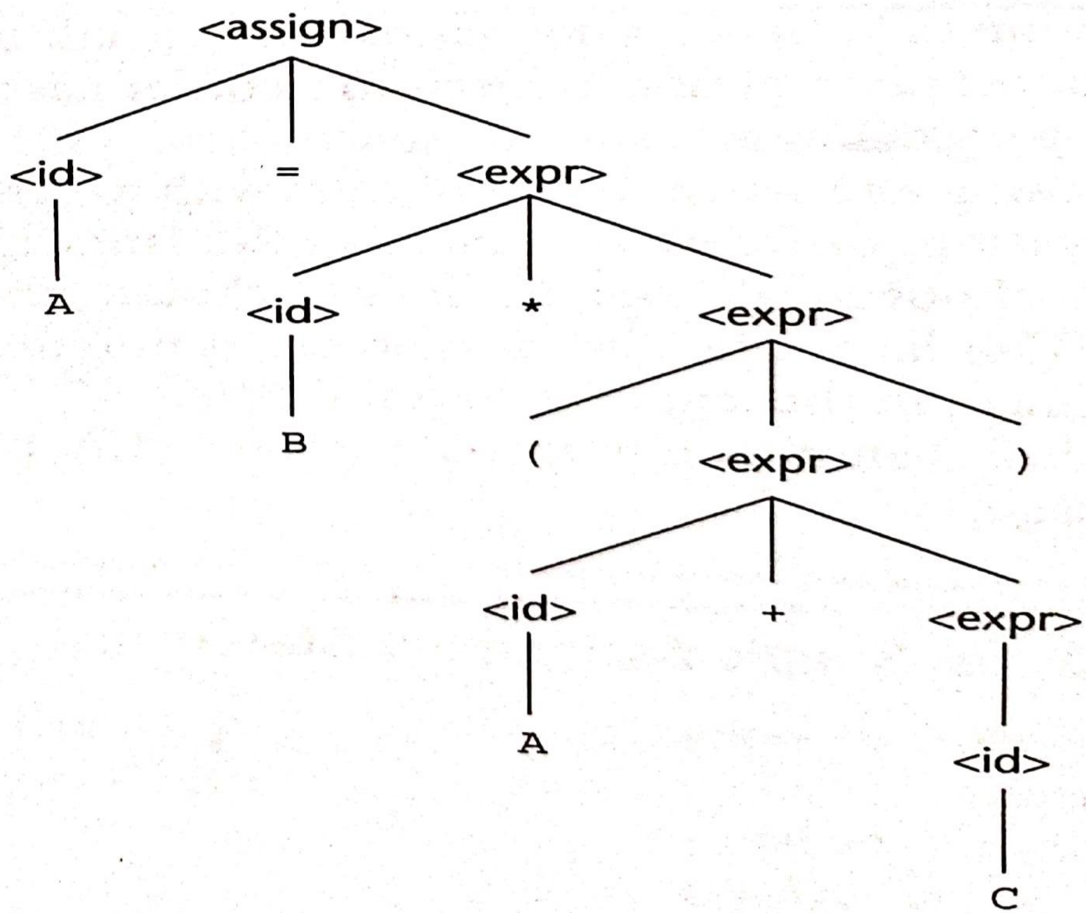
<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

Parse Tree:

The hierarchical structure of the sentences described by the grammars is known as Parse Tree.

Figure 1

A parse tree for the simple statement
A = B * (A + C)



Ambiguity:

A grammar that generates a sentential form for which there exist two or more distinct parse trees is known as ‘Ambiguous Grammar’.

Otherwise if there exist two or more LMDs or two or more RMDs for sentential form, then the grammar is called as ambiguous grammar.

An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>

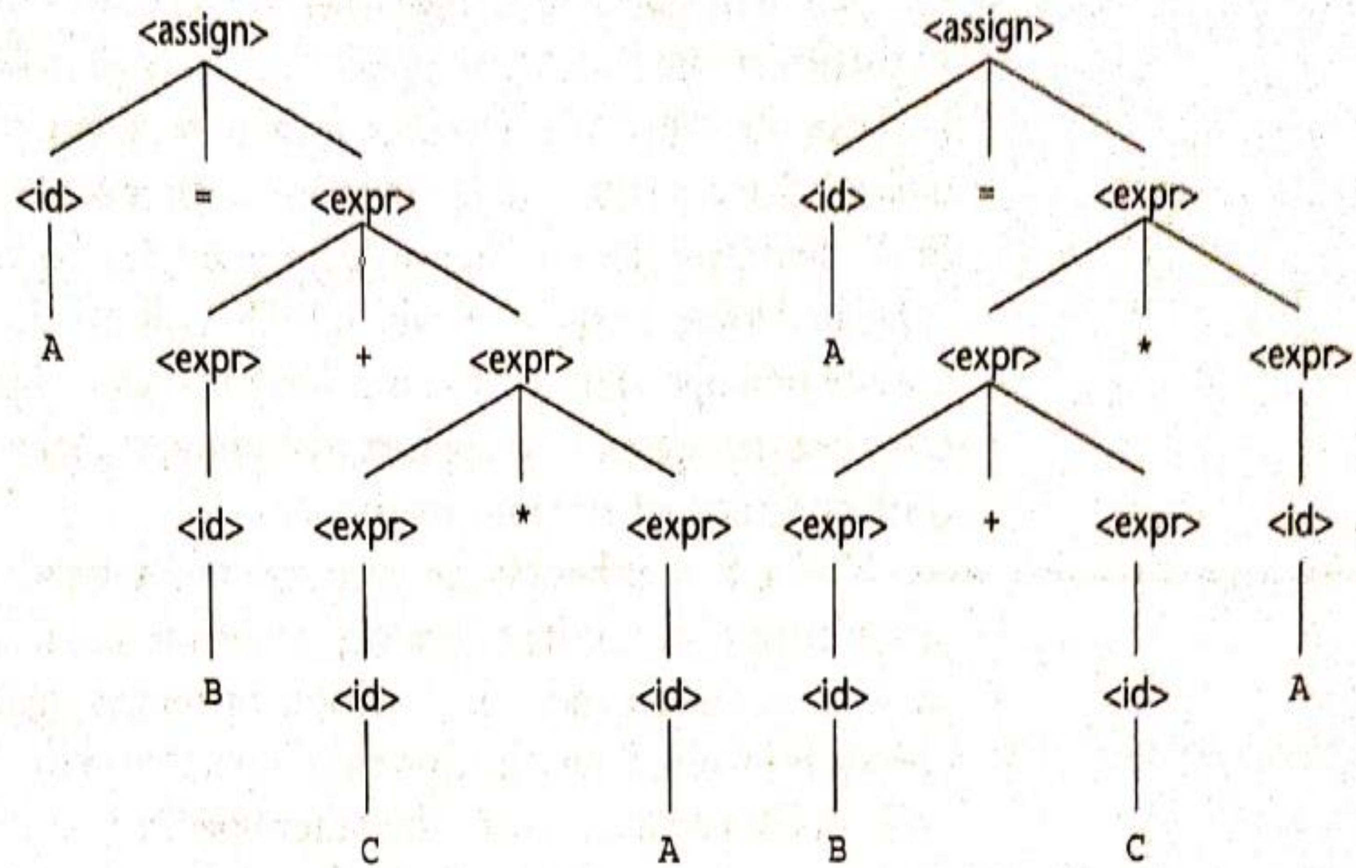
| <expr> * <expr>

| (<expr>)

| <id>

Figure 2

Two distinct parse trees for the same sentence, A = B + C * A



Left Recursive grammar

E→ E+T

Right Recursive grammar

E→T+E

Extended BNF:

- Optional part of RHS

```
<if_stmt> → if (<expression>) <statement> [else <statement>]
```

- Repetitions

```
<ident_list> → <identifier> {, <identifier>}
```

- Alternation

```
<term> → <term> (* | / | %) <factor>
```

In BNF, a description of this <term> would require the following three rules:

```
<term> → <term> * <factor>
       | <term> / <factor>
       | <term> % <factor>
```

Grammars and Recognizers

Lex

Yacc

Attribute Grammars.

An attribute grammar is an extension of CFG.

Thee extensions allow certain rules to be these extensions follow certain language rules to be conveniently described such as type compatibility.

Static semantics

Type compatibility

Declaration of variable before it is referenced.

The static semantics of a language are only indirectly related to the meaning of programs during the execution, rather they have to do with the legal format of the program. (syntax rather than semantics)

Static because these checks can be performed at compile time itself.

BNF can’t describe static semantics.

Attribute Grammar is a mechanism devised to describe and check the correctness of static semantic rules of a program.

Basic Concepts:

1. Attribute grammars are CFGs , which have been added with: (i) attributes

   (ii) attribute computation functions

   (iii) predicate functions.

(i) Attributes: are associated with the grammar symbols (both terminals and non terminals) , and are similar o variables with assigned values.

(ii) Attribute computation functions (Semantic functions): are associated with grammar rules and are used to specify how the attribute values are computed.

(iii) Predicate functions: state the static semantic rules associated with the grammar rules.

Other features of Attributes Grammars:

1. Attributes associate with Grammar Symbols.

$A(X) = S(X) \cup I(X)$

Synthesized attributes- computed based values of children

Inherited attributes- based on parents and siblings

2. Semantic functions

3. Predicate functions

Intrinsic attributes: are synthesized attributes of leaf nodes whose values are determined outside the parse tree.

Syntax rule: <proc_def> → **procedure** <proc_name> [1]

<proc_body> **end** <proc_name> [2] ;

Predicate: <proc_name> [1] string == <proc_name> [2] .string

In this example, the predicate rule states that the name string attribute of the <proc_name> nonterminal in the subprogram header must match the name string attribute of the <proc_name> nonterminal following the end of the subprogram.

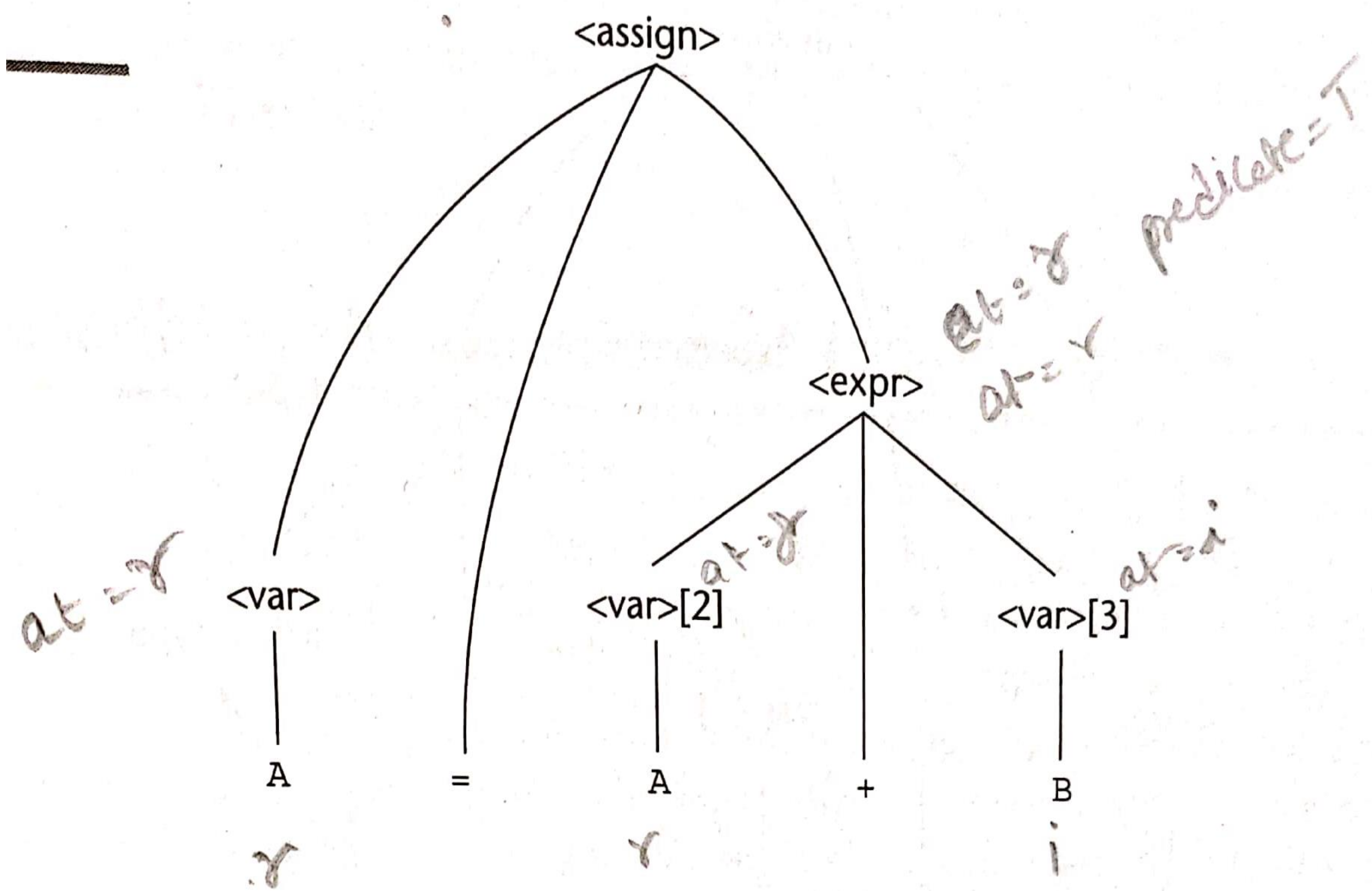
The syntax portion of our example attribute grammar is

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.


<assign>
<expr>
<var>
<var>[2]
<var>[3]
A
=
A
+
B
at = r
at = r
at = r
predicate = T
at = r
at = i
r
r
i

Computing the Attribute values

Computing the Attribute values of parse tree: Which is sometimes known as *decorating the parse* tree.

Top-down approach (if all attributes are inherited)

Bottom-up approach (if all attributes are synthesized)

Mixed approach (if a attributes are of both types)

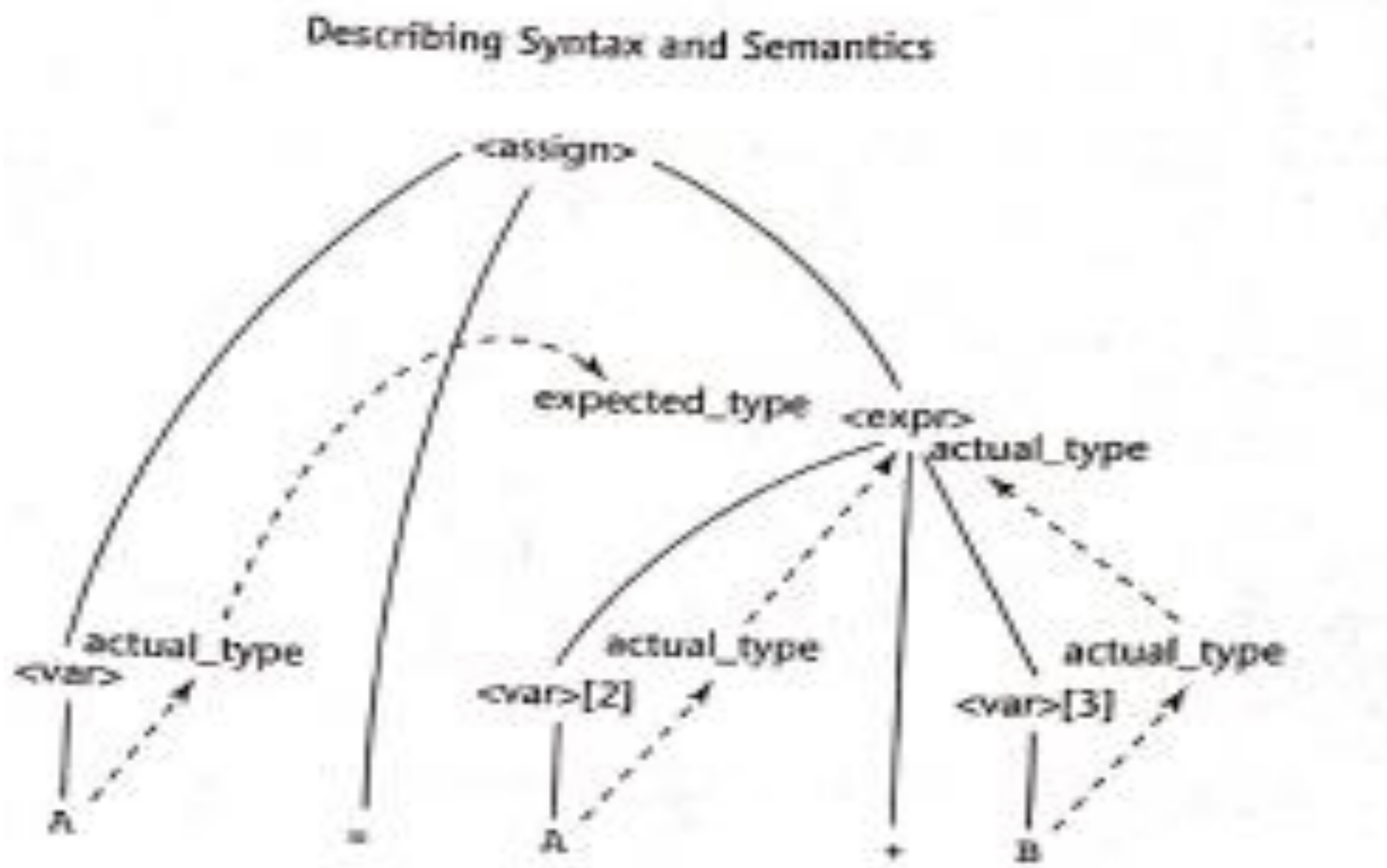
If all attributes have been computed, the tree is called *fully attributed* tree.

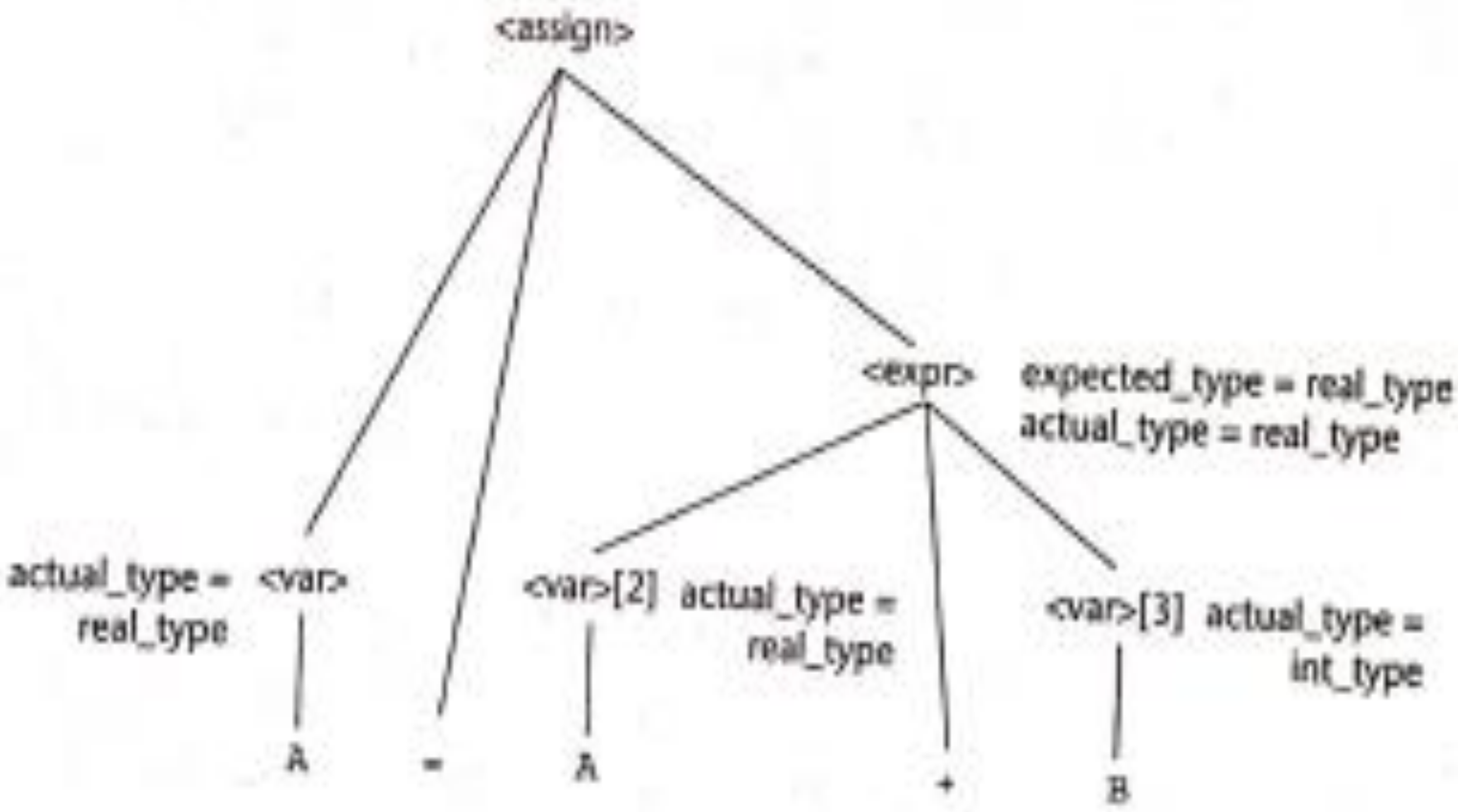
1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)

   <var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either

   TRUE or FALSE (Rule 2)

Flow of attribute values




<assign>
<expr>
expected_type = real_type
actual_type = real_type
actual_type = real_type
<var>
<var>[2]
actual_type = real_type
<var>[3]
actual_type = int_type
A
=
A
+
B

Describing the meaning of Programs. (Dynamic semantics)

Now we take up the task of describing the dynamic semantics of a (constructs) programming language.

It is easy to describe syntax than semantics.

No universally accepted notation of describing the semantics.

Need for describing the dynamic semantics(meaning):

1. Compiler designers
2. SW Developers

Operational semantics

The idea behind Operational semantics is to describe the meaning of statements or program by specifying the effects of running it on the machine.

The effects on the machine are viewed as the sequence of changes in its state.

The machine's state is the collection of the values in its storage.

Usually intermediate languages are used to describing the operational semantics.

C Statement	*Meaning*
`for (expr1 ; expr2 ; expr3) {`	`expr1 ;`
`...`	`loop: if expr2 == 0 goto out`
`}`	`...`
	`expr3 ;`
	`goto loop`
	`out: ...`

Denotational semantics

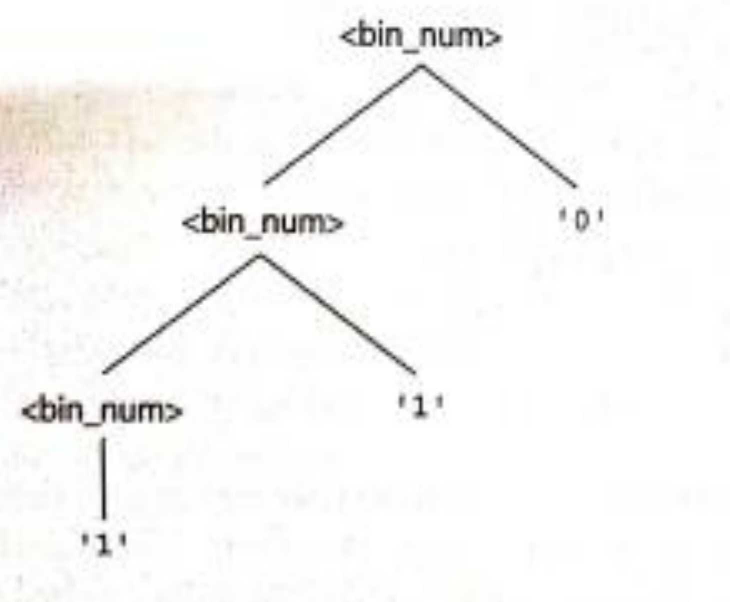
Is the most rigorous and most widely known formal method for describing the meaning of Programs.

It is solidly based on recursive function theory.

In operational semantics-Programming constructs are translated to simpler PL constructs.

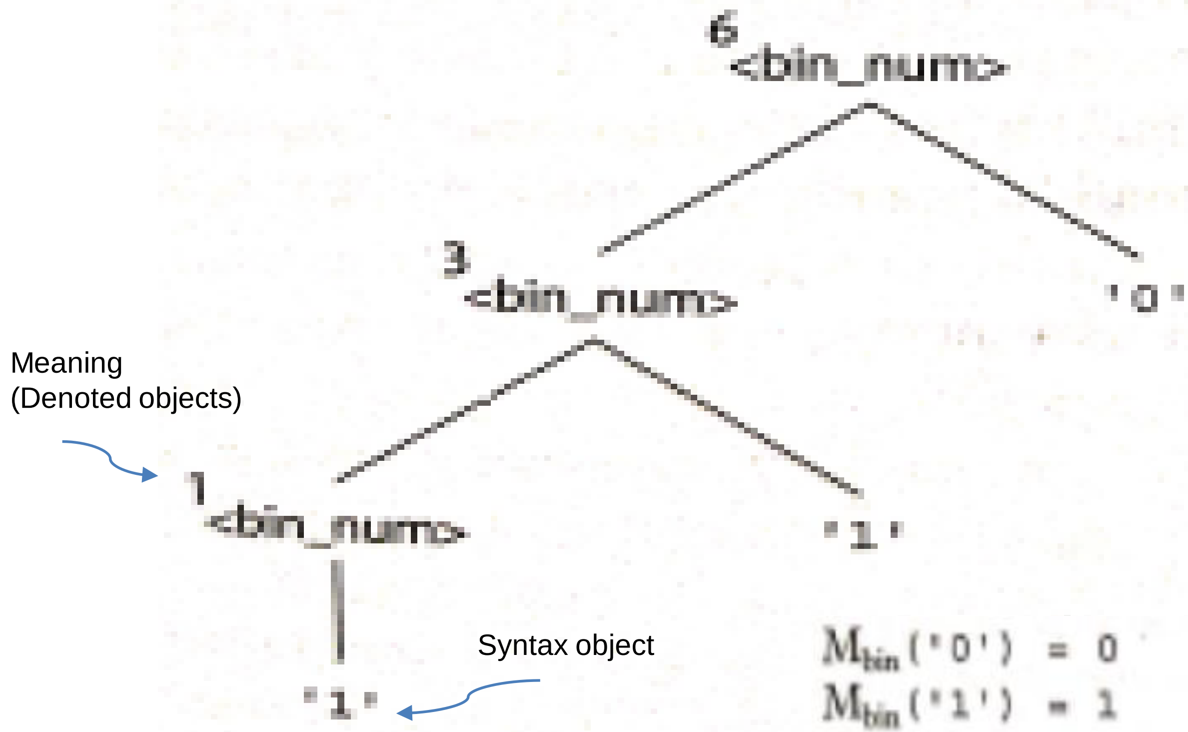
In Denotational semantics- Language constructs (entities) are mapped to mathematical objects (sets/functions). But it does not model step-by-step computational processing of programs.

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'



The semantic function, named M_{bin}, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers. The function M_{bin} is defined as follows:

$$M_{bin}(\text{'0'}) = 0$$
$$M_{bin}(\text{'1'}) = 1$$
$$M_{bin}(\text{<bin\_num> '0'}) = 2 * M_{bin}(\text{<bin\_num>})$$
$$M_{bin}(\text{<bin\_num> '1'}) = 2 * M_{bin}(\text{<bin\_num>}) + 1$$



$M_{bin}(\text{'0'}) = 0$

$M_{bin}(\text{'1'}) = 1$

$M_{bin}(\langle\text{bin\_num}\rangle\ \text{'0'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle)$

$M_{bin}(\langle\text{bin\_num}\rangle\ \text{'1'}) = 2 * M_{bin}(\langle\text{bin\_num}\rangle) + 1$

We assume that the syntax and static syntax are correct.

Axiomatic Semantics

An Axiom is a logical statement which is assumed to be true.

Axiomatic Semantics is based on mathematical logic.

Rather directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program.

The meaning of the statements is based on the relationships among program variables and constants which are same for every execution of the program.

Axiomatic Semantics has two applications:

1. Program verification
2. Program semantic specification

We use notions:

Assertions

Precondition

Post-condition

Weakest precondition

Using Precondition and post-condition

In axiomatic semantics the meaning of a specific statement is defined by its precondition and post-condition.

In effect the two assertions specify precisely the effect of executing the statement.

Assignment statements

Ex.1

$a = b / 2 - 1 \{a < 10\}$

$b / 2 - 1 < 10$

$b < 22$

Weekest precondition of the statement.

Ex.2

$x = x + y - 3 \{x > 10\}$

the weakest precondition is

$x + y - 3 > 10$

$y > 13 - x$

Sequences

```
y = 3 * x + 1;
x = y + 3;
{x < 10}
```

The precondition for the second assignment statement is

```
y < 7
```

which is used as the postcondition for the first statement. The precondition for the first assignment statement can now be computed:

```
3 * x + 1 < 7
x < 2
```

So, {x < 2} is the precondition of both the first statement and the two-statement sequence.

Selection

```
if x > 0 then
  y = y - 1
else
  y = y + 1
```

Suppose the postcondition, Q, for this selection statement is $\{y > 0\}$. We can use the axiom for assignment on the **then** clause

$$y = y - 1 \; \{y > 0\}$$

This produces $\{y - 1 > 0\}$ or $\{y > 1\}$. It can be used as the P part of the precondition for the **then** clause. Now we apply the same axiom to the **else** clause

$$y = y + 1 \; \{y > 0\}$$

Loops

loop.

```
while y <> x do y = y + 1 end {y = x}
```

{y = x}

For one iteration, it is

wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}

For two iterations, it is

wp(y = y + 1, {y = x - 1})={y + 1 = x - 1}, or {y = x - 2}

For three iterations, it is

wp(y = y + 1, {y = x - 2})={y + 1 = x - 2}, or {y = x - 3}

Program Proofs

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

```
{x = B AND t = A}
```

Next, we use this new precondition as a postcondition on the middle statement and compute its precondition, which is

```
{y = B AND t = A}
```

Next, we use this new assertion as the postcondition on the first statement and apply the assignment axiom, which yields

```
{y = B AND x = A}
```

Same as precondition of the program. Hence correct.

Summary

- What is Syntax and Semantics?
- Describing Syntax.
- Grammar, Derivation, Parse tree and ambiguity.
- Specifying the semantics:
- Static semantics: Attribute Grammar
- Dynamic Semantics- Operational semantics, Denotational semantics, Axiomatic semantics