Software II: Principles of Programming Languages

Lecture 3 – Formal Descriptions of a Programming Language

Lexics vs. Syntax Vs. Semantics

- Lexics refers to issues regarding the assembly of words that comprise a statement.
- Syntax refers to issues regarding the grammar of a statement.
- Semantics refers to issues regarding the meaning of a statement.

Lexical Structure of Programming Languages

- It was believed in the early days of programming language development that it was sufficient to be able specify the syntax of a programming language. We now know that this is not enough.
- This led to the development of context-free grammars and Backus-Naur Form.

Programming Language Syntax

- Syntax is defined as "***the arrangement of words as elements in a sentence to show their relationship***."
- Syntax provides a great deal of information that we need to understand a program and to guide its translation.
- Example

  $2 + 3 \times 4 = 14$ (*not 20 – multiplication takes precedence*)

Programming Language Semantics

- Semantics is defined as “***the meaning of a symbol or set of symbols***.”
- This is equally important in translating a programming correctly and may be more difficult to express unambiguously.

Tokens

- The lexical structure of program consists of sequence of characters that are assembled into character strings called ***lexemes*** which have directly related to ***tokens***, the element of a languages grammar to which they correspond.
- Tokens fall into several distinct categories:
  - reserved words
  - literals or constants
  - special symbols such as **`< =  +`**
  - identifiers, such as **`x24, average, balance`**

Reserved Words and Standard Identifiers

- Reserved words serve a special purpose within the syntax of a language; for this reason, they are generally not allowed to be used as user-defined identifiers.
- Reserved words are sometimes confused with standard identifiers, which are identifiers defined by the language, but serve no special syntactic purpose.
- The standard data types are standard identifiers in Pascal and Ada.

Free- and Fixed-Field Formats

- Fixed-field format is a holdover from the day of punch cards
  - A fixed field syntax uses the positioning within a line to convey information.
  - E g., FORTRAN, COBOL and RPG use fixed-field formats.
  - SNOBOL4 uses the first character on the line to distinguish between statement labels, continuations and comments
- Free-field formats allow program statements to be written anywhere on a line without regard to position on the line or to line breaks.

Delimiting Lexemes

- Most languages work with lexemes of differing length; this could create problems.
  - If the input is **`doif`** is the lexeme **`doif`** or are there two lexemes **`do`** and **`if`**?
  - The easiest way to handle this is to use the principle of longest substring, i.e., the longest possible string is the lexeme.
- As a result, we typically use white space as a delimiter separating lexemes in a source file.

Scanning FORTRAN

- FORTRAN breaks many of the rules of lexical analysis
- FORTRAN ignores white space, which leads to:

```
DO 99 I = 1, 10
vs
DO 99 I = 1.10
```

- FORTRAN allows keywords to be defined as variables:

```
IF = 2
IF (IF.LT.0)  IF = IF + 1
ELSE IF = IF + 2
```

Regular Expressions

- The lexemes of a programming languages are described formally by the use of regular expressions, where there are 3 operations, concatentation, repetition and selection:
  - a|b denotes a ***or*** b.
  - ab denotes a ***followed by*** b
  - (ab)* denotes a followed by b ***zero or more times***
  - (a|b)c denotes a or b followed by c

Extending Regular Expressions

- There are other operators that we can add to regular expression notations that make them easier to write:
  - **`[a-z]`** any character from **`a`** through z
  - **`r+`** one or more occurrences of **`r`**
  - **`?`** An optional term
  - **`.`** Any one character
- Examples
  - **`[0-9]+`** describes an integer
  - **`[0-9]+(\.[0-9]+)?`** describes an unsigned real

What Is A Grammar?

The grammar of a language is expressed formally as

G = (T, N, S, P) where

T is a set of ***terminals*** (the basic, atomic symbols of a language).

N is a set of ***nonterminals*** (symbols which denote particular arrangements of terminals).

S is the ***start symbol*** (a special nonterminal which denotes the program as a whole).

P is the set of ***productions*** (rules showing how terminals and nonterminal can be arranged to form other nonterminals.

An Example Of A Grammar?

- We can describe the manner in which sentences in English are composed:
  1. *sentence* $\rightarrow$ *noun-phrase verb-phrase* .
  2. *noun-phrase* $\rightarrow$ *article noun*
  3. *article* $\rightarrow$ **a** | **the**
  4. *noun* $\rightarrow$ **girl** | **dog**
  5. *verb-phrase* $\rightarrow$ *verb noun-phrase*
  6. *verb* $\rightarrow$ **sees** | **pets**

Start symbol

Non-terminals

Terminals

Parsing A Sentence

- Let's examine the sentence "***the girl sees a dog.***

sentence ⇒ *noun-phrase verb-phrase* . **(Rule 1)**

sentence ⇒ *article noun verb-phrase* . **(Rule 2)**

sentence ⇒ **the** *noun verb-phrase* . **(Rule 3)**

sentence ⇒ **the girl** *verb-phrase* . **(Rule 4)**

sentence ⇒ **the girl** *verb noun-phrase* . **(Rule 5)**

sentence ⇒ **the girl sees** *noun-phrase* . **(Rule 6)**

sentence ⇒ **the girl sees** *article noun* . **(Rule 2)**

sentence ⇒ **the girl sees a** *noun* . **(Rule 3)**

sentence ⇒ **the girl sees a dog** . **(Rule 3)**

Context –Free Grammars and BNFs

- Context-Free grammars are grammars where non-terminals (collections of tokens in a language) always are deconstructed the same way, ***regardless of the context*** in which they are used.
- BNF (Backus-Naur form) is the standard notation or ***metalanguage*** used to specify the grammar of the language.

Backus-Naur Form

BNF (***B***ackus-***N***aur ***F***orm) is a metalanguage for describing a context-free grammar.

- The symbol ::= (or ?) is used for ***may derive***.
- The symbol | separates alternative strings on the right-hand side.

<u>Example</u> E ::= E + T | T

T ::= T * F | F

F ::= id | constant | (E)

where E is ***Expression***, T is ***Term***, and F is ***Factor***

Syntax

- We can use BNF to specify the syntax of a programming language, and determine if we have a ***syntactically correct program***.
- Syntactic correctness does not mean that a program is semantically correct. We could write:

  `The home/ ran/ girl`

  and recognize that this is nonsensical even if the grammar is correct.
- A language is a set of finite-length strings with characters chosen from the language's alphabet.
  - This includes the set of all programs written in <*fill in you favorite programming language*> .

Grammar For Simple Assignment Statements

<*assignment statement*> ::= <*variable*> **=** <*arithmetic expression*>

<*arithmetic expression*> ::=<*term*> | <*arithmetic expression*> **+** <*term*> | <*arithmetic expression*> **-** <*term*>

<*term*> ::= <*primary*> | <*term*> ´ <*primary*> | <*term*> **/** <*primary*>

<*primary*> ::= <*variable*> | <*number*> | **(**<*arithmetic expression*>**)**

<*variable*> **::=** <*identifier*> **|** <*identifier*> **[**<*subscript list*>**]**

<*subscript list*> **::=** <*arithmetic expression*> | <*subscript list*>**,** <*arithmetic expression*>

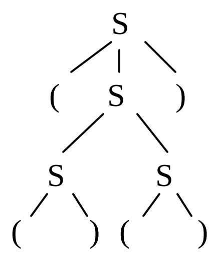
Generating Strings

- To generate strings that belong to our language, we use a single-replacement rule: the generation of strings in our language all begin with a single symbol which we replace with the right-hand side of a production:
- S → SS | (S) | ( )
- We can generate the string: (( ) ( ))

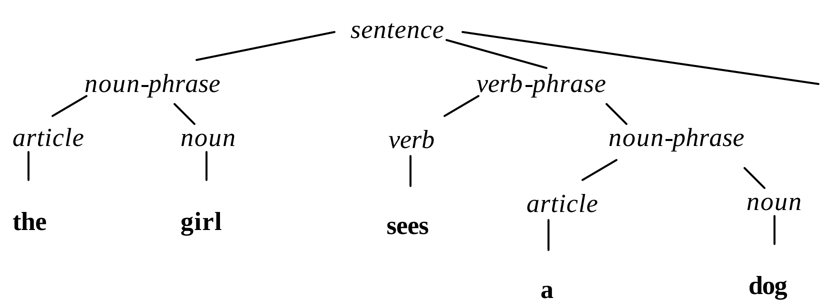
S ⇒(S) ⇒ (S S) ⇒ (( ) S) ⇒ (( ) ( ))

sentential forms

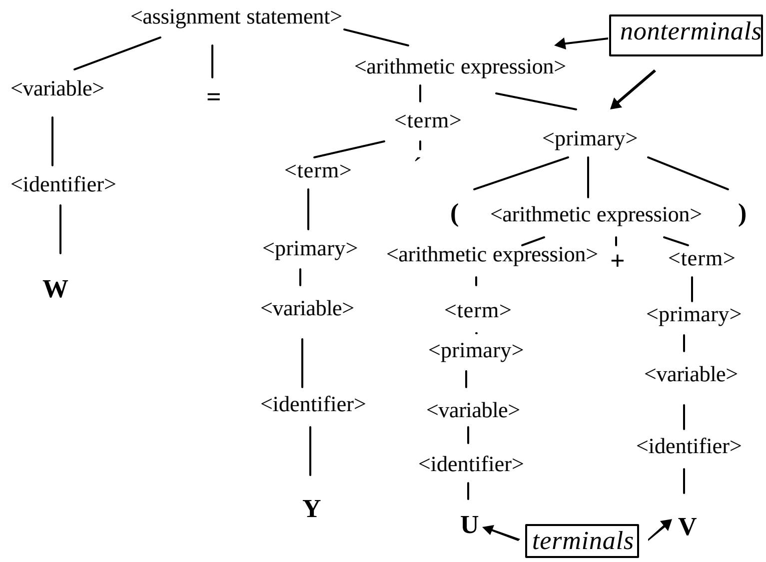
Parse Tree For `(()())`



Parse Tree for "`The girl sees a dog`"



Parse Tree for an Assignment Statement



Using BNF To Specify Regular Expressions

- We can also use BNF to specify how we assemble the words that comprise our language:

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
  | 9
<unsigned integer> ::= <digit> | <unsigned
  integer> <digit>
```

- These strings are much simple that the ones that comprise programs and are called ***regular expressions***.

Example - Another Expression Grammar

- Let's take a look at another simple expression grammar:

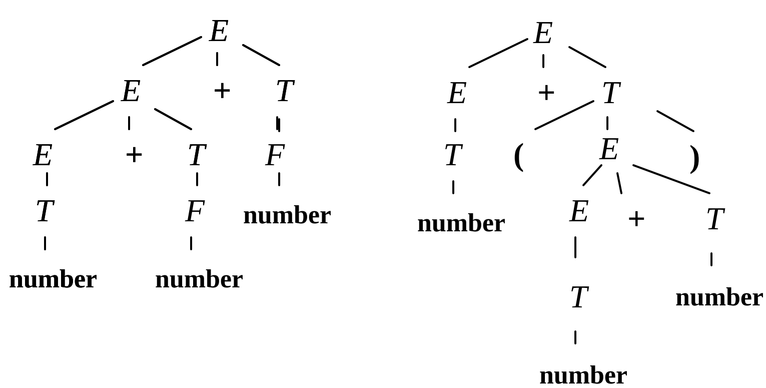
  $E \rightarrow E$ **+** $T \mid T$

  $T \rightarrow T$ ***** $F \mid T$

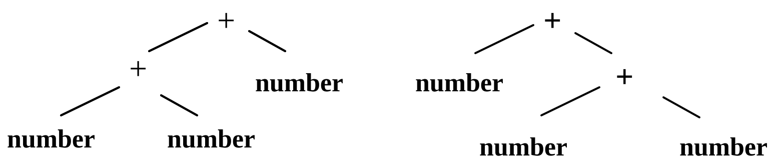
  $F \rightarrow$ **(**E**)** | **number**

- Let's parse the expressions **`3 + 4 + 5`** and **`3 + (4 + 5)`**

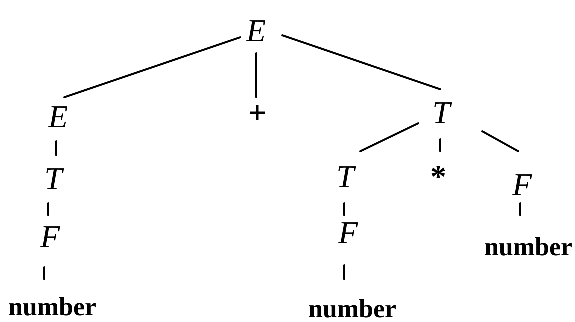
Expression Parse Trees



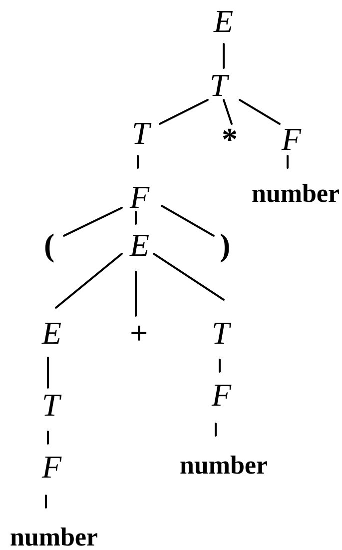
Abstract Syntax Trees for Expression



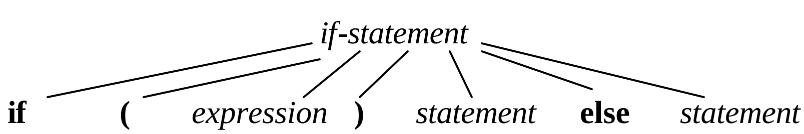
Parsing 3 + 4 * 5



Parsing (3 + 4) * 5

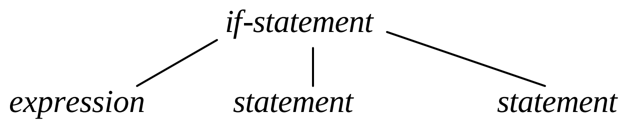


Parse Tree For If-Then-Else



if-statement ® **if (** *expression* **)** *statement* **else** *statement*

Abstract Syntax Tree For If-Then-Else



if-statement ® **if (** *expression* **)** *statement* **else** *statement*

Extended Backus-Naur Form

EBNF (***E***xtended ***B***ackus-***N***aur ***F***orm) adds a few additional metasymbols whose main advantage is replacing recursion with iteration.

- {***a***} means that ***a*** is occur zero or more times.
- [***a***] means that a appears once or not at all.

Example Our expression grammar can become:

E ::= T { + T }

T ::= F { * F }

F ::= id | constant | (E)

Left and right derivations

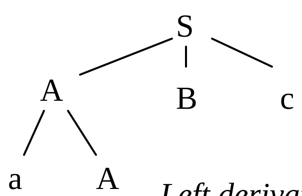
Remember our grammar:

S ::= A B c

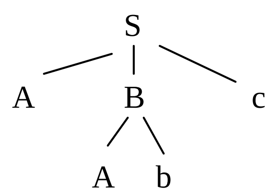
A ::= a A | b

B ::= A b | a

How do we parse the string ***abbbc***?



Left derivation



Right derivation

Ambiguity

- Any grammar that accurately describes the language is equally valid.
- Sometimes, there may be more than one way to parse a program correctly. If this is the case, the grammar is said to be ambiguous.
- **`They /are flying / planes.`**
  `They are/ flying planes.`
- Ambiguity (which is NOT desirable) is usually a property of the grammar and not of the language itself.

Ambiguous grammars

- While there may be an infinite number of grammars that describe a given language, their parse trees may be very different.
- A grammar capable of producing two different parse trees for the same sentence is called ***ambiguous***. Ambiguous grammars are highly undesireable.

Is it IF-THEN or IF-THEN-ELSE?

The IF-THEN=ELSE ambiguity is a classical example of an ambiguous grammar.

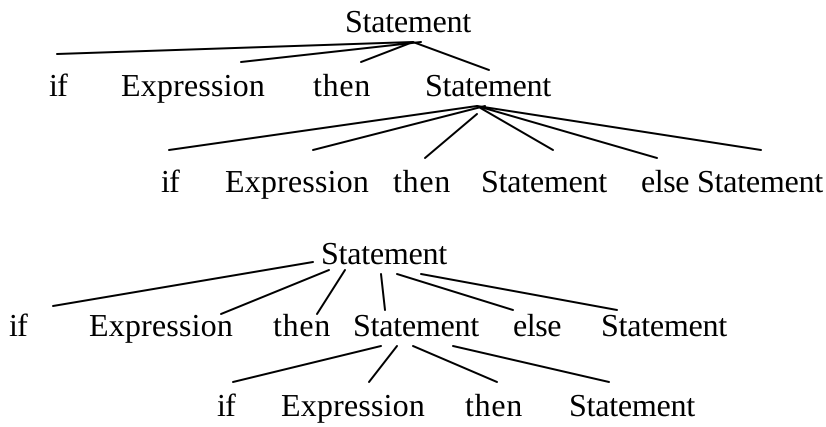
Statement ::= **if** *Expression* **then** *Statement* **else** *Statement*
| **if** *Expression* **then** *Statement*

How would you parse the following string?

```
IF x > 0
  THEN IF y > 0
         THEN z := x + y
         ELSE z := x;
```

Is it IF-THEN or IF-THEN-ELSE? (continued)

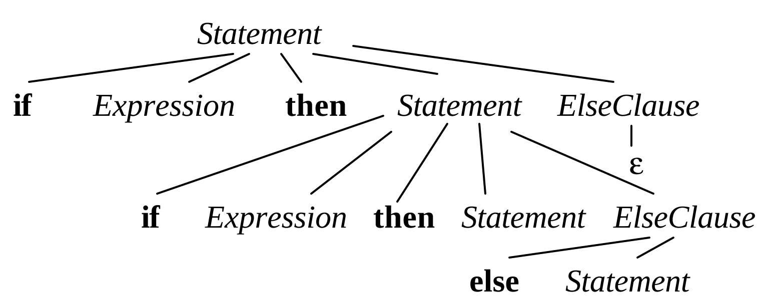
There are two possible parse trees:



Is it IF-THEN or IF-THEN-ELSE? (continued)

Statement ::= **if** *Expression* **then** *Statement ElseClause*

ElseClause ::= **else** *Statement* | ε



Ambiguous Languages

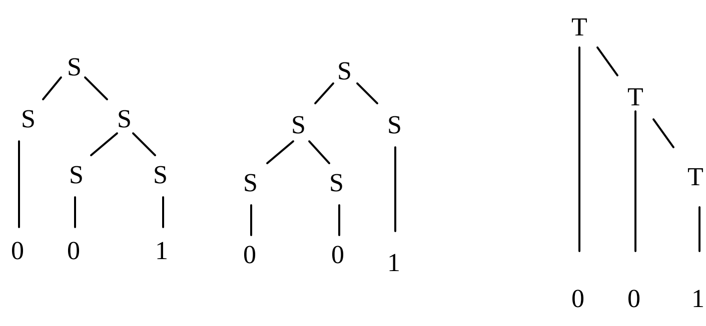
- If every grammar in a language is ambiguous, we say that the language is inherently ambiguous.
- If we have two grammars:

  G_1: $S \rightarrow SS \mid 0 \mid 1$

  G_2: $T \rightarrow 0T \mid 1T \mid 0 \mid 1$

  G_1 is ambiguous; G_2 is not; therefore the language is ***NOT*** inherently ambiguous

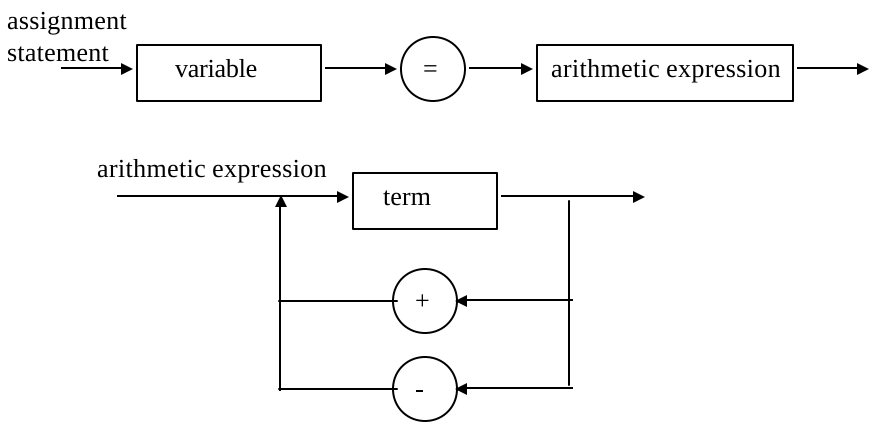
Ambiguity in Grammars



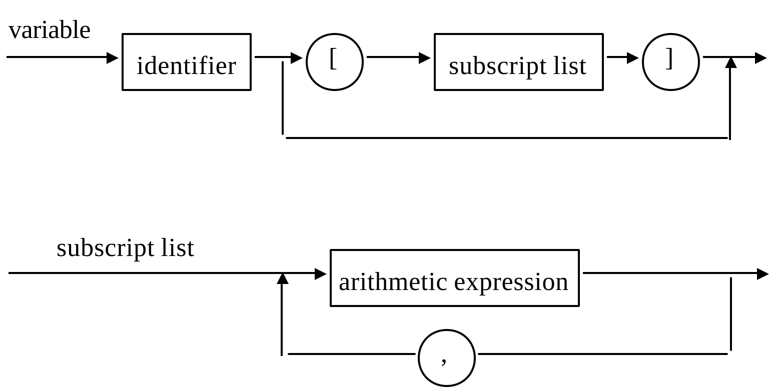
G1 : Ambiguous Grammar

G2: Unambiguous Grammar

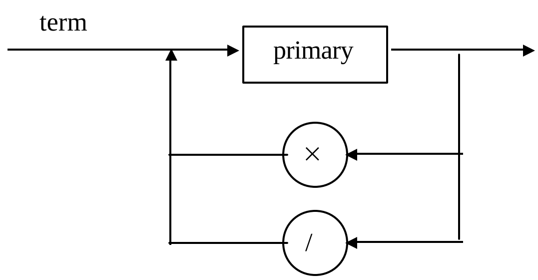
Syntax Charts for Simple Assignment Statements



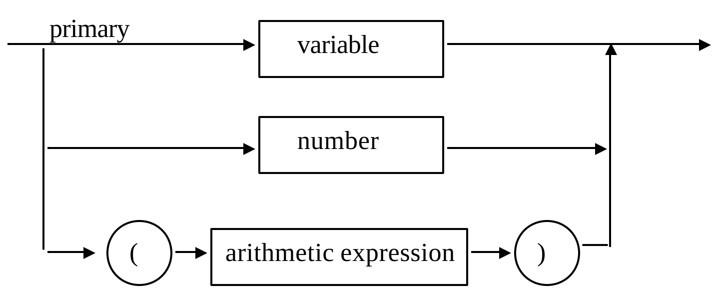
Syntax Charts for Simple Assignment Statements (continued)



Syntax Charts for Simple Assignment Statements (continued)



Syntax Charts for Simple Assignment Statements (continued)



What is an Attribute Grammar?

- An attribute grammar is an extension to a context-free grammar that is used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty.
- Examples
  - Describing the rule that float variables can be assigned integer values but the reverse is not true is difficult to describe complete in BNF.
  - The rule requiring that all variable must be declared before being used is impossible to describe in BNF.

Static vs. Dynamic Semantics

- The static semantics of a language is indirectly related to the meaning of programs during execution. Its names comes from the fact that these specifications can be checked at compile time.
- Dynamic semantics refers to meaning of expressions, statements and other program units. Unlike static semantics, these cannot be checked at runtime and can only be checked at runtime.

What is an Attribute?

- An ***attribute*** is a property whose value is assigned to a grammar symbol.
- ***Attribute computation functions*** (or semantic functions) are associated with the productions of a grammar and are used to compute the values of an attribute.
- ***Predicate functions*** state some of the syntax and static semantics rules of the grammar.

Definition of an Attribute Grammar

An attribute grammar is a grammar with the following added features:

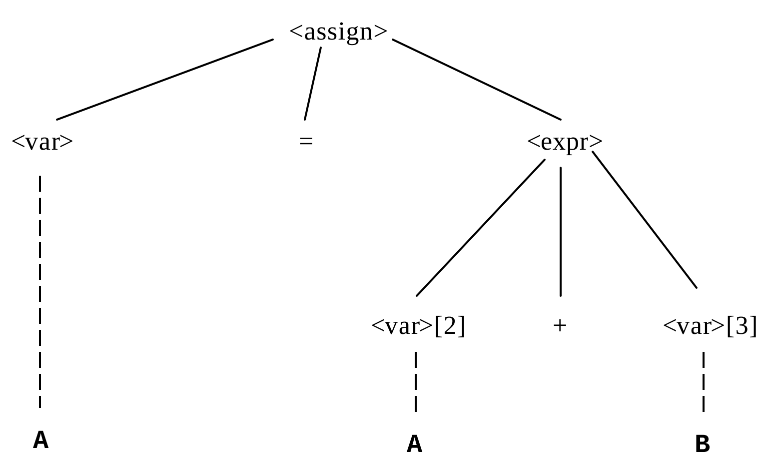
- Each symbol X has a set of attributes A(X).
- A(X) has two disjoint subsets:
  - S(X), synthesized attributes, which are passed up the parse tree
  - I(X), inherited attributes which are passed down the parse tree
- Each production of the grammar has a set of semantic functions and a set of predicate functions (which may be an empty set).
- Intrinsic attributes are synthesized attributes who properties are found outside the grammar (e.g., symbol table)

An Attribute Grammar for Assignments

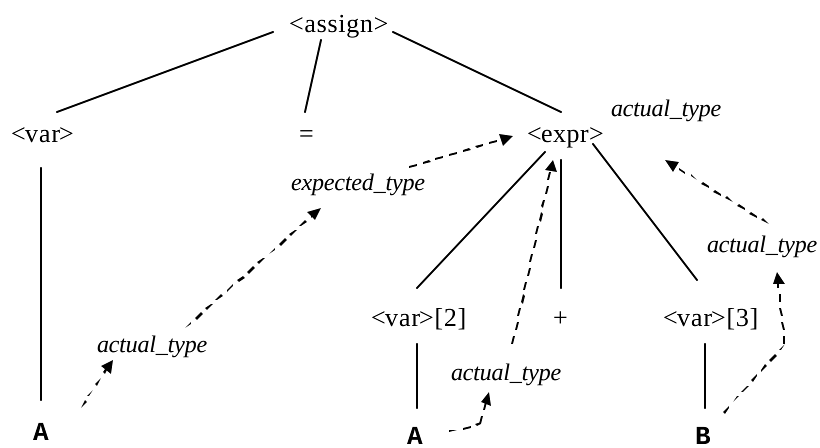
- Syntax rule: <assign> → <var> = <expr>
  Semantic rule: <expr>.expected_type ← <var>.actual_type
- Syntax rule: <expr> →<var>[2]+ <var>[3]
  Semantic rule: <expr>.actual_type ← if (<var>[2].actual_type = int)
  and if (<var>[3].actual_type = int) then int else real
  end if
  Predicate: <expr>.actual_type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

Parse Tree for **`A = A + B`**



Derivation of Attributes



Fully Attributed Parse Tree