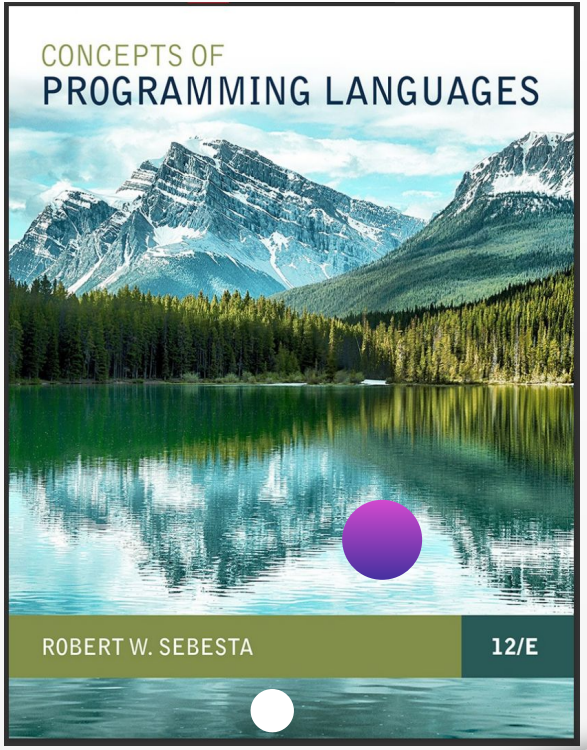
CHAPTER 3
Describing Syntax and Semantics

MEMBERS
Wayne Matthew Dayata
Jomar Leaño
Jade Andrie Rosales



Topic Outline

3.1 Introduction

3.2 The General Problem of Describing Syntax

3.3 Formal Methods of Describing Syntax

3.4 Attribute Grammars

3.5 Describing the Meanings of Programs: Dynamic Semantics

3.1 Introduction

Examining a Programming Language

QUESTION: What makes up a language definition?

SYNTAX

the **form or structure** of the expressions, statements, and program units

SEMANTICS

the **meaning** of the expressions, statements, and program units

Users of a language definition

- Other language designers, implementers, programmers (the users of the language)

Examining a Programming Language

```
while (boolean_expr) statement
```

NOTES:

- In a well-designed programming language, semantics should **follow directly** from syntax.
- the appearance of a statement should **strongly suggest** what the statement is meant to accomplish.

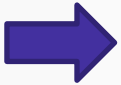
3.2 The General Problem of Describing Syntax

Terminology

- A ***sentence*** is a string of characters over some alphabet
- A ***language*** is a set of sentences
- A ***lexeme*** is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A ***token*** is a category of lexemes (e.g., identifier)

Example Statement

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



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

Formal Definition of Languages

- In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation**.

RECOGNIZER (parser)

- **reads input strings** over the alphabet of the language and decides whether the input strings **belong to the language**
- **Syntax analysis** part of a compiler

GENERATOR (grammar)

- device that can be used to generate the **sentences of a language**
- determine whether the **syntax of a particular statement is correct** by comparing it with the structure of the generator

3.3 Formal Methods of Describing Syntax

BNF/ Context-Free Grammars

Formal Methods of Describing Syntax

- The formal language generation mechanisms are usually called **grammars**
    - commonly used to describe the syntax of programming languages.

Backus-Naur Form and Context-Free Grammars

- It is a syntax description formalism that became the most widely used method for programming language syntax

BNF/ Context-Free Grammars

Context-Free Grammars

- Developed by **Noam Chomsky** in the mid-1950s
- **Language generators**, meant to describe the syntax of natural languages
- Define a class of languages called **context-free languages**

Backus-Naur Form (1959)

- Invented by **John Backus** to describe Algol 58
- BNF is equivalent to context-free grammars

BNF Fundamentals

- A **metalanguage** is a language that is used to describe another language. (BNF -> Programming languages)
- BNF uses abstractions for syntactic structures.
- The entire definition is called the **rule** or **production.**

<assign> □ <var> = <expression>

Left Hand Side (LHS)

abstraction being defined

Right Hand Side (RHS)

Definition □ mixture of tokens, lexemes, and references to other abstractions.

BNF Fundamentals

- A **grammar** is a finite nonempty set of rules.
  - Abstractions -> **nonterminals**.
  - Lexemes and tokens of the rules -> **terminals**
- A BNF description, or grammar, is simply a collection of rules.
- An abstraction can have **more than one RHS**
  - separated by the symbol **|**, meaning logical OR

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

or with the rule

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

BNF - Describing Lists

- We need a way of describing lists of syntactic elements in programming languages.
  - In math, we use ellipsis (. . .)
  - Here in BNF, this symbol does not exist.
- We apply **RECURSION.**
- A rule is recursive if its LHS appears in its RHS.

<ident_list> → identifier
| identifier, <ident_list>

3.3 Formal Methods of Describing Syntax

BNF – Grammars and Derivations

- The sentences of the language are **generated through a sequence of applications of the rules**, beginning with a special nonterminal of the grammar called the **start symbol**. (usually it is <program>)

- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

```
<program>   → begin <stmt_list>
end
<stmt_list> → <stmt> |
                <stmt> ; <stmt_list>
<stmt>      → <var> = <expr>
<var>       → a | b | c | d
<expr>      → <term> + <term> |
                <term> - <term>
<term>      → <var> | const
```

BNF – Grammars and Derivations

- Example derivation: **a = b + const**

```
<program> => <stmt_list>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

```
<program>   → begin <stmt_list>
end
<stmt_list> → <stmt> |
                <stmt> ; <stmt_list>
<stmt>      → <var> = <expr>
<var>       → a | b | c | d
<expr>      → <term> + <term> |
                <term> - <term>
<term>      → <var> | const
```

BNF – Grammars and Derivations

- Example derivation: **a = b + const**

```
<program> => <stmt_list>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

- Every string of symbols in the derivation, including <program>, is a **sentential form**.
- A **sentence** is a sentential form that has **only terminal symbols**.
- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded. The derivation continues **until the sentential form contains no nonterminals**.
- A derivation may be neither leftmost nor rightmost.

BNF – Grammars and Derivations

- Example 2: Grammar for assignment statement

Grammar

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

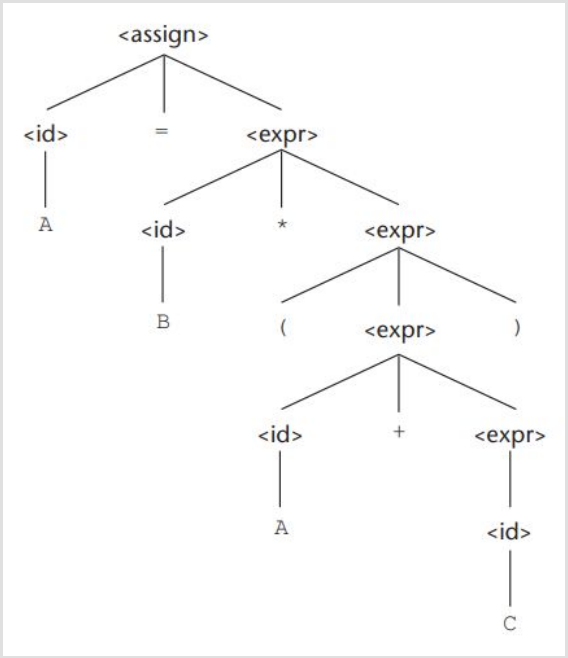
Example derivation:

B = C + (A * B)

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

BNF – Parse Tree

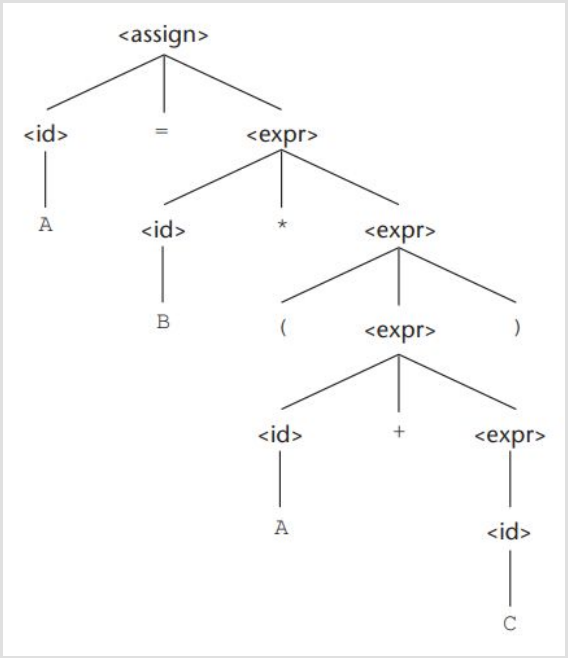
- A **parse tree** represents a hierarchical representation of a derivation by means of a tree.



BNF – Parse Tree

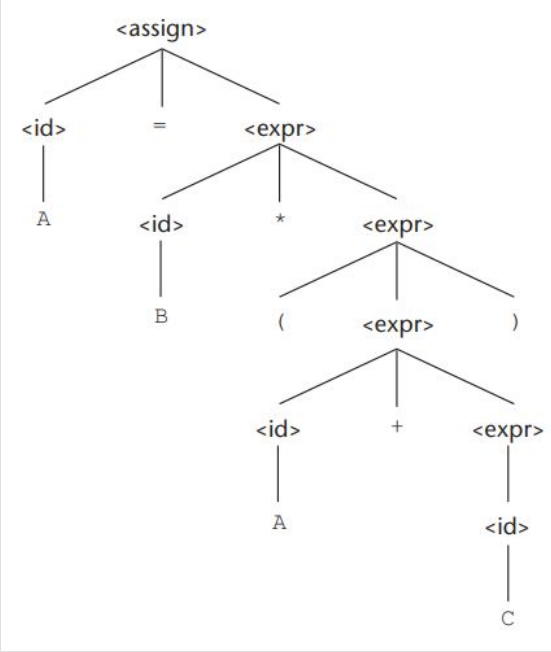
- A **parse tree** represents a hierarchical representation of a derivation by means of a tree.

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



BNF – Parse Tree

- A **parse tree** represents a hierarchical representation of a derivation by means of a tree.



OBSERVATIONS:

- **Internal node = nonterminal**
- **Leaf node = terminal**

BNF – Ambiguous Grammar

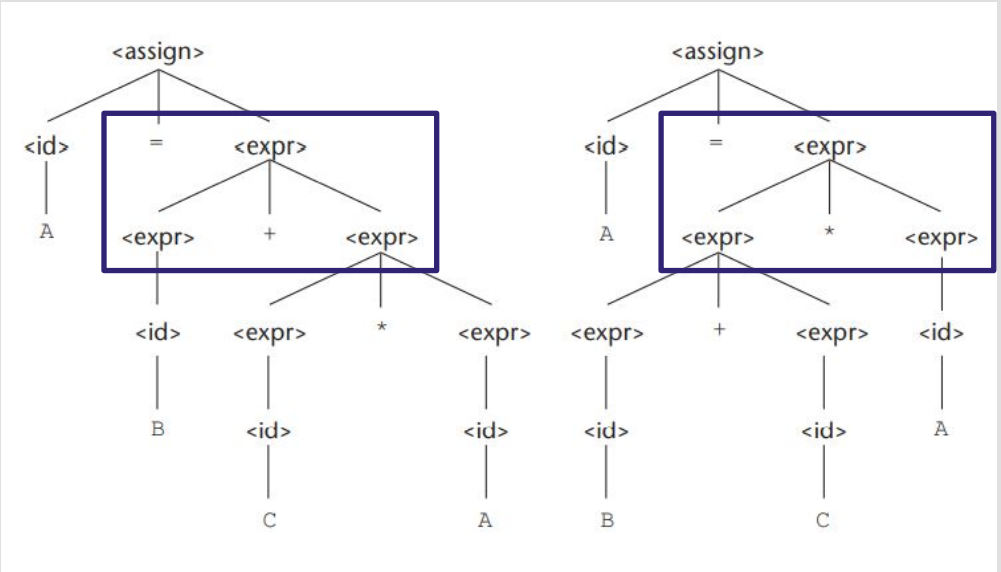
- What happens when we try to derive the following?

A = B + C * A

BNF – Ambiguous Grammar

- A grammar that generates a sentential form for which there are **two or more distinct parse trees** is said to be **ambiguous**.

- **Which comes first, addition or multiplication?**



BNF – Ambiguous Grammar

- Problem for ambiguous grammar:
  - Language structure cannot be determined uniquely

- Solution: We need to define
  - **Operator precedence**
  - **Associativity of operators**

- Grammar needs to be modified. (How?)

BNF – Ambiguous Grammar

Order of Precedence and Associativity in C

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** * **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

BNF – Ambiguous Grammar

Order of Precedence and Associativity in Java

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

BNF – Ambiguous Grammar

Order of Precedence and Associativity in Python

Precedence	Associativity	Operator	Description
18	Left-to-right	()	Parentheses (grouping)
17	Left-to-right	*f*(args...)	Function call
16	Left-to-right	*x*[index:index]	Slicing
15	Left-to-right	*x*[index]	Array Subscription
14	Right-to-left	**	Exponentiation
13	Left-to-right	~*x*	Bitwise not
12	Left-to-right	+*x* -*x*	Positive, Negative
11	Left-to-right	* / %	Multiplication Division Modulo
10	Left-to-right	+ -	Addition Subtraction
9	Left-to-right	<< >>	Bitwise left shift Bitwise right shift
8	Left-to-right	&	Bitwise AND
7	Left-to-right	^	Bitwise XOR
6	Left-to-right	|	Bitwise OR
5	Left-to-right	in, not in, is, is not, <, <=, >, >=, <>, == !=	Membership Relational Equality Inequality
4	Left-to-right	not *x*	Boolean NOT
3	Left-to-right	and	Boolean AND
2	Left-to-right	or	Boolean OR
1	Left-to-right	lambda	Lambda expression

BNF – Operator precedence

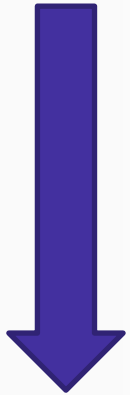
- Consider the grammar below. What are your observations?

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

BNF – Operator precedence

- Consider the grammar below. What are your observations?

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

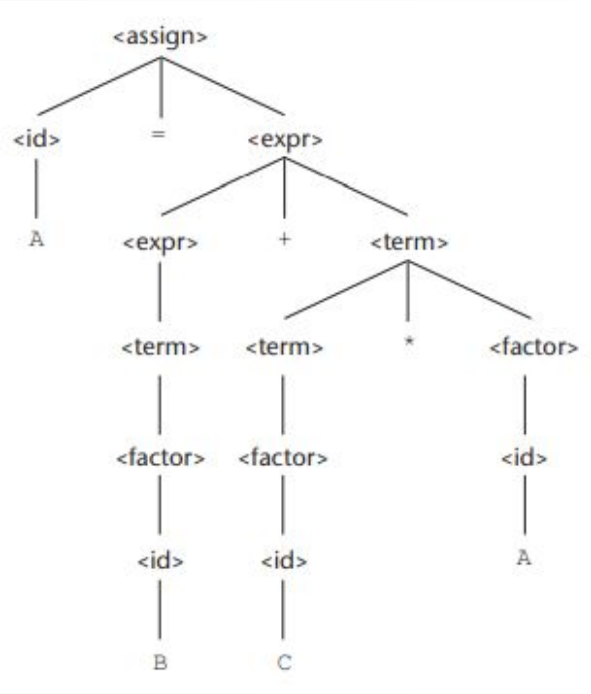


Higher precedence

BNF – Operator precedence

- Consider the grammar below. What are your observations?

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



BNF – Associativity of Operators

- As was the case with precedence, a grammar for expressions may correctly imply operator **associativity.**
- Addition is associative, but not in floating-point addition with limited digits of accuracy.
- Subtraction and division are not associative.

- Left recursive/associativity vs right recursive/associativity

<expr> □ <expr> + <term>
| <term>

Example: Addition -> Left associative

<factor> □ <exp> ** <factor>
<exp>

Example: Exponentiation -> Right associative

BNF – If-Else Grammar

- BNF Rules for a Java if-else statement:

<if_stmt> → if (<logic_expr>) <stmt>

| if (<logic_expr>) <stmt> else <stmt>

- However, this is ambiguous.

<stmt> → <if_stmt>

- How? Consider this:

if (<logic_expr>) if (<logic_expr>) <stmt> else <stmt>

Question: Which one belongs to which?

BNF – If-Else Grammar

- Solution: Make all **else** clauses match the nearest then clause.

<stmt> □ **<matched> | <unmatched>**

<matched □ **if (<logic_expr>) <matched> else <matched>**

| any non-if statement

<unmatched> □ **if (<logic_expr>) <stmt>**

| if (<logic_expr>) <matched> else <unmatched>

- Now the following only has one interpretation/parse tree.

if (<logic_expr>) if (<logic_expr>) <stmt> else <stmt>

Extended BNF

- Because of minor inconveniences in BNF, it has been extended in several ways.
- EBNF extensions do not enhance the descriptive powers of BNF but **only increase its readability and writability**.

01 Optional part of RHS

- Denoted by brackets []

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

Extended BNF

02 Implied iteration of tokens

- Denoted by braces {}, eliminates the need for recursion

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

03 Multiple-choice options

- Separate choices with | operator; enclose choices in parentheses ( )

<term> → <term> (* | / | %) <factor>

Extended BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → (<expr>)
      | id
```

BNF

Extended BNF

```
<expr>   → <term> {(+ | -) <term>}
<term>   → <factor> {(* | /) <factor>}
<factor> → <exp> {** <exp>}
<exp>    → (<expr>)
         | id
```

3.4 Attribute Grammars

Static Semantics

- BNF or Context-free grammars (CFGs) cannot describe all of the syntax of programming languages.
  - Examples: Type compatibility, variable declarations
- **Attribute grammar** – extension to CFGs

Types of semantics:

Static semantics

focuses on the legal forms of programs (syntax)

Dynamic semantics

deals with the meaning of expressions, statements, and program units.

Basic Concepts

- Attribute grammars are context-free grammars to which have been added the following:

Attributes

similar to variables in the sense that they can have values assigned to them

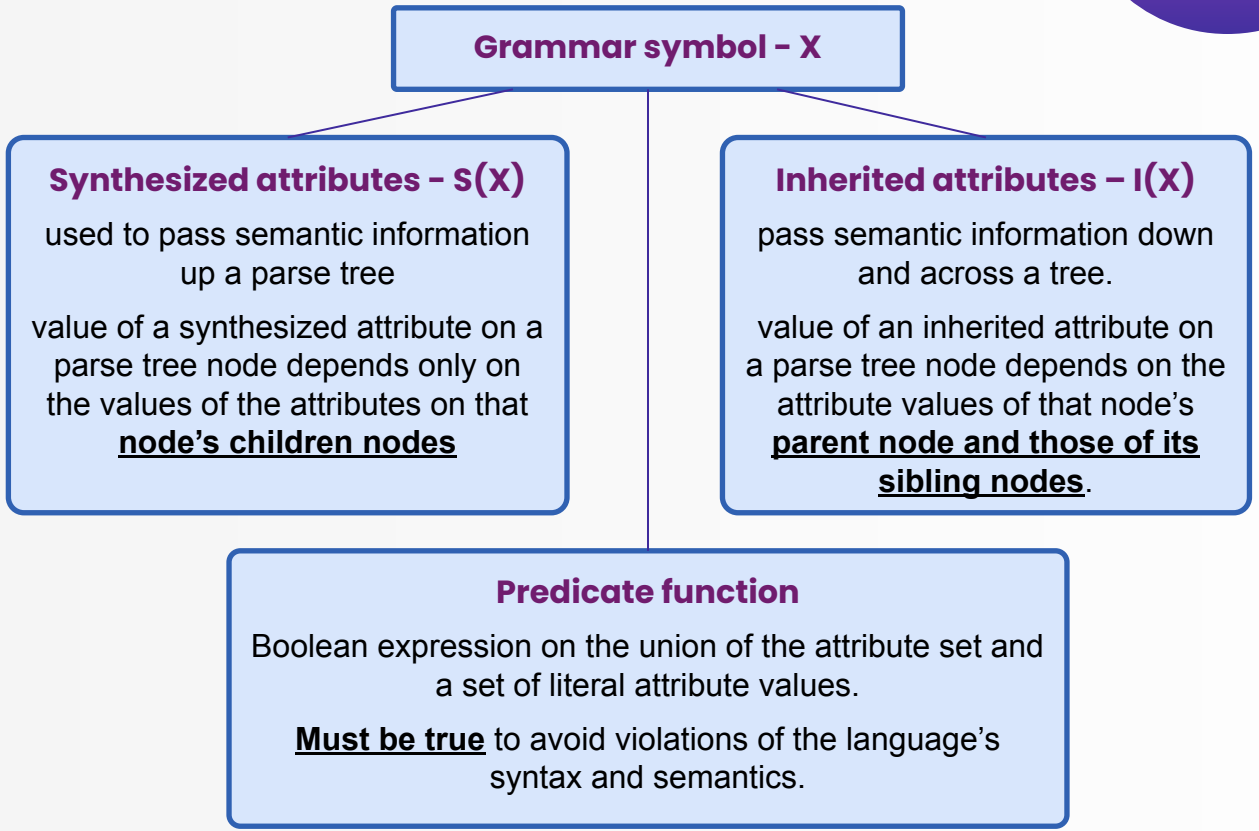
Attribute computation functions

specify how attribute values are computed

Predicate functions

state the static semantic rules of the language

Attribute Grammars Defined



Attribute Grammars Defined

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

- `actual_type`: synthesized for `<var>` and `<expr>`
- `expected_type`: inherited for `<expr>`

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
```

An Attribute Grammar for Simple Assignment Statements

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)
```

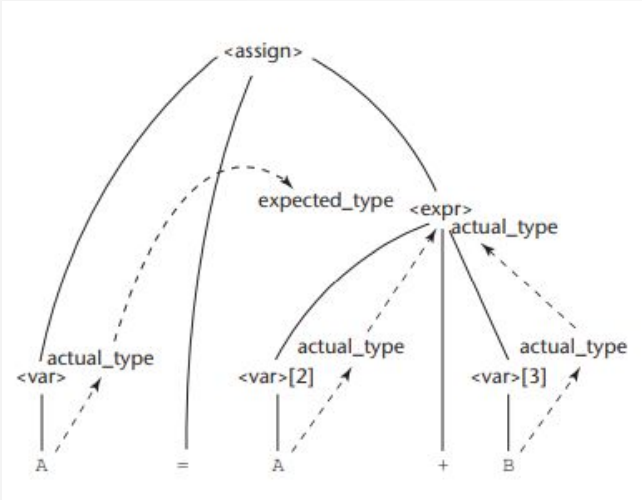
The look-up function looks up a given variable name in the symbol table and returns the variable's type.

Computing Attribute Values

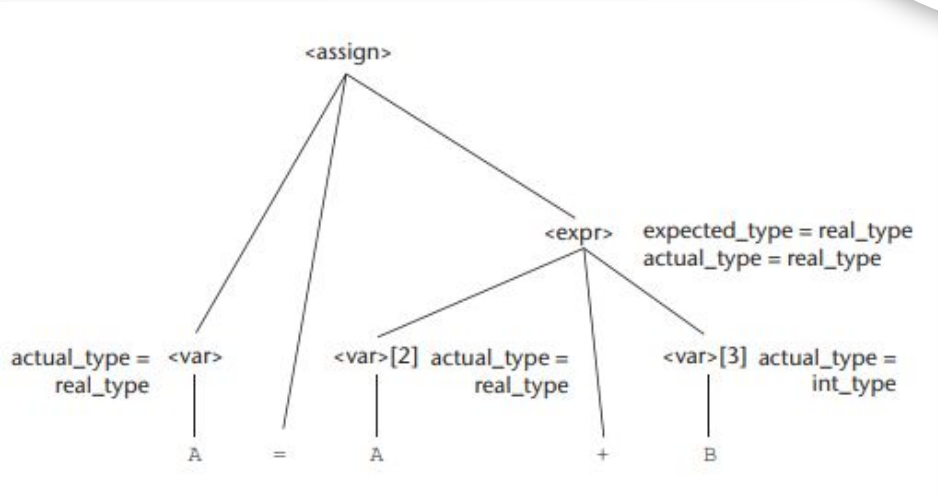
How are attribute values computed?

- If all attributes were inherited, the tree could be decorated in top-down order.
- If all attributes were synthesized, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Computing Attribute Values



Flow of attributes in the tree



Fully attributed parse tree

3.5 Describing the Meanings of Programs: Dynamic Semantics

Introduction

- There is no single, universally acceptable notation or formalism for describing semantics.
- Reasons underlying the need for a methodology and notation for describing semantics:
  - Programmers must know the **use of the language's statements**
  - Compiler writers must know exactly what **language constructs** do
  - **Correctness proofs** would be possible
  - **Compiler generators** would be possible
  - Designers could **discover ambiguities and inconsistencies**

General approaches for describing program semantics

Operational Semantics

method of **describing** the meaning of language constructs in terms of their effects on an ideal machine

Axiomatic semantics

based on **formal logic** and devised as a tool for proving the correctness of programs

Denotational semantics

mathematical objects are used to represents the meanings of languages constructs. Language entities are converted to these mathematical objects with recursive functions

Operational Semantics

- Aims to describe the meaning of a statement or program by specifying the **effects of running it on a machine**.
- Sequence of **changes in the machine's states** -> collection of values in its storage
- Executing a **compiled version** of the program on a computer.
- Utilizes **intermediate-level languages and interpreters** instead of real computers.

Operational Semantics

The process:

- Build a translator (translates source code to the machine code of an idealized computer)
- Build a simulator for the idealized computer

Evaluation of operational semantics:

- Good if used informally (language manuals, etc.)
- Extremely complex if used formally (e.g., VDL)

Operational Semantics

C Statement

```
for (expr1; expr2; expr3) {
   expr4;
}
```

Meaning

```
          expr1;
loop:  if expr2 == 0 then goto out
          expr4;
          goto loop;
out:    ...
```

- Operational semantics depends on programming languages **of lower level**, not mathematics and logic.
  - Unlike the other two semantic approaches

Operational Semantics

- Problems with this approach:
  - Individual steps in the machine language execution and the resulting changes to the state of the machine are too small and numerous.
  - Large and complex storage in real computers.

Levels of use of operational semantics:

Natural operational semantics

focuses on the final result of the execution of a complete program.

Structural operational semantics

determine the precise meaning of a program through examining the complete sequence of state changes that occur when the program is executed.

Denotational Semantics

- Solidly based on **recursive function theory**.
- **Most rigorous** and most widely known formal method for describing the meaning of programs.
- **Mathematical objects** are used to represents the meanings of languages constructs.
- Language entities are converted to these mathematical objects with recursive functions

Denotational Semantics

The method is named denotational because...

the mathematical objects denote the meaning of their corresponding syntactic entities

The process:

- Define a **mathematical object** for each language entity
- Define a **function** that maps instances of the language entities onto instances of the corresponding mathematical objects

The meaning of language constructs are defined by only the values of the program's variables.

Denotational Semantics - vs. Operational Semantics

- In operational semantics, state changes are defined by coded algorithms, written in some programming language.
- In denotational semantics, state changes are defined by mathematical function

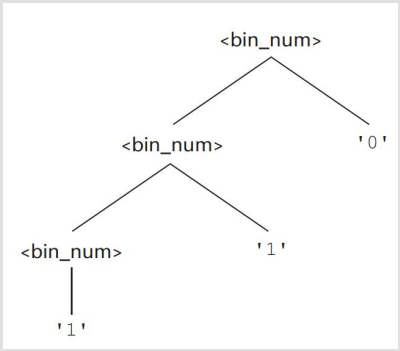
Denotational Semantics

- Example:
  - To introduce the denotational method we use a very simple language construct a character string representations of binary numbers. The syntax can be described by the following:

Grammar Rule

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

Parse Tree

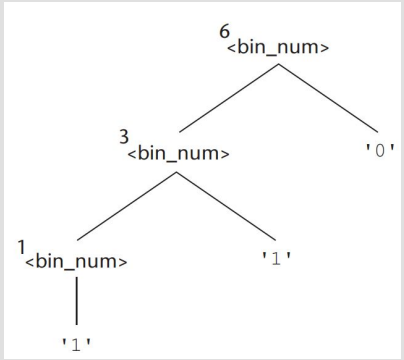


Denotational Semantics

- 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.

$$M_{bin}('0') = 0$$
$$M_{bin}('1') = 1$$
$$M_{bin}(\langle bin\_num\rangle\ '0') = 2 * M_{bin}(\langle bin\_num\rangle)$$
$$M_{bin}(\langle bin\_num\rangle\ '1') = 2 * M_{bin}(\langle bin\_num\rangle) + 1$$

Definition of function M_{bin}



Parse tree with denoted objects

Denotational Semantics - The State of a Program

- Let the state s of a program be represented as a set of ordered pairs.

  $S = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$

- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable.
  - The value of VARMAP (i_j , s) is v_j (the value paired with i_j in state s)

Denotational Semantics - Expressions

- **Expressions** are fundamental to most programming languages.
- The following is the BNF description of these expressions:

```
<expr> -> <dec_num> | <var> | <binary_expr>
<binary_expr> -> <left_expr> <operator> <right_expr>
<left_expr> -> <dec_num> | <var>
<right_expr> -> <dec_num> | <var>
<operator> -> + | *
```

Denotational Semantics - Expressions

- The only error we consider in expressions is a variable having an undefined value
- Map expressions onto Z ∪ {error}
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

```
<expr> -> <dec_num> | <var> | <binary_expr>
<binary_expr> -> <left_expr> <operator> <right_expr>
<left_expr> -> <dec_num> | <var>
<right_expr> -> <dec_num> | <var>
<operator> -> + | *
```

Denotational Semantics - Expressions

- Example

```
Me(<expr>, s) Δ = case <expr> of
                    <dec_num> => Mdec (<dec_num>, s)
                    <var> => if VARMAP (<var>, s) == undef
                               then error
                               else VARMAP (<var>, s)
                    <binary_expr> =>
                    if(Me (<binary_expr>.<left_expr>,s) == undef OR
                       Me (<binary_expr>.<right_expr>,s) == undef)
                    then error
                    else if (<binary_expr>.<operator> == '+')
                         then Me (<binary_expr>.<left_expr>,s) +
                              Me (<binary_expr>.<right_expr>,s)
                         else Me (<binary_expr>.<left_expr>,s) *
                              Me (<binary_expr>.<right_expr>,s)
```

Denotational Semantics - Assignment

- **Assignment statement** is an expression evaluation plus the setting of the target variable to the expression's value. In this case, the meaning function maps a state to a state. This function can be described with the following:

$$
\begin{aligned}
M_a\ (x = E,\ s)\ \triangle\ = \ &\text{if } M_e\ (E,s) == \text{error} \\
&\text{then error} \\
&\text{else } s' = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}, \text{ where} \\
&\quad \text{for } j = 1, 2, \ldots, n \\
&\quad\quad \text{if } i == x \\
&\quad\quad\quad \text{then } v_j = Me\ (E,s) \\
&\quad\quad\quad \text{else } v_j = VARMAP(i_j, s)
\end{aligned}
$$

Denotational Semantics - Logical Pretest Loop

- The denotational semantics of a **logical pretest loop** is deceptively simple. To expedite the discussion, we assume that there are two other existing mapping functions, M_{sl} and M_b, that map statement lists and states to states and Boolean expressions to Boolean values (or error), respectively.

```
M1 (while B do L, s) △ = if Mb (B,s) == undef
                         then error
                         else if Mb (B,s) == false
                              then s
                              else if Msl (L,s) == error
                                   then error
                                   else M1 (while B do L, Msl, (L,s))
```

Denotational Semantics - Logical Pretest Loop

- The value of the program variables after the statements in the loop have been executed the prescribed number of times, (assuming no errors)
- Loop converted from iteration to **recursion**, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion is easier to describe with mathematical rigor than iteration

Denotational Semantics

- Evaluation:
  - Can be used to prove the correctness of programs
  - Provides a rigorous way to think about programs
  - Can be an aid to language design
  - Has been used in compiler generation systems
  - Because of its complexity, it are of little use to language users

Axiomatic Semantics - Introduction

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called *assertions.*

Applications: program verification and program semantics specification

Axiomatic Semantics - Assertions and Weakest Preconditions

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics

Precondition = P,

Postcondition = Q;

Format: {P} Statement {Q}

Example:

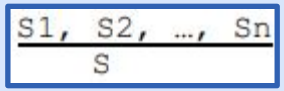
```
a = b + 1 {a > 1}

Possible precondition: {b > 10}
Weakest precondition: {b > 0}
```

Axiomatic Semantics

- If the weakest precondition can be computed from the given postcondition for each statement of a language, then **correctness proofs can be constructed** from programs in that language.
- The postcondition for the entire program is the **desired result**
  - Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.
- An Axiom is a logical statement that is assumed to be true.

Axiomatic Semantics - Sequences

- An **Inference Rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions.



The rule states that if S1, S2, …, and Sn are true, then the truth of S can be inferred. The top part of an inference rule is call its antecedent; the bottom part is called it consequent.

Axiomatic Semantics - Sequences

- An axiom for assignment statements

  $(x = E)\ \{Q_{x->E}\}\ x = E\ \{Q\}$

- The Rule of Consequence:

$$\frac{\{P\}S\{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Axiomatic Semantics - Sequences

- An inference rule for sequences of the form S1; S2

  {P1} S1 {P2}

  {P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axiomatic Semantics - Selection

- An inference rules for selection

  If B then S1 else S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S1\ \texttt{else}\ S2\ \{Q\}}$$

Axiomatic Semantics - Logical Pretest Loops

- An inference rule for logical pretest loops

  {P} while B do S end {Q}

$$\frac{\text{(I and B) S \{I\}}}{\text{\{I\} while B do S \{I and (not B)\}}}$$

Where I is the loop invariant (the inductive hypothesis)

Axiomatic Semantics

- Characteristics of the loop invariant must meet the following conditions:

- **P => I** – the loop invariant must be true initially
- **{I} B {I}** – evaluation of the Boolean must not change the
- validity of I
- **{I and B} S {I}** – I is not changed by executing by the body of the loop
- **(I and (not B)) => Q** – if I true and B is false, Q is implied
- The loop terminates – can be difficult to prove

Axiomatic Semantics

- The loop invariant I is a weakened version of the loop postcondition and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the exit loop condition, it must be strong enough to force the trite of the postcondition

Axiomatic Semantics

- Evaluation
  - Developing axioms or inference rules for all of the statements in a language is difficult
  - A good tool for correctness proofs, and excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
  - Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

THE END

Thank you for listening!

Presented by:

Wayne Matthew Dayata

Jomar Leano

Jade Andrie Rosales