Attribute Grammar

Principles of Programming Language

S.Venkatesan

Reference: Robert W. Sebesta, Concepts of Programming Languages, 10th Edition, Pearson

Lexical Analyser

index = 2 * count + 17;

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

Language

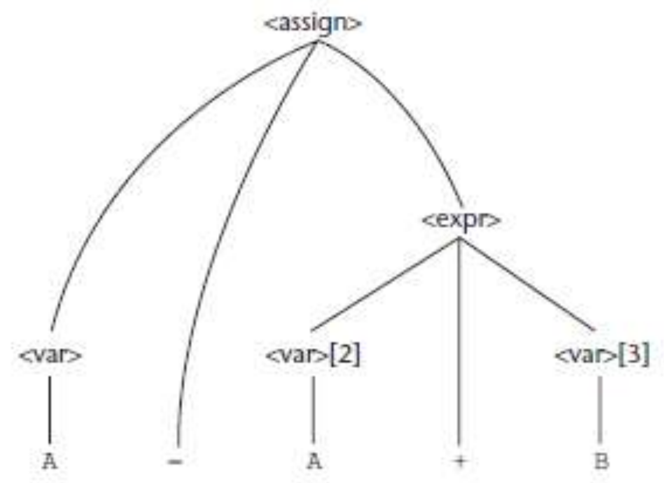
Generator Vs Recognizer

Formal Method of Defining Syntax

- Grammar [CFG]
- Representation – BNF and Extended BNF

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure (Parse Tree) of the sentences of the languages they define.

Parse Tree



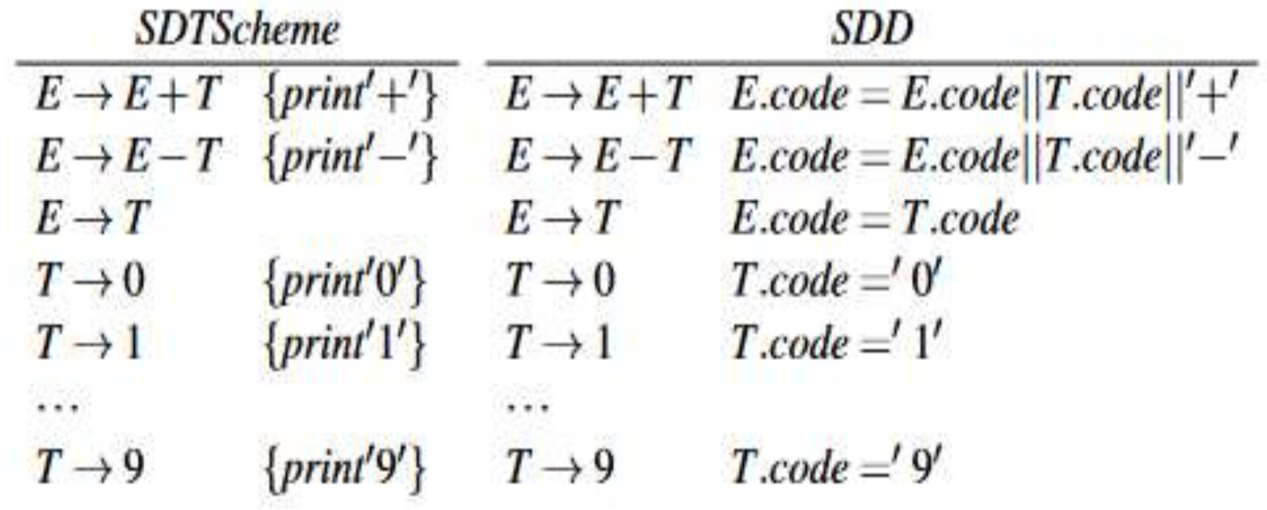
Attribute Grammar

- To describe more of the structure of a programming language that can be described with a context-free grammar.

- It is an extension to a context-free grammar.

- The extension allows certain language rules to be conveniently described, such as compatibility.

Infix to Postfix



SDTScheme		SDD	
$E \rightarrow E+T$	$\{print'+'\}$	$E \rightarrow E+T$	$E.code = E.code \| \| T.code \| \| '+'$
$E \rightarrow E-T$	$\{print'-'\}$	$E \rightarrow E-T$	$E.code = E.code \| \| T.code \| \| '-'$
$E \rightarrow T$		$E \rightarrow T$	$E.code = T.code$
$T \rightarrow 0$	$\{print'0'\}$	$T \rightarrow 0$	$T.code = '0'$
$T \rightarrow 1$	$\{print'1'\}$	$T \rightarrow 1$	$T.code = '1'$
$\cdots$		$\cdots$	
$T \rightarrow 9$	$\{print'9'\}$	$T \rightarrow 9$	$T.code = '9'$

http://www.cse.iitm.ac.in/~krishna/cs3300/lecture4.pdf

Not possible

- All variables must be declared before they are referenced.

Static Semantics

- Some characteristics of the structure of the programming languages that are difficult to describe and some impossible.

- For example, in Java. Assigning float to integer variable is not possible but reverse is legal.

- This can be done with BNF, however it needs additional terminals and rules. In such case, the grammar of a language will be too large.

- The size of the grammar determines the size of syntax analyser.

Static Semantic Rules

- Indirectly related to the meaning of programs during execution; rather it has to do with the legal forms of programs (syntax rather than semantics).

- In many languages it is for the type constraints.

- It is named static because to be done at the time compilation.

- To describe static semantics with BNF, attribute grammar was designed (Knuth 1968).

Attribute Grammar

- For describing and checking the correctness of the static semantic rules of a program.

- It is a CFG with an added attributes, attribute computation functions, and predicate functions.

- Attributes are associated with the grammar symbols (T and NT), are similar to variables in the sense that they can have values assigned to them.

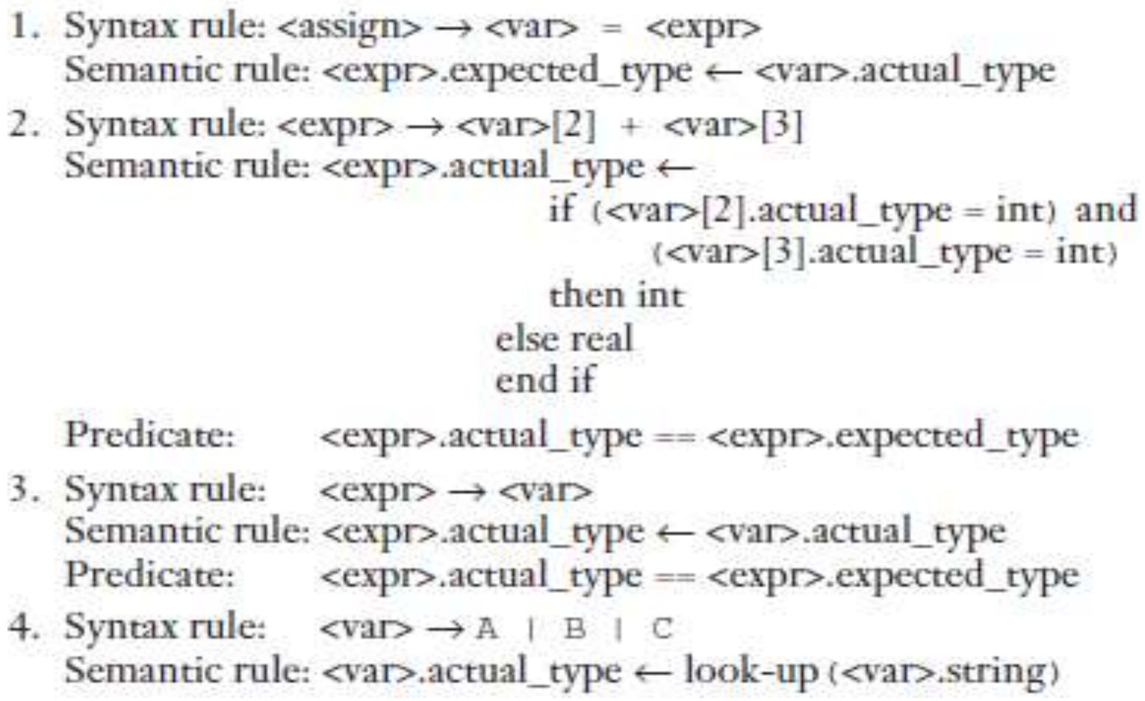
  – Attributed computation functions – semantic function associated with the grammar rules.

  – Predicate functions – state the static semantic rules of the language, are associated with grammar rules.

Features

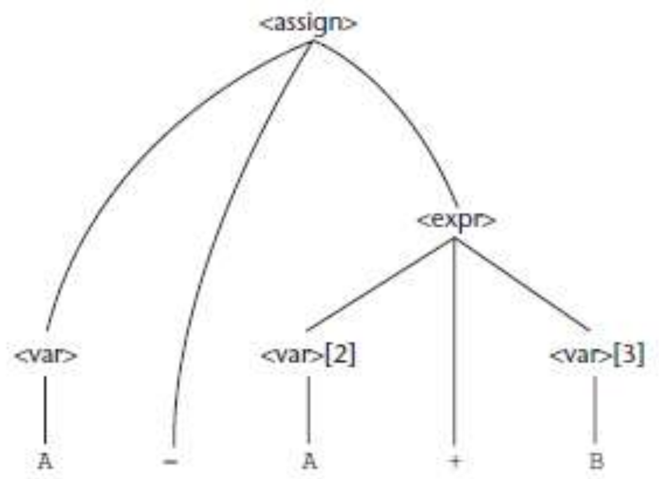
- Synthesized (actual type) and Inherited attributes (expected type).
- Semantic functions.
- Predicate function – true if associated NT is legal and false is illegal.
- Fully attributed and Intrinsic Attributes

Attribute Grammar

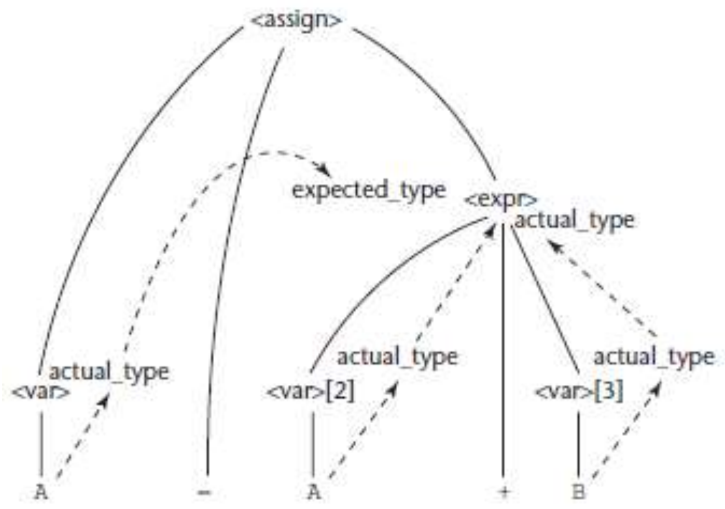


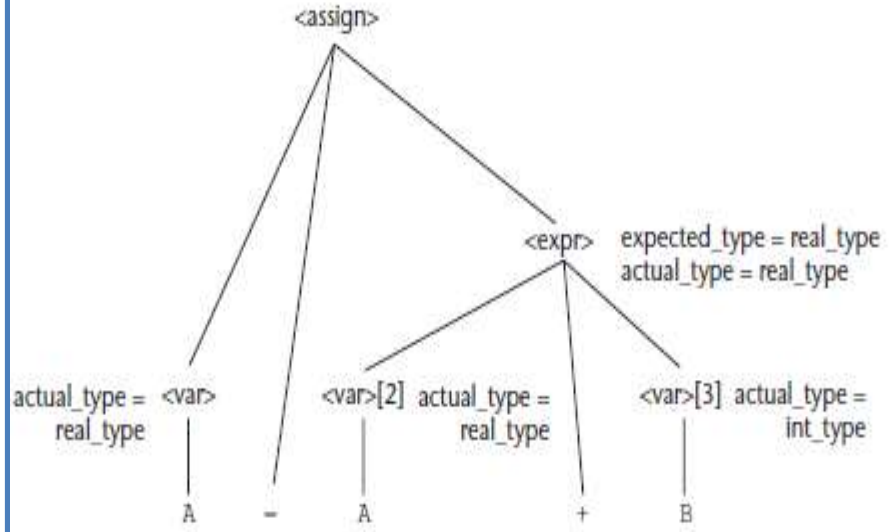
```
1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
                                  if (<var>[2].actual_type = int) and
                                         (<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)
```

Parse Tree



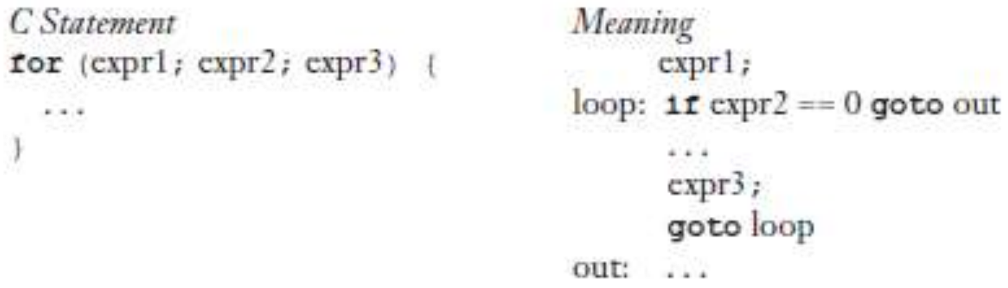
Computing Attribute Values





Dynamic Semantics

- This is to create an appropriate intermediate language.
- Operational semantics – describe the meaning of the statement.
  - Natural - the interest is in the final result of the execution of a complete program
  - Structural - operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed.



C Statement	*Meaning*

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

Denotational Semantics

- In denotational semantics, we define a language by assigning a mathematical meaning to functions; i.e., we say that each expression denotes a particular mathematical object.

- Operational - sourceExpression$_1$ → sourceExpression$_2$
- Denotational - sourceExpression$_1$ means → mathematicalEntity$_1$ = mathematicalEntity$_2$ means ← sourceExpression$_2$

- It has a domain and range
  - Domain is the collection of values that are legitimate parameters to the function.
  - The range is the collection of objects to which the parameters are mapped.
  - Syntactic domain - domain
  - Semantic domain - range

Difference

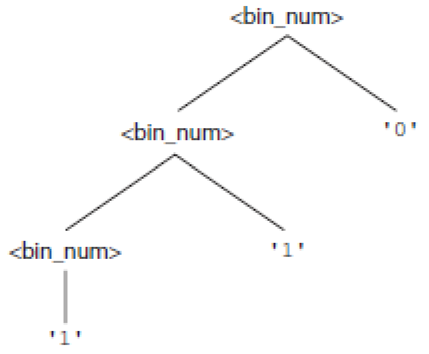
- In operational semantics - programming language constructs are translated into simpler programming language constructs, which becomes the basis of the meaning of the construct. Step by step processing of programs

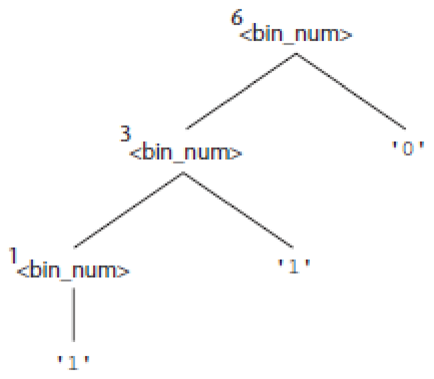
- In denotational semantics – programming language constructs are mapped to mathematical objects, either sets, or more often, functions. No step by step processing of programs.

Example

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

M_{bin}('0') = 0
M_{bin}('1') = 1
M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num>) + 1





The semantic function, named Mbin, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers.

Axiomatic Semantics

- Specifies, what can be proven about the program.
- Here, it is more about the relationship of the variables and constants.
- Applications
  - Program Verification
  - Program Semantics Specification

Assertions

- Constraints on the program variables at that point in the program.
  - Pre and Post Condition

```
sum = 2 * x + 1 {sum > 1}
```

- Weakest Pre-condition

- Inference Rule – top->antecedent and bottom -> consequent

$$\frac{S1, S2, S3 \ldots Sn}{S}$$

An **axiom** is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent.

Assignment Statement

- Let x = E be a general assignment statement and Q be its postcondition. Then, its weakest precondition, P, is defined by the axiom

  $P = Q_{x \rightarrow E}$
- which means that P is computed as Q with all instances of x replaced by E.

- For example, if we have the assignment statement and postcondition

  a = b / 2 - 1 {a < 10}
- the weakest precondition is computed by substituting b / 2 - 1 for a in the postcondition {a < 10}, as follows:

  b / 2 - 1 < 10

  b < 22
- {P} S {Q}

- rule of consequence = $\dfrac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$

- Try this : x = x + y - 3 {x > 10}

y > 13 - x

Sequences of statement

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

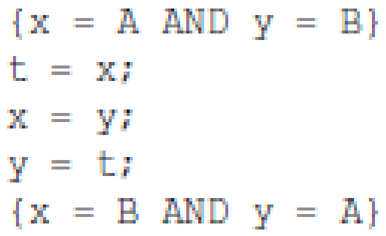
{x < 2}

- Selection

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

- Logical Pretest Loops - predicate transformer (one predicate is used for another predicate)

Program Proofs



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

Variables

- Attributes: (name, address, value, type, lifetime, and scope).

- Names
  - Are names case sensitive?
  - Are the special words of the language reserved words or keywords?

- The address of a variable is sometimes called its *l- value*, because the address is what is required when the name of a variable appears in the left side of an assignment.

- The **type** of a variable determines the range of values the variable can store and the set of operations that are defined for values of the type.

- A variable’s value is sometimes called its *r- value* because it is what is required when the name of the variable appears in the right side of an assignment statement. To access the *r- value*, the *l- value* must be determined first.

Binding

- A binding is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol.
- The time at which a binding takes place is called binding time.
- For example, the asterisk sym bol (*) is usually bound to the multiplication operation at language design time.
- At compile time, a variable in a Java program is bound to a particular data type

Type Binding

- Static Type Binding: explicit and implicit declaration -> default or inference

- Dynamic Type Binding – This is at the runtime.

Storage Bindings and Lifetime

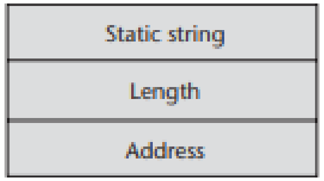
- The memory cell to which a variable is bound somehow must be taken from a pool of available memory. -> Allocation an Deallocation.

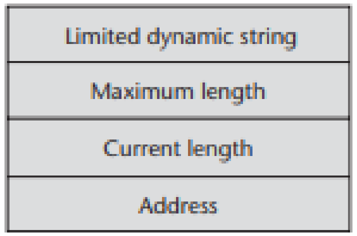
- Lifetime of a variable -> the time during which the variable is bound to a specific memory location.

- Static Variables - Bound to memory cells before program execution begins and remain bound to those same memory cells until pro gram execution terminates. [Global declaration and static keyword]

- Stack-Dynamic Variables

Data Types





- Primitive Data Types
- String
  - Should strings be a special kind of character array or a primitive type?
  - Should strings have static or dynamic length?
- Arrays – Static, Fixed Stack-Dynamic, Fixed Heap-Dynamic, Heap-Dynamic.
  - Rectangular and Jagged Arrays

Pointer – Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap- dynamic variable (the value a pointer references)?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both

Operations

- Assignment
- Dereferencing.

- **Dangling Pointers:** A dangling pointer, or dangling reference, is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

memory leakage

Reference Type

- A reference type variable is similar to a pointer, with one important and fundamental difference:
  - A pointer refers to an address in memory, while a reference refers to an object or a value in memory.

```
int result = 0;
int &ref_result = result;
```

Expressions

- An operator can be unary, meaning it has a single operand, binary, meaning it has two operands, or ternary, meaning it has three operands.

- Operator Evaluation Order – Precedence, Associativity, Parenthesis

- Operand Evaluation Order
  - Side effect: A side effect of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable.

```
a + fun(a)
```

```
int a = 5;
int fun1() {
    a = 17;
    return 3;
} /* end of fun1 */
void main() {
    a = a + fun1();
} /* end of main */
```

Referential Transparency and Side Effects

- A program has the property of referential transparency if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

- If the function fun has no side effects, **result1 and result2 will be equal**, because the expressions assigned to them are equivalent. However, suppose fun has the **side effect of adding 1 to either b or c.** Then **result1 would not be equal to result2**. So, that side effect violates the referential transparency of the program in which the code appears.

Assignment Statements

- Simple Assignments
- Conditional Targets
- Compound Assignment: `sum += value;`
- `Unary Assignment: sum = ++ count;`
- Assignment as an Expression
- Multiple Assignments