

Organization of Programming Languages

CS320/520N

Lecture 04

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Static Semantics

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages:
 - variables must be declared before they are referenced.
- Some syntactic rules are context free, but implementing them with a CFG would make it too large:
 - type compatibility rules (e.g. Java cannot assign float to integer)

This kind of rules are called **Static Semantics** rules.

Static Semantics

- **Static:** the rules can be checked at compile time.
- **Semantics:** the rules are only indirectly related to the meaning of programs (syntactic rather than semantic).
- **Attribute Grammars (Knuth, 1968):**
 - a formalism for describing both the syntax & static semantics of programs.
 - CFGs augmented to carry some semantic info on parse tree nodes.

Attribute Grammars

- An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
 - For each grammar symbol x there is a set $A(x)$ of **attributes**, where each attribute can have a value assigned to it.
 - Each rule has a set of **semantic functions** that are used to specify how attribute values are computed.
 - Each rule has a (possibly empty) set of **predicate functions** to check for attribute consistency (i.e. static semantics rules).

Attribute Grammars

- Let $X_0 \rightarrow X_1 \dots X_n$ be a rule.
- Set of attributes $A(X_0) = S(X_0) \cup I(X_0)$
 - $S(X_0)$ are **synthesized attributes**.
 - $I(X_0)$ are **inherited attributes**.
- Initially, there are **intrinsic attributes** on the leaves:
 - Synthesized attributes that are determined outside the parse tree.
 - Example: the type of a variable could come from the symbol table.

Attribute Grammars

- **Syntax:**

`<assign> -> <var> = <expr>`

`<expr> -> <var> + <var> | <var>`

`<var> -> A | B | C`

- **Attributes:**

- `actual_type`:

- synthesized for `<var>` and `<expr>`

- `expected_type`:

- inherited for `<expr>`

Attribute Grammars

- Semantic functions:
 - **Synthesized attributes:** $S(X_0) = f(A(X_1), \dots, A(X_n))$
 - **Inherited attributes:** $I(X_j) = f(A(X_0), \dots, A(X_n))$, for $i \leq j \leq n$:
 - $I(X_j) = f(A(X_0), \dots, A(X_{j-1}))$ to avoid circularity.
- Predicates:
 - Boolean expressions on the union of $A(X_0) \cup \dots \cup A(X_n)$ with a set of literal attribute values.
 - A false value for a predicate function indicates a violation of the syntax or static semantics rules of the language.
 - \Rightarrow allow only derivations where all predicates associated with a nonterminal evaluate to true.

Attribute Grammars

- **Syntax rule:** `<assign> -> <var> = <expr>`
 - **Semantic function:**
`<expr>.expected_type ← <var>.actual_type`
- **Syntax rule:** `<expr> → <var>[2] + <var>[3]`
 - **Semantic function:**
`<expr>.actual_type ←
if (<var>[2].actual_type == int &&
 <var>[3].actual_type == int)
then int
else float`
 - **Predicate:**
`<expr>.expected_type == <expr>.actual_type`

Attribute Grammars

- **Syntax rule:** $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$
 - **Semantic function:**
 $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
 - **Predicate:**
 $\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual_type}$
- **Syntax rule:** $\langle \text{var} \rangle \rightarrow A \mid B \mid C$
 - **Semantic function:**
 $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{lookup} (\langle \text{var} \rangle.\text{string})$

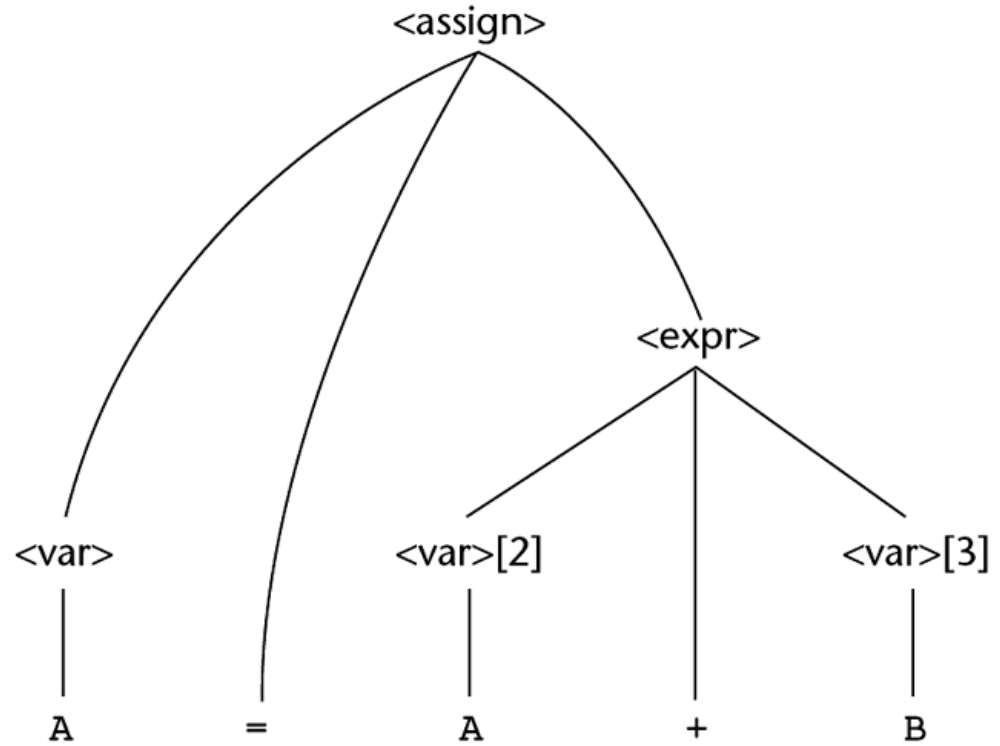
Attribute Grammars

- How are attribute values computed?
 - If all attributes were inherited, the parse tree could be decorated in top-down order.
 - If all attributes were synthesized, the parse 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.
- Determining attribute evaluation order in general case is a complex problem:
 - Build dependency graph showing all attribute dependencies.

Attribute Grammars: Example

Figure 3.6

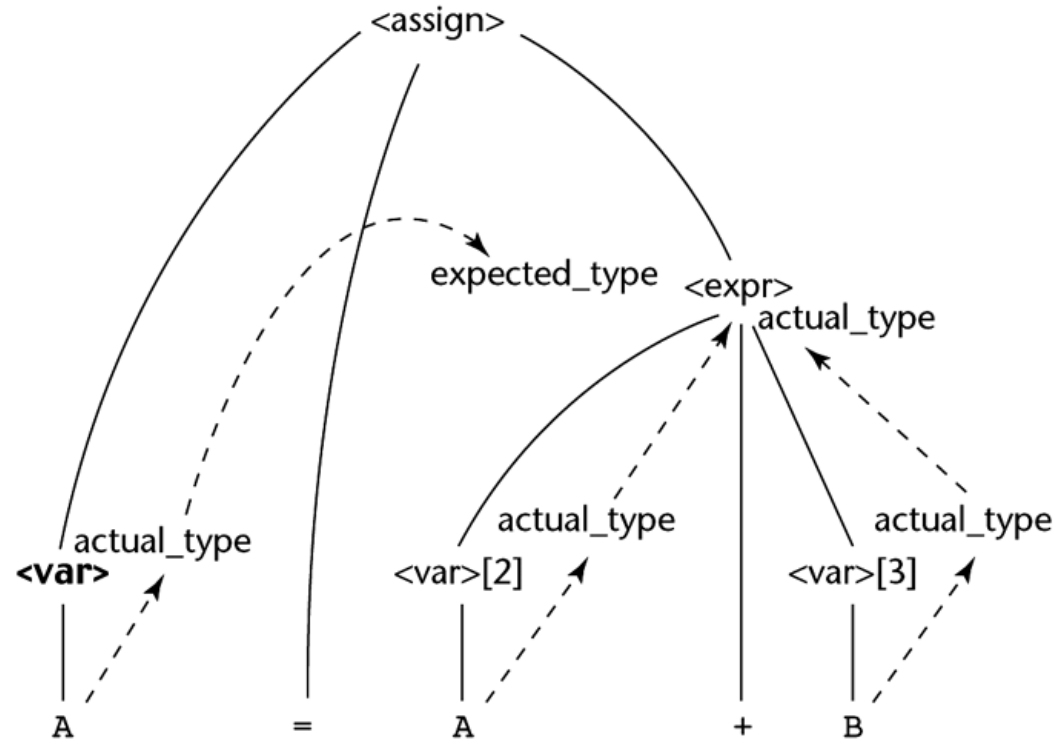
A parse tree for
 $A = A + B$



Attribute Grammars: Example

Figure 3.7

The flow of attributes in the tree



Attribute Grammars

`<var>.actual_type ← lookup (A)`

`<expr>.expected_type ← <var>.actual_type`

`<var>[2].actual_type ← lookup (A)`

`<var>[3].actual_type ← lookup (B)`

`<expr>.actual_type ← either int or float`

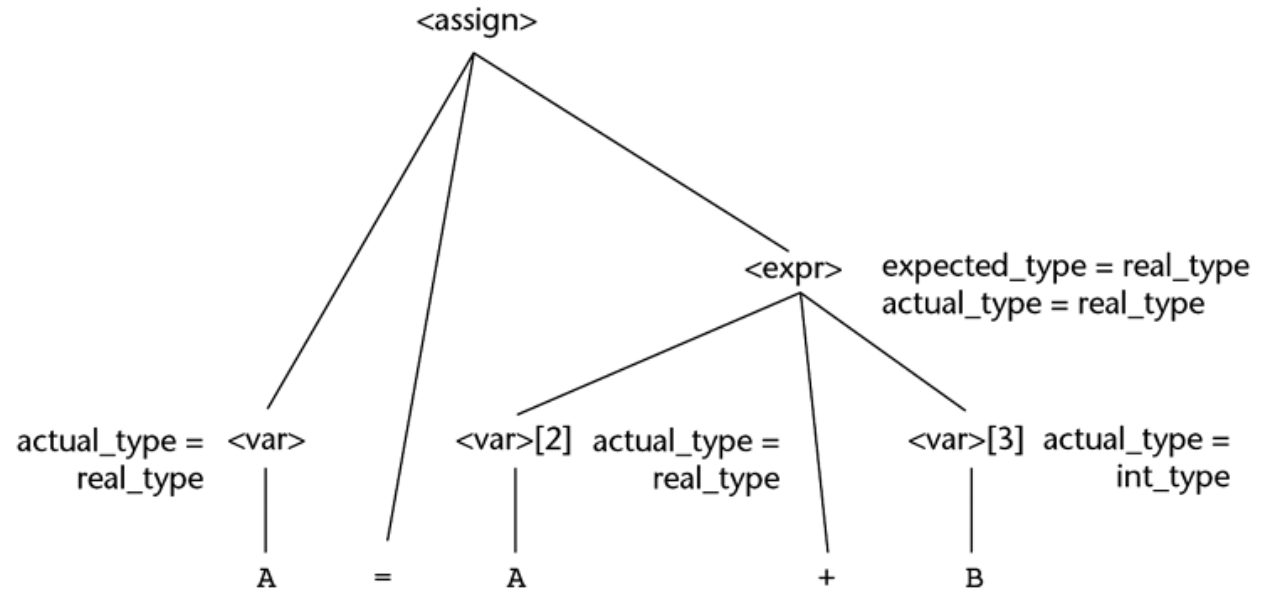
`<expr>.expected_type =? <expr>.actual_type`

`⇒ True or False`

Attribute Grammars: Example

Figure 3.8

A fully attributed parse tree



Dynamic Semantics

- **Dynamic Semantics** describes the meaning of expressions, statements and program units.
- Informal Semantics:
 - Describe what statements do using English explanations.
 - Pros:
 - less complex than formal descriptions.
 - still used by compiler writers.
 - Cons:
 - often imprecise and incomplete.
 - cannot be used to automatically prove statements about programs.

Dynamic Semantics

- There is no single widely acceptable notation or formalism for describing semantics.
- Three approaches for specifying the semantics of a PL:
 - Operational Semantics.
 - Axiomatic Semantics.
 - Denotational Semantics.

Operational Semantics

- Describe the meaning of a program by:
 - Translating it into a more easily understood language.
 - Executing its statements on a machine, either simulated or actual.
 - The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement.
- Uses of operational semantics:
 - Language manuals and textbooks.
 - Teaching programming languages.
- Good if used informally (language manuals, etc.), extremely complex if used formally.

Axiomatic Semantics

- Based on formal logic (Predicate Calculus):
 - Original purpose: formal program verification.
 - The meaning of statements is formulated in terms of logical expressions (assertions).
 - Axioms and inference rules are defined for each statement.
- Pros:
 - It is a good tool for correctness proofs, and an excellent framework for reasoning about programs.
- Cons:
 - Developing axioms or inference rules for all statements in a language is difficult.
 - It is not as useful for language users and compiler writers.

Denotational Semantics

- Originally developed by Scott and Strachey in the 1960s.
- The most rigorous & widely known method for describing the meaning of programs.
- Based on recursive function theory.

Denotational Semantics

- The process of building a denotational specification for a language:
 - 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 change in the values of the program's variables (the state of the program).

Denotational Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms.
- In denotational semantics, the state changes are defined by rigorous mathematical functions.

Program State

- The state of a program is given by the values of all its current variables:

$$s = \{ \langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle \}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable:

$$\text{VARMAP}(i_j, s) = v_j$$

- VARMAP can also return the special value undef.

Denotational Semantics: Example

- An expression grammar:

$\langle \text{expr} \rangle \rightarrow \langle \text{dec_num} \rangle \mid \langle \text{var} \rangle \mid \langle \text{binary_expr} \rangle$

$\langle \text{binary_expr} \rangle \rightarrow \langle \text{left_expr} \rangle \langle \text{op} \rangle \langle \text{right_expr} \rangle$

$\langle \text{left_expr} \rangle \rightarrow \langle \text{dec_num} \rangle \mid \langle \text{var} \rangle$

$\langle \text{right_expr} \rangle \rightarrow \langle \text{dec_num} \rangle \mid \langle \text{var} \rangle$

$\langle \text{op} \rangle \rightarrow + \mid *$

- **Syntax Directed Semantics:** syntactic entities are mapped to mathematical objects with concrete meaning.
- **Compositional Semantics:** the meaning of LHS is derived from the meaning of symbols on the RHS of the rule.

Decimal Numbers

- `<dec_num>` is mapped to a number from \mathbb{Z} through M_{dec}

`<dec_num>` → '0' | '1' | '2' | '3' | '4' | '5' |
'6' | '7' | '8' | '9' |
`<dec_num>` ('0' | '1' | '2' | '3' |
'4' | '5' | '6' | '7' |
'8' | '9')

$$M_{\text{dec}}('0') = 0, \quad M_{\text{dec}}('1') = 1, \quad \dots, \quad M_{\text{dec}}('9') = 9$$

$$M_{\text{dec}}(\text{<dec_num> '0'}) = 10 * M_{\text{dec}}(\text{<dec_num>})$$

$$M_{\text{dec}}(\text{<dec_num> '1'}) = 10 * M_{\text{dec}}(\text{<dec_num>}) + 1$$

...

$$M_{\text{dec}}(\text{<dec_num> '9'}) = 10 * M_{\text{dec}}(\text{<dec_num>}) + 9$$

Expressions

- `<expr>` is mapped onto $Z \cup \{error\}$.

```
Me(<expr>, s) =
  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)
```

Assignment Statements

- Map a state to another state or *error*.

```
Ma(x := <expr>, s) =  
  if Me(<expr>, s) == error  
  then error  
  else s' = {<i1, v1'>, <i2, v2'>, ..., <in, vn'>} ,  
           where for j = 1, 2, ..., n,  
             if ij == x  
             then vj' = Me(<expr>, s)  
             else vj' = VARMAP(ij, s)
```

Logical Pretest Loops

- Map a state to another state or *error*.

```
M1(while B do L, s) =  
  if Mb(B, s) == error  
  then error  
  else if Mb(B, s) == false  
  then s  
  else if Ms1(L, s) == error  
  then error  
  else M1(while B do L, Ms1(L, s))
```

The Meaning of Loops

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors.
 - In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions.
 - Recursion, when compared to iteration, is easier to describe with mathematical rigor.

Denotational Semantics

- Pros:
 - 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.
- Cons:
 - Because of its complexity, it is of little use to language users.

Summary

- **Static Semantics:**
 - Attribute Grammars.
- **Dynamic Semantics:**
 - Operational Semantics.
 - Axiomatic Semantics.
 - Denotational Semantics.