Programming Languages – Describing Syntax and Semantics

Jongwoo Lim

Introduction

- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units
  - Syntax and semantics provide a language's definition
    - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

Introduction

- Example: Fibonacci numbers in C/C++ and in Haskel.

```
int fibonacci(int iterations) {
  int first = 0, second = 1;  // seed values
  for (int i = 0; i < iterations; ++i) {
    int sum = first + second;
    first = second;
    second = sum;
  }
  return first;
}
```

```
fibRecurrence first second =
    first : fibRecurrence second (first + second)
fibonacci = fibRecurrence 0 1
main = print (fibonacci !! 10)
```

The General Problem of Describing Syntax: Terminology

- A **language** is a set of sentences.
- A **sentence** is a string of characters over some alphabet.

```
English: I like Programming Languages.
C/C++: index = 2 * count + 17;
```

- **Lexeme** is the lowest level syntactic unit of a language.

```
English: I, like, Programming, Languages, .
C/C++: index, =, 2, *, ;
```

- **Token** is a category of lexemes.

```
English: pronoun, verb, noun, symbol_period, …
C/C++: identifier, equal_sign, int_literal, …
```

Formal Definition of Languages

- **Recognizers**
  - Is the given sentence in the language?
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language.
  - Example: syntax analysis part of a compiler.
  - Detailed discussion of syntax analysis appears in Chapter 4.
- **Generators**
  - A device that generates sentences of a language.
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator.

BNF and 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.
  - Block structure:

```
John, whose blue car was in the garage, walked to the store.
(John, ((whose blue car) (was (in the garage))), (walked
(to (the store)))).
```

BNF and Context-Free Grammars

- **Backus-Naur Form** (1959)
  - Invented by John Backus to describe Algol 58.
  - BNF is equivalent to context-free grammars.
  - BNF is a metalanguage for programming languages.

BNF and Context-Free Grammars

- In BNF, **abstractions** are used to represent classes of syntactic structures – they act like syntactic variables.
  - Also called **nonterminal symbols**, or just **nonterminals**.
  - Nonterminals are often enclosed in angle brackets.

```
<identifier>, <equal_sign>, <int_literal>
```

- **Terminals** are lexemes or tokens.
- A **rule** has a left-hand side (LHS) and a right-hand side (RHS).
  - A left-hand side (LHS) is a nonterminal.
  - A right-hand side (RHS) is a string of terminals and/or nonterminals.

```
<assign> → <var> = <expression>
```

BNF Fundamentals

- A nonterminal can have more than one RHS.

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

  - Examples:

```
if (i == 0) a = b + 1;
if (a > 0.0) positive = true; else positive = false;
```

- Recursive definition:

```
<ident_list> → <ident>
             | <ident> , <ident_list>
```

  - Examples:

```
1         (O)
1,2,3,4   (O)
1,2 3,4   (X)
```

BNF Fundamentals

- **Grammar**: a finite non-empty set of rules
  - The sentences are generated through applications of the rules, beginning with the **start symbol** (a nonterminal).

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> - <var>
             | <var>
```

Derivation

- Derivation:
  - Repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols).
  - **Sentential form**: string of symbols in a derivation.
    - A sentence is a sentential form that has only terminal symbols.
  - **Leftmost derivation**:
    the leftmost nonterminal in each sentential form is expanded.
  - A derivation may be neither leftmost nor rightmost.

An Example 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
```

Example 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>
```

Another Example: Simple Assignment

- Example statement:

```
A = B * ( A + C )
```

- 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 )
```

Simple Assignment

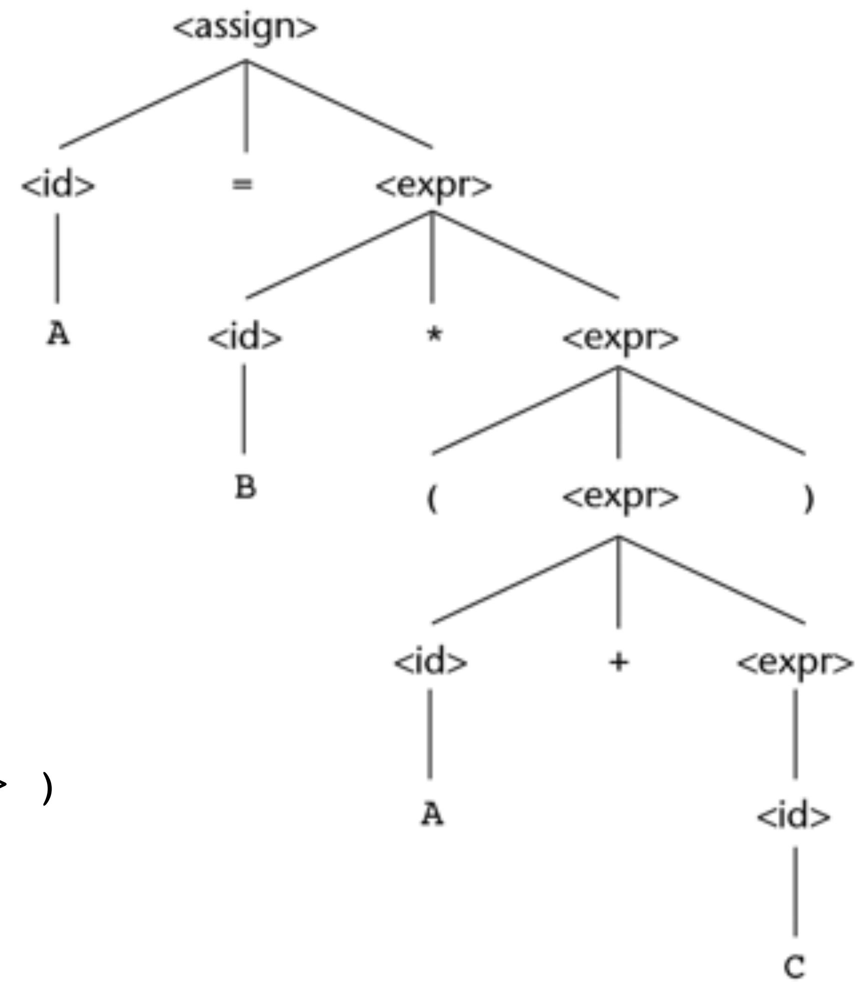
```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr> )
       | <id>
```

Parse Tree

- A hierarchical representation of a derivation.

```
Simple Assignment

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

```
A = B * ( A + C )
<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 )
```



Simple Assignment

- Can the grammar accept the following equation?

```
A = ( A + C ) * B
```

Simple Assignment	Modified Simple Assignment

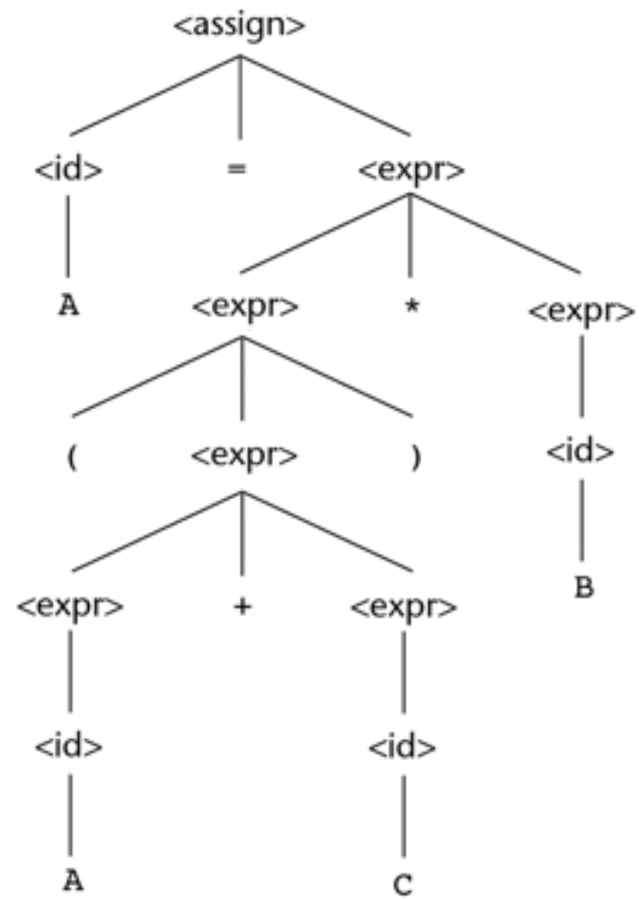
`<assign> → <id> = <expr>`
`<id> → A | B | C`
`<expr> → <id> + <expr>`
`        | <id> * <expr>`
`        | ( <expr> )`
`        | <id>`
	`<assign> → <id> = <expr>`
	`<id> → A | B | C`
	`<expr> → <expr> + <expr>`
	`        | <expr> * <expr>`
	`        | ( <expr> )`
	`        | <id>`

Parse Tree with Modified Grammar

Modified Simple Assignment

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

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



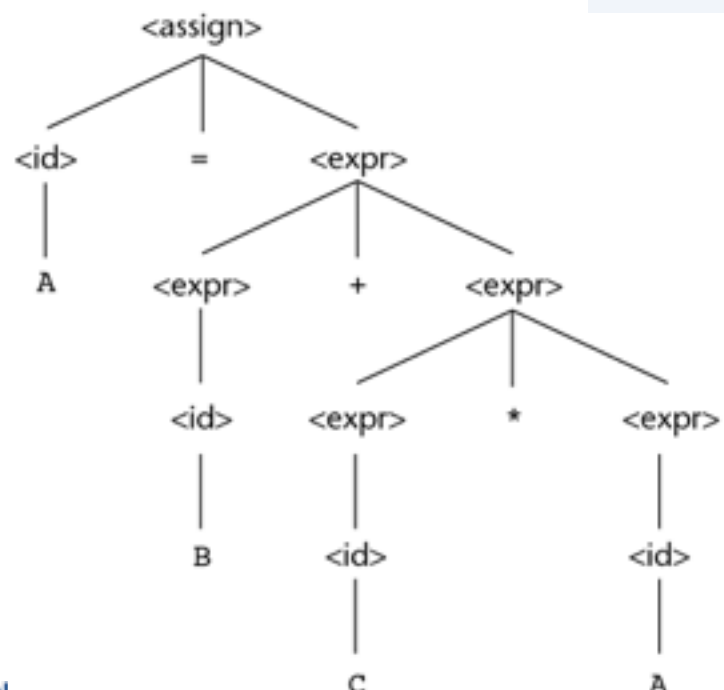
Ambiguous Grammar

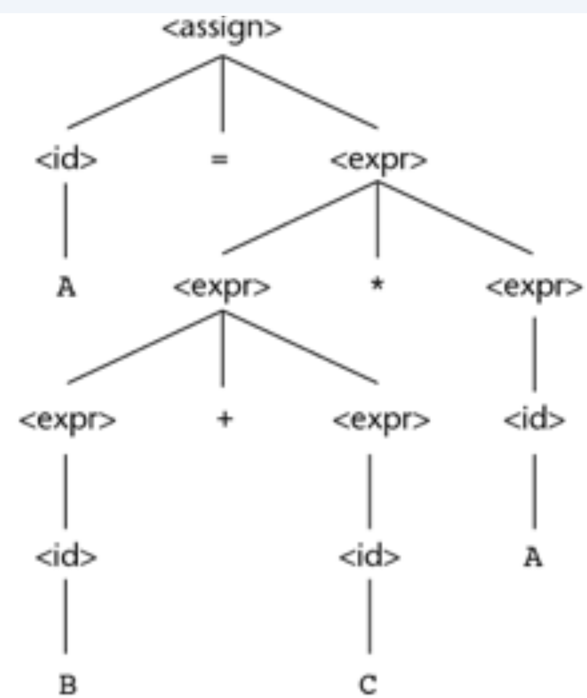
- Example statement:

  `A = B + C * A`

Modified Simple Assignment : Ambiguous

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

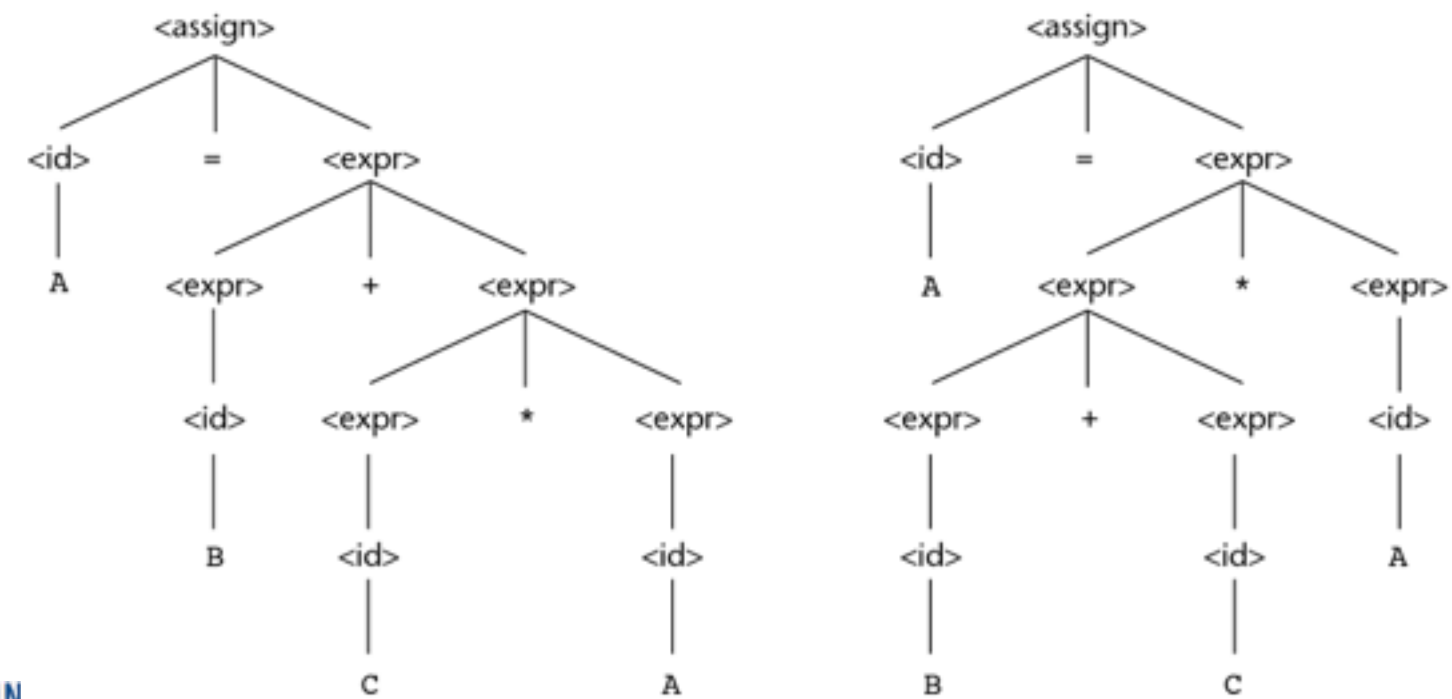




Ambiguous Grammar

- A grammar is **ambiguous** if there exists a string that can have more than one parse tree (leftmost derivation).

```
A = B + C * A
```

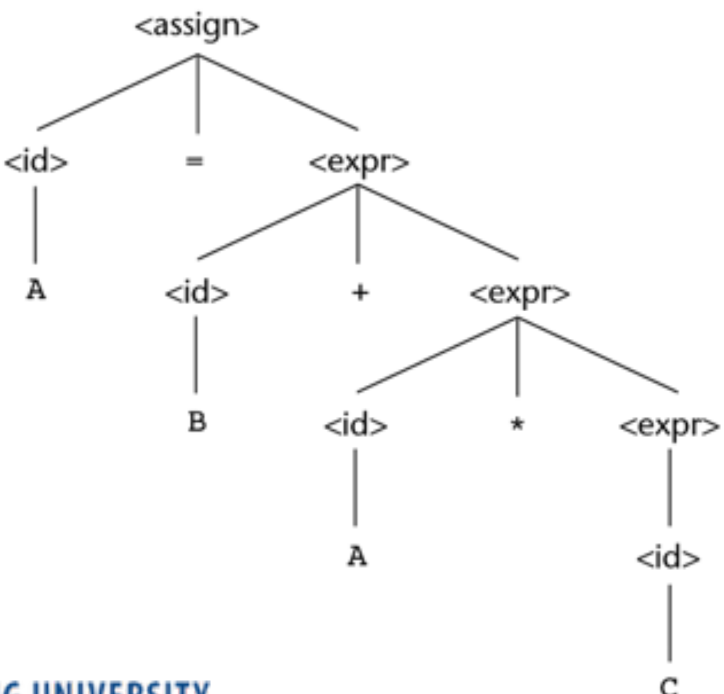


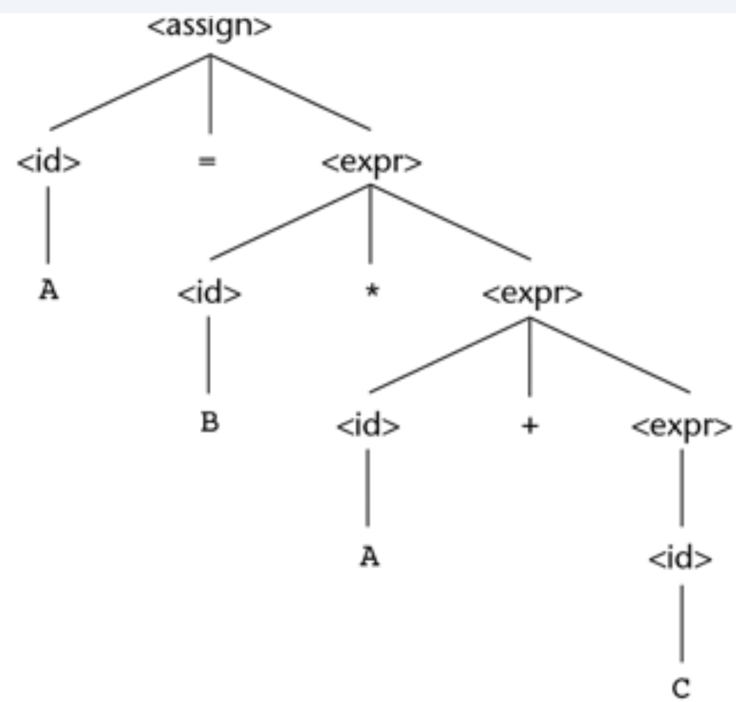
Simple Assignment Revisited

- The precedence order of operators is not usual.

```
A = B + A * C
A = B * A + C
```

```
Simple Assignment

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





An Unambiguous Expression Grammar

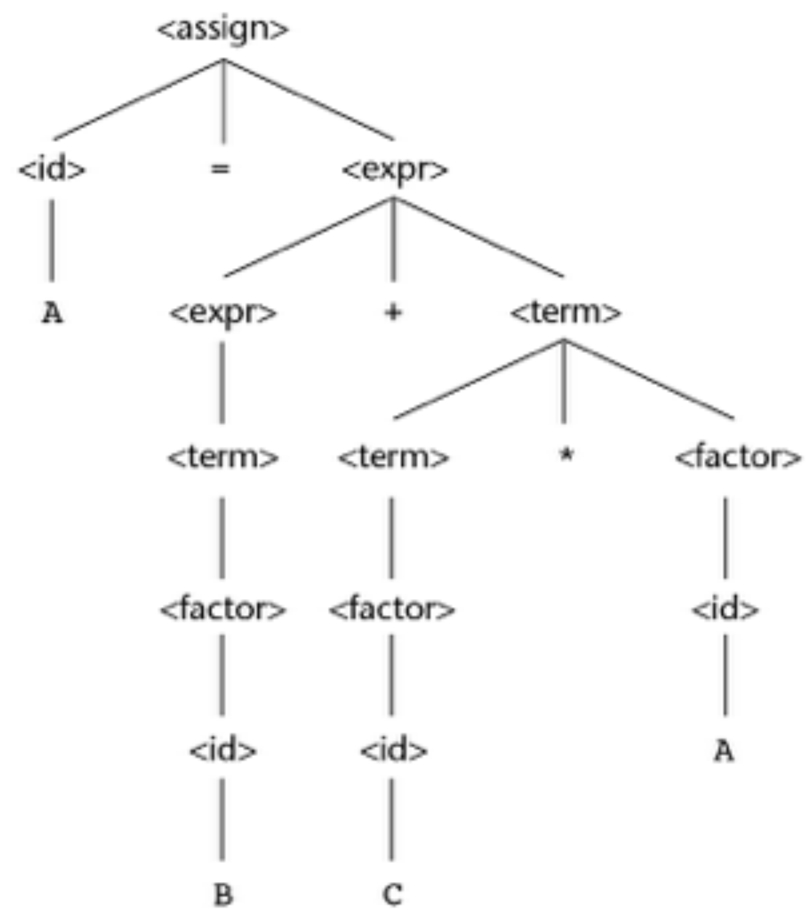
- Ambiguity can be resolved by indicating precedence levels of the operators.

```
Unambiguous Simple Assignment

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

Parse Tree with Unambiguous Grammar

```
Unambiguous Simple Assignment

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

```
A = B + C * A
<assign> => <id> = <expr>
    => A = <expr>
    => A = <expr> + <term>
    => A = <term> + <term>
    => A = <factor> + <term>
    => A = <id> + <term>
    => A = B + <term>
    => A = B + <term> * <factor>
    => A = B + <factor> * <factor>
    => A = B + <id> * <factor>
    => A = B + C * <factor>
    => A = B + C * <id>
    => A = B + C * A
```

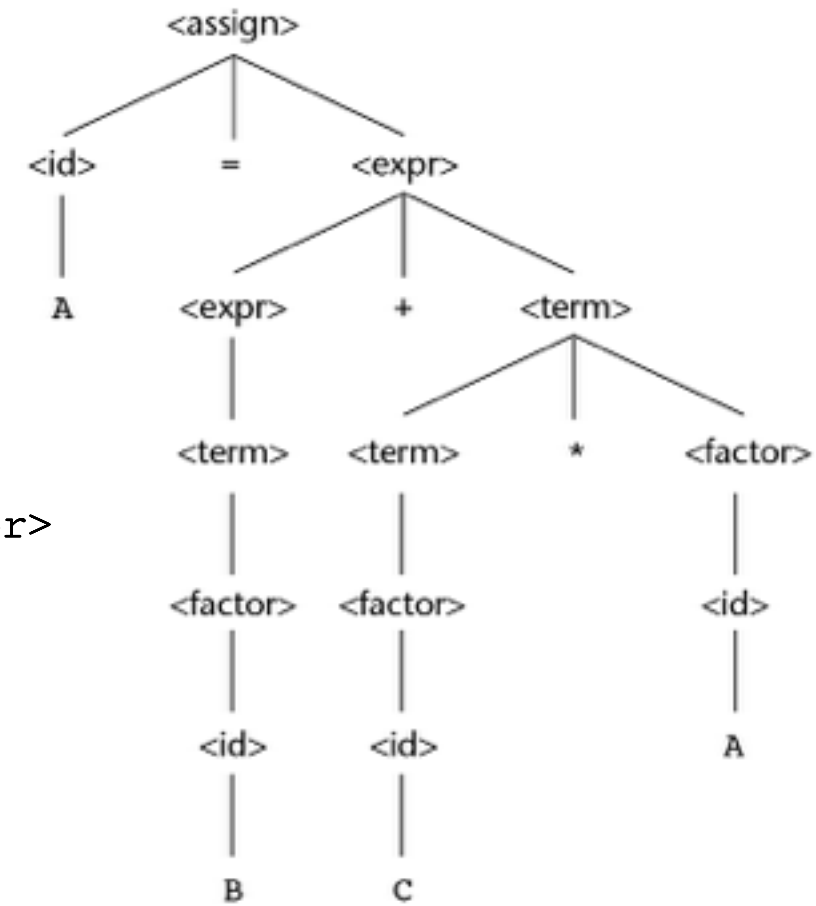


← Leftmost derivation

Parse Tree from Rightmost Derivation

```
Unambiguous Simple Assignment

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

```
A = B + C * A
<assign> => <id> = <expr>
    => <id> = <expr> + <term>
    => <id> = <expr> + <term> * <factor>
    => <id> = <expr> + <term> * <id>
    => <id> = <expr> + <term> * A
    => <id> = <expr> + <factor> * A
    => <id> = <expr> + <id> * A
    => <id> = <expr> + C * A
    => <id> = <term> + C * A
    => <id> = <factor> + C * A
    => <id> = <id> + C * A
    => <id> = B + C * A
    => A = B + C * A
```



Note: the parse trees are same.

Associativity of Operators

- Operators with the same precedence
  - Example:

```
A = A + B + C
  (A + B) + C == A + (B + C)
```

  - Can be problematic in floating-point operations.
    - Example: `(-1 + 1) + 1e-200 != -1 + (1 + 1e-200)`

  - Left- and right-recursion:

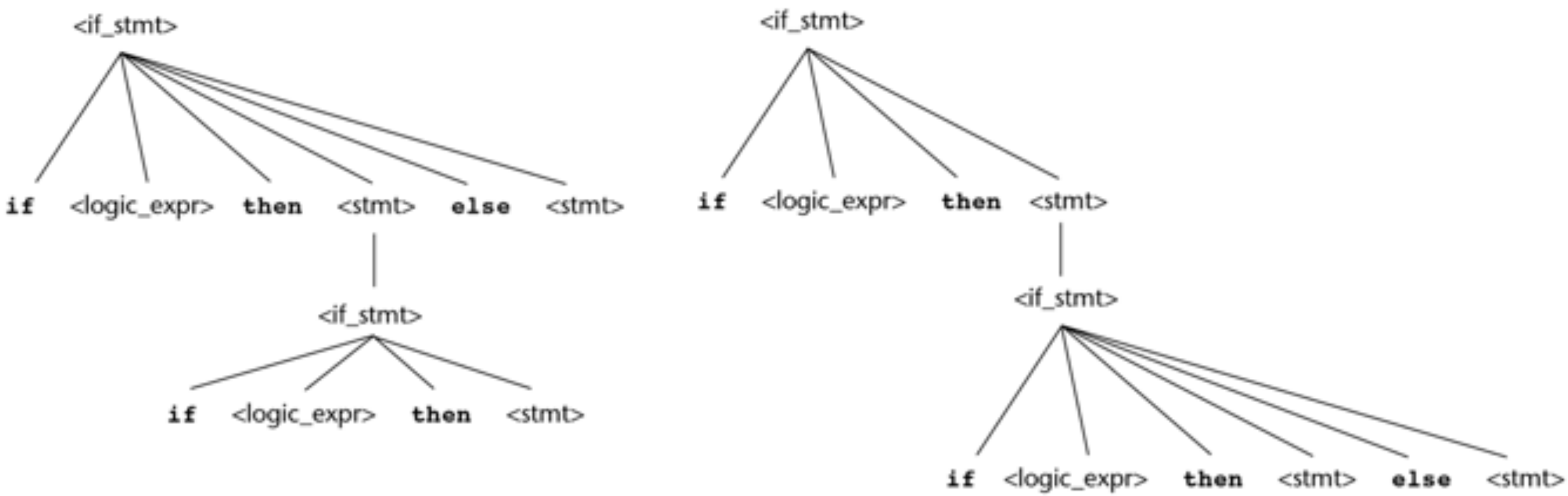
```
<expr> → <expr> + <term>  vs.
<factor> → <exp> ** <factor>
```

Unambiguous Grammar for if-then-else

- Dangling else:

```
if <logic_expr> then if <logic_expr> then <stmt> else <stmt>
```

Ambiguous if-then-else
`<stmt> → <if_stmt> | …`
`<if_stmt> → if <logic_expr> then <stmt>`
`          | if <logic_expr> then <stmt> else <stmt>`



Unambiguous Grammar for if-then-else

- Match `else` to the nearest `then`.

```
Unambiguous if-then-else

<stmt> → <matched> | <unmatched> | …
<matched> → if <logic_expr> then <matched> else <matched>
          | <any_non_if_statement>
<unmatched> → if <logic_expr> then <stmt>
            | if <logic_expr> then <matched> else <unmatched>
```

```
if <logic_expr> then if <logic_expr> then <stmt> else <stmt>
```

Extended BNF

- [ ] : optional parts (0 or 1)
- (| |) : alternative parts of RHSs.
- { } : repetitions (0 or more).   { }+ represents 1 or more.

<table>
<tr><th>BNF</th><th>Extended BNF</th></tr>
<tr><td><if_stmt> → if (<expr>) <stmt>
| if (<expr>) <stmt>
else <stmt></td><td><if_stmt> → if (<expr>) <stmt>
[else <stmt>]</td></tr>
<tr><td><expr> → <expr> + <term>
| <expr> - <term>
| <term></td><td><expr> → <term> {(+|-) <term>}</td></tr>
<tr><td><compound> → begin <stmts> end
<stmts> → <stmts> <stmt>
| <stmt></td><td><compound> → begin {<stmt>}+ end</td></tr>
</table>

Extended BNF

- Recent Variations:
  - Alternative RHSs are put on separate lines (intead of using |).
  - Use of a colon(:) instead of →.
  - Use of $_{\texttt{opt}}$ for optional parts.

`ConstructorDeclare → SimpleName ( FormalParamList`$_{\texttt{opt}}$ `)`

  - Use of `one of` for choices.

`AssignmentOperator → one of = *= /= %= += -= <<= >>= …`

Grammars and Recognizers

- Recognizer for a given context-free grammar can be constructed algorithmically.
  - yacc (yet another compiler compiler).

Static Semantics

- Nothing to do with ‘meaning’.
- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages.
- Categories of constructs that are trouble:
  - Context-free, but cumbersome.
    (e.g., types of operands in expressions)
  - Non-context-free.
    (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes.
  - Attributes (to symbols).
  - Attribute computation functions (semantic functions)   (to rules)
  - Predicate functions
- Primary value of AGs:
  - Static semantics specification.
  - Compiler design (static semantics checking).

Attribute Grammars: Definition

- Def: An **attribute grammar** is a context-free grammar with the following additions:
  - For each grammar symbol `x` there is a set `A(X)` of attribute values.
    - `A(X) = S(X) ∪ I(X);` synthesized and inherited attributes.
  - Each rule $X_0 \rightarrow X_1 \ldots X_n$ has a set of functions that define certain attributes of the nonterminals in the rule.
    - $S(X_0) = f(A(X_1), \ldots, A(X_n))$.
    - $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $1 <= j <= n$.
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency.
    - Boolean expression on $\{A(X_0), \ldots, A(X_n)\}$.
    - False predicate function value: violation of the syntax or static semantic rules.

Attribute Grammars: Definition

- **Intrinsic attributes** on the leaf nodes.
  - e.g. type of a variable comes from the symbol table, which is set from an earlier declaration statement.

```
int i;
…
i = i + 10;
```

- The parse tree is said to be **fully attributed** if all the attribute values are computed.

Attribute Grammars: An Example

An attribute grammar example	
Syntax rule:	<proc_def> → **procedure** <proc_name>[1] <proc_body> **end** <proc_name>[2] ;
Predicate:	<proc_name>[1].string == <proc_name>[2].string

Example 1:

```
procedure MyFunction
   ...
end MyFunction;
```

Example 2:

```
procedure MyFunction1
   ...
end MyFunction2;
```

Attribute Grammars: Simple Assignment

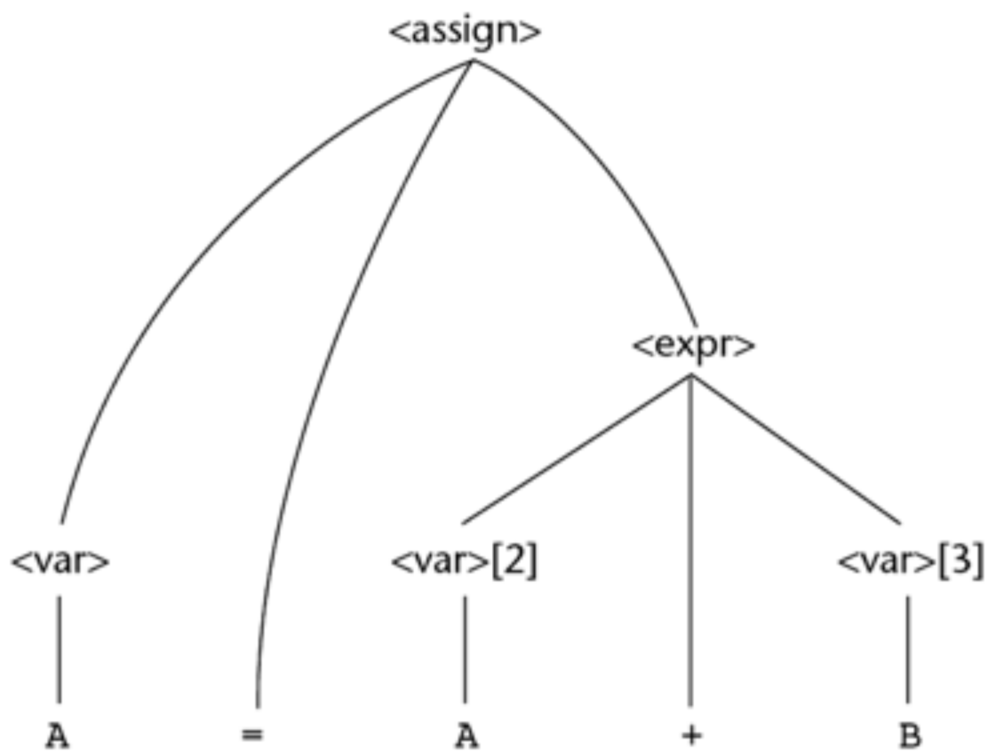
An attribute grammar for simple assignment	
Syntax rule:	<assign> → <var> = <expr> <expr> → <var> + <var> | <var> <var> → A | B | C

- Example attribute grammar for type checking:
  - actual_type: a synthesized attribute for <var> and <expr>.
  - expected_type: an inherited attribute for <expr>.

Attribute Grammars: Simple Assignment

Figure 3.6

A parse tree for
`A = A + B`



Attribute Grammars: Simple Assignment

An attribute grammar for simple assignment	
Syntax rule: Semantic rule:	`<assign> → <var> = <expr>` `<expr>.expected_type ← <var>.actual_type`

Syntax rule:
Semantic rule:
 	
Predicate:
	`<expr> → <var>[2] + <var>[3]`
	`<expr>.actual_type ←`
	`    if (var[2].actual_type == int) and`
	`       (var[3].actual_type == int)`
	`    then int`
	`    else real endif`
	`<expr>.actual_type == <expr>.expected_type`

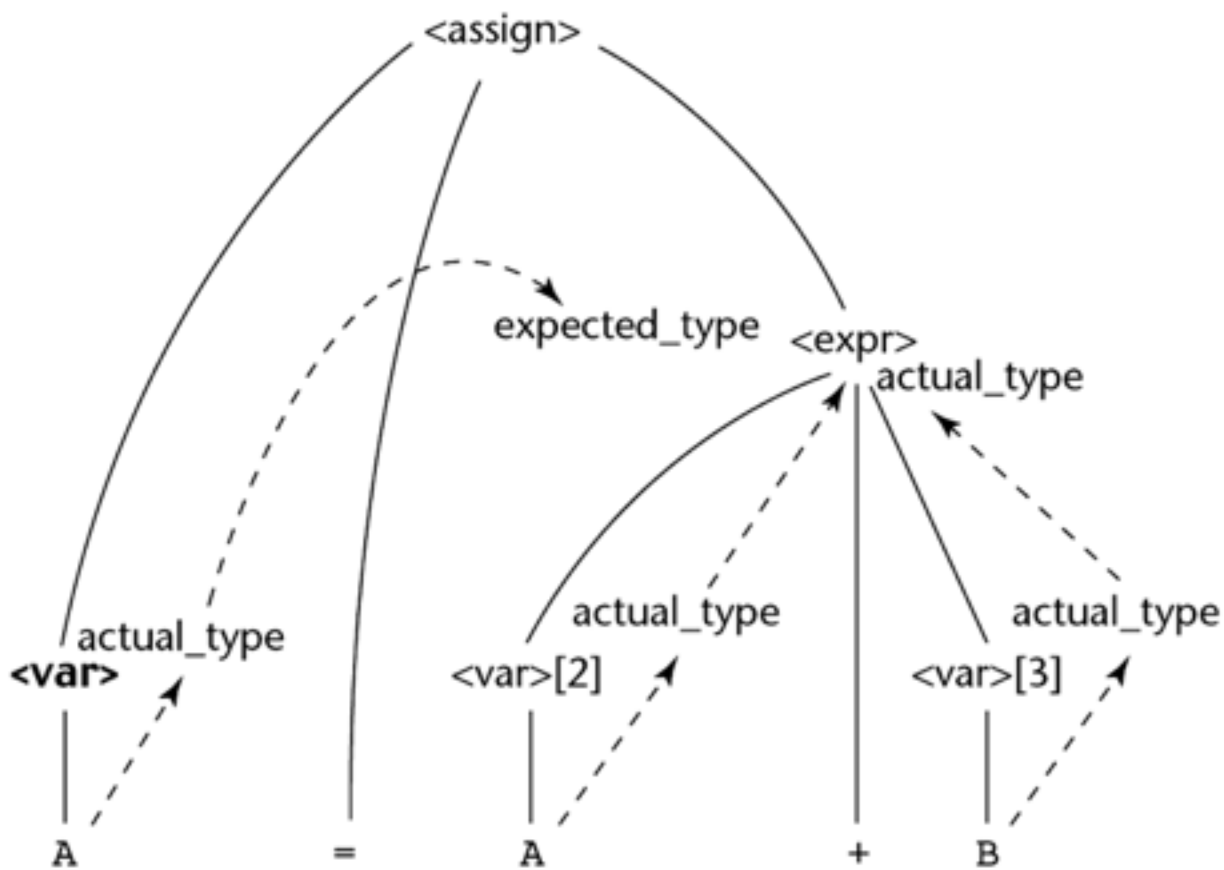
Syntax rule:
Semantic rule:
Predicate:
	`<expr> → <var>`
	`<expr>.actual_type ← <var>.actual_type`
	`<expr>.actual_type == <expr>.expected_type`

Syntax rule: Semantic rule:	`<var> → A | B | C` `<var>.actual_type ← look_up(<var>.string)`

Attribute Grammars (continued)

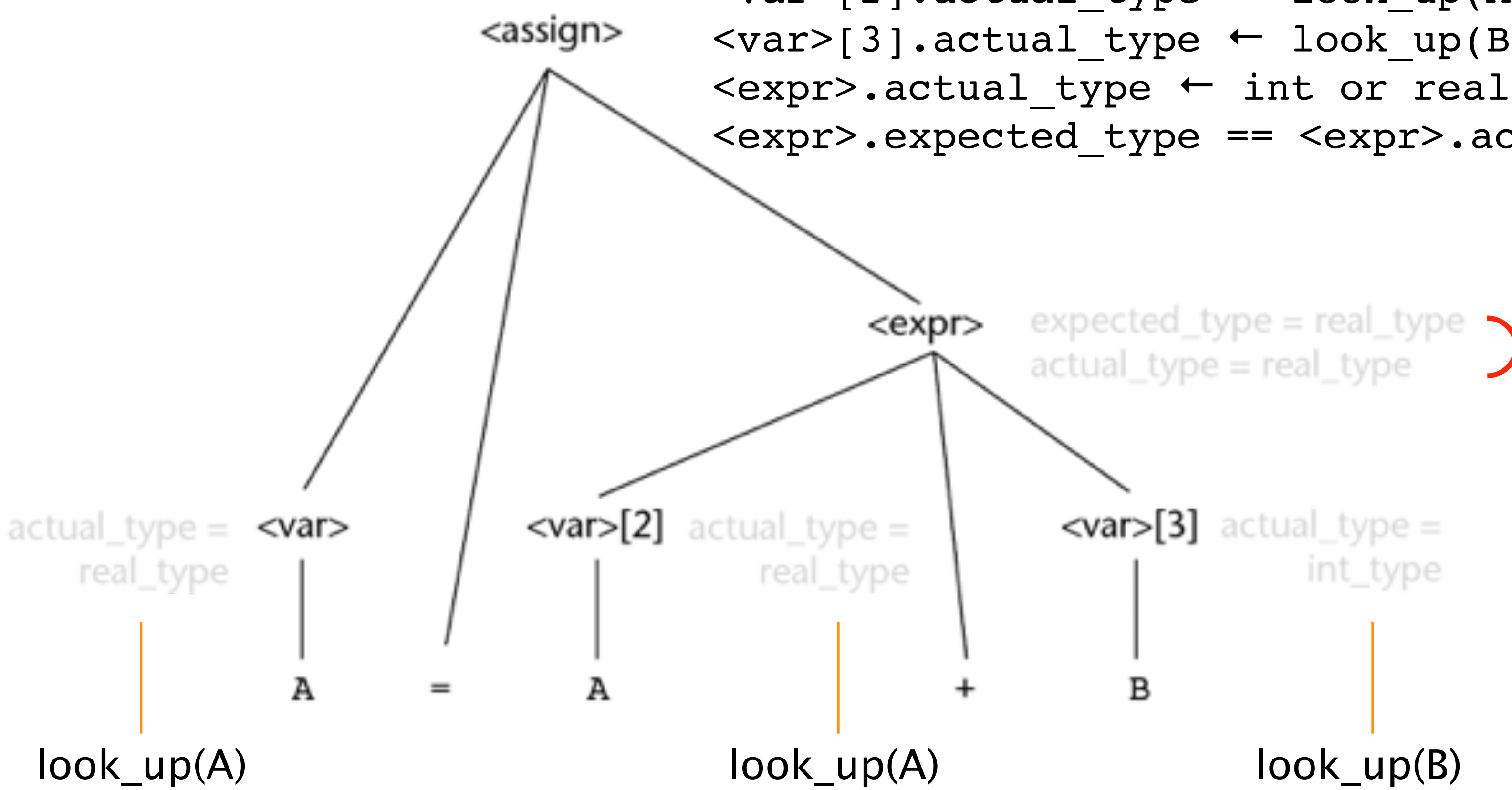
- How are attribute values computed?
  - Inherited attributes: decorated in top-down order.
  - Synthesized attributes: decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Attribute Grammars: Simple Assignment



Attribute Grammars: Simple Assignment

```
<var>.actual_type ← look_up(A)
<expr>.expected_type ← <var>.actual_type
<var>[2].actual_type ← look_up(A)
<var>[3].actual_type ← look_up(B)
<expr>.actual_type ← int or real
<expr>.expected_type == <expr>.actual_type
```



What happens if `A.actual_type` is `int_type` and `B.actual_type` is `real_type`?

(Dynamic) Semantics

- There is no single widely acceptable notation or formalism for describing semantics.
- Needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean.
  - Compiler writers must know exactly what language constructs do.
  - Correctness proofs would be possible.
  - Compiler generators would be possible.
  - Designers could detect ambiguities and inconsistencies.

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by 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.
  - **Natural** operational semantics : the final result of a program.
  - **Structural** operational semantics : the precise meaning of a program.
- To use operational semantics for a high-level language:
  - Design an appropriate intermediate language - clarity.
  - Virtual machine for the intermediate language is needed in natural operational semantics.

Operational Semantics

- Example:

C Statement	Meaning

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

- Uses of operational semantics:
  - Language manuals and textbooks.
  - Teaching programming languages.
- Evaluation
  - Good if used informally (language manuals, etc.).
  - Extremely complex if used formally (e.g.,VDL).

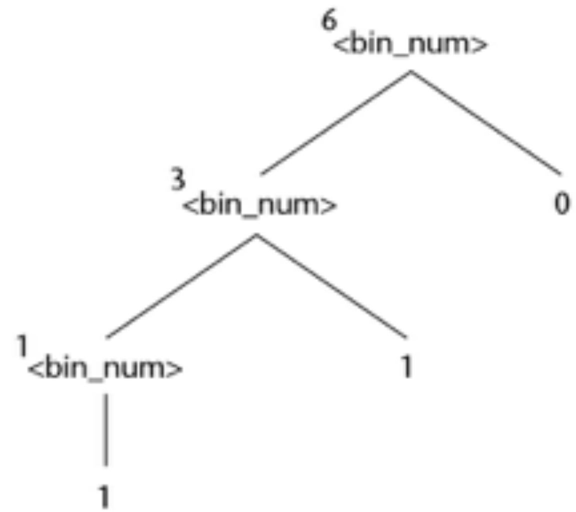
Denotational Semantics

- The most rigorous and widely known method.
  - Based on recursive function theory.
  - Originally developed by Scott and Strachey (1970).
  - The meaning of language constructs are defined by only the values of the program's variables.
  - Operational semantics: simpler language vs. mathematical objects.
- 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.
    - Syntactic domain → semantic domain.

Denotational Semantics: Examples

- Binary numbers.

```
Binary numbers

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



  - E.g. Parse tree for ‘110’:
    - Syntactic domain: all string representations of binary numbers.
    - Semantic domain: non-negative decimal numbers.
    - Semantic functions:

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

Denotational Semantics: Examples

- Decimal numbers.

```
Decimal numbers

<dec_num> → 0 | 1 | 2 | 3 … | 9
          | <dec_num> 0
          | <dec_num> 1
          ...
          | <dec_num> 9
```

  - E.g. '352'
    - Syntactic domain: all string representations of decimal numbers.
    - Semantic domain: non-negative decimal numbers.
    - Semantic functions:

$M_{dec}('0') = 0,\ M_{dec}('1') = 1,\ M_{dec}('2') = 2,\ ...,$
$M_{dec}('9') = 9$
$M_{dec}(\text{<dec\_num>}\ '0') = 10 * M_{dec}(\text{<dec\_num>})$
$M_{dec}(\text{<dec\_num>}\ '1') = 10 * M_{dec}(\text{<dec\_num>}) + 1$
...
$M_{dec}(\text{<dec\_num>}\ '9') = 10 * M_{dec}(\text{<dec\_num>}) + 9$

State of Program

- The state of a program is the values of all its current variables.

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

  - i_j : the name of a variable, v_j : the current value of the variable.
  - `undef` represents the value is currently undefined.
  - $\mathrm{VARMAP}(i_j, s) \rightarrow v_j$:

    map from states to states (or to values for expressions).
  - Example:

```
s = { <'a',1>, <'b',2>, <'c',3> }
VARMAP('a', s) → 1
VARMAP('i', s) → undef
```

Expressions

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

- Map `expressions` onto `Z ∪ {error}`.
  - We assume expressions have no side effects.
  - We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be a variable or a decimal number.

Expressions

```
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) == error OR
          Me(<binary_expr>.<right_expr>, s) == error)
      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

- Maps state sets to state sets U {error}

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

Example: a = b + c, { <a,0>, <b,1>, <c,2> }

    Ma(a = b + c, { <a,0>, <b,1>, <c,2> })
           Me(b + c, { <a,0>, <b,1>, <c,2> }) = 3
    → { <a,3>, <b,1>, <c,2> }
```

Logical Pretest Loops

- Maps state sets to state sets U {error}
  - M_b maps Boolean expressions to Boolean values (or error).
  - M_{sl} maps statement lists and states to states (or error).

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

Logical Pretest Loops

- The loop has been converted from iteration to recursion.
  - The recursive control is mathematically defined by other recursive state mapping functions.
  - Recursion is easier to describe with mathematical rigor.

Logical Pretest Loop Example

```
M_l(while B do L, s) Δ=
  if (M_b(B, s) == undef)
  then error
  else if (M_b(B, s) == false)
       then s
       else if (M_sl(L, s) == error)
            then error
            else M_l(while B do L, M_sl(L, s))

Example:  while i > 0 do sum = sum + i; i--;

  M_l(while i > 0 do a = a + i; --i, { <a,0>, <i,3> })
→ M_l(while i > 0 do a = a + i; --i, { <a,3>, <i,2> })
→ M_l(while i > 0 do a = a + i; --i, { <a,5>, <i,1> })
→ M_l(while i > 0 do a = a + i; --i, { <a,6>, <i,0> })
→ { <a,6>, <i,0> }
```

Denotational Semantics

- Evaluation of denotational semantics:
  - Can be used to prove the correctness of programs.
  - Provides a rigorous way to think about programs.
  - Can be an aid to language design.
    - Revise the design if it is too complex and difficult.
  - Has been used in compiler generation systems.
  - Because of its complexity, it is of little use to language users.

Axiomatic Semantics

- Axiomatic semantics:
  - Based on mathematical logic (predicate calculus).
    - What can be proven about the program?
  - Original purpose: formal program verification.
    - Also used for program semantics specification.
  - Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions).

Axiomatic Semantics

- The logic expressions are called assertions (predicates).
  - **Precondition**: assertion before a statement, stating the relationships and constraints among variables that are true at that point in execution.
  - **Postcondition**: assertion following a statement.
  - The precondition of a statement is the postcondition of the previous statement.
  - Preconditions for the statements are computed from given postconditions.

- Pre-, post form: `{P} statement {Q}`
  - Example: `{ x > 0 }  sum = 2 * x + 1  { sum > 1 }`
  - What would be the possible precondition for the postcondition?

Weakest Precondition

- A **weakest precondition** is the least restrictive precondition that will guarantee the postcondition.
  - Example: `{ ?? }  sum = 2 * x + 1  { sum > 1 }`

    `{ x > 10 }, { x > 50 }, { x > 0 }, …`
- Program proof process:
  - 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.

Inference Rule

- Inference rule:

$$\frac{S_1,\ S_2,\ \ldots,\ S_n}{S}\quad \begin{array}{l}\leftarrow \text{antecedent} \\ \leftarrow \text{consequent}\end{array}$$

  - If S_1, S_2, …, and S_n are true, then the truth of S can be inferred.
  - **Axiom**: a logical statement that is assumed to be true.
  - Either an axiom or an inference rule must be available for each kind of statement in the language.

Assignment Statements

- Axiom for assignment statements:

  $\{Q_{x \to E}\}$ x = E {Q}

  - $Q_{x \to E}$: Q with all instances of x replaced by E.
  - Examples:

```
    a = b / 2 - 1  { a < 10 }
      Weakest precondition: b < 22

    x = 2 * y – 3  { x > 25 }
      Weakest precondition: y > 14

    x = x + y – 3  { x > 10 }
      Weakest precondition: y > 13 - x
```

Assignment Statements

- Consider the following logical statement.

```
{ x > 3 }  x = x – 3  { x > 0 }
```

- Can we prove this statement?

```
x = x – 3  { x > 0 }
Weakest precondition: x > 3
```

  - This is same as the precondition – proven!

Assignment Statements

- Another example:

```
{ x > 5 }  x = x – 3  { x > 0 }
Weakest precondition: x > 3
```

  - However { x > 5 } implies { x > 3 }.

- Rule of consequence:

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

  - Precondition can always be strengthened.
  - Postcondition can always be weakened.

$$\frac{\{x>3\}\ x = x - 3\ \{x>0\},\ \{x>5\} \Rightarrow \{x>3\},\quad \{x>0\} \Rightarrow \{x>0\}}{\{\ x > 5\ \}\ x = x - 3\ \{\ x > 0\ \}}$$

Sequences

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

```
{ P1 }  S1  { P2 }
{ P2 }  S2  { P3 }
```

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

  - Sequences of assignments:

    `x1 = E1; x2 = E2`

    $\{\ P_{x2 \to E2}\ \}\quad x2 = E2\quad \{\ P\ \}$

    $\{\ (P_{x2 \to E2})_{x1 \to E1}\ \}\quad x1 = E1\quad \{\ P_{x2 \to E2}\ \}$

  - Weakest precondition with postcondition `P` is $\{(P_{x2 \to E2})_{x1 \to E1}\}$.
  - Example:

```
        y = 3 * x + 1; x = y + 3;  { x < 10 }
{ y < 7 }  x = y + 3;  { x < 10 }
{ x < 2 }  y = 3 * x + 1; x = y + 3;  { x < 10 }
```

Selection

- An inference rule for selection statements:

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

  - Example:

    if x > 0 **then** y = y – 1 **else** y = y + 1 { y > 0 }

    - { y > 1 } is the w.p. of y = y – 1 { y > 0 }
    - { y > -1 } is the w.p. of y = y + 1 { y > 0 }
    - Therefore,

      { y > 1 } **if** x>0 **then** y=y–1 **else** y=y+1 { y > 0 }

Logical Pretest Loops

- An inference rule for logical pretest loops:

  `{P}` **`while`** `B` **`do`** `S` **`end`** `{Q}`

$$\frac{\{\mathtt{I\ and\ B}\}\ \mathtt{S}\ \{\mathtt{I}\}}{\{\mathtt{I}\}\ \textbf{while}\ \mathtt{B}\ \textbf{do}\ \mathtt{S}\ \textbf{end}\ \{\mathtt{I\ and\ (not\ B)}\}}\ \circledast$$

  - where I is the loop invariant.

- How to find the loop invariant?
  - `P => I` ① : the loop invariant must be true initially.
  - `{I and B} S {I}` ② : I is not changed by executing the loop body.
  - `(I and (not B)) => Q` ③ : if I is true and B is false, Q is implied.
  - The loop terminates. ④ : this can be difficult to prove.

$$\frac{\dfrac{\{\mathtt{I\ and\ B}\}\ \mathtt{S}\ \{\mathtt{I}\}^{②}}{\{\mathtt{I}\}\ \textbf{while}\ \mathtt{B}\ \textbf{do}\ \mathtt{S}\ \textbf{end}\ \{\mathtt{I\ and\ (not\ B)}\}}\ \circledast\ ,\ \mathtt{P => I}^{①},\ \{\mathtt{I\ and\ (not\ B)}\}\ \mathtt{=> Q}^{③}}{\{\mathtt{P}\}\ \textbf{while}\ \mathtt{B}\ \textbf{do}\ \mathtt{S}\ \textbf{end}\ \{\mathtt{Q}\}}$$

Logical Pretest Loop: Example

- Weakest precondition predicate transformer:

```
wp(statement, postcondition) = precondition
```

  - It takes a predicate and returns the weakest precondition of the statement which is another predicate.

  - Example:

```
while y <> x do y = y + 1 end  { y == x }
  begin:   { y == x }
  1 iter:  wp(y = y + 1, { y == x }) = { y == x–1 }
  2 iter:  wp(y = y + 1, { y == x-1 }) = { y == x–2 }
  3 iter:  wp(y = y + 1, { y == x-2 }) = { y == x–3 }
           ...
           { y <= x }
{ y <= x } while y <> x do y = y + 1 end { y == x }
```

Logical Pretest Loop

- `{P}` **`while`** `B` **`do`** `S` **`end`** `{Q}`

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

- Does the invariant `I = { y <= x }` satisfy the four criteria?
  - `P => I` : true since `P == I`.
  - `{I` **`and`** `B} S {I}`
    - `{I` **`and`** `B}` : `{ y <= x` **`and`** `y <> x }`
    - Using assignment axiom: `{ y<x }` is the w.p. of `y=y+1` `{ y<=x }`
    - True since `{ y <= x` **`and`** `y <> x } => { y < x }`
  - `(I` **`and`** `(`**`not`** `B)) => Q`
    - `{ (y <= x)` **`and not`** `(y <> x) } => { y == x }`
    - `{ (y <= x)` **`and`** `(y == x) } => { y == x }`
  - The loop terminates.
    - For integer x and y, the loop terminates.

Logical Pretest Loop: Another Example

- **while** s > 1 **do** s = s / 2 **end** { s == 1 }

  ```
  { s == 1 }
  wp(s = s / 2, { s == 1 }) = { s / 2 == 1 }, or { s == 2 }
  wp(s = s / 2, { s == 2 }) = { s / 2 == 2 }, or { s == 4 }
  ...
  { s == 2^i, i>=0 } while s > 1 do s = s / 2 end { s == 1 }
  ```

  - This is not the weakest precondition (for integer operations).

- { s >= 1 } **while** s > 1 **do** s = s / 2 **end** { s == 1 }

Loop Invariant

```
{P} while B do S end {Q}
```

```
P => I
{I and B} S {I}
(I and (not B)) => Q
The loop terminates.
```

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

- The loop invariant `I`:
  - A weakened version of the loop postcondition, and also a precondition.
  - Weak enough to be satisfied prior to the beginning of the loop.
  - Strong enough to force the truth of the postcondition, when combined with the loop exit condition.
- Loop termination is hard to prove.
  - Total correctness vs. partial correctness.

Program Proofs: Example 1

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

- Use assignment axiom.

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

{ y == B and t == A }
          x = y; y = t;  { x == B and y == A }

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

Program Proofs: Example 2

```
{ n >= 0 }
    count = n;  fact = 1;
    while count <> 0 do
           fact = fact * count;
           count = count – 1;
    end
{ fact == n! }
```

- Find the loop invariant (using some ingenuity!):

```
    fact == (count+1)*(count+2)*…*(n-1)*n == n!/count!
```

```
I = (fact == n!/count!) and (count >= 0)
```

- Check the four criteria for the loop.

Program Proofs: Example 2

```
Program proofs – example 2

{ n >= 0 }
    count = n;   fact = 1;
    while count <> 0 do
        fact = fact * count;
        count = count – 1;
    end
{ fact == n! }

I = (fact == n!/count!) and (count >= 0)
```

- Loop body:
  - **P => I**

    It holds since P == I.

  - **{I and B} S {I}**

```
{I and B} :
  {(fact == n!/count!) and (count >= 0) and (count <> 0)}
{I} : {(fact == n!/count!) and (count >= 0)}

{Pc} count = count – 1; {I}
    Pc = (fact == n!/(count-1)!) and (count >= 1)
{P'} fact = fact * count; {Pc}
    P' = (fact == n!/count!) and (count >= 1)
It holds since I and B => P'.
```

Program Proofs: Example 2

```
Program proofs – example 2

{ n >= 0 }
   count = n;   fact = 1;
   while count <> 0 do
        fact = fact * count;
        count = count – 1;
   end
{ fact == n! }

I = (fact == n!/count!) and (count >= 0)
```

- Loop body (cont.):
  - **(I and (not B)) => Q**

```
((fact == n!/count!) and (count>=0) and not (count<>0))
  => ((fact == n!/count!) and (count==0))  **note 0! = 1
  => fact == n!
```

- Entire program:

```
{Pi2} fact = 1; {P}
{Pi2} fact = 1; {(fact == n!/count!) and (count >= 0)}
  =>  Pi2 = ((1 == n!/count!) and (count >= 0))

{Pi1} count = n; {Pi2}
{Pi1} count = n; {(1 == n!/count!) and (count >= 0)}
  =>  Pi1 = (1 == n!/n! and (n >= 0))
          = (n >= 0)
```

Evaluation of Axiomatic Semantics

- Evaluation of axiomatic semantics:
  - Developing axioms or inference rules for all of the statements in a language is difficult.
  - It is a good tool for correctness proofs, and an 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.

Summary

- BNF and context-free grammars are equivalent meta-languages.
  - Well-suited for describing the syntax of programming languages.
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language.
- Three primary methods of semantics description
  - Operation, denotational, axiomatic.