

# Attribute Grammars

An attribute grammar is a context-free grammar that has been extended to provide context-sensitive information by appending attributes to some of its nonterminals.

Each distinct symbol in the grammar has associated with it a finite, possibly empty, set of attributes.

- Each attribute has a domain of possible values.
- An attribute may be assigned values from its domain during parsing.
- Attributes can be evaluated in assignments or conditions.

## Two Classes of Attributes

- **Synthesized attribute** An attribute that gets its values from the attributes attached to the children of its nonterminal.
- **Inherited attribute** An attribute that gets its values from the attributes attached to the parent (or siblings) of its nonterminal.

# An Example

Recall the context-sensitive language from Chapter 1:

$$L = \{ a^n b^n c^n \mid n \geq 1 \}$$

Important result from computation theory:

No context-free grammar generates L.

An attempt:

$\langle \text{string} \rangle ::= \langle \text{a seq} \rangle \langle \text{b seq} \rangle \langle \text{c seq} \rangle$

$\langle \text{a seq} \rangle ::= \mathbf{a} \mid \langle \text{a seq} \rangle \mathbf{a}$

$\langle \text{b seq} \rangle ::= \mathbf{b} \mid \langle \text{b seq} \rangle \mathbf{b}$

$\langle \text{c seq} \rangle ::= \mathbf{c} \mid \langle \text{c seq} \rangle \mathbf{c}$

This context-free grammar generates the language

$$a^+ b^+ c^+ = \{ a^k b^m c^n \mid k \geq 1, m \geq 1, n \geq 1 \}$$

# Using an Attribute Grammar

Attach a synthesized attribute, *Size*, to each of the nonterminals:  $\langle \text{a seq} \rangle$ ,  $\langle \text{b seq} \rangle$ , and  $\langle \text{c seq} \rangle$ .

Domain of *Size* = Positive Integers

Impose a condition on the first production.

$\langle \text{string} \rangle ::= \langle \text{a seq} \rangle \langle \text{b seq} \rangle \langle \text{c seq} \rangle$

### condition

$$Size(\langle \text{a seq} \rangle) = Size(\langle \text{b seq} \rangle) = Size(\langle \text{c seq} \rangle)$$

$\langle \text{a seq} \rangle ::= \mathbf{a}$

$$Size(\langle \text{a seq} \rangle) \leftarrow 1$$

$\mid \langle \text{a seq} \rangle_2 \mathbf{a}$

$$Size(\langle \text{a seq} \rangle) \leftarrow Size(\langle \text{a seq} \rangle_2) + 1$$

$\langle \text{b seq} \rangle ::= \mathbf{b}$

$$Size(\langle \text{b seq} \rangle) \leftarrow 1$$

$\mid \langle \text{b seq} \rangle_2 \mathbf{b}$

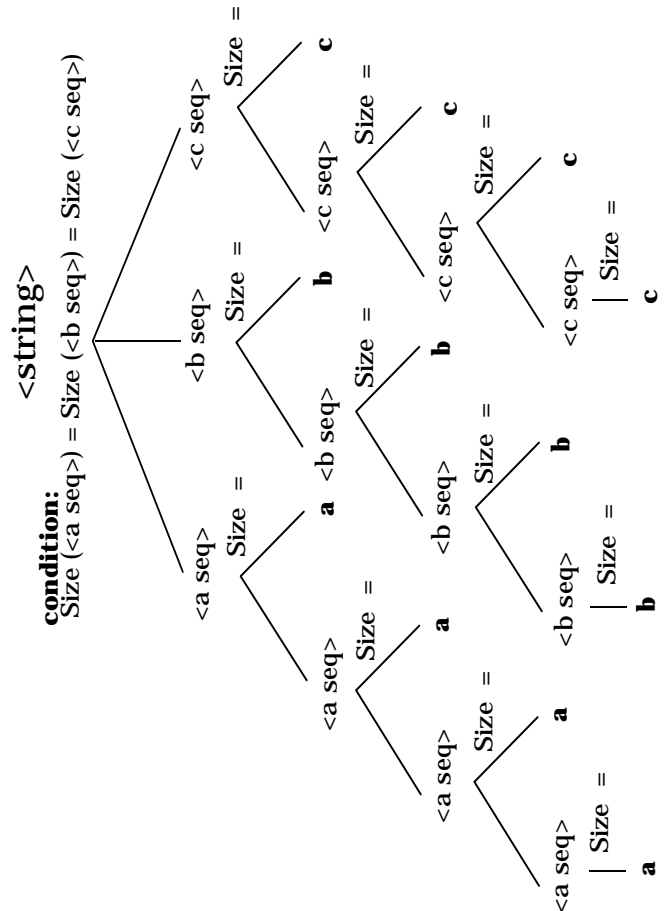
$$Size(\langle \text{b seq} \rangle) \leftarrow Size(\langle \text{b seq} \rangle_2) + 1$$

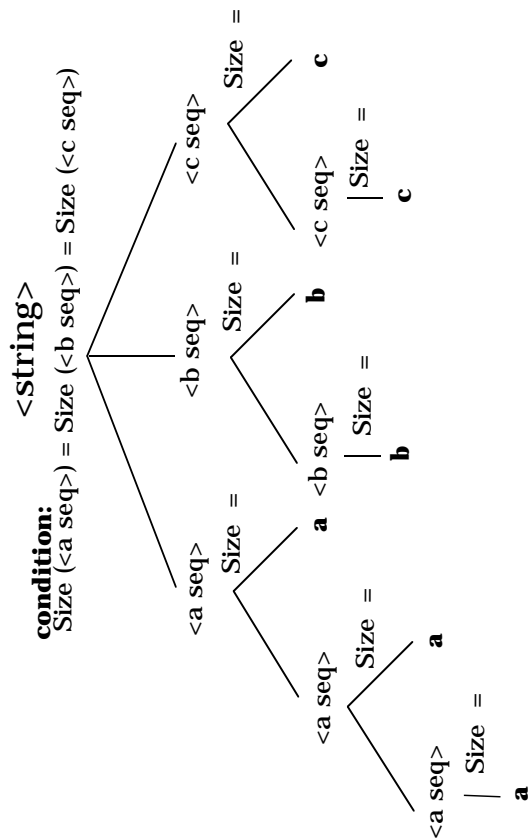
$\langle \text{c seq} \rangle ::= \mathbf{c}$

$$Size(\langle \text{c seq} \rangle) \leftarrow 1$$

$\mid \langle \text{c seq} \rangle_2 \mathbf{c}$

$$Size(\langle \text{c seq} \rangle) \leftarrow Size(\langle \text{c seq} \rangle_2) + 1$$





## Using an Inherited Attribute

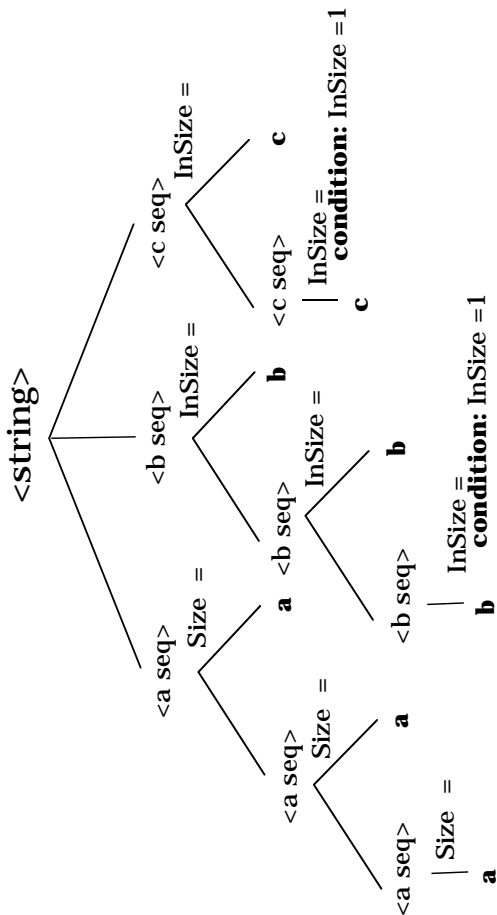
Attach a synthesized attribute *Size* to  $\langle a \text{ seq} \rangle$  and inherited attributes *InSize* to  $\langle b \text{ seq} \rangle$  and  $\langle c \text{ seq} \rangle$ .

$\langle \text{string} \rangle ::= \langle a \text{ seq} \rangle \langle b \text{ seq} \rangle \langle c \text{ seq} \rangle$   
 $InSize(\langle b \text{ seq} \rangle) \leftarrow Size(\langle a \text{ seq} \rangle)$   
 $InSize(\langle c \text{ seq} \rangle) \leftarrow Size(\langle a \text{ seq} \rangle)$

$\langle a \text{ seq} \rangle ::= a$   
 $Size(\langle a \text{ seq} \rangle) \leftarrow 1$   
 $| \langle a \text{ seq} \rangle_2 a$   
 $Size(\langle a \text{ seq} \rangle) \leftarrow Size(\langle a \text{ seq} \rangle_2) + 1$

$\langle b \text{ seq} \rangle ::= b$   
**condition:**  
 $InSize(\langle b \text{ seq} \rangle) = 1$   
 $| \langle b \text{ seq} \rangle_2 b$   
 $InSize(\langle b \text{ seq} \rangle_2) \leftarrow InSize(\langle b \text{ seq} \rangle) - 1$

$\langle c \text{ seq} \rangle ::= c$   
**condition:**  
 $InSize(\langle c \text{ seq} \rangle) = 1$   
 $| \langle c \text{ seq} \rangle_2 c$   
 $InSize(\langle c \text{ seq} \rangle_2) \leftarrow InSize(\langle c \text{ seq} \rangle) - 1$



## Definition

An **attribute grammar** is a context-free grammar augmented with attributes, semantic rules, and conditions.

Let  $G = \langle N, \Sigma, P, S \rangle$  be a context-free grammar. Write a production  $p \in P$  in the form:

$$X_0 ::= X_1 X_2 \dots X_{n_p} \text{ where } n_p \geq 1, X_0 \in N, \\ \text{and } X_k \in N \cup \Sigma \text{ for } 1 \leq k \leq n_p.$$

A **derivation tree** for a sentence in a context-free language has the properties:

- Each of its leaf nodes is labeled with a symbol from  $\Sigma$ , and
- Each interior node  $t$  corresponds to a production  $p \in P$  such that  $t$  is labeled with  $X_0$  and  $t$  has  $n_p$  children labeled with  $X_1, X_2, \dots, X_{n_p}$  in left-to-right order.

For each syntactic category  $X \in N$  in the grammar, there are two finite disjoint sets:

$I(X)$  of **inherited attributes** and

$S(X)$  of **synthesized attributes**

$I(S) = \emptyset$  where  $S$  is the start symbol.

Let  $A(X) = I(X) \cup S(X)$  be the set of attributes of  $X$ .

Each attribute  $Atb \in A(X)$  takes a value from some semantic domain associated with that attribute

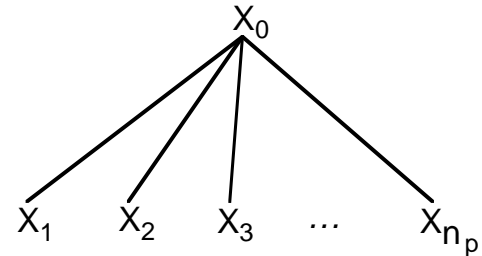
e.g. integers, strings of characters, structures of some type

The values of attributes are defined by **semantic functions** or **semantic rules** associated with the productions in  $P$ .

Consider again a production  $p \in P$  of the form:

$$X_0 ::= X_1 X_2 \dots X_{n_p}$$

Derivation Tree:



Each synthesized attribute  $Atb \in S(X_0)$  has its value defined in terms of the attributes in  $A(X_1) \cup A(X_2) \cup \dots \cup A(X_{n_p}) \cup I(X_0)$ .

Each inherited attribute  $Atb \in I(X_k)$  for  $1 \leq k \leq n_p$  has its value defined in terms of the attributes in  $A(X_0) \cup S(X_1) \cup \dots \cup S(X_{n_p})$ .

Each production may also have a set of conditions on the values of the attributes in

$$A(X_0) \cup A(X_1) \cup \dots \cup A(X_{n_p})$$

that further constrain an application of the production.

The parse of a sentence in the attribute grammar is satisfied

if only if

the context-free grammar is satisfied *and* all conditions in the attribute grammar are true.

The *semantics of a nonterminal* can be considered to be a distinguished attribute evaluated at the root node of the derivation tree of that nonterminal.

## Calculating Attribute Values for Binary Numerals

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	<i>Val</i>	---
<binary digits>	<i>Val</i>	<i>Pos</i>
<fraction digits>	<i>Val, Len</i>	---
<bit>	<i>Val</i>	<i>Pos</i>

### The Attribute Grammar

<binary numeral> ::= <binary digits> . <fraction digits>

$Val(\langle \text{binary numeral} \rangle) \leftarrow$

$Val(\langle \text{binary digits} \rangle) + Val(\langle \text{fraction digits} \rangle)$

$Pos(\langle \text{binary digits} \rangle) \leftarrow 0$

<binary digits> ::= <binary digits><sub>2</sub> <bit>

$Val(\langle \text{binary digits} \rangle) \leftarrow$

$Val(\langle \text{binary digits} \rangle_2) + Val(\langle \text{bit} \rangle)$

$Pos(\langle \text{binary digits} \rangle_2) \leftarrow Pos(\langle \text{binary digits} \rangle) + 1$

$Pos(\langle \text{bit} \rangle) \leftarrow Pos(\langle \text{binary digits} \rangle)$

$\langle \text{binary digits} \rangle ::= \langle \text{bit} \rangle$   
 $Val(\langle \text{binary digits} \rangle) \leftarrow Val(\langle \text{bit} \rangle)$   
 $Pos(\langle \text{bit} \rangle) \leftarrow Pos(\langle \text{binary digits} \rangle)$

$\langle \text{fraction digits} \rangle ::= \langle \text{fraction digits} \rangle_2 \langle \text{bit} \rangle$   
 $Val(\langle \text{fraction digits} \rangle) \leftarrow$   
 $Val(\langle \text{fraction digits} \rangle_2) + Val(\langle \text{bit} \rangle)$   
 $Len(\langle \text{fraction digits} \rangle) \leftarrow$   
 $Len(\langle \text{fraction digits} \rangle_2) + 1$   
 $Pos(\langle \text{bit} \rangle) \leftarrow - Len(\langle \text{fraction digits} \rangle)$

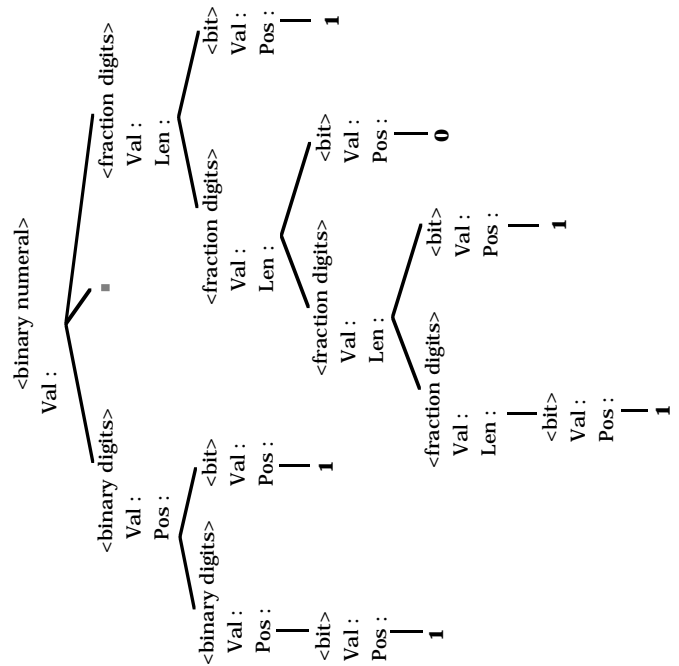
$\langle \text{fraction digits} \rangle ::= \langle \text{bit} \rangle$   
 $Val(\langle \text{fraction digits} \rangle) \leftarrow Val(\langle \text{bit} \rangle)$   
 $Len(\langle \text{fraction digits} \rangle) \leftarrow 1$   
 $Pos(\langle \text{bit} \rangle) \leftarrow - 1$

$\langle \text{bit} \rangle ::= 0$   
 $Val(\langle \text{bit} \rangle) \leftarrow 0$

$\langle \text{bit} \rangle ::= 1$   
 $Val(\langle \text{bit} \rangle) \leftarrow 2^{Pos(\langle \text{bit} \rangle)}$

Observe the order of evaluation of the attributes.

## Binary Numeral Semantics: 11.1101



## Checking Context Constraints in Wren

Augment the context-free grammar (concrete syntax) for Wren with attributes whose conditions check the context conditions of Wren.

1. The program name identifier may not be declared elsewhere in the program.
2. All identifiers that appear in a block must be declared in that block.
3. No identifier may be declared more than once in a block.
4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right must be of the same type.
5. An identifier occurring as an (integer) element must be integer variable.
6. An identifier occurring as a boolean element must be boolean variable.
7. An identifier occurring in a read command must be integer variable.

### Attribute

Type

### Value Types

{ integer, boolean, program, undefined }

**synthesized** for <type>

**inherited** for <expr>, <int expr>, <term>, <element>, <bool expr>, <bool term>, and <bool element>

Name

String of letters or digits

**synthesized** for <variable>, <identifier>, <letter>, and <digit>

Var-list

Sequence of Name values

**synthesized** for <variable list>

Symbol-table

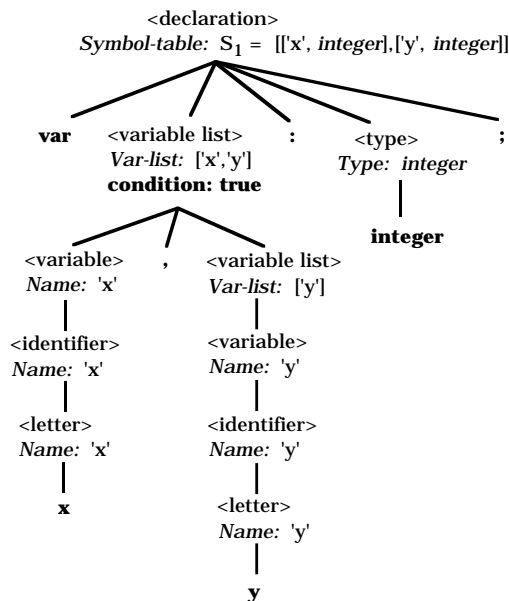
Set of pairs of the form [Name, Type]

**synthesized** for <dec seq> and <dec>

**inherited** for <block>, <cmd seq>, <cmd>, <expr>, <int expr>, <term>, <element>, <bool expr>, <bool term>, <bool element>, and <comparison>

## Declarations

**var** x,y : integer



$\langle \text{var list} \rangle ::= \langle \text{variable} \rangle$

$\text{Var-list}(\langle \text{var list} \rangle) \leftarrow \text{cons}(\text{Name}(\langle \text{variable} \rangle), \text{empty-list})$

$\langle \text{var list} \rangle ::= \langle \text{variable} \rangle, \langle \text{var list} \rangle_2$

$\text{Var-list}(\langle \text{var list} \rangle) \leftarrow \text{cons}(\text{Name}(\langle \text{variable} \rangle), \text{Var-list}(\langle \text{var list} \rangle_2))$

**condition:**

if  $\text{Name}(\langle \text{variable} \rangle)$  is not a member of  $\text{Var-list}(\langle \text{var list} \rangle_2)$   
 then error("")  
 else error("Duplicate variable in declaration list")

## Auxiliary Functions

build-symbol-table(var-list, type)

add-item(name, type, table)

table-union(table<sub>1</sub>, table<sub>2</sub>)

table-intersection(table<sub>1</sub>, table<sub>2</sub>)

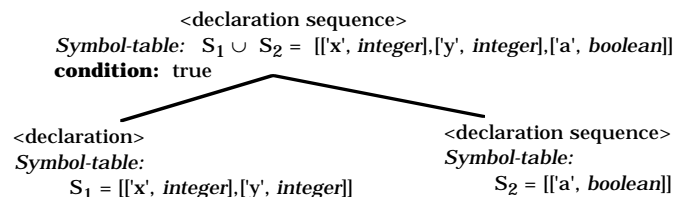
lookup-type(name, table)

```
table-union(table1, table2) =
  if empty(table1)
  then table2
  else
  if lookup-type(first-name(table1), table2) =
    undefined
  then cons(head(table1),
    table-union(tail(table1), table2))
  else table-union(tail(table1), table2)).
```

```
table-intersection(table1, table2) =
  if empty(table1)
  then empty
  else
  if lookup-type(first-name(table1), table2) ≠
    undefined
  then nonempty
  else
  table-intersection(tail(table1), table2).
```

## Declaration Sequences

**var** x,y : integer, var a : boolean



$\langle \text{dec seq} \rangle ::= \epsilon$

$\text{Symbol-table}(\langle \text{dec seq} \rangle) \leftarrow \text{empty-table}$

$\langle \text{dec seq} \rangle ::= \langle \text{dec} \rangle \langle \text{dec seq} \rangle_2$

$\text{Symbol-table}(\langle \text{dec seq} \rangle) \leftarrow \text{table-union}(\text{Symbol-table}(\langle \text{dec} \rangle), \text{Symbol-table}(\langle \text{dec seq} \rangle_2))$

**condition:**

if table-intersection(  
 $\text{Symbol-table}(\langle \text{dec} \rangle),$   
 $\text{Symbol-table}(\langle \text{dec seq} \rangle_2)) = \text{empty}$   
 then error("")  
 else error("Duplicate declaration of an identifier")

<declaration> ::= **var** <variable list> : <type>;  
*Symbol-table*(<declaration>) ←  
 build-symbol-table(  
   *Var-list*(<var list>), *Type*(<type>)).

## Passing Context from Declarations to Commands

<program> ::= **program**<identifier> **is** <block>  
*Symbol-table*(<block>) ←  
 add-item( (*Name*(<identifier>), *program*),  
 empty-table)

<block> ::= <dec seq> **begin**<cmd seq> **end**  
*Symbol-table*(<cmd seq>) ←  
 table-union( *Symbol-table*(<block>),  
   *Symbol-table*(<dec seq>))

### condition:

if table-intersection(  
   *Symbol-table*(<block>),  
   *Symbol-table*(<dec seq>)) = *empty*  
 then error(“”)  
 else error(“Program name used as  
   a variable”)

## Commands

<cmd> ::= <variable> := <expr>  
*Symbol-table*(<expr>) ←  
   *Symbol-table*(<cmd>)  
*Type*(<expr>) ←  
 lookup-type( *Name*(<variable >),  
   *Symbol-table*(<cmd>))

### condition:

case  
 lookup-type(*Name*(<variable>),  
   *Symbol-table*(<cmd>)) is  
*integer, boolean* : error(“”)  
*undefined* : error(“Target variable  
   not declared”)  
*program* : error(“Target variable same  
   as program name”)

<cmd> ::= **read**<variable>

### condition:

case lookup-type(*Name*(<variable>),  
   *Symbol-table*(<cmd>)) is  
*integer* : error(“”)  
*undefined* : error(“Variable not declared”)  
*boolean, program* : error(“Integer var  
   expected for read”)

<command> ::= **write**<expr>  
*Symbol-table*(<expr>) ←  
   *Symbol-table*(<command>)  
*Type*(<expr>) ← *integer*

<command> ::=  
**while**<boolean expr> **do**  
 <command sequence> **end while**  
*Symbol-table*(<boolean expr>) ←  
   *Symbol-table*(<command>)  
*Symbol-table*(<command sequence>) ←  
   *Symbol-table*(<command>)

<command> ::=  
**if** <boolean expr>  
   **then** <command sequence><sub>1</sub>  
   **else** <command sequence><sub>2</sub>  
**end if**  
*Symbol-table*(<boolean expr>) ←  
   *Symbol-table*(<command>)  
*Symbol-table*(<command sequence><sub>1</sub>) ←  
   *Symbol-table*(<command>)  
*Symbol-table*(<command sequence><sub>2</sub>) ←  
   *Symbol-table*(<command>)

## Expressions

Symbol-table is inherited down into the derivation trees for expressions.

Need to check <variable> when expecting an integer expression or a boolean expression.

<expr> ::= <int expr>  
*Symbol-table*(<int expr>) ←  
   *Symbol-table*(<expr>)  
*Type*(<int expr>) ← *Type*(<expr>)  
**condition:** *Type*(<expr>) ∉ { *boolean* }

<expr> ::= <bool expr>  
*Symbol-table*(<boolean expr>) ←  
   *Symbol-table*(<expr>)  
*Type*(<bool expr>) ← *Type*(<expr>)  
**condition:** *Type*(<expr>) ∉ { *integer* }



# Implementing Attribute Grammars

Parser produces a modified token list, not an abstract syntax tree.

## Program

```
<program> ::= program<identifier> is <block>
  Symbol-table(<block>) ←
    add-item((Name(<identifier>), program),
             empty-table)
```

```
program(TokenList) -->
  [program],[ide(I)],[is],
  { addItem(I,program,[ ],InitialSymbolTable) },
  block(Block, InitialSymbolTable),
  { flattenplus([program, ide(I), is, Block],
                TokenList) }.
```

## Block

```
<block> ::= <dec seq> begin <cmd seq> end
  Symbol-table(<cmd seq>) ←
    table-union(Symbol-table(<block>),
                Symbol-table(<dec seq>))
  condition:
    if table-intersection(Symbol-table(<block>),
                          Symbol-table(<dec seq>)) = empty
    then error("")
    else error("Program name used as a var")
```

```
block([ErrorMsg, Decs, begin, Cmds, end],
      InitialSymbolTable) -->
  decs(Decs,DecsSymbolTable),
  { tableIntersection(InitialSymbolTable,
                    DecsSymbolTable,Result),
    tableUnion(InitialSymbolTable,
               DecsSymbolTable, SymbolTable),
    ( Result=nonEmpty,
      ErrorMsg='ERROR: Program name
                used as variable'
    ; Result=empty, ErrorMsg=noError) },
  [begin], cmds(Cmds,SymbolTable), [end].
```

## Command Sequence

```
<cmd seq> ::= <command>
  Symbol-table(<cmd>) ←
    Symbol-table(<cmd seq>)
<cmd seq> ::= <command> ; <cmd seq>2
  Symbol-table(<cmd>) ←
    Symbol-table(<cmd seq>)
  Symbol-table(<cmd seq>2) ←
    Symbol-table(<cmd seq>)
```

```
cmds(Cmds,SymbolTable) -->
  command(Cmd,SymbolTable),
  restcmds(Cmd,Cmds,SymbolTable).
restcmds(Cmd, [Cmd, semicolon|Cmds],
          SymbolTable) -->
  [semicolon], cmds(Cmds,SymbolTable).
restcmds(Cmd,[Cmd],SymbolTable) --> [ ].
```

## Read Command

```
<cmd> ::= read<variable>
  condition:
    case lookup-type(Name(<variable>),
                    Symbol-table(<cmd>)) is
    integer : error("")
    undefined : error("Variable not declared")
    boolean, program : error("Integer var
                              expected for read")
```

```
command([read, ide(V), ErrorMsg],
        SymbolTable) -->
  [read], [ide(V)],
  { lookupType(V,SymbolTable,VarType),
    (VarType = integer, ErrorMsg=noError ;
     VarType = undefined,
     ErrorMsg='ERROR: Variable not declared' ;
     (VarType = boolean ; VarType = program),
     ErrorMsg='ERROR: Integer variable
               expected for read') }.
```

## Write Command

```

<command> ::= write<expr>
    Symbol-table(<expr>) ←
        Symbol-table(<command>)
    Type(<expr>) ← integer

```

```

command([write,E,SymbolTable] -->
    [write], expr(E,SymbolTable,integer).

```

## While Command

```

<command> ::=
    while<boolean expr> do
        <command sequence> end while
    Symbol-table(<boolean expr>) ←
        Symbol-table(<command>)
    Symbol-table(<command sequence>) ←
        Symbol-table(<command>)
    Type(<boolean expr>) ← boolean

```

```

command([while,Test,do,Body,end,while],
    SymbolTable) -->
    [while],
    boolexp( Test,SymbolTable,boolean), [do],
    cmds(Body,SymbolTable), [end], [while].

```

## Assignment Command

```

<command> ::= <variable> := <expr>
    Symbol-table(<expr>) ←
        Symbol-table(<cmd>)
    Type(<expr>) ←
        lookup-type(Name(<variable>),
            Symbol-table(<cmd>))

condition:
    case
    lookup-type(Name(<variable>),
        Symbol-table(<cmd>)) is
    integer, boolean : error("")
    undefined : error("Target variable
        not declared")
    program : error("Target variable same
        as program name")

```

```

command([ide(V),assign,E,ErrorMsg],
    Symtab) -->
    [ide(V)], [assign],
    { lookupType(V,SymbolTable,VarType) },
    ({ VarType = integer },
        (expr(E,Symtab,integer),
            { ErrorMsg=noError }
        ; expr(E,Symtab,boolean),
            { ErrorMsg='ERROR: Int expr expected' }
        )
    ;
    { VarType = boolean },
        (expr(E,Symtab,boolean),
            { ErrorMsg=noError }
        ; expr(E,Symtab,integer),
            { ErrorMsg='ERROR: Bool expr expected' }
        )
    ;
    { VarType = undefined,
        ErrorMsg='ERR: Target of asgn not decd'
    ;
    VarType = program,
        ErrorMsg='ERR: Prog name used as var' }
    expr(E,Symtab,undefined)).

```

## Expressions

```

<expr> ::= <integer expr>
    Symbol-table(<int expr>) ←
        Symbol-table(<expr>)
    Type(<int expr>) ← Type(<expr>)
    condition Type(<expr>) ∉ { boolean }

<expr> ::= <boolean expr>
    Symbol-table(<bool expr>) ←
        Symbol-table(<expr>)
    Type(<int expr>) ← Type(<expr>)
    condition Type(<expr>) ∉ { integer }

```

```

expr(E,SymbolTable,integer) -->
    intexpr(E,SymbolTable,integer).
expr(E,SymbolTable,boolean) -->
    boolexp(E,SymbolTable,boolean).
expr(E,SymbolTable,undefined) -->
    intexpr(E,SymbolTable,undefined).
expr(E,SymbolTable,undefined) -->
    boolexp(E,SymbolTable,undefined).

```

## Integer Expressions

```
<int expr> ::= <term>
  Symbol-table(<term>) ←
    Symbol-table(<int expr>)
  Type(<term>) ← Type(<int expr>)
<int expr> ::= <int expr>2 <weak op> <term>
  Symbol-table(<int expr>2) ←
    Symbol-table(<int expr>)
  Symbol-table(<term>) ←
    Symbol-table(<int expr>)
  Type(<int expr>2) ← Type(<int expr>)
  Type(<term>) ← Type(<int expr>)
```

```
intexpr(E, SymbolTable, Type) -->
  term(T, SymbolTable, Type),
  restintexpr(T, E, SymbolTable, Type).
restintexpr(T, E, SymbolTable, Type) -->
  weakop(Op), term(T1, SymbolTable, Type),
  restintexpr([T, Op, T1], E, SymbolTable, Type).
restintexpr(E, E, SymbolTable, Type) --> [ ].
```

## Terms

```
<term> ::= <element>
  Symbol-table(<element>) ←
    Symbol-table(<term>)
  Type(<element>) ← Type(<term>)
<term> ::= <term>2 <strong op> <element>
  Symbol-table(<term>2) ←
    Symbol-table(<term>)
  Symbol-table(<element>) ←
    Symbol-table(<term>)
  Type(<term>2) ← Type(<term>)
  Type(<element>) ← Type(<term>)
```

```
term(T, SymbolTable, Type) -->
  element(EI, SymbolTable, Type),
  restterm(EI, T, SymbolTable, Type).
restterm(EI, T, SymbolTable, Type) -->
  strongop(Op),
  element(EI1, SymbolTable, Type),
  restterm([EI, Op, EI1], T, SymbolTable, Type).
restterm(T, T, SymbolTable, Type) --> [ ].
```

## Element

```
<element> ::= <numeral>
<element> ::= <variable>
condition:
case lookup-type (Name(<variable>),
  Symbol-table(<element>)) is
  integer : error("")
  undefined : error("Var not declared")
  boolean, program :
    if Type(<element>) = undefined
    then error("")
    else error("Int var expected")
<element> ::= ( <expr> )
  Symbol-table(<expr>) ←
    Symbol-table(<element>)
  Type(<expr>) ← Type(<element>)
<element> ::= - <element>2
  Symbol-table(<element>2) ←
    Symbol-table(<element>)
  Type(<element>2) ← Type(<element>)
```

```
element([num(N)], SymTab, Type) --> [num(N)].
element([ide(I), ErrorMessage], Symtab, Type) -->
  [ide(I)],
  { lookupType(I, Symtab, VarType),
    (VarType = int, Type = int, ErrorMessage=noError
    ; VarType = undefined, ErrorMessage=
      'ERROR: Variable not declared'
    ; Type = undefined, ErrorMessage=noError
    ; (VarType = boolean ; VarType = program),
      ErrorMessage='ERROR: Int var expected') }.
element([lparen, E, rparen], Symtab, Type) -->
  [lparen], intexpr(E, Symtab, Type), [rparen].
element([minus|E], Symtab, Type) -->
  [minus], element(E, Symtab, Type).
```

**Try It** cp ~slonnegr/public/plf/context .