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 recognizes L.

An attempt:

<string> ::= <a seq> <b seq> <c seq>
<a seq> ::= **a** | <a seq> **a**
<b seq> ::= **b** | <b seq> **b**
<c seq> ::= **c** | <c seq> **c**

This context-free grammar recognizes 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: <a seq>, <b seq>, and <c seq>.

Domain of *Size* = Positive Integers

Impose a condition on the first production.

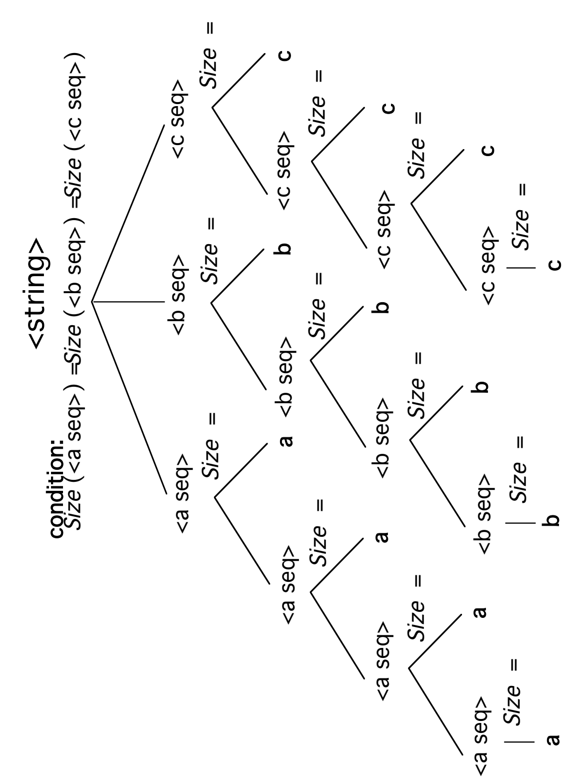
<string> ::= <a seq> <b seq> <c seq>

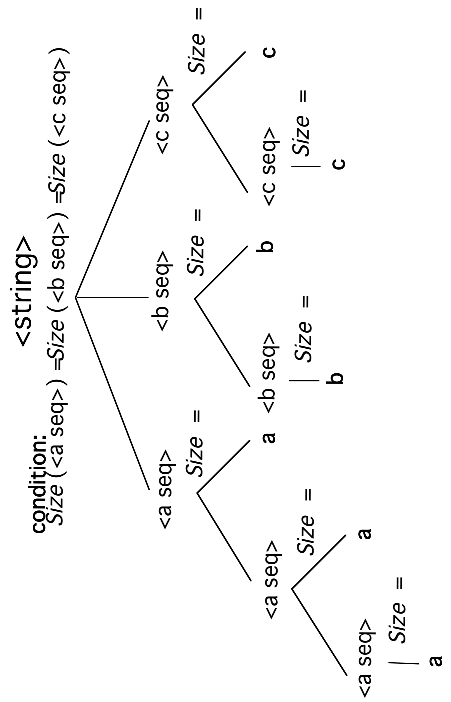
condition:
Size(<a seq>)=*Size*(<b seq>)=*Size*(<c seq>)

<a seq> ::= **a**
Size(<a seq>) ← 1
| <a seq>$_2$ **a**
Size(<a seq>) ← *Size*(<a seq>$_2$)+1

<b seq> ::= **b**
Size(<b seq>) ← 1
| <b seq>$_2$ **b**
Size(<b seq>) ← *Size*(<b seq>$_2$)+1

<c seq> ::= **c**
Size(<c seq>) ← 1
| <c seq>$_2$ **c**
Size(<c seq>) ←*Size*(<c seq>$_2$)+1





Using an Inherited Attribute

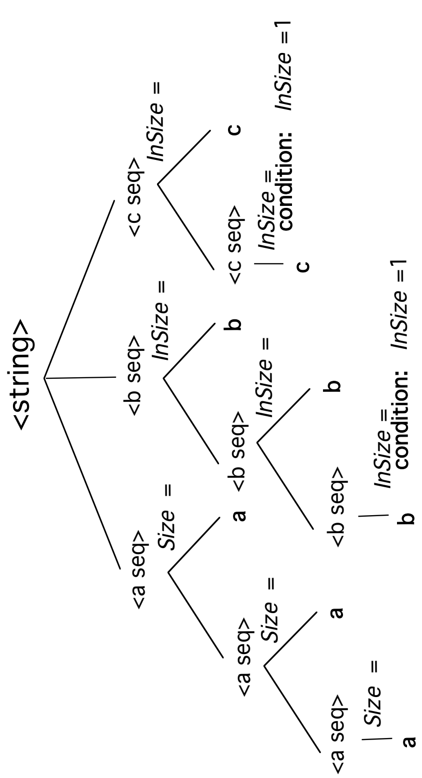
Attach a synthesized attribute *Size* to <a seq> and inherited attributes *InSize* to <b seq> and <c seq>.

<string> ::= <a seq> <b seq> <c seq>
InSize(<b seq>) ← *Size*(<a seq>)
InSize(<c seq>) ← *Size*(<a seq>)

<a seq> ::= **a**
Size(<a seq>) ← 1
| <a seq>$_2$ **a**
Size(<a seq>) ← *Size*(<a seq>$_2$)+1

<b seq> ::= **b**
condition:
InSize(<b seq>) = 1
| <b seq>$_2$ **b**
InSize(<b seq>$_2$) ← *InSize*(<b seq>)-1

<c seq> ::= **c**
condition:
InSize(<c seq>) = 1
| <c seq>$_2$ **c**
InSize(<c seq>$_2$) ← *InSize*(<c seq>)-1



Definition

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

Let G = <N,Σ,P,S> be a context-free grammar. Write a production p ∈P in the form:

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

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

a) Each of its leaf nodes is labeled with a symbol from Σ, and

b) Each interior node t corresponds to a production p ∈ P such that t is labeled with X_0 and t has n_p children labeled with X_1, X_2, …, X_{n_p} in left-to-right order.

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

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

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

I(S) = ∅ where S is the start symbol.

Let A(X) = I(X) ∪ S(X) be the set of attributes of X.

Each attribute *Atb* ∈ A(X) takes a value from some semantic domain associated with that attribute

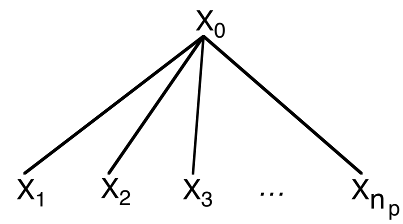
e.g. integers, strings of characters, structures of some type

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

Consider again a production p ∈ P of the form:

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

Derivation Tree:



Each synthesized attribute *Atb* ∈ S(X_0) has its value defined in terms of the attributes in
A(X_1) ∪ A(X_2) ∪ … ∪ A(X_{n_p}) ∪ I(X_0).

Each inherited attribute *Atb* ∈ I(X_k) for $1 \le k \le n_p$ has its value defined in terms of the attributes in
A(X_0) ∪ S(X_1) ∪ … ∪ S(X_{n_p}).

Each production may also have a set of conditions on the values of the attributes in
A(X_0) ∪ A(X_1) ∪ … ∪ 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>	*Val*	---
<binary digits>	*Val*	*Pos*
<fraction digits>	*Val, Len*	---
<bit>	*Val*	*Pos*

The Attribute Grammar

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

Val(<binary numeral>) ←
Val(<binary digits>)+*Val*(<fraction digits>)
Pos(<binary digits>) ← 0

<binary digits> ::= <binary digits>$_2$ <bit>

Val(<binary digits>) ←
Val(<binary digits>$_2$) + *Val*(<bit>)
Pos(<binary digits>$_2$) ← *Pos*(<binary digits>) + 1
Pos(<bit>) ← *Pos*(<binary digits>)

<binary digits> ::= <bit>

Val(<binary digits>) ← *Val*(<bit>)

Pos(<bit>) ← *Pos*(<binary digits>)

<fraction digits> ::= <fraction digits>$_2$ <bit>

Val(<fraction digits>) ←
Val(<fraction digits>$_2$) + *Val*(<bit>)

Len(<fraction digits>) ←
Len(<fraction digits>$_2$) + 1

Pos(<bit>) ← – *Len*(<fraction digits>)

<fraction digits> ::= <bit>

Val(<fraction digits>) ← *Val*(<bit>)

Len(<fraction digits>) ← 1

Pos(<bit>) ← – 1

<bit> ::= **0**

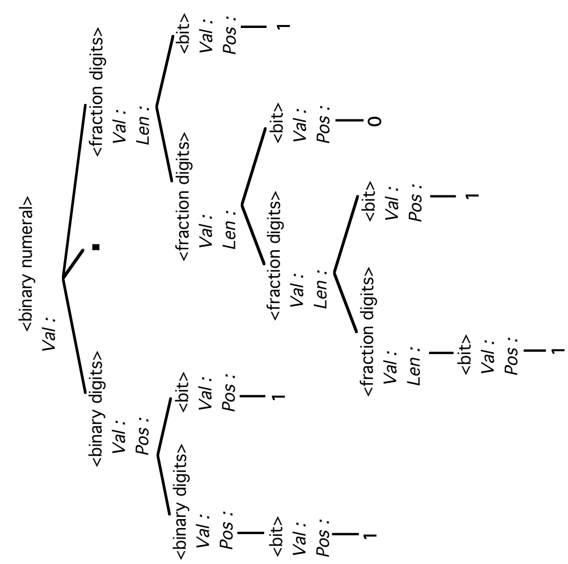
Val(<bit>) ← 0

<bit> ::= **1**

Val(<bit>) ← $2^{Pos\,(\text{<bit>})}$

Observe the order of evaluation of 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	**Value Types**
Type	{ *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>

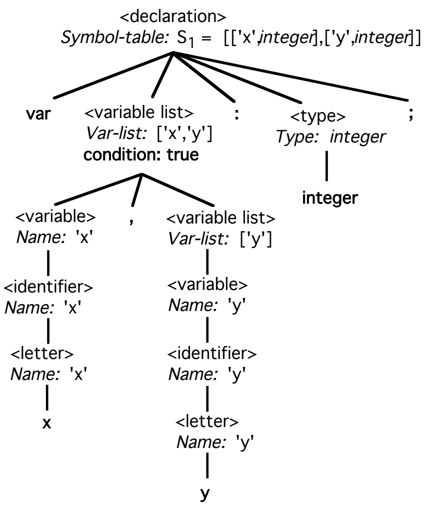
Symbol-table	Set of pairs of the form [Name, Type]

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**;



<var list> ::= <variable>

Var-list(<var list>) ←
cons(*Name*(<variable>), empty-list)

<var list> ::= <variable> **,** <var list>$_2$

Var-list(<var list>) ←
cons(*Name*(<variable>),*Var-list*(<var list>$_2$))

condition:
if *Name*(<variable>) is not a
member of *Var-list*(<var list>$_2$)
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$_1$, table$_2$)

table-intersection(table$_1$, table$_2$)

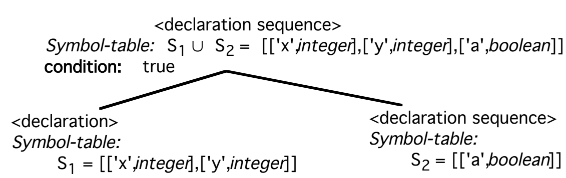
lookup-type(name, table)

table-union(table$_1$, table$_2$) =
if empty(table$_1$)
then table$_2$
else
if lookup-type(first-name(table$_1$),table$_2$) =
undefined
then cons(head(table$_1$),
table-union(tail(table$_1$), table$_2$))
else table-union(tail(table$_1$), table$_2$)).

table-intersection(table$_1$, table$_2$) =
if empty(table$_1$)
then *empty*
else
if lookup-type(first-name(table$_1$),table$_2$) ≠
undefined
then *nonempty*
else
table-intersection(tail(table$_1$), table$_2$).

Declaration Sequences

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



<dec seq> ::= ε
Symbol-table(<dec seq>) ← empty-table

<dec seq> ::= <dec> <dec seq>$_2$
Symbol-table(<dec seq>) ←
table-union(*Symbol-table*(<dec>),
Symbol-table(<dec seq>$_2$))

condition:
if table-intersection(
Symbol-table(<dec>),
Symbol-table(<dec seq>$_2$)) = *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>)

Type(<boolean expr>) ←*boolean*

<command> ::=
if <boolean expr>
then <command sequence>$_1$
else <command sequence>$_2$
end if

Symbol-table(<boolean expr>) ←
Symbol-table(<command>)

Symbol-table(<command sequence>$_1$) ←
Symbol-table(<command>)

Symbol-table(<command sequence>$_2$) ←
Symbol-table(<command>)

Type(<boolean expr>) ←*boolean*

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>) $\notin$ { *boolean* }

<expr> ::= <bool expr>

Symbol-table(<boolean expr>) ←
Symbol-table(<expr>)

Type(<bool expr>) ← *Type*(<expr>)

condition: *Type*(<expr>) $\notin$ { *integer* }

<bool expr> ::= <bool term>

Symbol-table(<bool term>) ←
Symbol-table(<bool expr>)

Type(<bool term>) ← *Type*(<bool expr>)

<bool expr> ::= <bool expr>$_2$ **or** <bool term>

Symbol-table(<bool expr>$_2$) ←
Symbol-table(<bool expr>)

Symbol-table(<bool term>) ←
Symbol-table(<bool expr>)

Type(<bool expr>$_2$) ← *Type*(<bool expr>)

Type(<bool term>) ← *Type*(<bool expr>)

<bool term> ::= <bool elem>

Symbol-table(<bool elem>) ←
Symbol-table(<bool term>)

Type(<bool elem>) ← *Type*(<bool term>)

<bool term> ::= <bool term>$_2$ **and** <bool elem>

Symbol-table(<bool term>$_2$) ←
Symbol-table(<bool term>)

Symbol-table(<bool elem>) ←
Symbol-table(<bool term>)

Type(<bool term>$_2$) ← *Type*(<bool term>)

Type(<bool elem>) ← *Type*(<bool term>)

<bool elem> ::= **true**

<bool elem> ::= **false**

<bool elem> ::= <variable>

condition:
case
lookup-type(*Name*(<variable>),
Symbol-table(<bool elem>)) is
boolean : error(“”)
undefined : error(“Variable not declared”)
integer, *program* :
if *Type*(<bool elem>) = *undefined*
then error(“”)
else error(“Bool variable expected”)

<bool elem> ::= **(** <bool expr> **)**

Symbol-table(<bool expr>) ←
Symbol-table(<bool elem>)

Type(<bool expr>) ← *Type*(<bool elem>)

<bool elem> ::= **not(** <bool expr> **)**

Symbol-table(<bool expr>) ← *Symbol-table*(<bool elem>)

Type(<bool expr>) ← *Type*(<bool elem>)

<comparison> ::= <integer expr>$_1$ <relation> <integer expr>$_2$

Symbol-table(<integer expr>$_1$) ← *Symbol-table*(<comparison>)

Symbol-table(<integer expr>$_2$) ← *Symbol-table*(<comparison>)

Type(<integer expr>$_1$) ← *integer*

Type(<integer expr>$_2$) ← *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 delcared' ;
    (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],
   boolexpr(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, Symtab, 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) -->
        boolexpr(E,SymbolTable,boolean).
expr(E,SymbolTable,undefined) -->
        intexpr(E,SymbolTable,undefined).
expr(E,SymbolTable,undefined) -->
        boolexpr(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(El,SymbolTable,Type),
        restterm(El,T,SymbolTable,Type).
restterm(El,T,SymbolTable,Type) -->
    strongop(Op),
    element(El1, SymbolTable,Type),
    restterm([El,Op,El1], 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),ErrorMsg],Symtab,Type) -->
  [ide(I)],
  { lookupType(I,Symtab,VarType),
    (VarType = int, Type = int, ErrorMsg=noError
    ; VarType = undefined, ErrorMsg=
                 'ERROR: Variable not declared'
    ; Type = undefined, ErrorMsg=noError
    ; (VarType = boolean ; VarType = program),
         ErrorMsg='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 .