CSE 3302 Notes 4: Semantic Analysis

(Last updated 9/26/12 7:32 AM)

Syntax vs. Semantics

$$\Sigma \quad \sigma \qquad \Sigma \quad \sigma$$

$$\overline{A \vee B} \leftrightarrow \overline{A} \wedge \overline{B}$$

$$(A \rightarrow B) \vee (B \rightarrow A)$$

Well-formed vs. meaningful

Context-free vs. context-sensitive

Static Semantics

Names declared, typechecking

`switch` statement `case`s are independent

Value assigned before use

Dynamic Semantics

What the generated code must do

Language definition . . .

`C` allowed pointer values and loops

4.1. The Role of the Semantic Analyzer

Assertions = Additional properties to be assured at specific execution points

C/C++ - can be disabled

Dynamic checks - Either built into interpreter or part of generated code

Static analysis

Detects errors

Can avoid some dynamic checking

Can reduce method dispatch costs

4.2. ATTRIBUTE GRAMMARS

General method for defining semantics

Alternatives:

Prototype language implementation (first Ada implementation in SETL, Pascal in Pascal)

Two-level grammar (Algol 68)

Denotational semantics (functional/logic languages)

Natural language . . .

Initially, just a formal method for semantics (Knuth, 1968)

Cornell Program Synthesizer (PL/I, 1981)

Compiler-compilers

E, T, F grammar in book - *synthesized* attributes (bottom-up)

From `http://homepage.cs.uiowa.edu/~slonnegr/plf/Book/`, chapter 3

Strings of form $a^n b^n c^n$ are the only ones acceptable.

Start with grammar:

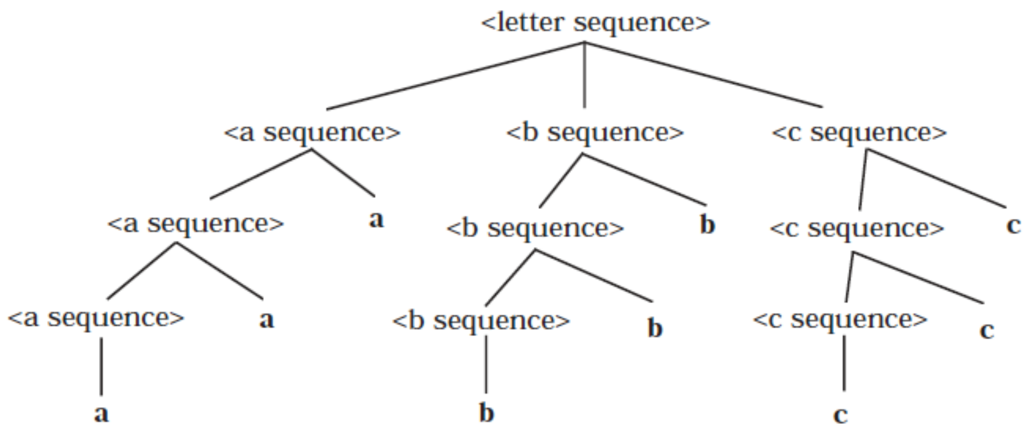
<letter sequence> ::= <a sequence> <b sequence> <c sequence>

<asequence> ::= **a** | <a sequence>**a**

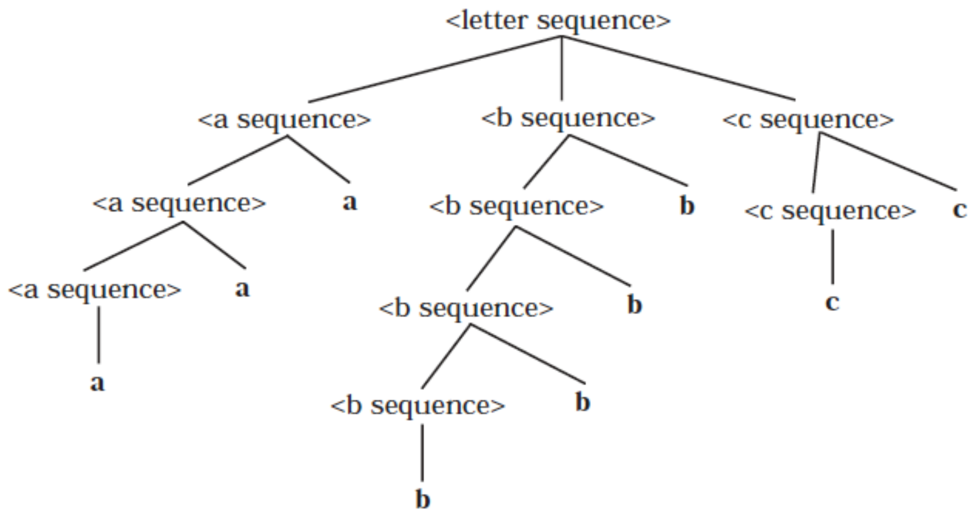
<bsequence> ::= **b** | <bsequence> **b**

<csequence> ::= **c** | <csequence> **c**

Accepted string:



Also accepted (?)



Attribute grammar to capture context-sensitivity with synthesized attributes:

<lettersequence> ::= <asequence> <bsequence> <csequence>

condition :

Size (<asequence>) = *Size* (<bsequence>) = *Size* (<csequence>)

<asequence> ::= **a**

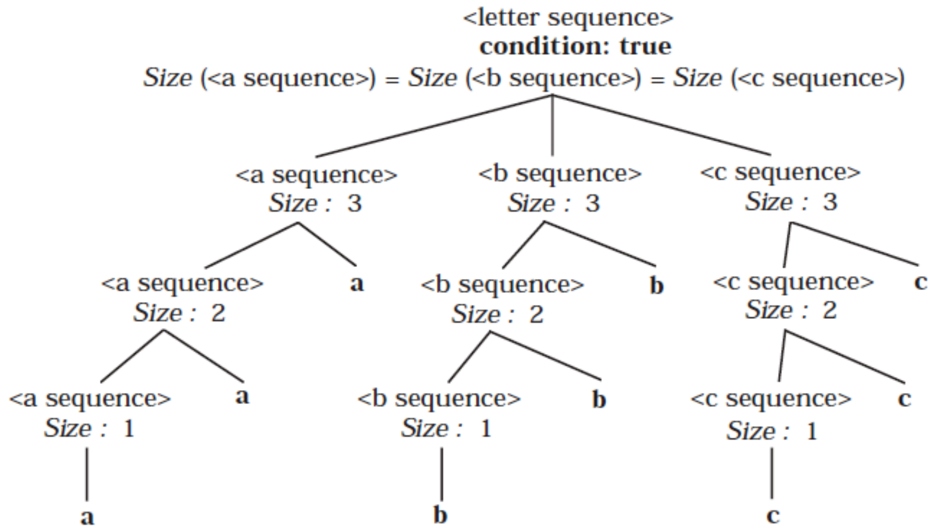
Size (<asequence>) $\leftarrow$ 1

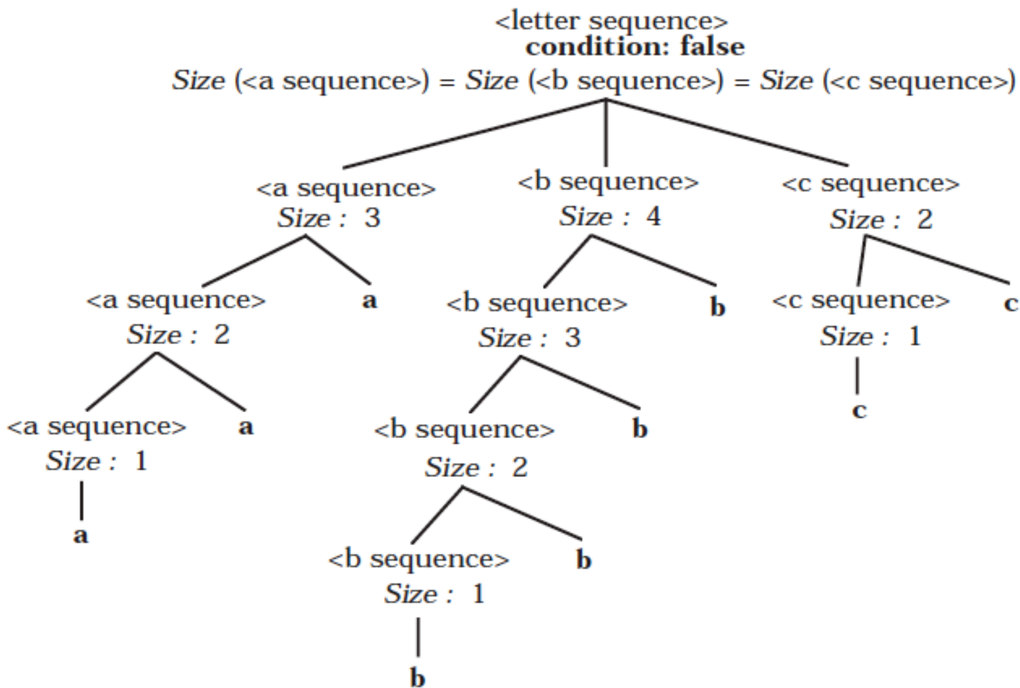
| <asequence> $_2$ **a**

Size (<asequence>) $\leftarrow$ *Size* (<asequence> $_2$) + 1

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

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





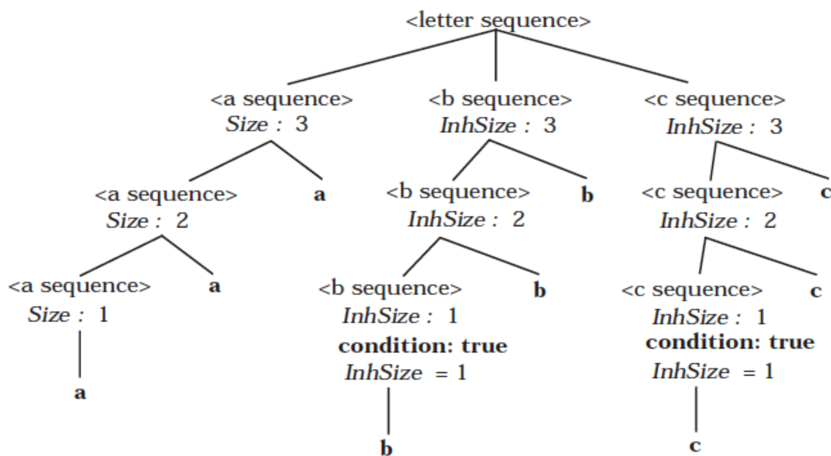
Same example, but taking advantage of *inherited* attributes:

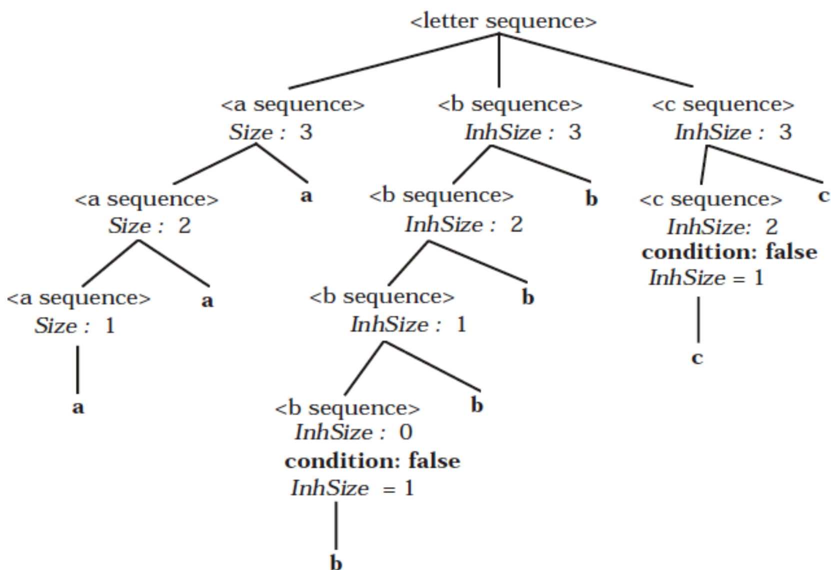
<lettersequence> ::= <asequence> <bsequence> <csequence>
InhSize (<bsequence>) ← *Size* (<asequence>)
InhSize (<csequence>) ← *Size* (<asequence>)

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

<bsequence> ::= **b**
condition: *InhSize* (<bsequence>) = 1
| <bsequence> $_2$ **b**
InhSize (<bsequence> $_2$) ← *InhSize* (<bsequence>) – 1

<csequence> ::= **c**
condition: *InhSize* (<csequence>) = 1
| <csequence> $_2$ **c**
InhSize (<csequence> $_2$) ← *InhSize* (<csequence>) – 1





Meaning of binary numbers using synthesized attributes:

<binary numeral> ::= <binary digits>$_1$. <binary digits>$_2$

Val (<binary numeral>) ← *Val* (<binary digits>$_1$) +

Val (<binary digits>$_2$) / 2$^{Len\ (\text{binary digits}_2)}$

<binary digits> ::=

<binary digits>$_2$ <bit>

Val (<binary digits>) ← 2 • *Val* (<binary digits>$_2$) + *Val* (<bit>)

Len (<binary digits>) ← *Len* (<binary digits>$_2$) + 1

| <bit>

Val (<binary digits>) ← *Val* (<bit>)

Len (<binary digits>) ← 1

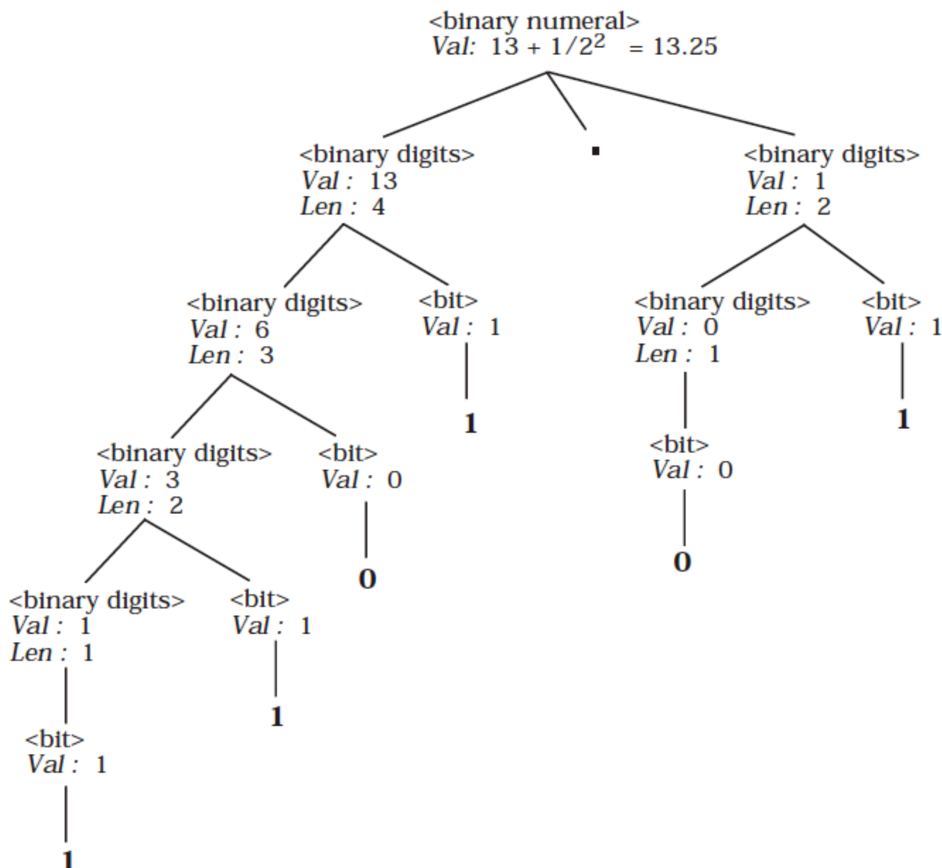
<bit> ::=

0

Val (<bit>) ← 0

| **1**

Val (<bit>) ← 1



But only the value of the entire number is computed. What about each bit’s contribution?

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	*Val*	—
<binary digits>	*Val*	*Pos*
<fraction digits>	*Val, Len*	—
<bit>	*Val*	*Pos*

<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>)

| <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>)

| <bit>

Val (<fraction digits>) ← *Val* (<bit>)

Len (<fraction digits>) ← 1

Pos (<bit>) ← - 1

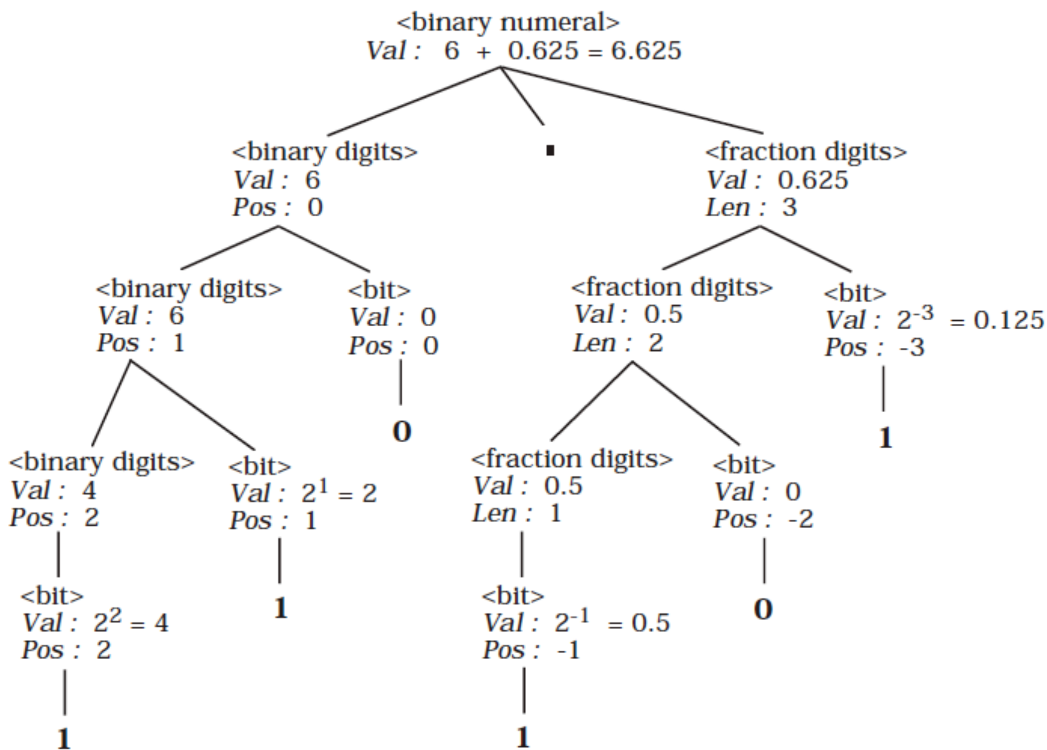
<bit> ::=

0

Val (<bit>) ← 0

| **1**

Val (<bit>) ← $2^{Pos\ (<bit>)}$



Small programming language (Wren) and its context sensitivities

```
program binary is
    var n,p : integer ;
begin
    read n; p := 2;
    while p<=n do p := 2*p end while ;
    p := p/2;
    while p>0 do
        if n>= p  then write  1; n := n–p else write  0 end if ;
        p := p/2
    end while
end
```

<program> ::= **program** <identifier> **is** <block>

<block> ::= <declaration seq> **begin** <command seq> **end**

<declaration seq> ::= ε | <declaration> <declaration seq>

<declaration> ::= **var** <variable list> **:** <type> **;**

<type> ::= **integer** | **boolean**

<variable list> ::= <variable> | <variable> **,** <variable list>

<command seq> ::= <command> | <command> **;** <command seq>

<command> ::= <variable> **:=** <expr> | **skip**

| **read** <variable> | **write** <integer expr>

| **while** <boolean expr> **do** <command seq> **end while**

| **if** <boolean expr> **then** <command seq> **end if**

| **if** <boolean expr> **then** <command seq> **else** <command seq> **end if**

<expr> ::= <integer expr> | <boolean expr>

<integer expr> ::= <term> | <integer expr> <weak op> <term>

<term> ::= <element> | <term> <strong op> <element>

<element> ::= <numeral> | <variable> | **(** <integer expr> **)** | – <element>

<boolean expr> ::= <boolean term> | <boolean expr> **or** <boolean term>

<boolean term> ::= <boolean element>
| <boolean term> **and** <boolean element>

<boolean element> ::= **true** | **false** | <variable> | <comparison>
| **not (** <boolean expr> **)** | **(** <boolean expr> **)**

<comparison> ::= <integer expr> <relation> <integer expr>

<variable> ::= <identifier>

<relation> ::= <= | < | = | > | >= | <>

<weak op> ::= + | –

<strong op> ::= * | /

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

<letter> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m**
| **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

<numeral> ::= <digit> | <digit> <numeral>

<digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Figure 1.8: BNF for 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 an integer variable.
6. An identifier occurring as a Boolean element must be a Boolean variable.
7. An identifier occurring in a read command must be an integer variable.

Figure 1.11: Context Conditions for Wren

<reserved word> ::= **program** | **is** | **begin** | **end** | **var** | **integer**
| **boolean** | **read** | **write** | **skip** | **while** | **do** | **if**
| **then** | **else** | **and** | **or** | **true** | **false** | **not**.

8. No reserved word may be used as an identifier.

1. An attempt is made to divide by zero.
2. A variable that has not been initialized is accessed.
3. A **read** command is executed when the input file is empty.
4. An iteration command (**while**) does not terminate.

Figure 1.12: Semantic Errors in Wren

Attribute	**Value Types**
Type	{ *integer*, *boolean*, *program*, *undefined* }
Name	String of letters or digits
Var-list	Sequence of Name values
Symbol-table	Set of pairs of the form [Name, Type]

Figure 3.9: Attributes and Values

Nonterminals	**Synthesized Attributes**	**Inherited Attributes**
<block>	—	*Symbol-table*
<declarationsequence>	*Symbol-table*	—
<declaration>	*Symbol-table*	—
<variable list>	*Var-list*	—
<type>	*Type*	—
<commandsequence>	—	*Symbol-table*
<command>	—	*Symbol-table*
<expr>	—	*Symbol-table, Type*
<integer expr>	—	*Symbol-table, Type*
<term>	—	*Symbol-table, Type*
<element>	—	*Symbol-table, Type*
<boolean expr>	—	*Symbol-table, Type*
<boolean term>	—	*Symbol-table, Type*
<boolean element>	—	*Symbol-table, Type*
<comparison>	—	*Symbol-table*
<variable>	*Name*	—
<identifier>	*Name*	—
<letter>	*Name*	—
<digit>	*Name*	—

Figure 3.10: Attributes Associated with Nonterminal Symbols

```
program p is
    var x, y : integer;
    var a : boolean;
begin
    :
end
```

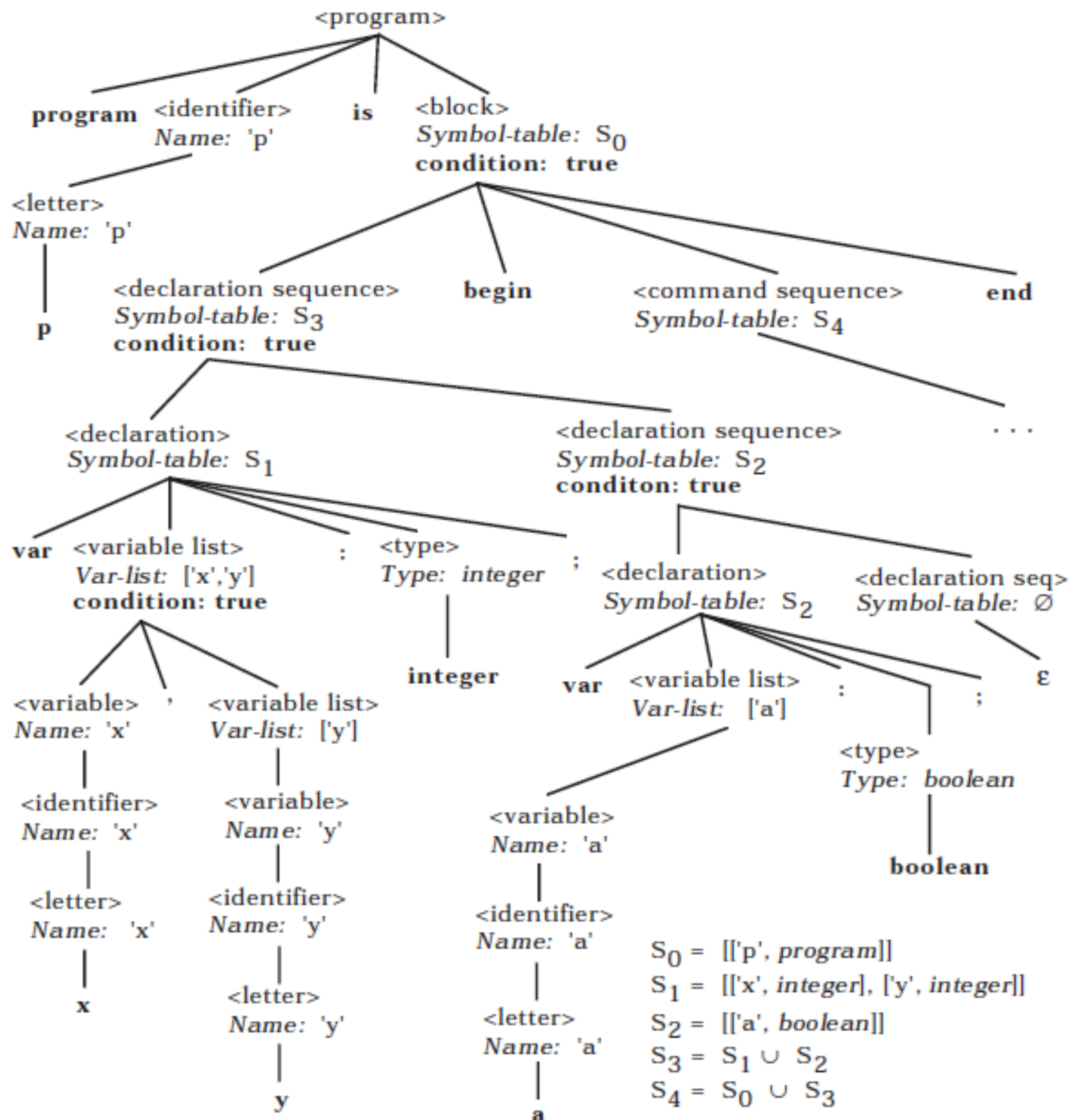


Figure 3.11: Decorated Parse Tree for Wren Program Fragment

Application: Structure Editor

Goal: Editor based on moving among program constructs

Various formatting possibilities - including highlighting, indenting

Use window dimensions in decisions

Eliding of "uninteresting code"

Simple formatting:

```
if (x>y) { temp=x; x=y; y=temp; }
```

```
if (x>y)
{
  temp=x;
  x=y;
  y=temp;
}
```

Synthesized attributes - space needed for construct

Inherited attributes - position of construct in display

Application: Connecting Defs to Uses for Data Flow Analysis

Def: Assigning value to a variable, e.g. (identifier, line#/token#)

Use: Value of variable is used in an expression

Basic block: Straight-line code, no branching

Historic approach: Treat program as flowchart (graph)

Construct-based approach:

INPUT(x) set: defs with some path to construct x without an intervening def
KILL(x) set: defs always invalidated before leaving construct x
GEN(x) set: defs in construct x with path to exit
OUTPUT(x): defs getting through or generated in construct x

$(\text{INPUT}(x) - \text{KILL}(x)) \cup \text{GEN}(x) = \text{OUTPUT}(x)$

Each construct (basic block, control structure, function, etc.) has equations and way to match defs to uses.

Could integrate def-use information with structure editor

4.3. Evaluating Attributes

Iterative Firing/Propagation/Queue/Petri Net/Data Flow Machine Ideas

- Build AST
- Initialize obvious values
- Initialize queue with non-finalized tree nodes
- Iterate with updates to queue until changes stop

Integrate With Parser

- OK for static semantics
- Not useful for substantial analysis
- Avoids storage issues

Circularity of attribute grammar

- Does an attribute value at a node depend on itself due to a path of synthesized/inherited attributes?