CSE 3302 Notes 3: Syntax & Semantics

(Last updated 11/21/15 8:49 AM)

References:

Gabbrielli-Martini: 2
Crockford: 2, 7, D
Wirth: 5, A
Dybvig: 3.1-3.2

3.1. Syntax vs. Semantics

In logic and formal systems:

Well-formed, syntactic/axiomatic, semantic/model-theoretic

$$\Sigma \vdash \sigma \qquad \Sigma \models \sigma$$

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

$$(A \rightarrow B) \vee (B \rightarrow A)$$ (https://en.wikipedia.org/wiki/Steve_Martin#Comedy)

Asides: Hilbert's Program (1900), Gödel's Incompleteness Theorem (1931), and https://en.wikipedia.org/wiki/Gödel,_Escher,_Bach

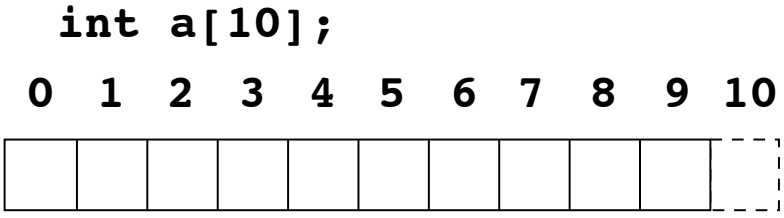
Static Semantics - Can't represent in "context-free" fashion

Context-sensitive checking beyond syntax (symbol tables)
Names declared, type checking (examine expression details)
`switch` statement `case`s are independent
Values assigned before use
Function call arguments - appropriate number and types

Dynamic Semantics - What generated code does

Error checking (types, array bounds, assertions)
State changes
Language definition . . .
C allowed pointer values and loops (see standard, 6.5.8, p. 85, "one past the last element")

```
int a[10];
```



Practical Language Definitions:

Pascal: Jensen and Wirth, *PASCAL: User Manual and Report*. No current standard, but see: http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=988167 "How to avoid getting schlonked by Pascal"

JavaScript: http://www.ecma-international.org/ecma-262/6.0/index.html

Scheme: http://www.r6rs.org/ and IEEE http://ieeexplore.ieee.org.ezproxy.uta.edu/stamp/stamp.jsp?tp=&arnumber=159138

Java: https://docs.oracle.com/javase/specs/index.html

C 99: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

C++ 2014: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

Coding Standards:

http://www.stroustrup.com/JSF-AV-rules.pdf (C++ Lockheed Martin)

https://github.com/google/styleguide/blob/gh-pages/README.md (Google)

3.2. Specifying Syntax: Regular Expressions and Context-Free Grammars

Railroad Diagrams

http://en.wikipedia.org/wiki/Syntax_diagram, also gives format of

Backus-Naur Form (BNF, productions free of nesting)

Extended BNF (EBNF, allows nesting/iteration to shorten grammar, Wirth uses heavily . . .)

Pascal-S examples: p. 54-56 of http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/Wirth-PascalS.pdf

PL/0 examples: http://ranger.uta.edu/~weems/NOTES3302/BASELINE/DIAGRAMS/

JavaScript: Appendix D of Crockford.

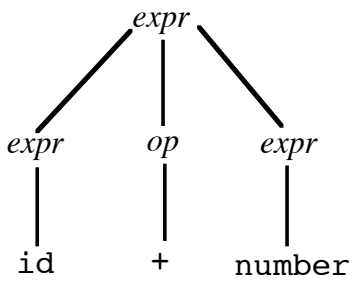
Concept of (Context Free) Grammar

Terminal Nonterminal Production Start Symbol

Derivation - leftmost and rightmost

Simple Expression Grammar:

expr → `id` | `number` | `-` *expr* | `(` *expr* `)` | *expr op expr*
op → `+` | `-` | `*` | `/`
expr ⇒ *expr op expr* ⇒ *expr* `+` *expr* ⇒ `id` `+` *expr* ⇒ `id` `+` `number`
expr $\overset{+}{\Rightarrow}$ `id` `+` `number`



Left and Right Recursion

x → *x* `a` *x* → `a` *x*

ε

Language generated by a grammar

Ambiguity (Gabbrielli figure 2.7 and if-then-else)

```
if (green)                  if (green)
  if (yellow)                 if (yellow)
    if (red)                    if (red)
      napTime();                  napTime();
else                            else
  wakeUp();                       wakeUp();
```

Pascal, JavaScript, and C standards address this informally ("nearest preceding **if**")

Classic Expression Grammar (BNF)

expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → `id` | `number` | `-` *factor* | `(` *expr* `)`
add_op → `+` | `-`
mult_op → `*` | `/`

Unfortunately, this approach does not generalize for languages with many precedence levels (e.g. sections 6.5 and A.2.1 of C 99 standard, but summary tables are common, e.g. p. 16 of *The Good Parts*)

Regular Expressions

Terminals (*expr*) *expr expr* *expr* | *expr* *expr**

Abbreviations: $expr^{k}$ $expr^{+}$

Binary strings: $(0 \mid 1)^{+}$

Binary strings with even length: $((0 \mid 1)(0 \mid 1))^{+}$

JavaScript name: $(a \mid b \mid ... \mid z \mid A \mid B \mid ... Z) (a \mid b \mid ... \mid z \mid A \mid B \mid ... Z \mid 0 \mid 1 \mid ... 9 \mid \_)^{*}$

JavaScript numbers?: see `parseInt`, `parseFloat`, and `Number` constructor . . .
or enter candidates into your browser's console

Scanning Examples:

"Cat Dog ++ +" in JavaScript (`http://ranger.uta.edu/~weems/NOTES3302/catDog.html`)

Lab 1 Fall 2012 in JavaScript - tokenizes expressions (`http://ranger.uta.edu/~weems/NOTES3302/lab1fall12.html`)

Wirth: PL/0 - `getsym/getch`, Pascal-S - `insymbol/nextch`

Playing out-of-bounds (Power Lexer, `http://ranger.uta.edu/~weems/NOTES3302/power.html`)

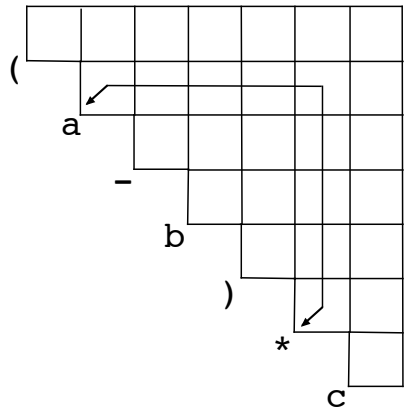
```
collect=/^(.*)(.*)\1{2}\2{3}\1{4}\2{5}$/,result;

result=collect.exec(input);
if (result==null)
  myOutput.value = "search bonked";
else
  myOutput.value = result[1]+" "+result[2];
```

(Chapter 7, p. 66 of *Good Parts* is useful, also `http://site.ebrary.com.ezproxy.uta.edu/lib/utarlington/reader.action?ppg=269&docID=10763621&tm=1435086045077`)

3.3. PARSING

Brute Force Recursive (find handles and reduce) - $O\left(n^{3}\right)$ time

What non-terminal symbols have a derivation leading to each substring of the input?



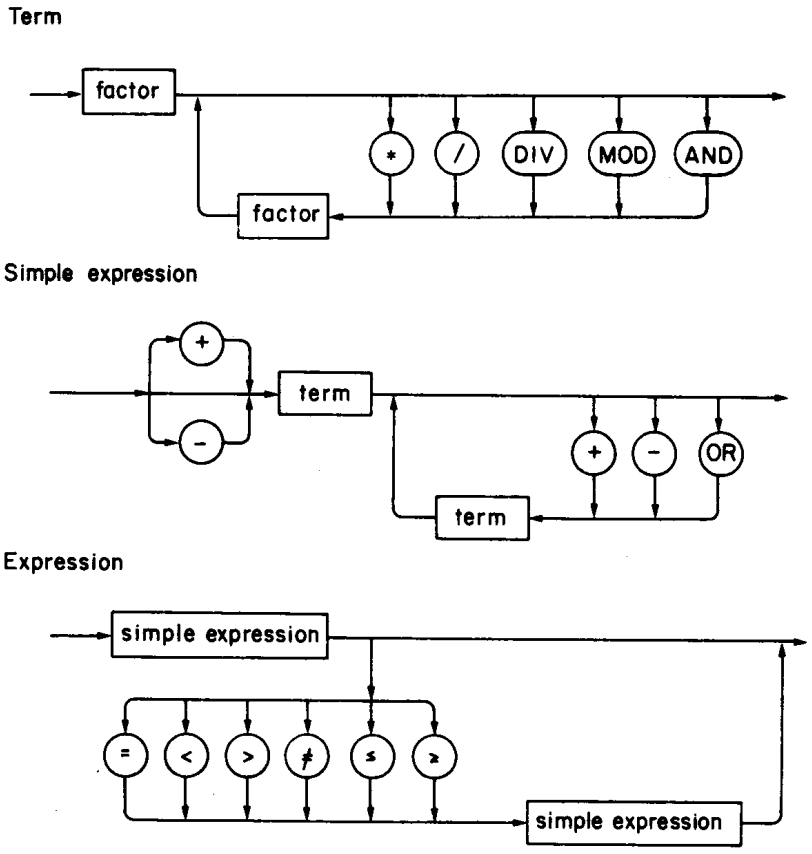
expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → `id` | `number` | `-` *factor* | `(` *expr* `)`
add_op → `+` | `-`
mult_op → `*` | `/`

Form of grammar is usually restricted to simplify details and achieve time bound.

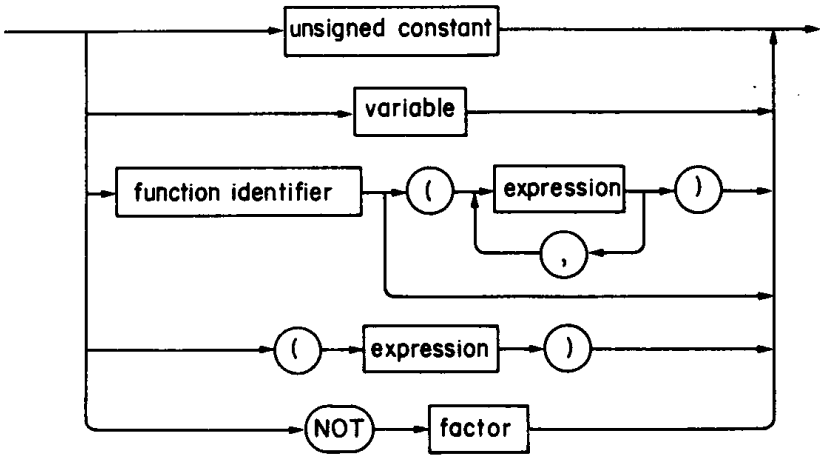
Similar to DP solution for optimal matrix multiplication

Recursive Descent (example of top-down)

"Pascal is a language that can be parsed with a lookahead of a single symbol. The compiler therefore uses the simple and efficient method of top-down parsing with one-symbol lookahead." (Pascal-S report)



Factor




```
function test(s1,s2,/*: symset;*/ n/*: integer*/)
{
  if (!s1[sym])
  {
    error(n);
    while (!s1[sym] && !s2[sym])
      getsym();
  }
} /*test*/
...
      function expression(fsys/*: symset*/)
      {
         var addop/*: symbol*/;
         function term(fsys/*: symset*/)
         {
            var mulop/*: symbol*/;
            function factor(fsys/*: symset*/)
            {
               var i;
               test(facbegsys, fsys, 24);
               while (facbegsys[sym])
               {
                  if (sym == ident)
                  {
                     i=position(id);
                     if (i == 0)
                        error(11);
                     else
                        switch (table[i].kind) {
                           case constant:
                              gen(lit, 0, table[i].val);
                              break;
                           case varible:
                              gen(lod, lev-table[i].level, table[i].adr);
                              break;
                           case vararr:
                              getsym();
                              if (sym!=lbrac)
                                error(31);
                              else
                                getsym();
                              expression(unionProps(constructSet(rbrac),fsys));
                              if (sym!=rbrac)
                                error(22);
                              gen(lda,lev-table[i].levela,table[i].adra);
                              break;
                           case proc:
```

```
                    error(21);
                    break;
                 case instream:
                    gen(rdi, 0, 0);
                    break;
                 case outstream:
                    error(21);
                    break;
                 default:
                    listingbox.innerHTML+=' bad kind in factor()';
                    listingbox.scrollTop=99999;
                    throw 'bad kind in factor()';
                 }
              getsym();
           }
           else if (sym == number)
           {
              if (num >  nmax)  /*BPW - shouldn't happen*/
              {
                 error(30);
                 num = 0;
              }
              gen(lit, 0, num);
              getsym();
           }
           else if (sym == lparen)
           {
              getsym();
              expression(unionProps(constructSet(rparen),fsys));
              if (sym == rparen)
                 getsym();
              else
                 error(22);
           }
           test(fsys, constructSet(lparen), 23);
        }
     } /*factor*/

     /*term*/
     factor(unionProps(constructSet(times,slash),fsys));
     while (sym==times || sym==slash)
     {
        mulop=sym;
        getsym();
        factor(unionProps(constructSet(times,slash),fsys));
        if (mulop==times)
           gen(opr,0,4);
        else
           gen(opr,0,5);
      }
  } /*term*/

  /*expression*/
  if (sym==plus || sym==minus)
  {
     addop = sym;
     getsym();
     term(unionProps(constructSet(plus,minus),fsys));
     if (addop == minus)
        gen(opr, 0,1);
  }
  else
  {
    term(unionProps(constructSet(plus,minus),fsys));
  }
  while (sym==plus || sym==minus)
  {
     addop = sym;
```

```
            getsym();
            term(unionProps(constructSet(plus,minus),fsys));
            if (addop==plus)
               gen(opr,0,2);
            else
               gen(opr,0,3);
         }
      } /*expression*/
```

Advantages

Easy to use for small, "well-designed" languages - especially non-expression constructs

Error recovery can be tailored

"Its main principle is that each parser always returns control after having advanced up to a symbol that may legally follow the sentential construct that the parser is supposed to process."

Disadvantages:

Languages with many precedence levels (C++) defy single-symbol lookahead

Pascal precedences simplify grammar, but changes use of parentheses

PL/0:

Allowable *beginning symbols* for each construct (see railroad diagrams)

```
declbegsys=constructSet(constsym, varsym, procsym);          Declarations
statbegsys=constructSet(beginsym, callsym, ifsym, whilesym); Statements
facbegsys=constructSet(ident, number, lparen);               Factors
```

Follow symbols (right end of construct) accumulate (using Pascal sets) when going deeper into recursion to allow error recovery from:

1. Drastic syntax errors by moving up to a containing construct (`s1` for `test`)
2. Minor errors by skipping over symbols for a contained construct (`s2` for `test`)

Expression parsing may also be addressed using: `http://en.wikipedia.org/wiki/Shunting-yard_algorithm`

3.4. ATTRIBUTE GRAMMARS

General method for defining semantics

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

Cornell Program Synthesizer (PL/I, 1981, `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=358746.358755`)

Compiler-compilers ("Attribute grammar paradigms—a high-level methodology in language implementation", especially example 1.3.1, `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=197409`)

Synthesized attributes (bottom-up)

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

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

Using JavaScript regular expressions: `http://ranger.uta.edu/~weems/NOTES3302/anbncn.html`

Start with grammar:

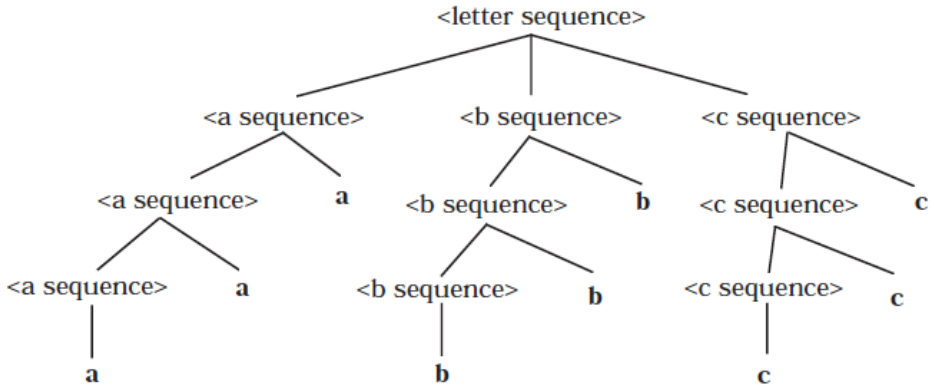
<letter sequence> ::= <a sequence> <b sequence> <c sequence>

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

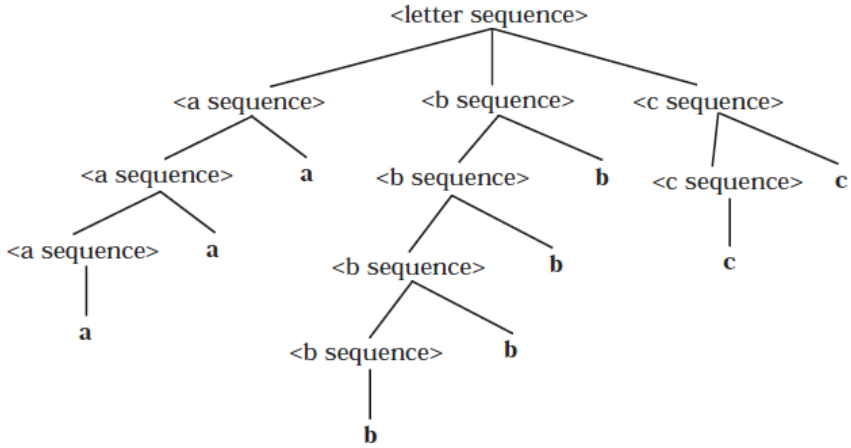
<bsequence> ::= **b** | <bsequence> **b**

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

Accepted string:



Also accepted (?)



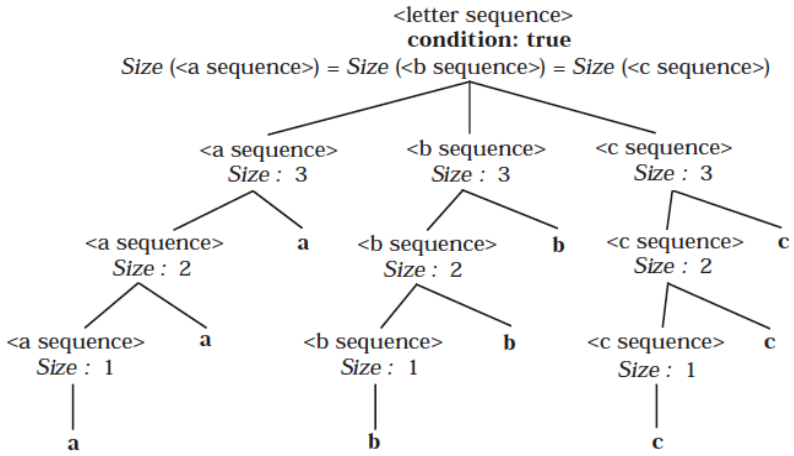
Attribute grammar to capture context-sensitivity with synthesized attributes:

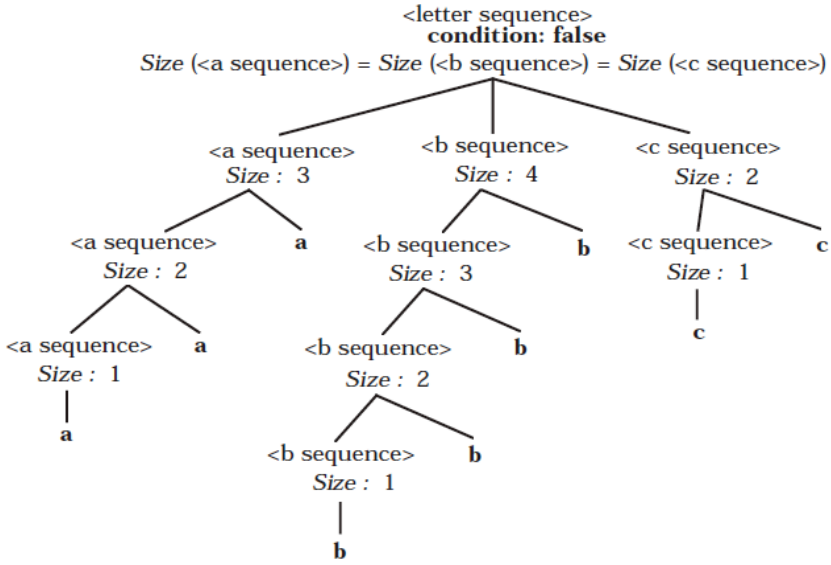
<lettersequence> ::= <asequence> <bsequence> <csequence>
condition :
Size (<asequence>) = *Size* (<bsequence>) = *Size* (<csequence>)

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





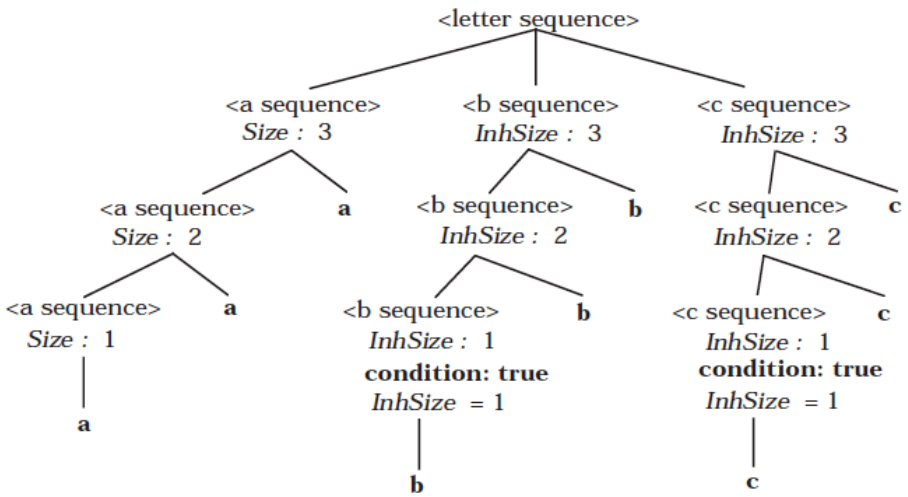
Same example, but taking advantage of *inherited* attributes (top-down):

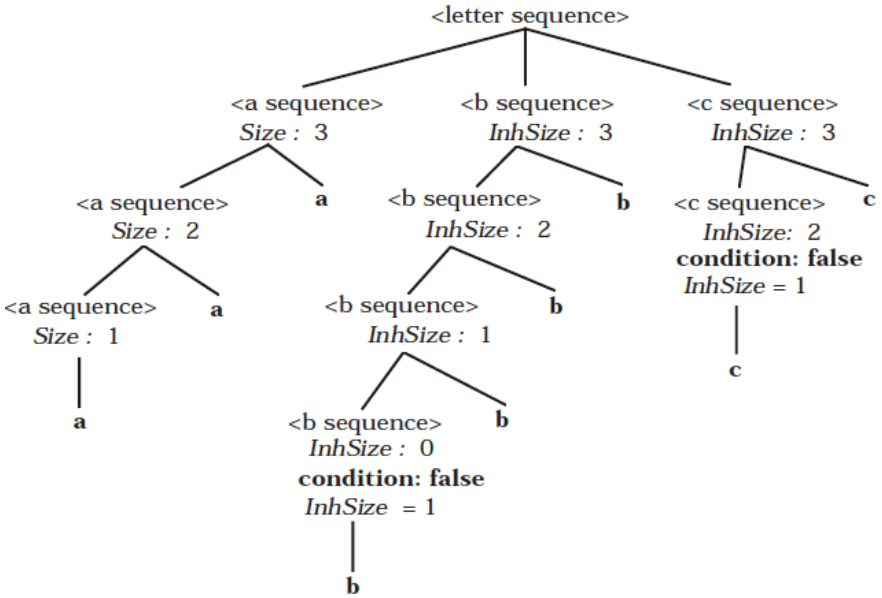
<lettersequence> ::= <asequence> <bsequence> <csequence>
InhSize (<bsequence>) ← *Size* (<asequence>)
InhSize (<csequence>) ← *Size* (<asequence>)

<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



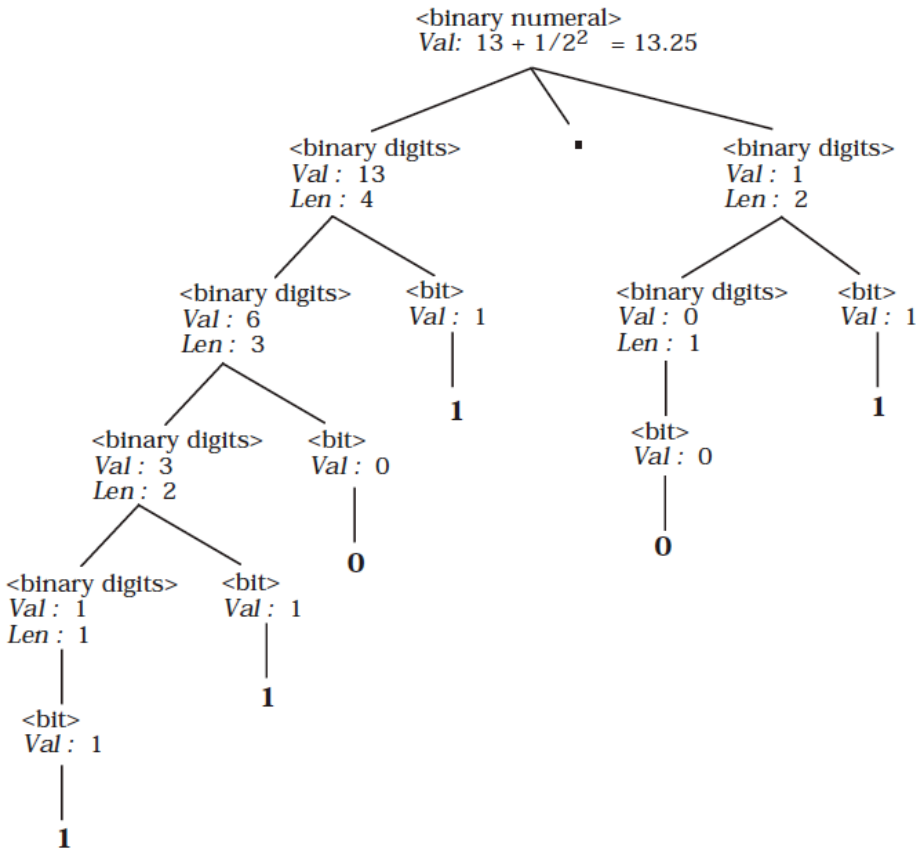


Meaning of binary numbers using synthesized attributes:

<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

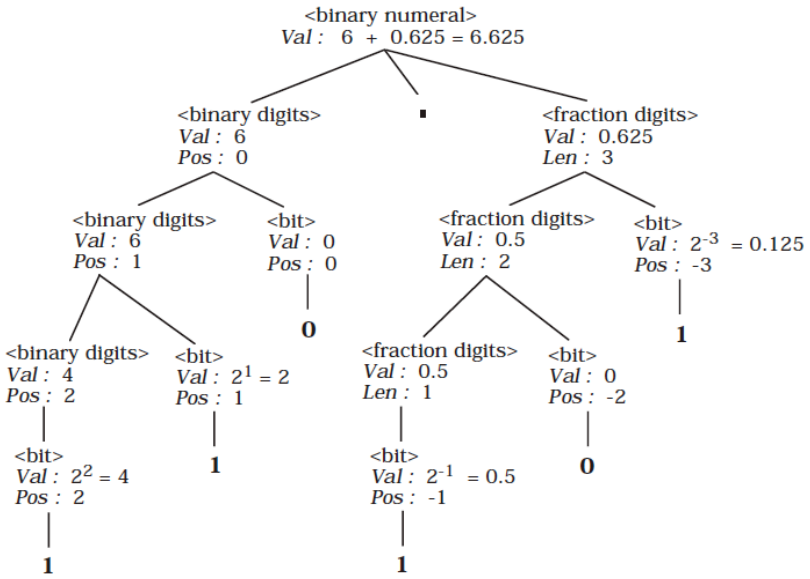
<bit> ::=

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
```

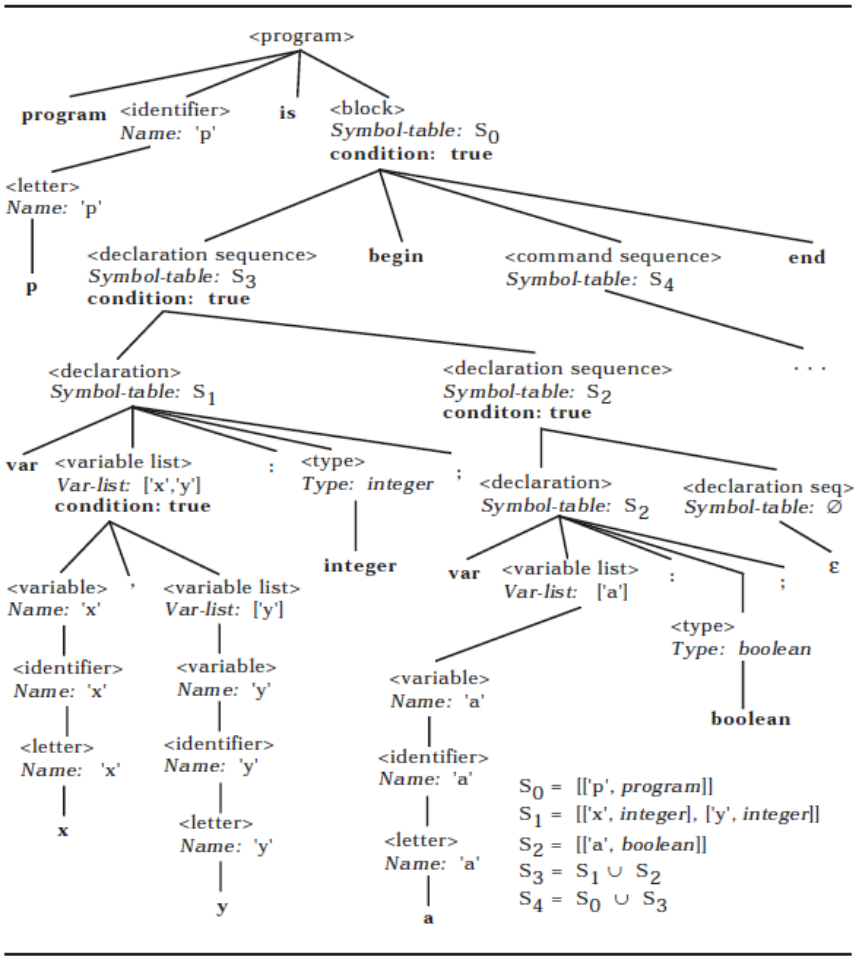


Figure 3.11: Decorated Parse Tree for Wren Program Fragment

3.5. Alternative: Operational Semantics

Historically: Prototype language implementation

IBM: PL/I
Wirth: PL/0, Pascal-S, Pascal in Pascal
NYU: First Ada implementation in SETL
Functional languages: SECD machine (stack, environment, control, dump), Gabbrielli 11.5

More formal versions (Gabbrielli 2.5): Still state-oriented, more abstract. Broad application to concurrency and type systems.

Big-Step: Similar in scope

Small-Step: More detail represented, as needed for concurrent languages