Programming Languages: An Active Learning Approach

Chapter 2 Specifying Syntax

Kent D. Lee

Specifying Syntax

- Once you've learned how to program in some language, learning a new programming language isn't all that hard. When learning a new language you need to know two things.
- Syntax refers to the words and symbols of a language and how to write the symbols down in the right order.
- Semantics is the word that is used when deriving meaning from what is written. The semantics of a program refers to what the program will do when it is executed.

Terminology

- **Syntax** is how things look
- **Semantics** is how things work (the meaning)

Terminology

Example 2.1

Apparently

```
a=b+c;
```

is correct C++ syntax. But is it really a correct statement?

1. Have `b` and `c` been declared as a type that allows the `+` operation?
2. Is `a` assignment compatible with the result of the expression `b+c`?
3. Do `b` and `c` have values?
4. Does the assignment statement have the proper form?

There are lots of questions that need to be answered about this assignment statement. Some questions could be answered sooner than others. When a C++ program is compiled it is translated from C++ to machine language as described in the previous chapter. Questions 1 and 2 are issues that can be answered when the C++ program is compiled. However, the answer to the third question above might not be known until the C++ program executes. The answers to questions 1 and 2 can be answered at *compile-time* and are called *static* semantic issues. The answer to question 3 is a *dynamic* issue and is probably not determinable until run-time. In some circumstances, the answer to question 3 might also be a static semantic issue. Question 4 is definitely a syntactic issue.

Terminology

- C++ terminals: **while**, **const**, **(**, **;**, **5**, **b**
- Terminal types are keywords, operators, numbers, identifiers, etc.

Terminology

- A **syntactic category** or **nonterminal** is a set of objects (strings) that will be defined in terms of symbols in the language (terminal and nonterminal symbols).
  - C++ nonterminals: <statement>, <expression>, <if-statement>, etc.
  - Syntactic categories define parts of a program like statements, expressions, declarations, and so on.
- A **metalanguage** is a higher-level language used to specify, discuss, describe, or analyze another language.

Backus Naur Form (BNF)

- Backus Naur Format (i.e. BNF) is a formal metalanguage for describing language syntax.
- BNF was used by John Backus to describe the syntax of Algol in 1963.

  $<$syntactic category$>$ ::= a string of terminals and nonterminals
  "::=" means "is composed of " (sometimes written as $\rightarrow$)

Backus Naur Form (BNF)

Example 2.2

BNF Examples from Java

<primitive type> ::= boolean

<primitive type> ::= char

Abbreviated

<primitive type> ::= boolean | char | byte | short | int | long | float | ...

<argument list> ::= <expression> | <argument list> , <expression>

<selection statement> ::=

```
    if ( <expression> ) <statement>
  | if ( <expression> ) <statement> else <statement>
  | switch ( <expression> ) <block>
```

<method declaration> ::=

```
  <modifiers> <type specifier> <method declarator>
  throws <method body>
| <modifiers> <type specifier> <method declarator>
  <method body>
| <type specifier> <method declarator> throws <method
  body>
| <type specifier> <method declarator> <method body>
```

The above description can be described in English as *the set of method declarations is the union of the sets of method declarations that explicitly throw an exception with those that don't explicitly throw an exception with or without modifiers attached to their definitions*. The BNF is much easier to understand and is not ambiguous like this English description.

The EWE Language

Example 2.3

Consider the C++ program fragment.

```
int a=0;
int b=5;
int c=b+1;
a=b*c;
cout << a;
```

The EWE code below implements the C++ program fragment above.

```
a := 0
b := 5
one := 1
c := b + one
a := b * c
writeInt(a)
halt
equ a M[0]    b M[1]    c M[2]    one M[3]
```

The EWE Language

Example 2.4

Here's another EWE program that computes the same thing as the C++ program fragment given above. This EWE program isn't quite as straightforward as the last one, but they do the same thing.

```
# int a=0;
R0:=0 # load 0 into R0
M[SP+12]:=R0
# int b = 5;
R1:=5 # load 5 into R1
M[SP+13]:=R1
# int c = b+1;
R2:=SP # b+1
R2:=M[R2+13] # load b into R2
R3:=1 # load 1 into R3
R2:=R2+R3
M[SP+14]:=R2
# a = b*c;
R4:=SP # b*c
```

```
R4:=M[R4+13] # load b into R4
R5:=SP
R5:=M[R5+14] # load c into R5
R4:=R4*R5
R6:=SP
M[R6+12]:=R4
R7:=M[SP+12]
writeInt(R7)
halt
equ SP M[10]   equ R0 M[0]   equ R1 M[0]
equ R2 M[0]    equ R3 M[1]   equ R4 M[0]
equ R5 M[1]    equ R6 M[1]   equ R7 M[0]
```

The EWE Language

- data memory locations specified by M[...]
- an instruction memory containing statements

EWE BNF

Listing 2.1: The EWE BNF

```
<eweprog> ::= <executable> <equates> EOF

<executable> ::=
<labeled instruction>
|  <labeled instruction> <executable>

<labeled instruction> ::=
Identifier ":" <labeled instruction>
|  <instr>

<instr> ::=
<memref> ":=" Integer
| <memref> ":=" String
  |  <memref> ":=" "PC" "+" Integer
  |  "PC" ":=" <memref>
|  <memref> ":=" <memref>
|  <memref> ":=" <memref> "+" <memref>
  |  <memref> ":=" <memref> "-" <memref>
  |  <memref> ":=" <memref> "*" <memref>
|  <memref> ":=" <memref> "/" <memref>
|  <memref> ":=" <memref> "%" <memref>
|  <memref> ":=" "M" "[" <memref> "+" Integer "]"
|  "M" "[" <memref> "+" Integer "]" ":=" <memref>
|  "readInt" "("<memref> ")"
|  "writeInt" "(" <memref> ")"
|  "readStr" "("<memref> "," <memref> ")"
|  "writeStr" "(" <memref> ")"
|  "goto" Integer
|  "goto" Identifier
|  "if" <memref> <condition> <memref> "then" "goto" Integer
|  "if" <memref> <condition> <memref> "then" "goto" Identifier
|  "halt"
|  "break"

<equates> ::=
null
|  "equ" Identifier "M" "[" Integer "]" <equates>

<memref> ::=
"M" "[" Integer "]"
|  Identifier

<condition> ::= ">=" | ">" | "<=" | "<" | "=" | "<>"
```

EWE BNF

☞ Practice 2.1

The following program is not a valid EWE program. Using the BNF for EWE list the problems with this program.

```
readln(A);
readln(B);
if A-B < 0 then
writeln(A)
else
writeln(B);
```

How could you rewrite this program so that it does what this program intends to do?

▸ View Solution

EWE BNF

☞ Practice 2.2

Write a EWE program to read a number from the keyboard and print out the sum of all the numbers from 1 to that number.

▸ View Solution

EWE BNF

Example 2.5

EWE is essentially an assembly language. It contains a few higher-level constructs, but very few. The EWE program given below upper cases all the characters in a string read from the keyboard. The simple way to write an assembly language program is to first write it in a high-level language. For instance, the program might look something like this in a C-like language.

```
s = input();
i = 0;
while s[i] != 0 {
  if ('a' <= s[i] && s[i] <= 'z')
    s[i] = s[i] - 'a' + 'A';
  i++;
}

printf("%s",s)
```

When writing the program in EWE you will want to program the opposite of any if-then or while loop conditions you wrote in the high-level language. This is because you are going to use a **goto** statement to assist in completing the code. If the condition is false in an if-then statement you will jump around the **then** part of the statement by jumping to code that is after the **then** part. The code below shows you the EWE code with the appropriate C code intermingled as comments. Comments in EWE begin with a pound sign (i.e. #).

```
zero:=0
one:=1
littlea := 97
littlez := 122
diff:=32
# s = input();
len:=100
readStr(s,len)
# i=0;
i:=100
# while s[i]!=0 {
loop: tmp:=M[i+0]
if tmp = zero then goto end
```

```
#   if ('a' <= s[i] && s[i] <= 'z')
if littlea > tmp then goto skip
if tmp > littlez then goto skip
#     s[i] = s[i] - 32;
tmp:=tmp-diff
M[i+0]:=tmp
skip:
#   i++;
i:=i+one
goto loop
# printf("%s",s)
end: writeStr(s)
halt

equ zero M[0] equ one M[1] equ littlea M[2]
equ littlez M[3] equ diff M[4] equ len M[5]
equ s M[100] equ tmp M[6] equ i M[7]
```

EWE BNF

☞ Practice 2.3

Write a EWE program that reads a list of numbers from the screen and prints them out in reverse order. In order to do this exercise you need to know something about indexed addressing (see the example above).
HINT: What kind of data structure lets you reverse the elements of a list?

▸ View Solution

Context-Free Grammars

- A context-free grammar is defined as a four tuple:

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$$

  where

  - $\mathcal{N}$ is a set of symbols called nonterminals or syntactic categories.
  - $\mathcal{T}$ is a set of symbols called terminals or tokens.
  - $\mathcal{P}$ is a set of productions of the form $n \rightarrow \alpha$ where n is a nonterminal and α is a string of terminals and nonterminals.
  - $\mathcal{S}$ is a special nonterminal called the start symbol of the grammar.

Context-Free Grammars

Example 2.6

A grammar for expressions in programs can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$

$\mathcal{T} = \{\mathit{identifier}, \mathit{number}, +, -, *, /, (,)\}$

$\mathcal{P}$ is defined by the set of productions

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T/F \mid F \\ F &\rightarrow (E) \mid \mathit{identifier} \mid \mathit{number} \end{aligned}$$

Derivations

- A *sentence* of a grammar is a string of tokens from the grammar.
- A *derivation* is a sequence of sentential forms that starts with the start symbol of the grammar and ends with the sentence you are trying to derive.
- A *sentential form* is a string of terminals and nonterminals from the grammar.
- In each step in the derivation, one nonterminal of a sentential form, call it A, is replaced by a string of terminals and nonterminals, β, where $A \rightarrow \beta$ is a production in the grammar.

Derivations

Example 2.7

Prove that the expression (5*x)+y is a member of the language defined by the grammar given in example 2.6 by constructing a derivation for it.
The derivation begins with the start symbol of the grammar and ends with the sentence.

$$E \Rightarrow \underline{E + T} \Rightarrow T + T \Rightarrow F + T \Rightarrow \underline{(E) + T} \Rightarrow (T) + T \Rightarrow \\ \underline{(T * F) + T} \Rightarrow (F * F) + T \Rightarrow (5 * F) + T \Rightarrow (5 * x) + T \Rightarrow \\ (5 * x) + F \Rightarrow \underline{(5 * x) + y}$$

The underlined parts are all examples of sentential forms.

Derivations

☞ Practice 2.4

Construct a derivation for the expression $4 + (a - b) * x$.

▸ View Solution

Types of Derivations

- Left-most derivation - Always replace the left-most nonterminal when going from one sentential form to the next in a derivation.
- Right-most derivation - Always replace the right-most nonterminal when going from one sentential form to the next in a derivation.

Types of Derivations

Example 2.8

The derivation of the sentence $(5 * x) + y$ in example 2.7 is a left-most derivation. A right-most derivation for the same sentence is:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y \Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

Types of Derivations

☞ Practice 2.5

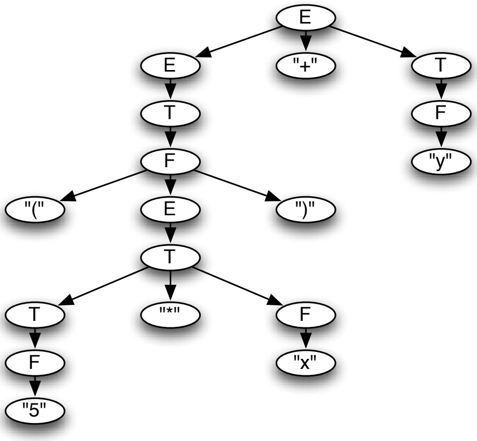
Construct a right-most derivation for the expression $x * y + z$.

▸ View Solution

Parse Trees

- A grammar for a language can be used to build a tree representing a sentence of the grammar.
- A parse tree is constructed with the start symbol of the grammar at the root of the tree.
- The children of each node in the tree must appear on the right hand side of a production with the parent on the left hand side of the same production.
- A program is syntactically valid if there is a parse tree for it using the given grammar.

A Parse Tree



Parse Trees

Example 2.9

The parse tree for the sentence derived in example 2.7 is depicted in figure 2. Notice the similarities between the derivation and the parse tree.

Parse Trees

☞ Practice 2.6

What does the parse tree look like for the right-most derivation of (5*x)+y?

▸ View Solution

Parse Trees

☞ Practice 2.7

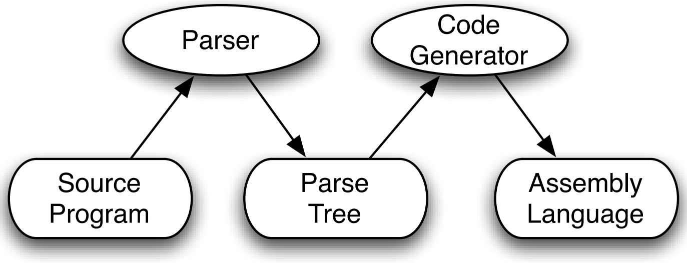
Construct a parse tree for the expression “4+(a-b)*x".
HINT: What has higher precedence, “+" or “*"? The grammar given above automatically makes “*" have higher precedence. Try it the other way and see why!

▸ View Solution

Parsing

- Parsing is the process of detecting whether a given string of tokens is a valid sentence of a grammar.
- A *parser* is a program that given a sentence, checks to see if the sentence is a member of the language of the given grammar.
  - A top-down parser starts with the root of the tree
  - A bottom-up parser starts with the leaves of the tree

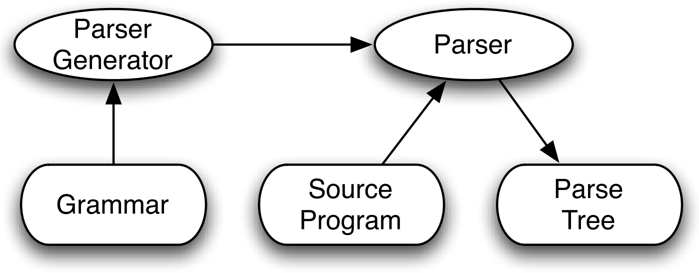
Flow of Data surrounding a Parser



Parser Generators

- A parser generator is a program that given a grammar, constructs a parser for the language specified by the grammar.
- Examples of parser generators are yacc and ml-yacc.

Flow of Data surrounding a Parser Generator



Other Forms of Grammars

- As a computer programmer you will likely learn at least one new language and probably a few during your career.
- A recent trend in programming languages is to develop domain specific languages.
- Programming language references almost always contain some kind of reference that describes the constructs of the language.
- A few examples of these grammar variations are given here to make you aware of notation that is often used in language references.

CBL (Cobol-like) Grammars

1. Optional elements are enclosed in brackets: [ ].
2. Alternate elements are vertically enclosed in braces: { }.
3. Optional alternates are vertically enclosed in brackets.
4. A repeated element is written once followed by an ellipsis: ...
5. Required <u>key words</u> are underlined; optional noise words are not.
6. Items supplied by the user are written as lower case or as syntactic categories from which an item may be taken.

CBL (Cobol-like) Grammars

Example 2.10

Here is the description of the COBOL ADD statement.

$$
\begin{array}{l}
\langle\text{Cobol Add statement}\rangle ::= \\
\qquad \text{ADD} \begin{Bmatrix} \text{identifier} \\ \text{number} \end{Bmatrix} \begin{bmatrix} \text{, identifier} \\ \text{, number} \end{bmatrix} \ldots\ \underline{\text{TO}} \\
\qquad\qquad \text{identifier } [\underline{\text{ROUNDED}}][\text{, identifier } [\underline{\text{ROUNDED}}]\ ] \ldots \\
\qquad\qquad [\text{ ON } \underline{\text{SIZE ERROR}} \ \langle\text{statement}\rangle\ ]
\end{array}
$$

One such add statement might be:

```
ADD A, 5 TO B ROUNDED, D
        ON SIZE ERROR PERFORM E-ROUTINE
```

Extended BNF (EBNF)

1. **item?** or **[item]** means item is optional.
2. **item*** or **{item}** means to take zero or more occurrences of an item.
3. **item+** means to take one or more occurrences of an item
4. Parentheses are used for grouping

Extended BNF (EBNF)

Example 2.11

Here is can example of method declarations in Java.

```
<method declaration> ::=
        <modifiers>? <type specifier>
                <method declarator> throws ? <method body>
```

Syntax Diagrams

1. A terminal is shown in a circle or oval.
2. A syntactic category is placed in a rectangle.
3. The concatenation of two objects is indicated by a flowline.
4. The aternation of two objects is shown by branching.
5. Repetition of objects is represented by a loop.

Syntax Diagrams

Example 2.12

Here are some descriptions of simple expressions in Pascal. Each of these different methods describe the same simple expressions in Pascal. Notice that some descriptions are more compact than the BNF. Each of them are unambiguous in their descriptions.
While BNF is less compact, it is the easiest to enter on a keyboard and for computer programs to read. There is a trade-off between computer readability and human readability that is at the center of many of our decisions about how to formally define programming languages.

BNF

<simple expr> ::=

    <term>

  | <sign> <term>

  | <simple expr> <adding operator> <term>

<sign> ::= “+" | “-"

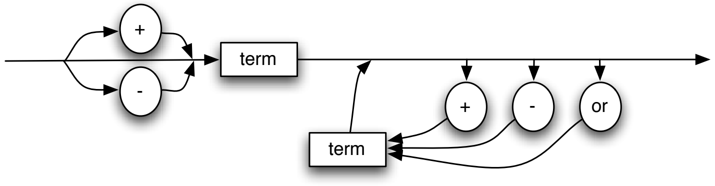
<adding operator> ::= “+" | “-" | “or"

CBL

<simple expr> ::=

$$\begin{bmatrix} + \\ - \end{bmatrix} \langle\text{term}\rangle \left[\begin{Bmatrix} + \\ - \\ \text{or} \end{Bmatrix} \langle\text{term}\rangle \right] \ldots$$

EBNF
<simple expr> ::= [<sign>] <term> {<adding operator> <term>}
Syntax Diagram



Syntax Diagrams

☞ Practice 2.8

According to the syntactic specification in example 2.12, which of these terminal strings are simple expressions, assuming that a, b, and c are legal terms:

1. a+b-c
2. -a or b+c
3. b - - c

▸ View Solution

Ambiguous Grammars

Example 2.13

The classic example is nested if-then-else statements. Consider the following Pascal statement:

```
if a<b then
if b<c then
writeln("a<c")
else
writeln("?")
```

Which *if* statement does the *else* go with? It's not entirely clear. According to the grammar, it could go with either. This means there is some ambiguity in the grammar for Pascal. This resolved by deciding the *else* should go with the nearest *if*. In a bottom-up parser this is called a shift/reduce conflict. In this case it is resolved by shifting instead of reducing.

Ambiguous Grammars

☞ Practice 2.9

Consider the expression grammar

> <expr> ::= identifier | <expr> <operator> <expr>
> <operator>::= “+" | “*"

Consider the terminal string `a * b + c`.
Give two parse trees for this expression. This ambiguity could be resolved by specifying a precedence of operators in the grammar. However, there are better methods than specifying precedence. Precedence of operators can also be specified by introducing extra productions. See example 2.6 on page 24 for a better way of writing the grammar for this language.

▸ View Solution

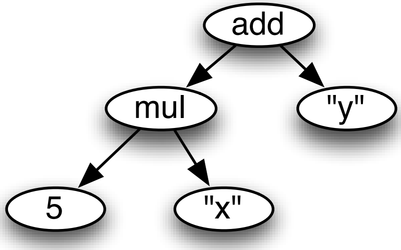
Abstract Syntax Trees

- An abstract syntax tree is like a parse tree except that non-essential information is removed.
  - Nonterminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.
  - Unit productions in the tree are collapsed.

Abstract Syntax Trees

Example 2.14

For example, the parse tree from figure 2 on page 32 can be represented by the following abstract syntax tree.



Abstract Syntax Trees

☞ Practice 2.10

What does the abstract syntax tree of 4+(a-b)*x look like?

▸ View Solution

Infix, Postfix, and Prefix Expressions

- The abstract syntax tree in example 2.14 represents a computation.
- We can recover the infix expression it represents by doing an inorder traversal of the abstract syntax tree.

```
Inorder_traverse(t a tree)
If t is an empty tree, do nothing
inorder_traverse(left subtree of t)
print the data of the root node in the tree t
inorder_traverse(right subtree of t)
```

Infix, Postfix, and Prefix Expressions

☞ Practice 2.11

Assume there is a BTNode class in your favorite object-oriented language with appropriate constructors, and getData, getLeft, and getRight member functions which return the data at a node, the left subtree, and the right subtree respectively. Write some code to implement this inorder traversal of a tree. Assume the AST in example 2.14 is given as input. What is the output? Is there anything wrong?

▸ View Solution

Infix, Postfix, and Prefix Expressions

☞ Practice 2.12

How does this code change to do a postorder traversal? What is the output given the tree in example 2.14.

▸ View Solution

Limitations of Syntactic Definitions

Example 2.15

These are all context-sensitive issues.

- In an array declaration in C++, the array size must be a nonnegative value.
- Operands for the && operation must be boolean in Java.
- In a method definition, the return value must be compatible with the return type in the method declaration.
- When a method is called, the actual parameters must match the formal parameter types.

Exercises

1. What does the word syntax refer to? How does it differ from semantics?
2. What is a token?
3. What is a nonterminal?
4. What does BNF stand for? What is its purpose?
5. Describe what the rules in lines 35-37 of the EWE BNF on page 17 mean. Answer this in some detail. Saying they define equates is not enough.
6. According to the EWE BNF, how many labels can an instruction have?
7. Given the grammar in example 2.6, derive the sentence (4+5)*3.
8. Draw a parse tree for the sentence (4+5)*3.
9. What kind of derivation does a top-down parser construct?

10. What would the abstract syntax tree for (4+5)*3 look like?
11. Describe how you might evaluate the abstract syntax tree of an expression to get a result? Write out your algorithm in English that describes how this might be done.
12. List four context-sensitive conditions in your favorite language.
13. Write a EWE program that prompts the user to enter three numbers and prints the max of the three numbers to the screen. Think about this before attempting to write it. It might be harder than you think at first.
14. Write a EWE program that prompts the user to enter a string and prints the reverse of that string to the screen.
15. Write a EWE program that prompts the user to enter a string and prints the string back to the screen with the first letter of each word upper cased.

16. Write a EWE program that asks the user to enter a number and prints either the square root of the number if it is an integer or the two integers the square root falls between if it is not an integer result. EWE does not operate on real numbers. It only works with integers and strings.
17. Using the EWE interpreter, write a program that prompts the user for a number and prints the factorial of that number.

Solution to Practice Problem 2.1

Here is a correct version of the program. As you can see there are several things wrong with the original.

```
readInt(A)
readInt(B)
C := A - B
zero := 0
if C >= zero then goto pastwrtA
writeInt(A)
goto end
pastwrtA:
writeInt(B)
end:
halt
equ A M[0]    equ B M[1]    equ C M[2]    equ zero M[3]
```

Go Back to Practice Problem

Solution to Practice Problem 2.2

The easiest way to write EWE programs is to write in a language like Java or Python and then translate the code to EWE. Reverse any relational operators to make the translation (see the previous exercise). So for instance, a less than operator becomes greater or equal when translated into EWE. Here is a Python version of the program.

```
n = input("Enter a postive integer:")
sum = 0
for x in range(n+1):
  sum = sum + x

print "The sum is", sum
```

And here is a EWE version.

```
readInt(n)
sum := 0
one := 1
x := 1
loop:
  if x > n then goto end
    sum := sum + x
    x := x + one
  goto loop
end:
writeInt(sum)
halt
equ sum M[0]   equ one M[1]   equ x M[2]   equ n M[3]
```

If you think hard about this problem there is a simpler version that is about three lines long. You have to find the formula that computes the sum of the first *n* integers, though.

▸ Go Back to Practice Problem

Solution to Practice Problem 2.3

You need to use indexed addressing to create a stack.

```
SP := 100
hundred := 100
zero := 0
one := 1
readloop:
  readInt(x)
  if x = zero then goto printloop
  M[SP+0] := x
  SP := SP + one
  goto readloop
printloop:
  SP := SP - one
  if SP < hundred then goto end
  x := M[SP+0]
  writeInt(x)
  goto printloop
end:
  halt
equ SP M[0]   equ hundred M[1]   equ x M[3]
equ zero M[4]   equ one M[5]
```

▸ Go Back to Practice Problem

Solution to Practice Problem 2.4

This is a left-most derivation of the expression.
$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow 4+T \Rightarrow 4+T*F \Rightarrow 4+F*F \Rightarrow 4+(E)*F \Rightarrow 4+(E-T)*F \Rightarrow 4+(T-T)*F \Rightarrow 4+(F-T)*F \Rightarrow 4+(a-T)*F \Rightarrow 4+(a-F)*F \Rightarrow 4+(a-b)*F \Rightarrow 4+(a-b)*x$

Go Back to Practice Problem

Solution to Practice Problem 2.5

This is a right-most derivation of the expression.
$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + z \Rightarrow T + z \Rightarrow T * F + z \Rightarrow$
$T * y + z \Rightarrow F * y + z \Rightarrow x * y + z$

Go Back to Practice Problem

Solution to Practice Problem 2.6

Exactly like the parse tree for any other derivation of (5*x)+y. There is only one parse tree for the expression given this grammar.

▸ Go Back to Practice Problem

Solution to Practice Problem 2.7

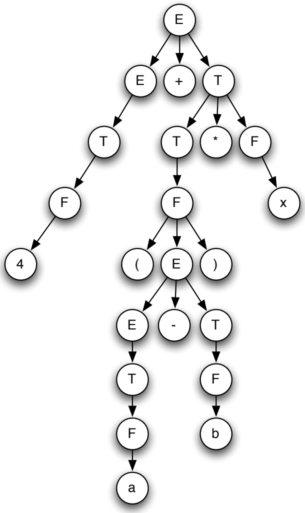


Figure: The parse tree for practice problem 2.7

Go Back to Practice Problem

Solution to Practice Problem 2.8

1. a+b-c is a valid simple expression.
2. -a or b + c is a valid simple expression.
3. b - - c is not a simple expression.

Go Back to Practice Problem

Solution to Practice Problem 2.9

In this problem we have a choice of putting the * or the + operator closer to the top of the tree. This will give us two different trees depending on which we choose.

▸ Go Back to Practice Problem

Solution to Practice Problem 2.10

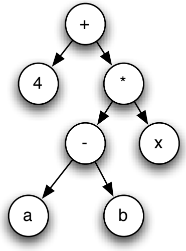


Figure: The parse tree for practice problem 2.10

Go Back to Practice Problem

Solution to Practice Problem 2.11

```
void inordertraverse(BTNode root) {
  if (root == nil) then return;

  inordertraverse(root.getLeft());
  System.out.println(root.getData()+ " ");
  inordertraverse(root.getRight());

}
```

The output would be 5 + x * y. The traversal has thrown away the parentheses. If parens are needed the inorder traversal code could be modified to produce a fully parenthesized expression.

Go Back to Practice Problem

Solution to Practice Problem 2.12

The println statement would move to the last line of the function. The postorder output would be 5 x + y *. No parens are needed in a postfix expression.

▸ Go Back to Practice Problem