Syntax Analysis

Symbol Table

A Symbol Table is a container for maintaining information extracted from a source file. Using a Symbol Table is a simpler alternative to using a Syntax Tree. At the most basic level a Symbol Table is just an associative map (key → value). Beyond a simple associative map a Symbol Table organizes information about defined constructs in a programming language and provides search mechanizes relevant to constructing a Compiler.

The Symbol Table for your Compiler will be populated with data during Syntax Analysis. You are not complete with Syntax Analysis until you have populated your Symbol Table.

See the Symbol Table Examples document.

Syntax Analysis

During Syntax Analysis your Compiler receives a sequence of Tokens from the Lexical Analyzer (Scanner) and verifies that each of the Tokens can be generated by the Grammar for the Source Language.

Various methods exist for parsing a Source file we will focus on one very generalized method: Recursive-Descent Parsing

Recursive-Descent Parsing

Recursive-Descent Parsing is a Top-Down parsing technique that may involve backtracking.

Top-Down Parsing produces a Parse Tree from the Top Down, meaning from the Root to Leaves.

Backtracking is a search strategy used in searching Trees that involves returning to a previously visited node in a Tree to search an alternative path (child) that originates at the nodes.

The Six Forms for Recursive Descent Parsing

Form 1: Sequence of terminals

NT → a b c

```
Void NT() {
     if (ct.type() != Token.a.type)
          genError();
     ct.next();
     if (ct.type() != Token.b.type)
          genError();
     ct.next();
     if (ct.type() != Token.c.type)
          genError();
     ct.next();
}

Matching on lexemes vs. Token type
if (ct.lexeme() != Token.a.lexeme)
     genError();
```

Form 2: Sequence of Non-terminals

NT → A B C

```
Void NT() {
     A();
     B();
     C();
}
```

Form 3: Mix of Terminals & Non-terminals

NT → a X Y b Z

```
Void NT() {
     if(ct.type() != Token.a.type)
          genError();
     ct.next();
     X();
     Y();
     if(ct.type() != Token.b.type)
          genError();
     ct.next();
     Z();
}
```

Form 4: Something is Optional

NT $\rightarrow$ X [a Y] Z

```
Void NT() {
     X();

     if(ct.type() == Token.a.type) {
          ct.next();
          Y();
     }
     Z();
}
```

For this to work the first non-terminal of Z must not be terminal “a”.
If it is NOT then you must look ahead by more than one Token:

```
     ct.peek();
```

Form 5: There is a choice

Case 5.1 NT $\rightarrow$ a X | b Y | c Z

```
Void NT() {
     switch(ct.type()) {
     case Token.a.type:
          ct.next();
          X();
          break;
     case Token.b.type:
          ct.next();
          Y();
          break;
     case Token.c.type:
          ct.next();
          Z();
          break;
     default:
          genError();
     }
}
```

Case 5.2   NT → X | Y Z

We must know the first terminal symbol of each Non-terminal X, Y and Z to encode this
Otherwise we must support backtracking.

```
Void NT() {
	switch(ct.type()) {
	case Token.X.first.type:
		// Don't proceed to ct.next();
		X();
		break;
	case Token.Y.first.type:
		// Don't proceed to ct.next();
		Y();
		break;
	case Token.Z.first.type:
		// Don't proceed to ct.next();
		Z();
		break;
	default:
		genError();
	}
}
```

Form 6: Zero or more Occurrences

NT → { a N }

```
Void NT() {
	while(ct.type == Token.a.type) {
		ct.next();
		N();
	}
}
```