Chapter 3

Syntactic Analysis I

- The Syntactic Analyzer, or Parser, is the heart of the front end of the compiler.
- The parser's main task is to analyze the structure of the program and its component statements.
- Our principle resource in Parser Design is the theory of Formal Languages.
- We will use and study context free grammars (They cannot handle definition before use, but we can get around this other ways)

1. Grammars

- **Informal Definition** -- a finite set of rules for generating an infinite set of sentences.
- **Def:** Generative Grammar: this type of grammar builds a sentence in a series of steps, refining each step, to go from an abstract to a concrete sentence.

- **Def:** Parse Tree: a tree that represents the analysis/structure of a sentence (following the refinement steps used by a generative grammar to build it.

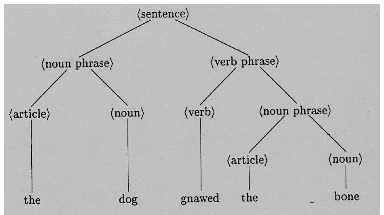

Figure 3.1

- **Def:** Productions/Re-Write Rules: rules that explain how to refine the steps of a generative grammar.

- **Def:** Terminals: the actual words in the language.

- **Def:** Non-Terminals: Symbols not in the language, but part of the refinement process.

1.1 Syntax and Semantics

- Syntax deals with the way a sentence is put together.

- Semantics deals with what the sentence means.

- There are sentences that are grammatically correct that do not make any sense.

- There are things that make sense that are not grammatically correct.
- The compiler will check for syntactical correctness, yet it is the programmers responsibility (usually during debugging) to make sure it makes sense.

1.2 Grammars: Formal Definition

- G = (T,N,S,R)
  - T = set of Terminals
  - N = set of Non-Terminals
  - S = Start Symbol (element of N)
  - R = Set of Rewrite Rules (α -> β)

- In your rewrite rules, if α is a single non-terminal the language is Context-Free.
- BNF stands for Backus-Naur Form
  - ::= is used in place of ->
  - in extended BNF { } is equivalent to ( )*

1.3 Parse Trees and Derivations

- a1 => a2 -- string a1 is changed to string a2 via 1 rewrite rule.
- $\alpha =^{*}=> \beta$ -- 0 or more re-write rules
- sentential forms -- the strings appearing in various derivation steps
- L(G) = { w | S $=_G{}^{*}=>$ w}

1.4 Rightmost and Leftmost Derivations

- Which non-terminal do you rewrite-expand when there is more than one to choose from.
  - If you always select the rightmost NonTerminal to expand, it is a Rightmost Derivation.
- Leftmost and Rightmost derivations are unique.

- **Def:** any sentential form occurring in a leftmost {rightmost} derivation is termed a left {right} sentential form.
- Some parsers construct leftmost derivations and others rightmost, so it is important to understand the difference.

- Given (pg 72) G_E = (T, N, S, R)
  - T = { i, +, -, *, /, (,)},
  - N = {E}
  - S = E
  - R = {
    - E -> E + E   E -> E - E
    - E -> E * E   E -> E / E
    - E -> (E)   E -> i }
- consider: (i+i)/(i-i)

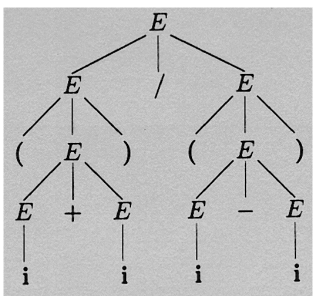

Figure 3.4(c)

1.5 Ambiguous Grammars

- Given (pg 72) G_E = (T, N, S, R)
  - T = { i, +, -, *, /, (,)},
  - N = {E}
  - S = E
  - R = {
    - E -> E + E   E -> E - E
    - E -> E * E   E -> E / E
    - E -> (E)   E -> i }
- consider: i + i * i

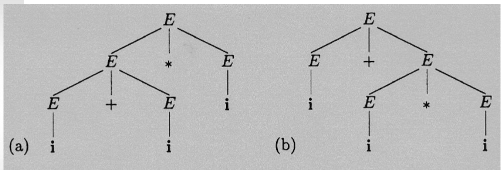

Figure 3.5

- a grammar in which it is possible to parse even one sentence in two or more different ways is ambiguous
- A language for which no unambiguous grammar exists is said to be inherently ambiguous

- The previous example is "fixed" by operator-precedence rules,
- or re-write the grammar
  - E -> E + T | E - T | T
  - T -> T * F | T / F | F
  - F -> (E) | i
- Try: i+i*i

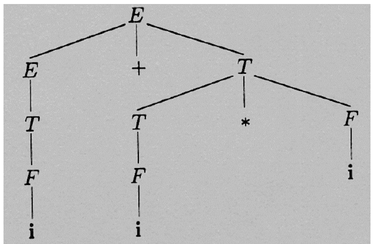

Figure 3.6

1.6 The Chomsky Hierarchy (from the outside in)

- Type 0 grammars
  - γAδ -> γβδ
  - these are called phrase structured, or unrestricted grammars.
  - It takes a Turing Machine to recognize these types of languages.

- Type 1 grammars
  - γAδ -> γβδ
    β != ε
  - therefore the sentential form never gets shorter.
  - Context Sensitive Grammars.
  - Recognized by a simpler Turing machine [linear bounded automata (lba)]

- Type 2 grammars:
  - A -> β
  - Context Free Grammars
  - it takes a stack automaton to recognize CFG's (FSA with temporary storage)
  - Nondeterministic Stack Automaton cannot be mapped to a DSA, but all the languages we will look at will be DSA's

- Type 3 grammars
  - The Right Hand Side may be
    - a single terminal
    - a single non-terminal followed by a single terminal.
  - Regular Grammars
  - Recognized by FSA's

1.7 Some Context-Free and Non-Context-Free Languages

- Example 1:
  - S -> S S
  - | (S)
  - | ( )
  - This is Context Free.

- Example 2:
  - $a^n b^n c^n$
- this is NOT Context Free.

- Example 3:
  - S -> aSBC
  - S -> abC
  - CB -> BC
  - bB -> bb
  - bC -> bc
  - cC -> cc
- This is a Context Sensitive Grammar

- L_2 = {wcw| w in (T-c)*} is NOT a Context Free Grammar.

1.8 More about the Chomsky Hierarchy

- There is a close relationship between the productions in a CFG and the corresponding computations to be carried out by the program being parsed.
- This is the basis of Syntax-directed translation which we use to generate intermediate code.

2. Top-Down parsers

- The top-down parser must start at the root of the tree and determine, from the token stream, how to grow the parse tree that results in the observed tokens.
- This approach runs into several problems, which we will now deal with.

2.1 Left Recursion

- Productions of the form (A->Aα) are left recursive.
- No Top-Down parser can handle left recursive grammars.
- Therefore we must re-write the grammar and eliminate both direct and indirect left recursion.

- How to eliminate Left Recursion (direct)
  - Given:
    - A -> Aα_1 | Aα_2 | Aα_3 | …
    - A -> δ_1 | δ_2 | δ_3 | ...
  - Introduce A'
    - A -> δ_1 A' | δ_2 A' | δ_3 A' | ...
    - A' -> ε | α_1 A' | α_2 A' | α_3 A' | ...
  - Example:
    - S -> Sa | b
    - **Becomes**
    - S -> bS'
    - S' -> ε | a S'

- How to remove ALL Left Recursion.
  - 1.Sort the nonterminals
  - 2.for each nonterminal
    - if B -> Aβ
    - and A -> γ_1 | γ_2 | γ_3 | ...
    - then B -> $\gamma_1\beta$ | $\gamma_2\beta$ | $\gamma_3\beta$ | ...
  - 3.After all done, remove immediate left recursion.

- Example:
  - S -> aA | b | cS
  - A -> Sd | e

  becomes

  - S -> aA | b | cS
  - A -> aAd | bd | cSd | e
- note: the S in A(3) -> but it is NOT left recursion

2.2 Backtracking

- One way to carry out a top-down parse is simply to have the parser try all applicable productions exhaustively until it finds a tree.
- This is sometimes called the brute force method.
- It is similar to depth-first search of a graph
- Tokens may have to be put back on the input stream

- Given a grammar:
  - S -> ee | bAc | bAe
  - A -> d | cA
- A Backtracking algorithm will not work properly with this grammar.
- Example: input string is bcde
  - When you see a b you select S -> bAc
  - This is wrong since the last letter is e not c

- The solution is Left Factorization.
  - **Def:** Left Factorization: -- create a new non-terminal for a unique right part of a left factorable production.
- Left Factor the grammar given previously.
  - S -> ee | bAQ
  - Q -> c | e
  - A -> d | cA

3. Recursive-Descent Parsing

- There is one function for each non-terminal these functions try each production and call other functions for non-terminals.
- The stack is invisible for CFG's
- The problem is -- a new grammar requires new code.

- **Example:**
  - S -> bA | c
  - A -> dSd | e
- **Code:**

```
function S: boolean;
 begin
   S := true;
   if token_is ('b') then
      if A then
         writeln('S --> bA')
      else
         S := false;
   else
      if token_is ('c') then
         writeln ('S --> c')
      else
         begin
         error ('S');
         S := false
         end
end; { S }
```



```
function A: boolean
begin
  A := true;
  if token_is ('d') then
    begin
    if S then
      if token_is ('d') then
       writeln('A --> dSd');
      else
        begin
          error ('A');
           A := false
        end
else
      A := false
   end
 else
   if token_is ('e') then
     writeln ('A --> e')
   else
     begin
     error ('A');
     A := false
     end
end; { A }
```

- Input String: bdcd

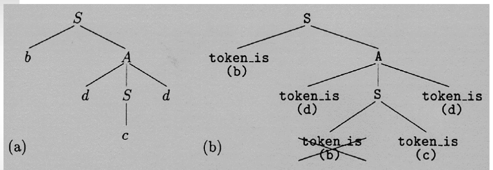

Figure 3.13

4. Predictive Parsers

- The goal of a predictive parser is to know which characters on the input string trigger which productions in building the parse tree.
- Backtracking can be avoided if the parser had the ability to look ahead in the grammar so as to anticipate what terminals are derivable (by leftmost derivations) from each of the various nonterminals on the RHS.

- First (α)
  (you construct first() of RHS's)
  - 1.if α begins with a terminal x,
    - then first(α) = x.
  - 2.if α =*=> ε,
    - then first(α) includes ε.
  - 3.First(ε) = {ε}.
  - 4.if α begins with a nonterminal A,
    - then first(α) includes first(A) - {ε}

- Follow(α)
  - 1.if A is the start symbol,
    - then put the end marker $ into follow(A).
  - 2.for each production with A on the right hand side
    Q -> xAy
    - 1.if y begins with a terminal q,
      - q is in follow(A).
    - 2.else follow(A) includes first(y)-ε.
    - 3.if y = ε, or y is nullable (y =*=> ε)
      - then add follow(Q) to follow(A).

- Grammar:
  - E -> T Q
  - Q -> + T Q | - T Q | epsilon
  - T -> F R
  - R -> * F R | / F R | epsilon
  - F -> (E) | I
- Construction of First and Follow Sets:

- First(E) = First(T) = First(F) = {i,(}
- First(Q) = {+,-,ε}
- First(R) = {*,/,ε}
- Follow(E) = {$,)}
- Follow(Q) = Follow(E) = {$,)}
- Follow(T) = First(Q) - ε + Follow(E)
  {+,-} + {),$}
- Follow(R) = Follow(T)
- Follow(F) = First(R) - ε + Follow(T)
  {*,/} + {+,-,),$}

- LL(1) Grammars
  - In a predictive parser, Follow tells us when to use the epsilon productions.
  - **Def:** LL(1) -- Left to Right Scan of the tokens, Leftmost derivation, 1 token lookahead.

- For a grammar to be LL(1), we require that for every pair of productions A -> alpha|beta
  - 1.First(alpha)-epsilon and First(beta)-epsilon must be disjoint.
  - 2.if alpha is nullable, then First(beta) and Follow(A) must be disjoint.
  - if rule 1 is violated, we may not know which right hand side to choose
  - if rule 2 is violated, we may not know when to choose Beta or epsilon.

4.1 A Predictive Recursive-Descent Parser

- The book builds a predictive recursive-descent parser for
  - E -> E + T | T
  - T -> T * F | F
  - F -> (E) | I
- First step is -- Remove Left Recursion

4.2 Table-Driven Predictive Parsers

- Grammar
  - E -> E + T | E - T | T
  - T -> T * F | T / F | F
  - F -> (E) | I
- Step 1: Eliminate Left Recursion.

- Grammar without left recursion
  - E -> T Q
  - Q -> + T Q | - T Q | epsilon
  - T -> F R
  - R -> * F R | / F R | epsilon
  - F -> (E) | I
- It is easier to show you the table, and how it is used first, and to show how the table is constructed afterward.

- Table

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ
F	i					(E)		

- Driver Algorithm:
  - Push $ onto the stack
  - Put a similar end marker on the end of the string.
  - Push the start symbol onto the stack.

- While (stack not empty do)
  - Let x = top of stack and a = incoming token.
  - If x is in T (a terminal)
  - if x == a then pop x and goto next input token
  - else error
  - else (nonterminal)
  - if Table[x,a]
  - pop x
  - push Table[x,a] onto stack in reverse order
  - else error
- It is a successful parse if the stack is empty and the input is used up.

■ Example 1: (i+i)*i (pg 108)

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		+TQ	−TQ				ε	ε
T	FR					FR		
R		ε	ε	*FR	/FR		ε	ε
F	i					(E)		

$E	(i + i)*i$	E → TQ	E ⇒ TQ
$QT	(i + i)*i$		

$QT	(i + i)*i$	T → FR	⇒ FRQ
$QRF	(i + i)*i$	F → (E)	⇒ (E)RQ
$QR)E(	(i + i)*i$		

$QR)E	i + i) * i$	E → TQ	⇒ (TQ)RQ
$QR)QT	i + i) * i$	T → FR	⇒ (FRQ)RQ
$QR)QRF	i + i) * i$	F → i	⇒ (iRQ)RQ
$QR)QRi	i + i) * i$	[pop and go to next token]	
$QR)QR	+i)*i$	R → ε	⇒ (iQ)RQ

$QR)Q	+ i)*i$	Q → +TQ	⇒ (i+TQ)RQ
$QR)QT+	+ i)*i$	[pop and go to next token]	
$QR)QT	i)*i$	T → FR	⇒ (i+FRQ)RQ
$QR)QRF	i)*i$	F →i	⇒ (i+iRQ)RQ
$QR)QRi	i)*i$	[pop and go to next token]	
$QR)QR	)*i$	R → ε	⇒ (i+iQ)RQ
$QR)Q	)*i$	Q → ε	⇒ (i+i)RQ
$QR)	)*i$	[pop and go to next token]	
$QR	*i$	R →*FR	⇒ (i+i)*FRQ
$QRF*	*i$	[pop and go to next token]	
$QRF	i$	F →i	⇒ (i+i)*iRQ
$QRi	$	[pop and go to next token]	
$QR	$	R → ε	⇒ (i+i)*iQ
$Q	$	Q → ε	⇒ (i+i)*i
$	$	[pop and go to next token]	

- Example 2: (i*) (pg 109)

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ
F	i					(E)		

Stack	Input	Production	Derivation
$E	(i*)$	$E \to TQ$	$E \Rightarrow TQ$
$QT	(i*)$	$T \to FR$	$\Rightarrow FRQ$
$QRF	(i*)$	$F \to (E)$	$\Rightarrow (E)RQ$
$QR)E(	(i*)$	[pop and go to next token]	
$QR)E	i*)$	$E \to TQ$	$\Rightarrow (TQ)RQ$
$QR)QT	i*)$	$T \to FR$	$\Rightarrow (FRQ)RQ$
$QR)QRF	i*)$	$F \to$ i	$\Rightarrow$ (iRQ)RQ
$QR)QRi	i*)$	[pop and go to next token]	
$QR)QR	*)$	$R \to *FR$	$\Rightarrow$ (i$*FRQ$)RQ
$QR)QRF*	*)$	[pop and go to next token]	
$QR)QRF	)$	***Error: no table entry for [F,)].	

4.3 Constructing the Predictive Parser Table

- Go through all the productions.
  X -> β is your typical production.
  - 1.For all terminals a in First(β), except ε, Table[X,a] = β.
  - 2.If β = ε, or if ε is in first(β) then For ALL a in Follow(X), Table[X,a] = ε.
- So, Construct First and Follow for all Left and right hand sides.

4.4 Conflicts

- A conflict occurs if there is more than 1 entry in a table slot. This can sometimes be fixed by Left Factoring, ...

- If a grammar is LL(1) there will not be multiple entries.

5. Summary

- Left Recursion
- Left Factorization
- First (A)
- Follow (A)
- Predictive Parsers (table driven)