2章　コンパイラの部品の構成
3章　アプローチ(4章へ)の準備
　　終端記号と非終端記号の違い
非終端記号　まだ何か入りますよ
終端記号　決められてる
follow集合　終端記号直前の非終端記号の集合
4章　ソースコードをコンピュータにとって意味のある最小単位(字句やtokenと呼ぶ)の集まりへと変換するもの。
5章　token列をコンピュータにとって構造が理解できるようにすること。
上向き　人間がやりやすい　下向き　コンピュータがやりやすい

6章　定義されていない名前の保存

7章　ソースコードの構造に誤りがあった時、指摘できる。
(8章　インタプリタ)
9章　抽象構文木から目的の低級言語を生成する。
10章　自動翻訳されたもののメモリ効率などを自動的に向上させる。

字句解析
まとめ

文字読み取り…　字句解析の手続きから呼ばれる。
　　原子プログラムの中の次の文字を返す
そしてコンパイラが読み込んだ文字の集まりの中から実際に有効な文字だけを次々と読みだしてくれる手続きを作ることで、以後の処理でカードの区切りなどを意識しないで済む
字句読み取り
コンパイラの処理場の最小単位の読み取り。状態遷移図を利用すると考えやすい。

文献
http://home.a00.itscom.net/hatada/lp/lex/lexical-analysis.html
ソースコードをプログラム上の最小単位に分割すること
最小単位を字句(token)と呼ぶ。例えば、
　　while (abc > 10) { abc = abc - 1; }
の場合、"while", "(", "abc", ">", "10", ")", "{", "abc", "=", "abc", "-", "1", ";", "}" に分解される。

構文解析

http://www.weblio.jp/content/%E6%A7%8B%E6%96%87%E8%A7%A3%E6%9E%90

単語や字句で構成される文を、定義された文法に従って解釈し、文の構造を明確にすることである。

構文解析の結果、構文解析木が得られる。

簡単なコンパイラにおいては三つ組やポーランド記法を使用する。

例えば、A := B*C + D/E
と言う式を、次のように表すことができる
:= (A,+(*(B,C), /(D,E)))
 ここからカッコと「,」を全て取り除くと
ABC*DE/+:=
このような表現形式をポーランド記法という
演算を読み込む際、終端記号(演算子 +や-)の優先順位を定義しなければならない。
文の構文解析に当たって、その文の前後を特殊な記号で囲う。
そして、解析木の一部分を解析、置き換えながら進む。
演算子順位には関数がある
演算子の種類が多くなると大きな行列になるので
>=<を決める
順位関数が存在することで順位行列の場所を節約できる

本を読んだまとめ
4 字句解析の論理的展開
字句解析の手順は以下の通りになっている。
文字読み取り→字句読み取り
字句読み取りの中に
正規表現→非決定性有限オートマトン→決定性有限オートマトン→有限オートマトンの最小化

文字読み取りでは,コンパイラが読み込んだ文字の集まりの中から有効な文字だけを次々と読み出すことである。
字句読み取りは名前、数、演算子等原始プログラム上のひとまとまりの文字の列で以後のコンパイラの最小単位となる字句を読み取ることである。
字句読み取りでは以下の順序で行われる
正規表現で字句の文法を決める。
バッカス記法で構文則を書き、構文図式に直す。
状態遷移図に直す
状態遷移図においてある状態から複数の行き先がある事や、空の遷移がある事を無くす。
こうすることにより、どのような文字を読み込んだのかが分かる決定性有限オートマトンが完成する。
最後に重なっている状態をまとめる

4章を簡単にしてみた。ただ、目的がわからない。
順序をまとめる形になった。
述べていることは方法論である。

matome
2章
コンパイラは以下の動きで動く
1.読み込み… 原始プログラムを通常レコード単位で読み込む。
2.字句解析… 読み込みによって読み込まれた文字を1字ごとに基本的要素ごとに取り出す。
また、取り出した文字が変数名か定数か演算子かの判別もする。
3.構文解析… プログラムのどの部分が文法のどの規則に対応するかを解析して、プログラムが文法的に正しいかどうか判定する。
中間語が生成される。その形は解析木である。
4.中間語生成
構文解析によって得られた解析木を実行すべき演算の列を作り出す。
5.最適化
目的プログラムを実行時効率の良いものとすること。
6.目的コード生成
中間語のプログラムから目的プログラムを生成する。
尚、2~6に関しては、変数名、定数、などの各種情報が入っている表と連携しながら、コンパイラが実行する。

4章
字句解析とは何か，そして字句解析をソースコードまで落とし込む。
4.字句解析は文字読み取りと字句読み取りに分けることができる。
文字読み取りでは空白も含めた文字を読み取る。
そして字句読み取りで読み取った文字を(if等の)最小単位の字句に分割する。

4.1 文字読み取り
コンパイラが原始プログラムを読み込み、読み込んだ文字の集まりの中から実際に有効な文字だけを次々と読み出すことである。

4.2字句読み取り
上記の様に字句解析に置いては読み取った字句から、意味のある最小の文字を読み出すものである.
字句読み取りの手続きは、状態遷移図を使うと考え安くなる.
字句は
<名前>→<英字>{<英字>|<数字>}という構文則で定義される名前を読み取る構文即は、
→0→1→2となる。
↑←↓
状態遷移図からプログラムを作ることが出来る.
書く状態ごとにプログラムを作る.
なお、プログラム終了の判断は1文字先読みしていないと分からないため、常に1文字先読みした状態で，プログラムは動く.
正規表現について。
字句読み取りのプログラムは字句の文法が正規文法であるなら，機械的に得ることが出来る.
正規表現と呼ばれる方法で出来，正規表現に置いて字句読み取り手続きはDFAと呼ばれている.

DFAのアルゴリズムをプログラムかすれば字句読み取り手続きの自動化が出来る.

実際に具体例を用いて説明する.

字句解析は文字読み取りと字句読み取りに分けることができる。
文字読み取りでは空白も含めた文字を読み取る。
そして読み取った文字を(if等の)最小単位の字句に分割する。
また、実際の字句解析において、読み込んだ文字をソースコードに落とすという流れ記述するために便利な以下の手順が一般的である
<バッカス記述>→正規表現→非有限オートマトン→有限オートマトン→ソースコードである。
しかし、私達はNFA以降、利用しない。
正規表現とは
正規表現とは文字列の中から探す文字を決める事である。
また、正規表現をA={a1,a2,...,an}とすると、
正規表現の定義は以下の様になる。
(i)εは正規表現である。
(ii)aiは正規表現である。
(iii)RとSが正規表現ならば、R|S,RS,{R}も正規表現である。

さらに、正規表現Rの値を[R]と書くと正規表現は以下の様に定義される。
(a) [ε]={ε}
(b) [ai]={ai}
(c) [R|S]=[R]U[S]
(d) [RS]=[R][S]
(e) [{R}]=[R]*

これが正規表現である。

bnf 文脈自由文法の定義方法である。
正規表現で表された文字列に名前を付けるのである。
文脈自由文法

Ruby Python にする
Ruby bnf　Python bnf から字句解析はこうだ　例を出す