情報処理演習
準備編

2008 年度担当: 中井央

2008.04.22

1 コンパイラの概要

一般にコンパイラの構成は主に次のようになっている。

- 字句解析
- 構文解析
- 意味解析
- 中間コード生成
- 最適化
- 目的コード生成

今回、作成の対象とするプログラミング言語 PL/0' (ぴーえるぜろだっしゅ) のコンパイラでは、目的言語は PL/0 マシンと呼ばれる仮想マシン (Virtual machine) の機械語である。その構成は次のようになっている。

- 字句解析
- 構文解析
- 意味解析 + 目的コード生成

以下、各々を概説する。

1.1 字句解析

なお、今回は字句解析、構文解析に関してはその仕様を予め与えてあるので、自分で 0 から作る必要はないが、その記述の意味を理解する必要はあるため、まず、それらを解説する。

字句解析は、与えられたソースプログラムから単語を切り出す作業である。より正確には言語を構成している最小単位を切り出す。「最小単位」のことを「トークン」と呼ぶ。プログラミング言語を構成している最小単位の種類は、キーワード (`if`, `while` など)、変数等の名前 (識別子と呼ぶ)、数字、記号 (算術演算子、比較演算子、括弧など)、などである。

あるプログラミング言語のコンパイラを作る際、まず、どのようなトークンがあるかを考える。トークンは種類 (自然言語の文法で言えば、品詞) を表現しているが、実際にはソースプログラム上では文字列であるので、どのような文字列がどのトークンかということを対応付ける必要がある。例えば、10 進 (正) 整数は 0 〜 9 までのいずれかの文字が 1 つ以上並んだ形になっている。どの文字がどのように並んでいるかを表現する方法に正規表現がある。SableCC ではあるトークンを構成する文字列を表現するのに正規表現を用いている。

SableCC が取り扱う文字コードセットは Java で採用されている UniCode である。SableCC での正規表現でまず、基本となるのはある一文字を指定する方法で、これはシングルクォートを用いる。例えば足し算のサインを表す文字 + を定義するのは次のようにする。

```
plus = '+';
```

1 つの定義は、左辺に名前、イコール記号 (=)、字句を表現する正規表現、セミコロン、からなる。左側の `plus` の部分は後に述べる文法中でトークンとして使用するための名前である。一文字を表現するには数値により、その UniCode を指定しても良い。例えば改行を表す記号は、UniCode(ASCII) では 10 か 13 である。実際には実行される環境によって異なり、UNIX 系だと 10 (LF)、Windows 系だと 13 + 10 (CR+LF)、Mac OS (バージョン 9 まで) は 13 (CR) となっている。このため、UNIX 環境で空白かタブか改行を表し

たい場合、次のように表現すれば良い。| が選択を表している。

```
blank = ' ' | 9 | 10 ;
```

0 〜 9 のような範囲は ['0' .. '9'] のように記述できる。もし、文法として 10 進数字一桁が必要であれば、次のような定義をすれば良い。

```
digit = ['0' .. '9'];
```

また、一般には (10 進の) 数字を表現したい場合、一桁を表現する数が 1 つ以上並んでいるもの全てを対象としたい。その場合、「1 つ以上の並び」を表現するには記号 + を用いる。10 進数は例えば次のように表現できる。

```
number = ['0' .. '9']+;
```

なお、同様に 0 個以上を表現するには * を用いる。例えば C 言語の識別子 (名前) として使用できる文字の種類を表現するには図 1 のようにすれば良い。これは、まず、先頭の一文字は、英字かアンダースコア ('_') であり、それに続く文字の種類は、英字かアンダースコアか数字であることを示している。

この他、集合演算が可能であるが説明は省略する。

コンパイラにおける字句解析のプログラムは、手書きで作ることも可能であるが、正規表現を与えることで自動で生成してくれるツールが世の中には存在する。SableCC もその 1 つである。

1.2 構文解析

次に文法について記述する。ここで対象とする文法は、文脈自由文法 (Context-Free Grammar, CFG と略される) と呼ばれるものである。例えば、自然言語の文法を簡略化して考えてみる。文は主部と述部からなる、主部は名詞からなる、述部は、動詞のみ、動詞＋名詞、動詞＋名詞＋名詞からなる、というように定義ができる。

このようにある部分がどのように構成されているかを表現するのが文法である。記号を使ってこの文法を表現すると例えば次のようになる。

```
bun := shubu jutsubu ;

shubu := meishi ;

jutsubu := doushi
        |  doushi meishi
        |  doushi meishi meishi
        ;
```

ある部分を定義するのにこの例では := を用いているが、SableCC では = を使う。SableCC での書き方は後ほど紹介する。CFG では上記の例のようにいくつかの規則からなっている。1 つの規則は 1 つの左辺を持ち、右辺は複数あってもよいが、その場合は | で区切る。1 つの右辺は複数の記号の並びとなる。この記号のうち、いずれかの規則の左辺となっているものを非終端記号と呼び、いずれの規則の左辺にもならない記号を終端記号と呼ぶ。上述の例では、bun, shubu, jutsubu が非終端記号、meishi, doushi が終端記号である。

終端記号はトークンと同じと考えて良い。SableCC の場合、字句の定義における左辺の名前は文法における終端記号名でもある。

コンパイラの構文解析器の役割は与えられたソースプログラムが文法にあっているかどうかをチェックすることである。ソースプログラムからまず、字句解析器によりトークンが切り出され、それが構文解析器に渡される。これを受けた構文解析器はこれまでに受け取ったトークンの並びの次にいま渡されたトークンが来ても良いかどうかを判断する。

構文解析の役割はこの他、与えられたソースプログラムの構造を明らかにする。すなわち、それを文法に照らし合わせ、木の形にする。構文解析の結果として得られる木のことを構文解析木という。

1.3 意味解析

SableCC によって得られた構文解析クラスのインスタンスは、構文解析をすると構文解析木だけを作成する。意味解析はこの木を走査しながら行なうことになる。SableCC ではこの木を走査するためのフレームワークを提供しており、ユーザは提供されているクラス (後述) を継承した独自の意味解析クラスを作成し、

```
identifier = (['a' .. 'z']|['A' .. 'Z']| '_')(['a' .. 'z']|['A' .. 'Z']| '_' | ['0' .. '9'])*
```

図 1: C 言語の識別子を表す SableCC における正規表現

そのクラスのインスタンスに解析木を走査させて、意味解析を行なわせる。

さて、C 言語のような手続き型言語を考えてみよう。C 言語では { と } で囲んだ中にコードを記述する。この際、前半にはその中で使用する変数の宣言を記述する。以下、この部分を宣言部分という。後半には、計算するためのコードが if 文や while 文などを使って記述されることになる。以下、この部分を実行部分という。

このような言語のコンパイラでは意味解析として、宣言部分については、宣言を記録していく。ただし、同じ名前が宣言部に二度現れた場合、それはエラーとして扱う。実行部分については、名前が出現した場合、それが宣言されているかチェックを行なう。この他、関数呼出しや演算では、引数 (被演算子) がその関数にとって妥当なものであるかの検査も行なう。

このときの具体的な作業としては、表を用意しておき、宣言部分では、宣言ごとにその名前と型名を記号表に登録する。登録する際は既に同じ名前が表の中にないかどうかチェックする。実行部分では表を使って、宣言されているか、関数呼出しや演算の記述は正しくされているかをチェックする。

一般にプログラミング言語は入れ子型のスコープ規則を採用している。スコープとは名前の有効範囲のことであり、C 言語の例で言えば、例えば { から対応する } までが 1 つのスコープとなり、その中の宣言部分で宣言された変数はその中だけで有効である。ここでは { と } で囲まれた部分のようにあるスコープを示す部分のことを「ブロック」と呼ぶことにする。ところで if 文や while 文などでは { と } で囲まれた複数の文を記述することができる。この際、同様に宣言部分も含まれる (宣言がなくても良い)。

次の例を見てみよう。

```
 1: main(){
 2:   int a, b;
 3:
 4:   ...
 5:
 6:   while(...){
 7:     int a;
 8:
 9:     a = ...;
10:
11:       = ...  b ...;
12:   }
13: }
```

2 行目では変数 a と b を宣言している。6 行目の while 文の本体は { と } で囲まれており、ブロックとなっている。そして、7 行目には a の変数宣言がある。この a は 2 行目の a とは別物である。また、11 行目では変数 b を参照しているがこれはこのブロック中に宣言はないため、1 つ外側のブロックで宣言された b を参照することになる。

さて、意味解析の仕事として、変数の宣言と使用についてのチェックがあることを述べたが、それを行なうにはこのようなスコープの概念を理解し、それを扱うような記号表を実装する必要がある。

1 つの比較的単純なやりかたは、線形リストのようなデータ構造で実現することである。ただし、どこからが新しいブロックに関する情報なのかがわかるようにブロックの境界を示すような要素を持たせる必要がある。

以下、具体的な意味解析の方法について概説する。詳細は後に具体的に SableCC を用いてコンパイラを作っていくことについて述べる際に記す。

まず、情報を持っているノードは、葉ノード、すなわち、トークンを表現しているノードである。すなわち、数を表現しているトークンのノードでは、その数を表現している文字列を保持しており、それをもとにしてその値を計算することは容易にできる。この処理は場合によっては字句解析に行なわせても良い。SableCC では、葉ノードが入力時の文字列を保持しているのでそれを使って意味解析時に値を作り出す必要がある。変数名も同様である。

あるノードで情報が得られたらそれをどこかに保存しておく必要がある。それは別のノードでそれを必要とするからである。例えば四則演算を表す文法は次のようになる。これは SableCC で cmm 言語の文法を記述した際の算術式部分である。

文法 G1

```
expression       =
 {explus}  expression plus  term |
 {exminus} expression minus term |
 {exterm}  term
        ;

term       =
 {termmult} term mult factor |
 {termdiv}  term div  factor |
 {termfactor} factor
        ;

factor       =
 {factid}    id |
 {factfc}    id lpar fparams rpar |
 {factnum}   number |
 {factparen} lpar expression rpar
```

例えば入力として 1 + 2 があったとしよう。これに対する構文解析木は図 2 である。この図における一番上の expression ノードは、expression plus term を表現している。ここで例えば、被演算子が妥当であるかとか各被演算子に対応したコードを得て、このノードでのコードを作るとかの作業を行なう必要がある。このためには、下位のノードで作り出された情報を取得できなければならない。

SableCC でこれを行なうにはハッシュを用いる。ハッシュはキーと値を組にして表に登録するデータ構造であり、一般に検索速度が速い。Java では API でハッシュが用意されている。SableCC で情報の受渡しをするにはハッシュテーブルに対して、ノードをキーとし、値をしまっておく。この際、意味解析全体で複数の種類の値を取り扱う必要があれば、それをまとめるクラス (気分は構造体) などを用意しておく。あるノードを訪問した際に情報を作成したら、それをそのノードをキーとし、ハッシュに登録する。そしてその上位のノードを訪問した際、その子ノードに関する情報が必要であれば、ハッシュにその子ノードをキーとして情報を取り出すように要求すれば良い。

ここまでは主に変数の宣言を例に話をしてきたが、実際のプログラミング言語には関数の宣言も存在する。

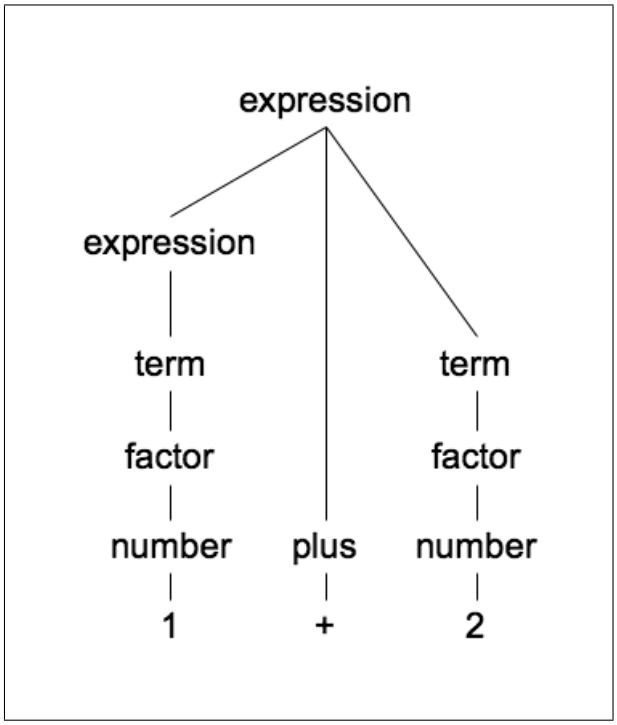

図 2: 1+2 に対する構文解析木

関数の宣言では扱う構造や情報が変数の場合よりも多くなるが、基本的な作業内容は同じである。

1.4 目的コード生成

この演習で作成するコンパイラの目的コードは PL/0 マシンと呼ばれる仮想計算機のコードである。これについては別の授業「プログラミング言語処理系」の仮想機械についてのテキストを参照して欲しい。テキストは次のページに置いてある。

http://www.slis.tsukuba.ac.jp/~nakai/2007/Compiler/

2 SableCC と Visitor パターン

SableCC はオブジェクト指向に基づいたコンパイラフレームワークである。Java により記述されており、出力されるプログラムも Java で記述されている。ここでは要求される Java についての知識について述べ、SableCC がベースとしている Visitor パターンについて概説する。

2.1 Java について要求される知識

まず、この演習において要求される Java の知識について列挙してみる。

- C 言語のプログラムが書けること。if, while, for などの制御文、関数の定義、呼出しなど。構造体が使えること。ポインタは深い理解がなくても良いが、できればリンクトリストをそらで書けるのが望ましい。
- クラスの概念を理解し、継承を使えること。
- 必要に応じてドキュメントを参照し、Java API が利用できること。

このあたりまでなら、大体、Java の入門書となっている本で勉強できる。私の授業、「プログラミング言語各論」(2007 年度) の第 1 回から第 3 回までのテキストでもこれらの内容をカバーしている。

実際は別途に示す手引通りに進めていけばそれほど難しい知識は必要としない。

2.2 Visitor パターン

SableCC に限らず、一般にコンパイラ生成系ではよく Visitor パターンが用いられる。

木のようなデータ構造を考える場合、一般には、関連する情報はノードの中に置くようにする。また、オブジェクト指向でそのような木を表現する際は、各ノードがその振舞いをするためのメソッドを持つようにする。

一方、木のようなデータ構造はあくまでも構造のみを表していて、その構造の上でユーザがやりたいことは色々あるため、データ構造と「やること」を分離した方がよい、という考え方もある。この考え方に基づいたプログラムの構成方法の 1 つが Visitor パターンに示されるものである。

Visitor を使う利点は、「やること」がデータ構造と (ある意味) 独立していれば、新たにやることを増やすような場合にデータ構造を表現しているクラスを変更する必要がないことである。

SableCC の場合、構文解析が正常に終了すればその結果としての構文解析木というデータ構造が得られる。意味解析はそのデータ構造を走査して行なう。各ノードはいずれかの文法規則に対応する。

木の訪問については、例えば、木のルートから訪問を始めてその最左の子、その最左の子、…、葉ノードへ、1 つ親へ戻る、右の子があればまた訪問、といった、深さ優先探索がある。このとき、あるノードに対しては、その親ノードからやってくる、その親ノードへ返る、という 2 つの訪問がある。この他、子ノードへの移動、子ノードからの戻りの後、その右の子ノードへ移動、…という動作もある。

本演習では、あるノードへ最初に訪問したときと親ノードへ返るときの 2 つのタイミングを考慮しておけば良い。これらは、ノード名の前に `in`、`out` をつけたメソッドにより実現される。各ノードの名前は SableCC の文法の書き方にしたがって一意に決定される。これについては演習のテキストで記述する。