

机器人编程语言解析器设计 step by step

mhuasong@wust.edu.cn

注：本教程是教材《机器人理论与技术基础》（闵华松、魏洪兴编著-机械工业出版社）第7章的实验。在学习机器人语言解析器编程之前，请阅读教材及讲义，该实验无需具备编译原理的基础，但如果有基础会更有助于读者掌握机器人编程语言解析器的设计原理。

第一步：设计指令集

表 7.9 指令设计表

指令类型	指令内容	指令关键词
状态指令	初始化机器人状态、进入准备状态、程序结束	INIT、BEGIN、END
移动指令	从一点移动机器人末端到另一点、以直线插补从当前点移动到目标点、直接控制关节转动角度	MOVEP、MOVEL、MOVEJ
变量定义指令	字节型变量定义、整形变量定义、空间点定义、关节位置变量定义	DEFN、DEFX

在教材中，我们给了一个最简易的指令集，见表 7.9。针对该指令集，我们编写一个 InstructionSet.h 头文件，定义了一些结构体，代码如下：

```
#ifndef _INSTRUCTIONSET_H
#define _INSTRUCTIONSET_H

#define WORDNUM 16 //指令集分词库的总数
#define FALSE 0
#define MOVPNUM 15 //
#define BEGINNUM 15 //
#define MOVLNUM 15 //
#define MOVJNUM 15 //

//指令集词库，对应表 7.11
typedef enum
{
    ENDFILE,ENDLINE,INVALID,
    INIT,BEGIN,MOVEP,MOVEL,MOVEJ,END,B,I,P,J,X,Y,Z,DEFN,DENFX,
    ID,NUM
}Tokentype;

//词法分析 FA 转换图，5 种状态，对应图 7.14
typedef enum
{
```

```
START, INID, INNUM, DONE, INPOINT  
}StateType;
```

//定义一个点三维坐标的结构体

```
typedef struct  
{  
    double x;  
    double y;  
    double z;  
}Point;
```

//指令类别

```
typedef struct  
{  
    unsigned char ucType;  
    char szValue[16];  
}STRU_CONST_VAR;
```

//字节数据

```
typedef struct  
{  
    int bn;  
    char Bvar[100];  
}B_CONST_VAR;
```

//整型数据

```
typedef struct  
{  
    int in;  
    int Ivar[100];  
}I_CONST_VAR;
```

//关节角

```
typedef struct  
{  
    double j1;  
    double j2;  
    double j3;  
    double j4;  
    double j5;  
    double j6;  
}Joint;
```

```
#endif
```

第二步，如何读取一个程序文件

这里我们使用课程的指令集，写一段文本程序：

```
INIT
DEFX P0 123 456 789
DEFX P1 123 456 798
DEFX J0 0 0 90 0 90 0
BEGIN
MOVEP P0
MOVEL P1
MOVEJ J0
END
```

我们把它存为一个文本文件，假设我们定义机器人程序的后缀名为 srl(simple robot language, 随便瞎取的名)的 test.srl。要解析这个程序文件，其流程就是一行行读入，然后再解析。主体程序的流程图见教材第 7 章的图 7.6。

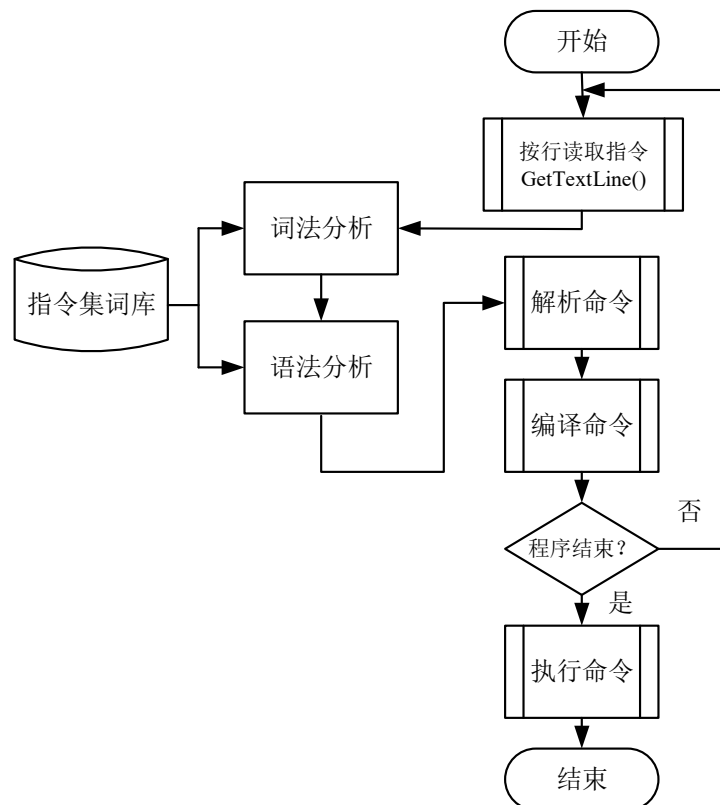


图 7.6 一个典型的机器人语言解析器流程

这里我们需要写一个按行读取指令的函数，C++中读取一行文本的函数原型如下：

```
char *fgets(char *str, int n, FILE *stream);
```

如果函数成功，则返回 `str`。如果在读取最大字符数（`n-1`）之前到达了换行符或文件结尾，则只有已经读取的字符和换行符会被添加到 `str` 中，并且一个 `null` 字符被添加到字符串末尾。如果发生错误或者在到达文件结尾前没有读取到任何字符，则返回 `NULL`。

这个函数很好写，随便写一下，代码没有很严谨：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <iostream>

#include "getToken.h"

using namespace std;

char lineBuf[255];
static int linepos = 0;
extern FILE *file;
int lineno=-1;

int readOneLine()
{
    int bufsize=0;

    if(fgets(lineBuf,80,file))
    {
        bufsize=strlen(lineBuf);
        linepos=0;
        lineno++;
        cout<<"Line No: "<<lineno<<endl;
        cout<<"Bufsize = "<<bufsize<<endl;
        printf("%s",lineBuf);
    }
    else
    {
        bufsize=-1;
    }
    return bufsize;
}
```

把 C++源文件存为 getToken.cpp，读取之后需要进行词法分析，我们先不写后面的代码，暂时只写这个函数，并定义一个包含头文件 getToken.h。

```
#ifndef _GETTOKEN_H_
#define _GETTOKEN_
#include<stdio.h>
#include "InstructionSet.h"

int readOneLine();
```

```
#endif
```

写个主程序 main.cpp 测试一下:

```
#include<stdlib.h>
#include<stdio.h>
#include<fstream>
#include<iostream>
#include "getToken.h"

using namespace std;
FILE* file;

int main(int argc, char** argv)
{
    file = fopen("test.srl","r");
    if(file == NULL)
    {
        cout<<"Open file error!"<<endl;
        exit(-1);
    }
    cout<<"Please press return key to read the robot program:"<<endl;
    getchar();
    while(readOneLine() != -1)
    {
        cout<<"Read one line!"<<endl;
    }
    cout<<"Got it!"<<endl;
    fclose(file);
    return 0;
}
```

至此, 我们简单地写了四个文件: main.cpp, getToken.cpp, getToken.h, InstructionSet.h, 把 C++ 代码编译一下, 可以直接编译:

```
g++ main.cpp getToken.cpp -o test
```

当然, 我们写一个 Makefile 文件是最好, 不会 make 的同学可以阅读我的另一门开源教程《嵌入式系统仿真实验》, 里面有讲一些简单的 Makefile 的知识, 我们写一个简单 Makefile:

```
CXX = g++
TARGET = test
OBJ = main.o getToken.o

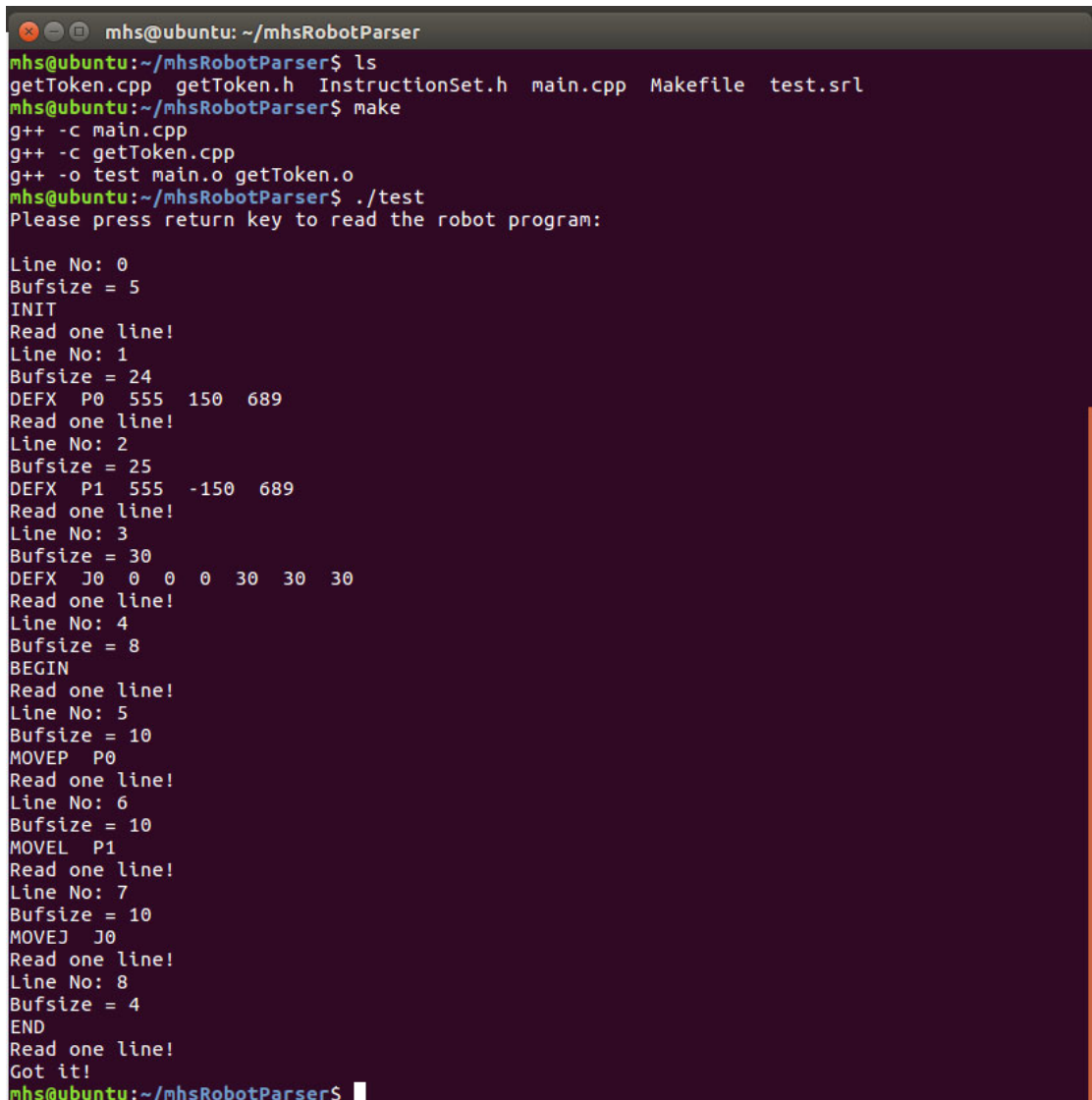
$(TARGET): $(OBJ)
    $(CXX) -o $(TARGET) $(OBJ)
```

```
main.o: main.cpp
$(CXX) -c main.cpp

getToken.o: getToken.cpp
$(CXX) -c getToken.cpp

clean:
rm -f *.o $(TARGET)
```

接下来，我们编译整个工程（目前为止只有两个小程序 main.cpp, getToken.cpp），编译执行结果如图 1 所示。



```
mhs@ubuntu: ~/mhsRobotParser
mhs@ubuntu:~/mhsRobotParser$ ls
getToken.cpp getToken.h InstructionSet.h main.cpp Makefile test.srl
mhs@ubuntu:~/mhsRobotParser$ make
g++ -c main.cpp
g++ -c getToken.cpp
g++ -o test main.o getToken.o
mhs@ubuntu:~/mhsRobotParser$ ./test
Please press return key to read the robot program:

Line No: 0
Bufsize = 5
INIT
Read one line!
Line No: 1
Bufsize = 24
DEFX P0 555 150 689
Read one line!
Line No: 2
Bufsize = 25
DEFX P1 555 -150 689
Read one line!
Line No: 3
Bufsize = 30
DEFX J0 0 0 0 30 30 30
Read one line!
Line No: 4
Bufsize = 8
BEGIN
Read one line!
Line No: 5
Bufsize = 10
MOVEP P0
Read one line!
Line No: 6
Bufsize = 10
MOVEL P1
Read one line!
Line No: 7
Bufsize = 10
MOVEJ J0
Read one line!
Line No: 8
Bufsize = 4
END
Read one line!
Got it!
mhs@ubuntu:~/mhsRobotParser$
```

图 1 读取程序

可以看到我们成功实现了源程序的按行读取。

第三步：按照教材图 7.14 的原理，进行词库查询

根据图 7.15，我们写一个 lex2token() 的词库查询函数，我们把它写入到刚才的 getToken.cpp 中去：

```

//获取下一个字符
static int getNextChar(void)
{
    int bufsize=0;
    bufsize=strlen(lineBuf);
    if(linepos<bufsize+1)
    {

        return lineBuf[linepos++];

    }
    else
        return lineBuf[linepos];

}

//查找词库
static Tokentype Wordslookup(char *s)
{
    int i;
    for(i=0;i<WORDNUM;i++)
    {
        if(!strcmp(s,Words[i].str))
        {
            return Words[i].tok;
        }
    }
    return INVALID;
}

void ungetNextChar(void)
{
    linepos-- ;
}

//词法分析，对应教材图 7.15
Tokentype lex2token(void)
{
    Tokentype currentToken = INVALID;
    int tokenStringIndex=0;
    StateType state = START;
    int save = 0;
    char c = 0;
    while(state!=DONE)

```

```

{
    c=getNextChar();
    save=1;
    switch(state)
    {
    case START:
        if(isalpha(c))
        {
            state=INID;
        }
        else if(isdigit(c) || (c=='-'))
        {
            state=INNUM;
        }
        else if(c==' ')
        {
            save=0;
        }
        else if(c=='/')
        {
            save=0;
        }
        else
        {
            state=DONE;
            switch(c)
            {
            case EOF:
                save=FALSE;
                currentToken=ENDFILE;
                break;
            case '\n':
            case '\r':
                save=0;
                currentToken=ENDLINE;
                break;
            case '*':
            case ':':
                break;
            }
        }
        break;
    case INID:
        if (!isalpha(c) )

```



```

        {

            ungetNextChar();
            save = FALSE;
            state = DONE;
            currentToken = ID;
        }
        break;
    case INNUM:
        if(isdigit(c))
        {state=INNUM;}
        else if(c=='.')
        {
            state=INPOINT;
        }
        else {
            ungetNextChar();
            save = 0;
            state = DONE;
            currentToken = NUM;
        }
        break;
    case INPOINT:
        if(!isdigit(c))
        {
            ungetNextChar();
            save=0;
            state=DONE;
            currentToken=NUM;
        }
        break;
    case DONE:
        break;
}

if ((save) && (tokenStringIndex <= 20)) {
    tokenString[tokenStringIndex] = c;
    tokenStringIndex++;
}

if (state == DONE) {
    tokenString[tokenStringIndex] = '\\0';
    if (currentToken == ID) {
        currentToken = Wordslookup(tokenString);
    }
    else{

```

```

        return currentToken;
    }
}
return currentToken;
}

```

记住更新 getToken.h 文件，把新增的函数定义写入头文件。

接下来，我们在主程序里测试一下，在 main.cpp 里面我们增加两行定义：

```
unsigned char ttype;
```

```
TokenType CurToken;
```

然后在 while 循环里加上刚才写的 lex2token()函数的测试：

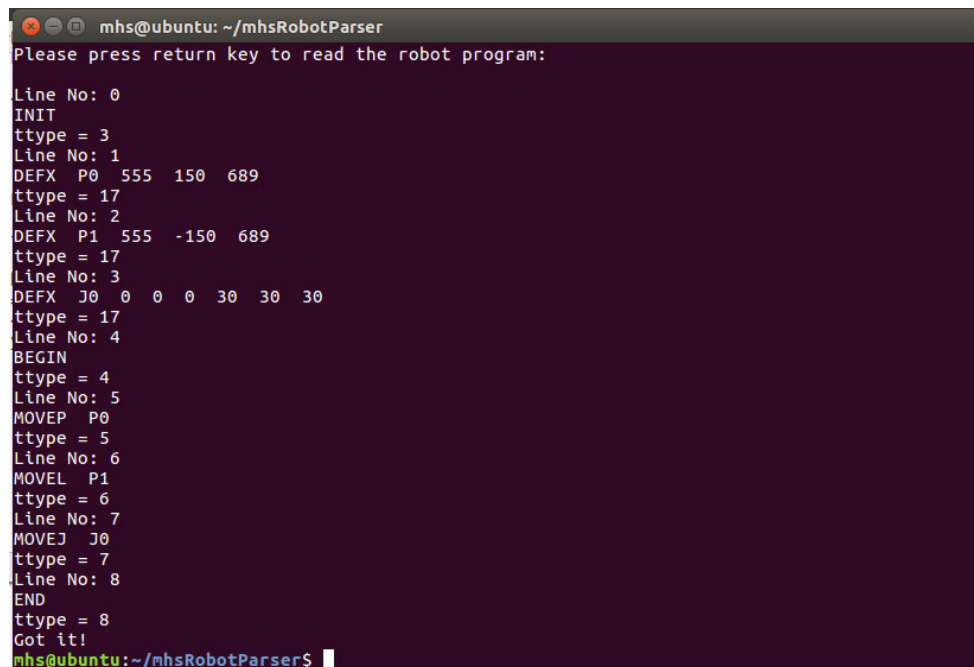
```

while(readOneLine() != -1)
{
    //cout<<"Read one line!"<<endl;

    CurToken = getToken();
    ttype=CurToken;
    cout<<"ttype = "<<static_cast<int>(ttype)<<endl;
    if(ttype==ENDFILE)
    {
        return false;
    }
}

```

重新 make 一下，再测试看看：



```

mhs@ubuntu: ~/mhsRobotParser
Please press return key to read the robot program:

Line No: 0
INIT
ttype = 3
Line No: 1
DEFX P0 555 150 689
ttype = 17
Line No: 2
DEFX P1 555 -150 689
ttype = 17
Line No: 3
DEFX J0 0 0 0 30 30 30
ttype = 17
Line No: 4
BEGIN
ttype = 4
Line No: 5
MOVEP P0
ttype = 5
Line No: 6
MOVEJ P1
ttype = 6
Line No: 7
MOVEJ J0
ttype = 7
Line No: 8
END
ttype = 8
Got it!
mhs@ubuntu:~/mhsRobotParser$

```

图 2. 词库查询分析

从图 2 可以看到我们成功地对源程序中的词法进行正确分析。得到分词 token 在词法表中的正确代码(ttype)。我们加一句错误的语句看看是否会出现错误呢？

比如，我们在 test.srl 里面加了一句 SET 语句，看看图 3 中，正确地识别了 ttype=2，对应词库表里面是 INVALID。



```
Line No: 4
SET X
ttype = 2
```

图 3. 词库查询分析测试

能够正确地对每句程序进行词库查找之后，我们再进行进一步实现对每条符合语法规则的语句进行语法分析。

第四步： 语法分析

语法分析的原理、过程及流程图见教材的 7.5.4 节。我们需要根据上一步得到的 currentToken，完成图 7.20 的语法分析处理整体流程处理程序。对照图 7.20 的流程图，我简单写一个语法分析函数：

```
int SyAnalyze(Tokentype CurToken)
{
    unsigned char ttype;

    ttype=CurToken;

    if(ttype==ENDFILE)
    {
        return false;
    }
    switch(ttype)
    {
        case INIT:
            cout<<"init"<<endl;
            curpos=init_analyze();
            break;
        case BEGIN:
            cout<<"begin"<<endl;
            curpos=Begin_SyAnalyze();
            //curangle=ikine(curpos);
            break;
        case END:
            cout<<"end"<<endl;
            endlable=End_SyAnalyze();
            break;
        case ENDLINE:
            cout<<"endline"<<endl;
            errInforDisplay(ERR_ZERO,lineno);
            getchar();
            break;
        case NUM:
```

```

        cout<<"num"<<endl;
        errInforDisplay(ERR_VALUE,lineno);
        getchar();
        break;
    case DEFN:
        cout<<"defn"<<endl;
        defn_analyze();
        break;
    case DEFX:
        cout<<"defx"<<endl;
        defx_analyze();
        break;
    case MOVEP:
        cout<<"movep"<<endl;
        curpos=movep_analyze();
        break;
    case MOVEL:
        cout<<"movel"<<endl;
        curpos=movel_analyze();
        break;
    case MOVEJ:
        cout<<"movej"<<endl;
        curpos=movej_analyze();
        break;
    default:
        errInforDisplay(ERR_COMLBL,lineno);
        getchar();
        break;
}
return ttype;
}

```

这个函数实现整体的语法分析，语法分析器针对上一步词法分析输出的词法单元(Token)进行判别，创建各类语句对应的语法树形结构（语法树 Syntax Tree）。

第五步：构建每种语句的语法树处理函数

例如：INIT 的语法树处理，我只是示范性的写了一下：

```

Eigen::Matrix4d init_analyze()
{
    cout<<"please move manipulator to initial pose."<<endl;
    Eigen::Matrix4d initpos;
    //具体如何初始化及控制机械臂运动到初始位姿,这里我们省略了
    if(ENDLINE!=lex2token())
    {

```

```

        errInforDisplay(ERR_COMDATA,lineno);
        getchar();
        exit(1);
    }

    return initpos;
}

```

在初始化程序语句(INIT)中，机器人需要做哪些初始化工作，可以在这个函数里去实现。这个函数我简单定义了一个 4x4 矩阵作为返回值，代表初始化的位姿矩阵，当然一般应该定义一个状态值，表明初始化是否成功如否，应该更符合软件工程的思想一些。

依次我们可以将 BEGIN、END、ENDLINE、NUM、DEFN、DEFX、MOVEP、MOVEL、MOVEJ 这些语句的语法树处理函数。

以 MOVEP 语句为例，参照图 7.9 所示的语法结构图，MOVEP 语句语法分析流程图如图 7.21 所示。

MOVEP 语法分析流程对 MOVEP 后面的词素进行分析，判别是 NUM 还是 P，然后进行相应的参数读入处理，并可按照我们在第五章运动规划中介绍的关节空间插补轨迹规划算法，计算出所有轨迹点的关节角序列，**这个留给读者自己规划去实现**。MOVEL 对应笛卡尔空间插补轨迹规划算法，而 MOVEJ 则直接给出机械臂所有轴的关节角。其余指令语法树的实现也较为简单，可参照实现。

对照教材中的图 7.21，MOVEP 的处理函数代码示范如下：

```

Eigen::Matrix4d movep_analyze()
{
    Tokentype movep_token=lex2token();
    Point movep_point;
    Eigen::Vector3d moveppos;
    Eigen::VectorXd movepangle(6);

    switch(movep_token)
    {
        case NUM:
            movep_point.x=atoi(tokenString);
            cout<<"x="<<movep_point.x<<endl;
            if(NUM!=lex2token())
            {
                errInforDisplay(ERR_VALUE,lineno);
                getchar();
                exit(1);
            }
            movep_point.y=atoi(tokenString);
            cout<<"y="<<movep_point.y<<endl;
            if(NUM!=lex2token())
            {
                errInforDisplay(ERR_VALUE,lineno);
                getchar();
            }
        }
    }
}

```

```

        exit(1);
    }
    movep_point.z=atoi(tokenString);
    cout<<"z="<<movep_point.z<<endl;
    break;
case P:
    lex2token();
    movep_point.x=pv[atoi(tokenString)].x;
    movep_point.y=pv[atoi(tokenString)].y;
    movep_point.z=pv[atoi(tokenString)].z;
    cout<<"x="<<movep_point.x<<endl;
    cout<<"y="<<movep_point.y<<endl;
    cout<<"z="<<movep_point.z<<endl;
    break;
default:
    errInforDisplay(ERR_COMDATA,lineno);
    getchar();
    exit(1);
    break;
}

moveppos(0)=movep_point.x;
moveppos(1)=movep_point.y;
moveppos(2)=movep_point.z;

//compute movepangle through inverse kinematics ,for example:
ikine(moveppos);
//planning CompileMatrix through trajectory planning function,for
example:jtraj(curangle,movepangle);
return CompileMatrix;
}

```

写完所有的语句对应的语法树处理函数之后，我们可以在主程序中调用这个语义分析函数，对整个程序进行解析。

写完重新 make 一下，执行结果如图 4。

```

Line No: 6
MOVEL P1
ttype = 6
movel
x=555
y=-150
z=689
-----
Line No: 7
MOVEJ J0
ttype = 7
movej
six angles is: 0;0;0;0.523599;0.523599;0.523599;
-----
Line No: 8
END
ttype = 8
end
解析结束
-----
Got it!
mhs@ubuntu:~/mhsRobotParser$

```

图 4. 样本程序解析示例图

第六步：执行

一般的计算机编译器，在对源程序完成语义分析之后，需要将源程序生成可执行的目标代码。有些编译器还会生成中间代码，将中间代码优化后，再翻译成目标机器代码。

机器人控制器一般采取解析执行的方式运行，控制程序读入经过语法分析后的合法语句，按照构建的语法树，生成可供控制的关节角序列，控制机器人完成相应运动。如果机器人采用总线控制，各关节为独立关节，则主控制器按照通信协议将相应的关节命令（编译器生成的关节角序列矩阵）发送给相应关节执行。在每条语句的语法树解析函数中，可以将需要执行的内容按顺序写入一个指令队列，然后再写一个指令队列的控制流程，进行执行。

这一步我不再写示范了，大家可以根据前面章节的实验代码，写出相应的执行，也可以把代码写到 ROS 中去控制一个机器人模型，然 ROS 执行文本程序，然后控制机器人模型。

结束语

机器人学是典型的交叉学科，开发人员需要具备多学科的专业知识和技能。这个教程是为有志开发自己的机器人底层软件的初学者写的，不要求读者具备高深的程序设计基础和编译原理的基础。但掌握基本原理之后，可以继续学习现代人工智能脚本语言的解析器原理，可以在更高级的脚本语言解析器的基础上加入自己的机器人控制语句，让自己的机器人语言可以实现更高级的功能。

先进的机器人编程语言需要达到如下要求：

1) 支持多种坐标系编程

在进行机器人编程时，除了建立绝对坐标系（世界坐标系）之外，为了方便机器人工作，还应支持建立其他坐标系，如关节坐标系、末端执行器坐标系、作业对象坐标系等参考坐标系，同时建立这些坐标系之间的转换关系。

2) 支持基本的程序设计常用结构

机器人编程系统应该支持基本的程序控制结构，包括简单程序、等待程序、分支程序、循环程序、子程序以至中断程序等结构化编程方法。

3) 支持机器人作业描述

机器人作业的描述与其环境模型密切相关，编程语言水平决定了描述水平。现有的机器人语言需要给出作业顺序，由语法和词法定义输入语句，并由它描述整个作业过程。先进的机器人编程语言可以支持任务级的作业描述，只需要按照某种规则描述机器人对象的初始状态和最终目标状态，机器人语言系统即可利用已有的环境信息和知识库、数据库自动进行推理、计算，从而自动生成机器人详细的动作、顺序和数据。

4) 支持多种运动规划

描述机器人需要进行的运动是机器人编程语言的基本功能之一。用户能够运用语言中的运动语句，可选择与多种路径规划器连接，允许用户规定路径上的点及目标点，决定采用何种规划算法以及避障算法。

5) 具有良好的编程环境

如同任何计算机系统一样，一个好的编程环境有助于提高程序员的工作效率。好的编程系统应具有下列功能：

1) 在线修改和重启功能

机器人在作业时需要执行复杂的动作和花费较长的执行时间，当任务在某一阶段失败后，从头开始运行程序并不总是可行，因此需要编程软件或系统必须有在线修改程序和随时重新启动的功能。

2) 传感器输出和程序追踪功能

因为机器人和环境之间的实时相互作用常常不能重复，因此编程系统应能随着程序追踪记录传感器的输入输出值。

3) 仿真功能

可以在没有机器人实体和工作环境的情况下进行不同任务程序的模拟调试。

4) 人机接口和综合传感信号

在编程和作业过程中，编程系统应便于人与机器人之间进行信息交换，方便机器人出现故障时及时处理，确保安全。而且，随着机器人动作和作业环境的复杂程度的增加，编程系统需要提供功能强大的人机接口。

随着计算机编程语言以及人工智能的发展，机器人编程语言也有了与人工智能编程语言相结合的趋势。例如在支持人工智能的通用编程语言，如 Java、python、lua 等脚本语言基础上，开发出机器人的编程语言，使得机器人编程系统更加强大、更加方便开发。