به نام خدا

فاز پایانی پروژه درس کامپایلر

استاد : سرکار خانم مهندس راضیه اسکندری

تیم همکاری

سجاد مؤمنی

حمید نصر اصفهانی

عرفان ذکری اصفهانی

نیمسال تحصیلی : بهمن 96

صورت مسأله

گرامری برای ما در فایل های دراپ باکس کامپایلر تعریف شده بود کـه دارای ابهـام و برخورد های reduce / reduce و همچنین shift / reduce بود کـه بـه همکـاری تـیم توسعه نرم افزار و با تغییر گرامر یا تغییر اولویت های گرامر تعریف شده برطرف شـود و در yacc پیاده سازی شود که اگر جمله ای توی ایـن گرامـر تعریـف نشـده بـود پیغـام خطای مناسب را بدهد و در غیر این صورت تمام کاهش را نمایش دهد.

نحوه نصب کردن نرم افزار yacc

برنامه yacc را با دستورات زیر در Linux نصب میکنیم :

```
sudo apt-get install bison
sudo apt-get install byacc
sudo apt-get install flex
```

نحوه کار با yacc

در اینجا باید یک فایل با فرمت .y ایجاد کنیم که ساختار اطلاعات ورودی مـا بـه شـکل زیر میباشد :

```
%{
اعلان ثابت ها
اعلان متغیر ها
%}
اعلان توکن های موجود در گرامر
%%
قواعد گرامر
%%
روال های پشتیبان به زبان C
```

فایل yacc از سه قسمت تشکیل شده که عبارتست از:

بخش اول :

اعلان متغیر ها و ثابت ها و همچنین توکن ها

بخش دوم :

تعریف قواعد گرامر ها

بخش سوم :

روال های پشتیبانی به زبان C

اگر قانونی به شکل زیر داشته باشیم :

<--- غیر پایانه1 | پایانه 2 | پایانه 3پایانه

آن را به صورت زیر پیاده سازی میکنیم :

: غیر پایانه3عملیاتی که باید انجام شود پایانه

عملیاتی که باید انجام شود |2پایانه

عملیاتی که باید انجام شود |1پایانه

;

اگر گرامر به صورت A---> €بود باید به شکل زیر بنویسیم :

A :

اول قوانین گرامر رو به صورت بالا تعریف میکنیم ، سپس برای هر کدام از غیر پایانه ها یک توکن تعریف میکنیم . این توکن ارتباط بین yacc و lex را برقرار میکند . چون زمانی که lex پایانه رو پیدا کند نام مربوط به آن را به yacc بازگشت میدهد .

در کد زیر لیستی از توکن های تعریف شـده کـه بـه برنامـه yacc بازگشـت میدهـد را مشاهده میکنید :

```
"print"                 { return PRINT;     }
"in"                    { return IN;        }
"false"                 { return FALSE;     }
"true"                  { return TRUE;      }
"end"                   { return END;       }
"case"                  { return CASE;      }
"write"                 { return WRITE;     }
"read"                  { return READ;      }
"return"                { return RETURN;    }
"continue"              { return CONTINUE;  }
"until"                 { return UNTIL;     }
"repeat"                { return REPEAT;    }
"break"                 { return BREAK;     }
"of"                    { return OF;        }
"switch"                { return SWITCH;    }
"else"                  { return ELSE;      }
"then"                  { return THEN;      }
"if"                    { return IF;        }
"while"                 { return WHILE;     }
"do"                    { return DO;        }
"down"                  { return DOWN;      }
"to"                    { return TO;        }
"for"                   { return FOR;       }
"int"                   { return INTEGER;   }
"double"                { return DOUBLE;    }
"char"                  { return CHARACTER; }
```

```
"bool"                  { return BOOLEAN;   }
"constant"              { return CONSTANT;  }
["].["]                 { return STRING;    }
['].['] 		{ return CHAR;      }
"program"               { return program;   }
```

در ادامه با استفاده از دستورات زیر از فایل .y که داریم استفاده کرده و دو فایل y.tab.c و y.tab.h را میسازیم :

مرحله اول :

این دستور دو فایل y.tab را میسازد و فایل yacc را اگر خطایی داشته باشد به ما میدهد :

```
yacc -d yacc.y
```

مرحله دوم :

این دستور فایل lex را بررسی میکند و اگر دارای اروری باشد آنرا به ما اطلاع میدهد و فایل lex.yy.c رو میسازد :

```
lex lex.l
```

مرحله سوم :

با اجرای این دستور با استفاده از دو فایل lex.yy.c و y.tab.c فایل اجرایی برنامه yacc را درست میکنیم که ما آنرا با نام compiler در نظر گرفتیم :

```
gcc -g lex.yy.c y.tab.v -o compiler
```

فایل y.tab.h رو به عنوان کتابخانه به برنامه lex اضافه میکنیم ، چون شامل توکن های گرامر و اطلاعات دیگر هست ، در فایل lex هم با استفاده از عبارات منظم چیز هایی رو که میخواهیم تشخیص میدهیم و به اسم همان توکن موجود در فایل yacc بازگشت میدهیم تا بتوانیم آنجا از آن توکن ها استفاده کنیم.

حال فایل اجرایی برنامه رو با دستور زیر اجرا میکنیم ، رشته ورودی از ما دریافت میکند . اگر رشته ورودی قابل پذیرش توسط گرامر تعریف شده باشد تمام کاهش هـای آن را چاپ میکند و در غیر اینصورت پیغام syntax error میدهد :

```
./compiler
```

توضیح کد

کدی که در زیر مشاهده میکنید قسمتی است که متغیر هایی که میخواهیم بـا آنهـا در قسمت توکن و اکشن ها استفاده کنیم را تعریف میکنیم :

```
%union {
     int num;
     char *id;
     char *str;
     double doub;
}
```

با کد زیر غیر پایانه ای که گرامر ما با آن شروع میشود را معرفی میکنیم :

```
%start Program
```

اگر lex یک IntNumber را تشخیص دهد با استفاده از خط کـد زیـر آن را بـه متغیـر تعریف شده به صورت توکن در yacc ارسال میکند :

```
Lex.l :                    yylval.num = atoi(yytext); return
IntNumber;
Yacc.y :           %token <num> IntNumber
```

با دستور زیر نیز مابقی توکن ها را تعریف میکنیم ، برای مثال :

```
%token FOR THEN DOWN TO DO WHILE SWITCH CASE OF
```

در قسمت بعد قواعدی که برای کاهش داریم را به صورت زیر مینویسیم:

```
Dec:
  VarDecs ‘:’   {prinntf(“VarDecs   > Dec \n”) ;}
| FuncDecs    {prinntf(“FuncDecs > Dec \n”) ;}
;
```

که در اینجا علامت | همان or میباشد و عبارتی که داخل پرانتز میباشد نشان دهنده ی کاری است که پس از دیدن این توکن باید انجام شود.

پس از این مرحله توابعی که نیاز داریم را تعریف میکنیم.مثلا توابعی که پس از دیـدن توکن ها فراخوانی میشوند.

۱-yyerror

زمانی که ورودی با هیچ قاعده ای سازگار نباشد و خطای نحـوی رخ دهـد ایـن تـابع فراخوانی می شود.

۲-declare_varible

زمانی که متغیر جدیدی تعریف می شود این تابع بررسی میکند که آیا این متغیر قـبلا تعریف شده یا خیر.اگر تعریف شده ب خطا رخ میدهد.

۳-constant_check

ببرسی می کند که متغیری که در حال تغییر آن هستیم آیا یک متغیر constatnt مـی باشد یا خیر.در صورتی که متغیر از نوع constatnt باشد خطا رخ می دهد.

۴-open_brace

با باز شدن پرانتز به متغیری که تعداد پرانتز ها را ذخیره میکند یکی اضافه می کند.

۵-close_brace

با بسته شدن پرانتز از متغیری که تعداد پرانتز ها را ذخیره می کند یکی کم می کند.

خطاها

در تعریف متغیر ها،متغیر float را نداشتیم ولی در تست کیس شماره ۱ متغیری با نوع float تعریف شده بود مه برای شناسایی آن در فایل lex.l آن را تعریف کرده و توکن مربوط را برای yacc ارسال میکنیم.

در تعریف حلقه for داشتیم:

```
<stmt> -> for <lvalue> ‘=’ <expr> ‘(‘ to | down ‘)’  <expr> do <block>
```

در صورتی که در تست کیس هایی که شامل حلقه ی for می باشد دو کاراکتر '(' ')' موجود نمی باشد و پس از رسیدن برنامه ی تحلیل لفوی به این مکان به دنبال ')' می گردد و چون موفق به پیدا کردن آن نمی شود خطا رخ می دهد برای مثال:

```
For I=0 to m do
```

بنابراین گرامر را به صورت زیر تغییر دادیم

```
<stmt> -> for <lvalue> ‘=’ <expr> <valfor> <expr> do <block>
<valfor> -> to| down to
```

مشخص کردن اولویت ها

با نوشتن دو خط کد زیر به ترتیب درست اولویت عملگر ها را مشخص میکنیم که کدام اولویت بیشتر یا کمتری داشته باشد

```
% left ADD SUB
% left MUL DEVIDE
```

حتی میتوانستیم به صورت زیر نیز بنویسیم که دارای اولویت بیشتری از همه ی آن ها باشد.

```
% right POW
```

ترتیب از بالا به پایین اولویت از کم به زیاد را نشان مـی دهـد و واژه ی left و right نشان دهنده ی اولویت چپ ترین و راست ترین ضرب،تقسیم ،جمـع و ... نسـبت بـه خودشان است.

خروجی برنامه پس از اجرا شدن فایل های تستی

Testcase1.txt : FAILD:line14

```
--||--lineNumber : 2
-- create new var #n# with type 1
ID  > IDDim
-- create new var #m# with type 1
ID  > IDDim
-- create new var #mat# with type 1
ID  > IDDim
IDDim '['IntNumber']'  > IDDim
IDDim '['IntNumber']'  > IDDim
-- create new var #mark# with type 1
ID  > IDDim
IDDim '['IntNumber']'  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Dec
--||--lineNumber : 3
ID  > IDList
Arg > Type IDList
ArgList > Arg
ArgList > ArgsList
--Debugging:add_function called
--||--lineNumber : 4
--open brace
--||--lineNumber : 5
-- create new var #i# with type 1
ID  > IDDim
-- create new var #j# with type 1
ID  > IDDim
IDDim  > IDDList
```

```
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Stmt
 --||--lineNumber : 6
Exp > IDD
Exp > IDD
IDD > lvalue
lvalue > Exp
IDD '[' Exp ']' > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
 --||--lineNumber : 7
Exp > IDD
IDD > lvalue
IntNumber > Exp
TO  > valfor
Exp > IDD
IDD > lvalue
lvalue > Exp
 --||--lineNumber : 8
Exp > IDD
Exp > IDD
IDD > lvalue
lvalue > Exp
IDD '[' Exp ']' > IDD
IDD > lvalue
lvalue > Exp
EQUAL > Logic
IntNumber > Exp
AND > Logic
Exp > IDD
Exp > IDD
IDD > lvalue
lvalue > Exp
IDD '[' Exp ']' > IDD
Exp > IDD
IDD > lvalue
```

```
lvalue > Exp
IDD '[' Exp ']' > IDD
IDD > lvalue
lvalue > Exp
Exp Logic Exp > Exp
Exp Logic Exp > Exp
'('Exp')' > Exp
 --||--lineNumber : 9
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp > ExpPlus
ExpPlus > ExpList
ID'('ExpList')'  > Exp
Exp > Stmt
Stmt > Block
IF Exp THEN Block > Stmt
Stmt > Block
FOR lvalue '=' Exp '('valfor')' Exp DO Block > Stmt
 --||--lineNumber : 10
Exp > IDD
Exp > IDD
IDD > lvalue
lvalue > Exp
IDD '[' Exp ']' > IDD
IDD > lvalue
lvalue > Exp
RETURN Exp > Stmt
 --||--lineNumber : 11
 < SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
 --close brace
Type ID '(' ArgsList ')' '{' SList '}' ';' > FuncDec
 --||--lineNumber : 12
No > FuncDecs
```

```
FuncDec And FuncDecs > FuncDecs
FuncDecs > Dec
No > DecList
Dec And DecList > DecList
Dec And DecList > DecList
 --open brace
 --||--lineNumber : 13
 -- Syntax Error : #n# is an already declared variable
ID  > IDDim
 -- Syntax Error : #m# is an already declared variable
ID  > IDDim
 -- Syntax Error : #i# is an already declared variable
ID  > IDDim
 -- Syntax Error : #j# is an already declared variable
ID  > IDDim
 -- create new var #c# with type 1
ID  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Stmt
 --||--lineNumber : 14
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp > Stmt
```

Testcase3.txt : OK

```
--||-- lineNumber : 2
 -- create new var #a# with type 1
ID  > IDDim
 -- create new var #b# with type 1
ID  > IDDim
 -- create new var #c# with type 1
ID  > IDDim
 -- create new var #i# with type 1
ID  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Dec
--||-- lineNumber : 3
No > DecList
Dec And DecList > DecList
-- open brace
--||-- lineNumber : 4
--||-- lineNumber : 5
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 6
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 7
Exp > IDD
IDD > lvalue
IntNumber > Exp
TO  > valfor
IntNumber > Exp
```

```
--||-- lineNumber : 8
-- open brace
--||-- lineNumber : 9
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
ADD > Aop
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp Aop Exp > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 10
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 11
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 12
 > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
-- close brace
'{' SList '} > Block
FOR lvalue '=' Exp '('valfor')' Exp DO Block > Stmt
--||-- lineNumber : 13
 > SList
```

```
        Stmt ';' SList > SList
        Stmt ';' SList > SList
        Stmt ';' SList > SList
        -- close brace
        program ID > program
```

Testcase4.txt : OK

```
--||-- lineNumber : 2
 -- create new var #a# with type 1
ID  > IDDim
 -- create new var #b# with type 1
ID  > IDDim
 -- create new var #c# with type 1
ID  > IDDim
 -- create new var #i# with type 1
ID  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Dec
--||-- lineNumber : 3
No > DecList
Dec And DecList > DecList
-- open brace
--||-- lineNumber : 4
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 5
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 6
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 7
```

```
        Exp > IDD
        IDD > lvalue
        lvalue > Exp
        LESSOREQ > Logic
        IntNumber > Exp
        Exp Logic Exp > Exp
        --||-- lineNumber : 8
        -- open brace
        --||-- lineNumber : 9
        Exp > IDD
        IDD > lvalue
        Exp > IDD
        IDD > lvalue
        lvalue > Exp
        ADD > Aop
        Exp > IDD
        IDD > lvalue
        lvalue > Exp
        Exp Aop Exp > Exp
        lvalue '=' Exp > Exp
        Exp > Stmt
        --||-- lineNumber : 10
        Exp > IDD
        IDD > lvalue
        Exp > IDD
        IDD > lvalue
        lvalue > Exp
        lvalue '=' Exp > Exp
        Exp > Stmt
        --||-- lineNumber : 11
        Exp > IDD
        IDD > lvalue
        Exp > IDD
        IDD > lvalue
        lvalue > Exp
        lvalue '=' Exp > Exp
        Exp > Stmt
        --||-- lineNumber : 12
        Exp > IDD
        IDD > lvalue
        Exp > IDD
```

```
IDD > lvalue
lvalue > Exp
ADD > Aop
IntNumber > Exp
Exp Aop Exp > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 13
 > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
-- close brace
'{' SList '} > Block
WHILE Exp DO Block > Stmt
--||-- lineNumber : 14
--||-- lineNumber : 15
 > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
-- close brace
program ID > program
```

Testcase5.txt : OK

```
--||-- lineNumber : 2
 -- create new var #a# with type 1
ID  > IDDim
 -- create new var #b# with type 1
ID  > IDDim
 -- create new var #c# with type 1
ID  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Dec
--||-- lineNumber : 3
No > DecList
Dec And DecList > DecList
-- open brace
--||-- lineNumber : 4
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 5
Exp > IDD
IDD > lvalue
IntNumber > Exp
lvalue '=' Exp > Exp
Exp > Stmt
--||-- lineNumber : 6
Exp > IDD
IDD > lvalue
lvalue > Exp
GREATER > Logic
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp Logic Exp > Exp
--||-- lineNumber : 7
```

```
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
ADD > Aop
--||-- lineNumber : 8
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp Aop Exp > Exp
lvalue '=' Exp > Exp
Exp > Stmt
Stmt > Block
--||-- lineNumber : 9
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
SUB > Aop
Exp > IDD
IDD > lvalue
lvalue > Exp
Exp Aop Exp > Exp
lvalue '=' Exp > Exp
Exp > Stmt
Stmt > Block
IF Exp THEN Block ELSE Block > Stmt
--||-- lineNumber : 10
--||-- lineNumber : 11
 > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
-- close brace
program ID > program
```

Testcase6.txt : OK

```
--||-- lineNumber : 2
 -- create new var #a# with type 1
ID  > IDDim
 -- create new var #b# with type 1
ID  > IDDim
 -- create new var #c# with type 1
ID  > IDDim
IDDim  > IDDList
IDDim  IDDList   > IDDList
IDDim  IDDList   > IDDList
Type IDDList > VarDec
VarDec > VarDecs
VarDecs > Dec
--||-- lineNumber : 3
No > DecList
Dec And DecList > DecList
-- open brace
--||-- lineNumber : 4
Exp > IDD
IDD > lvalue
IntNumber > Exp
TO  > valfor
IntNumber > Exp
--||-- lineNumber : 5
Exp > IDD
IDD > lvalue
IntNumber > Exp
TO  > valfor
IntNumber > Exp
--||-- lineNumber : 6
Exp > IDD
IDD > lvalue
Exp > IDD
IDD > lvalue
lvalue > Exp
ADD > Aop
Exp > IDD
IDD > lvalue
lvalue > Exp
```

```
Exp Aop Exp > Exp
lvalue '=' Exp > Exp
Exp > Stmt
Stmt > Block
FOR lvalue '=' Exp '('valfor')' Exp DO Block > Stmt
Stmt > Block
FOR lvalue '=' Exp '('valfor')' Exp DO Block > Stmt
--||-- lineNumber : 7
 > Stmt
--||-- lineNumber : 8
 > Stmt
--||-- lineNumber : 9
 > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
Stmt ';' SList > SList
-- close brace
program ID > program
```