هوالحکیم

دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده‌ی مهندسی برق و کامپیوتر

اصول طراحی کامپایلر
نیمسال دوم ۹۹-۱۳۹۸
http://courses.fouladi.ir/compiler

شرح پروژه شماره‌ی ۲

مرحله‌ی دوم

پروژه‌ی طراحی و پیاده‌سازی کامپایلر: ساخت پارسر

۱ پارسر: تحلیل‌گر نحوی

در این مرحله از پروژه، قصد داریم یک تحلیل‌گر نحوی (یا پارسر) برای زبان A طراحی و پیاده‌سازی کنیم.
برای این منظور،

- فایل YACC یا معادل آن Bison را به گونه‌ای آماده کنید که قواعد تولید استفاده شده برای تولید رشته‌ی ورودی را به ترتیب چاپ کند؛ یعنی در هنگام تجزیه، هرگاه از یک قاعده استفاده شد، آن را به صورت `lhs -> rhs` در یک خط مجزا چاپ کند.
- در صورت برخورد با اولین خطا، با چاپ عبارت `error` کار خاتمه می‌یابد.
- فایل ورودی کامپایلر یعنی برنامه‌ی منبع به زبان A با نام `test.a` و فایل ترتیب قواعد تولید با نام `test.txt` نام‌گذاری می‌شود.
- تابع مربوط به اسکنر با نام `yylex()` در داخل پارسر فراخوانی می‌شود و توکن بعدی را برمی‌گرداند.

گرامر زبان A

program	→	*block*
block	→	{ *decls stmts* }
decls	→	*decls decl* | ϵ
decl	→	*type* **id** ;
type	→	*type* [**num**] | **basic**
stmts	→	*stmts stmt* | ϵ
stmt	→	*loc* = *bool* ;
	|	**if** (*bool*) *stmt*
	|	**if** (*bool*) *stmt* **else** *stmt*
	|	**while** (*bool*) *stmt*
	|	**do** *stmt* **while** (*bool*) ;
	|	**break** ;
	|	*block*
loc	→	*loc* [*bool*] | **id**
bool	→	*bool* `||` *join* | *join*
join	→	*join* `&&` *equality* | *equality*
equality	→	*equality* `==` *rel* | *equality* `!=` *rel* | *rel*
rel	→	*expr* `<` *expr* | *expr* `<=` *expr* | *expr* `>=` *expr* | *expr* `>` *expr* | *expr*
expr	→	*expr* `+` *term* | *expr* `-` *term* | *term*
term	→	*term* `*` *unary* | *term* `/` *unary* | *unary*
unary	→	`!` *unary* | `-` *unary* | *factor*
factor	→	(*bool*) | *loc* | **num** | **real** | **true** | **false**

۲ راهنمایی برای اجرای برنامه

برای نمونه، گرامر زیر را در نظر بگیرید:

$$
\begin{array}{rcl}
line & \rightarrow & \textbf{eof} \\
 & | & expr\ \textbf{eof} \\
expr & \rightarrow & \textbf{num} \\
 & | & expr + expr \\
 & | & expr - expr \\
 & | & expr * expr \\
 & | & expr / expr \\
 & | & expr \hat{}\ expr \\
 & | & (expr)
\end{array}
$$

با مشخص کردن توکن‌های گرامر فوق و الگوی آنها، فایل LEX زیر را آماده می‌کنیم:

icalc.l

```
%{
#include <stdlib.h>
#include "y.tab.h"
%}

%option noyywrap

%%
"("      {printf("token found: <(>\n");   return('(');}
")"      {printf("token found: <)>\n");   return(')');}
"+"      {printf("token found: <+>\n");   return('+');}
"-"      {printf("token found: <->\n");   return('-');}
"*"      {printf("token found: <*>\n");   return('*');}
"/"      {printf("token found: </>\n");   return('/');}
"^"      {printf("token found: <^>\n");   return('^');}
"\n"     {printf("token found: <EOF>\n"); return('\n');}
[0-9]+   {yylval = atoi(yytext);
          printf("token found: <NUM,%d>\n",yylval); return(NUM);}
.        {printf("invalid token!\n");}
%%
```

سپس فایل YACC را برای گرامر می‌نویسیم:

icalc.y

```
%{
#include <math.h>
#include <stdio.h>
%}

%token NUM
%left '-' '+'
```

```
%left '*' '/'
%right '^'

%%
line  : '\n'
      | expr '\n'      {printf("%s\n", "line -> expr");}
      ;
expr  : NUM            {printf("%s\n", "expr -> NUM");}
      | expr '+' expr  {printf("%s\n", "expr -> expr + expr");}
      | expr '-' expr  {printf("%s\n", "expr -> expr - expr");}
      | expr '*' expr  {printf("%s\n", "expr -> expr * expr");}
      | expr '/' expr  {printf("%s\n", "expr -> expr / expr");}
      | expr '^' expr  {printf("%s\n", "expr -> expr ^ expr");}
      | '('expr')'     {printf("%s\n", "expr -> (expr)");}
      ;
%%

#include "lex.yy.c"

yyerror() {printf("input error!");}

int main()
{
    yyparse();
    return 0;
}
```

حال با فرض اینکه پوشه‌ی حاوی کامپایلر C (gcc.exe) و پوشه‌ی win_flex_bison-latest (حاوی win_flex.exe و win_bison.exe) در PATH ویندوز تعریف شده باشد یا آنها را در پوشه‌ی جاری قرار داده باشیم، فایل batch زیر را اجرا می‌کنیم:

run.bat

```
win_flex icalc.l
win_bison -dy icalc.y
gcc y.tab.c -o icalc.exe
pause
```

فایل پارسر icalc.exe هم‌اکنون آماده است. ورودی را در فایل input.txt قرار می‌دهیم:

input.txt

```
12*(5+3)/140
```

فایل batch زیر را اجرا می‌کنیم:

parse.bat

```
(icalc.exe <input.txt) >output.txt
```

فایل خروجی زیر به‌دست می‌آید:

output.txt

```
token found: <NUM,12>
expr -> NUM
token found: <*>
token found: <(>
token found: <NUM,5>
expr -> NUM
token found: <+>
token found: <NUM,3>
expr -> NUM
token found: <)>
expr -> expr + expr
expr -> (expr)
token found: </>
expr -> expr * expr
token found: <NUM,140>
expr -> NUM
token found: <EOF>
expr -> expr / expr
line -> expr
```

* فایل‌های نمونه‌ی فوق بر روی سایت درس قرار داده شده است.

۳ تحویل پروژه مرحله‌ی دو

تحویل پروژه مرحله‌ی دو از طریق آپلود در سایت می‌باشد. فایل‌های زیر (مشابه برنامه‌ی icalc توضیح داده شده در قسمت قبل) را در یک فایل zip با نام phase2 قرار دهید و در محل مشخص‌شده در سایت درس آپلود کنید:

project.l	*scanner LEX/FLEX file*
lex.yy.c	*generated C file for scanner using FLEX compiler*
project.y	*parser YACC/BISON file*
y.tab.c	*generated C file for parser using BISON compiler*
y.tab.h	*generated C header file for parser using BISON compiler*
project.exe	*parser executable file*
run.bat	*batch file for running FLEX, BISON, and GCC to get exe file*
parse.bat	*batch file for getting input and giving output*
test.a	*input file (in A language)*
test.txt	*output file*

phase2.zip

برای فایل test.a می‌توانید از کد نمونه‌ی زیر استفاده کنید:

test.a

```
{
   int i; int j; float[100] a; float v; float x;
   while ( true ) {
       do i = i+1; while ( a[i] < v );
       do j = j-1; while ( a[j] > v );
       if ( i >= j ) break;
       x = a[i]; a[i] = a[j]; a[j] = x;
   }
}
```

کامپایلر Bison و کتاب راهنمای آن به همراه منابع مفید دیگر در سایت درس قرار داده شده است.

۴ مرجع

برای آشنایی بیشتر با زبان‌های منبع و مقصد و جزئیات بیشتر در مورد آنها فصل ۲ و پیوست A از کتاب اصلی درس را ببینید:

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, **Compilers: Principles, Techniques and Tools**, Second Edition, Addison-Wesley, 2007.

همچنین راهنمایی‌های لازم برای استفاده از ابزارهای Flex/Bison از منبع زیر قابل دریافت است:

J. R. Levine, **flex & bison**, O'Reilly Media, Inc., 2009.