راهنمای استفاده از ابزار ParsGn

این برنامه گرامر یک زبان برنامه سازی را در قالب BNF دریافت کرده و با استفاده از روش های LR یک کامپایلر به صورت خودکار ایجاد می کند. کامپایلر ایجاد شده حاوی یک سری کد به زبان C++ است که به روش Table Driven از جدول تجزیه LR استفاده می کند. پس از ایجاد برنامه خودکار، کاربر می تواند قسمت های Scanner و قسمت CG (تحلیل معنایی و تولید کد) را متناسب با زبان برنامه سازی به صورت دستی تغییر دهد. مراحل اصلی ایجاد یک کامپایلر به شرح زیر است:

۱) ایجاد فایل گرامر BNF
۲) تولید خودکار کامپایلر و آزمون آن
۳) تغییرات دستی کامپایلر

مرحله ۱: ایجاد فایل گرامر BNF

در شاخه Samples تعدادی فایل با پسوند bnf به عنوان نمونه دیده می شود. (در قسمت ***ضمیمه*** نیز این فایلها آورده شده است.) هر فایل bnf دارای دو قسمت <tokens> و <bnf> است. بعد از نماد <tokens> کلیه پایانه ها معرفی می شوند. تمامی کلمات مفید به جز EOF اعم از علائم ویژه، کلمات کلیدی و ... باید به صورت رشته ای در داخل گیومه تعریف شوند. این رشته ها در یک یا چند خط قابل تعریف هستند. (نکته: برنامه به کلیه کلمات به ترتیب از ۱ به بعد شماره دهی می کند. برای EOF با وجودی که تعریف نمی شود به صورت پیش فرض، شماره صفر تخصیص می یابد.)

در قسمت <bnf> کلیه قواعد گرامر تعریف می شوند. هر قاعده شامل یک ناپایانه است و پس از آن یک علامت =:: و پس از آن جملات سمت راست با جداکننده های | آورده می شوند. در پایان هر قاعده آوردن یک نقطه الزامی است.

در داخل جملات برای اختیاری بودن از [ ] ، برای تکرار صفر یا بیشتر از { } ، برای نماد ε از & و برای فاکتور گیری از ( ) می توان استفاده کرد.

در این جملات پایانه ها باید داخل گیومه و ناپایانه ها بدون گیومه باشند. کنش های مفهومی نیز با یک علامت @ شروع می شوند. طول پایانه ها، ناپایانه ها و کنش های مفهومی از ٤٠ حرف متجاوز نباید باشد. کنش های مفهومی در هر جای گرامر قابل درج هستند و حتی چند کنش پشت سر هم نیز مجاز است.

در داخل فایلهای bnf می توان از comment تک خطی // نیز استفاده کرد.

مرحله ۲: تولید خودکار کامپایلر و آزمون آن

با اجرای برنامه ParsGn.exe مسیر فایل bnf از کاربر پرسیده می شود. نام فایل bnf طولی حداکثر ٦ حرف باید داشته باشد. مثلاً expr2.bnf و یا c:\Samples\expr2.bnf مسیرهای درستی هستند، اما exprsn2.bnf معتبر نیست.

پس از وارد سازی مسیر، یک عدد بیانگر نوع روش تجزیه LR باید وارد شود. عدد ۱ معادل SLR، عدد ۲ معادل LALR و عدد ۳ معادل CLR است. بهترین وضعیت برای زبانهای برنامه سازی روش ۲ است.

برای اجرای برنامه می توان به صورت خط فرمان نیز عمل کرد:

```
ParsGn.exe  c:\Samples\expr2.bnf  2
```

با اجرای برنامه پیغام هایی روی صفحه نمایش چاپ شده و یک سری فایل ساخته می شود:

۱) **expr2.log** : در فایلهای log موفقیت برنامه و یا بروز خطا در قالب Error و یا Warning گزارش می شود. در صورت مشاهده Error باید فایل BNF را بازبینی و تصحیح نمود. در صورت بروز خطا عملیات متوقف شده و هیچ کدی تولید نمی شود. مهمترین خطاهای متداول عبارتند از:

– عدم رعایت syntax فایل های bnf مانند فراموش کردن یک نقطه در انتهای قواعد گرامر. در این صورت نوع خطا و حتی نحوه تصحیح خطا گزارش می شود.

– استفاده از یک ناپایانه در سمت راست یک قاعده و عدم تعریف آن در قواعد بعدی. در این صورت آن ناپایانه به کاربر نشان داده می شود.

مهمترین اخطارهای متداول عبارتند از:

– عدم تعریف پایانه در قسمت <tokens> و استفاده از آن در قسمت <bnf>. در این صورت سیستم به طور خودکار آن را تعریف می کند و عملیات با موفقیت انجام می شود، اما بازبینی فایل bnf توصیه می شود.

– ابهام (conflict) های موجود در روش تجزیه LR مانند shift-reduce و یا reduce-reduce و در صورت تعریف کنش های مفهومی ابهام نوع action-action در زبانهای مختلف برنامه سازی امکان دارد. برنامه به صورت خودکار در موقع بروز ابهام، یک حالت پیش فرض را انتخاب کرده و بر اساس آن جدول تجزیه را تکمیل می کند، اما این کار ممکن است مورد رضایت کاربر نباشد. کاربر می تواند ابهام های شناسایی شده را پیگیری کرده و با نظر خود آنها را رفع کند. (مراجعه به قسمت *امکانات پیشرفته تر*، مبحث رفع ابهام)

۲) **expr2.cpp** : این برنامه کد اصلی کامپایلر است. در داخل این کد، در قسمت main کدی مانند زیر دیده می شود:

```
int main(/*int argc, char* argv[]*/)
{
 Initialize(/*argc,argv*/);
 if (PG_Parser( ))
   printf("\nSuccess\n");
 Finalize(/*argc,argv*/);
 return 0;
}
```

تابع PG_Parser به طور خودکار پیاده سازی شده و بدنه توابع Initialize و Finalize در فایل های سرآیند expr2SC.h و expr2CG.h تعریف شده اند و قابل پیاده سازی هستند. توصیه می شود که قسمت های بالای تابع main به هیچ وجه مورد تغییر قرار نگیرند.

نکته: کد تولید شده برای TC طراحی شده است. در صورتی که بخواهیم از این کدها در VC استفاده کنیم باید یک پروژه از نوع win32 console ساخته و پس از خط اول آن یعنی:

```
#include “stdafx.h”
```

کلیه خطهای این برنامه را کپی کرد. البته باید سه خط زیر را comment کرد:

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

۳) **expr2.tbl** : این فایل حاوی اطلاعات جدول تجزیه است. پس از آنکه از روی فایل expr2.cpp فایل اجرایی expr2.exe ساخته شد، در زمان اجرای کامپایلر این فایل باید دقیقاً در شاخه جاری حضور داشته باشد. کامپایلر در زمان اجرا آن را باز کرده و محتوای آن را در حافظه load کرده و مورد استفاده قرار می دهد.

٤) **expr2Tb.h** : متناظر با فایل expr2.tbl یک فایل حاوی یک آرایه باینری به طور خودکار ایجاد می شود. در بالای فایل expr2.cpp یک خط به صورت زیر دیده می شود:

```
//#define PG_INLINE_TABLE
```

اگر comment این خط را بردارید و برنامه را کامپایل کنید، در زمان اجرا حضور expr2.tbl ضرورتی نخواهد داشت و جدول تجزیه تعریف شده در فایل expr2Tb.h به صورت inline در دل برنامه اجرایی قرار می گیرد.

در صورتی که گرامر زبان برنامه سازی بزرگ باشد، در محیط TC به علت بزرگ بودن این فایل باینری برنامه کامپایل نخواهد شد. در این صورت برنامه باید توسط VC مورد استفاده قرار گیرد و یا خط اول فایل cpp غیر فعال مانده و از فایل tbl در زمان اجرا استفاده شود.

۵) **expr2Sc.h** : متناظر با قسمت <tokens> در فایل bnf یک Scanner پیش فرض پیاده سازی می شود. در بالای این فایل یک enum با نام TokenType پیاده سازی شده است. متناسب با ترتیب قرار گرفتن پایانه ها در قسمت <tokens> فایل bnf شماره دهی انجام شده است. در پایین، تابعی به نام TokenType Scanner( ) ایجاد شده است. این برنامه برای تست عملکرد پارسر مناسب است و نهایتاً توسط کاربر باید تغییر یابد.

٦) **expr2Cg.h** : در این فایل کنش های مفهومی پیاده سازی می شوند. در بالای فایل متناظر با کلیه کنش های مفهومی یک enum تعریف شده است. در تابع CG متناسب با یک شماره، یک تابع متناظر فراخوانی می شود. به ازای هر کنش، یک تابع همنام دیده می شود که در داخل آن یک printf دیده می شود. محتوای این توابع برای تست عملکرد پارسر مناسب است و نهایتاً توسط کاربر باید تغییر یابد.

۷) **expr2Er.h** : این فایل نقش Error Handler پارسر را به عهده دارد. گزارش خطاهای نحوی در این فایل دیده شده است. یکی دیگر از کارهای این فایل، تصحیح خطا است. برنامه به محض برخورد با خطا متوقف نمی شود بلکه تا حد امکان سایر خطاها را نیز گزارش می کند. پس از برخورد با هر خطا روشی برای تصحیح آن پیشنهاد می کند، به این شکل که بیان می کند چه کلماتی باید حذف و چه کلماتی اضافه شوند تا برنامه از لحاظ نحوی درست شود. (در این قسمت نیز می توان با انجام تغییراتی قابلیت های کشف و تصحیح خطا را بهینه کرد که در قسمت *امکانات پیشرفته تر* مورد بررسی قرار می گیرد.)

۸) **expr2.rep** : این فایل فقط جنبه توضیحی دارد و در آن گزارش کاملی از مراحل تولید جدول تجزیه LR دیده می شود. در بالای آن، گرامر معادل با فایل bnf دیده می شود که یک گرامر مستقل از متن است و از بسط قواعد بدست آمده. این قسمت که مابین دو comment محصور شده است، کاملاً مشابه یک فایل bnf (اما فاقد علائم خاص) است و می توان آن را کپی کرد و در قالب یک فایل bnf مورد استفاده قرار داد.

پس از گرامر، مجموعه های First و Follow نیز معرفی شده اند. پس از آن کلیه State های اتوماتای LR و یالهای خارج شده از آنها مورد نمایش قرار گرفته اند. در صورت وجود ابهام در قالب warning نوع ابهام بیان شده است. در پایان فایل، اطلاعاتی جهت تصحیح خطا نمایش داده شده است.

۹) در صورتی که فایل bnf فاقد کنش مفهومی باشد، فایلی مانند expr1Tr.cpp ایجاد می شود که قادر است کلیه درخت های تجزیه متناظر با یک جمله را (حتی اگر گرامر مبهم باشد) پیدا کرده و به

صورت متنی مورد نمایش قرار دهد. از این فایل ها در حیطه پردازش زبانهای طبیعی می توان استفاده نمود.

آزمون صحت کامپایلر: پس از آنکه محصولات ParsGn با موفقیت تولید شد، با کامپایل فایلی مانند expr2.cpp و اجرای آن یک صفحه سیاه ظاهر می شود. کلماتی را وارد کرده و نهایتاً متناظر با EOF کلید Ctrl+Z را فشار می دهیم تا جمله خاتمه یابد. کلمات باید از هم فاصله داشته باشند و عیناً کلمات تعریف شده در قسمت <tokens> باشند. مثلاً

id - id / (id * num) + num ^Z

کامپایلر شروع به کار کرده و پس از اجرای یک سری کنش مفهومی با موفقیت خاتمه می یابد:

```
@pushid
@sub
@pushid
@div
@pushid
@mult
@pushnum
@make
@make
@make
@add
@pushnum
@make

Success
```

اما اگر کلمات وارد شده به صورت زیر باشد، به محض برخورد با خطا گزارش آن و تصحیح خطا اجرا می شود:

id id / (id * num) + + num ^Z

```
Parser Error: 2-th Token (“id”) ,  in state 17
  Deleted  Tokens->
  Inserted Tokens->  +
Parser Error: 10-th Token (“+”) ,  in state 28
  Deleted  Tokens->
  Inserted Tokens->  id
```

با این حساب جمله صحیحی که مورد پیشنهاد کامپیایلر بوده به صورت زیر است:

id + id / (id * num) + **id** + num ^Z

با وارد سازی جملات درست و گرفتن Success و یا وارد سازی جملات غلط و گرفتن Error می توان از صحت گرامر تعریف شده اطمینان حاصل کرد. در صورت معلوم شدن خطای گرامر باید فایل bnf تصحیح و اجرای ParsGn مجدداً انجام گیرد.

مرحله ۳: تغییرات دستی کامپایلر

چنان که قبلاً گفته شد، قسمت های expr2Sc.h و expr2Cg.h باید توسط کاربر مورد تغییر قرار گیرد.
در بالای فایل expr2Sc.h یک enum تعریف شده است. بدون جابجا کردن محل آنها می توان اسامی خواناتری انتخاب کرد، مثلاً برای TOKEN1_ نام ID_ را می توان جایگزین کرد.
در تابع Initialize کارهایی مانند باز کردن فایل مورد کامپایل و بافر کردن آن در حافظه جهت شروع به کار Scanner انجام می شود.
در بالای فایل، تمام متغیرهای سراسری مانند IdValue (جهت ذخیره محتوای نشانه ها) و یا LineCounter (جهت شمارنده خطهای فایل ورودی) و یا سایر ساختمان داده ها باید تعریف شوند.
تابع Scanner را باید به گونه ای تغییر داد که بتواند فقط یک کلمه موجود در فایل ورودی را از نوع TokenType برگرداند و قبل از آن یک سری متغیرهای سراسری مانند IdValue را نیز پر کند. مدیریت خطاهای لغوی نیز در همین تابع توسط کاربر دیده شود.
در فایل expr2Cg.h یک سری توابع متناظر با کنش های مفهومی وجود دارد. کد داخل آنها باید پاک شده و کار با یک سری ساختمان داده ها که تحلیل معنایی و تولید کد میانی را انجام می دهند، جایگزین شود. تمام ساختمان داده های لازم مانند Symbol Table قبل از این توابع باید تعریف و پیاده سازی شوند.
توجه مهم: در مواردی معلوم می شود که باید کنش های مفهومی جدیدی اضافه شود و یا محل آنها تغییر یابد. در این موارد فایل bnf باید تغییر یابد و ParsGn مجدداً اجرا شود. با این کار کلیه کدهای نوشته شده rewrite می شود و کد پیش فرض جایگزین می شود. برای جلوگیری از این مشکل باید یک backup از کدهای موجود در فایل های expr2Sc.h و expr2Cg.h گرفته شده و سپس ParsGn اجرا شده و سپس کدهای واقعی را جایگزین کرد.

امکانات پیشرفته تر

الف) رفع ابهام:

گرامر مبهم ifthn2.bnf را در نظر بگیرید. با اجرای ParsGn در فایل ifthn2.log پیغام زیر را مشاهده خواهید کرد که بیانگر یک ابهام shift-reduce است:

```
Warning! Conflict at Node 10 token "else"  =>  1) S13    2) R5
```

با مراجعه به فایل ifthn2.rep و جستجوی کلمه warning در آن به این قسمت خواهید رسید:

```
Node 10:
Stm --> "if" Stm01 "then" @start_if Stm @end_if  . Stm02   { <*EOF*> "else" }
Stm02 -->  .  { <*EOF*> "else" }                    //reduce//
Stm02 --> . Stm03   { <*EOF*> "else" }
Stm03 --> . "else" @correct_if Stm @end_if   { <*EOF*> "else" }          //shift//
----links----
10 -- Stm02  --> 11
10 -- Stm03  --> 12
10 -- "else"  --> 13
----Shif Part----
t0  <*EOF*>:  R5
t4  "else":  S13    R5
----Goto Part----
n3  Stm02 :  G11
n4  Stm03 :  G12
--------------------
Warning! Conflict token "else"  =>  1) S13    2) R5
```

با دقت در این وضعیت مشخص می شود که برای آنکه else متعلق به نزدیکترین if باشد لازم است که shift به جای reduce انجام شود. پیش فرض برنامه ParsGn نیز بر اولویت shift است و از این رو کامپایلر ساخته شده نیاز به هیچ تغییری ندارد. اما اگر بخواهیم کاری کنیم که reduce ارجحیت داشته باشد، باید به فایل ifthn2.tbl مراجعه کنیم. با جستجوی Node 10 در فایل tbl به این قسمت می رسیم:

```
Node  10
2
   0   1   R5
   4   2   S13    R5            ←
2
  3   G11
  4   G12
```

در خط مشخص شده کافی است جای S13 را با R5 جابجا کرد:

```
   4   2   R5   S13            ←
```

در این صورت پارسر در این وضعیت با دیدن else از R5 استفاده خواهد کرد.

با اعمال این تغییرات دیگر نمی توان از جدول inline استفاده کرد و باید فایل tbl در زمان اجرا حضور داشته باشد. اما با استفاده از ابزار TBL2BIN.exe می توان از روی فایل tbl فایل سرآیند Tb.h متناظر را ایجاد کرد و نهایتاً از جدول باینری آن به صورت inline استفاده کرد.

با اجرای این برنامه یک مسیر از کاربر پرسیده می شود و از روی آن سرآیند ساخته می شود. به صورت خط فرمان نیز این کار امکان پذیر است. مثلاً :

```
tbl2bin.exe   c:\Samples\ifthn2.tbl
```

ب) تغییرات در قسمت تصحیح خطا:

الگوریتم به کار رفته به این شکل است که کامپایلر پس از رسیدن به یک وضعیت و برخورد به کلمه ای که از آن وضعیت نمی تواند حرکت کند، وارد مرحله تصحیح خطا می شود. در این حالت سعی می کند، صفر یا بیشتر کلمه نامناسب را حذف کند و در صورت نیاز صفر یا بیشتر کلمه را درج کرده تا نهایتاً از آن وضعیت بتوان تا چند حرکت بعد (مثلاً v حرکت) را بدون برخورد با خطا پیمایش کرد.

به صورت پیش فرض، درج و حذف هر کلمه هزینه 1 دارد، اما با تغییر این هزینه ها می توان تصحیح خطای هوشمندانه تری را ایجاد کرد. برای این منظور باید در فایل bnf در مقابل کلمات تعریف شده در قسمت <tokens> هزینه درج و حذف را معرفی نمود. برای معرفی هزینه درج عددی صحیح بین 1+ تا 100+ و برای هزینه حذف یک عدد بین 1- تا 100- در جلو کلمه نوشته می شود. در صورتی که در مقابل کلمه هیچ عددی قرار نگیرد فرض بر هزینه درج 1+ و حذف 1- است. در فایل expr3.bnf طریقه نوشتن این هزینه ها قابل مشاهده است. (توجه: در جلو کلمه می توان فقط یک عدد مثبت یا منفی در نظر گرفت که در این صورت برای عدد دیگر مقدار پیش فرض 1+ یا 1- در نظر گرفته می شود. همچنین الزامی ندارد که عدد مثبت اول نوشته شود و علامت عدد گویای نوع هزینه می باشد.)

پس از تولید محصولات با اجرای برنامه expr3.cpp و وارد سازی جمله خطا دار زیر:

```
id   id   /   (   id   *   num   )   +   +   num   ^Z
```

نتیجه زیر بدست می آید:

```
Parser Error: 2-th Token (“id”) ,  in state 17
  Deleted  Tokens->
  Inserted Tokens->  *
Parser Error: 10-th Token (“+”) ,  in state 28
  Deleted  Tokens-> +
  Inserted Tokens->
```

اگر این نتیجه را با نتیجه حاصل از فایل expr2.cpp برای همین جمله مقایسه کنید، می بینید که به جای درج + از درج * که با هزینه کمتری معرفی شده است، استفاده شده است.

حال یک جمله غلط دیگر را با expr3.cpp وارد می کنیم:

id id id id id id ^Z

نتیجه حاصل به صورت زیر است:

```
Parser Error: 2-th Token (“id”) ,  in state 17
  Deleted  Tokens-> id id id id
  Inserted Tokens->  *
```

در بالای فایل expr3Er.h خطی مشابه زیر تعریف شده است که تعداد حرکت های مطمئن بعد از خطا را تعریف می کند:

```
#define  PG_THRESHOLD_V  5
```

اگر این عدد را تبدیل به 2 کرده و برنامه را با جمله پرخطای فوق اجرا کنیم، نتیجه زیر بدست می آید:

```
Parser Error: 2-th Token (“id”) ,  in state 17
  Deleted  Tokens-> id
  Inserted Tokens->  *
Parser Error: 4-th Token (“id”) ,  in state 17
  Deleted  Tokens-> id
  Inserted Tokens->  *
Parser Error: 6-th Token (“id”) ,  in state 17
  Deleted  Tokens-> id
  Inserted Tokens->  *
```

چنان که مشاهده می شود، با کاهش این مقدار ثابت تعداد خطاهای گزارش شده بیشتر می شود اما با افزایش آن تعداد مراحل بعدی مورد پیش بینی بیشتر می شود. انتخاب مقدار بهینه و مناسب، کاملاً تجربی است و به زبان برنامه سازی مورد کامپایل بستگی دارد.

ضمیمه ۱: کد فایل های bnf موجود در شاخه Samples

expr1.bnf :

```
<tokens>
"id"   "num"    "+"    "-"
"*"    "/"    "("    ")"
<bnf>
Exp ::= Term { ( "+" | "-" ) Term } .    //expression
Term::=Fact{("*"|"/")Fact}.
Fact ::= "id" | "num" | "(" Exp ")". //factor
```

expr2.bnf :

```
<tokens>
"id"  "num"  "+"  "-"  "*"  "/"  "("  ")"
<bnf>
Exp ::= Term { ( "+" @add | "-" @sub ) Term @make } .
Term ::= Fact { ( "*" @mult | "/" @div ) Fact @make } .
Fact ::= "id" @pushid | "num" @pushnum | "(" Exp ")" .
```

expr3.bnf :

```
<tokens>
"id"  +10 -10
"num" +5  -5
"+"   +2  -2
"-"   +2  -2
"*"   +2  -2
"/"   +2  -2
"("   +1  -1
")"   +1  -1
<bnf>
Exp ::= Term { ( "+" @add | "-" @sub ) Term @make } .
Term ::= Fact { ( "*" @mult | "/" @div ) Fact @make } .
Fact ::= "id" @pushid | "num" @pushnum | "(" Exp ")" .
```

ifthn1.bnf :

```
<tokens> .
"other"  "if"  "then"  "else"  "true"  "false"
<bnf>
Stm ::= "other"  |
           "if" ("true"|"false") "then" Stm ["else" Stm] .
```

ifthn2.bnf :

```
<tokens>
"other"  "if"  "then"  "else"  "true"  "false"
<bnf>
Stm ::= "other"  @do_other |
        "if" ("true" @process_true |"false" @process_true)
        "then" @start_if Stm @end_if ["else" @correct_if Stm @end_if] .
```

ضمیمه ۲: توصیف نحو فایل های **bnf** با خود **bnf** !!

```
<tokens>
  "<tokens>"  "<bnf>"  "term"  "nterm"  "action"  "num"
  ":"   "::="   "."   "|"   "&"
  "["   "]"   "{"   "}"   "("   ")"
<bnf>
  Program ::= "<tokens>"  TermList  "<bnf>"  RuleList .
  TermList ::= { "term"["num"]["num"] } .
  RuleList ::= { Rule } .
  Rule ::= ["num" ":"]  "nterm"  "::="  Exp  "." .
  Exp ::= Exp "|" T  |  T .
  T ::= T  F | F .
  F ::= "term" | "nterm" | "action" | "&"
        |   "["  Exp  "]"
        |   "{"  Exp  "}"
        |   "("  Exp  ")"  .
```