گزارش پروژه نهایی درس اصول طراحی کامپایلر

استاد: دکتر منصوری

حسین خادمیان ۹۸۳۰۳۳۹

طراحی نهایی این پروژه از دو قسمت Scanner و Parser می شود.

همچنین متن پروژه با زبان C نوشته شده که فایل‌های shell script متناسب با کامپایلر gcc به همراه چندین تست کیس جهت تست ورودی ها همراه آن وجود دارد.

بخش اسکنر جهت تبدیل رشته ورودی به توکن های با معنی مجزا جهت ارسال به پارسر استفاده می شود. در این قسمت پیاده سازی تبدیل اعداد به مدل نوشتاری صورت میگیرد.

پارسر جهت پردازش اصلی و تبدیل عبارت ریاضی به کد میانی استفاده می شود.

Parser:

منطق پارسر با توجه به ساده سازی ورودی در اسکنر به شدت ساده شده و طبق الگو های حل شده در کلاس فقط نیاز به رعایت اولویت عملگر های * / در مقابل + – و همینطور اصل شرکت پذیری چپ به راست همه این‌ها بودم.

جهت قوانین هر پروداکشن هم همه پروداکشن ها یک رشته ساده تولید می‌کنند با این نکته که:

پروداکشن expr → number مقداری که در yylval قرار میگیرد جهت چاپ نهایی نیاز به eval شدن (ریخته شدن در یک متغیر دیگر) دارد اما بقیه پروداکشن ها به صورت کامل eval می‌شوند یعنی پس از محاسبه مقدار درون یک temp ذخیره شده اند.

بنابراین با متغیر evaled نسبت به ثبت وضعیت ذخیره شدن اقدام می‌کنم و این مقدار در هر پروداکشن بروز نگه داشته می شود. در آخر پروداکشن s→expr که وظیفه چاپ خروجی را دارد اگر با رسیدن به این پروداکشن evaled نشده بود انرا به صورت مجزا eval میکنیم و در آخر در خروجی چاپ می کنیم.

```
expr
   : NUMBER {  $$ = $1 = yylval from lex; evaled = 0;  }
   | expr PLUS  expr { $$ = eval(...); evaled = 1;  }
   | expr MINUS expr { $$ = eval(...); evaled = 1; }
   | expr MULT  expr { $$ = eval(...); evaled = 1; }
   | expr DIV   expr { $$ = eval(...); evaled = 1; }
   | LPAR expr  RPAR { $$ = $2; evaled = what expr is;   }
;
%start S : expr  { if (not evaled) $1=eval($1);     print_var($1); }
```

تابع main موجود در این پارسر ابتدا گزارش های flex را غیر فعال میکند و بعد پارسر را صدا میزند.

اسکنر:

در این بخش که از flex جهت پیاده سازی آن استفاده شده است سه ورژن مختلف از قوانین پیاده سازی کردم. که بهترین آن ورژن ۳ می شود. اما ورژن ۲ و یک هم هردو پاسخ صحیح متن سؤال را تولید می کنند.

ورژن ۱: در این ورژن اسکنر به طور کلی همه اعداد را یکجا طبق الگوی [۰-۹]+ دریافت می کند. سپس از پرارزش ترین رقم شروع به پردازش رشته عددی می کند. بر اساس جایگاه عدد (yyleng که نشان دهنده طول رشته و طول رشته نشان دهنده جایگاه رقم پرارزش است) تصمیم به ترکیب عبارت از پیش محاسبه شده و رقم جاری می شود. کل این فرایند در دو متغیر prev نگدارنده مقدار گذشته و digit مقدار رقم فعلی محاسبه می شود و پاسخ در yylval ثبت می شود. در انتها اگر طول رشته عددی ۱ باشد یعنی پردازش متن عدد تمام شده و توکن NUMBER تولید می شود. در غیر اینصورت با yyless(1) یک رشته از چپ (برارزش ترین رقم که محاسبه کردیم) جداشده و بقیه رشته جهت پردازش دوباره به پارسر داده می شود تا طبق همین الگو دوباره وارد حلقه پردازشی شود.

123 → digit=1 prev=nothing yylval=OneHun_ → 23

23 → digit=2 prev=OneHun_ yylval=OneHun_TwoTen_ → 3

3 → digit=3 prev=OneHun_TwoTen_ yylval=OneHun_TwoTen_Thr

→ NUMBER

در این شیوه همینطور که واضح است از کد نویسی جهت محاسبه جایگاه ارقام استفاده شده یعنی به طور دستی جایگاه ۱ ۲ ۳ ۴ ۵ ۶ جهت نوشتن معادل متنی به کار گرفته شده اند.

ورژن ۲: در این نسخه با توجه به ایده و الگوی مشاهده شده در نسخه یک گرامر توکن ها حالت هوشمند تری بیدا می کنند. الگو های :

Ones: ارزش مکانی یکان هرگروه هزارتایی است. این الگو شامل یک رقم عدد ۰ تا ۹ می شود.

Tens: از کنار هم قرار گرفتن دو رقم Ones در کنار هم الگوی دهگان تشکیل می شود.

Huns: نماینده الگوی صدگان است که یک رقم یکی در قسمت برارزش و دو رقم نماینده رهگان دارد.

ones: [0-9] **tens**: {ones}{ones} **huns**: {ones}{tens}

نحوه محاسبه رشته معادل هر کدام از این قسمت‌ها هم مثل روش اول ابتدا رقم ones پرارزش تبدیل می‌شود و با توجه به جایگاه (یکان دهگان صدگان) که از اسم هر الگو مشخص است به آن اضافه می‌شود و باقی عدد با yyless(1) به ارزش مکانی پایینتر جهت پردازش فرستاده میشود. در آخر اگر عدد به یک رقم بخورد دیگر ادامه نمیدهیم و توکن NUMBER با مقدار رشته معادل برگشت داده می شود.

جهت اعداد دارای ارزش مکانی هزارگان (یکان هزار – دهگان هزار – صدگان هزار) از الگو های بالا استفاده کردم به اینصورت که:

?{huns}{tens}{tens} این اگو که کوتاه شده دو الگوی {tens} و {huns}{tens} است جهت پردازش رقم دهگان در جایگاه مکانی دهگان یا دهگان–هزار استفاده می شود.

همین شیوه جهت پیاده سازی جایگاه مشترک صدگان و صدگان–هزار با الگوی ?{huns}{huns}{huns} استفاده شده است.

جهت پیاده سازی ارزش مکانی یکان و یکان–هزار مثل قبل از الگو مشترک ?{huns}{ones} استفاده کردم با این تفاوت که اگر طول اگوی یافت شده ۱ بود یعنی به انتهای مسیر بررسی رسیدیم و توکن NUMBER با رشته تولید شده در yylval بازگردانده می شود. در غیر اینصورت دور عبارتی که تا این مرحله به دست آمده را برانتز میگیریم.

54321 → match {tens}{huns}? yylval=FivTen_ **yyless(1)**→ 4321

4321 → match {ones}{huns}? yylval=(FivTen_Thr)Tho_ **yyless(1)**→ 321

321 → match {huns}{huns}? yylval=(FivTen_Thr)Tho_ThrHun_ **yyless(1)**→ 21

21 → match {tens}{huns}? yylval=(FivTen_Thr)Tho_ThrHun_Two_ **yyless(1)**→ 1

1 → match {ones}{huns}? yylval=(FivTen_Thr)Tho_ThrHun_Two_One

→ **NUMBER**

این نسخه با توجه به این پیش دانسته که کاربر همیشه ورودی درست وارد میکند پاسخ کامل است اما اگر منطق کار (اصل منطق برانتز گذاری در الگوی ?{huns}{ones} است) برای اعداد میلیون به کار گرفته شود شیوه پرانتز گزاری مناسب نیست و در کل میتوان گفت پاسخ عمومی نمیدهد.

<u>ورژن ۳:</u> در این نسخه مثل نسخه قبل از الگوی ترکیب ?huns با ones/tens/huns استفاده شده اما با این دید که همه اعداد در نوشتار رسمی در گروه‌های ۳ رقمی دارای ارزش یکان دهگان صدگان هستند.

حال اگر در حین اسکن چپ به راست اعداد دو رشته گروه و کل پاسخ نگه داری کنیم و منطق پرانتز گذاری را در گروه‌ها به کار ببریم مشکل حل می شود. این **راه‌کار به صورت عمومی** جواب می دهد.

برای تست صحت پاسخ رده های mil برای میلیون bil برای میلیارد اضافه کردم و همینطور الگو ها به جای علامت سؤال تبدیل به * یا همان کلوژر می‌شوند تا تعداد گروه‌های سه تایی بیشتر از ۰ یا ۱ هم بتوانند قبول کنند. در اینجا در الگوی * huns huns وارد گروه جدید می‌شویم و در * tens huns گروه را ادامه می دهیم. در الگوی * ones huns علاوه بر افزودن عدد به گروه. کل مقدار محاسبه گروه را با جایگاه مکانی در پرانتز می‌گذاریم و به پاسخ نهایی اضافه می کنیم.

دیگر نکات:

۱– الگو های قابل تعریف در flex تو در تو نیستند و فقط با yyless توانستم بین آن‌ها ارتباط برقرار کنم.

۲– تمامی اسکنرهای من کاراکتر های غیر عددی و علایم مورد استفاده را در نظر نمی‌گیرند یعنی از دید پارسر:

user_inp: money 321,123+79 is ok

→ lex_out: <NUM> <NUM> <PLUS> <NUM>

قسمت‌های اجرایی:

فایل build.sh:

به صورت خودکار از yacc و flex خروجی می‌گیرد همه را با هم در gcc کامپایل می‌کند و مسیر فایل خروجی را چاپ می‌کند

به عنوان ورودی می‌شود lexV را از قبل ست کنیم تا از ورژن های ۱ ۲ یا ۳ اسکنر نوشته شده استفاده کند به طور بیشفرض این مقدار ۳ است.

فایل run.sh:

این اسکریپت بس از درخواست کامپایل با اسکریپت قبلی . فایل اجرایی را صدا میزند تا کاربر بتواند وروی دلخواه را چاپ کند. به دلیل ماهیت نوع گرامر و زبان مسیله انتهای ورودی به هیچ عنوان به صورت هوشمند قابل حدس نیست و تنها معیار اتمام ورودی eof است. جهت وارد کردن eof میتوان از ctrl+d یا ctrl+z بسته به سیستم عامل و ترمینال مورد استفاده . استفاده کرد.

فایل test-numbers.sh:

این اسکریپت بس از کامپایل کد. اعداد ۱ – ۲۱ – ۳۲۱ – ... – ۶۵۴۳۲۱ – ... – ۳۲۱۹۸۷۶۵۴۳۲۱ را به عنوان ورودی ها جداگانه به برنامه می‌دهد تا خروجی اسکنر آن‌ها را تست کنیم. جهت مقایسه ورژن های اسکنر میتوان:

lexV=1|2|3 ./test-numbers.sh

استفاده کرد و صحت انتخاب منطق ۳ و بیشرفت اسکنر را در ورزن های مختلف دید.

فایل test-case.sh:

این اسکریپت بس از کامپایل کد یک سری تست کیس با محوریت تست پارسر اجرا میکند . دو تست کیس اولی مثال‌های آورده شده در متن پروژه استفاده شده است.

با توجه به بیشفرض پروژه که ورودی کاربر صحیح است یک سری تست های غیر صحیح مثل year 2021 جهت نمایش رفتار اسکنر و پارسر در کنار هم آورده شده.

توابع کمکی:

با توجه به مشکلات کار با malloc و رشته در زبان c. در این برنامه تمامی رشته‌ها به صورت یک عدد در یک لیست سراسری معرفی می شوند. رشته‌های zer تا nin معادل اعداد ۰ تا ۹ و اعداد ۱۰ تا ۱۸ معادل جایگاه های عددی ten hun tho mil bil و علایم plu min mul div هستند .اعداد منفی هم رشته خالی.

بقیه رشته‌ها با عملیات های روی این رشته‌ها ساخته می شوند. و عدد جدید میگیرند . مثلا:

concat(0, 11) → 37: ZerHun mix(1,2,3) → 45: One Two Thr

join(5,10,7) → 101: FivTen_Sev wrap(8)→ 51: (Eig)

print(101) → 97: Print FivTen_Sev eval(37)→ 54: Assign ZerHun to t4

eval(concat(wrap(mix(1,15,51)),10)) → 29: Assign (One Plu (Eig))Ten to t19