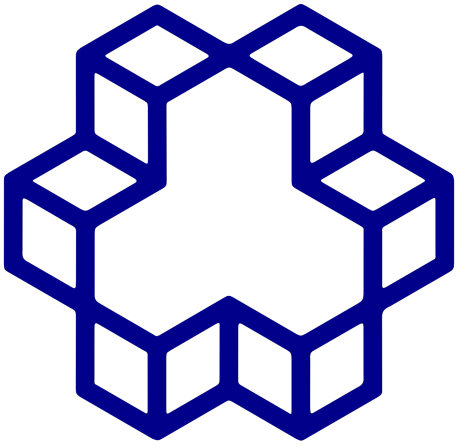
به نام خدا



دانشکده ریاضی
دانشگاه صنعتی خواجه نصیر الدین طوسی

شرح فاز سوم پروژه کامپایلر

پروژه درس کامپایلر
استاد دامن افشان

آفرین آخوندی ۴۰۱۱۵۲۸۳
عماد پورحسنی ۴۰۱۱۶۶۲۳

فهرست مطالب

۱ مقدمه

این فایل، مشخصات پیاده سازی و جزئیات فاز سوم پروژه درس کامپایلر را شرح میدهد. تحلیل گر های لغوی و نحوی و طرز تولید کد سه آدرسه که در ادامه به شرح پیاده سازی و ساز و کار آن پرداخته شده است، توسط عماد پورحسنی و آفرین آخوندی تهیه شده است.
تحلیل گر لغوی به کمک سازنده تحلیل گر لغوی flex آماده شده است.
تحلیل گر نحوی به کمک سازنده تحلیل گر نحوی bison آماده شده است.
تولید کد سه آدرسه توسط توابع و ساختار های کمکی که در symtab.c و symtab.h یا درواقع همان جدول نماد ها تعریف شده اند، در فایل translate.y به کمک bison انجام شده است.
اطلاعات بیشتر را میتوانید در مخزن گیتهاب پروژه بررسی کنید.

۲ پیاده سازی

در این بخش به ریز جزئیات پیاده سازی می پردازیم. ساختار دایرکتوری پروژه به صورت زیر است :

```
.
├── symboltable
│   ├── symtab.o
│   ├── symtab.h
│   ├── symtab.c
│   ├── README.md
│   └── changelog.md
├── specification
│   ├── token_list.txt
│   ├── specification.ebnf
│   ├── specification.bnf
│   ├── README.md
│   └── diagram
├── README.pdf
├── README.md
├── projectSpecification.pdf
├── parser
│   ├── translate.y
│   ├── translate.xml
│   ├── translate.tab.o
│   ├── translate.tab.h
│   ├── translate.tab.c
│   ├── testcases
│   ├── run_tests.sh
│   ├── parser-output.txt
│   ├── parser
│   ├── output.txt
│   ├── Makefile
│   ├── input.txt
│   ├── configure
│   ├── automaton.html
│   └── automaton.dot
├── LICENSE
└── lexer
    ├── TestCases
    ├── README.md
    ├── Makefile
    ├── lex.yy.o
    ├── lex.yy.c
    └── lex.l
```

۱.۲ تولید کد سه آدرسه

در هر مرحله پارسر ما توکن ها را از لکسر میگیرد و تصمیم میگیرد که باید شیفت انجام دهد یا reduce (تقلیل)
هر بار که یک تقلیل اتفاق می افتد، اطلاعات مربوط به token های تقلیل پیدا کرده به یک head پروداکشن در جدول نماد ها به صورت یک Quadruple ذخیره میشود. ساختار یک Quadruple مانند زیر است :

$$\text{Quadruple} \iff (\text{op, arg1, arg2, result})$$

نحوه ذخیره سازی این Quadruple ها در جدول نماد ها به این صورت است که struct ای به نام QuadrupleNode آن ها را به فرم یک لیست پیوندی دوطرفه نگه میدارد تا در ادامه بتوانیم آن ها را به صورت reverse یا عادی پیمایش کنیم.
چند تا حالت برای Quadruple ها پیش می آید :

- Assignment ها:
  - Binary Assignment مانند

$$a := 2 + 3 \iff (:=, 2, 3, t_1)\ \&\ (:=, t_1, \_, a)$$

  - Copy Assignment مانند

$$a := 2 \iff (:=, 2, \_, a)$$

- Unconditional Jump مانند

$$\text{GOTO } L_1 \iff (\text{GOTO}, \_, \_, L_1)$$

- Conditional Jump مانند

$$\text{IF x GOTO } L_1 \iff (\text{IF, x}, \_, L_1)$$

برای مثال یک شرطی ساده به چهار Quadruple نیاز دارد

$$\text{if x then x := 3;} \iff (\text{IF, x}, \_, L_0)\ \ \&\ \ (\text{LABEL}, \_, \_, L_0)$$
$$\&\ \ (\text{LABEL}, \_, \_, L_1)\ \ \&\ (\text{GOTO}, \_, \_, L_1)$$

که در واقع ترکیب conditional_jump و unconditional_jump ها است. برای حلقه ها نیز ترکیبی از این Quadruple ها استفاده میشوند.
در لحظه تقلیل پیدا کردن قوانین میشود این quadruple ها را ساخت و به ترتیب آن ها را به جدول نماد ها اضافه کرد. سپس در انتهای کار، آن ها را به کمک تابع print_TAC که در symtab.c تعریف شده است پیمایش و چاپ میکنیم.
در این پیاده سازی قابلیت تولید کد های سه آدرسه برای

- if-else statements
- While loops
- For loops
- Assignments

وجود دارد.
که در ادامه به شرح جزئیات میپردازیم.

۱.۱.۲ ****نکته مهم****

در کد سه آدرسه نهایی، موقعیت label ها و پرش های متناظر به آن ها صحیح است اما امکان دارد که عدد لیبل ها به طور منظم جلو نرود، دلیلش این است که به خاطر مبهم بودن قانون مربوط به If-Else ناچار هستیم که سه لیبل ،else_label true_label و next_label را بسازیم فارغ از این که Else داریم یا نه. بدون تغییر دادن گرامر و حل مشکل dangling-else پارسر ساز های مبتنی بر LALR مانند bison نمیتوانند embedded_action مناسب برای این قانون انجام دهند، زیرا مبهم بودن این گرامر اجازه ریدوس شدن درست را نمیدهد، حتی اگر برای action اولویت قائل شویم. (سکشن 3.4.8 در داکیومنتیشن bison را ببینید.)[۱]

۲.۲ کد سه آدرسه برای Assignment ها

برای Assignment ها در برنامه ورودی کد سه آدرسه به نحوه زیر تولید میشود :

$$a := 2 + c \times 12$$

Quadruple ها به ترتیب زیر ایجاد می‌شوند :

کدهای سه آدرسی	Quadruples	توضیحات
$t1 := c \times 12$	$(\times, c, 12, t1)$	انتساب باینری
$t2 := 12 + t1$	$(+, 12, t1, t2)$	انتساب باینری
$a := t2$	$(=, t2, -, a)$	انتساب کپی

۱.۲.۲ لیست پیوندی Quadruple ها

لیست پیوندی Quadruple ها به صورت زیر است:

```
NULL <- (*, c, 12, t1) <-> (+, 12, t1, t2) <-> (:=, t2, -, a) -> NULL
```

۲.۲.۲ پیاده‌سازی

در پیاده‌سازی ما، مراحل زیر انجام می‌شود:

- شناسایی و ایجاد Quadruple ها
- قراردادن آنها در جدول نماد (Symbol Table)
- دسترسی به نتیجه Quadruple ها با $q.result$
- پیوند زدن Quadruple ها در هر مرحله
- استفاده از لیست پیوندی
- چاپ کدهای سه آدرسی مرتبط

```
t1 := c * 12
t2 := 12 + t1      <-> a := 2 + c * 12
a  := t2
```

۳.۲ کد سه آدرسه برای If Else Statement ها

۱.۳.۲ تحلیل دستورات شرطی

در تولید کد سه آدرسی، دستورات شرطی به روش‌های مختلفی ترجمه می‌شوند:

۲.۳.۲ ساختار ساده If

- شرط در یک متغیر موقت محاسبه می‌شود
- از دستورات جهش (GOTO) برای کنترل جریان استفاده می‌شود
- برچسب‌های مختلف (L0, L1) برای مسیریابی منطقی استفاده می‌شوند

۳.۳.۲ ساختار If-Else

- شامل دو مسیر اجرایی متفاوت است
- شرط اصلی با یک متغیر موقت ارزیابی می‌شود
- دستورات GOTO برای انتقال به بلوک‌های متناظر استفاده می‌شوند

۴.۳.۲ استفاده از BackPatch

برای تولید لیبل های متناظر برای عبارات شرطی که Else دارند باید از BackPatching استفاده کرد. به این طور که در ابتدا آن ها را UNFILLED قرار میدهیم

```
Quadruple gotoquad = create_quadruple(symboltable, "GOTO", "UNFILLED ELSE", "", "");
add_quadruple(symboltable, gotoquad);
//printf("goto  ---- ← NEXT LABEL FOR IF-ELSE STMT\n");

Quadruple elabelquad = create_quadruple(symboltable, "LABEL","UNFILLED NEXT", "", "");
add_quadruple(symboltable, elabelquad);
//printf("LABEL ---- ← ELSE LABEL FOR IF-ELSE STMT \n");
```

شکل ۱: Corresponding Code

و سپس بعد از reduce شدن کامل شرط

```
stmt -> IF_KW expr THEN_KW stmt ELSE_KW stmt
```

آن ها را backpatch میکنیم.

```
if (strcmp($5,"hasElse")==0){
    Quadruple labelquad = create_quadruple(symboltable, "LABEL",$<label_pair>3.label2, "", "");
    add_quadruple(symboltable, labelquad);
    //printf("%s : \n", NEXT_label);

    // BACKPATCH THE IF-ELSE LABELS!
    // Using the  backpatch function from symtab.c here.

    BackPatchLabels(symboltable, $<label_pair>3.label2, $<label_pair>3.label1);
}
```

شکل ۲: Corresponding Code

۵.۳.۲ مثال کامل

```
if x > 2 then
    x := true;
else
    x := false;
```

معادل کد سه آدرسی:

```
    t0 := x > 2
    IF t0 GOTO L3
    GOTO L4
L3:
    x := true
    GOTO L5
L4:
    x := false
L5:
```

۶.۳.۲ الگوریتم تولید

- محاسبه شرط در یک متغیر موقت
- تولید دستور شرطی IF
- ایجاد label مسیر
- تولید دستورات داخل بلوک‌های شرطی
- مدیریت جریان اجرا با GOTO

۴.۲ تولید کد سه آدرسه برای حلقه‌ها

۱.۴.۲ ساختار حلقه While

- محاسبه شرط در یک متغیر موقت
- تولید برچسب‌های ورودی و خروج از حلقه
- مدیریت جریان با دستورات IF و GOTO

۲.۴.۲ ساختار حلقه For

- مقداردهی اولیه متغیر کنترلی
- محاسبه شرط پایان حلقه
- تولید برچسب برای تکرار و خروج
- افزایش متغیر کنترلی در هر تکرار

۳.۴.۲ مثال عملیاتی

```
while i < 20 do
    for j := 0 to 12 do
        y := inFor;
```

معادل کد سه آدرسی:

```
t0 := i < 20
L0:
    IF t0 GOTO L1
    GOTO L2
L1:
    j := 0
    t1 := j <= 12
L3:
    IF t1 GOTO L4
    GOTO L5
L4:
    y := inFor
    j := j + 1
    GOTO L3
L5:
    GOTO L0
L2:
```

۴.۴.۲ استراتژی تولید

- تولید متغیرهای موقت برای شروط
- مدیریت label های جهش
- flow control با دستورات شرطی
- پیگیری متغیرهای حلقه

۵.۲ جدول نماد ها

جدول نماد ها به کمک hashtable پیاده سازی شده است و شرح آن در فایل symtab.c قابل مشاهده است. این هش تیبل به صورت خطی کار کرده و اعداد صحیح و حقیقی را در مرحله تحلیل لغوی ذخیره میکند.
در مرحله تحلیل نحوی ، شناسه های در زمان declaration شناسایی شده و با scope و type و kind مشخص در جدول نماد ها قرار میگیرند. همچنین Quadruple ها در جدول نماد ها قرار میگیرند و ساختار لیست پیوندی دارند. در نهایت Quadruple ها نیز در خروجی قرار خواهند گرفت.
جدول نماد نهایی توسط parser در فایل output.txt قرار میگیرد.

۳ ساختن و اجرا کردن

** در کل برای ساخت و اجرای برنامه ابتدا اسکریپت configure را اجرا کنید ، سپس make را اجرا کنید و نهایتا فایل parser آماده برای اجرا است.
در نظر داشته باشید که تمام این مراحل باید در دایرکتوری parser باشید.
خروجی کد سه آدرسه در فایل output.txt خواهد بود .**

برای ساخت و اجرای پروژه‌ی پارسر، یک سیستم ساخت مبتنی بر configure و Makefile پیاده‌سازی شده است. این سیستم امکان ساخت خودکار و مدیریت وابستگی‌های پروژه را فراهم می‌کند.

۱.۳ configure

اسکریپت configure وظیفه‌ی بررسی پیش‌نیازها و آماده‌سازی محیط برای ساخت پروژه را بر عهده دارد. این اسکریپت شامل بخش‌های زیر است:

```
#!/bin/bash
echo "Checking for required programs..."

# Check for GCC
if ! command -v gcc >/dev/null 2>&1; then
    echo "Error: gcc compiler not found"
    exit 1
fi
```

این اسکریپت به طور سیستماتیک موارد زیر را بررسی می‌کند:

- وجود کامپایلر gcc
- نصب بودن ابزار bison برای تولید پارسر
- نصب بودن ابزار flex برای تولید لکسر

- وجود هدرفایل‌های ضروری سیستمی

در صورت عدم وجود هر یک از موارد فوق، اسکریپت با پیام خطای مناسب خاتمه می‌یابد. برای اجرای اسکریپت configure، دستور زیر استفاده می‌شود:

```
./configure
```

۲.۳ ساختار Makefile

Makefile طراحی شده برای این پروژه شامل چندین بخش اصلی است که در ادامه تشریح می‌شوند:

۱.۲.۳ تعریف متغیرهای اصلی

متغیرهای پایه‌ای برای تنظیم ابزارها و پرچم‌های کامپایل به صورت زیر تعریف شده‌اند:

```
CC = gcc
YACC = bison
LEX = flex
CFLAGS = -Wall -g
LDFLAGS = -lfl
```

این متغیرها امکان پیکربندی مجدد ابزارها و گزینه‌های کامپایل را بدون نیاز به تغییر در سایر بخش‌های Makefile فراهم می‌کنند.

۲.۲.۳ مدیریت پروژه

برای مدیریت فایل‌های تولید شده، دستور clean به صورت زیر تعریف شده است:

```
clean:
    rm -f $(TARGET) $(OBJECTS) translate.tab.c translate.tab.h
```

این دستور امکان حذف تمامی فایل‌های تولید شده و شروع مجدد فرآیند ساخت را فراهم می‌کند.
دستورات اصلی برای استفاده از Makefile عبارتند از:

- ساخت پروژه: make
- پاک‌سازی فایل‌های تولید شده: make clean

سیستم ساخت طراحی شده از ویژگی تشخیص وابستگی‌ها بهره می‌برد و تنها فایل‌هایی را مجدداً می‌سازد که تغییر کرده‌اند. این ویژگی باعث بهینه‌سازی زمان ساخت و تسهیل فرآیند توسعه می‌شود.

۳.۳ نمونه ورودی و خروجی

نمونه ورودی برنامه :

```
program prg1;

integer num, divisor,quotient;
begin
 num := 61;
 divisor := 2;
 quotient := 0;
 if num=1 then
  return false;
 else if num=2 then
  return true;
 while divisor<(num/2) do
 begin
  quotient:=num/divisor;
```

```
  if divisor * quotient=num then
   return false;
  divisor:=divisor+1;
 end
 return true;
end
```

نمونه خروجی برنامه :

```
Emad Pourhassani := 40116623
Afarin Akhoundi  := 40115283
----------------------------

Symbol Table Contents:
--------------------------------------------------
Name                  Kind       Type       Scope
--------------------------------------------------
t5                    TEMPORARY  UNKNOWN    0
t6                    TEMPORARY  UNKNOWN    0
2                     CONSTANT   INT        0
t2                    TEMPORARY  UNKNOWN    0
t7                    TEMPORARY  UNKNOWN    0
t4                    TEMPORARY  UNKNOWN    0
prg1                  FUNCTION   UNKNOWN    1
t0                    TEMPORARY  UNKNOWN    0
num                   VARIABLE   INT        1
t3                    TEMPORARY  UNKNOWN    0
quotient              VARIABLE   INT        1
divisor               VARIABLE   INT        1
61                    CONSTANT   INT        0
0                     CONSTANT   INT        0
t1                    TEMPORARY  UNKNOWN    0
1                     CONSTANT   INT        0
--------------------------------------------------

Quadruples List:
--------------------------------------------------
No.   Op          Arg1            Arg2            Result
--------------------------------------------------
1     :=          61                              num
2     :=          2                               divisor
3     :=          0                               quotient
4     =           num             1               t0
5     IF          t0                              L0
6     GOTO        L1
7     LABEL       L0
8     RETURN      false
9     GOTO        L2
10    LABEL       L1
11    =           num             2               t1
12    IF          t1                              L3
13    GOTO        L4
14    LABEL       L3
15    RETURN      true
16    LABEL       L4
```

```
17    LABEL     L2
18    /         num           2             t2
19    <=        divisor       t2            t3
20    LABEL     L6
21    IF        t3                          L7
22    GOTO      L8
23    LABEL     L7
24    /         num           divisor       t4
25    :=        t4                          quotient
26    *         divisor       quotient      t5
27    =         t5            num           t6
28    IF        t6                          L9
29    GOTO      L10
30    LABEL     L9
31    RETURN    false
32    LABEL     L10
33    +         divisor       1             t7
34    :=        t7                          divisor
35    GOTO      L6
36    LABEL     L8
37    RETURN    true
--------------------------------------------------

Three Address Codes:
----------------------------
  num := 61
  divisor := 2
  quotient := 0
 t0 := num = 1
  IF t0 GOTO L0
  GOTO L1
L0:
  RETURN false
  GOTO L2
L1:
 t1 := num = 2
  IF t1 GOTO L3
  GOTO L4
L3:
  RETURN true
L4:
L2:
 t2 := num / 2
 t3 := divisor <= t2
L6:
  IF t3 GOTO L7
  GOTO L8
L7:
 t4 := num / divisor
  quotient := t4
 t5 := divisor * quotient
 t6 := t5 = num
  IF t6 GOTO L9
  GOTO L10
L9:
  RETURN false
```

```
L10:
 t7 := divisor + 1
  divisor := t7
  GOTO L6
L8:
  RETURN true
```

۴ مراجع

[1] Charles Donnelly and Richard Stallman. **Bison Manual**. Free Software Foundation 2021.