قبل از پیاده سازی پروژه به نکات زیر توجه داشته باشید:

- هدف از انجام پروژه‌ها، یادگیری عمیق‌تر مطالب درسی است و پروژه‌ها به صورت انفرادی انجام می‌شوند. در نتیجه هرگونه تقلب موجب کسر نمره خواهد شد.
- مهلت تحویل پروژه روز ۱۵ تیر ساعت ۸ صبح و نحوه تحویل از طریق سامانه کورسز است.
- پروژه تحویل انلاین دارد و در صورت حاضر نشدن در ارائه آنلاین نمره پروژه ۰ تلقی می‌شود.
- در صورت وجود سوال می توانید از طریق کانال تلگرام درس با تدریسیاران در ارتباط باشید.

گرامر

مقدمه‌ای برANTLR

ANTLR، که مخفف «ANother Tool for Language Recognition» است، یک ابزار قدرتمند برای تولید تجزیه‌گر «Parser» است. با استفاده از یک دستورالعمل گرامری، ANTLR یک تجزیه‌گر را تولید می‌کند که می‌تواند درخت تجزیه را بسازد و آن را پیمایش کند. در واقع، ANTLR یک ابزار است که به شما اجازه می‌دهد تا قوانینی را تعریف کنید که چگونه ورودی‌ها باید تجزیه شوند. بعد از آن، این قوانین را به یک زبان برنامه‌نویسی خاص ترجمه می‌کند تا بتوانید از آن در برنامه‌های خود استفاده کنید.

در مرحله اول پروژه، دانشجویان باید گرامر یک زبان را که در مستندات توضیح داده شده است بنویسند. پس از نوشتن گرامر، با استفاده از تجزیه‌گر درختی ANTLR، باید درخت تجزیه را برای موارد تست داده شده رسم کنند. در قسمت های بعدی، دانشجویان باید یک کامپایلر برای این زبان با استفاده از ANTLR بنویسند تا کد سه آدرسی را برای فایل ورودی تولید کنند.

نحوه کار کردن با ANTLR

ما در اینجا قصد داریم گام به گام نحوه کار با ANTLR را به شما آموزش دهیم.

زبان :

زبانی که قصد داریم تجزیه کنیم، یک زبان ریاضی ساده شامل عملیات جمع و تفریق است. این زبان از اعداد صحیح و پرانتزها برای تعیین اولویت عملیات پشتیبانی می‌کند.

$$< \boldsymbol{Expression} > \quad ::== \quad < \boldsymbol{Expression} > + < \boldsymbol{Term} >$$

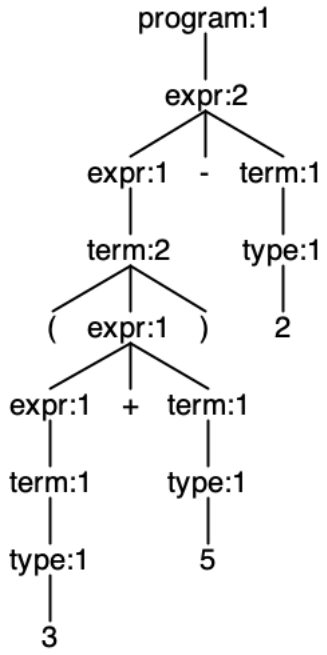
$$
\begin{array}{lll}
 & | & < Expression > - < Term > \\
 & | & < Term > ; \\
 & & \\
< Term > & ::== & < Type > \\
 & | & (< Expression >) ; \\
 & & \\
< Type > & ::== & int
\end{array}
$$

گرامر:

```
grammar ArithGrammer;

// Starting rule
program: expr ;

expr    : expr '+' term
        | expr '-' term
        | term
        ;

term    : type
        | '(' expr ')'
        ;

type    : INT ;

INT     : [0-9]+ ;

WS      : [ \t\r\n]+ -> skip ;
```

این گرامر به شما این اجازه را می‌دهد تا ورودی‌هایی که عملیات جمع/تفریق انجام می‌دهند را تجزیه کنید. با کمک دستور زیر خروجی درخت پارس شده را رسم می‌کنیم.

antlr4-parse ArithGrammer.g4 program -tree test/test003.txt -gui



در نهایت با کمک دستور زیر فایل‌هایی برایمان ساخته می‌شود که حاوی کد Lexer ، Parser و Listener می‌باشد. در قسمت بعدی به این فایل‌ها پرداخته می‌شود.

```
antlr4 -o src ArithGrammer.g4
```

پیاده سازی

در این پروژه باید به کمک ابزار ANTLR که در بخش قبل یاد گرفتید، گرامر زبانی که در ادامه شرح داده می‌شود را بنویسید تا در قسمت بعدی، کامپایلری جهت تولید کد سه آدرسه در زبان C پیاده سازی کنید. گرامر زبان پروژه در ادامه آورده شده است. توجه کنید این گرامر را باید در فرمت antlr4 پیاده‌سازی کنید و اطمینان حاصل کنید که به درستی فایل‌های تست را تجزیه می‌کند. به عنوان مثال باید ترتیب عملیات ریاضی حفظ شود. همچنین ابهاماتی در گرامر وجود دارد که باید رفع شوند.

$$
\begin{array}{lll}
< prog > & ::= & < func - list > \\
\\
< func - list > & ::= & < func - def >< func - list > \\
 & & | < func - def > \\
\\
< func - def > & ::= & < data - type >< id > (< param - list >) < code - block > \\
\\
< param - list > & ::= & < param >, < param - list > \\
 & & | < param > \\
 & & | \epsilon \\
\\
< param > & ::= & < data - type >< id > \\
\\
< data - type > & ::= & \text{int} \mid \text{double} \mid \text{boolean} \mid \text{void} \\
\\
< code - block > & ::= & \{< stmt - list >\} \\
\\
< stmt - list > & ::= & < stmt >< stmt - list > \\
 & & | \epsilon \\
\\
< stmt > & ::= & ; \\
 & & | < code - block > \\
 & & | < data - type >< var - list >; \\
 & & | < id >=< expr >; \\
 & & | < id > ++; \\
 & & | < id > --; \\
 & & | \text{return } ; \\
 & & | \text{return } < expr >; \\
 & & | \text{decide } (< expr >) < stmt > \\
 & & | \text{decide } (< expr >) < stmt > else < stmt > \\
 & & | < expr >; \\
\\
< loop - stmt > & ::= & \text{loop } (< expr >) < stmt > \\
\\
< init - stmt > & ::= & < data - type >< id >=< expr > \\
 & & | < id >=< expr > \\
 & & | \epsilon \\
\\
< post - stmt > & ::= & < id >=< expr > \\
 & & | < id > ++ \\
 & & | < id > -- \\
 & & | \epsilon
\end{array}
$$

$$
\begin{aligned}
< var-list > \quad &::= \quad < var-list >< var > \\
&\quad\;\; | < var > \\
\\
< var > \quad &::= \quad < id > \\
&\quad\;\; | < id >=< expr > \\
\\
< expr > \quad &::= \quad < number > \\
&\quad\;\; | < id > \\
&\quad\;\; | \text{true} \\
&\quad\;\; | \text{false} \\
&\quad\;\; | < string-lit > \\
&\quad\;\; | < id > (< args >) \\
&\quad\;\; | (< expr >) \\
&\quad\;\; | < unop >< expr > \\
&\quad\;\; | < expr >< binop >< expr > \\
\\
< args > \quad &::= \quad < expr >, < args > \\
&\quad\;\; | < expr > \\
\\
< unop > \quad &::= \quad \text{-} \mid \text{!} \\
\\
< binop > \quad &::= \quad + \mid \text{-} \mid * \mid / \mid == \mid \text{!=} \mid < \mid \leq \mid > \mid \geq \mid \text{and} \mid \text{or} \\
\\
< number > \quad &::= \quad < integer > \mid < double >
\end{aligned}
$$

تحلیل‌گر لغوی

شما باید توکن‌های زبان را با توجه به گرامر مشخص کنید. هر رشته‌ای که بین دو جهت‌نما نوشته نشده است یک توکن است. همچنین موارد زیر نیز توکن‌های زبان هستند:

- اعداد صحیح: اعداد صحیح با صفر شروع نمی‌شوند.
- اعداد ممیز شناور: به صورت integer.integer تعریف می‌شوند.
- شناسه‌ها: با ارقام شروع نمی‌شوند و شمال حروف کوچک و بزرگ انگلیسی، ارقام و کاراکترهای خاص مانند – و _ هستند. همچنین کلیدواژه‌ها نمی‌توانند شناسه باشند.
- کلیدواژه‌ها: شامل ... ,if, else, while, for هستند.
- رشته‌ها: دنباله‌ای از مجموعه کاراکترهای پایه‌ای به جز Quotation mark و Reverse solidus و Line feed همچنین رشته‌ها می‌توانند شامل Simple escape sequences نیز باشند. دقت کنید رشته‌ها فقط به تابع اولیه printString ورودی داده می‌شوند. (در فازهای بعدی بیشتر با این تابع آشنا می‌شوید)
- کامنت: در این زبان می‌توان با کمک // یا /* */ کامنت تعریف کرد.

تحلیلگر نحوی

دقت کنید ANTLR یک تجزیه‌گر top-down است. از درس می‌دانیم ملاحظاتی برای گرامری که به این تجزیه‌گرها داده می‌شود وجود دارد. با توجه به قابلیت‌های ANTLR آنها را در صورت لزوم اعمال کنید. همچنین چند ابهام در گرامر وجود دارد. ابهام اول درباره

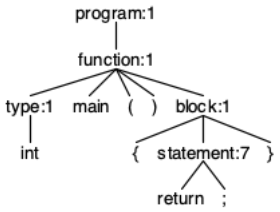
dangling elseاست که به روش nearest if باید رفع شود. ابهام دوم مربوط به اولویت عملگرهاست. از جدول زیر برای اولویت‌دهی و رفع این ابهام استفاده کنید:

Precedence	Operator	Associativity	Description
1	- !	Right-to-left	Unary minus and logical NOT
2	* / %	Left-to-right	Multiplication, division, and remainder
3	+ -	Left-to-right	Addition and subtraction
4	< > <= >=	Non-associative	Relational operators
5	== !=	Non-associative	Equality operators
6	&&	Left-to-right	Logical AND
7	||	Left-to-right	Logical OR

خروجی

در فایل زیپ همراه پروژه چند فایل test قرار داده شده که برای تمامی آنها می‌بایست درخت تجزیه شده را رسم کنید. برای مثال یک برنامه و خروجی مد نظر آن در ادامه قابل مشاهده می‌باشند.

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



بعد از بدست آوردن خروجی‌های مدنظر، با کمک دستور آموزش داده شده در بخش‌های قبلی کد مربوط به Lexer, Parser, Listener را تولید کنید و با مطالعه کد Lexer, Parser بخش‌های متفاوت را توضیح دهید.

گزارش

برای گزارش فاز اول پروژه، فایلی خلاصه و کوتاه تهیه کنید که شامل اطلاعات ذیل باشد.

۱. بخش اولویت عملگرها و روش استفاده شده برای رفع dangling else را توضیح دهید.
۲. خروجی دو درخت تجزیه شده را به دلخواه انتخاب کنید و به صورت خلاصه توضیح دهید.
۳. فایل‌های تولید شده توسط antlr را به صورت خلاصه توضیح دهید.

تولید کد

در قسمت قبلی پروژه با استفاده از ابزار Antlr برای گرامری که در تعریف پروژه آمده بود lexer و parser طراحی کردیم. حال در این فاز می‌خواهیم به تحلیل معنایی کد و تولید کد سه آدرسه بپردازیم. در ادامه به توضیح تحلیل معنایی و مجموعه دستورات قابل استفاده در کد سه‌آدرسه‌ی خروجی می‌پردازیم. سپس نحوه‌ی پیاده‌سازی این موارد و در نهایت بخش‌های امتیازی پروژه را شرح می‌دهیم.

شیوه ی پیاده سازی

در فاز قبل مشاهده کردیم که antlr با گرفتن grammar می‌تواند فایل‌های مربوط به فرایند compile را تولید کند. یکی از این فایل‌ها Listener است. وظیفه‌ی این فایل و توابع داخلی آن، این است که در حین پیمایش درخت parse عملیات لازم برای هر گره از درخت را حین ورود و خروج از آن گره پیاده‌سازی کند. در این بخش از پروژه می‌خواهیم با اعمال تغییرات و تکمیل این فایل listener کد سه آدرسه تولید کنیم و هم‌چنین در صورت رخداد خطای معنایی، این خطا را نمایش دهیم. در این پروژه هدف این است که گرامر پارس شده را تبدیل به کد سه‌آدرسه‌ای بکنیم که در زبان برنامه نویسی C قابل اجرا باشد. در ادامه یک Template به شما داده می‌شود که شما تنها لازم دارید بدنه تابع main را تکمیل کنید و کد را اجرا کنید. اگر کد سه آدرسه را به درستی تولید کرده باشید باید بدون خطا اجرا شود و خروجی مورد نظر را تولید کند. هدف ما نوشتن یک کامپایلر از صفر نیست و قصد داریم تولید کد سه آدرسه و پارس شدن یک گرامر را یاد بگیریم.

همانطور که می‌توان مشاهده کرد، فایل Listener تولید شده توسط Antlr برای هر non-terminal گرامر دو تابع enter و exit دارد. شما با تکمیل کردن این توابع می‌توانید مشخص کنید که حین پیمایش درخت parse چه عملیاتی در هر گره باید صورت گیرد. مثلا با توجه به روش‌های ارائه شده برای تولید کد سه‌آدرسه در کلاس می‌توانید تنها با تکمیل کردن توابع exit کد سه آدرسه‌ی مربوط به یک گره را تولید کنید. کدهای نوشته شده حین پیمایش (walk) کردن روی درخت (توسط antlr) اجرا می‌شوند و عملیات لازم حین ورود و خروج گره‌ها به صورت خودکار انجام می‌شود. (برای مشاهده‌ی توالی اجرای ماژول‌های compiler تولید شده می‌توانید فایل Parser نوشته شده توسط antlr را مشاهده کنید. اگر کد antlr را با زبان پایتون تولید کرده باشید نام این فایل Parser.py است.)

به عنوان مثال برای قطعه گرامر زیر در فایل GrammarListener.py تغییرات زیر اعمال شده است:

⟨statements⟩::= ⟨statement⟩ ⟨statements⟩ | ⟨empty⟩

⟨statement⟩::= ;

| return ;

| return ⟨expression⟩ ;

| if (⟨expression⟩) ⟨statement⟩

| if (⟨expression⟩) ⟨statement⟩ else ⟨statement⟩

| ⟨expression⟩ ;

```
# Enter a parse tree produced by GrammarParser#statements.
    def enterStatements(self, ctx:GrammarParser.StatementsContext):
```

```
        pass

    # Exit a parse tree produced by GrammarParser#statements.
    def exitStatements(self, ctx:GrammarParser.StatementsContext):
        if len(ctx.statement()) == 1:
            # Single statement
            statement = ctx.statement(0)
            ctx.code = statement.code
        else:
            # Multiple statements
            statements_code = [statement.code for statement in ctx.statement()]
            ctx.code = "\n".join(statements_code)


    # Enter a parse tree produced by GrammarParser#statement.
    def enterStatement(self, ctx:GrammarParser.StatementContext):
        pass

    # Exit a parse tree produced by GrammarParser#statement.
    def exitStatement(self, ctx: GrammarParser.StatementContext):
        if ctx.SEMICOLON():
            # Empty statement
            ctx.code = ""

        elif ctx.getChild(0).getText() == "return":
            # Return statement
            if ctx.expression():
                # Return with expression
                expression_code = ctx.expression().code
                j = self.j()
                k = self.j()
                ctx.code = f"m[top+{j}].{ctx.expression().type} =
m[top+{k}].{ctx.expression().type};\n"
                ctx.code += "top = m[top+{0}].i;\n".format(j)
            else:
                # Return without expression
                ctx.code = "top = m[top].i;\n"
            ctx.code += "return;\n"
        elif ctx.getChild(0).getText() == "if":
            # If statement
            condition_code = ctx.expression().code
            if_statement = ctx.statement(0).code

            if ctx.getChild(5).getText() == "else":
                else_statement = ctx.statement(1)
                ctx.code = f"{ctx.expression().code} if ({condition_code}) goto
label_{self.new_label()}: {{\n{if_statement.code}}} else goto
```

```
label_{self.new_label()}: {{\n{else_statement.code}}}\n"
            else:
                ctx.code = f"{ctx.expression().code} if ({condition_code}) goto
label_{self.new_label()}: {{\n{if_statement.code}}}\n"

        else:
            # Expression statement
            expression_code = ctx.expression().code
            ctx.code = f"{expression_code};\n"
```

ممکن است فایل نوشته شده برایتان مفهومی نداشته باشد. أصلا m[top] چی هست؟ در بخش تولید کد همونطور که قبلا گفتم یک template برای شما آماده شده است. در این template برای اینکه یک کامپیوتر را شبیه سازی بکنیم ارائه‌ای تعریف کردیم به اسم m که فرض می‌کنیم حافظه اصلی (RAM) کامپیوتر در حین پیاده سازی می‌باشد. برای تولید کد سه آدرسه بعضی وقت‌ها نیاز دارید متغیرهایی را تعریف کنید و یا از Stack و Heap استفاده کنید که همه این‌ها با استفاده از آرایه m انجام می‌شود. این آرایه طبق تعریفات ما در بخش **تحلیلگر نحوی**، فقط می‌تواند int, double و یا void باشد (void در اینجا برای برگرداندن void از توابع استفاده می‌شود). در ادامه مثال‌هایی برایتان اورده‌ایم که نمونه‌هایی از پیاده سازی توابع exit می‌باشد.

به طور خلاصه و ساده بخواهیم توضیح دهیم شما باید کد سه آدرسه هر دستور را به نحوی در زبان C تولید کنید که بدون خطا اجرا شود. مثلا اگر قرار است دو متغیر را جمع کنید، ابتدا باید هر کدام را تعریف کنید، مقدار دهی کنید، جمع کنید و حاصل را ذخیره کنید. همه این تعریف کردن، مقدار دهی و ذخیره کردن نتیجه با آرایه m انجام می‌شود. توجه شود که این صرفا یک روش پیاده سازی است و ممکن است روش های خلاقانه ی دیگری نیز موجود باشد.

تحلیل معنایی

همانطور که در فاز اول گفته شد، زبان مورد استفاده در این پروژه مشابه زبان C است. در این زبان هر برنامه مجموعه‌ای از توابع دارد که حتما شامل یک تابع به نام main است و اجرای آن از همین تابع شروع می‌شود. اسم توابع نمی‌تواند تکراری باشد. هر تابع می‌تواند تعدادی پارامتر داشته باشد. نوع هر پارامتر و نام آن باید مشخص باشد. پارامترها به صورت pass-by-value به تابع ارسال می‌شوند. یعنی مقدار آنها در رکورد فعالیت تابع فراخوانی شده کپی می‌شود و می‌توان به آن مقداری را assign کرد.

نوع‌های تعریف شده در این زبان شامل boolean ، double ، int و void هستند. هر تابع نیز باید یکی از این نوع‌ها را بازگرداند. تابعی که خروجی آن void است نمی‌تواند دستور return x داشته باشد. از طرفی نوع خروجی یک تابع باید با نوع expression جلوی return مطابق باشد. همچنین در دستورات assignment نوع طرفین باید مطابقت داشته باشد. هر تابع دارای یک بلوک کد اصلی است و هر بلوک کد مجموعه‌ای از دستورات را شامل می‌شود. دستورات قابل تعریف در گرامری که در فاز اول پروژه داده شده بود تعریف شده‌اند. دستورات کنترلی مشابه زبان C رفتار می‌کنند. دستور decide همان دستور if، دستور loop همان while در زبان C است. عباراتی که به عنوان شرط دستورات کنترلی قرار می‌گیرند باید دارای ارزش بولی باشند.

در مورد کار با متغیرها باید گفت هر متغیر باید قبل از استفاده حتما تعریف شده باشد. از طرفی هر بلوک کد یک اسکوپ را تعریف می‌کند و اسکوپ‌های داخلی به متغیرهای اسکوپ‌های خارجی دسترسی دارند. اما برعکس آن ممکن نیست. همچنین بازتعریف در اسکوپ‌های داخلی باعث می‌شود متغیر تعریف شده در اسکوپ خارجی داخل اسکوپ داخلی mask شود. هر متغیر باید فقط یک بار در یک اسکوپ تعریف شود. مقدار پیشفرض برای متغیرهای عددی ۰ و برای boolean معادل false است. برای معناشناسی expressionها می‌توانید از جدول زیر استفاده کنید. دقت کنید که هیچ تبدیل نوعی به صورت صریح یا ضمنی در این زبان امکان‌پذیر نیست.

Operator	Operands Types	Result Type	Description
-	Double (integer)	Double (integer)	Unary minus
!	Boolean	Boolean	Logical NOT
%	Integer	Integer	Remainder
* /	Double (integer)	Double (integer)	Multiplication and division
+ -	Double (integer)	Double (integer)	Addition and subtraction
< > <= >=	Double (integer)	Boolean	Relational operators
== !=	Double, integer and boolean	Boolean	Equality operators
&&	Boolean	Boolean	Logical AND
||	Boolean	Boolean	Logical OR

تولید کد

در بخش تولید کد قرار است تا ورودی‌ای که گرامر پارس می‌کند را به صورت کد سه آدرسه‌ای بنویسم که به زبان C کامپایل می‌شود. برای شبیه‌سازی زبان سطح پایین،دسترسی به حافظه مانند دسترسی به آرایه m است. در نتیجه برای اختصاص حافظه به رکوردهای فعالیت باید از این آرایه استفاده کنید. این آرایه، دو اشاره‌گر متفاوت دارد، یکی به ابتدای آرایه و یکی به انتهای آن. در ادامه‌ی تعریف پروژه به این آرایه به عنوان مموری اشاره می‌کنیم. هر سلول حافظه یک نمونه از cell است. در این سلول می توان ،int double و یا *void ذخیره کرد. تایپ double در برنامه را به عنوان تایپ double در C و همچنین تایپ های int و boolean در برنامه به عنوان int در C ذخیره می‌شوند. از تایپ *void برای ذخیره کردن لیبل ها در آرایهm استفاده می‌شود. (مطالعه بیشتر)

همچنین متغیر top مانند رجیستر اشاره‌گر سر استک عمل می‌کند و باید اندیس ابتدای رکورد فعالیت تابع فعلی را نشان دهد. متغیر bottom نیز به انتهای استک اشاره دارد که محل ذخیره کردن Activation Recordها می‌باشد. همانطور که در درس مطالعه کردید، هر تابع مقادیر مورد نیاز خود را به شکل خاصی ذخیره می‌کند که بعد از اجرای تابع، حافظه تخصیص داده شده رها می‌شود. مقادیر زیر را می‌بایست در activation record ذخیره کنید:

1. **Function Arguments**
2. **Local Variables**
3. **Return Address**
4. **Return Value**
5. **Temporary Values**
6. **Previous Frame Pointer (امتیازی)**

تمامی داده‌های مورد نیاز توابع از جمله ورودی توابع، متغیرهای محلی، آدرس بازگشت از تابع، مقدار return، داده‌های temporary و پوینتر به فریم قبلی باید در activation records ذخیره شوند. هر یک activation record چهار مقدار از حافظه رم را اشغال می‌کند و به شکل زیر است. برای نحوه استفاده و assign کردن این مقادیر statementهای شماره ۱۷ به بعد را مطالعه کنید.

```
int x;
int y;
```

```
double d;
int b; // boolean value, can only be 0 or 1
```

دقت کنید مجاز نیستید هیچ متغیر یا آرایه دیگری تعریف کنید. هنگام تولید کد برای expression ها متغیرهای موقت تولید شده و در حافظه ذخیره می‌شوند تا مقدار کل عبارت محاسبه شود. برنامه‌ای که خروجی می‌دهید فقط باید به فرم زیر باشد:

```
#include<stdio.h>

// Helper macros for common operations
#define printInt(k) printf("%d\n", k)
#define printDouble(x) printf("%f\n",x)
#define printString(t) printf("%s\n", t)

// Functions to read integers and doubles from the user
int readInt() {
      int x;
      scanf(" %d", &x);
      return x;
}

double readDouble() {
      double x;
      scanf(" %f", &x);
      return x;
}

// Define a union for different types of cells
union cell {
      int i;
      double d;
      void* l;
};

// Define a large array of cells to act as the memory

int stack_size = 1000000;
union cell m[stack_size];
int top = 0; // Represents the current top of the stack

int bottom = stack_size - 1; // Represents the start of the current
function's activation record



int main() {
```

```
    // Initialize your environment or variables if necessary
    // For example, setting the initial top of the stack or initializing
variables

    // Your three-address code statements go here
    // Replace statement_1, statement_2, ..., statement_n with actual
operations
    // Example:
    // m[top].i = 5; // Assigning integer 5 to the top of the stack
    // top++; // Moving top of the stack

    // Use labels as targets for goto statements
    // Labels correspond to positions in your three-address code where
control may jump to
    label_1:
    // Corresponding operations for label_1
    // ...

    // Continue with other statements and labels as needed
    // ...

    return 0; // End of main function
}
```

هر یک از statement ها باید به یکی از فرم‌های زیر باشند: (منظور از N اعداد صحیح نامنفی است و $j,k,r \in N$.) دقت کنید j,k,r به صورت ثوابت عددی در برنامه خروجیتان ظاهر می‌شوند و متغیر نیستند چرا که کامپایلر دقیقا مقدار آنها را هنگام تولید کد می‌داند. (به کمک جدول نمادها یا مفاهیم دیگر)

1. `m[top + j].{i, d} = m[top + k].{i, d} op m[top + r].{i, d};` ($op \in \{+,-,*,/,\%\}$)

2. `m[top + j].{i, d} = op m[top + k].{i, d};` ($op \in \{-,!\}$)

3. `m[top + j].{i, d} = m[top + k].{i, d};`

4. `printInt(m[top + j].i);`

5. `printDouble(m[top + j].d);`

6. `printString("some string");`

7. `m[top + j].i = readInt();`

8. `m[top + j].d = readDouble();`

9. `top += j;`

10. `top -= j;`

11. `top = m[top + j].i;`

12. `m[top + j].i = top;`

13. `goto label;`

14. `goto *(m[top + j].l);`

15. `m[top + j].l = &&label;`

16. `if (m[top + j].{i, d} relop m[top + k].{i, d}) goto label;` (*relop* ∈ {<,>,<=,>=,==,! =})

```
// Activation Record
```

17. `bottom -= j;`

18. `bottom += j;`

```
// Each record takes up four spaces. Therefore, don't forget to add
the bottom += 4 at the end.

// Write
```

19. `m[bottom].i = m[top + k].i; m[bottom + 1].i = m[top + r].i; m[bottom + 2].d = m[top + e].d; m[bottom + 3].i = m[top + g].i; bottom += 4;`

```
// Read
```

20. `m[top + j].i = m[top + k].{i, d}`

دقت کنید در ابتدای هر statement حداکثر یک label می‌تواند قرار بگیرد. همچنین می‌توان کد را به گونه‌ای تغییر داد که نیازی به دستورات;m[top + j].i = top; و top = m[top + j].i نباشد. در واقع دو دستور ;top += j و ;top -= j کافی هستند. (خوب است درباره دلیل این موضوع فکر کنید و سعی کنید بدون استفاده از این دو دستور کد تولید کنید ولی برای راحتی کارتان نیازی نیست این مورد را در کدتان اعمال کنید.) دقت کنید این چهار دستور هنگام فراخوانی توابع و بازگشت از توابع کاربرد دارند چرا که top به ابتدای رکورد فعالیت سراستک اشاره می کند و تغییرات آن هنگام ایجاد یا حذف رکورد فعالیت رخ می دهد. در واقع ذخیره مقداری از حافظه در top

هنگام بازگشت از تابع کاربرد دارد و گویی مقدار control link یا محل رکورد فعالیت قبلی در top ذخیره می شود. همچنین ذخیره مقدار top در حافظه هنگام فراخوانی تابع کاربرد دارد چرا که به کمک آن می توان ابتدای رکورد فعالیت فعلی را به عنوان control link در رکورد فعالیت تابع جدید ذخیره کرد تا هنگام بازگشت از آن این مقدار بازیابی شود.

Example 1 - Operation:

```
{
int a = 5;
int b = 10;
int c = a + b;
}
```

Output:

```
// Macros, Functions and Union Cell declaration


int main() {
    // Initialize variables on the stack
    m[top].i = 5; // Assigning 5 to a, assuming 'a' is at m[top].i
    top++;
    m[top].i = 10; // Assigning 10 to b, assuming 'b' is at m[top].i
    top++;

    // Calculating c = a + b and assigning it to next position on stack
    m[top].i = m[top - 2].i + m[top - 1].i; // Assuming 'c' is at m[top].i

    // Print results or further manipulate variables
    // For example, to print 'c':
    printInt(m[top].i); // This will print the value of c, which is 15

    // Decrement top if you are done with variables
    top -= 3; // Adjust 'top' accordingly based on your program's needs

    return 0;
}
```

Example 2 - Function

```
int add(int x, int y) {
      return x + y;
}
```

Output:

```
// Macros, Functions and Union Cell declaration

int main() {
    // Save the current location of the bottom in the stack before calling
the function
    m[top].i = bottom; // Save previous bottom (return address) to the top
    top--;

    // Assign parameters to the new activation record
    bottom -= 4; // Move bottom to allocate new space for activation record
    m[bottom].i = 5;  // x
    m[bottom + 1].i = 10;  // y

    // Jump to the add function label
    goto add;

    add:
        // Perform addition using the activation record
        m[bottom + 2].i = m[bottom].i + m[bottom + 1].i;  // Adding x and y
        printInt(m[bottom + 2].i);  // Print the result of addition

        // After the function, restore the bottom to the previous
activation record
        top++;  // Move top back to get the previous bottom
        bottom = m[top].i; // Retrieve previous bottom (return address)

        // Simulate return by jumping to the end of the function (end of
main in this case)
        goto end;

    end:
        return 0; // End of main function
}
```

در مثال فوق توجه کنید که مقادیر ورودی تابع قبل از صدا زده شدن در انتهای استک و به صورت Activation Record ذخیره شده‌اند. در صورتی که تعداد ورودی تابع بیشتر باشد می‌بایست از Activation Recordهای بیشتری استفاده کنید. در نهایت وقتی تابع تمام می‌شود پوینتر استک به مقدار قبل اجرا برمی‌گردد تا مقادیر deallocate شوند.

Example 3 - Loop

```
int a = 10;
```

```
while (a > 0) {
      a = a - 1;
}
```

Output:

```
// Macros, Functions and Union Cell declaration


int main() {
    // Initialize variable a to 10
    m[top].i = 10;

    // Start of while loop
    loop_start:
        // Condition check: if a <= 0, exit the loop
        if (m[top].i <= 0) {
            goto loop_end;
        }

        // Body of while loop: a = a - 1
        m[top].i = m[top].i - 1;

        // Go back to the start of the loop
        goto loop_start;

    // End of while loop
    loop_end:

    // Print the final value of a (should be 0)
    printInt(m[top].i);

    return 0;
}
```

به صورت خیلی خلاصه کاری که باید در این مرحله انجام دهید این است که در کد سه آدرسه، آرایه‌ای از structها را به صورت مداوم بروزرسانی کنید و دستورالعمل را پیش ببرید. مثال‌هایی که در این تعریف پروژه مطالعه کردید دید خوبی از عملکرد کلی پروژه می‌دهند. کد سه آدرسه تولید شده برای هر دانشجو می‌تواند متفاوت باشد ولی برای گرفتن نمره‌ی کامل باید تمامی منطق لازم به درستی پیاده‌سازی شده باشد. همچنین کدی که تولید می‌کنید در نهایت باید توسط زبان C اجرا شود. مثال‌هایی که نوشته شده است لزوما با کد شما سازگار نخواهد بود و با نوع خاصی از پیاده‌سازی گرامر سازگار می‌باشد پس جواب هر دانشجو متفاوت خواهد بود.

امتیازی

- همانطور که در تعریف پروژه بخش‌های مربوط به Activation Record و توابع سبز شده‌اند، پیاده سازی این بخش امتیازی می‌باشد.
- به کامپایلر پروژه یک type checker اضافه کنید. این type checker باید خطاهای نوع در کد ورودی را تشخیص دهد و در صورت وجود خطای نوع، پیغام مناسب چاپ کند و از کامپایل شدن برنامه‌ی ورودی جلوگیری کند.
- می‌توانید کد را به گونه‌ای تغییر دهید که متغیرهای موقتی که هنگام محاسبه expression ها تولید می‌شوند فقط در صورت لزوم در Activation Record تابع ذخیره شوند. در حالت فعلی متغیرهای موقتی که برای محاسبه expression های بدنه هر تابع تولید می‌شوند در رکورد همان تابع ذخیره می‌شوند. اما بسیاری از این متغیرها می‌توانند بین توابع مختلف به اشتراک گذاشته شوند و تنها در صورت لزوم در رکورد فعالیت ذخیره شوند. این کار باعث می‌شود از حافظه مصرفی رکوردهای فعالیت کاسته شود.