برای یک زبان ساده با تنها دو نوع دستوری frontend در این پروژه، هدف ما طراحی یک کامپایلر است:

```
Type int a، b، c؛
a = (b * c) + a / b
```

که شامل تعریف یک متغیر از نوع int و اختصاص دادن مقادیر و عبارات به آن است.
ما ابتدا گرامری را که این زبان را توصیف می‌کند تعریف می‌کنیم:

```
Goal -> (Statement;)*
Statement -> Define | Assign
Define -> type int Id (, Id)*
Assign -> Id = Expr
Expr -> Term ((+|-) Term)*
Term -> Factor ((*|/) Factor)*
Factor -> Id | Number | (Expr)
Id -> (^[0-9][a-zA-Z0-9])+
Number -> [0-9]+
```

ما در این گرامر به دنبال Goal هستیم که یک دنباله از Statements، جدا شده با نقطه و کاماست.
هر دستوری یا تعریف یک متغیر است یا یک اختصاص. در یک اختصاص، یک شناسه (که به طور معمول یک نام متغیر است) و یک عبارت (Expr) اختصاص داده می‌شود. یک Expr شامل عبارت‌هایی است که به صورت ریاضی با هم عمل می‌کنند. برای اطمینان از صحت حضور درست، ابتدا non-terminals (یا گره‌های کم عمق در درخت AST) حاوی جمع و تفریق را مشاهده می‌کنیم و در ارتفاع‌های معمولی‌تر، ضرب و تقسیم را مشاهده می‌کنیم. شناسه‌ها (یا نام‌های متغیر) شامل حروف الفبایی و ارقام هستند، اما باید حداقل 1 کاراکتر و نباید با یک عدد شروع شود.
در لایه اول frontend، در لکسر، جریان کاراکترها را پردازش می‌کنیم و بر اساس آن‌ها توکن مناسب را ایجاد می‌کنیم. به عبارت دیگر، تجزیه لغوی رخ می‌دهد.

ما در ابتدا هر نوع کاراکتر فضای سفید مانند ' '، '\t'، 'n' و ... را نادیده می‌گیریم.

```
while (*BufferPtr && charinfo::isWhitespace(*BufferPtr))
{
++BufferPtr;
}
```

اگر شرط برقرار نشود، آنگاه یا این یک کاراکتر نال است یا یک کاراکتر غیر فضای سفید است.

```
if (!*BufferPtr)
{
token.Kind = Token::eoi;
return ;
}
```

اگر کاراکتر نال باشد، یک توکن پایان ورودی (eoi) ایجاد می‌شود.
سپس بررسی می‌کنیم که آیا این یک حرف است یا خیر. در این صورت، کلمه کامل را برای بررسی کلمات کلیدی یا شناسایی شناسه پیدا می‌کنیم.
تابع Name رشته‌ای از توالی مشخص شده از کاراکترها در بوفر دریافت می‌کند. سپس بر اساس متن، نوع توکن مناسب را اختصاص می‌دهیم. اگر این عدد باشد، مانند قبل عمل می‌کنیم و توکن را تشکیل می‌دهیم.

```
if (charinfo::isLetter(*BufferPtr))
{
const char *end = BufferPtr + 1 ;
while (charinfo::isLetter(*end))
++end;

llvm::StringRef Name(BufferPtr, end - BufferPtr);
Token::TokenKind kind =
Name == "type" ? Token::KW_type : (Name == "int" ? Token::KW_int :
Token::ident);
formToken(token, end, kind);
return ;
}


else if (charinfo::isDigit(*BufferPtr))
{
const char *end = BufferPtr + 1 ;
while (charinfo::isDigit(*end))
end = end + 1 ;
formToken(token, end, Token::number);
return ;
}
```

اگر هیچ کدام از موارد فوق برقرار نباشد، آنگاه باید یکی از کاراکترهای خاص مانند + یا ; باشد.

```
else
{
switch (*BufferPtr)
{
#define CASE(ch, tok) \
case ch: \
formToken(token, BufferPtr + 1 , tok); \
break
CASE ( '+' , Token::plus);
CASE( '-' , Token::minus);
CASE( '*' , Token::star);
CASE( '/' , Token::slash);
CASE( '=' , Token::equal);
CASE( '(' , Token::Token::l_paren);
CASE( ')' , Token::Token::r_paren);
CASE( ';' , Token::Token::semi_colon);
CASE( ',' , Token::Token::comma);
#undef CASE
default :
formToken(token, BufferPtr + 1 , Token::unknown);
}
return ;
}
```

اگر در هیچ یک از دسته‌ها نیومده باشد، آنگاه این یک کاراکتر معتبر نیست و از نوع توکن ناشناخته است که باید بعداً با آن برخورد شود. بنابراین، در حال حاضر به پارسر می‌رسیم که توکن‌های تولید شده را پردازش می‌کند و درخت AST را تولید می‌کند و بررسی می‌کند که کد داده شده قواعد زبان را رعایت می‌کند یا خیر.

```
AST *Parser::parseGoal()
{
llvm::SmallVector<Statement *, 8 > Statements;
Statement *statement;
while (!Tok.is(Token::eoi) && (statement = parseStatement()))
{
Statements.push_back(statement);
}
return new Goal(Statements);
}
```

تابع parseGoal() به صورت تکراری دستورات را تا زمانی که به توکن پایان ورودی(eoi) برسد، پارس می‌کند. این تابع دستورات پارس شده را در یک بردار جمع‌آوری کرده و یک شیء AST را با ریشه درخت AST کد پارس شده بازمی‌گرداند. اگر تابع parseStatement() یک مقدار غیر null، به این معنی که یک دستور صحیح پارس شده است، بازگشت دهد، حلقه ادامه پیدا می‌کند.
در داخل حلقه while، Statements.push_back(statement) دستور پارس شده را به بردار دستورات Statements اضافه می‌کند. پس از حلقه، return new Goal(Statements) یک شیء جدید Goal ایجاد می‌کند که بردار Statements را به عنوان یک آرگومان می‌گیرد. شیء Goal ریشه درخت نحوی (AST) برای کد پارس شده را نشان می‌دهد.
در تابع parseStatement:

```
if (Tok.is(Token::KW_type))
{
llvm::SmallVector<llvm::StringRef, 8> Vars;
advance();
if (expect(Token::KW_int))
{
goto _error;
}
advance();
if (expect(Token::ident))
goto _error;
Vars.push_back(Tok.getText());
advance();
while (Tok.is(Token::comma))
{

advance();
if (expect(Token::ident))
goto _error;
Vars.push_back(Tok.getText());
advance();
}
if (consume(Token::semi_colon))
goto _error;
return new typeDecl(Vars);
}
```

اگر (Tok.is(Token::KW_type)) باشد، این شرط بررسی می‌کند که آیا توکن فعلی (Tok) یک توکن کلیدواژه نوع است که نشان دهنده تعریف نوع است یا خیر. اگر چنین باشد، کد درون این شرط اجرا می‌شود. if (expect(Token::KW_int)) { goto _error ;} بررسی می‌کند که آیا توکن بعدی کلمه "int" است یا خیر. اگر نباشد، به برچسب _error پرش می‌کند. advance() نشانگر پرش لکسر به توکن بعدی است. و غیره.

```
if (Tok.is(Token::ident))
{
Expr *Left = new Factor(Factor::Ident, Tok.getText());
advance();
if (expect(Token::equal))
goto _error;
BinaryOp::Operator Op = BinaryOp::Equal;
Expr *E;
advance();
E = parseExpr();
if (consume(Token::semi_colon))
goto _error;
Left = new BinaryOp(Op, Left, E);
return Left;
}
```

اگر شرط اول برقرار نشد (Tok.is(Token::KW_type))، کد شرط بعدی (if (((Tok.is(Token::ident)) را بررسی می‌کند. این شرط برای دستوراتی که یک شناسه به عنوان سمت چپ یک عبارت انتساب دارند، استفاده می‌شود. Expr *Left = new Factor(Factor::Ident, Tok.getText()); یک گره عبارت جدید Left را ایجاد کرده و شناسه را نشان می‌دهد. advance() لکسر را به توکن بعدی منتقل می‌کند. { (if (expect(Token::equal)) goto _error; } بررسی می‌کند که آیا توکن بعدی یک علامت مساوی است یا خیر. اگر نیست، به برچسب _error پرش می‌کند. BinaryOp::Operator Op = BinaryOp::Equal; عملگر مساوی را به Op اختصاص می‌دهد. Expr *E; یک اشاره‌گر E به یک گره عبارت تعریف می‌شود. advance() لکسر را به توکن بعدی منتقل می‌کند. E = parseExpr(); یک تابع parseExpr() را فراخوانی کرده و عبارت سمت راست انتساب را پارس می‌کند و نتیجه آن را به E اختصاص می‌دهد. if (consume(Token::semi_colon)) { goto _error; }" بررسی می‌کند که آیا توکن بعدی یک جداسازی نقطه‌ویرگول است یا خیر. اگر نیست، به برچسب _error پرش می‌کند.
Left = new BinaryOp(Op، Left، E);" یک گره عملگر دودویی جدید را ایجاد می‌کند.
در پایان با کمک IR، AST کد تولید می‌شود.