الف) و ب)

در فاز اول این پروژه و در گام اول ، گرامر این زبان برنامه نویسی فرضی را به علت تسلط بیشتر شخصی بر زبان PHP و همچنین سادگی نگارش با همین زبان توسعه می‌دهیم. مفاهیم گرامری این زبان شامل درک درست از variable ها و Operator ها می‌باشد که به صورت فرمال با استفاده از اصول متداول در نظریه زبان‌ها تعریف شده و در مجموع ، پذیرش انواع متغیر و چهار عمل اصلی (جمع،تفریف،ضرب،تقسیم) از اهداف این زبان است.

همچنین، می‌توانیم با استفاده از این گرامر و زیرساخت LLVM ، زبان برنامه‌نویسی خود را پیاده‌سازی کنیم و با استفاده از زبان C کد آبجکت تولید و اجرا کنید.

پ) و ت) و ث)

توسعه یک پارسر به صورت دستی و تحلیل معنایی بر روی AST (Abstract Syntax Tree) بخشی از پروژه ما خواهدبود. در ادامه، به طور خلاصه توضیح می‌دهیم که چگونه می‌توانیم این مراحل را انجام دهیم.

تولید پارسر (Parser) : برای تولید پارسر، ما باید از گرامر زبانی که در بخش قبلی تعریف کرده‌ایم، استفاده کنیم. می‌توانیم از روش‌های مختلف برای پیاده سازی پارسر استفاده کنیم، مانند بازگشت به عقب (Backtracking) ، LL1، LR و غیره. با استفاده از پارسر، ورودی را (کد برنامه) خوانده و AST را تولید میکنیم. برای تطبیق با استاندارد کد میانی LLVM ، نیاز به طراحی قواعد پارسر و تولید ساختار AST متناسب با LLVM IR داریم.

تحلیل معنایی (Semantic Analysis) : در این مرحله، روی AST تولید شده توسط پارسر پیمایش می‌کنیم و قواعد معنایی زبان برنامه‌نویسی خود را اعمال می‌کنیم. به عنوان مثال، می‌توانیم بررسی کنید که آیا تقسیم بر صفر انجام شده است یا نه، یا بررسی کنیم که نوع داده‌ها درست استفاده شده باشند. در این مرحله، ممکن است نیاز به چک کردن نحوه استفاده از متغیرها، توالی دستورات، صحت نوع داده‌ها و غیره داشته باشیم. در صورت وجود هرگونه خطا یا ناسازگاری، می‌توانیم پیغام خطا را گزارش کنیم.

توجه داشته باشیم که پیاده‌سازی پارسر و تحلیل معنایی مراحل پیچیده‌ای هستند و نیاز به آشنایی با مفاهیم زبان‌های برنامه‌نویسی و نظریه زبان‌ها دارند. همچنین، برای ایجاد همخوانی با کد میانیLLVM ، نیاز به درک اصول و قوانین زبان LLVM IR و نحوه تولید کد آبجکت از AST دارید.

با توجه به توصیفاتی که در بخش‌های قبلی ارائه شد، می‌توان پارسر برای زبان برنامه‌نویسی فرضی را پیاده سازی کرد. در ادامه یک نمونه پیاده سازی ساده را در زبان C برای ارائه می‌دهیم. این نمونه پارسر، ورودی را به صورت یک رشته (string) می‌گیرد و آن را تحلیل کرده و یک AST ساده را تولید می‌کند. این نمونه در فایل AST.c ارائه شده. این نمونه پارسر، عبارت ریاضی "a = 5" را تحلیل می‌کند و نتیجه را به صورت AST نمایش می‌دهد. ما می‌توانیم ورودی را تغییر دهیم و سایر عبارات ریاضی را نیز تست کنیم.

تشریح کد AST.CPP (به علت کامنت نویسی فارسی خواهشمند است source code با نرم افزار VS-Code یا نرم‌افزارهای مشابه باز شود.)

در سورس‌کد AST.CPP نمونه‌ای از یک پارسر ساده برای زبان برنامه‌نویسی فرضی با استفاده از زبان C آمده است. این پارسر برای تحلیل عبارات ریاضی ساده استفاده می‌شود.

1. در بخش اول کد، متغیرها و تعاریف توابعی که در طول کد استفاده می‌شوند، تعریف شده‌اند.

2. تابع `getNextToken` برای دریافت توکن بعدی از رشته ورودی استفاده می‌شود. این تابع با استفاده از متغیر `position` به عنوان اشاره‌گر به موقعیت جاری در رشته، توکن‌ها را بررسی می‌کند و اگر متغیر `currentToken` متعلق به نوع توکن `TOKEN_NUMBER` یا `TOKEN_VARIABLE` باشد، مقدار یا نام توکن را در `currentToken` ذخیره می‌کند.

3. توابع `makeNumberNode` و `makeVariableNode` برای ساختن نودهای متغیر و عددی استفاده می‌شوند. این نودها در ساختار درختی AST استفاده می‌شوند.

4. تابع `makeBinaryOpNode` برای ساختن نودهای عملگر دومینه استفاده می‌شود. این نودها نیز در ساختار AST قرار می‌گیرند.

5. تابع `parseParenthesesExpression` برای تحلیل عباراتی استفاده می شود که درون پرانتز قرار دارند. این تابع ابتدا توکن بعدی را می‌خواند و سپس عبارت را به کمک تابع `parseExpression` تحلیل می‌کند.

6. تابع `parseAtomicExpression` برای تحلیل عبارات عددی و متغیرها استفاده می شود. این تابع ابتدا توکن بعدی را بررسی می‌کند و در صورت مطابقت با توکن‌های `TOKEN_NUMBER` یا `TOKEN_VARIABLE`، نود متناظر را ساخته و با توجه به نوع توکن مقدار یا نام آن را در آن ذخیره می‌کند.

7. تابع `parseFactor` برای تحلیل عبارات فاکتوری استفاده می شود. این تابع ابتدا توکن بعدی را بررسی می‌کند و اگر با یکی از توکن‌های `TOKEN_NUMBER` یا `TOKEN_VARIABLE` مطابقت داشت، تابع `parseAtomicExpression` را صدا می‌زند تا فاکتور مربوطه را تحلیل کند.

8. تابع `parseTerm` برای تحلیل عبارات ترمی استفاده می‌شود. این تابع ابتدا تابع `parseFactor` را صدا می‌زند تا فاکتور اولیه را تحلیل کند و سپس در صورت وجود عملگرهای ضرب یا تقسیم بین فاکتورها، تحلیل را ادامه می‌دهد.

9. تابع `parseExpression` برای تحلیل عبارات ریاضی استفاده می‌شود. این تابع ابتدا تابع `parseTerm` را صدا می‌زند تا ترم اولیه را تحلیل کند و سپس در صورت وجود عملگرهای جمع یا تفریق بین ترم‌ها، تحلیل را ادامه می‌دهد.

10. در بخش اصلی برنامه، متغیرهای لازم برای شروع تحلیل (مانند `input` و `position`) تعریف شده‌اند و تابع `parseProgram` صدا زده می‌شود تا تحلیل برنامه‌ی ورودی را شروع کند.

11. در پایان برنامه، عبارت "a = 5;" در `input` تعریف شده است و تابع `parseProgram` صدا زده می‌شود تا تحلیل این عبارت را انجام دهد.

با اجرای این برنامه، خروجی به صورت AST نمایش داده می‌شود. برای عبارات دیگر، می‌توانید ورودی را تغییر داده و خروجی AST را مشاهده کنید.

تشریح کد output.c (به علت کامنت نویسی فارسی خواهشمند است source code با نرم افزار VS-Code یا نرم‌افزارهای مشابه باز شود.)

بطور خلاصه، کد بالا یک برنامه ساده را نشان می‌دهد که از زیرساخت LLVM استفاده می‌کند تا کد IR (Intermediate Representation) را از AST (Abstract Syntax Tree) تولید کند. در اینجا جزئیات بیشتری از کد را برای شما تشریح می‌کنم:

1. تعریف متغیرها:

- از کتابخانه LLVM-C استفاده می‌کنیم تا تعریف‌های مربوط به LLVM را در C استفاده کنیم.

- متغیرهای a، b و c به عنوان متغیرهای سراسری ایجاد می‌شوند با استفاده از LLVMAddGlobal.

- نوع داده متغیرها به عنوان integer تعریف می‌شود با استفاده از LLVMInt32Type.

2. تعریف بلوک‌ها:

- سه بلوک اصلی با نام‌های entry، calc و end ایجاد می‌شوند با استفاده از LLVMAppendBasicBlock.

3. تعریف تابع اصلی:

- تابع اصلی main تعریف می‌شود با استفاده از LLVMAddFunction.

- نوع تابع اصلی به عنوان LLVMFunctionType تعریف می‌شود.

- بلوک ورودی برای تابع اصلی با استفاده از LLVMAppendBasicBlock تعریف می‌شود.

4. ساخت برنامه‌نویسی توابع:

- یک برنامه‌نویسی توابع (LLVMBuilderRef) ایجاد می‌شود با استفاده از LLVMCreateBuilder.

5. قرار دادن دستورات در بلوک‌ها:

- دستورات محاسباتی مورد نیاز برای محاسبه مقدار متغیر c را در بلوک calc قرار می‌دهیم.

- از LLVMPositionBuilderAtEnd برای تنظیم مکان قرار گیری دستورات استفاده می‌کنیم.

- از توابع مختلفی مانند LLVMBuildAdd، LLVMBuildSub و LLVMBuildMul برای ساخت دستورات محاسباتی استفاده می‌شود.

- برای ذخیره نتیجه محاسبات در متغیر c از LLVMBuildStore استفاده می‌شود.

- با استفاده از LLVMBuild

RetVoid، دستور بازگشت در انتهای تابع اصلی تعریف می‌شود.

6. ذخیره کد IR در فایل:

- با استفاده از LLVMWriteBitcodeToFile، کد IR در فایل "output.ll" ذخیره می‌شود.

7. آزادسازی منابع:

- با استفاده از LLVMDisposeBuilder و LLVMDisposeModule، منابع مورد نیاز آزاد می‌شوند.

با تشکر از حسن توجه حضرتعالی.

سید امیر پارسا نصیری - **۹۹۲۴۳۱۱۱**