Trabalho de Compiladores
Linguagem C-

Leonardo Marques de Melo, Mateus Mendes da Silva, Matheus PIassi de Carvalho - 10A
Professor Ricardo Terra Nunes Bueno Villela
GCC130 - Compiladores

1. Introdução

Este trabalho, desenvolvido para a disciplina de Compiladores sob a orientação do Prof. Ricardo Terra, tem como objetivo a implementação progressiva de um compilador, seguindo as principais fases de um compilador, conforme abordado em sala de aula. A construção do compilador será realizada em etapas com, análise léxica, análise sintática e análise semântica, aplicando os conceitos teóricos estudados durante as aulas ministradas da disciplina.

Para a realização da primeira etapa, a análise léxica, será utilizada a ferramenta FLEX, que permite a definição de regras para a identificação de padrões no código e a geração de tokens. Esses tokens servirão como base para as fases subsequentes, garantindo a correta estruturação do processo de compilação.

Na análise sintática, utilizaremos a ferramenta BISON, que analisará o código fonte e os tokens gerados, e irá encontrar possíveis erros sintáticos

2. Referencial Teórico

2.1 Introdução a Compiladores

Um compilador é um programa que traduz código escrito em uma linguagem de programação (código-fonte) para uma linguagem de baixo nível, geralmente assembly, chamado, código objeto ou executável.

Diversas linguagens de programação utilizam compiladores para realizar suas traduções do código fonte para código de máquina, entre elas, C, C++, Java.

Para chegar no código objeto, o compilador passa por algumas fases, entre elas temos a análise léxica, análise sintática e análise semântica, e, após essas fases ainda temos por exemplo a otimização do código objeto.

2.2 Análise Léxica

A análise léxica é a fase inicial do processo de compilação, onde o código-fonte é transformado em uma sequência de tokens.A partir de expressões regulares, utilizadas para definir padrões que tokens possam seguir, definir palavras reservadas, que identificam os lexemas e os transformam em tokens, para que as próximas fases da compilação possam acontecer.

Por exemplo na seguinte expressão “float x = 5 + y;” Na análise léxica os seguintes tokens seriam reconhecidos na linguagem C :

float -> tipo
x -> identificador
= -> atribuição
5 -> int
+ -> operador
y -> identificador
; -> delimitador

Assim, nessa etapa do compilador esses lexemas são identificados em tokens, para que possam ser usados nas etapas seguintes.

2.2 Análise Sintática

Logo após o analisador léxico dividir o código fonte em diferentes tokens, retirando espaços e comentários, o compilador pega os tokens e verifica se eles respeitam uma ordem, seguindo as regras gramaticais da linguagem, essa fase da compilação se chama análise sintática.

Assim, tratando a estrutura do código, tentando construir uma árvore sintática que possa derivar a regra, e no caso de um erro sintático, essa parte deve conseguir se recuperar, e continuar derivando para que outros erros consigam ser identificados e informados.

2.3 Análise Semântica

A análise semântica é a terceira fase do processo de compilação, ocorrendo após as análises léxica e sintática. Enquanto a análise sintática verifica se o programa está estruturalmente correto conforme a gramática da linguagem, a análise semântica foca em validar o significado e a coerência lógica do código.

Durante esta fase, erros semânticos são identificados, como o uso de variáveis não declaradas, incompatibilidade de tipos em atribuições, tipo incorreto na chamada de funções.

3. Linguagem C-

3.1 Parte 1 - Análise Léxica

Começando o desenvolvimento do trabalho da fase de análise léxica de um compilador com a ferramenta FLEX, na aula do laboratório em que tivemos o primeiro contato e começamos a interagir e entender o funcionamento, de algo que vimos teoricamente na sala de aula. Em um primeiro momento, a ferramenta parecia estranha, mas conseguimos entender o funcionamento de suas expressões regulares, e a partir daí já tínhamos um início do desenvolvimento do trabalho.

Já com um início do trabalho e tendo entendido a ferramenta, começamos a criar as expressões regulares restantes, algumas tivemos mais dificuldades, como comentários e string, para strings tivemos um problema que era possível ter aspas duplas dentro da String, o que não é possível em linguagens como C++, para comentários adotados um padrão com um OR para comentários de única ou múltipla linha, e assim, tendo um único token de comentário para descartá-lo.

Para o tratamento de identificação de linhas e colunas tivemos que criar 2 variáveis iniciais, que a cada “\n” encontrado durante a leitura do código-fonte, é adicionado um a contagem de linhas, e a variável column_number, que identifica a coluna do lexema, que a cada “\n” encontrado ela é definida para 1, e a cada lexema lido é adicionado “yyleng”, que é o tamanho do lexema, assim conseguimos saber exatamente a localização do lexema e identificar possíveis erros e informá-los previamente.Para tratar um problema onde as tabulações não eram contadas, e assim fazendo com que o número da coluna fosse identificado de maneira errada, foi adicionado uma regra que ao encontrar “\t” o yyleng dele fosse adicionado ao contador da coluna.

Alguns dos desafios que tivemos foram:

1 - Como definir o que deveria ser feito na análise léxica ou na análise sintática? Em alguns pontos, tivemos dificuldades, inicialmente, em perceber que a função da análise léxica era apenas de encontrar os tokens e identificar erros léxicos, e não garantir a ordem deles ou estruturas mais complexas.

2 - Deveríamos criar regras para números inteiros e outra para números de ponto flutuante? No fim, percebemos que futuramente talvez uma definição separada venha a facilitar as coisas na análise sintática.

3 - Como identificar erros léxicos? Por fim, determinamos que o melhor caminho seria utilizar expressões regulares para identificar os erros. Decidimos trabalhar com 5: números de ponto flutuante com mais de um ponto, strings que não finalizam com aspas duplas, ID's iniciados com números, números de ponto flutuante escritos com vírgula, e ID's com símbolos inválidos.

Também foram feitos diversos testes em diferentes arquivos, visando testar os tokens e variações problemáticas deles. No fim, geramos um teste com todos os tokens, e um teste com todos os erros, para garantir que todas as expressões regulares eram funcionais.

Conseguimos implementar uma boa formatação das informações que são mostradas no terminal, seguindo a sequência de "Linha | Coluna | Lexema | Token", facilitando a visualização dos tokens encontrados pela análise léxica, e também a visualização das informações e erros encontrados.

Após todo esse processo, produzimos os diagramas de transição utilizando o sistema Draw.io para facilitar a elaboração destes. Os diagramas foram muito importantes, pois através deles conseguimos não só entender melhor o comportamento do analisador léxico, como também nos ajudou a identificar expressões regulares mal construídas ou erradas.

3.2 Parte 2 - Análise Sintática

A partir do que implementamos na análise léxica, começamos a desenvolver a análise sintática do nosso compilador, usando o programa Bison/Yacc. Antes de começar de fato, tivemos que corrigir alguns erros identificados na etapa anterior, como erros no cálculo de linhas em comentários, e um erro que tratava de erros com vírgulas.

Depois de participarmos da aula prática sobre o Bison, nós imediatamente começamos a trabalhar na nossa análise sintática, usando como base a descrição fornecida da linguagem C- (que consiste em uma gramática descrita usando a Forma Normal de Backus). Inicialmente, tentamos transcrever toda a gramática fornecida como base para o arquivo do Bison, o que nos poupou muito trabalho por um lado, mas dificultou a busca por erros no nosso código futuramente. A partir disso, passamos a trabalhar no nosso programa para que funcionasse adequadamente, usando como base os exemplos usados na aula prática e os pequenos trechos de código existentes na apostila da matéria.

Já no início do desenvolvimento, conseguimos encontrar algo bastante interessante: um conflito shift/reduce (que é quando o parser fica confuso sobre qual caminho seguir, se faz um shift, ou um reduce)! Ficamos bastante empolgados inicialmente (apesar de termos encontrado um problema), pois foi muito interessante

ver a teoria aparecendo na prática. A partir disso, precisamos refatorar algumas regras problemáticas, mas não conseguimos resolver todos.

Depois de algum tempo trabalhando em testes para cada regra, e de criarmos estruturas válidas e inválidas para a análise sintática, resolvemos implementar uma espécie de “debugging”, em que todas as produções consumidas são exibidas no terminal, o que nos ajudou bastante a entender os caminhos que o nosso analisador léxico estava tomando. Também removemos as saídas geradas pela análise léxica (exceto as dos erros) para ajudar na legibilidade.

Com tudo funcionando, começamos a desenvolver alguns tratamentos de erros sintáticos. O primeiro desafio foi entender como fazer para que a análise continue após achar um erro. Procurando na apostila da matéria conseguimos entender como fazer a recuperação de erros (precisávamos de definir uma referência, que, ao ser encontrada, permite que a análise continue a partir dali), e passamos a usar a palavra reservada “error” para as demais implementações.

Os erros escolhidos foram:

- Erro na Declaração de Função – Parâmetros Malformados (Geral)
- Erro na Lista de Parâmetros – Ausência de Vírgula
- Erro na Declaração de Variável – Identificador Inválido
- Erro na Declaração de Array – Tamanho Inválido
- Erro de Comando – Comando Incompleto ou Inexistente
- Erro em Comando de Seleção IF – Expressão Condicional Inválida
- Erro em Comando de Seleção IF – Parêntese de Fechamento Ausente
- Erro em Comando de Iteração WHILE – Expressão Condicional Inválida
- Erro em Comando de Retorno – Valor de Retorno Malformada
- Erro de Sintaxe Genérico – Função Malformada

Foi interessante revisar a gramática de modo a entender cada detalhe e definir onde poderiam existir erros. No fim, concluímos que existiam vários erros possíveis de serem tratados (o que mais uma vez confirmou o que aprendemos em aula), e escolher qual deles iremos utilizar foi um grande exercício de análise.

4. Conclusão

Para a primeira etapa do desenvolvimento do compilador da Linguagem C- , a análise léxica a partir do FLEX, conseguimos implementar todas as funções e expressões regulares que foram descritas no documento com as informações do trabalho prático. Com essa primeira etapa conseguimos compreender melhor como a análise léxica ocorre em um compilador real, as dificuldades e os diversos padrões que existem para detectar tokens e possíveis erros neles.

Os testes realizados com diversos códigos teste demonstraram que nosso analisador consegue identificar corretamente todos os tokens da linguagem C-, incluindo identificadores, palavras-chave, operadores, com uma taxa de precisão satisfatória. O sistema também se mostrou eficiente na detecção e relato de erros

léxicos, como números mal formados e strings não terminadas, fornecendo mensagens de erro claras com a indicação da linha e coluna do erro.

Para a etapa 2, após corrigirmos os erros encontrados na primeira etapa, foi possível entender de maneira satisfatória o funcionamento de um analisador sintático bottom-up. Apesar dos desafios iniciais, conseguimos compreender de forma mais aprofundada e prática sobre o desenvolvimento de um analisador sintático, onde pudemos experimentar diversas estruturas e trabalhar alguns erros da nossa linguagem.

Vale destacar que foi muito interessante ver os conceitos que aprendemos na matéria de Linguagens Formais e Autômatos serem aplicados até o momento. Ver como a teoria faz sentido em um compilador é muito gratificante, pois mostra que um conteúdo que às vezes parece distante e abstrato, realmente é funcional e aplicável.

5. Referências Bibliográficas

Repositório oficial do FLEX - westes/flex: The Fast Lexical Analyzer - scanner generator for lexing in C and C++. Disponível em: https://github.com/westes/flex

PEREIRA, Denilson Alves. Notas de aula da disciplina GCC130 - Compiladores. Lavras: Universidade Federal de Lavras, Departamento de Ciência da Computação, 2013.

MARANGON, Johni Douglas. Compiladores para Humanos. Disponível em: https://johnidm.gitbooks.io/compiladores-para-humanos/content/

KIELKOWSKI, Matheus. Como os Compiladores Funcionam?. Disponível em: https://matheuskiel.medium.com/como-compiladores-funcionam-c0647147487d