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 completo, seguindo as principais fases do processo de compilação conforme discutido ao longo das aulas teóricas e práticas. A construção do compilador foi estruturada em múltiplas etapas, cada uma correspondendo a uma fase clássica da arquitetura de compiladores: análise léxica, análise sintática, análise semântica, geração de código intermediário, e, futuramente, possíveis etapas de otimização e geração de código final. Esta abordagem incremental permite uma melhor compreensão da complexidade envolvida em cada fase e favorece o desenvolvimento de habilidades práticas essenciais na construção de linguagens de programação.

Para a realização da primeira etapa, a análise léxica, foi 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, utilizamos a ferramenta BISON, que analisará o código fonte e os tokens gerados em integração com o FLEX, e irá garantir que o código-fonte tem estruturas válidas, de acordo com as regras da linguagem. Neste processo, o analisador sintático deve também identificar erros de sintaxe.

Durante a análise semântica trabalhamos com a implementação de uma tabela de símbolos e de um gerador de código de três endereços, além de verificar que o código-fonte tenha sentido lógico. Além disso, o analisador semântico deve trabalhar na identificação de erros, como variáveis usadas sem declaração, e tipos inválidos.

2. Referencial Teórico

2.1. Introdução a Compiladores

Um compilador é um software responsável por traduzir programas escritos em uma linguagem de alto nível (como C, C++, Java, entre outras) para uma linguagem de mais baixo nível, geralmente código objeto ou linguagem de máquina, que pode ser diretamente executada por um processador. Diferente de um interpretador, que executa o código diretamente linha por linha, o compilador realiza a tradução completa do código-fonte antes da execução, o que costuma resultar em programas mais rápidos e eficientes.

O processo de compilação é dividido em diversas fases bem definidas, cada uma com funções específicas e interdependentes. Entre as etapas que compõem o front-end do compilador, destacam-se: a análise léxica, que quebra o código em tokens compreensíveis para o compilador; a análise sintática, que verifica a estrutura gramatical do código com base em uma gramática formal da linguagem; e a análise semântica, que garante que o código, além de estar corretamente estruturado, também tenha significado lógico dentro das regras da linguagem.

Essas fases são responsáveis por validar a estrutura e o significado do código-fonte antes que ele avance para etapas posteriores, como a geração de código intermediário, a otimização e finalmente a geração de código objeto. Juntas, essas fases constituem o pipeline de compilação, transformando abstrações de alto nível em instruções de máquina otimizadas para execução.

A construção de um compilador é uma tarefa desafiadora que exige o domínio de diversos conceitos da ciência da computação, como autômatos, linguagens formais, árvores de derivação, estruturas de dados, análise de escopos e tipos, além de habilidades práticas com ferramentas como *Flex*, *Bison*, ou equivalentes, que auxiliam na geração automática de partes do compilador, como os analisadores léxicos e sintáticos.

2.2. Análise Léxica

A *análise léxica* é a primeira etapa do processo de compilação. Nessa fase, o compilador percorre o código-fonte caractere por caractere com o objetivo de agrupar sequências de caracteres em unidades significativas chamadas *tokens*. Cada token representa uma categoria léxica da linguagem, como identificadores, palavras-chave, operadores, constantes numéricas, símbolos de pontuação, entre outros. Essa transformação é fundamental para que as próximas fases do

compilador possam trabalhar com estruturas mais abstratas e compreensíveis do que o texto do código-fonte em si.

A análise léxica é normalmente implementada com o uso de *expressões regulares*, que definem os padrões formais dos lexemas válidos para cada tipo de token. Por exemplo, uma expressão regular pode ser usada para definir o padrão de um identificador como uma letra seguida de letras ou dígitos, enquanto outra pode representar números inteiros ou palavras reservadas da linguagem.

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

Os tokens obtidos são usados nas etapas subsequentes para verificar estruturas de código e outras verificações de erro, conforme as regras da linguagem. Além disso, esta etapa trabalha para ignorar termos inúteis para a compilação, como comentários e espaços em branco.

2.3. 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*. Essas regras geralmente são descritas por Gramáticas Livres de Contexto, e são usadas para descrever a estrutura correta do programa, informando, por exemplo, a maneira como devem ser feitas as declarações, comandos, e blocos de código.

Nesta etapa o compilador constroi uma *árvore sintática* que representa o programa. Cada nó interno da árvore equivale à uma regra gramatical aplicada, e as folhas representam os *tokens.* Essa árvore é usada posteriormente em outras etapas da compilação, como na análise semântica e na geração de código intermediário.

Se a sequência de tokens analisada não estiver de acordo com as regras da linguagem, o analisador detecta um *erro de sintaxe.* Para tornar o analisador mais

eficiente, faz se necessário a utilização de técnicas de *recuperação de erro,* o que permite que a análise continue mesmo com um erro. Isso permite que o processo não pare no primeiro erro encontrado, e possibilita a detecção de múltiplos erros em uma só tentativa de compilação.

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. Isso inclui, por exemplo, verificações e conversão de tipos, escopo de variáveis, compatibilidade, dentre outros erros que não podem ser detectados pela análise sintática.

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. Para realizar essas validações, a *tabela de símbolos* (onde as informações sobre os identificadores são armazenadas) é consultada, onde é possível verificar informações como *nome, tipo* e *escopo* das variáveis.

Além disso, outra responsabilidade da análise semântica é a verificação de tipos, que é o que assegura que as operações serão feitas usando tipos compatíveis. Para isso, podem ser utilizadas conversões explícitas e implícitas, quando a linguagem permitir.

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

O segundo passo fundamental na construção de um compilador é a *análise sintática*, que representa uma etapa intermediária entre a análise léxica e a análise semântica. Essa fase é responsável por verificar se os tokens produzidos pela análise léxica estão organizados de acordo com a estrutura gramatical da linguagem de programação alvo. A análise sintática constrói uma *representação hierárquica* do programa — geralmente uma árvore sintática abstrata (AST) ou uma árvore de derivação — que reflete a maneira como os elementos do código-fonte se relacionam estruturalmente.

O principal objetivo da análise sintática é detectar erros de estrutura gramatical, como instruções malformadas, expressões com parênteses desbalanceados, declarações incompletas, entre outros problemas que não podem ser identificados apenas pela análise léxica. Para isso, utilizamos ferramentas de geração de analisadores sintáticos, como o Bison (Yacc), que, a partir de uma gramática formal da linguagem, gera um analisador capaz de reconhecer as estruturas sintáticas válidas. Também é válido ressaltar que é possível inserir trechos de código em C entre as regras, o que é muito útil para imprimir erros, e será bastante utilizado na análise semântica.

É importante destacar que a análise sintática depende fortemente da correta tokenização feita na fase léxica. Os tokens identificados pelo Flex/Lex são consumidos pelo Bison, que verifica sua ordenação e agrupações conforme a gramática. Caso haja inconsistências, mensagens de erro sintático são emitidas, e o analisador pode aplicar estratégias de recuperação de erro para continuar a análise e relatar múltiplos problemas no código-fonte.

Implementando as regras

O enunciado do trabalho propõe que a implementação da análise sintática seja realizada de forma incremental, a partir de uma gramática simplificada baseada na linguagem C-, com suporte a declarações, comandos condicionais, laços de repetição e expressões aritméticas. O principal desafio desta etapa foi adaptar a gramática formal para o formato do Bison, resolvendo conflitos como *shift/reduce* e implementando tratamento de erros personalizados para tornar o compilador mais robusto e informativo.

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.

Para fazer uma implementação correta, precisamos de entender com mais detalhes como a linguagem funcionava. Para isso, resolvemos implementar uma espécie de “debugging”, em que todas as produções consumidas são exibidas no terminal, o que nos ajudou bastante a compreender os caminhos que o nosso analisador estava tomando. Também removemos as saídas geradas pela análise léxica (exceto as dos erros) para ajudar na legibilidade.

Durante os testes iniciais, conseguimos encontrar um conflito shift/reduce (que é quando o parser fica confuso sobre qual caminho seguir, se faz um shift, ou um reduce). Quando o parser encontra uma cadeia como *if (cond) if (cond2) stmt; else stmt2;*, ele não sabe se o *else* pertence ao *if* mais interno ou ao mais externo. Para resolver essa ambiguidade, adicionamos as diretivas *%nonassoc LOWER_THAN_ELSE* e *%nonassoc ELSE*, e associamos a produção do *else_opt* ao símbolo *LOWER_THAN_ELSE* com a diretiva de precedência *%prec LOWER_THAN_ELSE.* Dessa forma, o Bison entende que o *else* deve ser associado ao *if* mais próximo, comportamento que está em conformidade com o padrão da linguagem C. Essa solução simples evita a reestruturação manual da gramática e elimina o conflito *shift/reduce*, tornando o parser mais confiável e fiel à semântica esperada.

A principal estratégia utilizada para corrigir os erros na implementação do analisador sintático foi criar exemplos que testassem as estruturas definidas pelas regras da linguagem. Conseguimos criar uma série de testes, alguns corretos e outros errados (seguindo as regras da linguagem), para garantir que o programa que fizemos cobria todos os requisitos do enunciado.

Tratamento de erros

Com tudo funcionando, começamos a desenvolver alguns tratamentos de erros sintáticos. O primeiro desafio foi entender se existia algum mecanismo do Bison para ajudar na detecção de erros. Além disso, precisávamos que a análise continuesse 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. Basicamente, podemos inserir a palavra reservada no lugar de algum não-terminal em uma regra, para capturar algum tipo específico de erro.

Os erros escolhidos foram:
- *Erro na Declaração de Função*: Verifica se a lista de parâmetros de uma função segue a sintaxe correta esperada pela linguagem. Isso inclui a presença de tipos válidos e identificadores apropriados.

- *Erro na Lista de Parâmetros:* Quando múltiplos parâmetros são declarados sem o uso da vírgula separadora (ex: *int a int b*), a gramática não consegue realizar a derivação correta da lista de parâmetros.

- *Erro na Declaração de Variável:* Um identificador malformado (como palavras reservadas usadas como nome de variável ou ausência total de nome após o tipo) causa erro na derivação da regra de declaração de variável.

- *Erro na Declaração de Array*: *Arrays* devem ser declarados com um valor inteiro positivo entre colchetes.

- *Erro de Comando:* representa qualquer estrutura que deveria ser um comando (como atribuições, chamadas de função, ou blocos *if, while, return, etc*.), mas que está malformada ou incompleta, como um ponto e vírgula isolado (;) ou uma expressão quebrada.

- *Erro em Comando de Seleção IF*: A expressão entre os parênteses de um if precisa ser uma expressão sintaticamente correta. Caso falte o conteúdo ou este esteja malformado (ex: *if ())* ou *if (5 +)*), o parser detecta o erro e imprime uma mensagem de erro contextualizada.

- *Erro em Comando de Seleção IF:* Caso o parêntese de fechamento da condição do if seja omitido (*if (x > 0*), o parser não consegue derivar a estrutura e aponta a ausência do fechamento da condição, permitindo melhor localização do erro para o programador.

- *Erro em Comando de Iteração WHILE:* Assim como no *if*, o *while* espera uma expressão condicional válida entre parênteses. Qualquer quebra na estrutura sintática dessa condição gera um erro específico de comando de iteração.

- *Erro em Comando de Retorno:* um *return* pode vir isolado (em funções *void*) ou com uma expressão. Caso a expressão esteja incorreta ou incompleta

(return x + ;), o erro é sinalizado com um aviso contextual, mantendo o fluxo de parsing.

- *Erro de Sintaxe Genérico*: Quando a estrutura da função inteira não consegue ser derivada corretamente (por exemplo, ausência de identificador, parâmetros e corpo), um erro de sintaxe genérico é emitido, indicando que a função está malformada e não pôde ser reconhecida como uma declaração válida.

Foi interessante revisar a gramática de modo a entender cada detalhe e definir onde poderiam existir erros. No fim, como foi aprendido durante as aulas, percebemos que existiam vários erros possíveis de serem tratados, e escolher qual deles utilizar foi interessante para entender mais aprofundadamente como a linguagem funciona, já que precisamos de replicar todos os erros para testá-los. As possibilidades de erros possíveis podem ser analisadas mais precisamente através da criação de uma tabela LR.

Após a apresentação do trabalho, foram identificados alguns erros. O primeiro deles era em relação à etapa anterior, de análise léxica, que consistia no fato de que ao escrever um número usando exponencial (usando um ‘E’, como em 24E2) nossa linguagem permitia que a escrita fosse usando um ‘e’, sendo que as regras da linguagem deveria ser um ‘E’, o que foi simples de resolver. Além disso, existia um problema na regra que gerava a *struct*, que, apesar de estar construída corretamente, não estava de acordo com as regras da linguagem. Para corrigir isso, tivemos que reestruturar a maneira como a regra para a *struct* funcionava, e depois disso conseguimos obter um resultado mais condizente com o esperado.

3.3. Parte 3 - Análise Semântica

O último passo do trabalho é a análise semântica, que é uma das últimas etapas do front-end de um compilador.

Seu principal objetivo é garantir a consistência semântica do programa fonte em relação à definição da linguagem, possibilitando uma análise mais aprofundada do que a análise de estruturas gramaticais feitas pela análise sintática para focar no significado e lógica do código. Além disso, outros objetivos desta etapa do compilador são:

- *Imposição de regras de linguagem não capturadas*: Nem sempre é possível expressar todas as regras da linguagem durante a análise sintática. Por isso, a análise semântica também age na imposição dessas regras que não foram capturadas, como: exigir que identificadores sejam declarados antes de

serem usados, ou garantir que um *break* estará sempre dentro de um loop, conforme as regras da linguagem.

- *Verificar e capturar informações de tipos***:** Garante que cada operador possua operandos compatíveis. Pode-se citar como exemplo disso garantir que uma soma será feita com operandos do mesmo tipo, que um vetor será sempre indexado usando um inteiro, ou mesmo que uma operação lógica de *and* será feita com booleanos, conforme as regras da linguagem. Dentro deste processo, existem as conversões de tipo que uma linguagem permite, que podem ser usadas pelo analisador semântico para criar operações visando garantir a compatibilidade de tipos.

- *Interagir com a tabela de símbolos*: A análise semântica reúne informações sobre os tipos e as salva na tabela de símbolos. Essas informações coletadas são cruciais para a geração de código intermediário e alocação na memória.

- *Gerar uma representação intermediária*: Um dos objetivos do front-end no geral é criar uma representação intermediária do programa, que será usada pelo back-end para gerar o código objeto. As informações coletadas pela análise semântica são extremamente importantes para esta última etapa.

Pode-se dizer que a análise semântica garante que o programa “faz sentido” logicamente.

É muito importante ressaltar que a análise semântica está intrinsecamente conectada às etapas anteriores do compilador. Na análise léxica (responsável por ler o programa fonte, e gerar tokens para cada lexema), os tokens gerados são usados posteriormente na análise semântica, além de facilitar o trabalho das outras etapas do front-end eliminando comentários e espaços em branco. A análise sintática (etapa que lida com a validação das estruturas do código-fonte, conforme as regras da linguagem) gera uma árvore sintaxe, ou árvore de derivação, que mostra a estrutura sintática do programa - árvore esta que é usada pela análise semântica para realizar suas verificações.

O enunciado do trabalho diz que devemos implementar a tabela de símbolos, o analisador semântico, e o gerador de código intermediário, incrementando as etapas anteriores feitas no Yacc/Bison e Flex/Lex. O primeiro desafio foi definir como nós faríamos a implementação de cada um desses requisitos.

Tabela de símbolos

Analisando a apostila da matéria e o livro usado como bibliografia básica, concluímos que o primeiro passo seria elaborar a tabela de símbolos, já que ela é

essencial para todas as etapas subsequentes da análise semântica. Em um compilador, a tabela de símbolos é uma estrutura central usada durante a análise semântica e a geração de código, podendo ser usada para validar o uso correto dos identificadores e associar cada um à uma posição da memória.

Primeiro, estudamos sobre como fazer a implementação de uma tabela hash usando C, o que nos levou à diversas implementações diferentes. Depois disso, integramos a nossa tabela com o código do Bison para realizar alguns testes simples de funcionalidade.

A versão inicial que tentamos implementar não era complexa, não tinha escopo nem uma maneira eficiente de guardar os dados que precisávamos, pois tinha como objetivo apenas entender se a nossa implementação estava estável e poderia ser utilizada para o resto do trabalho. Tendo garantido isso com alguns testes, implementamos uma *struct* chamada de *Symbol* para representar uma linha da tabela de símbolos, e que é composta por:

```c
typedef struct Symbol {
    char *name;                 // Nome do identificador
    tipoDado type;              // Tipo de dado
    tipoDado array_base_type;   // Tipo base do array (se for array)
    int address;                // Endereço relativo
    int scope_level;            // Nível de escopo
    int array_size;             // Tamanho do array (se for array)
    struct Symbol *next;        // Próximo símbolo na lista (para colisões)
} Symbol;
```

A ideia é armazenar o nome do identificador em *name*, seu respectivo tipo, endereço de memória, seu tamanho, e o próximo símbolo da tabela.

Em seguida, precisávamos entender como funcionam os escopos em uma tabela de símbolos. De acordo com o livro utilizado como bibliografia base pela matéria, entendemos que cada escopo de um código tem a sua própria tabela de símbolos, para garantir o isolamento das variáveis e definir a partir de onde elas podem ser acessadas. Para implementar isso, é necessário que uma tabela de símbolos tenha um ponteiro para outra tabela, que seria a sua tabela *pai*, representando um escopo dentro de outro (tabelas aninhadas). Deste modo, ao realizar uma busca em uma tabela de símbolos por um identificador, a busca é feita da tabela mais interna até a tabela mais externa, assegurando o escopo de cada variável.

Para implementar as tabelas aninhadas resolvemos seguir um caminho mais simples inicialmente, onde usamos uma única tabela de símbolos global, onde cada

símbolo tem um atributo que identifica seu escopo, e a tabela tem um atributo que define o escopo atual. O escopo atual da tabela é incrementado ao entrar em um escopo e decrementado ao sair, e as buscas são feitas em toda a tabela, procurando apenas por símbolos de escopo menor ou igual ao escopo atual. Apesar de não utilizar literalmente o conceito de tabelas aninhadas, esta implementação simula corretamente o conceito de escopo, assegurando o isolamento em declarações em escopos diferentes. Contudo, percebemos que seria difícil realizar algumas implementações mais complexas, como funções aninhadas e outras estruturas.

Com isso, optamos por uma implementação mais fiel à teoria, onde simplificamos a nossa *struct* que representava as entradas da tabela, e alteramos a maneira como a tabela de símbolos funcionava. A *struct* ficou com o seguinte formato:

```c
// Estrutura para um símbolo na tabela
typedef struct Symbol {
    char *name;
    char *type_string;      // String original do tipo (para compatibilidade)
    SymbolType type;        // Tipo enumerado para análise semântica
    int offset;
    int array_size;         // Para arrays - tamanho
    struct Symbol *next;
} Symbol;
```

A nova, e definitiva, estrutura guarda apenas o nome, o tipo do símbolo, em forma de texto e também usando *enum*, o tamanho do array(caso o símbolo seja um array) e o offset, que está relacionado com o endereço relativo do símbolo, além de um ponteiro para o próximo símbolo da tabela.

Em relação à implementação final da tabela de símbolos, dessa vez fizemos de modo que uma tabela sempre tenha um ponteiro para a sua tabela pai, o que é mais compatível com a teoria e faz mais sentido com o que havíamos feito até agora. Desse modo, conseguimos garantir que o escopo de cada variável estava garantido, de uma forma mais simples. Nesta estrutura também existe um ponteiro para o próximo escopo, o que não é obrigatório, e é usado apenas para imprimir a tabela de símbolos completa por motivos de debug.

```c
typedef struct SymbolTable {
    Symbol *table[HASH_TABLE_SIZE];     // Array de ponteiros para símbolos
    struct SymbolTable *parent;         // Ponteiro para tabela pai (escopo externo)
    struct SymbolTable *next_scope;     // Ponteiro para próxima tabela na ordem de criação
    int next_offset;                    // Próximo offset disponível
} SymbolTable;
```

Código de três endereços

O código de três endereços tem como estrutura básica x = y op z, sendo que ‘x’ equivale ao resultado, e ‘y’ e ‘z’ aos operandos. É possível entender o código de três endereços como uma representação linear de uma árvore de sintaxe ou de um DAG (Directed Acyclic Graph). De um modo geral, é constituído por endereços (nome, constantes, e temporários) e instruções (como atribuições, operações numéricas, e desvios).

É possível utilizar uma estrutura de dados para implementar a geração de código de três endereços. As mais utilizadas são as quádruplas - com campos para a operação realizada, dois campos para os operandos, e um para o resultado - e as triplas - que tem apenas campos para uma operação e dois para os argumentos, e referência outras posições da estrutura de dados quando precisa do resultado de uma operação anterior.

Como o enunciado do trabalho diz não ser necessário o uso dessas estruturas de dados, no compilador C-, a geração de código intermediário é realizada por meio da emissão direta de código de três endereços durante a análise sintática. Não há uma estrutura de dados dedicada para armazenar as instruções geradas; em vez disso, o código intermediário é impresso em tempo real, conforme as reduções das regras gramaticais são processadas.

Para representar resultados temporários das operações intermediárias, foi implementada a função *new_temp()*, responsável por criar identificadores de variáveis temporárias no formato *t0, t1, t2*, etc. Cada chamada a *new_temp()* gera uma nova variável temporária que é utilizada para compor as instruções de três endereços, principalmente nas produções relacionadas a expressões (expr, exprSoma, termo, etc.).

```
char* new_temp() {
    char* temp = (char*)malloc(10);
    sprintf(temp, "t%d", temp_count++);
    return temp;
}
```

Além disso, para o controle de fluxo em comandos condicionais e de repetição, como *if* e *while,* foi implementada a função *new_label()*, que gera rótulos no formato l0, l1, l2, etc. Esses rótulos são empregados para indicar pontos de desvio com as instruções goto, facilitando a tradução de construções de alto nível para o código intermediário. Por exemplo: no início de um bloco *while*, colocamos uma *label* para sinalizar que aquele é o início do bloco, e no fim do bloco colocamos um *goto* para a *label* criada.

```
char* new_label() {
    char* temp = (char*)malloc(10);
    sprintf(temp, "l%d", label_count++);
    return temp;
}
```

Para tratar corretamente as construções aninhadas de if e if-else, foi criada uma pilha de rótulos. Essa estrutura permite gerenciar a geração e o uso dos rótulos de maneira a evitar ambiguidades durante a tradução, garantindo que o par de rótulos associado a cada comando condicional seja recuperado na ordem correta (último a entrar, primeiro a sair). Esse mecanismo assegura que, durante a geração do código intermediário, a semântica do programa-fonte seja preservada, mesmo em casos com múltiplos blocos condicionais aninhados. Como resultado, o compilador imprime diretamente o código intermediário com variáveis temporárias e rótulos, mantendo um padrão de saída que pode ser posteriormente utilizado para etapas futuras, como otimização ou geração de código objeto.

Um dos desafios durante a implementação do gerador de código de três endereços, foi entender como gerar o código para uma função, desde a sua declaração até o momento em que esta é chamada. Para a chamada da função em si, o código de três endereços é gerado na regra *ativacao*, onde, inicialmente, cada argumento é avaliado e impresso como uma instrução *param*, preparando os parâmetros para a chamada. Em seguida, uma nova variável temporária é criada para armazenar o valor de retorno. Por fim, é emitida uma instrução para chamar a função, que atribui o seu resultado á uma variável. Esse processo assegura que a chamada da função seja representada de forma clara e modular no código

intermediário, mantendo a coerência com o modelo de execução do programa-fonte e facilitando sua posterior tradução para código de máquina.

O compilador implementado adotou uma estratégia de alocação com *padding*, utilizando um valor fixo de 10 como multiplicador para todos os cálculos de endereçamento de arrays multidimensionais. Esta decisão simplifica significativamente a geração de código de 3 endereços, eliminando a necessidade de estruturas de dados complexas para armazenar metadados específicos sobre dimensões e tipos de cada array. Ao usar um *stride* uniforme, o compilador pode aplicar uma fórmula única para todos os *arrays multidimensionais*, reduzindo a complexidade do analisador semântico.

Embora esta abordagem resulte em desperdício de memória e falta de verificação de tipos, ela serve como uma representação conceitual clara de como o processo de geração de código intermediário funciona, focando nos aspectos fundamentais da transformação de expressões de alto nível em operações elementares. O uso do placeholder "10" deve ser entendido como uma decisão que permite concentrar nos aspectos algorítmicos da geração de código, deixando como futuro aprimoramento a implementação de cálculos dinâmicos baseados nos tamanhos reais dos tipos e dimensões dos arrays.

Tratamento de erros

O tratamento de erros semânticos é realizado de forma integrada com a tabela de símbolos, usando verificações adicionais em diversas produções da gramática no Bison. A cada etapa relevante (declarações, expressões, comandos, chamadas de função), o analisador semântico realiza validações específicas para garantir que o código seja logicamente válido. Caso um erro seja detectado, uma mensagem descritiva é exibida no terminal, informando o tipo do erro, o contexto e a linha do código em que ocorreu.

A seguir, estão listados todos os erros semânticos tratados no programa:

- *Uso de identificador não declarado:* Verifica se variáveis, arrays ou funções foram devidamente declaradas antes de seu uso.

- *Chamada de função com identificador que não é função*: Impede que variáveis comuns sejam utilizadas como se fossem funções.

- *Incompatibilidade de tipos em atribuição*: Garante que o tipo do valor atribuído seja compatível com o tipo da variável de destino.

- *Operandos não numéricos em operações relacionais:* Operações como <, >, == só podem ser aplicadas a tipos numéricos (como int e float).

- *Operandos incompatíveis em operações aritméticas*: Soma, subtração e multiplicação devem envolver tipos compatíveis. Caso contrário, um erro é sinalizado.

- *Uso do operador unário - com tipo não numérico:* Apenas operandos numéricos podem ser negados com -.

- *Índice de array com tipo inválido:* O índice de um array deve obrigatoriamente ser um tipo inteiro (int ou char).

- *Uso de colchetes em identificadores que não são arrays*: Impede o uso de sintaxe de acesso a vetor em variáveis que não são arrays.

- *Redeclaração de símbolos no mesmo escopo:* A função de inserção na tabela de símbolos garante que identificadores não sejam declarados mais de uma vez no mesmo escopo.

- *Erro na declaração de parâmetros de função:* Parâmetros malformados, duplicados ou com tipo inválido são tratados no momento da declaração de funções.

- *Verificação implícita de tipos inválidos em expressões:* Caso uma expressão envolva identificadores com tipo TYPE_ERROR, outros erros subsequentes podem ser derivados e reportados.

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.

Na terceira etapa, foi possível ter uma boa visão do funcionamento das etapas finais de um compilador. Implementar as regras semânticas de cada regra gramatical foi um ponto crucial para o entendimento de como as informações transitam pela árvore sintática. Além disso, criar mecanismos lógicos para a geração de código de três endereços foi interessante para observarmos como o código é gerado para as diferentes estruturas definidas pelas regras da 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.

De um modo geral, o trabalho permitiu que nós tivéssemos uma visão ampla de como um compilador funciona, principalmente as etapas do front-end. Saber como o código fonte é transformado em código de máquina é importante para vermos como os conceitos de linguagens formais, estruturas de dados e análise de algoritmos se aplicam de maneira prática na construção de ferramentas fundamentais da computação. A vivência com a análise léxica, sintática e semântica nos ajudou a entender como erros são detectados, como estruturas são interpretadas e como representações intermediárias são criadas, permitindo ao compilador produzir um código eficiente e confiável. Além disso, o contato direto com ferramentas como Flex e Bison nos proporcionou experiência prática com geração automática de analisadores, consolidando os conteúdos vistos em aula e nos preparando para desafios mais complexos no desenvolvimento de linguagens, ferramentas de verificação e até mesmo na otimização de código.

Concluímos, portanto, que a implementação progressiva do compilador não apenas reforçou nosso entendimento teórico, como também nos proporcionou uma apreciação mais profunda do papel essencial que os compiladores desempenham no ecossistema de desenvolvimento de software.

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