UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Teoria da Computação e Compiladores - SCC0605

COMPILADOR DE PL-0

Artur Brenner Weber - 12675451
Gabriel Franceschi Libardi - 11760739
Guilherme Castanon Silva Pereira - 11801140
Gustavo Moura Scarenci de Carvalho Ferreira - 12547792
Matheus Henrique Dias Cirillo - 12547750

Docente responsável: Prof. Tiago A. S. Pardo

São Carlos
1º semestre/2024

SUMÁRIO

1 Introdução

No modelo de referência de um compilador, um **analisador sintático** (ou simplesmente um *parser*) lê o código-fonte de um programa como uma sequência de *tokens* e analisa se essa sequência respeita o conjunto de regras sintáticas da linguagem. O modelo de referência padrão de um compilador está esquematizado na figura [1]. No contexto da teoria das linguagens de programação, o analisador sintático verifica se uma sequência de *tokens* é uma palavra de uma linguagem livre de contexto, mas isso também tem uma grande variedade de outras aplicações como interpretação de expressões matemáticas e leitura de arquivos estruturados (como JSON).

A análise sintática é a segunda etapa do processo de compilação, a primeira sendo a análise léxica abordada no primeiro trabalho e dada neste como resolvida. A etapa sintática pode analisar as construções do código-fonte como sequências de *tokens* por diferentes algoritmos e estratégias. Neste trabalho, foi implementada a análise descendente preditiva e recursiva, com tratamento de erros pela estratégia conhecida como "modo pânico". A implementação do *parser* possibilita que as próximas fases do compilador, principalmente a análise semântica e a geração de código intermediário, continuem o processo de compilação.

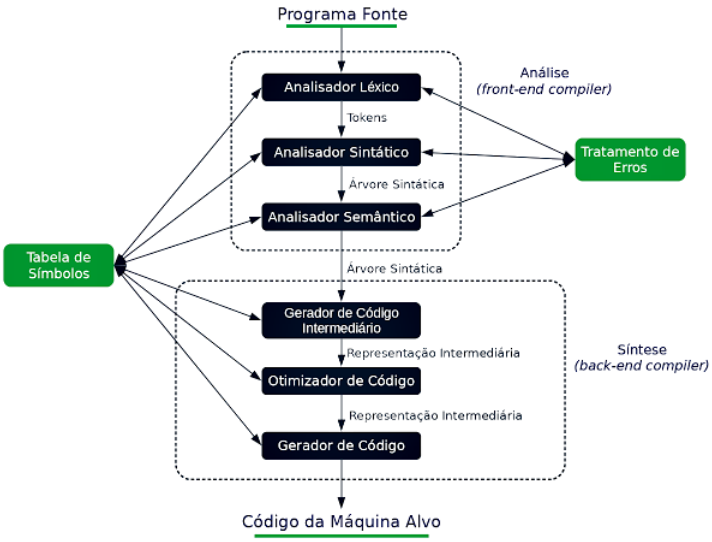

Figura 1: *Etapas de um compilador. [1]*

Uma linguagem de grande influência no estudo do funcionamento de compiladores é a PL/0, apresentada por Niklaus Wirth [2], sendo muito utilizada para o fim didático desde então. Nesse sentido, o presente projeto consiste na implementação de um analisador sintático para a linguagem PL/0, a qual é simplificada para diminuir a dificuldade de implementação do compilador. Esta possui apenas um tipo de dado (números inteiros) e as estruturas de controle básicas de linguagens de programação procedurais (WHILE, IF, PROCEDURE e CALL). Dessa forma, PL/0 é apenas uma linguagem de paradigma procedural com o mínimo de funcionalidade.

Para desenvolvimento do analisador, foi utilizada a gramática original da linguagem PL/0, obtida no livro [2]. A partir da documentação, geraram-se os grafos sintáticos, e estes foram mapeados para procedimentos na linguagem C. Sua função principal é receber um arquivo de texto com código em PL/0 e retornar se o programa foi aceito pela gramática, bem como fornecer mensagens *user-friendly* para erros sintáticos e léxicos identificados.

2 Desenvolvimento do Analisador Sintático

2.1. Fundamentos e Ideias Iniciais

Para a construção do analisador sintático, que consiste basicamente na programação das regras da gramática de PL/0 na linguagem C, inicialmente foram produzidos os grafos sintáticos da gramática que serviram como referência visual para a programação, além de ilustração neste relatório. O algoritmo adotado para a realização desse compilador foi o de análise descendente recursiva preditiva, conforme as aulas do [3] Prof. Dr. Thiago Pardo, docente da disciplina de Teoria da Computação e Compiladores, e o material [4] disponibilizado por Ian D. Allen. Isso foi possível porque a gramática da qual se partiu para a implementação é do tipo LL(1), em que não há recursão à esquerda e na qual os conjuntos primeiros de produções diferentes são disjuntos.

A fins de exemplo, o grafo sintático da regra de formação de `mais_termos`, uma produção um pouco mais complicada para implementação, e `declaracao`, uma bastante simples, são mostrados nas figuras 2 e 3, com suas implementações nos códigos 1 e 2. Ademais, todas as regras sintáticas de PL/0 foram convertidas em grafos, e esses podem ser vistos em sua plenitude na seção Apêndice, assim como a sua efetiva implementação e o compilador todo pode ser visto no repositório do GitHub [1].

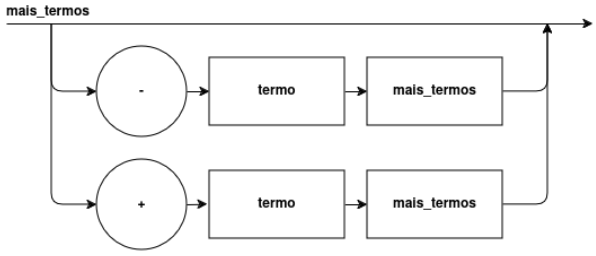

Figura 2: *Grafo sintático da regra de formação mais_termos*

```
1  void mais_termos(){
2      SyncTokens parent_tokens = {7, (char*[]){SEMICOLON, KW_END, PERIOD, RIGHTPAR, RELOP,
           KW_THEN, KW_DO}};
3      SyncTokens immediate_tokens = {0, NULL};
4
5      if(!strcmp(current_token->type, PLUS)){
6          immediate_tokens = (SyncTokens){3, (char*[]){IDENT, INT, LEFTPAR}};
7          MATCH(FIELD_TYPE, PLUS, "Expected '+'");
8
9          termo();
10
11         mais_termos();
12     } else if(!strcmp(current_token->type, MINUS)){
13         immediate_tokens = (SyncTokens){3, (char*[]){IDENT, INT, LEFTPAR}};
14         MATCH(FIELD_TYPE, MINUS, "Expected '-'");
15
16         termo();
17         mais_termos();
18     }
19     //epsilon
```

[1] https://github.com/GuScarenci/compilador-pl0

```
20 }
```

Código 1: *Código da regra* `mais_termos( )`

Figura 3: *Grafo sintático da regra de formação declaracao*

```
1 void declaracao(){
2     constante();
3     variavel();
4     procedimento();
5 }
```

Código 2: *Código da regra* `declaracao( )`

Além disso, para a análise sintática bem feita, é necessário a identificação de erros, e o reporte desses de maneira legível. Para esse fim, foi adotado a estratégia do modo pânico, na qual se consomem os *tokens* do programa até que se encontre um *token* de sincronização, em que esses tokens de sincronização são inicialmente dados pelos **seguidores** da regra atual na gramática. Tais seguidores, assim como os **primeiros**, importantes no contexto de análise sintática para a determinação de qual regra o compilador deve escolher para prosseguir, e também para determinar seguidores de outras produções, são mostrados na tabela 1.

Tabela 1: *Definições de Conjuntos de Caracteres*

Regras	Primeiros	Seguidores
`programa`	CONST | VAR | PROCEDURE | ident | CALL | BEGIN | IF | WHILE | .	λ
`bloco`	CONST | VAR | PROCEDURE | ident | CALL | BEGIN | IF | WHILE | λ	. | ;
`declaracao`	CONST | VAR | PROCEDURE | λ	ident | CALL | BEGIN | IF | WHILE | . | ;
`constante`	CONST | λ	VAR | PROCEDURE | ident | CALL | BEGIN | IF | WHILE | . | ;
`mais_const`	, | λ	;
`variavel`	VAR | λ	PROCEDURE | ident | CALL | BEGIN | IF | WHILE | . | ;
`mais_var`	, | λ	;
`procedimento`	PROCEDURE | λ	ident | CALL | BEGIN | IF | WHILE | . | ;
`comando`	ident | CALL | BEGIN | IF | WHILE | λ	; | END | .

Regras	Primeiros	Seguidores
`mais_cmd`	; | λ	END
`expressao`	- | + | ident | numero | (	; | END | . |) | = | <> | < | <= | > | >= | THEN | DO
`operador_unario`	- | + | λ	ident | numero | (
`termo`	ident | numero | (	- | + | ; | END | . |) | = | <> | < | <= | > | >= | THEN | DO
`mais_termos`	- | + | λ	; | END | . |) | = | <> | < | <= | > | >= | THEN | DO
`fator`	ident | numero | (	* | / | - | + | ; | END | . |) | = | <> | < | <= | > | >= | THEN | DO
`mais_fatores`	* | / | λ	- | + | END |) | = | <> | < | <= | > | >= | THEN | DO | . | ;
`condicao`	ODD | - | + | ident | numero | (	THEN | DO
`relacional`	= | <> | < | <= | > | >=	- | + | ident | numero | (

2.2. Código

2.2.1 Instruções de Execução

Para rodar o código, basta digitar no terminal:

```
$ make run ARGS="<source_file> <output_file>"
```

em que <source_file> é o caminho para o código fonte (um arquivo de texto) e <output_file> é a saída pedida com mensagens de sucesso ou erro de compilação, com os erros de compilação de uma maneira amigável e informativa caso haja erro no programa. A execução do programa sobrescreve todo o conteúdo de <output_file> com a saída do analisador sintático. A diretiva 'run' também compila automaticamente o código se o executável não foi encontrado. Para uma visão geral do funcionamento e implementação do analisador léxico, sugere-se testá-lo com o programa no arquivo `tests/program.pl0`, executando o comando:

```
$ make run ARGS="tests/program.pl0 stdout"
```

Para a melhor saída de erros, é recomendado que a saída seja o terminal (`stdout`) ou que ao jogar para a saída para um arquivo de texto, se faça um comando `cat` do arquivo para o terminal. Visto que a saída do programa faz destaque de erros e *warnings* com cores e isso é feito por meio de códigos nas saídas, que aparecerão escritos em um arquivo de texto.

2.2.2 Alterações no Analisador Léxico

Foram implementadas duas mudanças significativas no analisador léxico para melhorar a funcionalidade e a usabilidade do compilador. As mudanças incluem a adição de uma função para salvar a posição do *token* no arquivo de origem e a modificação do comportamento do macro `OPENFILE()` para permitir a utilização de `stdout` como destino de saída.

Salvando a posição do *token* no arquivo: Isso é fundamental para permitir uma saída de erro mais detalhada e útil para o usuário. Anteriormente, as mensagens de erro poderiam ser menos informativas, dificultando a identificação precisa do local onde o erro ocorreu.

Com a nova implementação, cada *token* agora armazena sua posição exata no arquivo, incluindo a linha e a coluna. Isso permite que o compilador forneça mensagens de erro mais claras e específicas, indicando exatamente onde no código-fonte o erro foi encontrado. Por exemplo, uma mensagem de erro fica com a aparência mostrada na figura 4, facilitando o trabalho do programador.

```
./tests/w3.pl0:1:11: error: Use '=' for CONST assignment
        1 | CONST john := 57;
./tests/w3.pl0:3:0: warning: Unexpected token after program ending.
        3 | CONST john = 5;

Compilation failed with 1 errors and 1 warnings.
```

Figura 4: *Erros e avisos detalhados*

Modificação na função `OPENFILE()` para permitir `stdout`: Anteriormente, esta função estava restrita a abrir arquivos apenas pelo seu *path*. No entanto, para aumentar a flexibilidade e facilitar o uso durante o desenvolvimento e a depuração, a função `OPENFILE()` foi modificada para permitir que `stdout` (passada como string `"stdout"`) seja usada como destino de saída.

Com essa modificação, os desenvolvedores podem agora redirecionar a saída do compilador diretamente para o terminal, o que é particularmente útil durante a fase de desenvolvimento e testes. Isso permite uma visualização imediata das mensagens de erro e outras saídas, sem a necessidade de abrir arquivos adicionais. Para utilizar essa funcionalidade, o usuário pode simplesmente especificar `stdout` como parâmetro ao chamar a função `OPENFILE`, direcionando assim todas as saídas do compilador para o console.

Essas melhorias no analisador léxico, ao fornecerem uma localização precisa dos *tokens* no arquivo e ao permitirem uma saída mais flexível, são passos importantes para tornar o processo de compilação mais eficiente e amigável para os desenvolvedores.

2.2.3 Modo pânico

O método `match_function()` e o macro `MATCH` trabalham juntos para gerenciar a análise sintática, especificamente para lidar com a verificação de correspondência dos *tokens* esperados durante a análise e a sincronização em caso de erros. [5] [6]

```
1  #define MATCH(field_type, str, error_message)\
2      do {\
3          int32_t result = match_function(field_type, str, error_message, immediate_tokens,
               parent_tokens);\
4          if(result == PARENT) return;\
5          if(result == SYNC_ERROR) exit(-1);\
6      } while(false)
```

Código 3: *Código do macro `MATCH()`*

O macro `MATCH`, como visto no código 3, é uma forma conveniente de invocar a função `match_function()` com um conjunto específico de parâmetros, incluindo o tipo de campo a ser comparado, a *string* de comparação e uma mensagem de erro. O *macro* encapsula a chamada à função `match_function()` em um *loop do-while* que é uma técnica comum para evitar a redefinição da variável `result`.

Dentro de `match_function()`, a função inicialmente verifica se o *token* atual corresponde ao *token* esperado. Se a correspondência for bem-sucedida, a função atualiza o *current_token* para

o próximo *token* no fluxo de *tokens* e retorna `SUCCESS`. Não existindo correspondência, a função imprime uma mensagem de erro apropriada e tenta sincronizar para um próximo *token*.

A grosso modo, a função escolhe um *token* de sincronização para decidir o quanto voltar na árvore de recursão. Não encontrando nenhuma opção válida para sincronizar, a função gera um erro catastrófico, indicando que não há *tokens* de sincronização disponíveis e, portanto, a análise não pode continuar de forma coerente.

Para alcançar isso a função entra em um *loop* que percorre os *tokens* restantes até encontrar um que faça parte de um dos seguintes conjuntos: os primeiros *tokens* do *token* que chamou `MATCH` (chamados de `immediate_tokens`) ou os tokens seguidores da regra onde `MATCH` foi chamado (chamados de `parent_tokens`). Se encontrar um *token* no conjunto `immediate_tokens`, a função retorna `IMMEDIATE`, permitindo que a análise continue a partir do ponto onde parou. Se encontrar um token no conjunto `parent_tokens`, a função retorna `PARENT`, indicando que a análise deve voltar para um nível superior da árvore de recursão. Se nenhum desses tokens for encontrado e o fluxo de tokens acabar, a função retorna `SYNC_ERROR`, levando a um erro catastrófico.

Para tratar de erros léxicos, a função verifica se o próximo *token* encontrado é válido ou não. Caso não seja, ela gera um erro, e continua procurando o próximo ponto de sincronização.

Em resumo, a função `match_function()` e a *macro* `MATCH` trabalham juntos para verificar a correspondência de *tokens* esperados, tratar erros lexicais e realizar a sincronização apropriada durante a análise sintática, decidindo dinamicamente se deve tentar sincronizar imediatamente ou voltar ao nível superior da árvore de recursão.

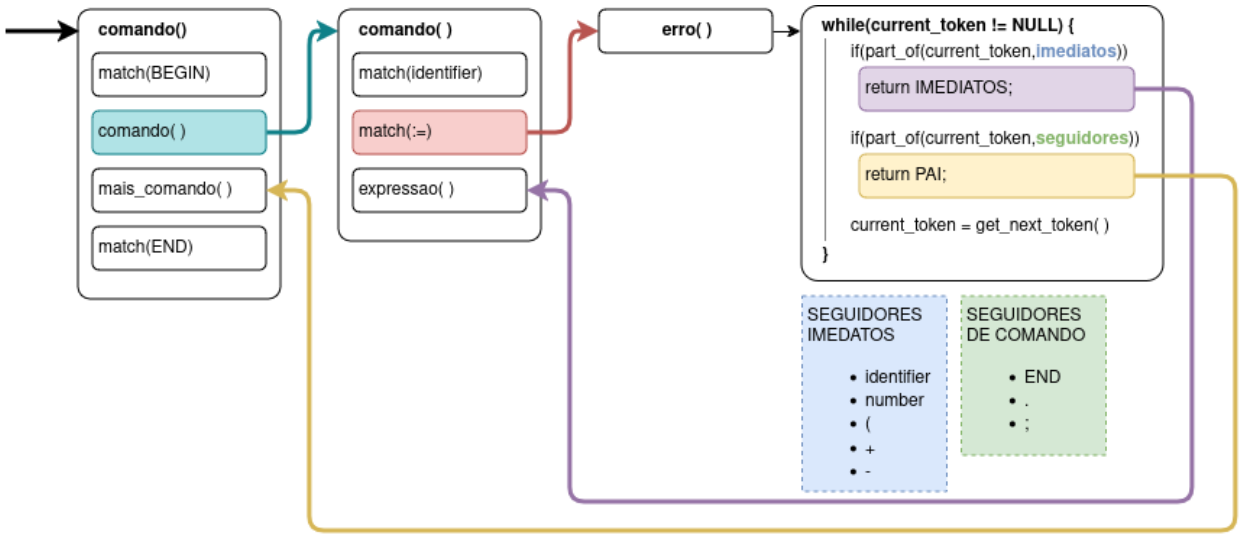

Figura 5: *Possível fluxo de erro do código.*

2.3. Possíveis Melhorias

2.3.1 Identificação de tipo de *tokens* mal formados

Uma potencial melhoria no analisador léxico envolve o armazenamento de *tokens* mal formados com o tipo de *token* que eles deveriam ser. Esta abordagem oferece uma maneira sofisticada de lidar com erros lexicais, permitindo que a análise sintática continue normalmente e reconheça o problema apenas como um erro léxico, evitando potenciais avalanches de erro.

Essa estratégia traz diversos benefícios. Primeiramente, permite que o analisador sintático continue a processar a entrada, tentando aplicar suas regras com base no tipo esperado do *token*. Isso pode ser especialmente útil em contextos onde um *token* mal formado ocorre esporadicamente e a análise sintática pode progredir apesar do erro. Em outras palavras, o analisador sintático poderia "dar *match*" com a regra correspondente ao tipo esperado do *token*, tratando o problema como exclusivamente léxico.

Além disso, ao permitir que o analisador sintático continue a analisar a entrada, mesmo na presença de *tokens* mal formados, o processo de análise torna-se mais resiliente. Isso significa que a análise sintática não é interrompida prematuramente por erros lexicais, possibilitando a detecção de mais erros em uma única execução. Isso facilita a depuração, pois os desenvolvedores podem focar na correção de erros lexicais sem serem distraídos por uma avalanche de erros sintáticos decorrentes de *tokens* mal formados.

Para implementar essa melhoria, seria necessário modificar o analisador léxico para que ele marque os *tokens* mal formados com o tipo esperado, ao invés de simplesmente sinalizar um erro. O analisador sintático também precisaria ser ajustado para reconhecer esses *tokens* especiais e tratar os erros como problemas lexicais específicos, permitindo a continuidade da análise.

2.3.2 Uso dos primeiros de maneira mais sofisticada para evitar avalanche de erros

Outra potencial melhoria no analisador sintático envolve o uso das regras de *primeiro* para identificar as regras que potencialmente chamaram o *token* atual e realizar uma busca reversa na árvore de recursão. Isso pode ser visualizado na Figura 6, onde a busca reversa ajuda a sincronizar a análise e evitar o problema de avalanche de erros.

Atualmente, ao encontrar um *token* inesperado, o analisador pode entrar em um estado de avalanche, gerando múltiplos erros sintáticos devido a um único erro léxico ou sintático. Para mitigar esse problema, podemos utilizar as regras de *primeiro* para determinar quais regras poderiam ter gerado o próximo *token*.

Na Figura 6, o fluxo de análise sintática começa com a função `bloco()`, que chama `declaracao()`, que por sua vez chama `constante()`. Dentro de `constante()`, diferentes *tokens* são esperados em sequência, incluindo `CONST`, `ident`, `=`, `numero` e `;`.

Se ocorrer um erro durante a correspondência de um desses *tokens*, o analisador entra na função `erro()` para tratar a situação. Ele procura nos próximos *tokens* qualquer uma da lista de seguidores de `<constante>`, e ao encontrar um deles, calcula quais regras esse *token* de sincronização é o primeiro, e tenta encontrar pela árvore de recursão qual regra pode ser a geradora.

Esse processo de busca reversa na árvore de recursão permite que o analisador identifique a posição provável onde o erro ocorreu e sincronize a análise no nível apropriado, minimizando os efeitos de avalanche. Implementar essa melhoria requer ajustes no analisador sintático para manter e consultar as regras de *primeiro* e seguidores durante a análise e a manipulação de erros. Isso pode aumentar a complexidade do analisador, mas traz um benefício significativo ao reduzir a propagação de erros e fornecer mensagens de erro mais precisas e úteis.

Em resumo, utilizando as regras de *primeiro* para fazer uma busca reversa na árvore de recursão, como ilustrado na Figura 6, o analisador sintático pode melhorar a resiliência e precisão na detecção e tratamento de erros, evitando o problema de avalanche e facilitando a correção de erros pelos desenvolvedores.

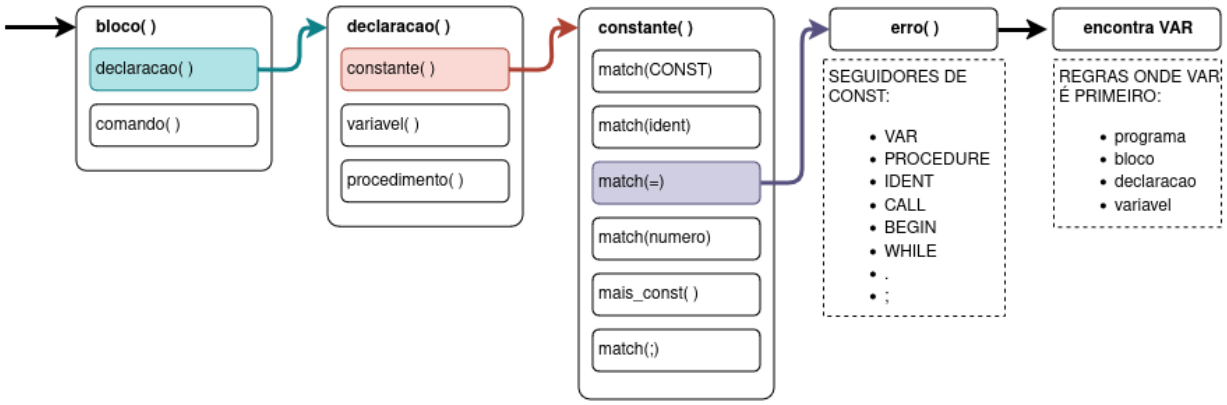

Figura 6: *Busca reversa na árvore de recursão utilizando regras de primeiro para melhorar a sincronização do analisador sintático.*

2.4. Escolhas

Durante o desenvolvimento do analisador sintático, fizemos algumas escolhas específicas de projeto para melhorar a precisão da análise e a capacidade de gerar avisos úteis.

2.4.1 Uso do ponto-e-vírgula ";" como token de sincronização do primeiro ponto-e-vírgula ";" na regra de procedimento

Optamos por remover o ponto-e-vírgula dos conjuntos de *tokens* imediatos do primeiro ponto-e-vírgula na regra de `<procedimento>`.

```
<procedimento> ::= PROCEDURE ident ; <bloco> ; < procedimento>
```

Isso porque, apesar de permitido, é bem incomum o `<bloco>` que vem de imediato ser λ, situação que coloca o ponto-e-vírgula como simbolo adjacente do primeiro ponto-e-vírgula. Desse modo, a inclusão de ponto-e-vírgula como *token* de sincronização sempre, já que esses são definidos estaticamente para cada caso, para cobrir apenas um caso específico e raro, ao nosso ver, estava gerando mais problemas do que resolvendo. A remoção desse *token*, baseado nos testes que fizemos, previne erros sintáticos desnecessários (avalanches de erros).

2.4.2 Final do arquivo no ponto final

Outra escolha importante foi considerar arquivos com programas válidos que contenham *tokens* após o ponto final (`.`). Em vez de tratar qualquer *token* após o ponto final como um erro, decidimos permitir que o analisador aceite esses *tokens* adicionais, mas marque o código como compilado com avisos (*warnings*). Isso significa que um código que termina corretamente com um ponto, mas possui *tokens* extras depois, será compilado com sucesso, mas com avisos para informar o desenvolvedor sobre a presença desses *tokens* supérfluos - e o código de montagem gerado seria apenas a tradução do programa válido que vem antes do ponto, se a compilação seguisse até a conclusãon.

Essas escolhas de projeto foram feitas visando tornar o analisador sintático mais robusto e informativo, permitindo uma análise mais precisa e fornecendo *feedback* útil para melhorar a qualidade do código.

2.5. Organização do Projeto

Para organização física dos arquivos do projeto, optou-se pela estrutura de diretórios mostrada na figura 7.

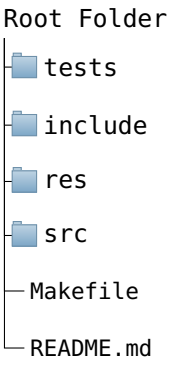

Figura 7: *Organização do diretório de implementação*

Após reflexão da equipe considerando, modularização e tamanho de código, a estrutura do diretório é formada por três principais pastas, além do *Makefile*,usado para comandos de compilação e execução, e *README*,com detalhes e informações do projeto.

A primeira, *include*, contém os arquivos header *.h* dos códigos C: `error_handler`, `IO_utils`, `lexer`, `rdp`, `state_machine` e `str_utils`, além das estruturas de dados e *macros* utilizadas.

A pasta *src* contém todos os arquivos fonte do programa em C, enquanto *res* armazena os arquivos referentes as etapas léxicas. As funções mais relacionadas a etapa de anaĺise sintática e correção de erros estão em `rdp.c` e `erro_handler.c`.

Além disso, a pasta *tests* contém vários códigos desenvolvidos pelo grupo em PL/0 para testar compilações feitas com sucesso e erros, sendo os arquivos corretos os que o nome começa com a letra r (*right*) e os arquivos errados os que o nome começa com a letra w (*wrong*).

2.6. Ambiente de Teste e Desenvolvimento

Para desenvolvimento do código, utilizou-se o programa *Visual Studio Code* nos sistemas operacionais Windows 11 (utilizando WSL) e Fedora 39. Para a compilação do programa em cada SO, utilizaram-se os compiladores GCC 11.4.0 e GCC 13.3.1, respectivamente.

3 Conclusão

Para concluir o projeto de desenvolvimento do analisador sintático para a linguagem PL/0, a equipe refletiu sobre os objetivos e resultados alcançados ao longo do processo. O grupo implementou com sucesso um analisador sintático funcional, que conta com um analisador léxico, feito em etapa anterior, capaz de processar código-fonte escrito em PL/0 e segmentá-lo em *tokens* adequados para a análise sintática. Essa etapa do compilador verifica a ordem correta dos *tokens* conforme a gramática da linguagem em questão. Focando na análise sintática, essa proposta envolveu o estudo da gramática e a confecção de grafos sintáticos até finalmente a união desses conhecimentos para a codificação das regras do analisador em C, assim como seu tratamento de erros.

A implementação em C foi desafiadora e educacional, proporcionando uma experiência prática com técnicas fundamentais de compilação. Utilizou-se a gramática formal da linguagem PL/0 para a análise das regras de formação e codificação, assim como para a identificação de primeiros e seguidores dos símbolos na gramática usados na implementação do tratamento de erros por modo pânico, possibilitando o analisador sintático reconhecer corretamente as regras de formação da linguagem PL/0 e seus elementos sintáticos da linguagem, como identificadores, literais e operadores, além de identificar erros e se recuperar dos mesmos para dar mensagens de erro compreensíveis ao usuário.

Além disso, por meio deste trabalho, foi possível consolidar a importância de uma análise léxica bem feita, já que ela foi fundamental para a realização da análise sintática. Para a etapa sintática, a função `get_next_token()`, implementada para a análise léxica em um trabalho anterior, foi o ponto de partida para a construção da etapa de análise sintática.

Em suma, o projeto alcançou seus objetivos didáticos, proporcionando um entendimento profundo das etapas iniciais do processo de compilação e das técnicas envolvidas na construção de analisadores sintáticos. O produto final é uma ferramenta útil, que pode servir como base para estudos adicionais e projetos mais complexos no campo da compilação e análise de linguagens de programação.

Referências Bibliográficas

[1] J. D. Marangon, “Compiladores para humanos,” 2017. Recuperado de https://johnidm.gitbooks.io/compiladores-para-humanos/content/.

[2] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice Hall, 1976.

[3] T. Pardo, “Aula 15 - asd preditiva recursiva.” https://edisciplinas.usp.br/pluginfile.php/8350380/mod_resource/content/0/Aula%2015%20-%20ASD%20preditiva%20recursiva.pdf, 2024. Acessado em 21 de junho de 2024.

[4] I. D. Allen, “Recursive descent parsing.” https://teaching.idallen.com/cst8152/98w/recursive_decent_parsing.html, 1998. Acessado em 21 de junho de 2024.

[5] T. Pardo, “Aula 16 - tratamento de erros sintáticos, asd preditiva não recursiva.” https://edisciplinas.usp.br/pluginfile.php/8353398/mod_resource/content/0/Aula%2016%20-%20Tratamento%20de%20erros%20sint%C3%A1ticos%2C%20ASD%20preditiva%20n%C3%A3o%20recursiva.pdf, 2024. Acessado em 21 de junho de 2024.

[6] I. D. Allen, “Panic mode error recovery.” https://teaching.idallen.com/cst8152/98w/panic_mode.html, 1998. Acessado em 21 de junho de 2024.

A Apêndice

Figura 8: *Grafo da regra programa.*

Figura 9: *Grafo da regra bloco.*

Figura 10: *Grafo da regra declaracao.*

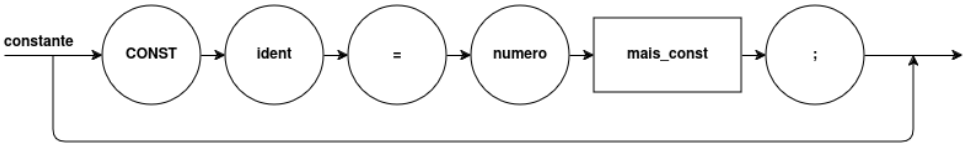

Figura 11: *Grafo da regra constante.*

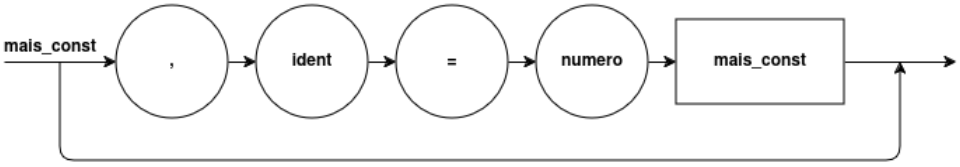

Figura 12: *Grafo da regra mais_const.*

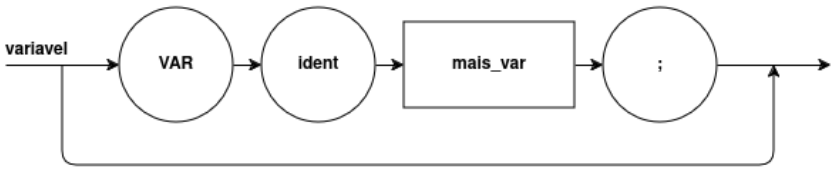

Figura 13: *Grafo da regra variavel.*

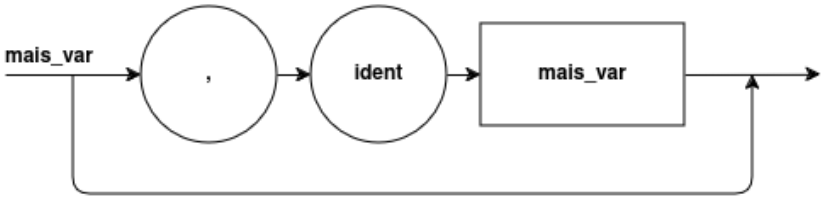

Figura 14: *Grafo da regra mais_var.*

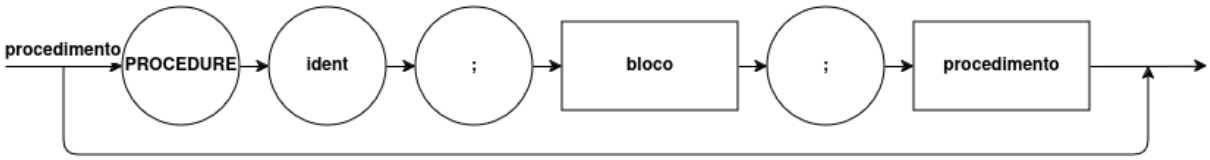

Figura 15: *Grafo da regra procedimento.*

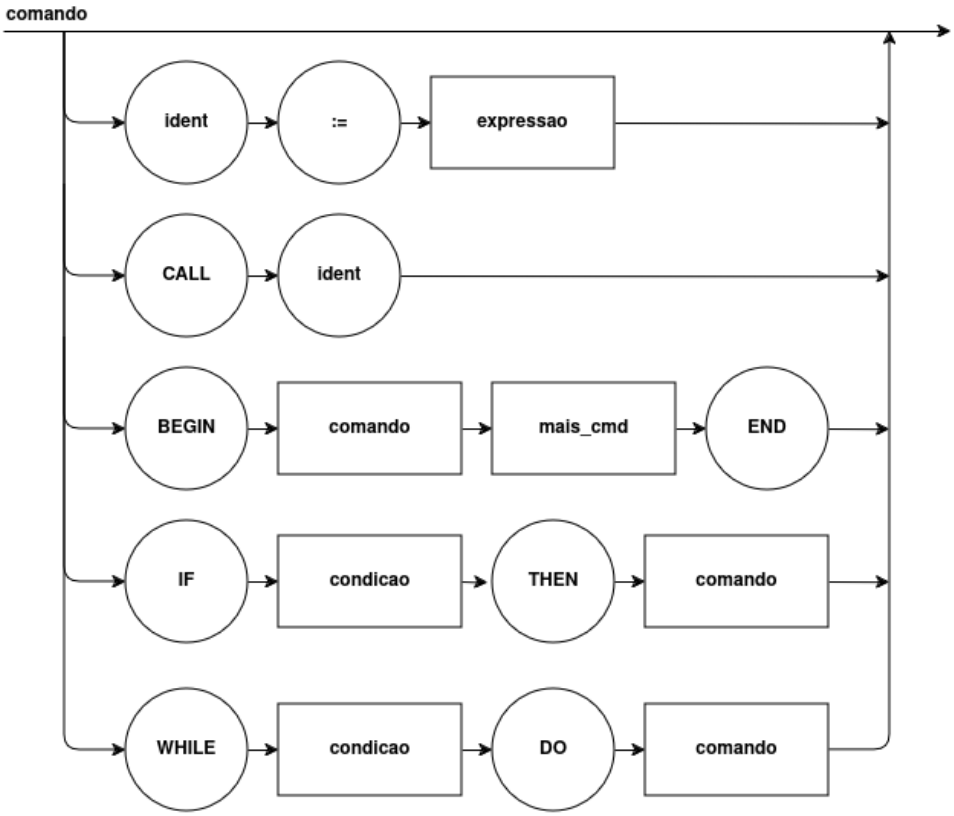

Figura 16: *Grafo da regra comando.*

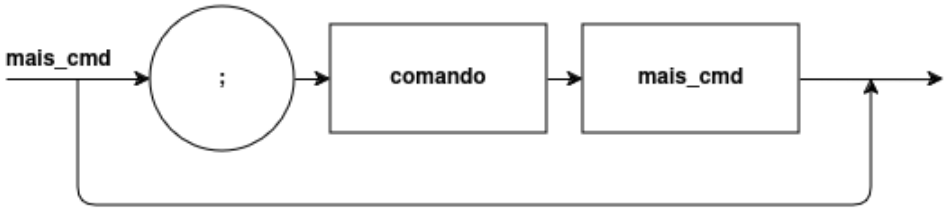

Figura 17: *Grafo da regra mais_cmd.*

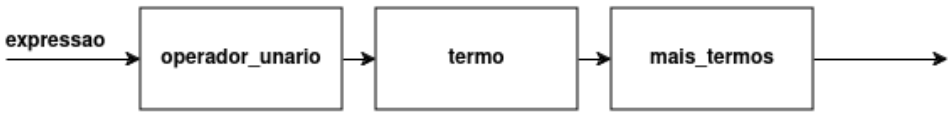

Figura 18: *Grafo da regra expressao.*

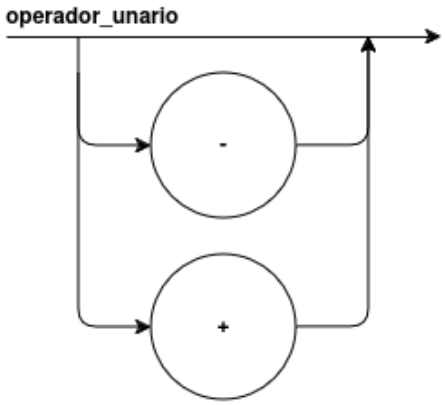

Figura 19: *Grafo da regra operador_unario.*

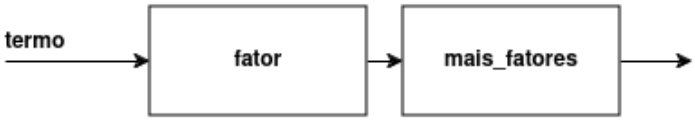

Figura 20: *Grafo da regra termo.*

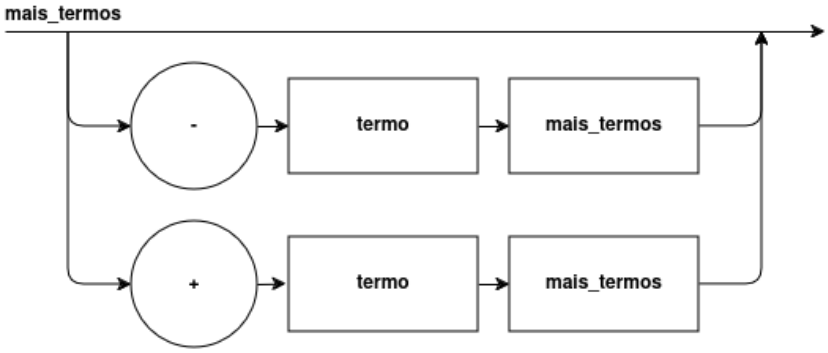

Figura 21: *Grafo da regra mais_termos.*

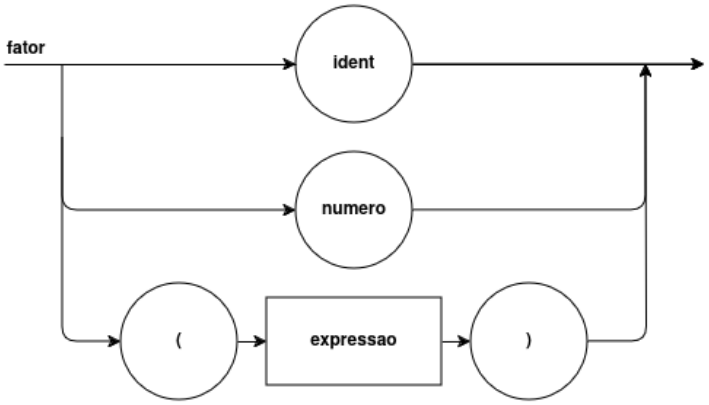

Figura 22: *Grafo da regra fator.*

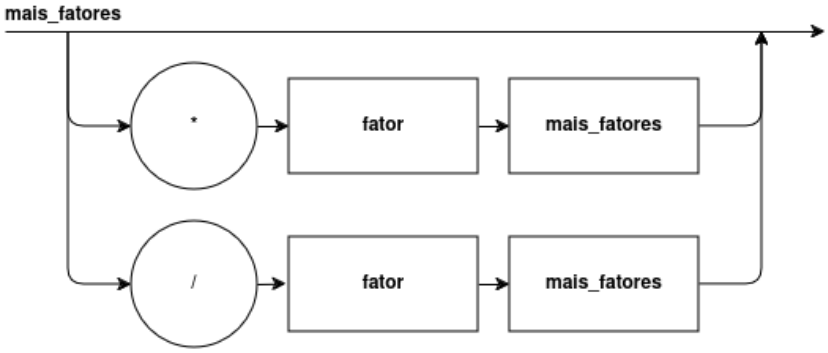

Figura 23: *Grafo da regra mais_fatores.*

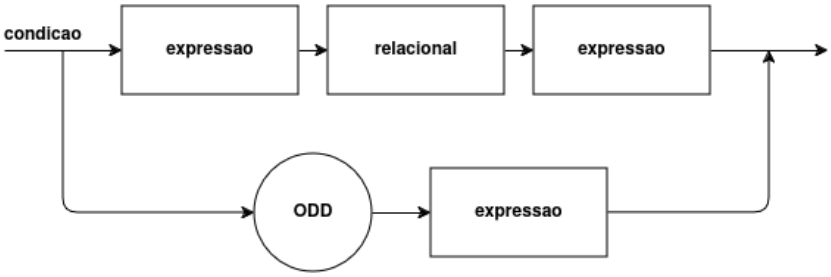

Figura 24: *Grafo da regra condicao.*

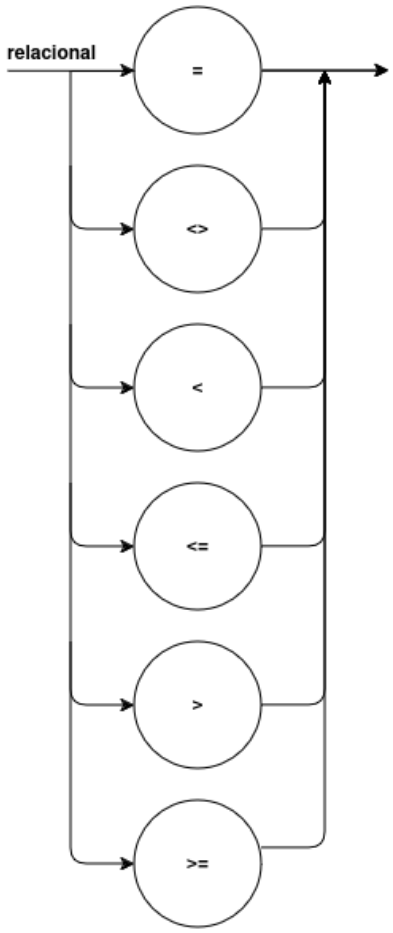

Figura 25: *Grafo da regra relacional.*