INSTITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA
DE MINAS GERAIS - CAMPUS BAMBUÍ
BACHARELADO ENGENHARIA DA COMPUTAÇÃO

Carlos Eduardo de Sousa
Kevenn Henrique de Paula Silva
Marco Antônio da Silva

DESENVOLVIMENTO DE UM COMPILADORPARA A LINGUAGEM MANCO UTILIZANDO ANTLR4

Bambuí-MG
2023

CARLOS EDUARDO DE SOUSA
KEVENN HENRIQUE DE PAULA SILVA
MARCO ANTÔNIO DA SILVA

DESENVOLVIMENTO DE UM COMPILADORPARA A LINGUAGEM MANCO UTILIZANDO ANTLR4

Trabalho apresentado à disciplina de Compiladores, do Bacharelado em Engenharia de Computação, como requisito parcial para obtenção dos créditos. Professor: Cláudio Ribeiro de Sousa

Bambuí-MG
2023

SUMÁRIO

1 INTRODUÇÃO

A construção do front-end de um compilador é uma etapa crucial no processo de desenvolvimento de compiladores. O front end é responsável por analisar o código fonte e transformá-lo em uma representação intermediária, como uma árvore sintática abstrata. Essa representação intermediária é então utilizada pelo back-end do compilador para gerar o código objeto ou executável.

A construção do front-end envolve a definição da gramática da linguagem de programação, a implementação do analisador léxico e do analisador sintático, além de possíveis otimizações no processo de análise. A qualidade e eficiência do front-end afetam diretamente a qualidade e performance do compilador como um todo.

O presente trabalho visa documentar a implementação do front-end de um compilador para a linguagem de programação denominada MANCO, que se caracteriza por ser imperativa e apresentar uma sintaxe simples e intuitiva. A implementação do front-end da linguagem MANCO foi realizada utilizando a ferramenta ANTLR, que é uma das ferramentas mais utilizadas para a construção de compiladores. ANTLR é uma ferramenta de reconhecimento de linguagens de programação que permite a geração automática de analisadores léxicos e sintáticos a partir de uma gramática.

Nesta documentação, serão descritos os passos necessários para a implementação do front-end da linguagem MANCO, desde a definição da gramática da linguagem até a geração dos analisadores léxico e sintático com a ferramenta ANTLR. Além disso, serão apresentados exemplos de códigos fonte na linguagem MANCO.

2 DESCRIÇÃO DO COMPILADOR

Como dito anteriormente na introdução, foi utilizado a ferramenta ANTLR4 na construção deste compilador. O ANTLR4 (Another Tool for Language Recognition) é uma ferramenta de geração de analisadores léxicos e sintáticos, amplamente utilizada para desenvolver compiladores, interpretadores e outras ferramentas de análise de linguagem. Ele permite que você escreva gramáticas de linguagem que descrevem a estrutura e o comportamento de uma linguagem de programação e, a partir dessas gramáticas, gera automaticamente código em Java ou em outra linguagem de programação para implementar o analisador léxico e sintático.

Aqui estão algumas razões pelas quais utilizou-se a ferramenta ANTLR4 na construção do compilador:

1. Facilidade de uso: O ANTLR4 oferece uma sintaxe de gramática clara e fácil de usar que permite descrever a estrutura e o comportamento de uma linguagem de programação de forma eficiente.
2. Gerador de código: O ANTLR4 gera automaticamente código em Java ou em outra linguagem de programação para implementar o analisador léxico e sintático.
3. Flexibilidade: O ANTLR4 oferece suporte a uma ampla gama de linguagens, incluindo Java, C#, Python, JavaScript e muitas outras.
4. Performance: O ANTLR4 oferece melhorias significativas na performance em comparação com as versões anteriores, o que permite análise de linguagem mais rápida e eficiente.
5. Documentação e recursos: O ANTLR4 tem uma ampla documentação disponível, bem como uma série de recursos adicionais, incluindo exemplos, tutoriais e bibliotecas de terceiros.

Em resumo, o ANTLR4 foi uma ferramenta poderosa e flexível que nos ajudou a simplificar e acelerar o desenvolvimento do compilador para a linguagem MANCO.

3 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

Os requisitos funcionais são aqueles que descrevem as funcionalidades e as características que o compilador possui. Eles especificam o que o sistema deve fazer.

Já os requisitos não funcionais são aqueles que descrevem as restrições, limitações e qualidades do compilador, como desempenho, escalabilidade, segurança, compatibilidade, etc. Eles especificam como o sistema deve ser.

3.1 Requisitos Funcionais

Os requisitos funcionais do front-end do compilador construído incluem:

RF 01 - Análise léxica: capacidade de reconhecer e classificar os elementos básicos da linguagem, como identificadores, palavras-chave, números, símbolos e constantes.

RF 02 - Análise sintática: capacidade de verificar a validade da estrutura do programa, identificando erros como falta de parênteses ou colchetes desbalanceados.

RF 03 - Verificação semântica: capacidade de verificar a validade dos significados dos elementos do programa, incluindo tipos de variáveis, compatibilidade de tipos e outras regras semânticas da linguagem.

RF 04 - Geração de árvore de análise sintática (AST): capacidade de construir uma representação estruturada do programa, como uma árvore de análise sintática, que pode ser usada para otimização e geração de código intermediário.

RF 05 - Geração de tabelas de símbolos: capacidade de construir e gerenciar uma tabela de símbolos, que contém informações sobre identificadores e variáveis, incluindo nome, tipo, escopo e outras informações relevantes.

RF 06 - Gerador de mensagens de erro: capacidade de gerar mensagens de erro claras e precisas para informar o desenvolvedor sobre problemas no código.

3.2 Requisitos Não Funcionais

RNF 01 - Desempenho: O front-end deve ser capaz de analisar o código fonte rapidamente, sem sobrecarregar o sistema.

RNF 02 - Escalabilidade: O front-end deve ser capaz de lidar com grandes quantidades de código fonte sem comprometer sua eficiência.

RNF 03 - Compatibilidade: O front-end deve ser capaz de ser executado tanto no Windows quanto no Linux.

RNF 04 - Robustez: O front-end deve ser capaz de lidar com código fonte mal escrito ou com erros de sintaxe, sem interromper o processo de análise.

RNF 05 - Confiabilidade: O front-end deve ser capaz de produzir resultados precisos e consistentes, independentemente do código fonte que está sendo analisado.

RNF 06 - Usabilidade: O front-end deve ser fácil de usar, com uma interface intuitiva e um conjunto de recursos que ajudem os programadores a identificar e corrigir erros de sintaxe.

RNF 07 - Manutenibilidade: O front-end deve ser fácil de manter e atualizar, permitindo aos desenvolvedores corrigir bugs e adicionar novos recursos sem prejudicar o funcionamento geral do compilador.

4 ANALISADOR LÉXICO

Fazendo um paralelo com a língua portuguesa a análise léxica em compiladores está relacionada com a análise léxica da língua portuguesa, pois nos dois casos é feita uma análise tentando identificar palavras/tokens.

No processo de análise léxica, dada uma expressão regular ou um autômato que reconhece certos tipos de tokens, é criado um algoritmo de uma máquina de estados implementada em um código para reconhecer os tokens definidos pela linguagem.

Na entrada do analisador léxico é recebido um conjunto de caracteres e deste conjunto é extraído todos os tokens encontrados com base em (autômatos que formam) regras para formação de tokens. A ferramenta ANTLR utilizada neste trabalho codifica a etapa do analisador léxico desta forma, (basicamente codificando os autómatos em linhas de códigos).

Tabela 1 - Classificação de tokens

Token	Código	Descrição
id (identificador)	1	Qualquer conjunto de caracteres que seja aceito pelo autômato da figura 1.
palavra_reservada	2	Conjunto de caracteres que formam as seguintes palavras: [programa, fimprog, se, senão, para , enquanto, faça]
token_num (numeral)	3	Qualquer conjunto de caracteres que seja aceito pelo autômato da figura 3.
token_relacional	4	Estes símbolos são os =, > e **<, !=** . Ou seja, qualquer caractere que é aceito pelo autômato da figura 4.
token_atribuição	5	Este símbolo é o = . Ou seja qualquer cadeia de caracteres que é aceita pelo autômato da figura 4.
token_final	6	Este símbolo expressa o final do comando, é o carácter **;** .Ou seja, qualquer caractere que é aceito pelo autômato da figura 5.
token_aritmético	7	Estes símbolos são os +, -, * e / . Ou seja, qualquer caractere que é aceito pelo autômato da figura 6.

token_parenteses e token_chaves	8	Referente aos símbolos: **‘(’** , **‘) ’** , **‘{‘** e ‘**}**’. Estão descritos no tópico 4.1.6.

Fonte: Elaborado pelos autores (2023)

Tabela 2 - Classificação de tokens

Símbolo	Código	Descrição
‘ ‘ (espaço), /n (quebra de linha), /t (tabulação), $...$ (comentário) $$.../n (comentário)	9	Estes símbolos são ignorados pelo analisador, para isso acontecer sua devida representação está representada pelo autômato da figura 7.

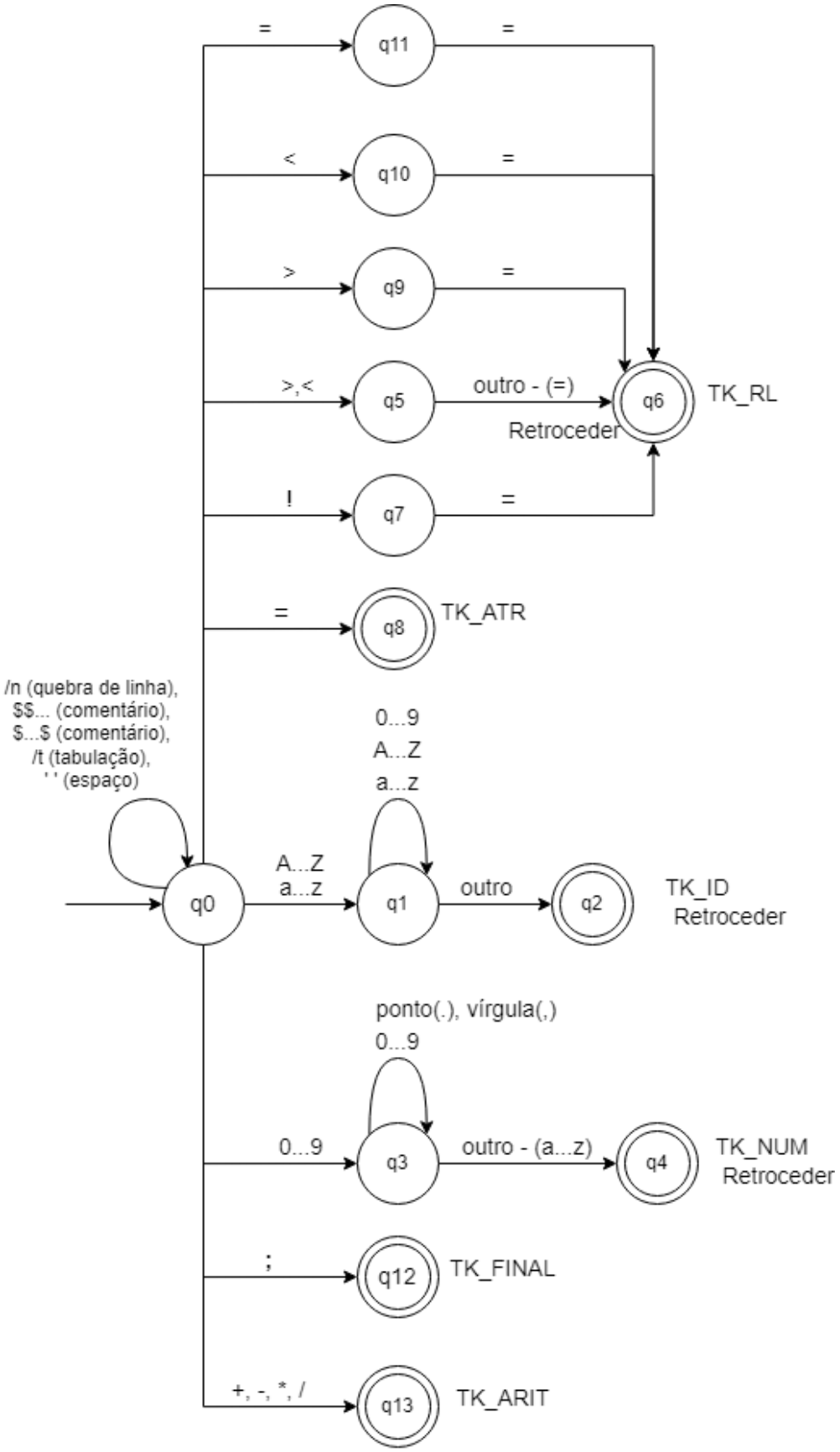
Fonte: Elaborado pelos autores (2023)

4.1 Processo de reconhecimento de tokens

A entrada do compilador é uma cadeia de caracteres que passará pela análise léxica. Nesta etapa, essa cadeia de caracteres será dividida em tokens e símbolos identificados nas tabelas 1 e 2. Os símbolos presentes na tabela 2 serão desconsiderados, pois são irrelevantes para a análise sintática e semântica.

A divisão da cadeia de caracteres em tokens é realizada através de um algoritmo baseado em um grande autômato (ilustrado na figura 1) composto por vários outros autômatos específicos para reconhecer diferentes tipos de tokens.

Figura 1 - Autômato completo, onde é reconhecido todos os tokens

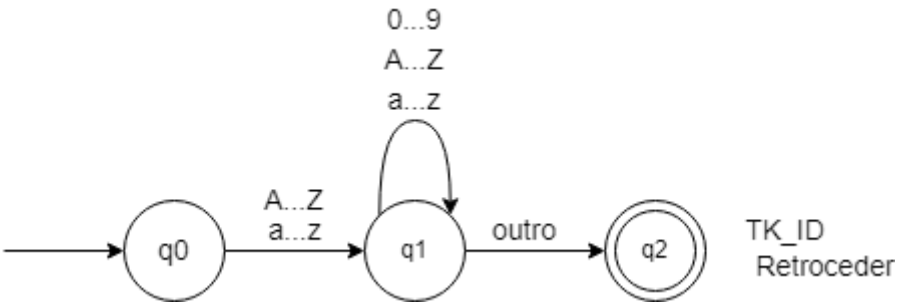


Fonte: Elaborado pelos autores (2023)

4.1.1 Autômato para identificar tokens id

Para reconhecer um Token_ID é necessário que a cadeia de caracteres seja reconhecida/aceita pelo seguinte autômato:

Figura 2 - Autômato para identificar identificadores



Fonte: Elaborado pelos autores (2023)

Caso seja aceita, é então reconhecido esse token como um identificador e a cadeia de caracteres a ser analisada volta um carácter para passar novamente pelo processo de reconhecimento de token, pois certamente o último carácter pertence a outro token ou é um símbolo ignorável.

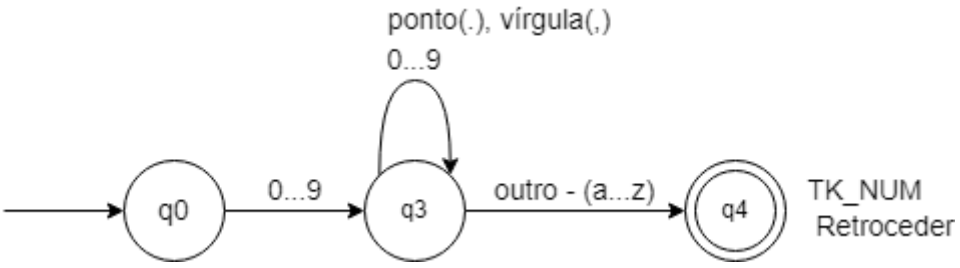
4.1.2 Para identificar tokens palavra_reservada

Todo token_ID pode ser reconhecido como um token palavra_reservada, então sempre que um token for reconhecido como id, ele deve passar por uma segunda etapa para verificar se este token é uma das seguintes palavras reservadas: programa, fimprog, se, senão, para , enquanto, faça. Os mesmos podem estar armazenados em estruturas de dados como mapas hash.

4.1.3 Autômato para identificar tokens num (NUMBER)

Para reconhecer um Token_NUMBER é necessário que a cadeia de caracteres seja reconhecida/aceita pelo seguinte autômato:

Figura 3 - Autômato para identificar numerais



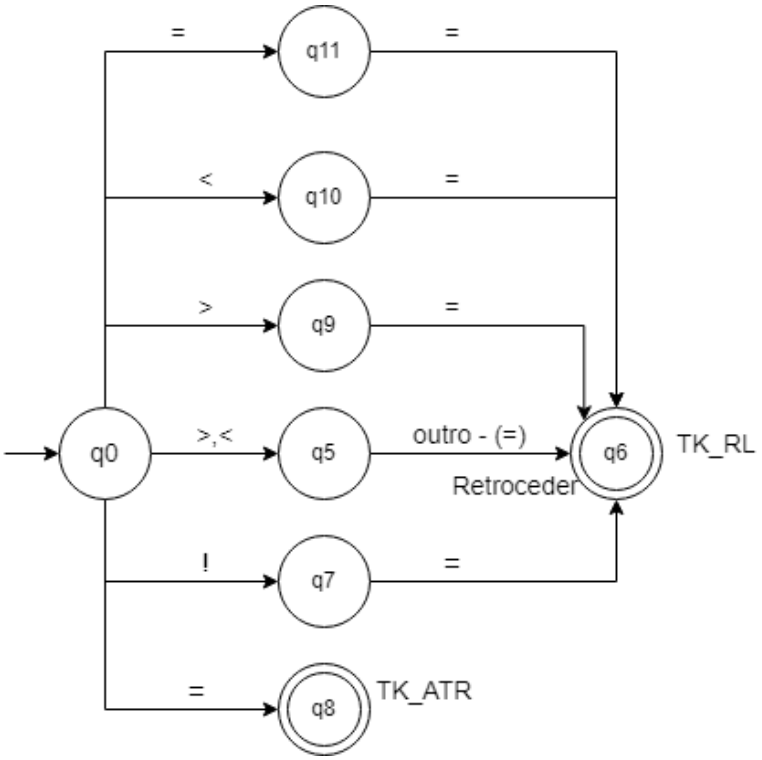
Fonte: Elaborado pelos autores (2023)

Caso seja aceita, é então reconhecido esse token como um número e a cadeia de caracteres a ser analisada volta um carácter para passar novamente pelo processo de reconhecimento de token, pois certamente o último carácter pertence a outro token ou é um símbolo ignorável.

4.1.4 Autômato para identificar tokens de relacionais

Para reconhecer um Token relacional (MAIORIGUAL, MENORIGUAL, IGUAL, DIF, MENOR, MAIOR) é necessário que a cadeia de caracteres seja reconhecida/aceita pelo seguinte autômato nos estados q6:

Figura 4 - Autômato para identificar tokens relacionais



Fonte: Elaborado pelos autores (2023)

Caso seja aceita, é então reconhecido esse token como um token relacional e a cadeia de caracteres a ser analisada volta um carácter para passar novamente pelo processo de reconhecimento de token, pois certamente o último carácter pertence a outro token ou é um símbolo ignorável.

4.1.5 Autômato para identificar tokens de atribuição

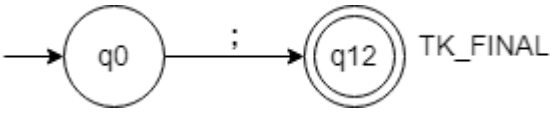
Para reconhecer um Token_ATTR é necessário que a cadeia de caracteres seja reconhecida/aceita pelo autômato da figura 4 no estado q8.

Caso seja aceita, é então reconhecido esse token como um símbolo de atribuição.

4.1.6 Autômato para identificar token final, parênteses e colchetes

Para reconhecer um Token_Final (SC) é necessário que a cadeia de caracteres seja reconhecida/aceita pelo seguinte autômato:

Figura 5 - Autômato para identificar tokens finais



Fonte: Elaborado pelos autores (2023)

Caso seja aceita, é então reconhecido esse token como um símbolo final.

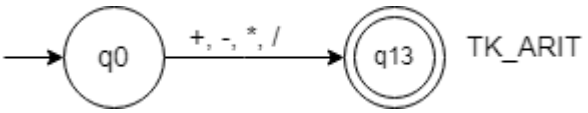
Como os autômatos para identificar os tokens abaixo, são da mesma forma que este, não foi desenvolvida uma figura para demonstração dos mesmos.

- Os tokens de parênteses AP referente a ‘(’ e FP referente a ‘) ’;
- Os tokens de chaves ACH referente a ‘{‘ e FCH referente a ‘}’.

4.1.7 Autômato para identificar tokens aritméticos

Para reconhecer um Token aritmético (SOM, SUB, MUL, DIV) é necessário que o caracter seja reconhecido/aceito pelo seguinte autômato:

Figura 6 - Autômato para identificar tokens aritméticos



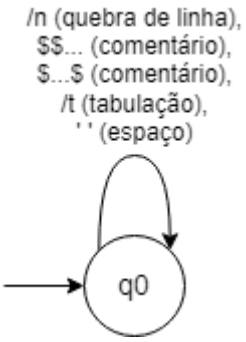
Fonte: Elaborado pelos autores (2023)

Caso seja aceito, é então reconhecido esse token como um token aritmético.

4.1.8 Autômato para ignorar símbolos de código 9 (tabela 2)

Para ignorar um símbolo de código 9 é necessário que o caracter seja reconhecido pelo seguinte estado:

Figura 7 - Autômato para identificar símbolos de código 9



Fonte: Elaborado pelos autores (2023)

Caso seja aceito pelo estado, é então ignorado esse símbolo para que continue analisando a cadeia de caracteres de forma que encontre tokens válidos.

5 ANALISADOR SINTÁTICO

A partir das regras sintáticas criadas na etapa do analisador sintático, será construído uma linguagem/idioma. Conforme o autor Isidro (2023) diz, a estrutura básica da linguagem, como a forma que as palavras são combinadas para formar frases, expressões e as regras de produção que descrevem como as sentenças são construídas a partir de símbolos elementares, serão todos implementados pela análise sintática. Basicamente é verificar se a sequência/conjunto dos símbolos/lexemas/tokens que foram reconhecidos como válidos pelo analisador léxico, eles efetivamente formam/obedecem uma regra da nossa gramática. Portanto se na análise léxica só estava preocupada com o reconhecimento de símbolos, na análise sintática só está preocupada com a sequência destes símbolos.

Em geral, as regras sintáticas são expressas usando notação formal, como a gramática livre de contexto (Context-free grammar, CFG) ou a gramática de Backus-Naur (BNF). Essas notações são usadas para especificar a estrutura sintática da linguagem e ajudam a definir as regras para a geração e análise de sentenças.

Ao criar o analisador sintático, é usado essas regras para verificar se a entrada de dados é sintaticamente válida ou não.

Para extrair a estrutura sintática da entrada de dados usamos técnicas como a análise:

- top-down (ou descendente) - inicia na raiz da árvore e segue para as folhas. Irá verificar, a partir do símbolo inicial, qual das regras será usada em função dos símbolos lidos. A análise sintática feita neste trabalho será feita usando este tipo de análise.
- análise bottom-up (ou ascendente) - inicia a partir das folhas em direção a raiz.

5.1 Restrições das regras sintáticas

As gramáticas livres de contexto devem ser basicamente LL(1), onde o primeiro token de cada regra identifica qual é a regra que eu preciso implementar. As restrições que temos que manter nas regras sintáticas, são basicamente:

- As regras não podem ter recursividade à esquerda, se tem recursividade à esquerda fazemos a eliminação da recursividade à esquerda:

  Exemplo:

  -A -> A α | β

  Eliminação da recursividade à esquerda:

-A -> β A’

-A’ -> α A’ | ε

- Se temos regras que têm derivações distintas com o mesmo prefixo, nós temos que fazer a fatoração à esquerda.

  Exemplo:

  COMANDOIF : 'if' AP EXPR FP BLOCO

  | 'if' AP EXPR FP BLOCO 'else' BLOCO

  Transformando com a fatoração à esquerda:

  COMANDOIF : 'if' AP EXPR FP BLOCO ('else' BLOCO | ε)

  Então eu tenho um prefixo em comum, que pode ou não ter um bloco.

5.2 Tradução das regras sintáticas em código

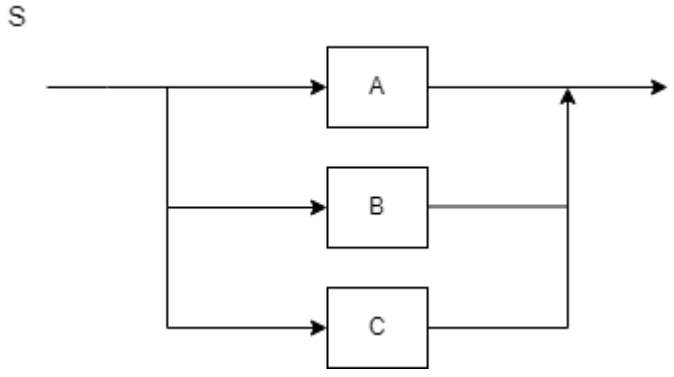
A ideia é que o compilador traduza as regras gramaticais/sintáticas para grafos sintáticos e dos grafos sintáticos para procedimentos. Todos os símbolos não terminais da nossa gramática tornam-se procedimentos/funções/métodos e os símbolos terminais consumo de tokens.

Exemplo:

S : A | B | C

Alternativas A B C mapeadas como um grafo com diferentes caminhos, onde cada um dos não terminais torna-se um retângulo que será mapeado para um determinado procedimento/método.

Figura 8 - Grafo sintático



Fonte: Elaborado pelos autores (2023)

O grafo da figura 8 origina o procedimento:

begin

se (símbolo está em First(A)){A}

senão se (símbolo está em First(B)) {B}

senão se (símbolo está em First(C)){C}

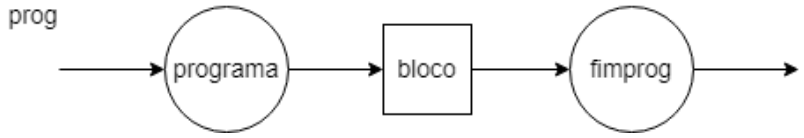
senão{erro}

end

O primeiro procedimento irá chamar o procedimento subsequente e assim por diante, até que se obtenha os símbolos referentes às regras sintáticas no padrão definido. Se for chamado o procedimento inicial e no final for encontrado o fim do arquivo, está tudo certo, caso contrário haverá erro, pois todos os procedimentos terminaram e ainda restou/restaram token/tokens no arquivo. A ferramenta ANTLR utilizada neste trabalho codifica a etapa do analisador sintático desta forma, (basicamente codificando os grafos sintáticos em linhas de códigos).

prog : 'programa' bloco 'fimprog'

Figura 9 - Grafo sintático da regra inicial prog



Fonte: Elaborado pelos autores (2023)

Todas as regras sintáticas definidas na etapa do analisador sintático, estão logo a seguir, com exceção da regra ‘prog’ que se encontra acima. Esta regra foi usada como exemplo para a criação do grafo sintático anterior. Para evitar trabalho desnecessário, não será criado neste documento, todos os grafos sintáticos para cada regra sintática do compilador. Basicamente o mesmo padrão é repetido para todas as regras, onde os terminais e palavras reservadas são representados com círculos e as variáveis com retângulos.

bloco : (cmd)+

cmd : cmddecl
| cmdleitura
| cmdescrita
| cmdattrib

| cmdsesenao
| cmdpara
| cmdenquanto
| cmdfacaenquanto

cmddecl : (declaravar)+

declaravar : tipo ID (VIR ID)* SC

tipo : 'numero' | 'texto'

cmdleitura : 'leia' AP ID FP SC

cmdescrita : 'escreva' AP texto FP SC

texto : (PALAVRAS | ID)*

cmdattrib : ID ATTR (expr | PALAVRAS) SC

cmdpara:'para' AP (cmdattrib)? termo oprel termo (SC termo '++')? FP ACH (cmd+)? FCH

cmdenquanto : 'enquanto' AP termo oprel termo FP ACH (cmd+)? FCH

cmdfacaenquanto : 'faca' ACH (cmd+)? FCH 'enquanto' AP termo oprel termo FP SC

cmdsesenao: 'se' AP termo oprel termo FP ACH (cmd+)? FCH ('senao' ACH (cmd+)? FCH)?

expr : termo (op termo)*

termo : ID | NUMBER

op : SOM | SUB | MUL | DIV

oprel : MAIOR | MENOR | IGUAL | DIF | MAIORIGUAL| MENORIGUAL

6 ANALISADOR SEMÂNTICO

Em um compilador, o analisador semântico é uma parte importante do processo de compilação que verifica se o código-fonte é válido seguindo as regras semânticas da linguagem de programação usada. O analisador semântico verifica as declarações de variáveis, funções e outros elementos do código para garantir que eles tenham o significado correto e sejam usados de maneira consistente. Ele também pode verificar coisas como tipos de dados, escopo de variáveis e correspondência de argumentos em chamadas de função. Se houver erros semânticos, o analisador semântico gera uma mensagem de erro para o programador/usuário.

6.1 Razões para realizar a análise semântica

As regras sintáticas/gramaticais só checam a forma da estrutura de um programa ou comando, sem garantir que o conteúdo tenha significado. Por isso, é necessário incluir regras semânticas adicionais manualmente para verificar a correção semântica da entrada.

6.2 Análise semântica e sua abordagem diferenciada em relação à análise léxica e sintática

Segundo o autor Isidro (2020), na análise semântica, não há procedimento mecânico/sistematizado de conversão, ou seja, não existe uma forma de mapear diretamente regras semânticas para um código que vai verificar essas regras. Entretanto, pode-se utilizar estruturas de dados para auxiliar na codificação das regras semânticas.
Exemplo de procedimentos sistematizados/mecânicos na análise léxica e sintática:

- Na análise léxica é criada uma expressão regular ou um autômato, deste autômato o código é redigido.
- Na análise sintática, com a gramática transformada para LL(1), geramos o grafo sintático, do grafo sintático nós geramos a sequência de procedimentos/consumo de tokens redigindo assim o código.

Anteriormente discutido, a análise semântica não é uma tarefa mecânica e não pode ser executada de maneira sistemática e automatizada. No entanto, existe outra ferramenta além das estruturas de dados para ajudar no processo de codificação de regras semânticas, chamada gramática de atributos. Esta permite escrever regras semânticas em conjunto com cada regra

sintática, adicionando atributos semânticos a cada uma delas. Este assunto será abordado no próximo tópico.

6.3 Sintetização de regras semânticas na gramática/sintaxe

Com toda a parte simbólica e estrutural da linguagem definida, no analisador semântico é preciso verificar se a entrada de dados gerada pelo usuário, geram efetivamente operações válidas na linguagem, e para isso são criadas as rotinas.

As rotinas são criadas no ANTLR injetando código java dentro das regras gramaticais. O parser se encarrega da estrutura e a semântica adiciona componentes extras a ela. Obviamente o arquivo da gramática fica ‘sujo/feio’ após a injeção de código java nas estruturas sintáticas, mas possibilita realizar várias coisas interessantes do ponto de vista de recuperar elementos que foram analisados pelo parser e pelo scanner, manipulando-os com base em rotinas pré-estabelecidas.

6.4 Rotinas/verificações implementadas no compilador

É utilizado neste trabalho a estrutura de dados mapa-hash para guardar os símbolos/tokens, com seu devido nome, valor e tipo.

As rotinas/regras semânticas implementadas neste compilador são as seguintes:

- Declarações de variáveis são possíveis;
- Reconhecer tipos e símbolos;
- É possível verificar qual o tipo de variável declarada;
- Identificar se a variável utilizada foi declarada;
- Identificar a existência de uma declaração prévia para a variável declarada;
- Apenas valores com tipos compatíveis com o da variável podem ser atribuídos a ela;
- O tipo da variável permanecerá inalterado após ser atribuído um valor;
- Em uma determinada expressão, os tipos são compatíveis/consistentes;
- É possível realizar operações aritmética entre dois valores;
- É possível realizar operações relacionais dentro de comandos if-else;
- Inicializar variáveis com valor null.

6.4.1 Como estas regras/rotinas semânticas são executadas/verificadas?

A semântica é orientada pela sintaxe, o que significa que enquanto o código é avaliado pelos critérios sintáticos, também são verificadas pequenas regras semânticas. A semântica é controlada pela sintaxe, permitindo a aplicação completa das regras semânticas codificadas dentro das estruturas sintáticas.

Graças a estas rotinas, o programa inserido pelo usuário não será apenas uma sequência de caracteres, mas também terá significado.

7 LINGUAGEM DE PROGRAMAÇÃO MANCO

A linguagem MANCO é uma linguagem imperativa (ou procedural), conseguindo usar comandos e instruções para realizar tarefas. Nesse tipo de construção, as instruções devem ser passadas ao computador na sequência em que devem ser executadas. Como todas as linguagens imperativas, o desenvolvedor tem o controle completo sobre as ações que o programa deve realizar. Isso inclui a definição de variáveis, laços de repetição, estruturas de decisão, entre outras características. A linguagem MANCO é uma ferramenta poderosa para soluções práticas e eficientes, especialmente quando se trata de problemas numéricos e lógicos.

7.1 Capacidades da linguagem

Nesta seção será contextualizado e exemplificado tudo o que a linguagem é capaz de realizar.

Vale ressaltar que as definições de toda a estrutura da linguagem já foram definidas nas seções 4, 5 e 6. Para que não haja eventuais dúvidas sobre quais são os padrões de caracteres, estruturas ou o sentido semântico das estruturas da linguagem é recomendado a leitura destas seções. Dito isto, também é importante mencionar que todos os exemplos desta seção estão de certa forma vinculados, por exemplo, não irá ser repetido o mesmo processo de declaração de variáveis já mostrado anteriormente, a declaração será apenas reutilizada.

7.1.2 Inserção de comentários

É possível inserir cadeias de caracteres que são ignoradas pelo compilador, e desta forma criar comentários. Os comentários são usualmente utilizados para facilitar a compreensão de um código ou realizar debugs.

Exemplo de código 1:

```
$$comentario realizado com sucesso
$comentario realizado com sucesso$
```

7.1.3 Declaração de variáveis

A linguagem de programação MANCO é uma linguagem fortemente tipada, o que significa que todas as variáveis precisam ser declaradas com um tipo específico e apenas valores com tipos compatíveis podem ser atribuídos a elas. A linguagem reconhece dois tipos de dados, sendo eles, 'numero' e 'texto'. Onde 'número' é qualquer valor inteiro, e texto é qualquer cadeia de caracteres de símbolos reconhecidos pela linguagem. A declaração das variáveis com estes tipos, é codificada da seguinte forma:

Exemplo de código 1:

```
numero a;
texto t1;
```

Também é permitido a declaração de variáveis por concatenação, onde o tipo da variável é chamada apenas uma vez, como mostrado abaixo.

Exemplo de código 2:

```
numero a, b, c, d, e;
texto t1, t2, t3, t4;
```

Caso seja utilizada uma variável que não foi declarada em algum lugar do código, será emitida uma mensagem de erro no console.

7.1.4 Atribuição de valores

A linguagem consegue atribuir valores do tipo 'numero' e 'texto' a uma variável já declarada, da seguinte forma:

Exemplo de código 3:

```
t1 = ~Ola  mundo, estou vivo~;
t3 = ~     ~;
b = 255;
```

Ou caso uma variável tenha o mesmo tipo, da variável que está sendo atribuída, então também será permitido a declaração desta atribuição.

Exemplo de código:

```
t2 = t1;
a = b;
```

7.1.5 Operações aritméticas

Há quatro tipos de operações aritméticas permitidas, sendo elas a soma, subtração, divisão e multiplicação. Essas operações passam pelos seguintes requisitos:

- Só é possível realizar estas expressões aritméticas se elas forem atribuídas para alguma variável, e esta variável deve ser do tipo ‘numero’;
- Expressões aritméticas não podem ocorrer com tipos diferentes de ‘numero’;
- É permitido apenas dois operadores por expressão aritmética;
- Variáveis e número podem ser operandos na expressão.

Exemplo de código:

```
a = 3 +  1;
b =  a -  2;
c = b * a;
d = 10 / 5;
```

Se for realizada uma divisão por 0, será emitida a seguinte mensagem de erro:
ERROR SEMANTICO./ by zero

7.1.6 Comando de escrita

É possível realizar comando de escrita, onde as informações passadas serão mostradas na tela do console para o usuário. Neste comando é permitido concatenar variáveis e texto. As variáveis, ao contrário do texto, possuem uma quebra de linha por padrão, então sempre que for utilizado este comando para escrever uma variável na tela, será feita a quebra de linha.
Exemplo de código:

```
escreva(~Ola  mundo, estou vivo~);
t1 = ~  ~;
escreva(~O valor da variável  a esta a seguir  ~ a);
```

7.1.7 Comando de leitura

Para realizar um comando de leitura, é necessário que passe para o comando leia a variável que irá guardar o valor digitado pelo usuário, e o valor inserido pelo usuário deve ser do mesmo tipo que esta variável.

Ao ser executado o comando leia, o compilador irá esperar que o usuário digite um valor.

Exemplo de código:

```
leia(a);
leia(t1);
```

7.1.8 Operações relacionais

Há cinco tipos de operações relacionais permitidas, sendo elas a comparação de maior, menor, maior igual, menor igual e diferente. Essas operações passam pelos seguintes requisitos:

- Expressões relacionais podem ser feitas somente dentro dos parênteses dos comandos ‘se senao’, ‘para’, ‘enquanto’, ‘faca enquanto’.
- Os operandos desta operação só aceitam variáveis do tipo ‘numero’ ou números inteiros.
- É permitido apenas dois operadores por expressão relacional.

Exemplo de código:

```
se ( a == b){} senao{  }
para (a = 0; a <= 10; a++){}
enquanto (a > b){ }
faca { }enquanto(a != b);
```

7.1.9 Comando ‘se senão’

Para realizar o comando ‘se senao’ nesta linguagem, basta informar a expressão relacional dentro do parêntese do comando ‘se’. Após isso, caso a expressão seja verdadeira, um determinado bloco de comandos será feito, caso a expressão seja falsa, um outro bloco de comandos será executado.

Exemplo de código:

```
se ( a == b){
        escreva (~Esta expressao relacional e verdadeira      ~);
}
senao{
        escreva(~ Esta expressao relacional e falsa    ~);
```

```
}
```

É possível também utilizar o comando ‘se’ com apenas a primeira condição, onde se a expressão não for verdadeira, nada acontece e o programa apenas continua para a próxima linha de comando, após as chaves do comando ‘se’.

Exemplo de código:

```
se ( a == b){
        escreva (~Esta expressao relacional e verdadeira      ~);
}
```

7.1.10 Comando ‘para’

Esta linguagem de programação reconhece a estrutura do comando ‘para’, mas não realiza nenhuma operação de compilação. A estrutura deste comando é basicamente a mesma que a do comando ‘se’, a diferença é que primeiramente é inserido no parênteses uma atribuição, depois uma expressão relacional e depois um incremento de uma variável.

Caso no futuro esse comando tenha seu back-end implementado ele fará o seguinte: Enquanto a expressão aritmética for verdadeira irá executar um bloco de comandos, somando a cada novo ciclo 1 a variável inserida.

Exemplo de código:

```
para (a = 0; a <= 10; a++){
        escreva(~Ola  mundo, estou vivo~);
}
```

7.1.11 Comando ‘enquanto’

Esta linguagem de programação reconhece a estrutura do comando ‘enquanto’, mas não realiza nenhuma operação de compilação. A estrutura deste comando é basicamente a mesma que a do comando ‘se’.

Caso no futuro esse comando tenha seu back-end implementado ele fará o seguinte: Enquanto a expressão aritmética for verdadeira irá executar um bloco de comandos.

Exemplo de código:

```
enquanto (a > b){
        escreva(~Ola  mundo, estou vivo~);
```

```
}
```

7.1.12 Comando ‘faca enquanto’

Esta linguagem de programação reconhece a estrutura do comando ‘faca enquanto’, mas não realiza nenhuma operação de compilação. A estrutura deste comando é basicamente a mesma que a do comando ‘se’.

Caso no futuro esse comando tenha seu back-end implementado ele fará o seguinte: Executar um bloco de comandos, e depois verificar se a expressão aritmética é verdadeira, enquanto ela for verdadeira, execute esse bloco de comandos.

Exemplo de código:

```
faca {
        escreva(~Ola  mundo, estou vivo~);
 }enquanto(a != b);
```

8 INSTRUÇÕES DE INSTALAÇÃO

Para executar o compilador é necessário utilizar uma IDE para desenvolvimento Java de sua preferência. No decorrer do desenvolvimento deste trabalho, o compilador foi testado no Eclipse IDE - Version: 2022-12 (4.26.0)(Link no Anexo I).

A seguir está uma lista de instruções para instalar o compilador da linguagem MANCO em uma IDE Java:

1. Certifique-se de ter a última versão da JDK (Java Development Kit) instalada em seu sistema (Anexo II).
2. Baixe o arquivo de instalação (Link no Apêndice I) do compilador MANCO em um local de sua escolha, este compilador utiliza a ferramenta Antlr 4, caso queira saber mais sobre, visite o Anexo III.
3. Abra a IDE Java de sua escolha.
4. No menu principal, selecione "File" e em seguida "Import".
5. Selecione a opção "Existing Projects into Workspace" e clique em "Next".
6. Clique em "Browse" e selecione a pasta de instalação baixada do compilador MANCO.
7. Clique em "Finish" para concluir a importação do projeto.
8. Verifique se o compilador MANCO foi corretamente importado e se está funcionando corretamente, você pode fazer isso escrevendo seu código no arquivo input.mnc que se encontra na raiz do projeto.
9. Agora você está pronto para começar a usar a linguagem MANCO em sua IDE Java.
   Obs.: As etapas podem variar ligeiramente de acordo com a IDE Java utilizada.

9 GUIA DE USUÁRIO

1. Instalação: Para a instalação do ambiente de desenvolvimento, siga o passo a passo da seção 8 deste documento, você pode encontrar links necessários na seção 12(Anexo) e na seção 13 (Apêndice) deste documento.
2. Ambiente de desenvolvimento: MANCO ainda é uma linguagem de desenvolvimento em seus estágios iniciais, necessitando ainda da utilização da IDE Eclipse, sendo ela na sua versão 2022-12 (4.26.0).
3. Sintaxe Básica: Toda sua sintaxe base está documentada na seção 7 deste documento, sendo uma das seções mais importantes para pessoas que desejam estudar a linguagem.
4. Tipos de dados: Recomenda-se uma atenção especial nas seções **7.1.3** e **7.1.4** onde é falado sobre os tipos de variáveis e comandos de atribuição suportados pela linguagem MANCO.
5. Exemplos: Alguns exemplos de estruturas condicionais, declarações de variáveis, laços de repetições podem ser encontrada na seção 7 deste documento, é fortemente recomendável que execute os mesmos para maior entendimento da linguagem MANCO.

10 LIMITAÇÕES E PROBLEMAS CONHECIDOS

Como toda linguagem de programação, problemas e limitações inconvenientes são de certa forma comuns. Neste tópico serão listados os principais problemas e limitações conhecidos da linguagem/compilador produzido neste trabalho .

10.1 Linguagem Manco: Limitações e problemas x Oportunidades para melhorias

Principais limitações e problemas conhecidos:

1. Há uma observação importante a se fazer em relação ao tipo token number. A linguagem consegue reconhecer/interpretar um número fracionado pois a sua regra léxica foi definida para aceitar esse tipo de padrão, (exemplo 2,4), entretanto não foi feio nenhuma operação que consegue compilar um número fracionário. Por exemplo, para a linguagem conseguir fazer algum tipo de operação aritmética, é preciso de dois números inteiros, ou variáveis do tipo ‘numero’;
2. A expressão aritmética possui essa mesma característica, sua regra sintática foi definida para aceitar operações com mais de dois operandos, entretanto a expressão aritmética só pode ser realizada apenas com dois operandos nesta linguagem;
3. Os comandos ‘para’, ‘enquanto’ e ‘faca enquanto’, são reconhecidos pela linguagem, são interpretados pela linguagem, mas eles não são compilados, porque não possuem back-end;
4. A linguagem possui apenas dois tipos de variáveis, sendo elas ‘numero’ e ‘texto’;
5. A linguagem não consegue declarar variáveis dentro de um escopo definido;
6. Qualquer variável declarada dentro do programa, pode ser usada em qualquer lugar;
7. Não é possível concatenar expressões relacionais com operadores lógicos and e or;
8. Não é possível criar funções ou procedimentos;
9. De maneira geral não é possível modularizar o código.

Agora que foram listadas as restrições e limitações, a linguagem Manco pode ser atualizada de uma forma bem mais assertiva, mirando em suas principais limitações.

11 CONCLUSÃO

Em suma, a implementação do front-end de um compilador é uma etapa fundamental no desenvolvimento de compiladores. Neste trabalho, documentamos a implementação do front-end de um compilador para a linguagem de programação MANCO, utilizando a ferramenta ANTLR. A utilização de uma ferramenta como o ANTLR foi fundamental para a agilidade e eficiência da implementação do front-end, permitindo a geração automática de muitos dos componentes necessários, como o analisador léxico e sintático. Além disso, a qualidade e eficiência do front-end afetam diretamente a qualidade e performance do compilador como um todo.

Este trabalho representou um importante passo na compreensão e na aplicação dos conceitos envolvidos na construção de compiladores, e esperamos que ele seja útil para futuras implementações de compiladores.

REFERÊNCIAS

ISIDRO, Professor. Compiladores - Curso Completo. **YouTube**, 2020. Disponível em: https://www.youtube.com/playlist?list=PLjcmNukBom6--0we1zrpoUE2GuRD-Me6W. Acesso em: 16 de janeiro de 2023.

12 ANEXOS

ANEXO I

https://www.eclipse.org/downloads/

ANEXO II

https://www.oracle.com/technetwork/pt/java/javase/downloads/index.htm

ANEXO III

https://www.antlr.org

13 APÊNDICE

APÊNDICE I

https://github.com/Carlos-Eduardo99/Trabalho-Final-Compiladores.git