



Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática



Compiladores

Relatório

Alunos: José Rafael Silva Hermoso ra: 112685
Renan Augusto Leonel ra: 115138

Maringá - Paraná
2023

SUMÁRIO

1. Introdução
2. Desenvolvimento
3. Dificuldades e facilidades encontradas
4. Conclusão
5. Referências

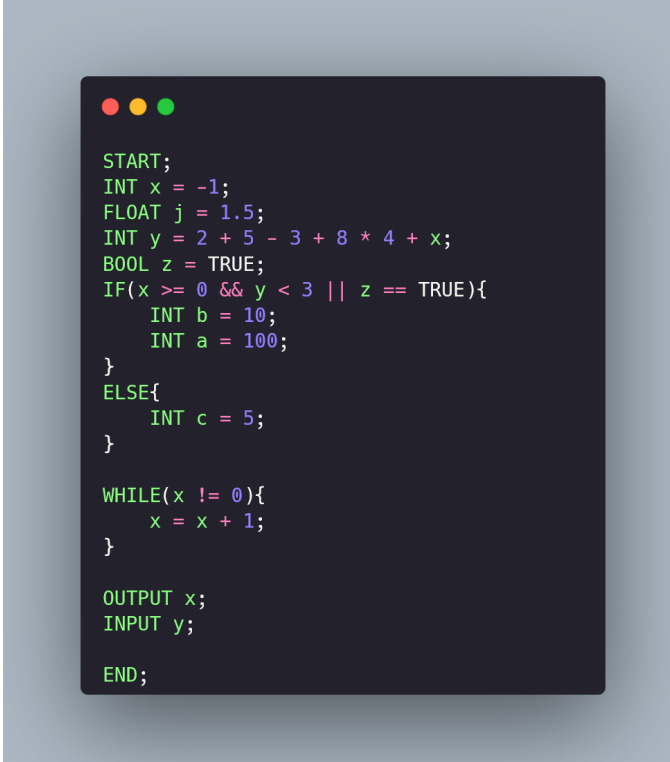
1. Introdução

Este trabalho possuiu como objetivo implementar o *front-end* de um compilador, que realiza análise léxica, sintática, semântica e geração de código intermediário, utilizando a linguagem de programação Java e a ferramenta ANTLR. A linguagem criada para ser utilizada como entrada é baseada na linguagem C, onde o código intermediário resultante é gerado em C++.

2. Desenvolvimento

Para o desenvolvimento do trabalho, foi utilizada a linguagem Java, em conjunto com a ferramenta ANTLR. O ANTLR é uma ferramenta que auxilia na geração de analisadores léxicos e sintáticos ao possibilitar a criação de gramáticas para especificar as regras, simplificando o processo de análise.

O compilador desenvolvido foi desenvolvido em 4 etapas: análise léxica, sintática, semântica e geração de código intermediário. Primeiramente, a análise léxica é responsável por analisar o código utilizado como entrada para agrupar os caracteres lidos em tokens léxicos, que podem incluir números, palavras-chave, símbolos, entre outros. O trecho de código utilizado em todas as etapas do desenvolvimento pode ser encontrado na Figura 1 abaixo:



```
START;
INT x = -1;
FLOAT j = 1.5;
INT y = 2 + 5 - 3 + 8 * 4 + x;
BOOL z = TRUE;
IF(x >= 0 && y < 3 || z == TRUE){
    INT b = 10;
    INT a = 100;
}
ELSE{
    INT c = 5;
}

WHILE(x != 0){
    x = x + 1;
}

OUTPUT x;
INPUT y;

END;
```

Figura 1: Código de entrada utilizado.

A Figura 2 representa a gramática elaborada para que seja possível o agrupamento dos caracteres em tokens léxicos, apresentando também todos os tokens que são reconhecidos pela linguagem desenvolvida.

```
RESERVED_WORDS: 'START' | 'IF' | 'ELSE' | 'WHILE' | 'INPUT' | 'OUTPUT' ;
INT_TYPE: 'INT';
FLOAT_TYPE: 'FLOAT';
BOOLEAN_TYPE: 'BOOL';
STRING_TYPE: 'STRING';
END: 'END';
SEMICOLON: ';';
OP_PAR: '(';
CL_PAR: ')';
OP_BRA: '{';
CL_BRA: '}';

BOOLEAN_VALUES: 'TRUE' | 'FALSE';

IDENTIFIER: [a-zA-Z][a-zA-Z0-9]*;

INT_VALUES: ('+' | '-')? ('0' .. '9')+;
FLOAT_VALUES: ('+' | '-')? ('0' .. '9')+ '.' ('0' .. '9')+;
STRING_VALUES: '"' ( ~["\\] | '\\' . )* '"';

ASSIGNMENT_OP: '=' ;
ADD_OP: '+' | '-';
MULT_OP: '*' | '**' | '/' | '%';
REL_OP: '>' | '>=' | '<' | '<=';
EQU_OP: '==' | '!=';
LOGICAL_OP: '&&' | '||';

WS: (' ' | '\t' | '\r' | '\n') -> skip;
LINE_COMMENT: '#' ~[\r\n]* -> skip;
BLOCK_COMMENT : '/*' .*? '*/' -> skip;
```

Figura 2: Gramática para a linguagem desenvolvida

Podemos gerar autômatos para a gramática elaborada, que resultam nas figuras abaixo:

reconhecedor de palavras chave

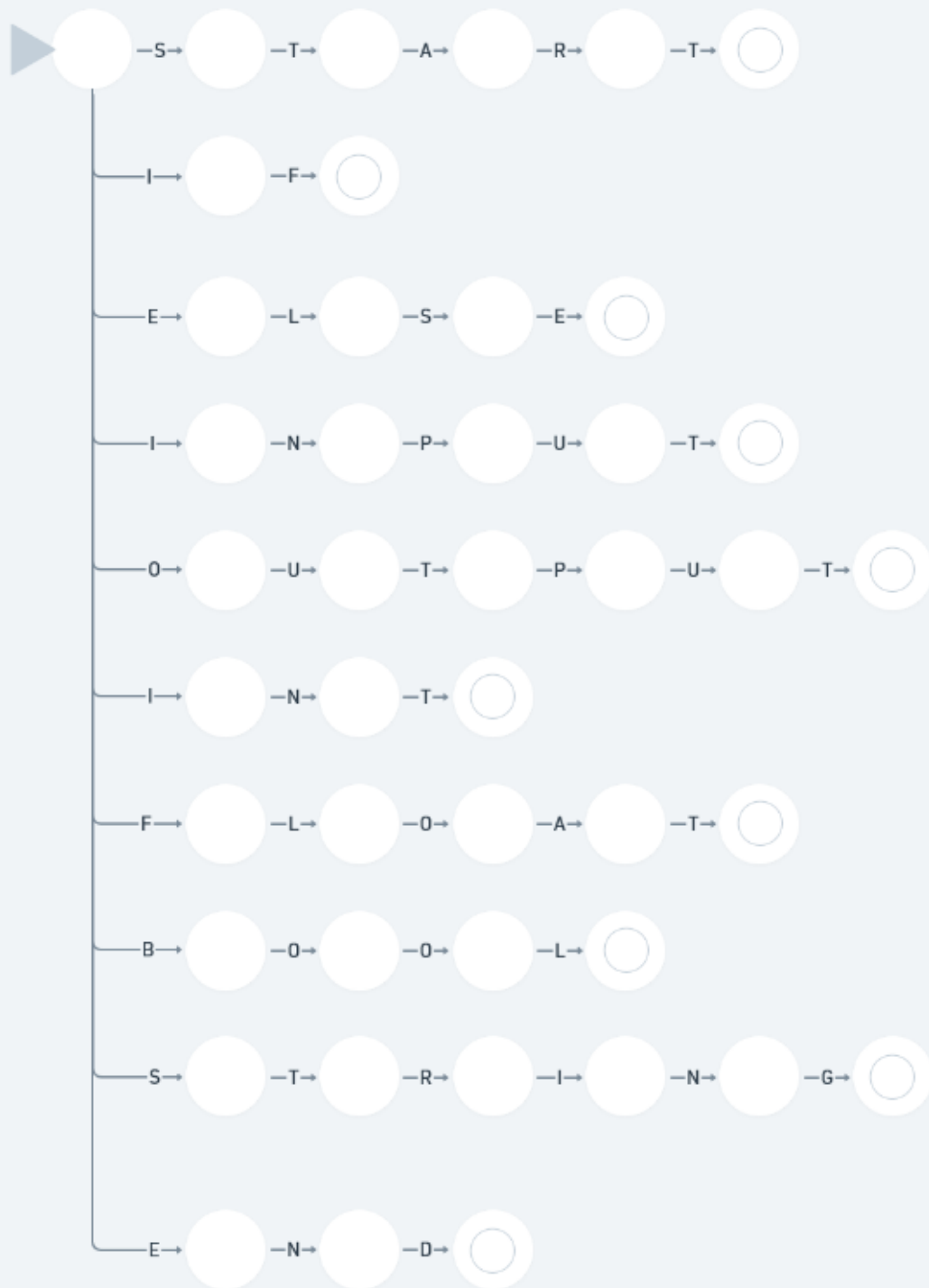


Figura 3: Autômato reconhecedor de palavras-chave.

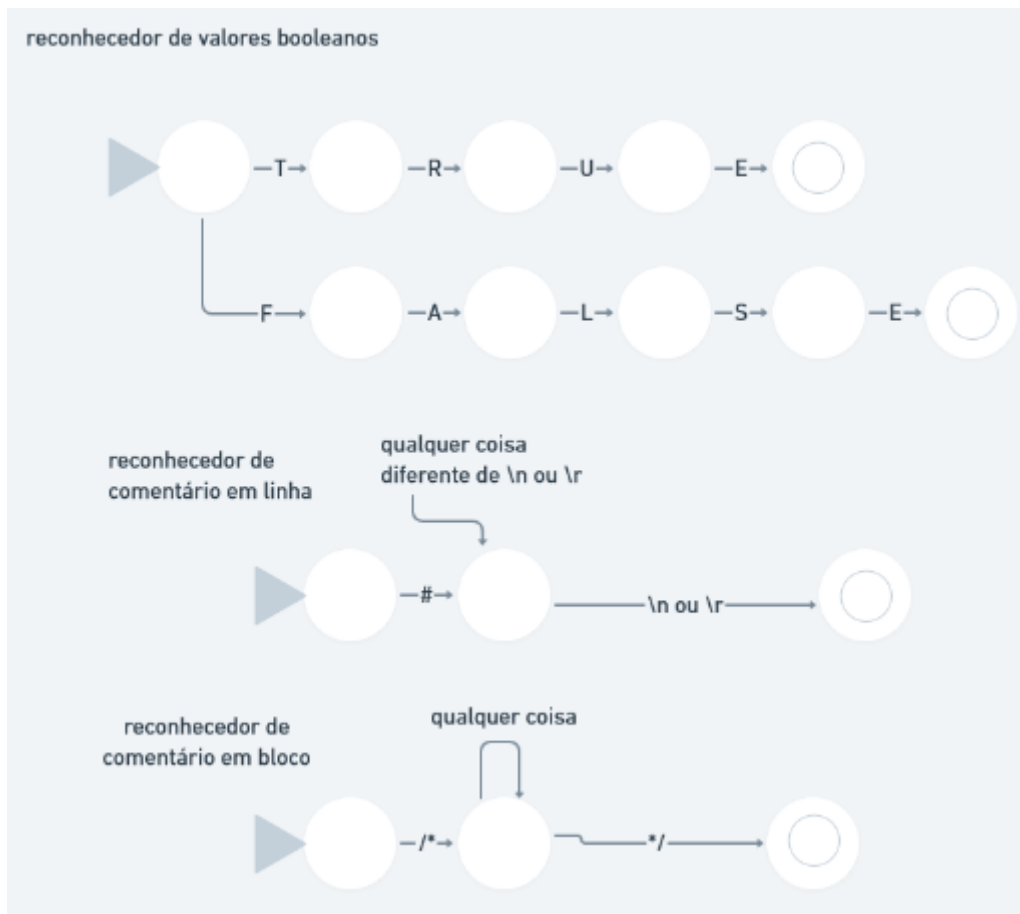


Figura 4. Autômato reconhecedor de valores booleanos e comentários.

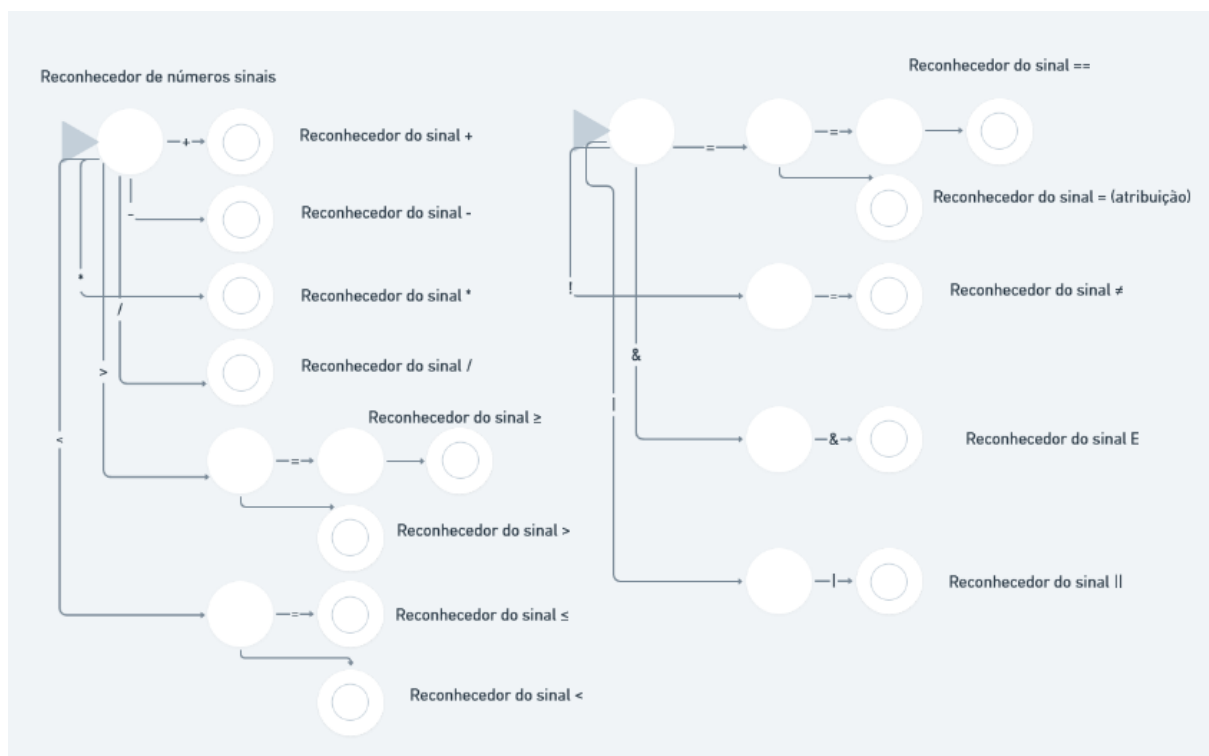


Figura 5. Autômato reconhecedor de operadores lógicos, aritméticos e condicionais.

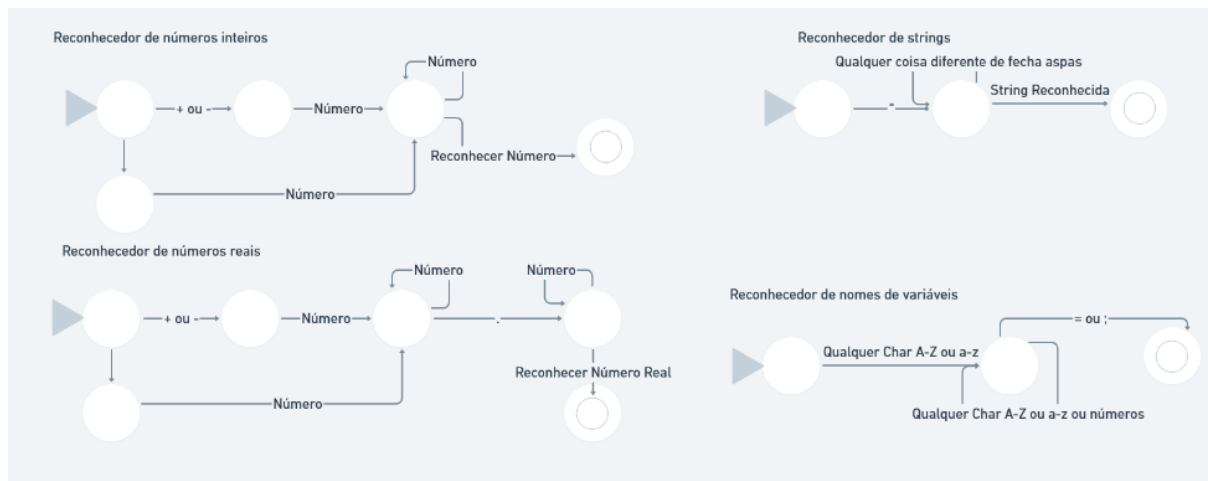


Figura 6: Autômato reconhecedor de números inteiros, reais, strings e nomes de variáveis

Após o agrupamento dos caracteres em tokens, foi realizada a análise sintática do compilador, que consiste na verificação se a estrutura gramatical do código segue as regras sintáticas definidas. A partir da gramática da Figura 2, foram elaboradas as regras sintáticas para a linguagem, como mostra a Figura 7, e a árvore sintática abstrata, como mostra a Figura 8. A gramática e as regras sintáticas utilizadas podem ser encontradas no arquivo .g4, que é a extensão da ferramenta ANTLR utilizada.

A linguagem desenvolvida reconhece quatro tipos de variáveis (inteiro, float, booleano e string), input e output de variáveis, assim como blocos de código if e while. Além disso, também reconhece todos os operadores lógicos e relacionais especificados na gramática apresentada na Figura 2, como por exemplo, &&, ||, >=, entre outros.

```

start: 'START' SEMICOLON cmdList* END SEMICOLON EOF ;
cmdList: declaration | assignment | exeCmd ;
declaration: (INT_TYPE IDENTIFIER SEMICOLON ) |
             (FLOAT_TYPE IDENTIFIER SEMICOLON ) |
             (STRING_TYPE IDENTIFIER SEMICOLON ) |
             (BOOLEAN_TYPE IDENTIFIER SEMICOLON ) |
             (INT_TYPE IDENTIFIER '=' mathExp SEMICOLON ) |
             (FLOAT_TYPE IDENTIFIER '=' mathExp SEMICOLON ) |
             (STRING_TYPE IDENTIFIER '=' STRING_VALUES SEMICOLON ) |
             (BOOLEAN_TYPE IDENTIFIER '=' relExp SEMICOLON );

mathExp: mathTerm ((ADD_OP | MULT_OP) mathTerm)*;
mathTerm: INT_VALUES | FLOAT_VALUES | IDENTIFIER | '(' mathExp ')';

relExp: relTerm ((LOGICAL_OP | EQU_OP | REL_OP) relTerm)*;
relTerm: mathExp | BOOLEAN_VALUES | '(' relExp ')';

assignment: IDENTIFIER ASSIGNMENT_OP (mathExp | relExp | STRING_VALUES) SEMICOLON;

exeCmd: inputCmd | outputCmd | whileCmd | ifCmd ;

inputCmd: 'INPUT' IDENTIFIER SEMICOLON;

outputCmd: 'OUTPUT' (IDENTIFIER | STRING_VALUES) SEMICOLON;

whileCmd: 'WHILE' OP_PAR relExp CL_PAR OP_BRA cmdList* CL_BRA;

ifCmd: 'IF' '(' relExp ')' '{' cmdList* '}' elseCmd?;

```

Figura 7. Regras sintáticas para a linguagem

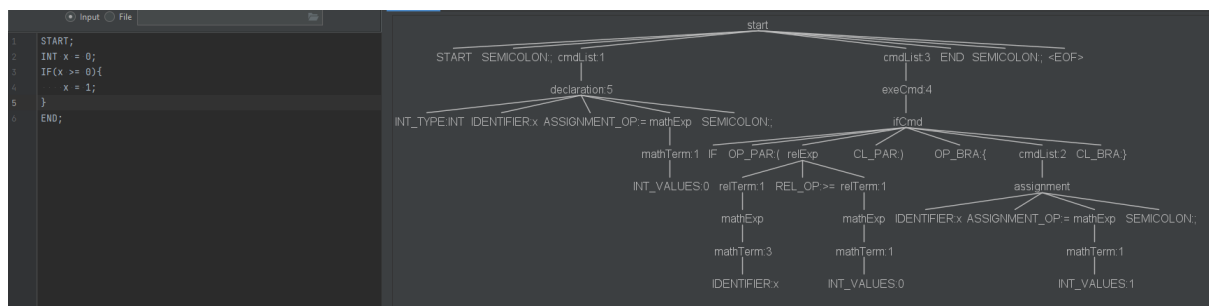


Figura 8. Árvore sintática abstrata

Em seguida, foi realizada a análise semântica, que consiste em verificar se o código faz sentido considerando as regras sintáticas estipuladas na fase anterior, analisando o contexto e considerando informações como tipo das variáveis e se a variável foi declarada antes de seu uso. A análise semântica é fundamental para que o processo de geração de código não possua erros. Para a implementação, foi utilizado uma tabela de símbolos, com o intuito de inserir e verificar informações a

partir das declarações e utilização dos elementos declarados. Essa tabela foi usada para controle de escopo e verificações de declarações e tipos de variáveis, para evitar atribuições indevidas e múltiplas variáveis iguais no mesmo escopo. Este compilador utiliza apenas uma tabela de símbolos, portanto seu escopo é global.

A ferramenta ANTLR, após a análise sintática, gera uma árvore que pode ser percorrida através dos *visitors*. Os *visitors* são interfaces geradas pela ferramenta e correspondem às regras definidas na gramática, cada regra possui o seu próprio *visitor*. Como se tratam de interfaces, é possível através de herança fazer sua própria implementação, o que permite aplicar as regras semânticas da linguagem de forma separada para cada regra que foi definida, facilitando o cumprimento das regras e dispensando uma possível caminhada em pós ordem na árvore.

Por fim, a última etapa de desenvolvimento do compilador consistiu na geração de código intermediário. Essa etapa consiste na geração de uma interpretação intermediária para o programa fonte, onde foi utilizada uma representação no modelo da linguagem C++ como código-alvo.

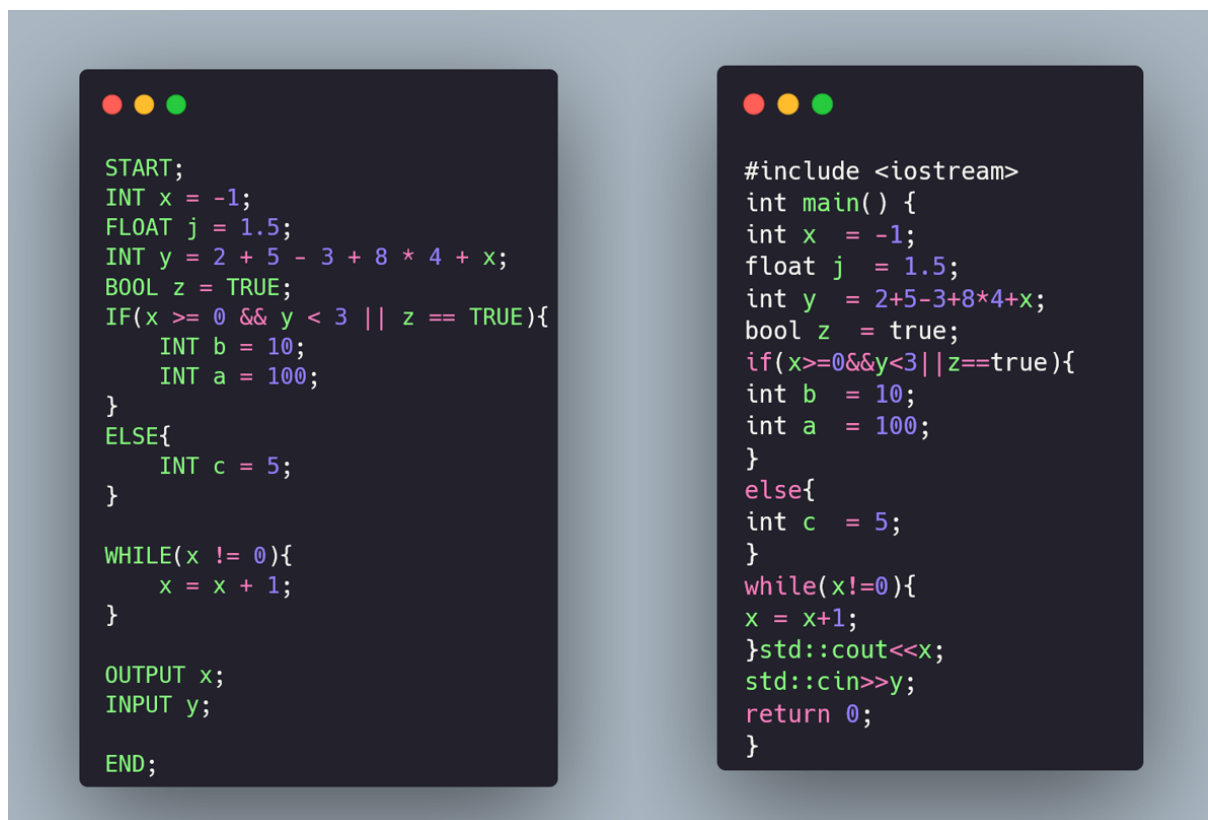


Figura 10: Código de entrada (à esquerda) e código gerado em C++ (à direita).

A geração de código também utilizou os *visitors*, porém de forma separada da análise semântica, com uma nova implementação da sua interface. Nesse ponto a avaliação da árvore gerada já foi realizada, então as regras praticamente foram ignoradas e o código de entrada foi reescrito na forma que a linguagem C++ aceita.

3. Dificuldades e facilidades encontradas

Durante o desenvolvimento, foram encontradas algumas dificuldades que foram superadas ao final do trabalho. Podemos citar, primeiramente, o processo de instalação da ferramenta, processo este que exige com que o desenvolvedor insira diversos plugins nos arquivos fonte do projeto Java.

Além disso, o processo de instalação e localização dos arquivos no projeto pode variar de acordo com a IDE utilizada, pois a estrutura de pastas para que o ANTLR consiga compilar o arquivo de forma correta deve ser seguida à risca. Para a geração da árvore sintática, foi necessário utilizar a IDE IntelliJ, onde a partir desta etapa optamos por migrar o projeto do Netbeans para o IntelliJ, sendo necessário readequar alguns arquivos para que o funcionamento não fosse afetado.

A principal dificuldade encontrada no desenvolvimento do compilador está diretamente relacionada com a definição da gramática e, conseqüentemente, das regras sintáticas. Durante a etapa de análise semântica e geração de código intermediário, foram necessárias diversas reavaliações e alterações das regras sintáticas para que o resultado esperado pudesse ser alcançado.

Como exemplo, podemos citar especificamente as regras “ifCmd” e “elseCmd”, que originalmente eram apenas uma regra. Após sucessivas tentativas de gerar o código correto para um exemplo em que existam os blocos `if(){}` `else{}`, foi necessário separar a regra original em duas, onde a partir disso obtivemos o resultado esperado.

Como facilidade encontrada, podemos citar o reaproveitamento da lógica para a ampla maioria das funções de análise semântica e geração de código intermediário, onde se faz necessário entender o funcionamento de uma função para uma regra específica, e replicar o mesmo fluxo de lógica para as demais funções desta etapa. Um outro ponto forte da ferramenta é o tratamento de recursão a esquerda de forma automática, não sendo necessário fazer a remoção da recursão de forma manual com regras adicionais.

4. Conclusão

Com base no trabalho apresentado, podemos concluir que a ferramenta ANTLR facilita no desenvolvimento das etapas de análise léxica e sintática, ao permitir a utilização de gramáticas e regras sintáticas de forma simplificada. O uso da ferramenta facilitou o processo de construção do compilador, uma vez que esse processo é bem complicado, passando por definições de *tokens* e chegando a definição de regras semânticas para dar sentido ao que foi escrito. Todo esse

processo, foi realizado com a ferramenta, sobrando apenas a geração de código, que também utilizou recursos da ferramenta, porém sua grande maioria foi feita de forma manual. O objetivo foi alcançado e a linguagem *FCp* pode ser compilada para C++.

5. Referências

<https://github.com/antlr/grammars-v4>

<https://wwwantlr.org/api/maven-plugin/latest/source-repository.html>

<https://wwwantlr.org/download.html>