ALEF DE LIMA
ELIAS GOMES DA SILVA
JULIANA FREITAS FERREIRA
RENNAN DE ALMEIDA SANTOS
WESLEY EGBERTO DE BRITO

INTRODUZINDO JOVENS À LÓGICA DE PROGRAMAÇÃO

Osasco
2016

ALEF DE LIMA
ELIAS GOMES DA SILVA
JULIANA FREITAS FERREIRA
RENNAN DE ALMEIDA SANTOS
WESLEY EGBERTO DE BRITO

INTRODUZINDO JOVENS À LÓGICA DE PROGRAMAÇÃO

Monografia apresentada ao Centro Universitário Fundação Instituto de Ensino para Osasco – UNIFIEO, como requisito parcial para obtenção do título de Graduação em Ciência da Computação, sob orientação da Professora Drª Andreia Cristina Grisolio Machion.

Osasco
2016

RESUMO

Esta monografia aborda o desenvolvimento de um kit para auxílio na introdução da lógica de programação para jovens, levando em consideração o engessamento da estrutura escolar e metodologias atuais de ensino do tema nas escolas. Propõe-se um modelo de fácil manuseio, amigável, interativo e de certa forma, lúdico. O Kit é composto por uma interface gráfica, que proporciona uma programação visual utilizando-se blocos de operações, ações e condições inspirados em peças de quebra-cabeças, um compilador, responsável pela tradução do quebra-cabeça montado pelo usuário e um robô, que interpreta a saída gerada pelo compilador e executa os comandos inseridos pelo usuário na interface.

Palavras-chave: lógica de programação, compilador, ensino de programação, interatividade, robótica.

ABSTRACT

This monograph broaches the development of a kit to support the introduction of programming logic for young people, taking into account the inflexibility of the school structure and the current methodologies for teaching the subject in schools. We propose a model easy to use, friendly, interactive and somewhat playful. The kit consists of a graphic interface that provides a visual programming using blocks of operations, actions and conditions inspired by puzzle pieces, a compiler, responsible for the translation of the "puzzle" created by the user and a robot, that interprets the output generated by the compiler and executes commands entered by the user in the interface.

Keywords: logic programming, compiler, programming teaching, interactivity, robotics.

LISTA DE ILUSTRAÇÔES

Figura 1 – A interface da aplicação 14
Figura 2 – Exemplo de utilização dos operadores 14
Figura 3 – Operadores e operandos 15
Figura 4 – Condição SE/ENTÃO 15
Figura 5 – Condição PARA 15
Figura 6 – Condição ENQUANTO 16
Figura 7 – Ações AVANÇAR e RECUAR 16
Figura 8 – Ações VIRAR DIREITA e VIRAR ESQUERDA 17
Figura 9 – Ações de acender as luzes LED 17
Figura 10 – Ações de apagar as luzes LED 17
Figura 11 – Gramática da linguagem 23
Figura 12 – Estrutura do compilador 23
Figura 13 – AFND do token ID 25
Figura 14 – AFD gerado a partir do AFND da Figura 13 26
Figura 15 – Resultado da união. 28
Figura 16 – AFD do analisador léxico 30
Figura 17 – Item I_0 33
Figura 18 – Item I_1 33
Figura 19 – Coleção de itens da gramática de exemplo 34
Figura 20 – Árvore gramatical gerado na simulação da Tabela 8 52
Figura 21 – Blocos selecionados na interface gráfica 53
Figura 22 – Árvore gramatical gerada 55
Figura 23 – Arduino Uno R3 59
Figura 24 – Esquema de funcionamento do motor DC. 60
Figura 25 – Variação de ondas. 61
Figura 26 – Seleção de sentido de rotação na ponte H. 62
Figura 27 – Entradas e saídas da ponte H. 63
Figura 28 – Diagrama de interligação 67
Figura 29 – Robô 68

LISTA DE TABELAS

Tabela 1 – Execução do AFD da Figura 14 26
Tabela 2 – Tokens da linguagem 27
Tabela 3 – Tabela de transições da Figura 19 34
Tabela 4 – Conjunto First e Follow das variáveis 35
Tabela 5 – Tabela Ação/Desvio 36
Tabela 6 – Seção Ação 39
Tabela 7 – Seção Desvio 43
Tabela 8 – Simulação do analisador sintático 46
Tabela 9 – Tipos de dados e seus comandos 50
Tabela 10 – Tabela de pinos 59
Tabela 11 – Tabela de ativação dos motores 64
Tabela 12 – Dificuldade na utilização da aplicação 69

SUMÁRIO

1. INTRODUÇÃO 7
1.1. OBJETIVO 11
1.2. OBJETIVOS ESPECÍFICOS 11
1.3. JUSTIFICATIVA 11
2. PROJETO DE SOFTWARE 13
2.1. APLICAÇÃO 13
2.1.1. Operadores/operandos 14
2.1.2. Condições 15
2.1.3. Ações 16
2.2 COMPILADOR 18
2.2.1. Análise léxica 19
2.2.2. Análise sintática 19
2.2.3. Análise semântica 20
2.2.4. Gerador de código 21
2.3. A LINGUAGEM CONSTRUÍDA E SEU COMPILADOR 21
2.3.1. Analisador léxico 24
2.2.2. Analisador sintático 32
2.2.3. Gerador de código e o interpretador 48
2.2.4. Exemplo de compilação completa 53
2.4. INTEGRAÇÃO 56
3. PROJETO DE HARDWARE 58
3.1. ARDUINO 58
3.2. MOTORES 60
3.3. PONTE H 61
3.3.1. PWM 61
3.3.2. O circuito 61
3.4. BATERIA 64
3.5. COMUNICAÇÃO SEM FIO 65
3.6. SENSORES E LEDs 65
4. PRIMEIROS RESULTADOS 68
5. CONCLUSÃO 70
5.1. TRABALHOS FUTUROS 70
BIBLIOGRAFIA 71

1. INTRODUÇÃO

Estudos demonstram que o ensino da lógica de programação tem auxiliado as pessoas na resolução de problemas por meio do desenvolvimento da lógica, comunicação, raciocínio e criatividade (DIM; ROCHA, 2011). Mas esse conteúdo só é apresentado em cursos superiores ou técnicos de áreas específicas, por isso, apenas uma pequena parte dos estudantes acaba tendo tal oportunidade de desenvolvimento.

Nos períodos iniciais dos cursos superiores de tecnologia e engenharia, existe um rol de disciplinas comuns das áreas de matemática e programação que exige dos alunos habilidades relacionadas ao raciocínio lógico. Desde os primeiros meses os alunos encontram dificuldades no estudo de algoritmos, o que segundo Raabe e Silva (2005 *apud* DIM; ROCHA, 2011) ocasiona reprovações e até mesmo desistências desses cursos. Tais dificuldades podem ser atribuídas ao ensino anteriormente aplicado aos alunos, já que, de forma geral, os pensamentos crítico, lógico e construtivo não são amplamente estimulados na formação básica (MALTEMPI; VALENTE, 2000).

Nesses cursos, na maioria das vezes, a lógica de programação é apresentada de forma pouco flexível. Os comandos e suas estruturas são apresentados de forma direta, com grande foco na sintaxe das linguagens e há pouca preocupação ou tempo suficiente para apresentar ao aluno a possibilidade de utilizar a linguagem de programação para expressar o raciocínio lógico. A utilização de algoritmos para resolução de problemas exige que o aluno interprete e reflita sobre ele e que seja criativo, sequencial e direto ao utilizar uma linguagem de programação (MALTEMPI; VALENTE, 2000). O fato de ter que detalhar a solução de forma sequencial e objetiva, o que a linguagem de programação exige, faz com o que aluno tenha maior dificuldade, já que, em geral, não está habituado a resolver problemas de maneira, e sim a seguir exemplos semelhantes ao problema dado para chegar a uma resolução.

A metodologia de ensino apresentada atualmente nas escolas, “siga o exemplo”, em uma aula de matemática, por exemplo, não estimula o aluno a pensar sobre como chegar a uma determinada solução. A forma de solução já lhe é apresentada, e o que ele deve fazer é simplesmente replica-la a partir do exemplo e substituir os valores e chegar ao resultado esperado. Em nenhum momento ele se preocupa se a forma que ele está utilizando para resolver um problema é a única ou se existe alguma outra maneira de resolver e chegar ao mesmo resultado (VALENTE, 1993; MALTEMPI e VALENTE, 2000). Isso significa que o ensino está engessado por

técnicas muitos utilizadas nos livros didáticos do tipo “resolva conforme o exemplo”, tornando o estudo repetitivo. Os conceitos também são apresentados de forma desvinculadas das nossas vidas, sem exemplos do mundo real, ajudando assim com a falta de interesse e de vontade do aluno.

O ensino da matemática não é diferente do ensino de computação, pois ambas requerem raciocínio lógico e também podem ser usadas para reforçar essa habilidade. A origem da palavra “matemática” significa técnica de entender ou compreender (“tica” significa técnica e “matema” significa compreender). Em Kline (1973 *apud* VALENTE, 1993), tem-se algumas justificativa para o ensino da matemática que podem ser resumidas em:

- Transmitir os fatos matemáticos: ensino dos conceitos matemáticos que todos devem conhecer;
- Pré-requisito para o sucesso: as profissões que possuem maior destaque exigem um conhecimento básico em matemática;
- Beleza intrínseca à estrutura matemática: transmitir a beleza e o poder mental que nos permite visualizar e construir todo o raciocínio inicial de um teorema ou equação, e
- Treino da mente: propiciar o desenvolvimento disciplinado do raciocínio lógico-dedutivo.

Sendo assim, o ensino da matemática, quando considera tais justificativas e reforça a resolução de problemas, ajuda desenvolver o raciocínio.

No entanto, atualmente o estudo da matemática, muitas vezes, é sinônimo de fobia entre os alunos. Isso ocorre porque o processo de fazer matemática (pensar e raciocinar) é resultado da imaginação, intuição, tentativas e erros. O matemático efetua toda uma organização do raciocínio para que, a partir da confusão mental, obtenha uma solução. Porém, o que o aluno faz não parece em nada com o que o matemático faz. Ele apenas memoriza o que já foi consumado pelo matemático. De todo conceito, lógica e raciocínio que são passados, ele apenas memoriza as fórmulas resultantes. E mesmo que houvesse a apropriação do conceito em um determinado contexto, a aplicação desse conceito em outro deve ser encarada de forma diferente (VALENTE, 1993).

Com isso, é possível perceber que não é exigido do aluno qualquer raciocínio lógico, pois toda a lógica da resolução já foi desenvolvida anteriormente. Não se está

querendo afirmar que os alunos devem ser capazes de criar fórmulas próprias para resolver um determinado problema, mas ele deve tentar enxergar o porquê de se estar utilizando tal fórmula; Como se chega a essa conclusão; A partir de qual conceito; se existe ou não outra maneira de se fazer. Esse tipo de questionamento crítico faz com que o aluno tente, mesmo que não consiga, pensar de uma forma lógica. E hoje isso não é visto no modelo de ensino atual, pois nada disso é exigido dele. Consequentemente, quando se depara com problemas nos quais tem que usar o raciocínio lógico para resolvê-los, tem maior dificuldade.

Felizmente, hoje existem várias tecnologias que podem ajudar a mudar ou complementar esse modelo de ensino, fazendo com que os alunos desde mais cedo, ainda na adolescência, comecem a pensar e raciocinar de forma lógica (BERNARDI *et al.*, 2007). Mas se já existem essas tecnologias, por que ainda não estão sendo aplicadas na grande maioria das escolas? Isso poderia ser prejudicial de alguma forma no desenvolvimento dos alunos? Existe algum receio sobre qual seria o papel do professor nesse modelo de ensino? Também é comum diversos artigos, reportagens que abordam o tema sobre as crianças em contato com novas tecnologias, como os smartphones, que de alguma forma acabam prejudicando seu desenvolvimento, pois levam-nas ao isolamento em seu mundo “próprio”. Com isso, deixam de conviver com as coisas mais simples ao seu redor, com brincadeiras simples de crianças ou adolescentes (VALENTE, 1993; BATISTA e NAPOLITANO, 2003).

Em uma sala de aula, onde o aluno tenha contato com uma tecnologia que o apoie em seu desenvolvimento e esteja sendo supervisionado e auxiliado por um professor capacitado, que continua com seu importante papel de condução e direcionamento do aluno, a tecnologia é apenas mais uma ferramenta para auxílio da aprendizagem e, mesmo que o aluno se prenda por alguns momentos nessa tecnologia, essa por sua vez, foi desenvolvida para que o aluno de alguma forma se desenvolva intelectualmente (VALENTE, 1993; BATISTA e NAPOLITANO, 2003).

Segundo Papert (1988 *apud* VALENTE, 1993), o uso do computador no ensino permite que o aluno tenha maior interesse e motivação, visto que ele está construindo algo de seu interesse, tendo maior envolvimento afetivo, o que torna a aprendizagem mais significativa. Utilizando o computador, ele poderá construir algo com suas próprias mãos, para isso ele terá que pensar nos conceitos na hora de desenvolver a ideia para resolver o problema e a partir das tentativas, dos erros e da reflexão sobre

os resultados obtidos, conseguirá ter um maior desenvolvimento que resulta nas alterações de sua estrutura mental. Essas alterações são feitas conforme o aluno evolui seu raciocínio abstrato durante as tentativas de se resolver um problema.

Em Piaget (1975 *apud* BERNARDI *et al.*, 2007), o conhecimento evolui progressivamente, por meio de estruturas de raciocínio que substituem umas às outras, através de estágios. É no estágio operatório formal que a criança começa a desenvolver seu pensamento como o de um adulto, podendo desenvolver ideias abstratas e iniciar seu raciocínio lógico. Por isso, o incentivo ao desenvolvimento do raciocínio lógico é muito importante nesse estágio, que é desenvolvido entre 12 e 15 anos de idade.

Assim, pode-se notar os benefícios do contato com essas tecnologias educacionais desde mais cedo. Maltempi e Valente (2000) dizem:

Na década de 60 se reconheceu esse potencial e iniciou-se o desenvolvimento da linguagem de programação "Logo" destinada, principalmente, a crianças. Diversos autores relataram resultados positivos no uso do "Logo" em ambientes de aprendizagem de conceitos relacionados ou não a programação.

As crianças e adolescentes que tiverem contato com essas tecnologias desenvolvidas com o intuito de estimular o raciocínio lógico terão futuramente em sua carreira acadêmica, em cursos de tecnologia, ou não, nas quais existam disciplinas que exigem o raciocínio lógico, maior facilidade em resolver problemas, pois anteriormente, em sua adolescência ou infância, sua mente já foi treinada e acostumada a pensar de forma crítica, criativa e lógica.

E como uma forma de potencializar a aprendizagem, pode-se integrar o lúdico às tecnologias para o ensino. Como dito por Dallabona e Mendes (2004), por meio da descoberta e da criatividade, uma criança pode ter uma melhora na sua aprendizagem. Se o ensino é feito de uma forma mais amigável, natural, mas sem deixar de ser planejado e organizado, a criança acaba por absorver os conteúdos de uma forma melhor e consegue aplicar os ensinamentos em diversas situações.

A partir dessa percepção, surgiram diversos projetos voltados ao ensino de lógica de programação para crianças e adolescentes. Os maiores exemplos que podem ser citados no Brasil são o Code Club Brasil, um projeto gratuito que permite que voluntários ensinem presencialmente a programação utilizando jogos, animações e aplicativos (CODE CLUB BRASIL, 2016); a iai? Art, uma instituição que oferece cursos pagos para qualquer faixa etária utilizando softwares e robótica (IAI?

INSTITUTO DE ARTES INTERATIVAS LTDA, 2016), e o SuperGeek que oferece uma solução parecida com a iai? Art (SUPERGEEKS, 2016). Outras soluções oferecem algo mais concreto, por exemplo um robô que executa os movimentos definidos pelo aluno: o Primo.io. Essa solução é dividida em duas partes, a primeira é uma interface na qual o usuário define as ações do robô, e a segunda consiste num robô que executa as ações que foram definidas. Nessa interface, as ações do robô são definidas utilizando-se programação em blocos.

Neste projeto, propõe-se o desenvolvimento de um kit para introduzir a lógica de programação utilizando uma interface, um compilador e um robô. Na interface, a ideia é que o usuário possa montar um roteiro, daí o robô deve executar os comandos e ações a serem organizador e compilados. O compilador deve validar a tradução desses comandos para a linguagem utilizada pelo robô. E por fim, o robô deve executar o roteiro inicialmente montado pelo usuário.

1.1. OBJETIVO

Desenvolver um kit para introduzir lógica de programação para jovens.

1.2. OBJETIVOS ESPECÍFICOS

- Obter uma interface na qual o usuário possa selecionar ações que serão compiladas para um robô;
- Obter um compilador que valide os comandos inseridos pelo usuário;
- Obter um robô que siga os comandos especificados pelo usuário.

1.3. JUSTIFICATIVA

Pesquisas mostram que a aprendizagem de alguns conteúdos pode ser mais efetiva quando ocorre em determinadas idades (PIAGET 1975 *apud* BERNARDI *et al.*, 2007) e a lógica é um desses conteúdos. No entanto, para uma criança ou para um adolescente nem sempre é possível a abstração, logo, o uso de elementos concretos junto com o lúdico pode propiciar à aprendizagem da lógica.

A motivação deste trabalho é fornecer um novo instrumento que, além de divertir, mostra elementos básicos da lógica de programação, no lugar de usar métodos mais tradicionais e mecânicos, resultando no abandono ou na falta de interesse.

Também há formas de aprender programação e sua lógica virtualmente, através de simulações, mas a reprodução dos comandos em um robô, como é o proposto, pode propiciar uma melhor aprendizagem, com elementos intuitivos e divertidos.

2. PROJETO DE SOFTWARE

Para este projeto foi desenvolvida uma aplicação em que o usuário seleciona os comandos para que o robô os execute. A aplicação pode ser dividida em duas partes principais: a interface gráfica e o compilador.

A interface gráfica disponibiliza ao usuário um conjunto específico de comandos que são apresentados em forma de blocos, facilitando assim a interação durante a construção do algoritmo, e a partir desses blocos selecionados é gerado o código que é tratado pelo compilador. O compilador é responsável por transformar o código fonte que a interface gráfica gera, a partir dos blocos que foram selecionados, em instruções que o robô possa interpretar para executar uma determinada ação.

2.1. APLICAÇÃO

Desenvolvida com JavaFX, uma biblioteca da linguagem Java, a aplicação proporciona uma programação amigável e interativa dos comandos que serão executados pelo robô.

As peças utilizadas pelo usuário na programação, são baseadas em blocos de quebra-cabeça e tem o intuito de auxiliar no entendimento, representando visualmente o fluxo de execução do código que será compilado para o robô. Essas peças são divididas em três categorias: **operadores/operandos**, **condições** e **ações**.

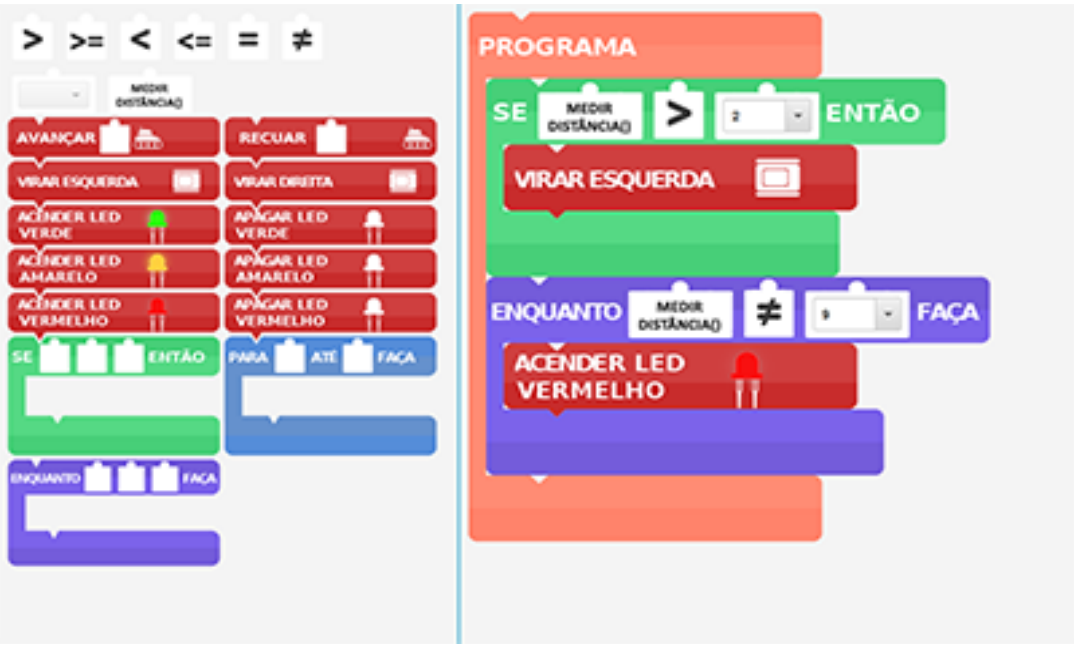
Os operadores/operandos são: maior (>), maior ou igual (>=), menor (<), menor ou igual (<=), igual (=), diferente (=/=), medir distância e números. Eles são utilizados para comparar dois valores e retornar verdadeiro ou falso e esse resultado é combinado com as peças que representam as condições.

As condições SE/ENTÃO (*if*), PARA (*for*) e ENQUANTO (*while*) são combinadas com os operadores e de acordo com o resultado obtido pela comparação (Verdadeiro ou Falso) executa as ações programadas pelo usuário.

As ações são os movimentos que o robô pode realizar. Elas se resumem em virar à esquerda e direita, avançar, recuar e acender as luzes LED vermelha, amarela e verde.

Conforme podemos observar na Figura 1, a interface da aplicação é dividida em duas áreas. A parte localizada à esquerda possui todas as peças que podem ser utilizadas pelo usuário e à direita é localizada a área de montagem dos comandos. Todas as peças podem ser arrastadas para a área de montagem e encaixadas de acordo com a lógica criada.

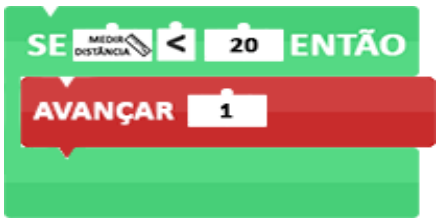
Figura 1 – A interface da aplicação.



2.1.1. Operadores/operandos

Os operadores e operandos são utilizados para testar se uma informação é falsa ou verdadeira, como no exemplo ilustrado na Figura 2, se a distância for menor que 20 centímetros é executada a ação de avançar uma vez.

Figura 2 – Exemplo de utilização dos operadores.



A Figura 3 ilustra as peças que representam os operadores, na primeira linha, e os operandos, na segunda linha de peças, que são utilizados na construção de um código e todas elas são da cor branca.

Todos os elementos funcionam exatamente como na matemática, exceto o MEDIRDISTANCIA(), já que utiliza do sensor ultrassônico, que mede a distância do robô até um obstáculo à sua frente e retorna essa medida em centímetros. Com isso, é possível testar se a medida até o obstáculo corresponde a um número específico e assim programar uma ação.

Figura 3 – Operadores e operandos.

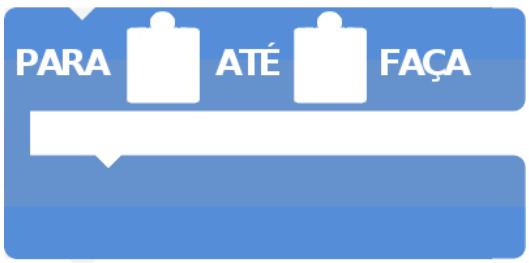
2.1.2. Condições

As condições devem ser utilizadas em conjunto com os operadores e operandos para realizarem testes e uma determinada ação.

Figura 4 – Condição SE/ENTÃO.

Na Figura 4, a peça que representa a condição SE/ENTÃO (*if*), da cor verde, possui espaço para o encaixe de um operador, no centro, e dois operandos, a direita e esquerda, que serão utilizadas em um teste lógico, que caso retorne verdadeiro, realiza as ações contidas dentro da condição.

Figura 5 – Condição PARA.



De cor azul, a peça da condição PARA (*for*), mostrada na Figura 5, possui dois espaços para encaixar apenas operandos, e semelhante à condição SE/ENTÃO realiza um teste lógico, que sendo verdadeiro, executa o conjunto de ações definidas pelo usuário até que esse teste seja falso.

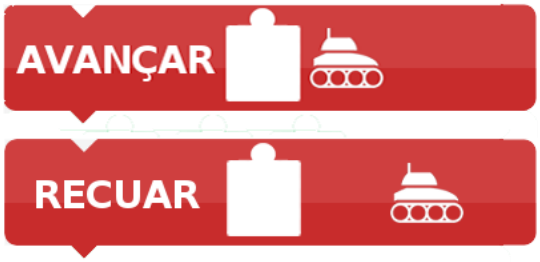
Figura 6 – Condição ENQUANTO.

Na Figura 6, a peça da condição ENQUANTO de cor roxa, que realiza as ações definidas pelo usuário em loop até que o teste lógico, inserido nos três espaços destinados as peças de operadores (centro) e operandos (direita e esquerda), seja falso.

2.1.3. Ações

Todas as peças são da cor vermelha e representam as quatro ações básicas do robô e as de acender as luzes LED. Elas podem ser utilizadas sozinhas ou como resultados de condições criadas pelo usuário.

Figura 7 – Ações AVANÇAR e RECUAR.



As peças AVANÇAR e RECUAR, ilustradas na Figura 7, fazem com que o robô ande um número especifico de passos, definido pelo usuário, para a frente ou para trás. Um passo é uma medida pré-definida de cerca de 16 centímetros, a depender da carga da bateria disponível, ou 2000 milissegundos dos motores em funcionamento.

Figura 8 – Ações VIRAR DIREITA e VIRAR ESQUERDA.

As ações VIRAR DIREITA e VIRAR ESQUERDA, mostradas na Figura 8, fazem com que seja ativado apenas o motor oposto a direção que irá virar. No caso de virar à direta, é ativado apenas o motor esquerdo e o contrário acontece para a direção oposta. O motor fica em funcionamento por 1650 e 1720 milissegundos para virar à direta e esquerda, respectivamente.

Figura 9 – Ações de acender as luzes LED.

Figura 10 – Ações de apagar as luzes LED

São mostradas na Figura 9 as peças que fazem com que o robô acenda suas luzes LED. Cada peça faz com que a luz seja acesa até ser dado o comando de apaga-las, utilizando as peças mostradas na Figura 10.

2.2 COMPILADOR

O computador é uma máquina capaz de executar operações, mas para que ele saiba o que e como executar é necessário programá-lo. Para isso são escritos programas, que nada mais são do que uma sequência de instruções específicas que descrevem uma tarefa a ser realizada. Essas instruções podem manipular, redirecionar ou modificar os dados de maneira lógica e são escritos em uma linguagem. A única linguagem compreendida pelo computador é a chamada “linguagem de máquina”, em que cada instrução é definida por uma sequência numérica e que varia para cada arquitetura. Os códigos escritos nessa linguagem são extremamente difíceis de ser interpretados e escritos pelo homem. Com a finalidade de facilitar a tarefa de programar, foram criadas as linguagens simbólicas, que são conjuntos de palavras e regras facilmente compreendidas, que posteriormente são convertidas para linguagem de máquina (MOGENSEN, 2000).

Já que o homem não entende a linguagem de máquina e o computador não entende as linguagens simbólicas, foi criado também o compilador. Ele é um programa que realiza a tradução de códigos escritos numa linguagem (programa fonte) para outra (programa alvo). Essa tradução é dividida em diversas etapas que visam encontrar erros de escrita e de semântica na aplicação e assim gerar como resultado um arquivo com o código final (BORNAT, 1979).

As instruções contidas nesse arquivo podem ser executadas tanto pelo sistema operacional (neste caso, ele será composto pelas instruções em linguagem de máquina), quanto por outras aplicações que rodam sob o sistema operacional (neste caso, ele é composto por código interpretável). Um exemplo desse último caso é a plataforma Java em que o compilador gera um arquivo com os bytecodes que são interpretados pela máquina virtual Java (JVM).

Um compilador efetua a tradução em diversas fases que podem ser ou não executadas em sequência. A cada fase, o compilador gera uma representação que é utilizada como entrada para a fase seguinte.

As fases de um compilador típico são: análise léxica, análise sintática, análise semântica e geradora de código. A fase geradora de código também pode ser

precedida pelas fases de geração de código intermediário e de otimização de código (AHO *et al.*, 1986).

2.2.1. Análise léxica

Nesta primeira fase é feita a leitura do programa fonte de forma sequencial, é lido caractere por caractere para determinar se uma sequência de caracteres (token) é reconhecida pelo compilador, ou seja, se ela possui algum significado. Os caracteres que o compilador consegue reconhecer são definidos pelo seu alfabeto (conjunto finito de caracteres). Quando um token é reconhecido, é feita sua classificação para determinar seu tipo na gramática da linguagem, ou seja, classifica o token de acordo com o que ele representa (APPEL, 2002). Por exemplo, pode-se definir se um token representa um número inteiro ou decimal.

Os tipos de token mais comuns nas linguagens de programação são:

- NUM: representa um valor inteiro;
- REAL: representa um valor decimal;
- ID: representa um identificador.

Para poder efetuar a leitura e classificação dos tokens, o compilador utiliza um autômato finito determinístico que consiga ler todas as possíveis sequências de caracteres que os tokens possam ter. A construção desse autômato será demonstrada posteriormente na seção 2.3.1.

Ao iniciar a leitura do programa fonte, o analisador léxico parte do estado inicial do autômato e a cada caractere lido é verificado se, para o estado atual, existe uma transição para outro estado que é definida a partir do caractere lido. Se existir uma transição válida, o analisador salva o estado de destino daquela transição e então efetua a leitura de mais um caractere; se não existir, então o compilador verifica se o último estado em que passou representa um token. Se for um token então é armazenado numa lista que será enviada para a próxima fase, se não é apresentado um erro para o programador. Esse processo é repetido até que o fim da entrada seja atingido (AHO *et al.*, 1986; MOGENSEN, 2000).

2.2.2. Análise sintática

Nesta fase, também chamada de análise gramatical, é verificado se a sequência de tokens recebida da fase de análise léxica contém um programa válido.

Toda linguagem de programação possui uma gramática livre de contexto que define as regras dessa linguagem. Tais regras definem a sintaxe da linguagem, por exemplo, define a forma de uma declaração de variável, declaração de um tipo, um laço de repetição, entre outros. Portanto, a utilização da gramática permite definir, de maneira abstrata, todas as regras de sua sintaxe e suas utilizações (AHO *et al.*, 1986; MOGENSEN, 2000).

A partir desta gramática é construído, também, um autômato finito determinístico que, por meio da leitura de cada token da lista gerada na análise léxica, pode identificar se aquela sequência é um programa válido, ou seja, se ela pode ser gerada a partir da gramática da linguagem (AHO *et al.*, 1986; MOGENSEN, 2000).

A execução do autômato da análise sintática é feita de forma similar ao autômato da análise léxica, mas no lugar de caracteres são utilizados tokens e o autômato utiliza duas pilhas para poder armazenar os estados percorridos e os tokens lidos. No fim do processo, é gerada uma árvore gramatical que representa o programa na forma válida. Nessa árvore, os nós são as variáveis e as folhas são os terminais da gramática, ambos serão discutidos posteriormente.

Se ocorrer algum erro durante esse processo é possível exibir ao programador alertas com maiores detalhes sobre o erro, pois a partir da gramática é possível determinar quais seriam os próximos tokens esperados na sintaxe, por exemplo, numa declaração de uma constante é possível informar ao programador que é obrigatório a definição do valor daquela variável no momento de sua criação.

2.2.3. Análise semântica

Nesta fase é verificado se existem erros semânticos no programa fonte, ou seja, se a sintaxe do programa fonte não viola as especificações da linguagem e se são seguidas conforme definida (AHO *et al.*, 1986). Por exemplo, se uma variável é declarada com o tipo inteiro então é verificado em todos os lugares que é atribuído um valor para ela, se esse valor é do tipo inteiro.

O analisador semântico efetua a análise da árvore gramatical que foi retornada pelo analisador sintático. Algumas tarefas de validação do analisador semântico são: declarações de identificadores, verificação de tipos, definições de métodos, escopos de variáveis, entre outros. A partir da raiz, são feitos passeios na árvore efetuando criações e validações dos atributos dos elementos de acordo com as regras semânticas. Durante esses passeios, a árvore sintática também é alterada e

detalhada, por exemplo, quando é preciso trocar uma operação por causa dos tipos, ou inserir um nó com operação de conversão, que foi definido na especificação como automático para o programador. Também é construída a tabela de símbolos em que são registrados os nomes de variáveis e funções referenciadas durante o programa, e nos locais em que são encontradas tais referências é colocada uma referência para o registro da tabela. Assim, ao final será retornada a árvore gramatical abstrata (AHO *et al.*, 1986).

2.2.4. Gerador de código

Nesta fase, após a verificação de todos os erros e efetuadas as devidas mudanças no programa fonte, é gerado o programa alvo com os códigos específicos para a plataforma ou linguagem.

A partir da árvore sintática abstrata, para cada nó e folha da árvore é gerado o código que faça a mesma operação ou que represente o mesmo valor. Dependendo da arquitetura, também são definidos os registradores, que cada variável ocupará na Unidade Central de Processamento (CPU), ou sua posição relativa na memória.

O programa alvo reflete, mediante instruções mais próximas à linguagem de máquina, os comandos do código-fonte. Como cada máquina ou cada plataforma possui um conjunto diferente de instruções e de meios de acesso ao sistema operacional, em geral é necessário que exista um gerador de código distinto para cada plataforma.

2.3. A LINGUAGEM CONSTRUÍDA E SEU COMPILADOR

Neste projeto foi criada uma linguagem de programação própria para a qual apenas a sintaxe básica deve ser interpretada pelo robô. Essa sintaxe foi baseada em alguns elementos da linguagem Java e é composta por: laços de repetição *while* e *for*; controle de fluxo *if* e *else*; pelas funções padrão *viraEsquerda*, *viraDireita*, *avanca*, *retrocede*, *acendeLedVerde*, *acendeLedVermelho*, *acendeLedAmarelo*, *apagaLedVerde*, *apagaLedAmarelo*, *apagaLedVermelho* e *medeDistancia*; pelos operadores relacionais >=, >, <=, <, ==, *!*=; operadores lógicos *and* e *or*; e pelos operadores aritméticos +, -, * e /.

Essa linguagem foi construída utilizando gramática livre de contexto e um analisador sintático LALR(1), e o alfabeto do compilador é composto pela união do

alfabeto ocidental, dos números de 0 a 9, e dos caracteres especiais: *(*, *)*, *{*, *}*, *;*, >, <, =, *!*, +, -, */*, * e ponto final (.).

Foi utilizado um analisador LALR(1) devido à sua facilidade de implementação e sua vantagem na resolução de conflitos shift/reduce e reduce/reduce. Um desses conflitos pode ser mostrador nas seguintes regras de um if else: *IF → if (EXP_BOOL) { STMT_LIST }* e *IF → if (EXP_BOOL) { STMT_LIST } else { STMT_LIST }*. Um analisador que não utilize um lookahead (ou seja, que não leia o primeiro item da lista de tokens para tomar uma decisão de ação), ao chegar no primeiro terminal *}*, ele não sabe se deve efetuar uma redução, pois terminou a primeira regra do *IF*, ou se deve continuar com os empilhamentos, pois o próximo token *else* indica a continuação da segunda regra do *IF*. Um analisador LALR(1), utilizando um lookahead, ao ler o token *else*, efetuará um empilhamento.

Uma gramática livre de contexto é definida por 4-tuplas:

$$G = (T, V, P, S)$$

Onde:

- Terminais (*T*): conjunto de tokens, conhecido como terminais;
- Não-terminais ou variáveis (*V*): conjunto de símbolos que podem ser substituídos por uma de suas produções;
- Regras de produção (*P*): conjunto de regras no formato: $\alpha \rightarrow \beta_1 ... \beta_n$, onde $\alpha \in V$ e $\beta_1 ... \beta_n \in (T \cup V)$;
- Símbolo inicial (*S*): variável inicial da gramática, onde $S \in V$.

A Figura 11 mostra a gramática da linguagem de programação construída neste projeto:

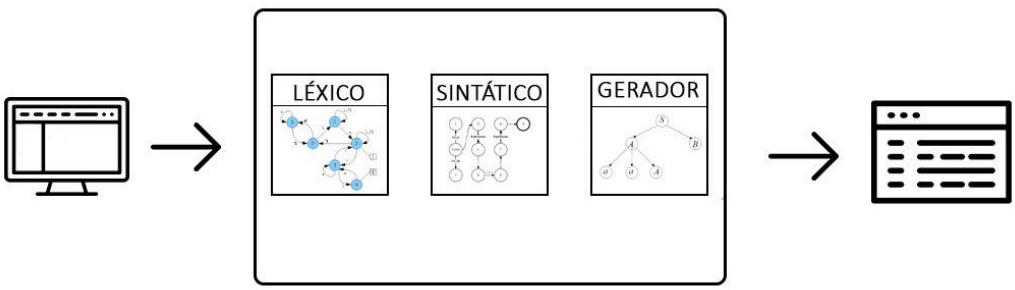
Figura 11 – Gramática da linguagem.

```
PGM → program { STMT_LIST } $
STMT_LIST → STMT STMT_LIST | ∈
STMT → CTRL_FLUX | FUNCTION ;
CTRL_FLUX → IF | FOR | WHILE
IF → if ( EXP_BOOL ) { STMT_LIST } ELSE
ELSE → else { STMT_LIST } | ∈
FOR → for (i = EXP ; i OP_REL EXP ; i = i OP_ARIT EXP ) { STMT_LIST }
WHILE → while ( EXP_BOOL ) { STMT_LIST }
FUNCTION → id ( PARAMS )
PARAMS → VAR | ∈
EXP_BOOL → EXP OP_REL EXP MORE_EXP_BOOL
MORE_EXP_BOOL → OP_BOOL EXP_BOOL | ∈
EXP → TERM MORE_TERM
MORE_TERM → OP_ARIT_LO TERM MORE_TERM | ∈
TERM → FACTOR MORE_FACTOR
MORE_FACTOR → OP_ARIT_HI FACTOR MORE_FACTOR | ∈
FACTOR → VAR | FUNCTION
OP_REL → < | <= | > | >= | == | !=
OP_ARIT → OP_ARIT_LO | OP_ARIT_HI
OP_ARIT_LO → + | -
OP_ARIT_HI → * | /
OP_BOOL → and | or
VAR → num | real
```

Cada produção na gramática discutida é composta por uma variável (palavras em maiúscula), que pode definir uma ou mais regras, e pela própria regra (conteúdo à direita do →) que pode ser composta por uma ou mais variáveis ou terminais (palavras em minúsculo) ou pode ser vazia (símbolo ∈).

O compilador foi construído contendo apenas três componentes: analisador léxico, analisador sintático e gerador de código. Esses componentes foram escolhidos pois apenas eles são necessários para efetuar a tradução do código gerado pela interface para o código que é interpretado pelo robô. Na Figura 12, mostra-se a estrutura do compilador.

Figura 12 – Estrutura do compilador.



Após a interface gráfica enviar o código gerado a partir dos blocos selecionados, o compilador efetua a análise em cada processo, gerando as saídas necessárias para a fase seguinte. No final, é gerado o código para interpretação do robô.

2.3.1. Analisador léxico

O analisador léxico do compilador é, basicamente, um autômato finito determinístico (AFD) que foi construído a partir da união dos AFD de cada token. O AFD de cada token é criado a partir da conversão do autômato finito não-determinístico (AFND) que, por sua vez, é criado a partir da expressão regular que representa as possíveis sequências de caracteres daquele token.

Como exemplo, será feita a construção de um AFD para o token ID, que representa um identificador para o compilador. Primeiro será criada a expressão regular, depois será gerado o AFND e, então, criado o AFD.

Uma expressão regular é uma notação que permite definir, precisamente, um conjunto discreto de sequências de caracteres (*string*) que seguem um determinado padrão (por isso a expressão regular é utilizada apenas nesta parte do compilador, visto que essa não possui recursos suficientes para expressar a recursividade necessária, como exemplo IF aninhados) (AHO *et al*., 1986), os caracteres dessa *string* são definidos pelo alfabeto do compilador criado. Esse alfabeto é o composto por caracteres alfanuméricos, letras de A a Z, maiúscula e minúscula, e os dígitos de 0 a 9.

A expressão regular do token ID deve expressar uma sequência de caracteres em que o primeiro caractere é uma letra (maiúscula ou minúscula) seguida ou não de letras e/ou números. Com isso, tem-se a expressão:

*[a-zA-Z][a-zA-Z0-9]**

Onde:

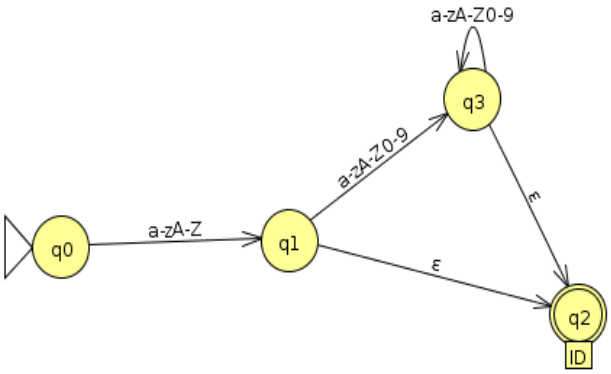
- [a-zA-Z]: permite representar qualquer letra, maiúscula ou minúscula;
- [a-zA-Z0-9]*: permite representar qualquer sequência de caracteres alfanumérico, maiúscula ou minúscula, vazia ou não.

A partir da expressão regular do token, aplica-se o algoritmo e Thompson para poder gerar o AFND que consiga reconhecer as *strings* naquele padrão (AHO *et al.*, 1986).

Para facilitar a leitura e a visualização dos autômatos nas imagens que serão mostrados neste trabalho, escolheu-se manter as transições, que tenham os mesmos estados de origem e destino, agrupadas em uma única transição e rotulada com a expressão regular da união de todos os rótulos. Por exemplo, na Figura 13, a transição do estado *q0* para o estado *q1* é a união das 52 transições originais, uma para cada letra maiúscula e minúscula do alfabeto de A a Z.

Na Figura 13, tem-se o AFND que foi criado a partir da expressão regular do token ID.

Figura 13 – AFND do token ID.



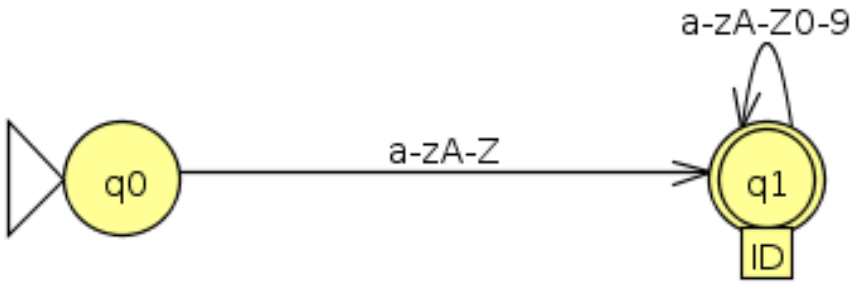
O autômato da Figura 13 possui 4 estados, um inicial (estado *q0*) e um final (estado *q2*) e 5 transições. Por ser um AFND, é permitida a existência de transições vazias (transições rotuladas com épsilon – caractere ε), ou seja, transições que mudam o estado do autômato sem consumir nenhum caractere da entrada (stream de caractere contidos no programa fonte). Se, no fim da leitura de toda a entrada, o autômato estiver no estado *q2*, a entrada é considerada válida e o token ID é retornado. Considerando que os estados *q1* e *q3* possuem transições vazias para o estado *q2*, se, após a leitura de toda a entrada, o autômato estiver no estado *q1* ou no estado *q3* a entrada também será considerada como válida.

A execução do autômato se inicia no estado inicial *q0* (que possui um triângulo ao lado que indica estado inicial), efetua a leitura de um caractere da entrada e verifica se existe alguma transição rotulada com aquele determinado caractere lido. Se for uma letra maiúscula ou minúscula, conforme rotulada pela expressão *"a-zA-Z"* na transição, então efetua a mudança para o estado *q1*. Se não, retorna um erro

informando caractere inválido. No estado *q1* tem-se uma transição, rotulada com a expressão *"**a**-zA-Z0-9"*, para o estado *q3*, que consome qualquer letra ou número. E, também, tem-se uma transição vazia para o estado *q2*. Portado, se autômato estiver no estado *q1*, ele pode mudar para o estado *q2* sem efetuar nenhuma leitura da entrada ou mudar para o estado *q3* efetuando a leitura de um caractere alfanumérico. E, por fim, no estado *q3* temos duas transições, uma transição vazia para o estado *q2* e uma transição que consome caractere alfanumérico para o estado *q3* (um ciclo).

A partir do AFND da Figura 13, pode-se obter o AFD da Figura 14, efetuando uma conversão utilizando as operações de fechamento-ε e de movimento (AHO *et al.*, 1986; APPEL, 2002).

Figura 14 – AFD gerado a partir do AFND da Figura 13.



O autômato da Figura 14 é um AFD, pois não possui nenhuma transição vazia e todos os estados possuem apenas uma transição para um determinado rótulo ou caractere. A execução do autômato é parecida com a execução do AFND da Figura 13, exceto que, para cada transição, obrigatoriamente será necessário a leitura de um caractere da entrada. Se existir uma transição com tal rótulo então a mudança para o estado é feita, se não é lançado um erro.

Como exemplo, na Tabela 1 é mostrada a simulação do autômato.

Tabela 1 – Execução do AFD da Figura 14.

Estado do autômato	**Caracteres lidos**	***Stream* da Entrada**	**Ação**
q0		"avanca"	
q1	'a'	"vanca"	Consome 'a' e muda o estado para q1
q1	'a', 'v'	"anca"	Consome 'v' e permanece no estado q1
q1	'a', 'v', 'a'	"nca"	Consome 'a' e permanece no estado q1
q1	'a', 'v', 'a', 'n'	"ca"	Consome 'n' e permanece no estado q1

Estado do autômato	Caracteres lidos	*Stream* da Entrada	Ação
q1	‘a’, ‘v’, ‘a’, ‘n’, ‘c’	“a”	Consome ‘c’ e permanece no estado q1
q1	‘a’, ‘v’, ‘a’, ‘n’, ‘c’, ‘a’	“”	Consome ‘a’ e permanece no estado q1
q1	‘a’, ‘v’, ‘a’, ‘n’, ‘c’, ‘a’	“”	Verifica que acabou a entrada, estado em *q1* então retorna o token ID contendo a entrada lida (“avanca”)

Na Tabela 1, mostra-se a simulação do AFD do token ID para a entrada “avanca”, após a leitura de cada caractere da entrada, o autômato finaliza a execução no estado *q1*, que é final, então é retornado o token ID com um atributo com o valor dos caracteres lidos, no exemplo tem-se “avanca”.

Na Tabela 2, tem-se os tokens e suas respectivas expressões regulares que foram criadas para a linguagem.

Tabela 2 – Tokens da linguagem.

Token	Expressão regular	Descrição
SKIP	(“ “ | \n | \t)+	Sequência de espaços em brancos, tabulações ou quebra de linhas
COUNTER	i	Contador do laço de repetição *for*
SECO	;	Ponto-e-vírgula
IF	if	Controle de fluxo *if*
ELSE	else	Controle de fluxo *else*
FOR	for	Laço de repetição *for*
WHILE	while	Laço de repetição *while*
PROGRAM	program	Declaração de um programa na linguagem fonte
AND	and	Operador lógico *and*
OR	or	Operador lógico or
ID	[a-zA-Z][a-zA-Z0-9]*	Identificador
NUM	[0-9]+	Número inteiro
REAL	[0-9]+”.”[0-9]+	Número decimal
LPAR	“(“	Abertura de parêntese
RPAR	“)”	Fechamento de parêntese
LCURB	{	Abertura de chaves
RCURB	}	Fechamento de chaves
BINOP_ADD	“+”	Operador binário de soma
BINOP_SUB	“-”	Operador binário de subtração
BINOP_MUL	“*”	Operador binário de multiplicação
BINOP_DIV	“/”	Operador binário de divisão
OP_ASSIGN	=	Operador de atribuição
RELOP_EQ	==	Operador relacional de igualdade
RELOP_NEQ	!=	Operador relacional de desigualdade
RELOP_GT	>	Operador relacional maior
RELOP_GTE	>=	Operador relacional maior ou igual

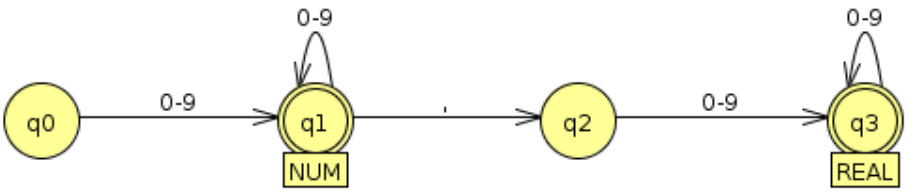
Token	Expressão regular	Descrição
RELOP_LT	<	Operador relacional menor
RELOP_LTE	<=	Operador relacional menor ou igual

Tendo definido os tokens da linguagem e suas expressões regulares, foram gerados os AFND de cada expressão regular utilizando o algoritmo de Thompson (AHO *et al.*, 1986). A partir dos AFNDs, foram gerados os AFDs de cada token utilizando as operações de fechamento-ε e de movimento (AHO *et al.*, 1986; APPEL, 2002). E, por fim, tendo os AFD dos tokens, foi feita a união deles.

Se, durante a união, houver algum estado que possa ter duas transições com o mesmo rótulo então é utilizado o estado do token com maior prioridade. Por exemplo, a partir do estado inicial, lendo um símbolo numérico (dígito entre 0 e 9), é possível avançar para o primeiro estado dos AFD do token NUM e REAL, pois apenas esses AFD consomem símbolo numéricos. Nesse caso, define-se qual token tem mais prioridade, ou seja, qual token que será utilizado caso o analisador pare a execução naquele estado.

Na Figura 15, mostra-se o resultado da união dos AFD dos tokens NUM e REAL.

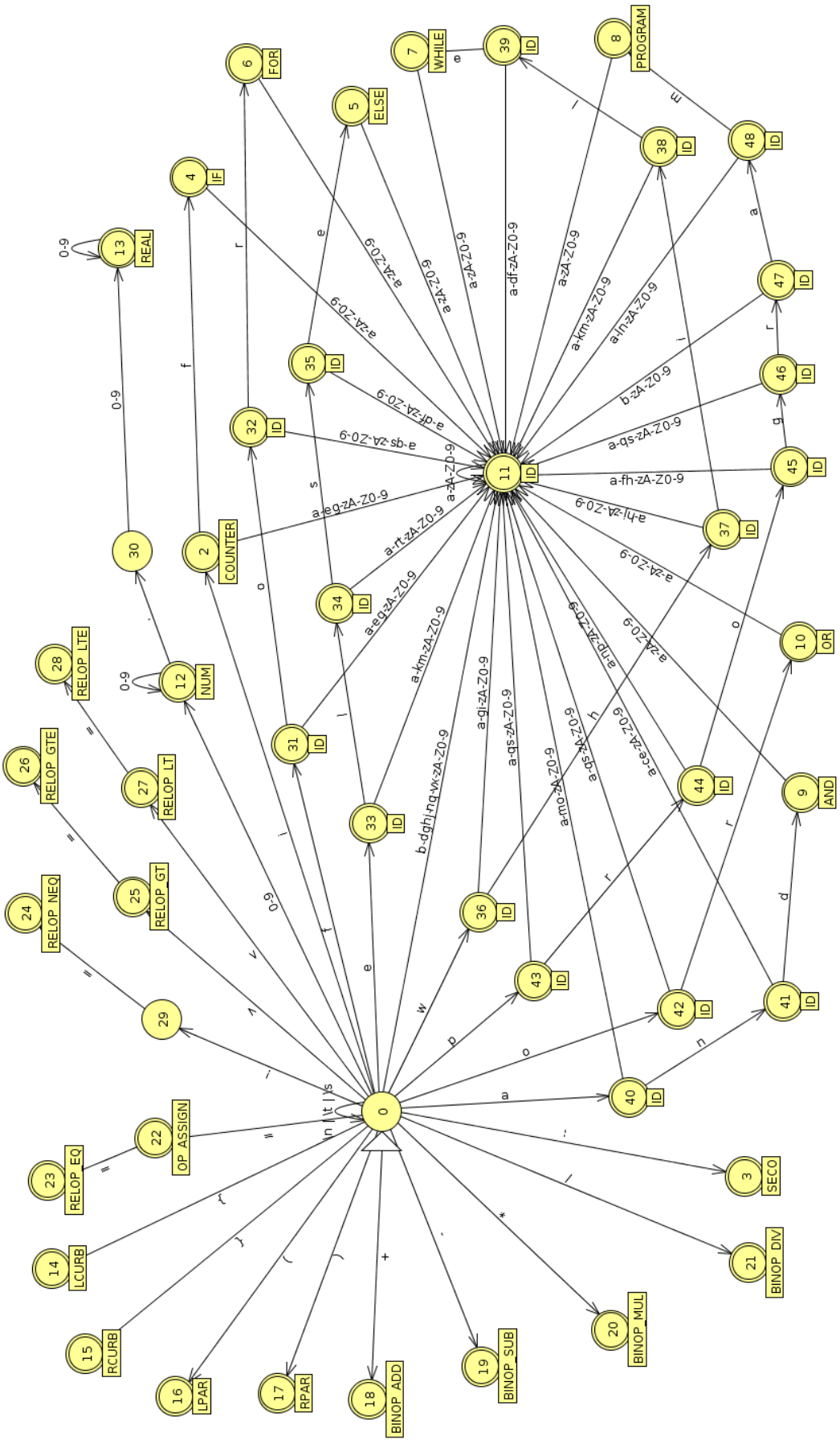
Figura 15 – Resultado da união..



Como mostrado na Figura 15, deu-se prioridade para os números inteiros.

O resultado da união de todos os AFD dos tokens pode ser visto na Figura 16. Em cada estado final do AFD, foi colocado um rótulo com o token que é retornado quando o autômato finaliza a leitura de uma entrada naquele estado.

Figura 16 – AFD do analisador léxico.



O analisador léxico inicia sua execução no estado *0*, a cada símbolo lido da entrada é verificado se existe uma transição rotulada com ele. Se houver, o autômato avança para o estado de destino determinado pela transição. Se não, é verificado se o último estado que o autômato esteve é final, se for então retorna token vinculado naquele estado. Se não for final, então é apresentado um erro.

Como exemplo, considerando que na entrada tem-se o código “if (x>3)”, o autômato, a partir do estado inicial *0*, efetua leitura do caractere *'i'* e avança para o estado *2* (se o autômato parasse no próximo caractere seria retornado o token *COUNTER*), depois efetua a leitura do caractere *'f'* e avança para o estado *4*. Na leitura do próximo caractere da entrada, espaço em branco (espaço após o *“if”*), é verificado que não existe uma transição a partir do estado *2* em que o rótulo seja um espaço em branco. Então o analisador para a leitura e verifica se o estado em que parou é um estado final. O último estado foi o *2*, que é final, sendo assim é retornado o token *IF*.

Na próxima chamada do analisador ele continuará do último ponto em que parou da chamada anterior, o caractere de espaço, e iniciará no estado *0*. No estado *0* temos uma transição que termina nele mesmo quando o caractere é um espaço, isso significa que ele irá consumir a entrada (o espaço) sem mudar para outro estado. Na próxima chamada, será lido o caractere *'('* e após a verificação é feito o avanço para o estado *16*. Na próxima leitura, verifica-se que não existe uma transição rotulada com o caractere *'x'* a partir do estado *16*. Então, após a verificação do último estado, é retornado o token *LPAR*.

Após a execução da leitura de toda a entrada, tem-se a seguinte sequência de tokens: *IF*, *LPAR*, *ID(“x”)*, *RELOP_GT, NUM(3)*, *RPAR*. Nota-se que, com os tokens *ID* e *NUM*, foram retornados os símbolos que foram lidos, pois essa entrada pode ser um valor, nome de variável ou função, então é necessário saber exatamente qual era seu valor.

Internamente, no programa, o autômato é representado por uma matriz em que as linhas são os seus estados, enumerados a partir de 0, e as colunas são os símbolos do alfabeto. As células contêm o número da linha dos estados de destino daquela transição (rotulada pela coluna) para o estado atual (indicado pela linha), se não houver uma transição válida então é preenchido com um valor negativo (-1).

2.2.2. Analisador sintático

O analisador sintático do compilador também é um AFD em que seus estados estão as possíveis regras em que se está analisando e as transições são rotuladas com as variáveis e terminais da gramática.

A partir da gramática são criadas coleções de regras, em que cada conjunto de regras é agrupado em itens, sendo assim, cada item deve possuir um conjunto de regras com um marcador da posição atual da regra. A posição atual, geralmente, é indicada por um ponto de posição (•).

Por exemplo, a partir da regra de produção S → N podemos ter os seguintes itens com uma regra cada: I_0 *(S → • N)* e I_1 *(S → N •)*.

A partir dessa coleção de itens, é criado um AFD em que os estados são os próprios itens e os rótulos das transições são as variáveis (V) e terminais (T) da gramática.

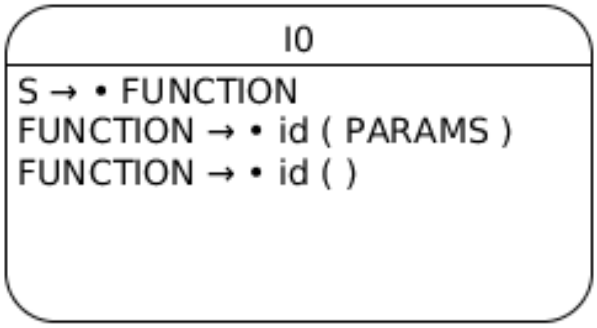
Como exemplo, será criado um AFD para a parte da gramática que define uma chamada de função. Para o exemplo, foi criado uma regra de produção que não existe na gramática original da linguagem, a regra *S → FUNCTION* foi criada apenas para ter uma regra inicial. As regras dessa gramática de exemplo são:

S → FUNCTION
FUNCTION → id (PARAMS)
FUNCTION → id ( )
PARAMS → VAR
VAR → num
VAR → real

Para iniciar a construção, é criado um primeiro item I_0 que inicia a partir da regra inicial. Então, no exemplo, I_0 terá, inicialmente, apenas a regra *S → FUNCTION*. Nessa regra é colocado um ponto de posição em seu início, assim, tem-se *S → • FUNCTION*. Toda vez que temos um ponto de posição antes de uma variável da gramática, são adicionados ao item todas as possíveis derivações dessa variável (no nosso exemplo, a variável é *FUNCTION*) e também é incluído um ponto de posição no início de cada regra adicionada.

Com isso, o item I_0 será composta pelas regras da Figura 17.

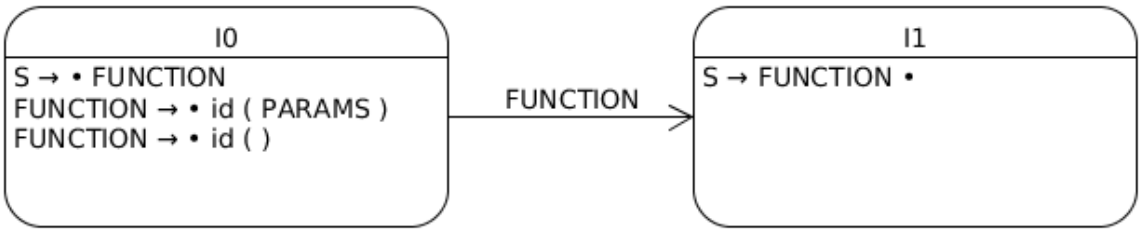
Figura 17 – Item I_0.



A partir desse item inicial, são criados todos os outros itens utilizando avanços do ponto de posição através das regras. Em cada regra do item, para cada variável e terminal que é precedido pelo ponto de posição, é criado uma nova transição para um novo item. No item I_0, na primeira regra tem-se uma nova transição com o rótulo *FUNCTION*, e neste novo item de destino, são adicionadas as regras do item I_0 que são precedidos pelo ponto de posição (no exemplo tem-se apenas uma regra). Nessas regras adicionadas, o ponto de posição é avançado para a próxima posição na regra, e se, nessa nova posição, a próxima posição for uma variável, então são copiadas todas as possíveis derivações.

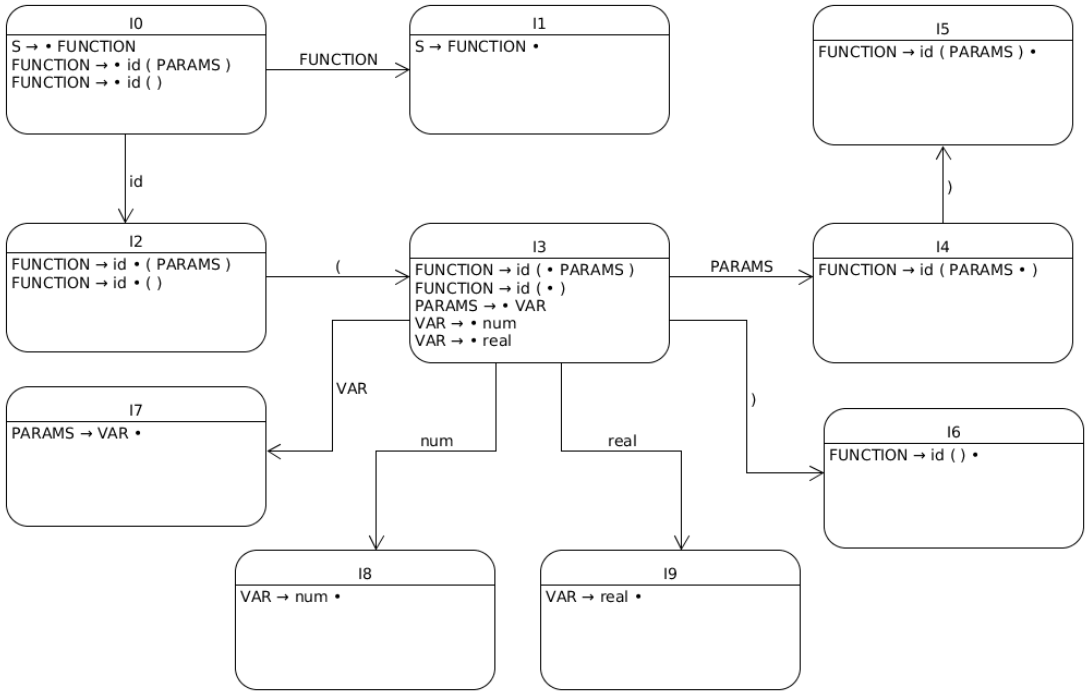
Na Figura 18, mostra-se a coleção com o novo item I_1 criado a partir da transição rotulada com *FUNCTION*.

Figura 18 – Item I_1.



Esse processo é repetido até que não sejam criados novos itens na coleção. A Figura 19 mostra a coleção de itens da gramática de exemplo.

Figura 19 – Coleção de itens da gramática de exemplo.



A coleção de itens da Figura 19 é convertida num AFD em que cada item é um estado, o item I_0 é inicial e o item I_1 é final, pois ele contém o ponto de posição no fim da regra inicial.

A partir desse AFD é construída a tabela de transições, mostrada na Tabela 3, que contém, inicialmente, apenas as transições entre os itens da coleção. Para cada item é criada uma linha na tabela e para cada transição é criada uma coluna, e na célula conterá o item de destino.

Tabela 3 – Tabela de transições da Figura 19.

Item	FUNCTION	Id	(	PARAMS	)	VAR	num	real
0	1	2						
1								
2			3					
3				4	6	7	8	9
4					5			
5								
6								
7								
8								
9								

Depois de gerada a tabela de transições, são calculados os conjuntos First e Follow das regras de produção da gramática. O conjunto First contém os primeiros terminais (tokens) possíveis que uma determinada variável pode derivar. E o conjunto Follow contém os primeiros terminais possíveis que uma variável irá preceder, ou seja, os terminais que podem suceder aquela variável na gramática (AHO *et al.*, 1986; APPEL, 2002).

Para a gramática de exemplo, os conjuntos First e Follow são mostrados na Tabela 4.

Tabela 4 – Conjunto First e Follow das variáveis.

Variável	First	Follow
S	id	$
FUNCTION	id	$
PARAMS	num, real	)
VAR	num, real	)

O cifrão ($) indica fim de arquivo.

Após obter os conjuntos First e Follow, é criada a tabela Ação/Desvio. Essa tabela será utilizada pelo analisador sintático para poder verificar se a sequência de tokens recebida é ou não valida. A tabela será dividida em duas seções, a seção Ação e a seção Desvio. A seção de Ação conterá as transições de mudança (representadas na tabela por shift e um estado de destino, exemplo s1) e redução (representadas por *reduction* e um índice de uma regra – r1, em que 1 seria a primeira regra de produção da gramática), e a seção de desvio conterá as transições de desvio (representadas por goto e um estado de destino – g1).

Quando o analisador está num estado e o token da entrada o leva a uma ação de mudança, ele adiciona o estado na pilha de estados percorridos e efetua a mudança para o estado indicado na tabela. Se o token o leva a uma ação de redução, então será removido da pilha a mesma quantidade de elementos da regra indicada na tabela, por exemplo, se na tabela indica a regra 1 (no exemplo, *S → FUNCTION*) então será removido da pilha 1 estado, pois a regra contém apenas um elemento. Após a remoção, o estado no topo da pilha será recuperado, e a partir dele será verificado, na seção Desvio da tabela, qual o estado de destino para a transição rotulada com a variável da regra que foi reduzida.

A tabela Ação/Desvio, mostrada na Tabela 5, é construída criando uma linha para cada item da coleção de itens. Para cada transição rotulada com um terminal, é criada uma coluna na seção Ação com ação de mudança para o item de destino. E para as transições rotuladas com uma variável, é criada uma coluna na seção de Desvio com ação de desvio para o item de destino.

Se, em um item, houver alguma regra em que o ponto de posição estiver no fim dela, ou seja, já tiver percorrido toda a regra (por exemplo, *VAR → real* •), é criada uma coluna, na seção Ação da tabela, para cada elemento do conjunto Follow, daquela regra percorrida, com ação de redução. Se já existir uma coluna para um determinado elemento então nada será feito.

E, por fim, é incluída uma coluna na seção Ação da tabela que é utilizada para o fim do arquivo, assim se estiver no estado final e o símbolo de fim de arquivo (geralmente é utilizado cifrão - $) for o próximo da entrada significa que a sequência de tokens foi aceita, ou seja, a partir da gramática é possível gerar aquela entrada.

Para facilitar a construção da tabela, as regras da gramática são enumeradas para serem referenciadas nas ações de reduções.

1. *S → FUNCTION*
2. *FUNCTION → id (PARAMS)*
3. *FUNCTION → id ( )*
4. *PARAMS → VAR*
5. *VAR → num*
6. *VAR → real*

Tabela 5 – Tabela Ação/Desvio.

	Ação						**Desvio**		
Item	**$**	**Id**	**(**	**)**	**num**	**real**	**FUNCTION**	**PARAMS**	**VAR**
0		s2					g1		
1	Aceito								
2			s3						
3				s6	s8	s9		g4	g7
4				s5					
5	r2								
6	r3								
7				r4					
8				r5					
9				r6					

Quando o analisador sintático, estando no estado 1, efetua a leitura do caractere de fim de arquivo (cifrão) então retorna que a palavra foi aceita, juntamente com a árvore gramatical para aquela sequência.

Para a construção do compilador foi utilizada a gramática, na forma mais extensa, mostrada no Quadro 1, e com a enumeração necessária para a construção da tabela de Ação/Desvio.

Quadro 1 – Gramática da linguagem

1. *PGM → program { STMT_LIST }*
2. *STMT_LIST → STMT STMT_LIST*
3. *STMT_LIST → STMT*
4. *STMT → CTRL_FLUX*
5. *STMT → FUNCTION ;*
6. *CTRL_FLUX → IF*
7. *CTRL_FLUX → FOR*
8. *CTRL_FLUX → WHILE*
9. *IF → if (EXP_BOOL) { STMT_LIST } ELSE*
10. *IF → if (EXP_BOOL) { STMT_LIST }*
11. *ELSE → else { STMT_LIST }*
12. *FOR → for (i = EXP ; i OP_REL EXP ; i = i OP_ARIT EXP) { STMT_LIST }*
13. *WHILE → while (EXP_BOOL) { STMT_LIST }*
14. *FUNCTION → id (PARAMS)*
15. *FUNCTION → id ( )*
16. *PARAMS → VAR*
17. *EXP_BOOL → EXP OP_REL EXP MORE_EXP_BOOL*
18. *EXP_BOOL → EXP OP_REL EXP*
19. *MORE_EXP_BOOL → OP_BOOL EXP_BOOL*
20. *EXP → TERM MORE_TERM*
21. *EXP → TERM*
22. *MORE_TERM → OP_ARIT_LO TERM*
23. *MORE_TERM → OP_ARIT_LO TERM MORE_TERM*
24. *TERM → FACTOR MORE_FACTOR*
25. *TERM → FACTOR*
26. *MORE_FACTOR → OP_ARIT_HI FACTOR MORE_FACTOR*
27. *MORE_FACTOR → OP_ARIT_HI FACTOR*
28. *FACTOR → VAR*
29. *FACTOR → FUNCTION*
30. *OP_ARIT → OP_ARIT_LO*
31. *OP_ARIT → OP_ARIT_HI*
32. *OP_REL → RELOP_LT*
33. *OP_REL → RELOP_LTE*
34. *OP_REL → RELOP_GT*
35. *OP_REL → RELOP_GTE*
36. *OP_REL → RELOP_EQ*
37. *OP_REL → RELOP_NEQ*
38. *OP_ARIT_LO → +*
39. *OP_ARIT_LO → -*
40. *OP_ARIT_HI → **
41. *OP_ARIT_HI → /*
42. *OP_BOOL → and*
43. *OP_BOOL → or*
44. *VAR → num*
45. *VAR → real*

A partir dessa gramática foi construída a coleção de itens do APÊNDICE A. A coleção de itens final contém 92 itens, sendo que o item 0 será o estado inicial e o item 1 será o estado final.

Tendo construída a coleção de itens, é possível gerar a tabela de Ação/Desvio que será utilizada pelo autômato. A tabela contém 92 linhas e 52 colunas, das quais 28 colunas são da seção Ação e 24 são da seção Desvio. As Tabelas 6 e 7 mostram as seções Ação e Desvio, respectivamente.

Tabela 6 – Seção Ação

Item	$	I	;	if	else	for	while	program	and	or	id	num	real	{	}	(	)	+	-	*	/	=	==	!=	>	>=	<	<=
0								s2																					
1	Aceito																												
2														s3															
3				s14		s15	s16				s17																		
4															s5														
5	r1																												
6				s14		s15	s16				s17				r3														
7															r2														
8				r4		r4	r4				r4				r4														
9			s10																										
10				r5		r5	r5				r5				r5														
11				r6		r6	r6				r6				r6														
12				r7		r7	r7				r7				r7														
13				r8		r8	r8				r8				r8														
14																s24													
15																s25													
16																s26													
17																s18													
18												s22	s23				s19												
19			r15						r15	r15							r15	r15	r15	r15	r15		r15	r15	r15	r15	r15	r15	r15
20																	s88												
21																	r16												
22			r44						r44	r44							r44	r44	r44	r44	r44		r44	r44	r44	r44	r44	r44	r44
23			r45						r45	r45							r45	r45	r45	r45	r45		r45	r45	r45	r45	r45	r45	r45
24											s17	s22	s23																

Item	$	I	;	if	else	for	while	program	and	or	id	num	real	{	}	(	)	+	-	*	/	=	==	!=	>	>=	<	<=
25		s27																										
26											s17	s22	s23															
27																						s28		s28				
28											s17	s22	s23															
29																	s30											
30														s31														
31				s14		s15	s16				s17																	
32															s33													
33				r10	s34	r10	r10				r10				r10													
34														s35														
35				s14		s15	s16				s17																	
36															s37													
37				r11		r11	r11				r11				r11													
38																							s43	s44	s41	s42	s39	s40
39											r32	r32	r32															
40											r33	r33	r33															
41											r34	r34	r34															
42											r35	r35	r35															
43											r36	r36	r36															
44											r37	r37	r37															
45											s17	s22	s23															
46																	s47											
47														s48														
48				s14		s15	s16				s17																	
49															s89													
50									s66	s67							r18											

Item	$	I	;	if	else	for	while	program	and	or	id	num	real	{	}	(	)	+	-	*	/	=	==	!=	>	>=	<	<=
51			r21						r21	r21							r21	s55	s56				r21	r21	r21	r21	r21	r21
52			r25						r25	r25							r25	r25	r25	s61	s62		r25	r25	r25	r25	r25	r25
53			r28						r28	r28							r28	r28	r28	r28	r28		r28	r28	r28	r28	r28	r28
54			r29						r29	r29							r29	r29	r29	r29	r29		r29	r29	r29	r29	r29	r29
55											r38	r38	r38															
56											r39	r39	r39															
57			r20						r20	r20							r20						r20	r20	r20	r20	r20	r20
58											s17	s22	s23															
59			r23						r23	r23							r23	s55	s56				r23	r23	r23	r23	r23	r23
60			r22						r22	r22													r22	r22	r22	r22	r22	r22
61											r40	r40	r40															
62											r41	r41	r41															
63											s17	s22	s23															
64			r27						r27	r27							r27	r27	r27	s61	s62		r27	r27	r27	r27	r27	r27
65			r24						r24	r24								r24	r24				r24	r24	r24	r24	r24	r24
66											r42	r42	r42															
67											r43	r43	r43															
68																	r17											
69											s17	s22	s23															
70																	r19											
71			s72																									
72		s73																										
73																								s43	s44	s41	s42	s39	s40
74											s17	s22	s23															
75			s76																									
76		s77																										

Item	$	I	;	if	else	for	while	program	and	or	id	num	real	{	}	(	)	+	-	*	/	=	==	!=	>	>=	<	<=
77																						s78						
78		s79																										
79																		s55	s56	s61	s62							
80											r30	r30	r30															
81											r31	r31	r31															
82											s17	s22	s23															
83																	s84											
84														s85														
85				s14		s15	s16				s17																	
86															s87													
87				r12		r12	r12				r12				r12													
88			r14						r14	r14							r14	r14	r14	r14	r14		r14	r14	r14	r14	r14	r14
89				r13		r13	r13				r13				r13													
90			r26						r26	r26							r26	r26	r26				r26	r26	r26	r26	r26	r26
91				r9		r9	r9				r9				r9													

Tabela 7 – Seção Desvio

Item	$	PGM	STMT_LIST	STMT	CTRL_FLUX	IF	ELSE	FOR	WHILE	FUNCTION	PARAMS	MORE_EXP_BOOL	EXP_BOOL	EXP	MORE_TERM	TERM	MORE_FACTOR	FACTOR	OP_REL	OP_ARIT	OP_ARIT_LO	OP_ARIT_HI	OP_BOOL	VAR
0		g1																						
1																								
2																								
3			g4	g6	g8	g11		g12	g13	g9														
4																								
5																								
6			g7	g6	g8	g11		g12	g13	g9														
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18											g20													g21
19																								
20																								
21																								
22																								
23																								
24										g54			g29	g38		g51		g52						g53
25																								
26										g54			g46	g38		g51		g52						g53
27																								
28										g54				g71		g51		g52						g53
29																								
30																								
31			g32	g6	g8	g11		g12	g13	g9														
32																								
33							g91																	
34																								
35			g36	g6	g8	g11		g12	g13	g9														
36																								

Item	S	PGM	STMT_LIST	STMT	CTRL_FLUX	IF	ELSE	FOR	WHILE	FUNCTION	PARAMS	MORE_EXP_BOOL	EXP_BOOL	EXP	MORE_TERM	TERM	MORE_FACTOR	FACTOR	OP_REL	OP_ARIT	OP_ARIT_LO	OP_ARIT_HI	OP_BOOL	VAR
37																								
38																			g45					
39																								
40																								
41																								
42																								
43																								
44																								
45										g54				g50		g51		g52						g53
46																								
47																								
48			g49		g6		g8		g11		g12	g13	g9											
49																								
50												g68											g69	
51															g57						g58			
52																	g65					g63		
53																								
54																								
55																								
56																								
57																								
58										g54						g59		g52						g53
59															g60						g58			
60																								
61																								
62																								
63										g54								g64						g53
64																						g63		
65																								
66																								
67																								
68																								
69										g54			g70	g38		g51		g52						g53
70																								
71																								
72																								
73																			g74					
74										g54				g75		g51		g52						g53
75																								

Item	S	PGM	STMT_LIST	STMT	CTRL_FLUX	IF	ELSE	FOR	WHILE	FUNCTION	PARAMS	MORE_EXP_BOOL	EXP_BOOL	EXP	MORE_TERM	TERM	MORE_FACTOR	FACTOR	OP_REL	OP_ARIT	OP_ARIT_LO	OP_ARIT_HI	OP_BOOL	VAR
76																								
77																								
78																								
79																				s82	s80	s81		
80																								
81																								
82										s54				s83		s51		s52						s53
83																								
84																								
85			s86	s6	s8	s11		s12	s13	s9														
86																								
87																								
88																								
89																								
90																								
91																								

Para simular o processo de análise sintática, será utilizado o código a seguir para efetuar a análise.

```
program {
    acendeLedVerde();
}
```

A partir do código anterior, o analisador léxico gera a sequência de tokens a seguir para o analisador sintático.

PROGRAM, LCURB, ID(“acendeLedVerde”), LPAR, RPAR, SECO, RCURB, END_OF_FILE

O analisador sintático inicia sua execução no estado *0*, efetua a leitura de um token na entrada e então verifica se existe uma ação. Se existir, executa a ação para aquele estado e se não existir, apresenta um erro para o usuário. Quando o analisador está no estado *1* e o próximo token é o END_OF_FILE, então aquela sequência de tokens pode ser gerada pela gramática.

A Tabela 8 mostra a simulação do analisador sintático.

Tabela 8 – Simulação do analisador sintático.

Tokens na entrada	Pilha de token e variáveis	Pilha de estados	Ação
PROGRAM, LCURB, ID, LPAR, RPAR, SECO, RCURB, END_OF_FILE		0	Shift 2 – Empilha o estado 2
LCURB, ID, LPAR, RPAR, SECO, RCURB, END_OF_FILE	PROGRAM	0, 2	Shift 3 – Empilha o estado 3
ID, LPAR, RPAR, SECO, RCURB, END_OF_FILE	PROGRAM, LCURB	0, 2, 3	Shift 17 – Empilha o estado 17
LPAR, RPAR, SECO, RCURB, END_OF_FILE	PROGRAM, LCURB, ID	0, 2, 3, 17	Shift 18 – Empilha o estado 18
RPAR, SECO, RCURB, END_OF_FILE	PROGRAM, LCURB, ID, LPAR	0, 2, 3, 17, 18	Shift 19 – Empilha o estado 19
SECO, RCURB, END_OF_FILE	PROGRAM, LCURB, ID, LPAR, RPAR	0, 2, 3, 17, 18, 19	Reduce 15 – Retira 3 elementos das duas pilhas (de estados e de tokens). Estando o estado 3 no topo, então é empilhado o estado de destino com a transição rotulada com FUNCTION, da seção Desvio. Então, criado a

Tokens na entrada	Pilha de token e variáveis	Pilha de estados	Ação
			partir dos tokens retirados, é empilhado FUNCTION
SECO, RCURB, END_OF_FILE	PROGRAM, LCURB, FUNCTION (ID, LPAR, RPAR)	0, 2, 3, 9	Shift 10 – Empilha o estado 10
RCURB, END_OF_FILE	PROGRAM, LCURB, FUNCTION (ID, LPAR, RPAR), SECO	0, 2, 3, 9, 10	Reduce 5 – Retira 2 elementos das duas pilhas. Estando o estado 3 no topo da pilha, é empilhado o estado de destino com transição STMT. Então, é empilhado STMT.
RCURB, END_OF_FILE	PROGRAM, LCURB, STMT (FUNCTION (ID, LPAR, RPAR), SECO)	0, 2, 3, 6	Reduce 3- Retira 1 elemento das duas pilhas. Estando o estado 3 no topo da pilha, é empilhado o estado de destino com transição STMT_LIST. Então, é empilhado STMT_LIST.
RCURB, END_OF_FILE	PROGRAM, LCURB, STMT_LIST (STMT (FUNCTION (ID, LPAR, RPAR), SECO))	0, 2, 3, 4	Shift 5 – Empilha o estado 5
END_OF_FILE	PROGRAM, LCURB, STMT_LIST (STMT (FUNCTION (ID, LPAR, RPAR), SECO)), RCURB	0, 2, 3, 4, 5	Reduce 1 – Retira 4 elementos das duas pilhas. Empilha o estado de destino com transição PGM, e, então, empilha PGM.
END_OF_FILE	PGM (PROGRAM, LCURB, STMT_LIST (STMT (FUNCTION (ID, LPAR, RPAR), SECO)), RCURB)	0, 1	Aceita – Aceita a sequência de tokens recebida.

Na Tabela 8, tem-se 4 colunas:

- Tokens na entrada: Sequência de tokens na entrada que ainda serão lidos;
- Pilha de tokens e variáveis: Pilha de tokens que já foram lidos e variáveis que já foram reduzidas, o conteúdo dessa pilha será utilizado para construir a árvore gramatical;
- Pilha de estados: Pilha de estados que o analisador sintático percorreu;
- Ação: Ação que é executada quando existe uma transição para aquele token.

No fim do processamento, o analisador sintático retorna um único elemento da pilha de tokens e variáveis, na simulação da Tabela 8 é retornado PGM.

A cada ação de redução que é encontrada durante o processamento, o analisador cria um objeto do tipo de dados que representa a regra que foi reduzida, e armazena os tokens e variáveis que foram desempilhados. Na simulação da Tabela 8, a primeira redução encontrada, *r15*, após desempilhar os 3 tokens (ID, LPAR e RPAR), é criado um objeto do tipo que represente FUNCTION e, então, é atribuído os 3 tokens desempilhados.

Para cada variável da gramática, é criado um tipo de dados equivalente que contém cada elemento de sua regra. Se uma mesma variável possui mais de uma regra, então é criado um tipo separado para cada uma delas. A variável *FUNCTION*, por exemplo, possui 2 regras, então foram criados os tipos a seguir:

- *PARAM_FUNCTION*: Tipo para a regra *FUNCTION → id (PARAMS)*, contém os seguintes atributos:
  - *tokenId*: Variável para o token ID da regra;
  - *tokenLpar*: Variável para o token LPAR da regra;
  - *tokenParams*: Variável para o token PARAMS da regra;
  - *tokenRpar*: Variável para o token RPAR da regra.
- *NOPARM_FUNCTION*: Tipo para a regra *FUNCTION → id ( )*, contém os seguintes atributos:
  - *tokenId*: Variável para o token ID da regra;
  - *tokenLpar*: Variável para o token LPAR da regra;
  - *tokenRpar*: Variável para o token RPAR da regra.

2.2.3. Gerador de código e o interpretador

Esta é a última fase do compilador, nela, basicamente, são gerados os comandos a partir da árvore sintática retornada pelo analisador sintático.

Os comandos que são gerados para o interpretador do robô são parecidos com os comandos da linguagem de programação Assembly (THE ART OF ASSEMBLY LANGUAGE). Escolheu-se utilizar comandos parecidos com os comandos do Assembly devido à facilidade da geração dos códigos e da construção do interpretador para os mesmos.

Um comando é uma palavra que contém uma instrução e um ou mais argumentos para aquela instrução. O programa destino é uma sequência de comandos que são armazenadas em um *array* de *strings*.

Para interpretar tais comandos, o robô utiliza variáveis globais que atuam como algo equivalente aos registradores de um processador e, por isso, serão referenciados aqui como registradores. O interpretador possui os seguintes registradores:

- instrução: registrador que aponta para a posição atual do da instrução em execução do *array* de comandos;
- contagem: registrador utilizado para controlar as iterações em um laço de repetição *for*;
- comparação: registrador utilizado para armazenar o resultado de uma comparação entre dois valores, quando o resultado de uma comparação for verdadeiro o valor armazenado será 1, se for falso será 0;
- dados: registradores que armazenam os valores convertidos que foram extraídos do comando;

Os comandos possuem um dos seguintes formatos: <*instrução*><*valor*> e *c*<*valor*><*op*><*valor*>:

- <instrução><valor>: executa um comando utilizando, ou não, o argumento contido em <*valor*>, o comando recebido em <*instrução*> deve ser um dos itens a seguir:
  - z: instrução que efetua um salto condicional para a instrução contida no índice do *array* definido pelo argumento. O salto somente será efetuado se, e somente se, o valor contido no registrador de comparação for 0;
  - j: instrução que efetua um salto incondicional para a instrução contida no índice do *array* definido pelo argumento;
  - i: instrução que inicializa o registrador de contagem com o valor do argumento;
  - p: efetua a soma do registrador de contagem com o valor do argumento;
  - w: aciona os motores para avançar a quantidade de passos recebido no argumento;
  - a: aciona os motores para poder girar o robô em 90º para a esquerda;
  - d: aciona os motores para poder girar o robô em 90º para a direita;
  - s: aciona os motores para recuar a quantidade de passos recebido no argumento;
  - r: liga a LED vermelha;
  - t: desliga a LED vermelha;

- y: liga a LED amarela;
- u: desliga a LED amarela;
- g: liga a LED verde;
- h: desliga a LED verde;
- m: efetua o acionamento do sensor de distância para poder calcular a distância até o próximo objeto, a distância é calculada em centímetros.

- c<valor1><operador><valor2>: efetua a comparação dos dois valores recebidos utilizando o operador relacional definido em <*operador*>, os operadores relacionais disponíveis são >, >=, <, <=, ==, *!*=. Se o conteúdo em <*valor1*> for um *"c"*, o interpretador utilizará o valor do registrador de contagem na comparação.

Para gerar os comandos que são específicos para um conjunto de regras, é utilizado o método que é definido nos tipos daquelas regras. Alguns métodos podem apenas gerar chamadas para os métodos de seus atributos, sendo, assim, apenas um delegador (isso, geralmente, ocorrerá quando existe produções unitárias na gramática, ou seja, regras no formato $E \rightarrow F$, onde *E* e *F* são variáveis de gramática, então o tipo criado para a variável *E* será apenas um encapsulador do tipo criado para a variável *F*).

Na Tabela 9, mostra-se os principais tipos de dados (classes) das variáveis da gramática e os comandos que são gerados pelos seus métodos.

Tabela 9 – Tipos de dados e seus comandos.

Regra de produção	Nome do tipo de dados	Comandos
PGM → program { STMT_LIST }	Pgm	Efetua chamada (delega) para o atributo STMT_LIST.
STMT_LIST → STMT STMT_LIST	MultiStmtList	Delega para os atributos STMT e STMT_LIST.
STMT_LIST → STMT	SingleStmtList	Delega para o atributo STMT.
STMT → CTRL_FLUX	CtrlFluxStmt	Delega para o atributo CTRL_FLUX.
STMT → FUNCTION ;	FunctionStmt	Delega para o atributo FUNCTION.
CTRL_FLUX → IF	IfCtrlFlux	Delega para o atributo IF.
CTRL_FLUX → FOR	ForCtrlFlux	Delega para o atributo FOR.
CTRL_FLUX → WHILE	WhileCtrlFlux	Delega para o atributo WHILE.
IF → if (EXP_BOOL) { STMT_LIST } ELSE	IfElse	"c" + expressão retornada da delegação para EXP_BOOL; "z" + linha índice final do retorno da delegação para STMT_LIST; Delegação para STMT_LIST; "j" + índice retornado da delegação para ELSE;

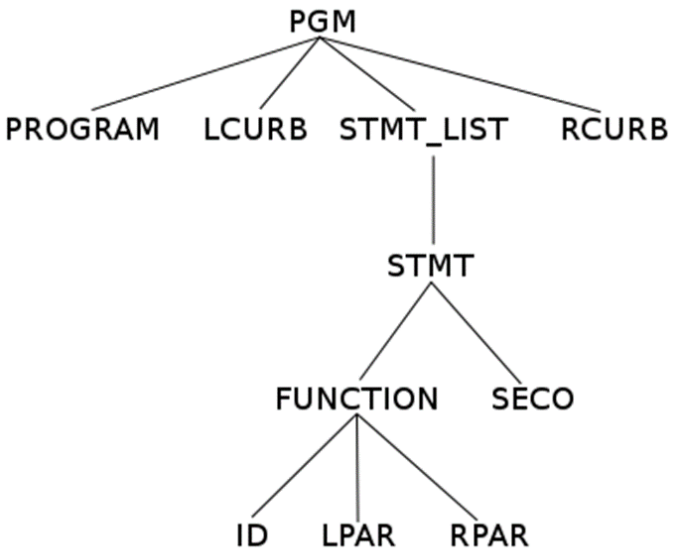
Regra de produção	Nome do tipo de dados	Comandos
IF → if (EXP_BOOL) { STMT_LIST }	SimpleIf	Delegação para ELSE. “c” + expressão retornada da delegação para EXP_BOOL; “z” + índice retornado da delegação para STMT_LIST; Delegação para STMT_LIST.
ELSE → else { STMT_LIST }	Else	Delegação para STMT_LIST.
FOR → for (i = EXP ; i OP_REL EXP ; i = i OP_ARIT EXP) { STMT_LIST }	For	“i” + valor da delegação para o primeiro atributo EXP; “cc” + operador em OP_REL + valor da delegação para o segundo EXP; “z” + índice retornado da delegação STMT_LIST. Delegação para STMT_LIST; “p” + valor da delegação para o terceiro EXP; Delegação para STMT_LIST.
WHILE → while (EXP_BOOL) { STMT_LIST }	While	“c” + expressão retornada da delegação para EXP_BOOL; “z” + índice retornado da delegação para STMT_LIST; Delegação para STMT_LIST; “j” + índice inicial do bloco de comandos.
FUNCTION → id (PARAMS)	ParamFunction	Lexema do token id + valor da delegação para PARAMS.
FUNCTION → id ( )	NoParamFunction	Lexema do token id + “0”
PARAMS → VAR	Param	Delegação para VAR
EXP_BOOL → EXP OP_REL EXP MORE_EXP_BOOL	MultiExpBool	Valor da delegação para o primeiro EXP + operador em OP_REL + valor da delegação para o segundo EXP + delegação para MORE_EXP_BOOL.
EXP_BOOL → EXP OP_REL EXP	SingleExpBool	Valor da delegação para o primeiro EXP + operador em OP_REL + valor da delegação para o segundo EXP.
MORE_EXP_BOOL → OP_BOOL EXP_BOOL	MoreExpBool	Operador lógico em OP_BOOL + delegação para EXP_BOOl.
EXP → TERM MORE_TERM	MultiExpBool	Delegação TERM + delegação para MORE_TERM
EXP → TERM	SingleExpBool	Delegação para TERM
MORE_TERM → OP_ARIT_LO TERM MORE_TERM	MultiMoreTerm	Operador aritmético em OP_ARIT_LO + delegação para TERM + delegação para MORE_TERM.
MORE_TERM → OP_ARIT_LO TERM	SingleMoreTerm	Operador aritmético em OP_ARIT_LO + delegação para TERM.
TERM → FACTOR MORE_FACTOR	MultiTerm	Delegação para FACTOR + Delegação para MORE_FACTOR
TERM → FACTOR	SingleTerm	Delegação para FACTOR
MORE_FACTOR → OP_ARIT_HI FACTOR MORE_FACTOR	MultiMoreFactor	Operador aritmético em OP_ARIT_HI + delegação para FACTOR + delegação para MORE_FACTOR
MORE_FACTOR → OP_ARIT_HI FACTOR	SingleMoreFactor	Operador aritmético em OP_ARIT_HI + delegação para FACTOR
FACTOR → VAR	VarFactor	Delegação para VAR

Regra de produção	Nome do tipo de dados	Comandos
FACTOR → FUNCTION	FunctionFactor	Delegação para FUNCTION
OP_ARIT → OP_ARIT_LO	LowOpArit	Delegação para OP_ARIT_LO
OP_ARIT → OP_ARIT_HI	HiOpArit	Delegação para OP_ARIT_HO
OP_REL → RELOP_LT OP_REL → RELOP_LTE OP_REL → RELOP_GT OP_REL → RELOP_GTE OP_REL → RELOP_EQ OP_REL → RELOP_NEQ	OpRel	Lexema do token do atributo relOp.
OP_ARIT_LO → + OP_ARIT_LO → -	OpAritLo	Lexema do token do atributo opArit.
OP_ARIT_HI → * OP_ARIT_HI → /	OpAritHi	Lexema do token do atributo opArit.
OP_BOOL → and OP_BOOL → or	OpBool	Lexema do token do atributo opBool.
VAR → num VAR → real	Var	Lexema do token do atributo var.

Tendo a árvore gramatical, retornada pelo analisador sintático, e executando os métodos de geração de código em cada elemento dela, no final tem-se um *array* de comandos que será enviado ao robô.

Para exemplificar, será feita a geração de código da árvore gramatical que foi gerada na simulação anterior, mostrada na Tabela 8. Para facilitar a visualização, a árvore é mostrada graficamente na Figura 20.

Figura 20 – Árvore gramatical gerado na simulação da Tabela 8.



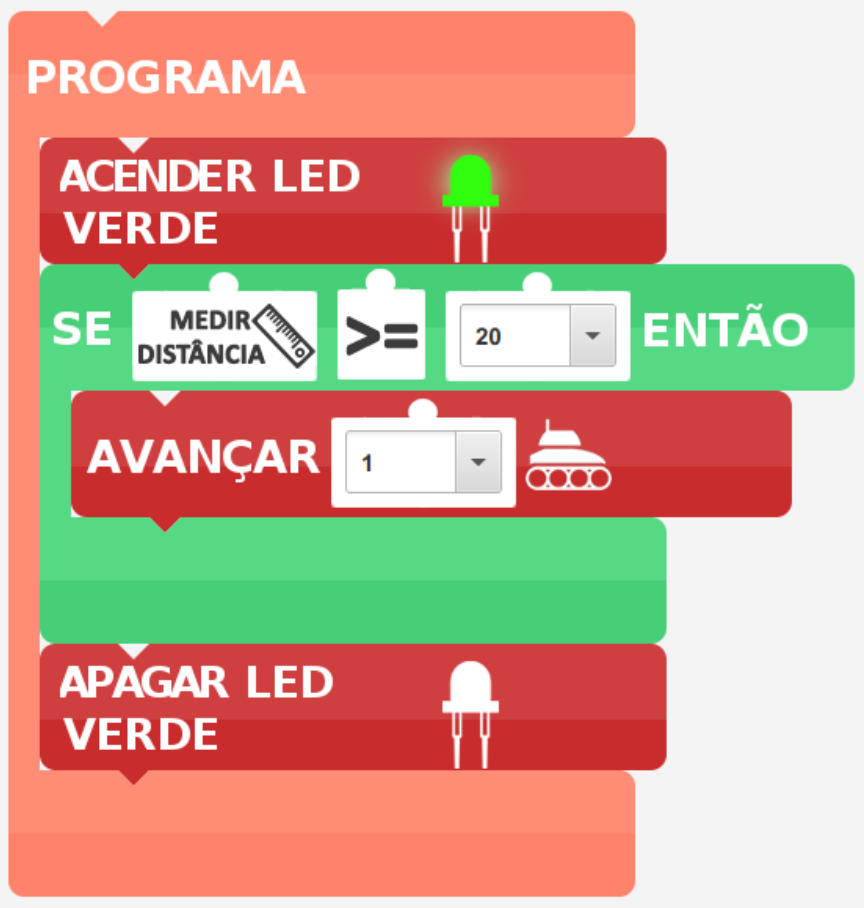
O processo inicia efetuando a chamada do método de geração da raiz da árvore, PGM. O método de PGM efetua a chamada do método de STMT_LIST, pois apenas esse atributo irá gerar instruções. Em STMT_LIST, sendo uma produção unitária (STMT_LIST → STMT), será chamado apenas o método do STMT. E STMT, efetua a chamada do método de FUNCTION e, por fim, FUNCTION gera a instrução que o robô irá interpretar. Para gerar a instrução, é lido o lexema armazenado no token ID (no exemplo, “acendeLedVerde”) e é verificado na lista de funções padrão se existe alguma função com o mesmo nome. No caso, será retornado “g”.

No final do encadeamento das chamadas, será gerada a instrução: g0. O robô ao interpretar ler a instrução “g”, irá acender o LED verde.

2.2.4. Exemplo de compilação completa

Para mostrar um exemplo completo da compilação, nesta seção será feito a compilação do código que a interface gráfica gera a partir dos blocos selecionados, mostrado na Figura 21.

Figura 21 – Blocos selecionados na interface gráfica.



Tendo esses blocos selecionados, ao clicar no botão “Executar”, a aplicação irá percorrê-los, recursivamente, gerando o código do programa fonte específico de cada bloco.

O bloco “programa” irá gerar a *string* de código “program { }”. E, entre a abertura e fechamento de chaves ({ e }), irá incluir o código que seus blocos internos geram: o bloco “Acender LED Verde” irá gerar a *string* “acendeLedVerde()”; o bloco de condição SE/ENTÃO gera o código utilizando o operador e seus operandos utilizados no teste condicional, no exemplo inicia a *string* com “if(medirDistancia()>=20){}”, em seguida inclui, dentro das chaves, o código que seus blocos internos geram, no caso o bloco de ação Avançar e seus argumento, geram a *string* “avanca(1)”; e, por último, é incluído também, dentro das chaves de programa, o código gerado pelo bloco de comando “Apagar LED Verde”.

No final desse processo, será obtido o código mostrado no Quadro 2, que foi formatado para facilitar a visualização.

Quadro 2 – Código gerado pela interface

```
program {
      acendeLedVerde();
      if ( medeDistancia() >= 20 ) {
            avanca(1);
      }
      apagaLedVerde();
}
```

O código que foi gerado, mostrado no Quadro 2, então é enviado para o compilador efetuar o tratamento e a geração do código final que o robô irá interpretar.

O compilador, após receber esse código, inicia o processamento nas fases de análise. Primeiro, o analisador léxico efetua a leitura de cada caractere para poder identificar os tokens ali contidos. Após toda a leitura da entrada, será gerado a sequência de tokens a seguir:

PROGRAM, LCURB, ID(“acendeLedVerde”), LPAR, RPAR, SECO, IF, LPAR, ID(“medeDistancia”), LPAR, RPAR, RELOP_GTE, NUM(20), RPAR, LCURB,

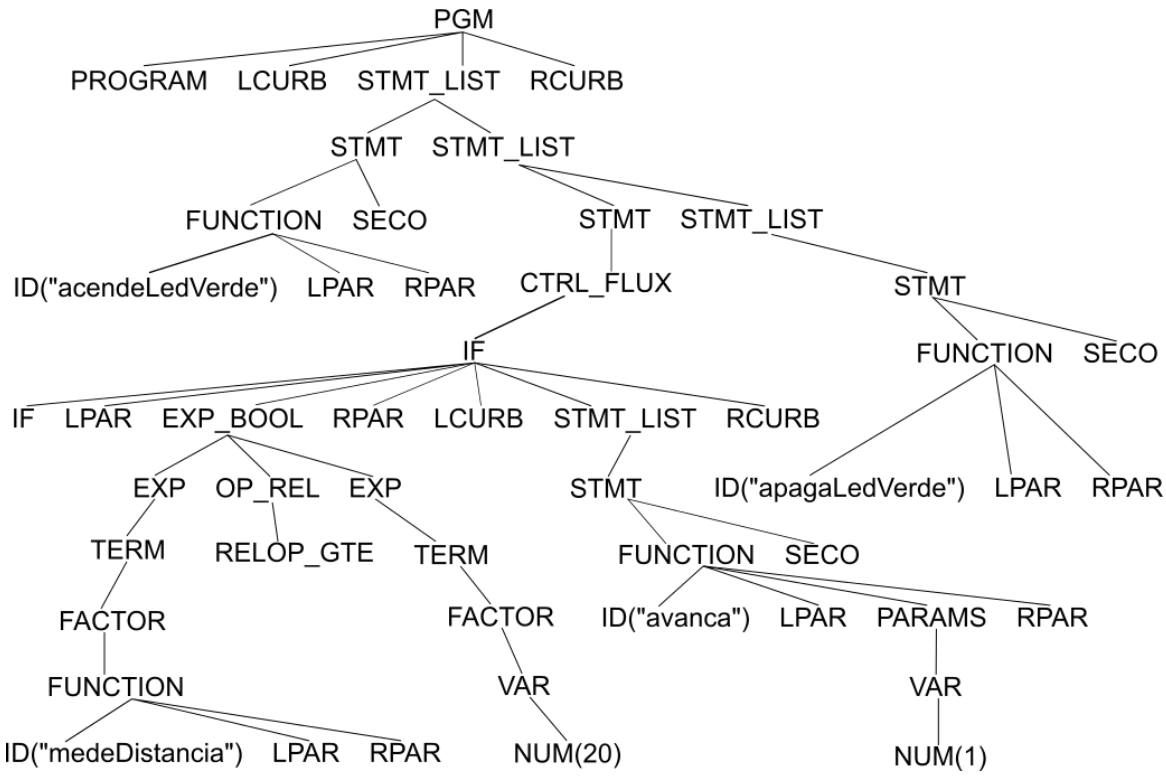
ID(“avanca”), LPAR, NUM(1), RPAR, SECO, RCURB, ID(“apagaLedVerde”), LPAR, RPAR, SECO, RCURB, END_OF_FILE

Após o analisador léxico gerar a lista de tokens, inicia-se a fase de análise sintática. O analisador sintático recebe a lista de tokens e inicia o processamento dela. No final do processo, será gerada a árvore gramatical a seguir.

PGM(PROGRAM, LCURB, STMT_LIST (STMT(FUNCTION(ID(“acendeLedVerde”), LPAR, RPAR), SECO), STMT_LIST(STMT(CTRL_FLUX(IF(IF, LPAR, EXP_BOOL(EXP(TERM(FACTOR(FUNCTION(ID(“medeDistancia”), LPAR, RPAR)))), OP_REL(RELOP_GTE), EXP(TERM(FACTOR(VAR(NUM(20)))))), RPAR, LCURB, STMT_LIST(STMT(FUNCTION(ID(“avanca”), LPAR, PARAMS(VAR(NUM(1))), RPAR), SECO)), RCURB))), STMT_LIST(STMT(FUNCTION(ID(“apagaLedVerde”), LPAR, RPAR), SECO)))), RCURB)

Na Figura 22, mostra-se, graficamente, essa árvore gramatical gerada.

Figura 22 – Árvore gramatical gerada.



E, por último, o compilador efetua a geração do código final a partir da raiz PGM da árvore. No Quadro 3, tem-se o código gerado a partir dessa árvore.

Quadro 3 – Código gerado

g0|cm>=20|z4|w1|h0|$

Com o código final gerado, a aplicação envia para o robô efetuar a interpretação e execução.

O robô recebe a *string* e divide-a entre seus comandos, usando como delimitador o caractere *pipe* (|), e armazena em um *array*. Por fim, percorre o *array* de comandos efetuando a interpretação de cada comando. A seguir é listado a interpretação de cada comando.

- g0: Acende o LED verde;
- cm>=20: Aciona o sensor de distância para efetuar uma medição, armazena o valor retornado em centímetros e então efetua a comparação do valor. Se o valor obtido é maior ou igual a 20, atribui-se ao registrador de comparação o valor 1, se não atribuí 0;
- z4: Efetua um salto para o índice 4 do *array* de comandos se o valor do registrador de comparação for 0;
- w1: Aciona o motor para efetuar o movimento de 1 passo;
- h0: Desliga o LED verde;
- $: Identifica o fim dos comandos, então finaliza a interpretação e reinicia o estado do robô, desliga os LEDs.

2.4. INTEGRAÇÃO

Nesta fase do projeto é realizado o envio dos comandos que foram gerados pelo compilador para o robô.

O envio é feito utilizando dois dispositivos bluetooth que já estejam conectados com uma sessão (pareados). Ao iniciar sua execução, a aplicação efetua a tentativa de pareamento com um dispositivo de um determinado nome e endereço bluetooth, que são determinados pelo dispositivo utilizado pelo robô – neste projeto é um módulo HC-05 que possui o nome do dispositivo “linvor” por padrão e endereço no formato

FF:FF:FF:FF:FF:FF, sendo FF um valor em hexadecimal. Uma vez pareados, não há nenhum outro processo necessário além do envio dos comandos.

O módulo bluetooth tem como principal função a comunicação serial sem fio, então logo após o compilador gerar os comandos, o dispositivo recebe esses dados em formato texto e os modula para serem enviados utilizando os protocolos do Bluetooth para poder codificarem esses dados em sinais a serem transmitidos.

Essa modulação é feita utilizando a técnica Gaussian Frequency Shift Keying (GFSK), a modulação transmite os dados definindo um conjunto discreto de frequência para cada bit durante a transmissão (por exemplo, bit 1 transmite 2100 Hz e bit 0 transmite em 1300 Hz) e é utilizada uma função gaussiana para diminuir a largura espectral dos sinais, suavizando a transição entre os valores dos impulsos (NEAR COMMUNICATIONS).

O módulo Bluetooth do robô recebe os sinais de transmissão de dados e os demodula novamente para formato texto. Após a transferência estar completa, o robô segue com a execução da interpretação dos comandos recebidos.

3. PROJETO DE HARDWARE

A construção do robô, que faz parte do kit aqui proposto, foi feita utilizando-se diversas soluções de hardware disponíveis, como o Arduino. Apresenta-se aqui o seu funcionamento.

3.1. ARDUINO

O Arduino é o “cérebro” do robô, ou seja, é o controlador de todos os componentes. Constitui-se em uma plataforma de código aberto baseada em software e hardware e que visa a criação modelos e projetos com maior facilidade. Foi criado em 2005 pelo Instituto de Design de Interação (IVREA) e era inicialmente destinada a estudantes sem experiência em eletrônica e programação, mas logo passou a ser conhecida por comunidades mais amplas e a placa começou a mudar para se adaptar às novas necessidades e desafios, diferenciando-se de placas de 8 bits comuns, para aplicações da internet das coisas, wearable, impressões 3D e ambientes incorporados (ARDUINO, 2016).

O ambiente de desenvolvimento, ou “Arduino Software”, contém um editor de texto para escrever códigos, que são chamados de “esboços”, uma área de mensagem, um console de texto e uma barra de ferramentas com botões para funções comuns e uma série de menus. Essa IDE permite a conexão entre hardware e software. Tanto o hardware como o software são totalmente código aberto e podem receber contribuições de todo o mundo para sua evolução (ARDUINO, 2016).

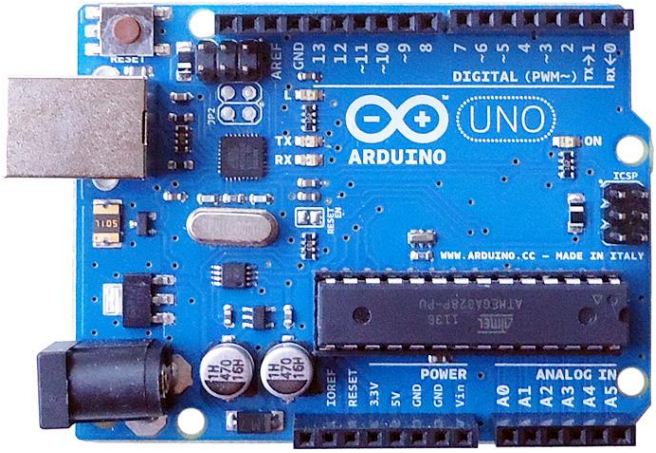
Existem diversos microcontroladores e plataformas disponíveis que oferecem funcionalidades semelhantes, Stamp Parallax Basic da NetMedia BX-24 e Phidgets, Handyboard do MIT (ARDUINO, 2016) são alguns exemplos. Porém, essas ferramentas possuem detalhes complexos de programação de microcontoladores, que em uma pequena solução pode ser um grande problema. Algumas vantagens do Arduino sobre as outras plataformas são:

- Solução barata, em comparação com outras plataformas de microcontoladores;
- Ambiente de programação amigável;
- Fácil utilização e suficientemente flexível para usuários avançados;
- Multiplataforma, ou seja, pode ser executado em diversos sistemas operacionais, como Windows e plataformas baseadas em Unix, enquanto a maioria dos sistemas de microcontoladores são limitados ao Windows;

- Código aberto. O hardware e software são vendido sob a licença de utilização Creative Commons, ou seja, são de código aberto e permitem alterações internas.

Neste projeto foi utilizado um Arduino de modelo Uno R3, que pode ser visto na Figura 23.

Figura 23 – Arduino Uno R3.



Fonte: ARDUINO – 2016

O Arduino possuí duas formas de alimentação. Por uma porta USB tipo B ou por uma entrada para a fonte externa de no máximo 12 volts.

A plataforma tem diversas entradas para as mais diversas finalidades. É possível adicionar sensores, luzes LED e *shields*, que são placas criadas para ampliar as possibilidades e adicionar novas funções ao Arduino. No projeto, foram utilizadas as conexões já existentes e a Tabela 10 mostra em quais portas foram ligados os dispositivos.

Tabela 10 – Tabela de pinos.

Pinos do Arduino	Dispositivos
0	Modulo Bluetooth HC05 (tx)
1	Modulo Bluetooth HC05 (rx)
2	Sensor SR04 (ECHO)
3	Sensor SR04 (TRIG)
4	Ponte H (IN1)
5	Ponte H (IN2)
6	Ponte H (IN3)
7	Ponte H (IN4)

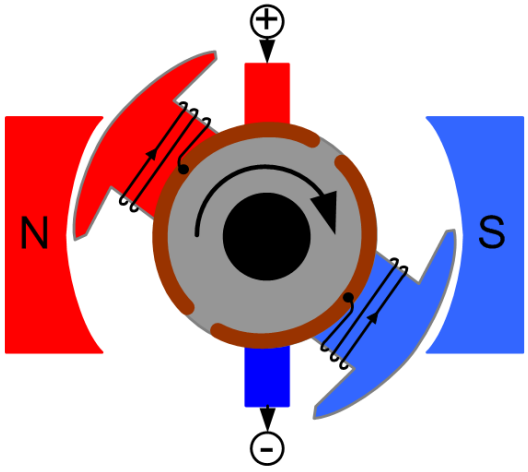
Pinos do Arduino	Dispositivos
8	LED_Alerta vermelho
9	LED_Alerta Amarelo
10	LED_Alerta Verde
11	LED Vermelha
12	LED Amarela
13	LED Verde

3.2. MOTORES

A movimentação do robô foi elaborada utilizando um par de motores de corrente continua, também conhecidos como motores DC (*direct current*), que apesar do seu funcionamento de fácil entendimento, já que funciona aproveitando as forças de atração e repulsão geradas por eletroímãs e imãs permanentes, é muito funcional.

Um motor deste tipo converte a energia elétrica em energia mecânica como qualquer motor, mas tem uma característica que o individualiza, deve ser alimentado com tensão contínua. Essa tensão pode provir de pilhas e baterias, no caso de pequenos motores, ou de uma rede de corrente alternada após retificação, no caso de motores maiores.

Figura 24 – Esquema de funcionamento do motor DC.



Fonte: VIDA DE SILÌCIO.

Como mostrado na Figura 24, dentro do motor DC há dois ímãs permanentes, um positivo (N) e um negativo (S). Ao ser energizado, o rotor (*armature*) passa a ter também, de cada lado, os polos negativo e positivo e as forças de atração e repulsão fazem com que o rotor se movimente e gire dentro de seu eixo.

3.3. PONTE H

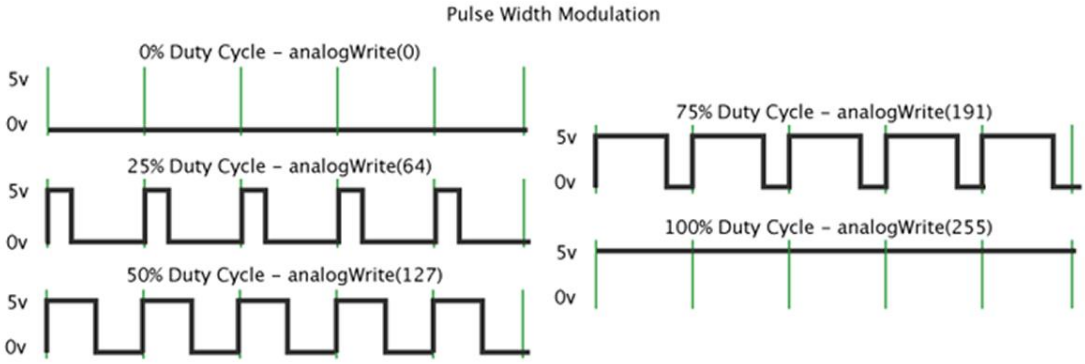
Para o controle do par de motores DC utilizados, foi escolhida a ponte H, que permite a utilização da modulação por largura de pulso (PWM - *Pulse Width Modulation*).

3.3.1. PWM

A PWM é uma técnica que converte sinais analógicos em digitais e também emite ondas com sinal lógico alto, formando ondas quadradas. Desse modo é possível calcular porcentagem de tempo ou *duty cycle* (ciclo de trabalho).

O sinal mais alto no Arduino equivale a 5V (volts) e o mais baixo 0V, logo, essa onda varia entre 0V (0% de *duty cycle*) e 5V (100% de *duty cycle*), como pode ser visto na Figura 25.

Figura 25 – Variação de ondas.



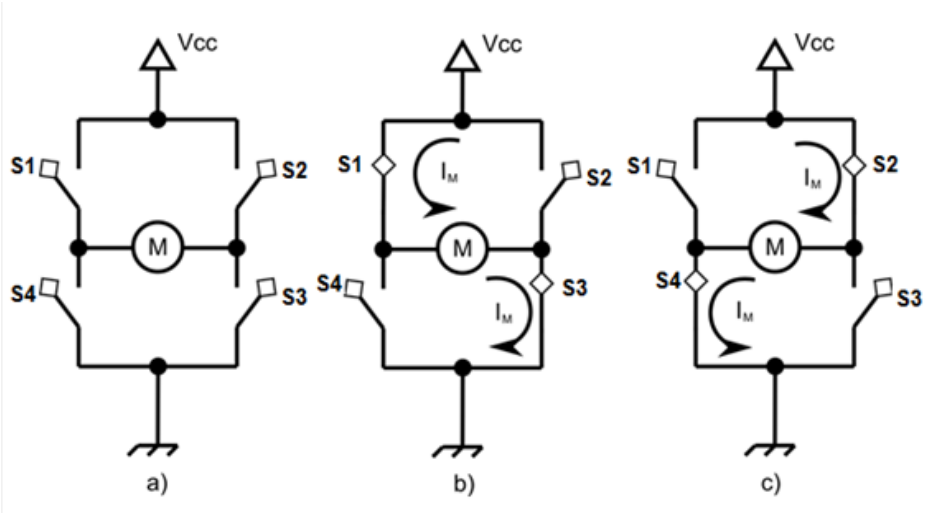
Fonte: ARDUINO – 2016

As representações da Figura 26 são exemplos de ondas do *duty cycle* entre 0 a 100%. O tempo que a onda fica 5V dentro um ciclo representa a quantidade de tempo, em porcentagem, que o sistema ficará ativo.

3.3.2. O circuito

A ponte H é um circuito utilizado para controlar um motor DC a partir dos sinais gerados pelo microcontrolador como, por exemplo, o sinal *duty cycle*. Ela é responsável por controlar a corrente que será distribuída aos motores e recebe esse nome devido ao formato que é montado o circuito, semelhante à letra “H”.

Figura 26 – Seleção de sentido de rotação na ponte H.



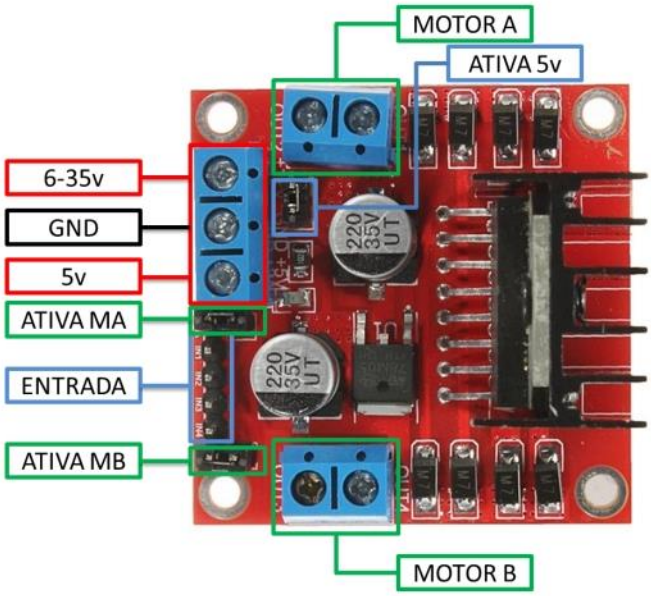
Fonte: VIDA DE SILÌCIO – 2016

Devido à disposição dos seus componentes, torna-se extremamente fácil selecionar o sentido da rotação de um motor, apenas invertendo a polaridade, como pode-se ver na Figura 26.

a) Quando nenhum par de chaves está acionado, o motor está desligado;
b) Quando o par S1-S3 é acionado a corrente percorre S1-S3 fazendo com que o motor gire em um sentido;
c) Já quando o par S2-S4 é acionado a corrente percorre por outro caminho fazendo com que o motor gire no sentido contrário.

Por meio das ondas geradas, da ponte H e do par de motores, é possível ter os comandos **acelerar**, **frear** e **virar** necessários para a movimentação do robô, como são exemplificados na Figura 27.

Figura 27 – Entradas e saídas da ponte H.



Fonte: FILIPEFLOP – 2016

- “Ativa MA” e “Ativa MB”: são os pinos responsáveis pelo controle PWM dos motores A e B. Se estiverem com jumper, não haverá controle de velocidade, pois os pinos estarão ligados durante todo o tempo em 5V.
- “Ativa 5V” e “5V”: a ponte H possui um regulador de tensão integrado e, quando o driver está operando entre 6-35V, esse regulador disponibiliza uma saída regulada de +5V no pino “5V” para o uso externo (com jumper), podendo alimentar, por exemplo, outro componente eletrônico.
- “6-35v” e “GND”: nestes pinos é conectada a fonte de alimentação externa quando o driver estiver controlando um motor que opere entre 6-35V.
- “Entrada”: este barramento é composto pelos pinos “IN1”, “IN2”, “IN3” e “IN4”, responsáveis pela rotação do motor A (IN1 e IN2) e motor B (IN3 e IN4).

A Tabela 11 mostra a ordem de ativação do motor A através dos pinos IN1 e IN2 e a do motor B com os pinos IN3 e IN4 com HIGH e LOW, que significa fornecer 5V ou 0V, respectivamente, para o motor por meio do Arduino.

Tabela 11 – Tabela de ativação dos motores.

MOTOR	IN1	IN2	IN3	IN4
Sentido horário	HIGH	LOW	HIGH	LOW
Sentido anti-horário	LOW	HIGH	LOW	HIGH
Ponto morto	LOW	LOW	LOW	LOW
Freio	HIGH	HIGH	HIGH	HIGH

O Quadro 4 contém o código utilizado no Arduino para realizar as movimentações do robô.

Quadro 4 – Código utilizado para ativação do motor.

```
//COMANDOS PARA PONTE H CONTROLAR OS MOTORES
//Se o valor for 'w' o robo avança
if(input=='w')
{
  if (cmMsec > 20)//verifica se a distancia do objeto é maior que 20 cm
  {
  Serial.println("Robo Avançando");
  //Motores A e B no sentido Horario
  //referente ao motor A
  digitalWrite(IN1,HIGH);
  digitalWrite(IN2,LOW);
  digitalWrite(IN3,HIGH);
  digitalWrite(IN4,LOW);
  delay(2000);
  }

  else if(cmMsec < 20)
  {
     digitalWrite(IN1,HIGH);
     digitalWrite(IN2,HIGH);
     digitalWrite(IN3,HIGH);
     digitalWrite(IN4,HIGH);
      delay(500);

    for (int i=0; i<3;i++){
      digitalWrite(ledRed, HIGH);
      delay(100);
      digitalWrite(ledRed, LOW);
      delay(100);
    }

  }
```

3.4. BATERIA

Existem baterias de diversos tipos e composições, como de níquel-cádmio (NiCd), hidreto metálico de níquel (NiMH), íons de lítio (Li-Ion) e polímeros de lítio (Li-Po). Cada uma com suas particularidades, como potência, possibilidade de recarga ou não e segurança no manuseio.

Considerando os fatores apresentados, foi escolhida um conjunto de quatro pilhas do tipo NiMH com capacidade de 2100mAh cada, ligadas em paralelo para totalizarem 8400mAh, para alimentar os motores, e uma bateria de 9V com 250mAh, para alimentar o sensor ultrassônico e conjunto de luzes LED, pois são recarregáveis e não possuem alto risco de explosão como as baterias Li-Po. Esse conjunto permite uma autonomia de utilização de cerca de 2 horas O cálculo da autonomia é demostrado no Quadro 5.

Quadro 5 – Cálculo da autonomia.

Ligado na bateria de 9V c/ 250mAh	
6x luzes LED	**20mA**
1x sensor ultrassônico	**15mA**
Total	**135mA**
Autonomia aprox. 2h (consume 270mA em 2h)	
Ligado nas 4 baterias de 2100mAh (em paralelo, total 8400mAh)	
Motor A	**2100mA**
Motor B	**2100mA**
Total	**4200mA**
Autonomia aprox. 2h (consome 8400mA em 2h)	

3.5. COMUNICAÇÃO SEM FIO

A comunicação entre o robô e o computador com a interface gráfica criada para especificar os comandos é feita utilizando o padrão bluetooth. Foi utilizado um módulo bluetooth que permite a transmissão dos dados sem fio a curta distância (cerca de 10 metros) e fica conectado ao Arduino por meio de uma porta serial.

3.6. SENSORES E LEDs

Alguns elementos que visam o aumento na interação no kit foram usados. Um deles é o sensor ultrassônico, que tem por objetivo permitir a medição da distância entre o robô e um obstáculo à sua frente.

O sensor possui um emissor chamado TRIG e um receptor chamado de ECHO, que emitem pulsos sonoros ultrassônicos. Para medir a distância de um obstáculo, o TRIG emite pulsos sonoros que colidem e retornam, sendo recebidos pelo ECHO. É

feita a medição do tempo entre o envio e recebimento dos pulsos e através de cálculos é possível obter a distância em centímetros. Isso é possível utilizando funções disponíveis na biblioteca “ultrasonic” para a linguagem C.

O Quadro 6 contém um exemplo de código utilizando as funções do sensor ultrassônico.

Quadro 6 – Código utilizado para ativar o sensor ultrassônico

```
//A variável input recebe o valor na Serial (para imprimir no serial monitor)
input= Serial.read();

//variaveis referentes ao sensor ultrasonico
float cmMsec,inMsec;

//lê os dados do sensor, com o tempo de retorno do sinal
long microsec = ultrasonic.timing();

//Calcula a distancia e converte para centimetros
cmMsec = ultrasonic.convert(microsec, Ultrasonic::CM);
```

Já as luzes LED têm dupla função. Podem ser acesas pelo usuário, mas também servirão como aviso visual quando o sensor ultrassônico for ativado. O Quadro 7 mostra o código utilizado na função de acender a luz LED que pode ser utilizada pelo usuário.

Quadro 7 – Código para acender e apagar luz LED vermelha

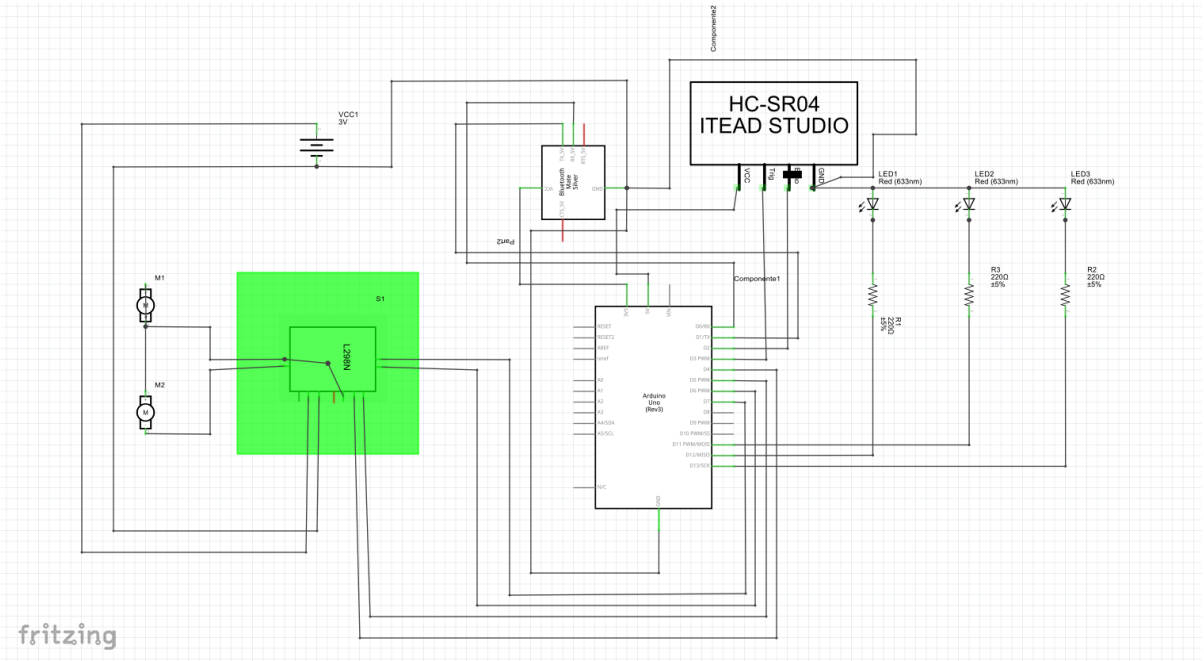
```
//COMANDO PARA ACENDER E APAGAR LEDS
//ACENDENDO:
//vermelho
if(input=='r')
{
  Serial.println("Acender ledRed");
  digitalWrite(ledRed, HIGH);
}

//APAGANDO:

//vermelho
if(input=='t')
{
  Serial.println("Apagando ledRed");
  digitalWrite(ledRed, LOW);
}
```

A Figura 28 mostra uma representação das ligações dos componentes entre si e que formam o robô.

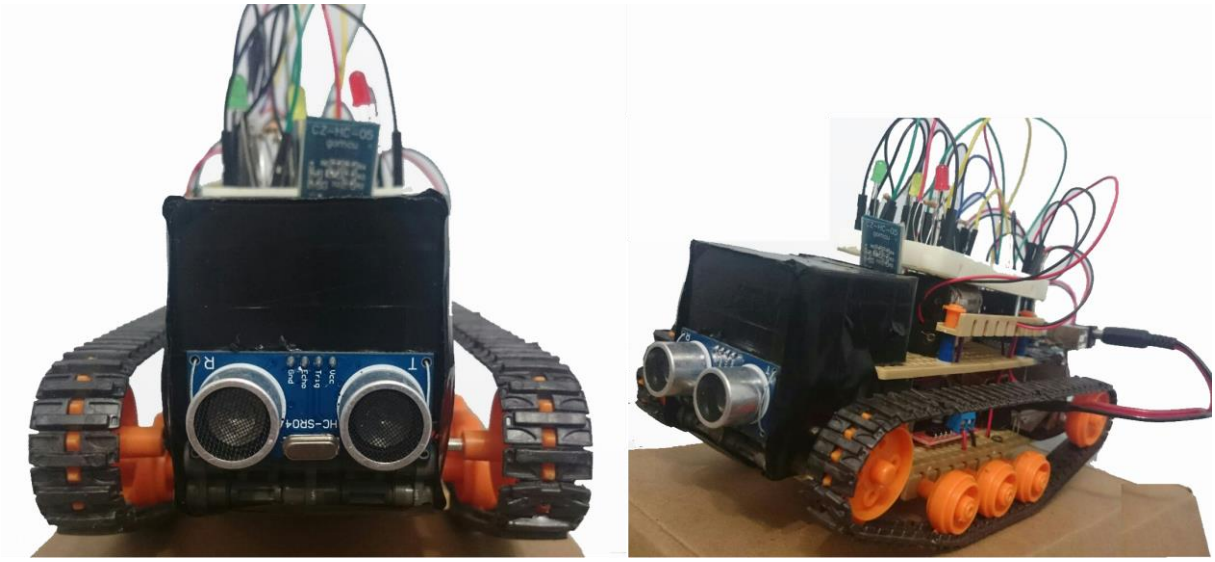
Figura 28 – Diagrama de interligação.



4. PRIMEIROS RESULTADOS

Após a montagem dos componentes e programação da aplicação e do Arduino, o robô resultante foi este mostrado na Figura 29 e também alguns testes com voluntários foram feitos.

Figura 29 – Robô.



Os testes foram realizados em um dia e tinham como objetivo avaliar a usabilidade do kit. Três pessoas foram voluntárias nesse teste, para que fosse possível obter o mínimo de dados para comparação. Elas tinham grande diferença de idades, para avaliar as diferentes reações entre adultos e adolescentes, e diferença de escolaridade também. Eram dois homens, um de 16 e outro de 50 anos, e uma mulher, de 52 anos.

Foi dado para todos o mesmo roteiro para ser executado. Ele começa com um desafio de nível fácil (acender e apagar o LED de cor verde), de nível médio (avançar dois passos e virar à direita) e, por fim, de nível difícil (fazer com que o robô fizesse o movimento simulando um “8”). Todos conseguiram realizar de forma satisfatória todos os desafios e apenas o senhor de 50 anos entrou um pouco de dificuldade.

Após a execução do roteiro, algumas perguntas foram feitas aos voluntários para avaliar a receptividade e interesse futuro em programação e em computação. Mas primeiro, os voluntários foram perguntados sobre a usabilidade da aplicação e todos julgaram-a como de fácil utilização. Em seguida, cada um foi questionado sobre qual ponto sentiram mais dificuldade e os resultados são mostrados na Tabela 12.

Tabela 12 – Dificuldade na utilização da aplicação.

Sexo	Idade	Dificuldade
Homem	16 anos	Identificar a peça de número decimal
	50 anos	
Mulher	52 anos	

Por fim, foi perguntado qual era a avaliação sobre o projeto, se julgavam interessante, e se seguiriam nessa área, como por exemplo, estudar em uma escola técnica quando mais novos. Todos avaliaram como o kit positivamente, acharam-o muito interessante, e, se tivessem oportunidade, pensariam em cursar uma escola técnica com matérias relacionadas caso tivessem oportunidade.

5. CONCLUSÃO

Neste trabalho, foi desenvolvido um kit composto por uma interface gráfica, um compilador e um robô, como objetivo principal. Desenvolvemos uma interface de fácil utilização para que o usuário possa criar seus programas intuitivamente, por meio de peças baseadas no formato de quebra-cabeça e assim permitindo a montagem de uma sequência de comandos que serão compilados e reproduzidos pelo robô. A forma com que foi pensada, as peças permitem enxergar o fluxo de execução do programa, ou seja, a ordem de reprodução das funções e suas condições.

O compilador foi criado para permitir uma melhor comunicação entre a aplicação e o robô. Ele permite que cada peça de quebra-cabeça do programa seja representada por um pequeno conjunto de letras e números. Com isso, evitando ao máximo o sobrecarregamento do meio de transmissão sem fio utilizado e fornecendo uma maior disponibilidade e agilizando todo processo de tradução do robô, já que não precisa tratar grande volume de dados.

E o robô fica responsável por traduzir as informações criadas pelo usuário e compiladas e consolidar todo esse trabalho, convertendo os comandos em movimentações condicionadas, ou não. Como recebe um pequeno fluxo de dados a cada execução, o robô tem apenas que receber e traduzir os comandos para suas funções de virar, andar ou acender um LED, por exemplo.

5.1. TRABALHOS FUTUROS

O projeto foi desenvolvido com o objetivo de apoiar o aluno em sua vida escolar utilizando-se a lógica de programação. Como um trabalho futuro, há a possibilidade de apresenta-lo em escolas e salas de aula e avaliar sua usabilidade e aceitação, permitindo realizar melhorias que possam contribuir para uma utilização melhor e mais fácil, deixando o aluno mais engajado ao utilizar o kit.

Também há a possibilidade de incluir novos comandos, como operações matemáticas, e novas formas de interação utilizando-se outros tipos de sensores, como de temperatura e luminosidade, no robô.

BIBLIOGRAFIA

AHO et al. **Compiladores**: Princípios, Técnicas e Ferramentas. 2ª Edição. Editora LTC. 1986.

APPEL, Andrew W. **Modern Compiler Implementation in Java**. 2ª Edição. University of Cambridge:Cambridge University Press. 2002.

ARAÚJO, Everton Coimbra de, **Algoritmos**: Fundamentos e Prática. 3ª Edição. Florianópolis:Visual Books, 2007.

ARDUINO, 2016, **Arduino – Introduction**. Disponível em: https://www.arduino.cc/en/Guide/Introduction. Acessado em: 24/05/2016.

ARDUINO, 2016, **Arduino Playground – Appendix3**. Disponível em: http://playground.arduino.cc/ArduinoNotebookTraduccion/Appendix3. Acessado em: 24/05/2016.

AVOZANI, Mariel *et al.*. **Clube de Programação nas Escolas**: Novas Perspectivas Para o Ensino da Computação. UNIJUI. Farroupilha: 2014. 5 p. Disponível em < http://www.projetos.unijui.edu.br/moeducitec/moeducitec/principal/82.pdf >. Acessado em 12/06/2016

BERNARDI *et al*. **O Desenvolvimento do Raciocínio Lógico através de Objetos de Aprendizagem**. Renote. Santa Maria: 2007. 10 p.. Disponível em <http://www.seer.ufrgs.br/renote/article/viewArticle/14253>. Acessado em 20/03/2016.

BORNAT, **Richard. Undertanding and Writing Compilers**: A do-it-yourself guide. 1ª Edição. Middlesex University:Macmillan Press Ltd. 1979.

CALEGARI, Paulo Ferreira. **Aplicação da Robótica no Ensino-aprendizagem de Lógica de Programação para Crianças**. Araranguá: UFSC, 2015. 53 p. Trabalho de Conclusão do Curso - Bacharelado em Tecnologias da

Informação e Comunicação. Disponível em <https://repositorio.ufsc.br/handle/123456789/133649>, acessado em 28/02/2016.

COLLING, Juliane *et al*. **Programação de Computadores Como Meio de Desenvolvimento do Raciocínio Logico em Crianças e Adolescentes**. FAI Faculdades. Itapiranga: 2014. 8 p. Disponível em < http://faifaculdades.edu.br/eventos/SEMIC/2014/5SEMIC/arquivos/resumos/RES19.pdf >. Acessado em 12/06/2016

DALLABONA, Sandra Regina; MENDES, Sueli Maria, Schmditt. **O Lúdico na Educação Infantil**: Jogar, brincar, uma forma de educar. Instituto Catarinense de Pós-Graduação. Itajaí: 2004. 13 p. Disponível em <http://www.posuniasselvi.com.br/artigos/rev04-16.pdf>. Acessado em 04/04/2016.

DIM, Cleyton A.; ROCHA, Francisco. **Uma Ferramenta Para Aprendizagem de Lógicas e Estímulo do Raciocínio e da Habilidade de Resolução de Problemas em um Contexto Computacional no Ensino Médio**. UFPA. Belém: 2011. 2 p. Disponível em <http://www.dimap.ufrn.br/csbc2011/anais/eventos/contents/WEI/Wei_Secao_6_Artigo_2_Dim.pdf>. Acessado em 28/02/2016.

FERREIRA, Tiago K. *et al*. **Potencializando a Aprendizagem da Lógica com Uso de Ambiente de Programação de Alto Nível**. Faculdade Meridional, Passo Fundo: 2012. 4 p. Disponível em <http://gepid.upf.br/senid/2012/anais/96177.pdf>. Acessado em 20/02/2016.

FILIPEFLOP, 2016, **Motor DC com driver ponte H L298N**. Disponível em: http://blog.filipeflop.com/motores-e-servos/motor-dc-arduino-ponte-h-l298n.html. Acessado em: 24/05/2016.

GRUNE, Dick; JACOBS, Ceriel J. H. **Parsing Techniques**: A Practical Guide. Chichester, Inglaterra:Ellis Horwood. 1990. -(http://dickgrune.com/Books/PTAPG_1st_Edition/)

MALTEMPI, Marcus V.; VALENTE, José A. **Melhorando e Diversificando a Aprendizagem via Programação de Computadores**. International Conference on Engeneering and Computer Education–ICECE. São Paulo: 2000. 3 p. Disponível em <https://www.researchgate.net/publication/267564687> - Acessado em 28/02/2016.

MOGENSEN, Torben. **Basics of Compiler Design**, 3ª Edição. University of Copenhagen:Lulu. 2000.

NEAR COMMUNICATIONS, **Modulação GFSK**. Disponível em < - https://sites.google.com/site/nearcommunications/modulacao-gfsk>. Acessado em 08/06/2016

THE ART OF ASSEMBLY LANGUAGE, **Art of Assembly Language Programming and HLA by Randall Hyde**. Disponível em <http://www.plantation-productions.com/Webster/www.artofasm.com/index.html>. Acessado em 28/02/2016.

VALENTE, José A.,. **Por Quê o Computador na Educação?**. Unicamp/Nied. Campinas: 1993. 25 p. Disponível em <http://xa.yimg.com/kq/groups/23266122/1283528405/name/Por+Qu%C3%AA+o+Computador+na+Educa%C3%A7%C3%A3o.pdf>. Acessado em 03/03/2016

VALENTIM, Henryethe. **Um Estudo Sobre o Ensino-aprendizagem de Lógica de Programação**. Curitiba: UTFPR, 2000. 12 p. Trabalho de Conclusão do Curso - Programa de Pós-Graduação. Universidade Tecnológica Federal do Paraná.

VIDA DE SILÌCIO, 2016, **Módulo Ponte H L298N – Montando seu robô Arduino**. Disponível em: http://blog.vidadesilicio.com.br/arduino/modulo-ponte-h-l298n-arduino/. Acessado em: 24/05/2016.