

Compilador: análise léxica, sintática, semântica e geração de código da linguagem T++

Noemi Pereira Scherer¹

¹Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

{noemischerer13}@gmail.com

Abstract. *This meta-article describes the procedures and results of the development of a lexical, syntactic, semantic analyzer and intermediate code generation. The first one is responsible for verifying the input of character lines from a source code written T++, producing a sequence of lexical symbols. The second analyzes the source code in T++ and returns an abstract tree. The third runs through this tree in order to find and report errors in the language. And the last one uses Python LLVM to generate and run parsed code. For the creation of these analyzers, a Python language library called PLY and YACC was used.*

Resumo. *Este meta-artigo descreve os procedimentos e resultados do desenvolvimento de um analisador léxico, sintático, semântico e geração de código intermediário. O primeiro é responsável por verificar as entradas de linhas de caracteres de um código fonte escrito T++, produzindo uma sequência de símbolos léxicos. O segundo analisa o código fonte em T++ e retorna uma árvore abstrata. O terceiro percorre essa árvore com o objetivo de encontrar e informar erros na linguagem. Por fim, o quarto utiliza o LLVM do Python para gerar e executar o código analisado. Para a criação desses analisares, foi utilizado uma biblioteca da linguagem Python denominado PLY e YACC.*

1. Introdução

Um compilador é responsável pela tradução de um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina. De forma geral, um compilador não produz diretamente um código de máquina e sim um programa em linguagem simbólica (*assembly*) semanticamente equivalente ao código em linguagem de alto nível. Por meio de montadores, esse programa em linguagem simbólica é traduzida em uma linguagem de máquina [Ivan L. M. Ricarte 2014].

Um compilador executa dois tipos de atividade, para desempenhar suas tarefas. A primeira é a análise do código fonte, no qual a estrutura e significado do programa de alto nível são reconhecidos. A segunda é a síntese do programa equivalente em linguagem simbólica [Ivan L. M. Ricarte 2014].

Para exemplificar essas duas atividades, considere a Figura 1 na qual descreve um código na linguagem C, que para um compilador é uma sequência de caracteres em um arquivo texto. O primeiro passo é reconhecer que agrupamentos de caracteres têm significado para o programa, por exemplo, saber que `int` é uma palavra-chave da linguagem e que `a` e `b` são variáveis neste programa. Posteriormente, o compilador deve reconhecer que a sequência `int a` corresponde a uma declaração de uma variável inteira cujo identificador recebeu o nome `a`.

Figura 1. Exemplo de código na linguagem C

```
int a, b, valor;  
a = 10;  
b = 20;  
valor = a*(b+20)
```

A primeira análise é a léxica, que é um processo que analisa a entrada de linhas de caracteres (código fonte de um programa) e produz uma sequência de símbolos léxicos denominados *tokens*, sendo facilmente manipulado por um *parser* (leitor de saída) [Wikibooks 2018].

O analisador léxico lê cada caractere do programa fonte e, traduz em uma saída os *tokens*. Dessa forma, é possível reconhecer as palavras reservadas, constantes, identificadores e outras palavras que pertencem a linguagem de programação analisada. O analisador também pode executar outras tarefas como o tratamento de espaços, eliminação de comentários e contagem do número de linhas que o programa possui [Wikibooks 2018].

O analisador léxico funciona de duas maneiras o primeiro e segundo estado da análise. O primeiro é responsável por ler a entrada de caracteres mudando o estado em que esses se encontram. Quando o analisador encontra um caractere o qual ele não considera como correto, ele volta à última análise que foi aceita. No segundo são repassados os caracteres encontrados para produzir um valor. O tipo do léxico é combinado com seu valor constituindo um símbolo, que pode ser denominado parser.

A implementação de um analisador léxico requer uma descrição do autômato que reconhece as sentenças da gramática ou expressão regular de cada *token* que possui os seguintes procedimentos [Wikibooks 2018]:

- Estado inicial, que recebe como argumento a referência para o autômato e retorna o seu estado inicial;
- Estado final, que recebe como argumentos a referência para o autômato e a referência para o estado corrente. O procedimento retorna verdadeiro se o estado especificado é elemento do conjunto de estados finais do autômato, ou falso caso contrário; e
- Próximo estado, que recebe como argumento a referência para o autômato, para o estado corrente e para o símbolo sendo analisado. O procedimento consulta a tabela de transições e retorna o próximo estado do autômato, ou o valor nulo se não houver transição possível.

A segunda análise é a sintática (*parser*) que determina a estrutura gramatical dos *tokens* segundo uma determinada gramática formal. Ou seja, ela é um processo que determina se uma cadeia de símbolos léxicos pode ser gerada por uma gramática [Wikibooks 2013].

Nesta análise o compilador deve reconhecer que a sequência de *tokens* corresponde a comandos relacionados à linguagem [Ivan L. M. Ricarte 2014], como o código da Figura 1, o primeiro comando é a declaração de variáveis e os três outros são comandos de atribuições.

A análise sintática transforma um texto na entrada em uma estrutura de dados, como uma árvore. Por meio dessa análise obtêm-se um grupo de *tokens*, para que o analisador use um conjunto de regras para construir uma árvore sintática da estrutura [Wikibooks 2018].

Essa análise aceita linguagens livre de contexto, e existem duas formas de determinar se a entrada de dados pode ser derivada de um símbolo inicial com as regras de uma gramática formal [Wikibooks 2013]:

Descendente (*top-down*): um analisador pode iniciar com o símbolo inicial e transformá-lo na entrada de dados. O analisador inicia dos maiores elementos e os quebra em partes menores, como exemplo, analisador sintático LL.

Ascendente (*bottom-up*): um analisador pode iniciar com um entrada de dados e tentar reescrevê-la até o símbolo inicial. O analisador tenta localizar os elementos mais básicos, e então os maiores que contêm os elementos mais básicos, e assim por diante, como por exemplo, analisador sintático LR.

A terceira análise é a semântica, responsável por verificar os erros de semântica no código fonte e coletar informações necessárias para a próxima fase da compilação, a geração de código. O objetivo dessa análise é trabalhar no nível de inter-relacionamento entre partes distintas do programa [Wikibooks 2014].

Considere o seguinte exemplo de código em C (Figura 1):

Figura 2. Exemplo de código em C

```
int fl(int a, float b) {  
    return a%b;  
}
```

A tentativa de compilar esse código irá gerar um erro detectado pelo analisador semântico, mais especificamente pelas regras de verificação de tipos, indicando que o operador módulo % não pode ter um operador real [Ivan L. M. Ricarte 2003a].

Na análise semântica, são utilizadas verificações de tipos para certificar se um determinado operando recebe outro do mesmo tipo. Como exemplo, bem comum nas linguagens de programação, a análise semântica retorna um erro quando uma variável de tipo numérica (real ou inteira) recebe um valor do tipo texto (*string*) [Wikibooks 2014]. Também são verificados erros no código, os exemplos de erros semânticos mais comum são:

- Uma variável não declarada;
- Uma multiplicação entre tipos de dados diferentes;
- Atribuição de um literal para outro tipo, como um inteiro em uma *string* ou vice-versa.

Para guardar as informações sobre os nomes declarados no programa, a semântica utiliza uma tabela de símbolos (TS). A TS é referenciada toda vez que um nome é encontrado no programa fonte. Alterações são feitas na TS sempre que um novo nome ou nova informação sobre um já existente é obtida. A TS deve ser criada de forma a permitir

inserções e consultas da forma mais eficiente possível, além de permitir o seu crescimento dinâmico. Cada entrada na tabela é a declaração de um nome, que pode ser implementada como um registro (*struct*) contendo campos (nome, tipo, classe, tamanho, escopo, etc.) que a qualificam [Wikibooks 2014].

Por fim, gera-se um código intermediário para obter o resultado final. A análise léxica e sintática apresentam técnicas com forte embasamento teórico e conceitual para permitir reconhecer os símbolos e as expressões tipicamente utilizadas em linguagens de programação de alto nível. No entanto, a operação de um compilador requer mais que o simples reconhecimento da validade de um programa; é preciso gerar o código equivalente que será efetivamente executado pelos processadores [Ivan L. M. Ricarte 2003b].

A geração de código não associa-se diretamente para a linguagem *assembly* do processador-alvo. O analisador sintático gera um código para uma máquina abstrata, com uma linguagem próxima a *assembly* porém, independente de processadores específicos. Em uma segunda etapa de geração de código, esse código intermediário é traduzido para a linguagem *assembly* desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores [Ivan L. M. Ricarte 2003b].

A linguagem utilizada para a geração de um código em formato intermediário entre a linguagem de alto nível e a linguagem *assembly* representa, de forma independente do processador para o qual o programa será gerado, todas as expressões do programa original. Existem duas formas usuais para esse tipo de representação, a notação pós-fixa e o código de três endereços [Ivan L. M. Ricarte 2003b].

O gerador de código independente do LLVM é uma estrutura que fornece um conjunto de componentes reutilizáveis para traduzir a representação interna do LLVM para o código de máquina para um destino especificado - em formato de montagem ou em formato de código de máquina binário. O gerador de código independente de destino do LLVM consiste em seis componentes principais [LLVM 2018]:

- interfaces de descrição de alvo abstratas que capturam propriedades importantes sobre vários aspectos da máquina, independentemente de como elas serão usadas;
- classes usadas para representar o código que está sendo gerado para um destino. Essas classes devem ser abstratas o suficiente para representar o código da máquina para qualquer máquina de destino;
- Classes e algoritmos usados para representar código no nível do objeto, o MC Layer. Essas classes representam construções de nível de montagem, como rótulos, seções e instruções;
- Algoritmos independentes de alvos utilizados para implementar várias fases de geração de código nativo (alocação de registros, agendamento, representação de quadros de pilha, etc).
- Implementações das interfaces de descrição de destino abstratas para destinos específicos. Essas descrições de máquina fazem uso dos componentes fornecidos pelo LLVM e podem, opcionalmente, fornecer passagens personalizadas específicas do destino para construir geradores de códigos completos para um destino específico.
- Os componentes JIT independentes de destino. O LLVM JIT é completamente independente de destino.

1.1. Objetivo

O objetivo desse trabalho é desenvolver um compilador capaz de realizar a análise léxica, sintática, semântica e a geração de códigos fontes escrito na Linguagem T++, retornando uma árvore sintática abstrata, erros presentes na linguagem e os resultados do código.

2. A Linguagem T++

A linguagem T++ foi desenvolvida especialmente para ser utilizada na disciplina de compiladores. Ela é uma linguagem simples, contendo algumas palavras reservadas, símbolos e arranjos uni e bidimensionais. A Tabela 2 mostra todas as palavras reservadas e símbolos que a linguagem T++ permite.

Tabela 1: *Tokens* permitidos pela Linguagem T++

Lista de tokens	
Palavras Reservadas	Símbolos
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, vírgula
retorna	:= atribuição
até	< menor
leia	> maior
escreve	<= menor-igual
inteiro	>= maior-igual
notação científica	<>= diferença
	() abre e fecha parênteses
	: dois pontos
	[] abre e fecha colchetes
	{ } comentário
	ou-lógico
	&& e-lógico
	! negação
Fim lista de tokens	

A construção do código dessa linguagem é inteiramente em Português. Os números podem ser inteiros, flutuantes (notação científica ou não), e as declarações de variáveis devem obrigatoriamente começar com letras precedente de várias letras e/ou números.

Apesar de T++ ser uma linguagem simples, ela permite a execução de algoritmos complexos, como de ordenação. A Figura 3 demonstra o algoritmo de busca soma vetor na linguagem t++.

Figura 3. Exemplo do código BubbleSort na linguagem T++

```
inteiro: T
T:= 4
inteiro: V1[T]
inteiro somavet(inteiro: vet[], inteiro: tam)
    inteiro: result
    result := 0
    inteiro: i
    i := 0
    repita
        result := result + vet[i]
        i := i + 1
    até i = tam - 1
    retorna(result)
fim
inteiro principal ()
    inteiro: x
    x := somavet(V1,T)
    retorna(0)
fim
```

3. Análise léxica

3.1. Expressões Regulares e Autômatos

Uma expressão regular é responsável por identificar cadeia de caracteres de uma determinada linguagem, como palavras ou padrões. Elas são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular [AHO 2008].

Essa expressão pode ser associada com a aritmética, entretanto ao invés dela denotar um número (como $2+2$), a expressão regular denota uma linguagem regular. Por exemplo: $a0^+$, que resultará em $\{ "a0", "a00", "a000", \dots \}$ [AHO 2008].

As expressões regulares são utilizadas para a especificação léxica de uma determinada linguagem. Para sua demonstração são utilizados autômatos finitos determinísticos.

Um autômato finito é um modelo computacional composto de uma fita de entrada dividida em células, nas quais contém os símbolos das cadeia. A principal parte do modelo é um controle finito, o qual indica em qual estado o autômato se encontra [USP 2012].

O modelo do autômato apresenta um estado inicial e um conjunto de estados finais. O estado inicial refere-se ao início de funcionamento do modelo, dependendo do símbolo presente na fita o próximo estado será escolhido, enquanto que os estados finais indicam o término do processo com sucesso, ou seja, se a cadeia presente na fita de entrada tiver sido completamente lida e o modelo não se encontrar em um estado final, considera-se a cadeia não aceita, ou caso contrário, como aceita [USP 2012].

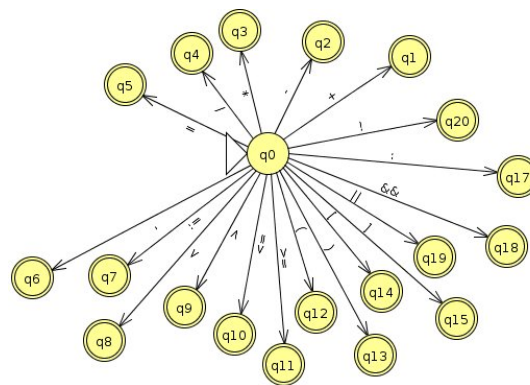
Para cada *token* permitido na Linguagem T++ é criado uma expressão regular,

o qual tem como objetivo analisar uma cadeia de caracteres no código. As expressões regulares utilizadas nesse trabalho estão descritas na Tabela 2.

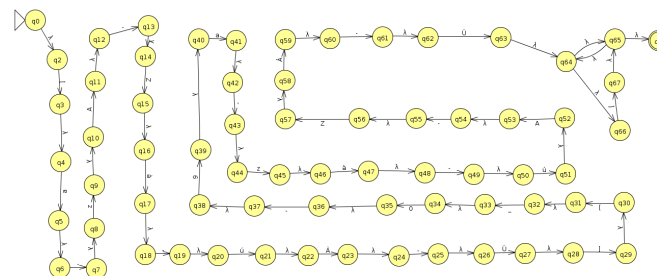
Tabela 2. Expressões regulares de cada token

Tokens	Expressão regulares
Soma	\+
Subtração	-
Multiplicação	*
Divisão	\/
Igual	\=
Vírgula	,
Atribuição	\:=+
Menor	\i
Maior	\i
Menor Igual	\i=
Maior igual	\i=
Abre parênteses	\(
Fecha parênteses	\)
Abre colchetes	\[
Fecha Colchetes	\]
Dois pontos	:
E lógico	&&
OU lógico	
Negação	!
ID	[a-zA-Zà-úÀ-Ú][_0-9a-zA-Zà-úÀ-Ú]*
Notação Científica	[0-9]+(\.[0-9]+)*(e E)(\+ \-)*[0-9]+(\.[0-9]+)*
Flutuante	[0-9]+(\.[0-9]+)(e(\+ \-)?(d+))?
Inteiro	[0-9]+
Comentário	{ [^ \ { ^ \ }] }
Nova Linha	\n+

A ordem de execução de cada expressão regular é representada em forma de autômatos. Os autômatos mais simples são dos operadores (soma, subtração, divisão, multiplicação, abre e fecha parênteses e colchetes, atribuição, igualdade, vírgula, dois pontos, ou e e-lógico e negação) que são representados na Figura 4.



Os autômatos da expressão regular do ID (identificador) estão representados de duas formas, uma resumida (Figura 5), e outro detalhada (Figura 6). O seu objetivo é identificar qualquer palavra ou letra presente no código, podendo ser letras maiúsculas, minúsculas, com ou sem assento.



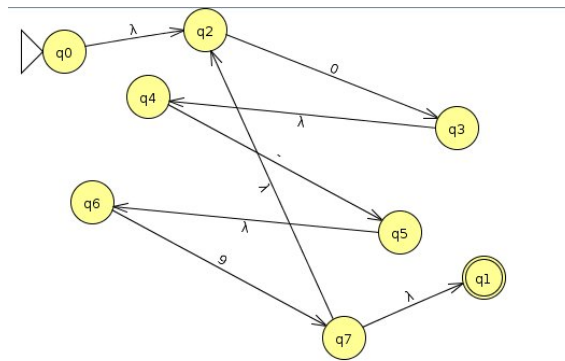


Figura 7. Autômato dos números inteiros.

O autômato da expressão regular dos números flutuantes está representado na Figura 8. Seu objetivo é identificar qualquer número que esteja separado por ponto no código, como 22.5 ou 0.22.

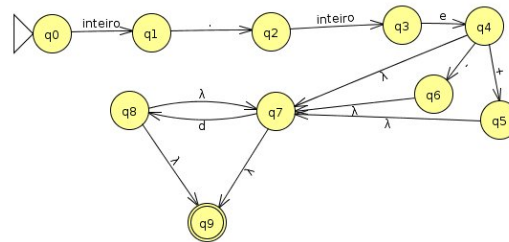


Figura 8. Autômato dos números flutuantes.

O autômato da expressão regular dos números flutuantes em notação científica está representado na Figura 9. Seu objetivo é identificar valor que possuem expoente no código, como 10E20 ou 10e-20.

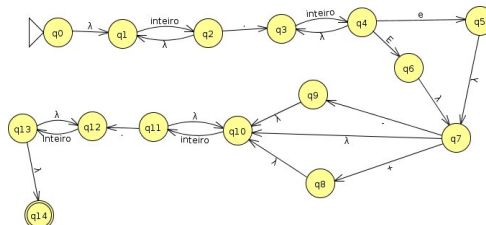


Figura 9. Autômato dos números flutuantes em notação científica.

Os autômatos da expressão regular dos comentários estão representados de duas formas, uma resumida (Figura 10) e outra detalhada (Figura 11). Os comentários na linguagem T++ são representadas por qualquer palavra que esteja entre '{}', como exemplo: {Esse é um comentário }.

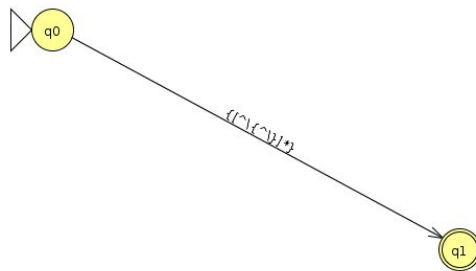


Figura 10. Autômato resumido do comentário.

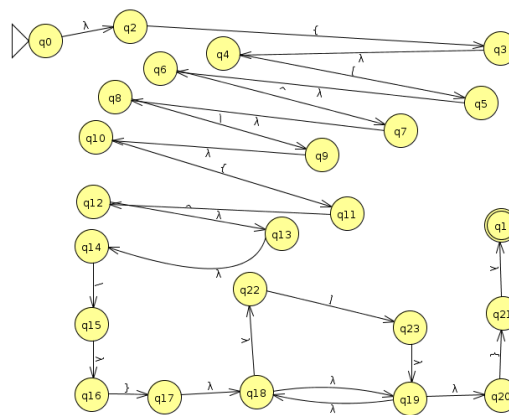


Figura 11. Autômato completo do comentário.

Por fim, o autômato da expressão regular da nova linha está representado representado na Figura 12. Seu objetivo é encontrar qualquer linha no código denotado por `\n`.

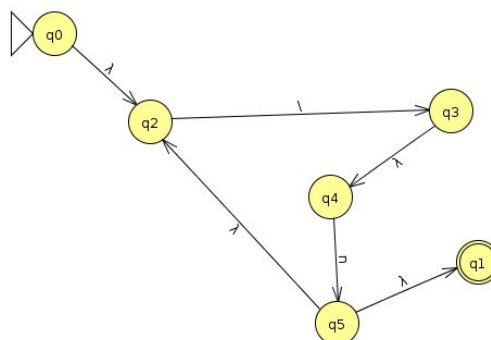


Figura 12. Autômato da nova linha.

4. Análise sintática

4.1. Descrição da gramática BNF

A *Backus-Naur Form* (BNF) é uma forma matemática de descrever uma linguagem, de modo que defina a gramática da linguagem sem ambiguidade ou divergência

[Lucas Thomaz 2009].

Primeiramente é definido um símbolo inicial para a gramática (programa) e depois são criadas regras (produções) para substituição desse símbolo por outro [Lucas Thomaz 2009].

A produção de uma regra segue a lógica de que o símbolo da esquerda do ':=’ pode ser substituído por um símbolo a direita. As alternativas são separadas por — (ou).

As produções criadas e utilizadas para o desenvolvimento da análise sintática estão descritas na Tabela 3.

Tabela 3: Produções BNF.

Descrição BNF	
programa :=	lista_declaracoes
lista_declaracoes :=	lista_declaracoes declaracao declaracao error
declaracao:=	declaracao_variaveis inicializacao_variaveis declaracao_funcao
declaracao_variaveis :=	tipo DOIS_PONTOS lista_variaveis
declaracao_variaveis :=	tipo DOIS_PONTOS error
inicializacao_variaveis :=	atribuicao
lista_variaveis :=	lista_variaveis VIRGULA var var
var :=	ID ID indice
indice :=	indice ABRE_COL expressao FECHA_COL ABRE_COL expressao FECHA_COL
indice :=	indice ABRE_COL error FECHA_COL ABRE_COL error FECHA_COL error FECHA_COL ABRE_COL error indice error FECHA_COL indice ABRE_COL error
tipo :=	INTEIRO FLUTUANTE
declaracao_funcao :=	tipo cabecalho cabecalho
cabecalho :=	ID ABRE_PAR lista_parametros FECHA_PAR corpo FIM
cabecalho :=	ID ABRE_PAR lista_parametros FECHA_PAR corpo error
lista_parametros :=	lista_parametros VIRGULA parametro parametro vazio
parametro :=	tipo DOIS_PONTOS ID parametro ABRE_COL FECHA_COL
corpo :=	corpo acao

Continuação descrição BNF	
	vazio
acao :=	expressao declaracao_variaveis se repita leia escreva retorna
se :=	SE expressao ENTAO corpo FIM SE expressao ENTAO corpo SENAO corpo FIM
se_error :=	SE expressao error corpo FIM error SENAO corpo FIM
repita :=	REPITA corpo ATE expressao
repita_error :=	REPITA corpo error
atribuicao :=	var ATRIBUT expressao
leia :=	LEIA ABRE_PAR var FECHA_PAR
escreva :=	ESCREVA ABRE_PAR expressao FECHA_PAR
retorna :=	RETORNA ABRE_PAR expressao FECHA_PAR
expressao :=	expressao_logica atribuicao
expressao_logica :=	expressao_simples expressao_logica operador_logico expressao_simples
expressao_simples :=	expressao_aditiva expressao_simples operador_relacional expressao_aditiva
expressao_aditiva :=	expressao_multiplicativa expressao_aditiva operador_soma expressao_multiplicativa
expressao_multiplicativa :=	expressao_unaria expressao_multiplicativa operador_multiplicacao expressao_unaria
expressao_unaria :=	fator operador_soma fator operador_negacao fator
operador_relacional :=	MENOR MAIOR IGUAL DIFERENCA MENOR_IGUAL MAIOR_IGUAL
operador_soma :=	SOMA SUB
operador_negacao :=	NEGACAO
operador_logico :=	E_LOGICO

Continuação descrição BNF	
	OU_LOGICO
operador_multiplicacao :=	MULT DIVISAO
fator :=	ABRE_COL expressao FECHA_COL var chamada_funcao numero
numero :=	INTEIRO FLUTUANTE NOTACAO_CIENTIFICA
chamada_funcao :=	ID ABRE_PAR lista_argumentos FECHA_PAR
lista_argumentos :=	lista_argumentos VIRGULA expressao expressao vazio

5. PLY - Python Lex Yacc

O PLY é uma ferramenta da linguagem Python para construção de compiladores populares lex e yacc. O principal objetivo do PLY é permanecer fiel ao modo como as ferramentas lex/yacc tradicionais funcionam. Isso inclui fornecer validação extensiva de entradas, relatórios de erros e diagnósticos [David M. Beazley 2011].

Como o PLY foi desenvolvido principalmente como uma ferramenta de instrução, é possível notar a exigência em sua especificação de regras de *token* e gramática. Em parte, isso acontece para detectar erros comuns de programação feitos por usuários iniciantes.

O PLY consiste em dois módulos separados; lex.py e yacc.py, ambos encontrados em um pacote do Python chamado ply. O módulo lex.py é usado para dividir o texto de entrada em uma coleção de *tokens* especificados por uma coleção de regras de expressão regular. O yacc.py é usado para reconhecer a sintaxe da linguagem que foi especificada na forma de uma gramática livre de contexto [David M. Beazley 2011].

As duas ferramentas são destinadas a trabalhar juntas. O lex.py fornece uma interface externa na forma de uma função `token()` que retorna o próximo *token* válido no fluxo de entrada. Já a yacc.py chama isso repetidamente para recuperar *tokens* e invocar regras gramaticais. A saída de yacc.py é geralmente uma *Abstract Syntax Tree* (AST) [David M. Beazley 2011].

5.1. PLY para análise léxica

Para o desenvolvimento da análise léxica, foi utilizado apenas o módulo lex.py. Para isso, o primeiro passo foi instalar o ply no ambiente linux utilizando o seguinte comando:

```
pipe3 install ply
```

É importante ressaltar que a versão do python deve ser acima de 3, para aceitar caracteres com acento, pois na linguagem T++ existem palavras reservadas que obrigatoriamente possuem acentos, como então e senão.

Para utilizar a ferramenta no código fonte deve-se importar o ply por meio do seguinte comando:

```
import ply.lex as lex
```

No código fonte da análise léxica é necessário descrever quais são as palavras reservadas e os *tokens* da linguagem, como demonstra na Figura 13.

```
4 # Palavras reservadas
5 reservadas = {
6     'se': 'SE',
7     'então': 'ENTÃO',
8     'senão': 'SENAO',
9     'fim': 'FIM',
10    'repita': 'REPITA',
11    'vazio': 'VAZIO',
12    'até': 'ATE',
13    'leia': 'LEIA',
14    'escreva': 'ESCREVA',
15    'retorna': 'RETORNA',
16    'principal': 'PRINCIPAL',
17    'inteiro': 'INTEIRO',
18    'flutuante': 'FLUTUANTE'
19 }
20
21 # Lista de tokens
22 tokens = ['SOMA', 'SUB', 'MULT', 'DIVISAO', 'IGUAL', 'VIRGULA',
23          'ATRIBUICAO', 'MENOR', 'MAIOR', 'MENOR IGUAL', 'MAIOR IGUAL',
24          'ABRE PAR', 'FECHA PAR', 'DOIS PONTOS', 'ABRE COL', 'FECHA COL',
25          'E LOGICO', 'OU LOGICO', 'NEGACAO', 'ID', 'NOVA LINHA', 'COMENTARIO',
26          'NOTACAO_CIENTIFICA'] + \
27     list(reservadas.values())
```

Figura 13. Palavras reservadas e *tokens* no código fonte da análise léxica.

Para o reconhecimento das palavras reservadas e *tokens*, são definidas as expressões regulares de cada uma no código fonte por meio de regras e funções (Figura 14). As expressões regulares já foram definidas na Tabela 2.

```
25 # Regras de expressões regulares
26 t_SOMA = r'\+'
27 t_SUB = r'\-'
28 t_MULT = r'\*'
29 t_DIVISAO = r'\/'
30 t_IGUAL = r'\='
31 t_VIRGULA = r'\,'
32 t_ATRIBUICAO = r'\:='
33 t_MENOR = r'\<'
34 t_MAIOR = r'\>'
35 t_MENOR_IGUAL = r'\<='
36 t_MAIOR_IGUAL = r'\>='
37 t_ABRE_PAR = r'\('
38 t_FECHA_PAR = r'\)'
39 t_ABRE_COL = r'\['
40 t_FECHA_COL = r'\]'
41 t_DOIS_PONTOS = r'\:'
42 t_E_LOGICO = r'&&'
43 t_OU_LOGICO = r'\|\|'
44 t_NEGACAO = r'!'
45
46
47 def t_ID(t):
48     r'[a-zA-Zâ-úÁ-Ú][0-9a-zA-Zâ-úÁ-Ú]*'
49     t.type = reservadas.get(t.value, 'ID')
50     return t
51
52 def t_NOTACAO_CIENTIFICA(t):
53     r'[0-9]+(\.[0-9]+)*(e|E)(\+|\-)?[0-9]+(\.[0-9]+)*'
54     t.type = "NOTACAO_CIENTIFICA"
55     return t
```

Figura 14. Expressões regulares das palavras reservadas e *tokens* no código fonte da análise léxica.

5.2. PLY para análise sintática

Utiliza-se o `yacc.py` para analisar a sintaxe da linguagem. Ele utiliza uma técnica conhecida como analisador LR ou *shift-reduce parsing LR*. Essa análise LR é uma técnica *bottom up* que tenta reconhecer o lado direito de várias regras gramaticais.

Sempre que um lado direito válido é encontrado na entrada, os símbolos de gramática são substituídos pelo símbolo de gramática no lado esquerdo [David M. Beazley 2011].

A implementação do analisador LR é responsável por deslocar símbolos da gramática para uma pilha, analisando a pilha e o próximo *token* de entrada para padrões que correspondam a uma das regras gramaticais [David M. Beazley 2011]. Como exemplo, considere a expressão $3 + 5 * (10 - 20)$ e a gramática definida na Figura 15, a saída utilizando o analisador é demonstrada na Figura 16.

```

expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )

```

Figura 15. Exemplo de regras gramaticais.

Step	Symbol Stack	Input Tokens	Action
1		3 + 5 * (10 - 20)\$	Shift 3
2	3	+ 5 * (10 - 20)\$	Reduce factor : NUMBER
3	factor	+ 5 * (10 - 20)\$	Reduce term : factor
4	term	+ 5 * (10 - 20)\$	Reduce expr : term
5	expr	+ 5 * (10 - 20)\$	Shift +
6	expr +	5 * (10 - 20)\$	Shift 5
7	expr + 5	* (10 - 20)\$	Reduce factor : NUMBER
8	expr + factor	* (10 - 20)\$	Reduce term : factor
9	expr + term	* (10 - 20)\$	Shift *
10	expr + term *	(10 - 20)\$	Shift (
11	expr + term * (10 - 20)\$	Shift 10
12	expr + term * (10	- 20)\$	Reduce factor : NUMBER
13	expr + term * (factor	- 20)\$	Reduce term : factor
14	expr + term * (term	- 20)\$	Reduce expr : term
15	expr + term * (expr	- 20)\$	Shift -
16	expr + term * (expr -	20)\$	Shift 20
17	expr + term * (expr - 20)\$	Reduce factor : NUMBER
18	expr + term * (expr - factor)\$	Reduce term : factor
19	expr + term * (expr - term)\$	Reduce expr : expr - term
20	expr + term * (expr)\$	Shift)
21	expr + term * (expr)	\$	Reduce factor : (expr)
22	expr + term * factor	\$	Reduce term : term * factor
23	expr + term	\$	Reduce expr : expr + term
24	expr	\$	Reduce expr
25		\$	Success!

Figura 16. Exemplo do analisador LR.

De acordo com a Figura 16, uma máquina de estado subjacente e o *token* de entrada atual determinam o passo seguinte. Se o próximo *token* estiver descrito em uma regra gramatical válida, ele é transferido para a pilha. Se a parte superior da pilha conti-

ver um lado direito válido de uma regra gramatical, ela é "reduzida" e os símbolos substituídos pelo símbolo à esquerda. Se o *token* de entrada não puder ser deslocado e o topo da pilha não corresponder a nenhuma regra gramatical, ocorre um erro de sintaxe. Uma análise é considerada bem-sucedida quando o analisador atinge um estado em que a pilha de símbolos esteja vazia e não exista mais *tokens* de entrada.

5.2.1. A implementação utilizando YACC

Para o desenvolvimento da análise sintática, foi utilizado o módulo `ply.yacc` e os *tokens* obtidos da análise léxica. Para isso, foi necessário realizar duas importações no código:

```
import ply.yacc as yacc
from lexer import Lexica
```

Cada regra gramatical é definida por uma função Python, no qual cada uma contém a especificação gramatical livre de contexto apropriada. As instruções que compõem o corpo da função implementam as ações responsáveis por adicionar valores na árvore abstrata, como mostra parte do código na Figura 17.

```
def p_programa(self, p):
    """
    programa : lista_declaracoes
    """
    p[0] = Tree('programa', [p[1]])

def p_lista_declaracoes(self, p):
    """
    lista_declaracoes : lista_declaracoes declaracao
                       | declaracao
                       | error
    """
    if (len(p) == 3):
        p[0] = Tree('lista_declaracoes', [p[1], p[2]])
    elif ( p.slice[1] == 'declaracao' ):
        p[0] = Tree('lista_declaracoes', [p[1]])
    elif ( p.slice[1] == 'error' ):
        print( "Erro: declaração incompleta! \n")

def p_declaracao(self, p):
    """
    declaracao : declaracao_variaveis
               | inicializacao_variaveis
               | declaracao_funcao
    """
    p[0] = Tree('declaracao', [p[1]])
```

Figura 17. Regras gramaticais representadas por meio de funções na linguagem Python.

Cada função aceita um único argumento *p* que é uma sequência contendo os valores de cada símbolo gramatical na regra correspondente. Os valores de *p[i]* são mapeados para símbolos de gramática, como o exemplo ***p[0] = Tree('declaracao'), [p[1]]***, no qual é adicionada na árvore o valor *p[1]*, que pode ser: *declaracao_variaveis*, *inicializacao_variaveis* ou *declaracao_funcao*.

A primeira regra definida na especificação *yacc* determina o símbolo de gramática inicial. Sempre que a regra inicial for reduzida pelo analisador e não houver mais entrada disponível, a análise será interrompida uma árvore abstrata contendo a gramática e valores será retornada.

A estrutura da árvore abstrata é criada em uma classe denominada *Tree* (Figura 18). Os símbolos encontrados pelo analisador sintático são adicionados nessa árvore, que é imprime todos seus nós ao final da análise por meio de uma função chamada *prinTree* (Figura 19).

```
class Tree:
    def __init__(self, type_node, child=[], value=''):
        self.type = type_node
        self.child = child
        self.value = value

    def __str__(self):
        return self.type
```

Figura 18. Estrutura da classe *Tree* em Python.

```
def prinTree(node, level=" "):
    if node != None and node.child != None:
        print("%s %s %s" % (level, node.type, node.value))
        for son in node.child:
            if (node.child != None):
                prinTree(son, level + " ")
```

Figura 19. Função responsável por imprimir a árvore abstrata em python.

Para capturar os erros de sintaxe encontrados, existe um denominada *p_error* que retorna a linha e a coluna do código onde ocorreu o erro, como demonstra a Figura 20.

```
def p_error(self, p):
    print(p)
    if p:
        print("Erro sintático: '%s', linha %d" % (p.value, p.lineno))
        p.lexer.skip(1)
        #exit(1)

    else:
        yacc.restart()
        print('Erro sintático: definições incompletas!')
        p.lexer.skip(1)
        #exit(1)
```

Figura 20. Função *error* em Python.

Para construir o analisador, é necessário chamar a função *yacc.yacc()*, no

qual examina o módulo e tenta construir todas as tabelas de análise de LR para a gramática especificada.

6. Análise Semântica

A análise semântica analisa possíveis erros no código da linguagem T++, percorrendo a árvore abstrata gerada pela análise sintática. Para isso, foi necessário importar o código da análise sintática e na função inicial obter a árvore gerada por ela:

```
from syntax import Syntax  
self.tree = Syntax(code).ast
```

Para verificar a existências de erros, foi analisado cada nó filho (saída da árvore), percorrendo suas instruções até chegar em alguma instrução no código, como declaração e inicialização de variáveis ou chamadas de funções, verificando a existência de alguma inconsistência não condizente com a estrutura permitida na linguagem. À medida que essas instruções são percorridas, ao identificar variáveis ou funções, elas são adicionadas em uma tabela de símbolos, armazenando seus valores que auxiliam a definir se a ação é ou não permitida no código.

A tabela de símbolos é um dicionário de dados responsável por armazenar todas as variáveis e funções presente no código e seus atributos, como o escopo, nome, tipo e quantidade de parâmetros, em caso de funções. A seguir é demonstrado um exemplo da tabela de símbolo:

```
{'principal': ['funcao', 'principal', 'void',  
'inteiro', 0],  
'global-a': ['variavel', 'a', False, 'flutuante',  
1],}
```

Para as funções, como a `principal`, na primeira posição do vetor é armazenado o atributo (função), na segunda o nome da função, na terceira armazena os tipos dos parâmetros (se tiver), na quarta armazena o tipo da função e a na sexta um valor 0 e 1, sendo 1 função utilizada e 0 não utilizada.

Já para as variáveis, como a `global-a`, significa que ela possui escopo global, e a primeira posição do vetor armazena se é variável, a segunda o seu valor (a), a terceira se ela foi inicializada (True sim e False não), a quinta armazena o tipo da variável e a sexta um valor de binário que identifica se a variável foi utilizada.

Inicialmente, o primeiro nó filho será "programa", dessa forma a análise semântica inicia-se na função `programa`, recebendo a árvore como parâmetro. O próximo passo é verificar seu filho, que é "lista_declarações", chamando respectivamente a função com esse nome. A partir de então é verificado se existe declaração de variáveis, função ou inicialização de variáveis. Ao passo que identifica os nós filhos, é chamada a função que condiz com ele.

Em algumas dessas funções são realizadas a análise de erro, com o auxílio da tabela de símbolos. Para melhor compreensão, considere a função `verifica_var` demonstrada na Figura 21.

A função `verifca_var` é responsável por identificar e verificar se a variável

Figura 21. Análise semântica: Função `verifica_var` responsável por determinar erros das variáveis.

```
def verifica_var(self, node):
    nome = self.escopo + "-" + node.value
    apenasNome = node.value

    if nome not in self.tabelaSimbolos:
        nome = "global-" + node.value
        if nome not in self.tabelaSimbolos:
            print("Erro: Variável '" + node.value + "' não declarada")
        else:
            if self.tabelaSimbolos[nome][2] == False:
                print("Erro: Variável '" + apenasNome + "' não inicializada.")

    if nome in self.tabelaSimbolos:
        self.tabelaSimbolos[nome][4] = 1
        return self.tabelaSimbolos[nome][3]
```

(`node`) recebida por parâmetro foi declarada ou inicializada. Para isso é pesquisado na tabela de símbolos o escopo correspondente a variável, se é global ou se está em alguma função. Se a variável não estiver na tabela, é mostrado uma mensagem de erro informando que essa não foi declarada. Se ela estiver, é realizado uma nova verificação, na qual analisa se a posição 2 na tabela de símbolos é `False`, se sim, outra mensagem é informada dizendo que a variável é utilizada, porém não foi inicializada. Por fim, se a variável estiver na tabela, é atribuído um novo valor na posição 4, que irá identificar que essa variável está sendo utilizada no programa.

Todas as verificações seguem essa mesma ideia, verificar os atributos do nó filho na tabela de símbolos, analisa-los e retornar uma mensagem de erro, caso necessário. Apenas três funções são chamadas ao terminar de percorrer a árvore, passando apenas a tabela de símbolos, são elas `verifica_main`, `verifica_variaveis` e `verifica_funcoes`, cujo objetivo é, respectivamente, verificar se a função principal foi declarada, se as variáveis e as funções são utilizadas.

Além de realizar as verificações, durante o percurso dos nós, é criado uma nova árvore abstrata resumida, com o objetivo de adicionar apenas as variáveis, funções e expressões no código. Para isso, foi utilizado a biblioteca `Graphviz` do Python.

7. Geração de código intermediário LLVM

Na etapa da geração de código, a árvore sintática abstrata é percorrida novamente, com o objetivo de gerar um código intermediário (RI) utilizando o LLVM-IR da linguagem Python. Como auxílio, utiliza-se a tabela de símbolos obtidas na análise e a própria árvore obtida da análise semântica.

No código da geração, foi necessário importar as bibliotecas `LLVMLite`, a semântica e o subprocesso `call` (para a compilação do código), por meio dos seguintes comandos:

```

from llvmlite import ir
from semantica import *
from subprocess import call

```

Além das importações, foi necessário obter a árvore abstrata da semântica, a tabela de símbolos e criar o módulo llvm responsável pela geração por meio do `ir.Module()`. A Figura 22 demonstra essas importações e a chamada de outras funções auxiliares, como a `define_variaveis_auxiliares()` e `salva_arquivo`, com o objetivo de salvar e compilar a saída do código.

```

9 class Geracao():
10     def __init__(self, code, optz=True, debug=True):
11         s = Semantica(code.read())
12         self.tree = s.tree
13         self.tabelaSimbolos = s.tabelaSimbolos
14         self.cria_modulo()
15         self.define_variaveis_auxiliares()
16         self.geracao_codigo(self.tree)
17         self.salva_arquivo()
18
19
20     def cria_modulo(self):
21         self.modulo = ir.Module("ModuloLLVMLITE.bc")
22

```

Figura 22. Algumas chamadas de funções do código de geração.

A geração tem início na função `geracao_codigo`, que é responsável por receber a árvore da semântica e analisar os nós filhos, chamando funções que criam ações de acordo com os nós. Por exemplo, para gerar um código de declaração de variável: a `geracao_codigo` chama `lista_declaracoes` passando o próximo nó da árvore, que chama `declaracao`, no qual verifica se recebeu algum nó de declaração de variáveis, para então chamar a função `declaracao_variaveis`, responsável pela geração de código.

As estratégias utilizadas para a geração seguiram o padrão do LLVM. Algumas delas são, inicializar variáveis globais e locais utilizando o `ir.IntType(32)` ou `ir.FloatTpe()`; Carregar uma variável e dar store nela por meio do `builder.load(variável, nome)`, `builder.store(variavel, nome)`, respectivamente; Chamar um procedimento com `builder.call(função, valores)`; entre outras.

Para executar o código intermediário gerado, é preciso: ter instalado LLVM com versão superior a 3.4, ou possuir o executável do LLC; e ter a biblioteca `print_scanf.o`. No próprio código, existe uma função responsável por realizar todas as execuções e mostrar no terminal a saída. Essa função é `salva_arquivo` e parte dela está descrita na Figura 23.

Se desejar executar sem ser pelo executável, na primeira chamada `call` deve ser excluído o `./`.

Figura 23. Fatorial em T++.

```
(...)  
call("./llc_gera.ll --mtriple \"x86_64-unknown-linux-gnu\"",  
  
call("gcc -c_gera.s", shell=True)  
call("gcc -o_saidaGeracao_gera.o_print_scanf.o",  
  
call("./saidaGeracao", shell=True)
```

8. Os resultados

8.1. Resultados da análise léxica

Com o código fonte da análise léxica pronto, é possível executá-lo de forma que receba um arquivo contendo o código na linguagem T++, leia cada *token* do mesmo, e retorne um arquivo de saída identificando cada *token* encontrado, bem como sua linha e coluna.

Para executar o código pelo terminal utiliza-se o seguinte comando:

```
python3 lexer.py nomeDoArquivo.tpp
```

Considere a Figura 24 que demonstra um código feito em T++ responsável por calcular o fatorial de um número.

Figura 24. Fatorial em T++.

```
inteiro: n  
inteiro fatorial(inteiro: n)  
    inteiro: fat  
    se n > 0 então {não calcula se n > 0}  
        fat := 1  
        repita  
            fat := fat * n  
            n := n - 1  
        até n = 0  
        retorna(fat) {retorna o valor do fatorial de n}  
    senão  
        retorna(0)  
fim  
fim  
inteiro principal()  
    leia(n)  
    escreva(fatorial(n))  
    retorna(0)  
fim
```

Ao executar a análise léxica o código fatorial (Figura 24), tem-se a seguinte saída:

```

LexToken(INTEIRO, 'inteiro', 1, 0)
LexToken(DOIS_PONTOS, ':', 1, 7)
LexToken(ID, 'n', 1, 9)
LexToken(NOVA_LINHA, '\n\n', 1, 10)
LexToken(INTEIRO, 'inteiro', 3, 12)
LexToken(ID, 'fatorial', 3, 20)
LexToken(ABRE_PAR, '(', 3, 28)
LexToken(INTEIRO, 'inteiro', 3, 29)
LexToken(DOIS_PONTOS, ':', 3, 36)
LexToken(ID, 'n', 3, 38)
LexToken(FECHA_PAR, ')', 3, 39)
LexToken(NOVA_LINHA, '\n', 3, 40)
LexToken(INTEIRO, 'inteiro', 4, 45)
LexToken(DOIS_PONTOS, ':', 4, 52)
LexToken(ID, 'fat', 4, 54)
LexToken(NOVA_LINHA, '\n', 4, 57)
LexToken(SE, 'se', 5, 62)
LexToken(ID, 'n', 5, 65)
LexToken(MAIOR, '>', 5, 67)
LexToken(INTEIRO, '0', 5, 69)
LexToken(ENTAO, 'então', 5, 71)
LexToken(COMENTARIO, '{ não calcula se n >= 0 }', 5, 77)
LexToken(NOVA_LINHA, '\n', 5, 99)
LexToken(ID, 'fat', 6, 108)
LexToken(ATRIBUICAO, ':=', 6, 112)
LexToken(INTEIRO, '1', 6, 115)
LexToken(NOVA_LINHA, '\n', 6, 116)
LexToken(REPITA, 'repita', 7, 125)
LexToken(NOVA_LINHA, '\n', 7, 131)
LexToken(ID, 'fat', 8, 144)
LexToken(ATRIBUICAO, ':=', 8, 148)
LexToken(ID, 'fat', 8, 151)
LexToken(MULT, '*', 8, 155)
LexToken(ID, 'n', 8, 157)
LexToken(NOVA_LINHA, '\n', 8, 158)
LexToken(ID, 'n', 9, 171)
LexToken(ATRIBUICAO, ':=', 9, 173)
LexToken(ID, 'n', 9, 176)
LexToken(SUB, '-', 9, 178)
LexToken(INTEIRO, '1', 9, 180)
LexToken(NOVA_LINHA, '\n', 9, 181)
LexToken(ATE, 'até', 10, 190)
LexToken(ID, 'n', 10, 194)
LexToken(IGUAL, '=', 10, 196)
LexToken(INTEIRO, '0', 10, 198)
LexToken(NOVA_LINHA, '\n', 10, 199)

```

```
LexToken(RETORNA, ' retorna ', 11, 208)
LexToken(ABRE_PAR, ' ( ', 11, 215)
LexToken(ID, ' fat ', 11, 216)
LexToken(FECHA_PAR, ' ) ', 11, 219)
LexToken(COMENTARIO, ' { retorna ... } ', 11, 221)
LexToken(NOVA_LINHA, ' \n ', 11, 255)
LexToken(SENAO, ' senão ', 12, 260)
LexToken(NOVA_LINHA, ' \n ', 12, 265)
LexToken(RETORNA, ' retorna ', 13, 274)
LexToken(ABRE_PAR, ' ( ', 13, 281)
LexToken(INTEIRO, ' 0 ', 13, 282)
LexToken(FECHA_PAR, ' ) ', 13, 283)
LexToken(NOVA_LINHA, ' \n ', 13, 284)
LexToken(FIM, ' fim ', 14, 289)
```

É possível notar que o código da análise léxica avaliou cada palavra do código em T++, o classificou e retornou seu valor em um arquivo de saída. Isso é exatamente o objetivo da análise léxica, retornar todos os *tokens* que um determinado código possui.

8.2. Resultados da análise sintática

O código da análise sintática recebe um arquivo contendo o código na linguagem T++. Ele chama a análise léxica para obter os *tokens* e analisar se esses correspondem a gramática descrita.

```
1 inteiro: a[1024]
2
3 inteiro func(inteiro: a[])
4     retorna(0)
5 fim
6
7 inteiro principal()
8     inteiro: i,b
9     i:= 1
10
11     retorna(0)
12 fim
```

Figura 25. Código em T++.

Considere o código fatorial da Figura 25, a saída obtida quando executado a análise sintática desse código é uma árvore abstrata que está representada na Figura 32.

8.3. Resultados da análise semântica

O código da análise semântica recebe um arquivo contendo o programa na linguagem T++. Ele chama a análise sintática para obter a árvore abstrata gerada, analisando a estrutura e verificando se existem erros no programa. Como saída, tem-se a tabela de símbolos, possíveis mensagens de erros e uma árvore sintática abstrata anotada.

Como exemplo de saída, considere o código descrito na linguagem T++, especificado na Figura 26.

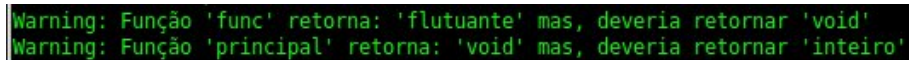
Figura 26. Exemplo de código na linguagem T++.

```
flutuante : a
inteiro : b

func ()
  a := 10.2
  retorna (a)
fim

inteiro principal ()
  b := 5
  func ()
fim
```

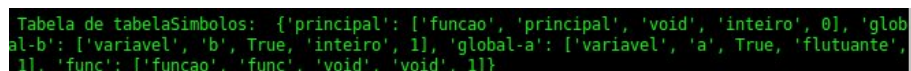
A saída obtida ao executar a análise semântica retorna um erro e um aviso. O primeiro indica que a função `func` deveria retornar vazio, mas retorna uma variável do tipo flutuante. Já o segundo é que a função `principal` deveria retornar inteiro, mas não retorna nenhum valor. A Figura 27 mostra essa saída obtida pelo terminal.



```
Warning: Função 'func' retorna: 'flutuante' mas, deveria retornar 'void'
Warning: Função 'principal' retorna: 'void' mas, deveria retornar 'inteiro'
```

Figura 27. Saída da análise semântica: erros.

Além das mensagens de erros, também foi obtido como saída a tabela de símbolos correspondente ao código. A variável `b` foi armazenada como escopo global, tendo como características, tipo inteiro, sendo inicializada e utilizada. A variável global `a` também é inicializada e utilizada, porém possui tipo flutuante. Já a função `func`, não possui tipo nem listas de parâmetros e é utilizada. Por fim, a função `principal` é do tipo inteiro e também não possui parâmetros de entrada. A Figura 28 apresenta os valores da tabela de símbolos obtidos.



```
Tabela de tabelaSímbolos: {'principal': ['funcao', 'principal', 'void', 'inteiro', 0], 'global-b': ['variavel', 'b', True, 'inteiro', 1], 'global-a': ['variavel', 'a', True, 'flutuante', 1], 'func': ['funcao', 'func', 'void', 'void', 1]}
```

Figura 28. Saída da análise semântica: tabela de símbolos.

Por fim, é gerada e salvo em formato SVG uma árvore abstrata resumida da análise sintática (Figura 29).

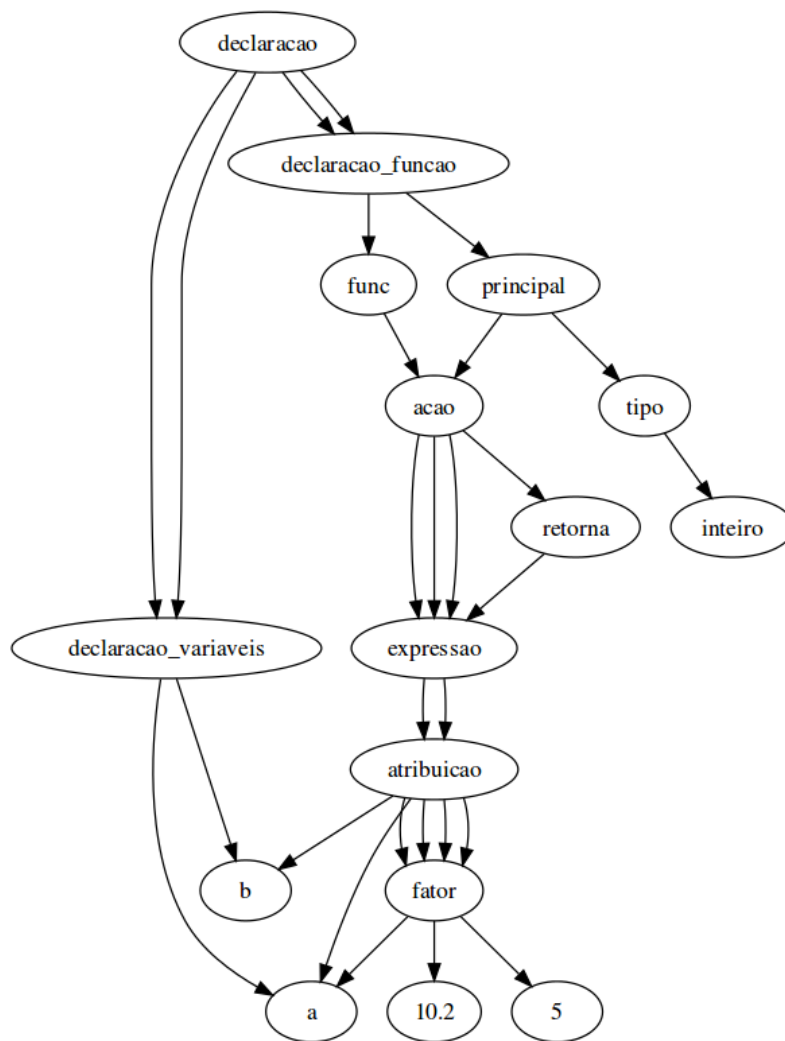


Figura 29. Saída da análise semântica: árvore sintática abstrata anotada.

8.4. Resultados da geração de código

O código da geração de código recebe um arquivo contendo o programa na linguagem T++. Ele chama a análise semântica para obter as mensagens e a árvore gerada. Como saída, é obtido um código IR gerado em LLVM e possíveis mensagens de erros da linguagem.

Como exemplo de saída, considere um código simples descrito na linguagem T++, que deve mostrar como saída o valor da variável `ret` (Figura 30).

A saída obtida ao executar a geração o módulo LLVM contendo todas as instruções necessárias para executar o programa e o resultado da variável `ret` que é o valor 1. A Figura 31 mostra a saída obtida no terminal.

```

1 {Condicional}
2 inteiro: a
3
4 inteiro principal()
5     inteiro: ret
6
7     a := 10
8     se a > 5 então
9         ret := 1
10    senão
11        ret := 10
12    fim
13    escreva(ret)
14    retorna(ret)
15 fim

```

Figura 30. Exemplo de código na linguagem T++.

```

; ModuleID = "ModuloLLVMLITE.bc"
target triple = "unknown-unknown-unknown"
target datalayout = ""

declare float @"escrevaFlutuante"(float %.1")
declare i32 @"escrevaInteiro"(i32 %.1")
declare float @"leiaFlutuante"()
declare i32 @"leiaInteiro"()

@"a" = common global i32 0, align 4
define i32 @"main"()
{
entry:
    %"principal-ret" = alloca i32
    store i32 10, i32* @"a"
    %"a_cmp" = load i32, i32* @"a", align 4
    %"se_a" = icmp sgt i32 %"a_cmp", 5
    br i1 %"se_a", label %"entao", label %"senao"
entao:
    store i32 1, i32* %"principal-ret"
    br label %"fim"
senao:
    store i32 10, i32* %"principal-ret"
    br label %"fim"
fim:
    %.8 = load i32, i32* %"principal-ret"
    %.9 = sitofp i32 %.8 to float
    %.10 = call float @"escrevaFlutuante"(float %.9)
    %.11 = load i32, i32* %"principal-ret"
    ret i32 %.11
}
1.000000

```

Figura 31. Saída da geração de código: Resultados obtidos pelo terminal.

Referências

AHO, A. V. (2008). *Compuladores: Princípios, técnicas e ferramentas*. SP: Pearson Addison-Wesley, 2nd edition.

- David M. Beazley (2011). Ply (python lex-yacc). <http://www.eng.utah.edu/~cs3100/lectures/l14/ply-3.4/doc/ply.html>. Acessado em 26-03-2018.
- Ivan L. M. Ricarte (2003a). Análise semântica. <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node71.html>. Acessado em 14-05-2018.
- Ivan L. M. Ricarte (2003b). Geração de código e otimização. <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node70.html>. Acessado em 03-06-2018.
- Ivan L. M. Ricarte (2014). Compiladores. <http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node37.html>. Acessado em 14-05-2018.
- LLVM (2018). The llvm target-independent code generator. <https://llvm.org/docs/CodeGenerator.html>. Acessado em 03-06-2018.
- Lucas Thomaz (2009). Compiladores: Bnf e ebnf. <https://lucasrthomaz.wordpress.com/2009/03/10/compiladores-bnf-e-ebnf/>. Acessado em 14-05-2018.
- USP (2012). Autômatos finito determinísticos. https://edisdisciplinas.usp.br/pluginfile.php/4159708/mod_resource/content/1/De_representacao_de_linguagens_ate_Automato_Finito_-_2012.pdf. Acessado em 27-03-2018.
- Wikibooks (2013). Construção de compiladores: Análise sintática. https://pt.wikibooks.org/wiki/Constru%C3%A7%C3%A3o_de_compiladores/An%C3%A1lise_sint%C3%A1tica. Acessado em 14-05-2018.
- Wikibooks (2014). Construção de compiladores: Análise semântica. https://pt.wikibooks.org/wiki/Constru%C3%A7%C3%A3o_de_compiladores/An%C3%A1lise_sem%C3%A2ntica. Acessado em 17-06-2018.
- Wikibooks (2018). Construção de compiladores: Análise léxica. https://pt.wikibooks.org/wiki/Constru%C3%A7%C3%A3o_de_compiladores/An%C3%A1lise_l%C3%A9xica. Acessado em 25-03-2018.

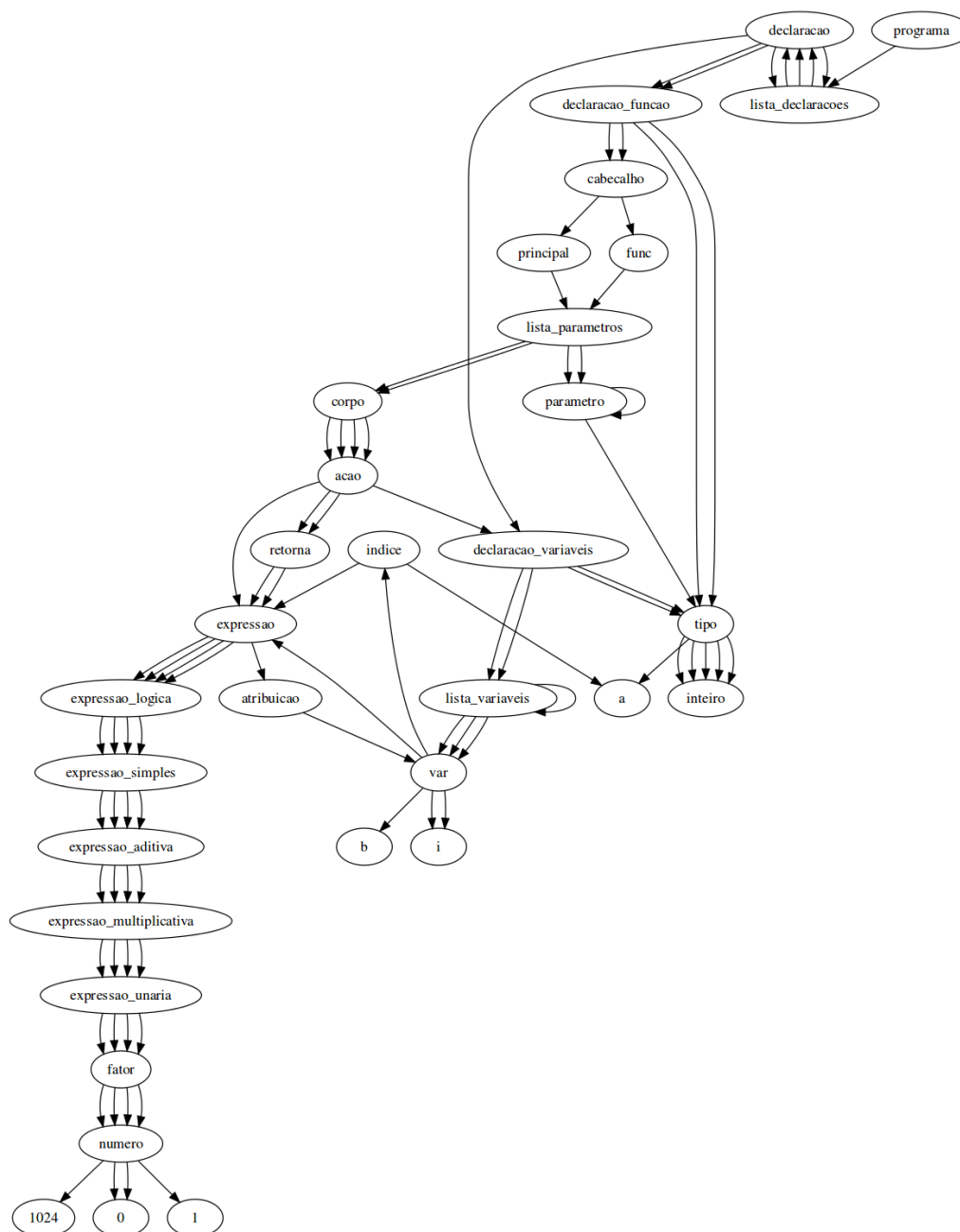


Figura 32. Resultado análise sintática: árvore sintática abstrata.