Implementando um Compilador para uma Linguagem de Programação Básica

Endrew R. T. Hang[1], Thiago P. B. Silva[1]

[1]Departamento de Ciência da Computação - Universidade do Estado de Santa Catarina(UDESC) - Joinville,SC - Brasil

{endrew.rth, thiago.pbs}@edu.udesc.br

Abstract. The objective of this work is to demonstrate, through the development of a basic programming language, the steps and foundations necessary to carry out the construction of a compiler. First the language will be analyzed on the lexical, syntactic and semantic aspects, then the construction of the intermediate code will be carried out.

Resumo. O objetivo deste trabalho é demonstrar por meio do desenvolvimento de uma linguagem de programação básica as etapas e os fundamentos necessários para realizar a construção de um compilador. Primeiramente a linguagem será analisada sobre os aspectos léxico, sintático e semântico, depois a construção do código intermediário será realizada.

1. Introdução

A linguagem desenvolvida pelos autores tem como base a linguagem C, trazendo algumas mudanças nos símbolos com o intuito de demonstrar como é feito o processo de análise e geração de código a partir dessa simbologia diferente. Para a etapa de análise léxica e sintática as ferramentas Flex e Bison foram utilizadas, elas utilizam a linguagem C como base e para a análise semântica uma tabela de símbolos usando HashMap foi utilizada em conjunto com essas ferramentas. Na geração do código intermediário foi utilizado o Jasmin, o mesmo foi implementado com o uso do Flex,Bison e C a partir dos resultados obtidos na fase de análise.

2. Fase de Análise

Esse tópico aborda a fase de análise do compilador e como dito anteriormente essa é dividida em três partes que foram separadas em três subtópicos trazendo os detalhes de desenvolvimento de cada etapa.

2.1 Análise Léxica

A análise léxica é feita com base nos símbolos definidos como pertencentes e portanto aceitos pela linguagem construída, eles são definidos no arquivo “trabalho_flex.l”. Caso

o usuário digite um token que não é aceito pela linguagem, configurando um erro léxico, o compilador irá informá-lo sobre o erro mostrando uma mensagem como pode ser visto nas Figuras 1 e 2.

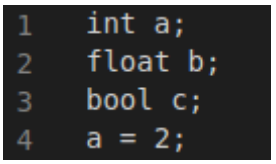

Figura 1: entrada

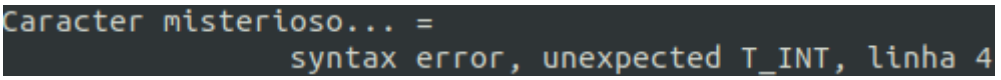

Figura 2: saída

A tabela de tokens que define os símbolos aceitos pela linguagem pode ser visualizada a seguir.

Símbolo	Nome	Símbolo	Nome
\n	T_NOVA_LINHA	const	T_CONST
if	T_IF	true	T_TRUE
then	T_THEN	false	T_FALSE
else	T_ELSE	int	T_TIPO_INT
while	T_WHILE	float	T_TIPO_REAL
do	T_DO	bool	T_TIPO_BOOL
for	T_FOR	sin	T_SIN
fun	T_FUN	cos	T_COS
return	T_RETURN	log	T_LOG
scan_int	T_SCAN	exit	T_EXIT
print	T_PRINT	{DIGITO}+	T_INT
//	comentários	void	T_TIPO_VOID

Símbolo	Nome	Símbolo	Nome
{DIGITO}+\.{DIGITO}+	T_REAL	,	T_VIR
{LETRA}({LETRA}|{DIGITO})*	T_VARIAVEL	;	T_PONTO_VIR
("\""|"\'").*("\""|"\'")	T_STR	:	T_RECEBE
+	T_MAIS	==	T_IGUAL
-	T_MENOS	!=	T_DIFERENTE
*	T_MULTIPLICA	<	T_MENOR
/	T_DIVIDE	<=	T_MENOR_IGUAL
%	T_RESTO	>	T_MAIOR
(	T_ABRE_PAR	>=	T_MAIOR_IGUAL
)	T_FECHA_PAR	!	T_NEGAR
{	T_ABRE_CH	&&	T_AND
}	T_FECHA_CH	||	T_OR

2.2 Análise Sintática

No caso da análise sintática além da definição dos tokens é necessário também a descrição da gramática, esta é definida no arquivo "trabalho_bison.y" e sua estrutura é retratada a seguir.

G = (Vn, Vt, P, programa)

Vn: {programa, comando, constantes, declaracao_funcao, declaracao, estrutura, atribuicao, funcao, expressao, comparacao, const_int, const_bool, const_real, tipo, tipo_fun, nomes, valor_variavel, chamada_funcao, parametros, estrutura_if, estrutura_for, estrutura_while, blocos, else, argumento, return, realiza, func_realiza, nova_linha, comandos_blocos, numero}

Vt: {T_NOVA_LINHA, T_IF, T_THEN, T_ELSE, T_WHILE, T_DO, T_FOR, T_FUN, T_RETURN, T_SCAN, T_PRINT, T_CONST, T_TRUE, T_FALSE, T_TIPO_INT, T_TIPO_REAL, T_TIPO_BOOL, T_TIPO_VOID, T_SIN, T_COS, T_LOG, T_EXIT, T_INT, T_REAL, T_VARIAVEL, T_STR, T_MAIS, T_MENOS, T_MULTIPLICA, T_DIVIDE, T_RESTO, T_ABRE_PAR, T_FECHA_PAR, T_ABRE_CH, T_FECHA_CH, T_VIR, T_PONTO_VIR, T_RECEBE, T_IGUAL, T_DIFERENTE, T_MENOR, T_MENOR_IGUAL, T_MAIOR, T_MAIOR_IGUAL, T_NEGAR, T_AND, T_OR}

P:

programa:
| programa comando

comando: constantes
| declaracao_funcao
| declaracao
| estrutura
| atribuicao
| funcao
| T_EXIT
| expressao T_PONTO_VIR
| comparacao T_PONTO_VIR
| T_NOVA_LINHA

constantes: T_CONST T_TIPO_INT const_int
| T_CONST T_TIPO_BOOL const_bool
| T_CONST T_TIPO_REAL const_real

const_int: T_VARIAVEL T_RECEBE T_INT T_PONTO_VIR

const_bool: T_VARIAVEL T_RECEBE expressao T_PONTO_VIR

const_real: T_VARIAVEL T_RECEBE T_REAL T_PONTO_VIR

declaracao: tipo nomes T_PONTO_VIR

tipo: T_TIPO_INT
| T_TIPO_REAL
| T_TIPO_BOOL

nomes: T_VARIAVEL valor_variavel
| nomes T_VIR T_VARIAVEL valor_variavel

valor_variavel:
| T_RECEBE expressao

atribuicao: T_VARIAVEL T_RECEBE expressao T_PONTO_VIR

expressao: expressao T_MAIS expressao
| expressao T_MENOS expressao
| expressao T_MULTIPLICA expressao
| expressao T_DIVIDE expressao
| expressao T_RESTO expressao
| T_MENOS expressao
| T_ABRE_PAR expressao T_FECHA_PAR
| T_SIN T_ABRE_PAR expressao T_FECHA_PAR
| T_COS T_ABRE_PAR expressao T_FECHA_PAR
| T_LOG T_ABRE_PAR expressao T_FECHA_PAR
| T_INT
| T_REAL
| T_TRUE
| T_FALSE
| T_VARIAVEL

chamada_funcao: T_VARIAVEL T_ABRE_PAR T_FECHA_PAR
| T_VARIAVEL T_ABRE_PAR parametros T_FECHA_PAR

parametros: expressao
| parametros T_VIR expressao

comparacao: T_NEGAR expressao
| expressao T_IGUAL expressao
| expressao T_OR expressao
| expressao T_AND expressao
| expressao T_DIFERENTE expressao
| expressao T_MENOR expressao
| expressao T_MENOR_IGUAL expressao
| expressao T_MAIOR expressao
| expressao T_MAIOR_IGUAL expressao

estrutura: estrutura_if
| estrutura_for
| estrutura_while

estrutura_if: T_IF T_ABRE_PAR comparacao T_FECHA_PAR blocos else

estrutura_for: T_FOR T_ABRE_PAR expressao T_VIR expressao T_FECHA_PAR blocos T_PONTO_VIR

estrutura_while: T_WHILE T_ABRE_PAR comparacao T_FECHA_PAR blocos T_PONTO_VIR

declaracao_funcao: T_FUN tipo_fun T_VARIAVEL T_ABRE_PAR T_FECHA_PAR T_ABRE_CH fun_realiza return T_FECHA_CH T_PONTO_VIR
| T_FUN tipo_fun T_VARIAVEL T_ABRE_PAR argumento T_FECHA_PAR T_ABRE_CH fun_realiza return T_FECHA_CH T_PONTO_VIR

return: T_RETURN expressao T_PONTO_VIR nova_linha
| T_RETURN T_PONTO_VIR nova_linha

tipo_fun: T_TIPO_INT
| T_TIPO_REAL
| T_TIPO_BOOL
| T_TIPO_VOID

func_realiza:
| realiza;

nova_linha:
| T_NOVA_LINHA

argumento: argumento T_VIR tipo T_VARIAVEL
| tipo T_VARIAVEL

else: T_PONTO_VIR
| T_ELSE comando
| T_ELSE blocos

realiza: comandos_blocos
| realiza comandos_blocos

comandos_blocos: constantes
| declaracao
| estrutura
| atribuicao
| funcao
| T_EXIT

| expressao T_PONTO_VIR
| comparacao T_PONTO_VIR
| T_NOVA_LINHA

funcao: T_SCAN T_ABRE_PAR T_VARIAVEL T_FECHA_PAR T_PONTO_VIR
| T_PRINT T_ABRE_PAR numero T_FECHA_PAR T_PONTO_VIR

numero: T_INT
| T_REAL
| T_VARIAVEL

blocos: T_ABRE_CH realiza T_FECHA_CH

Caso um erro sintático seja identificado, ou seja, as estruturas definidas pela gramática não forem respeitadas pelo usuário, uma mensagem como ilustrada nas Figuras 3 e 4 será exibida.

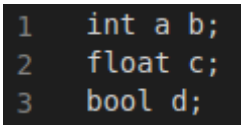

Figura 3: entrada

```
syntax error, unexpected T_VARIAVEL, expecting T_VIR or T_PONTO_VIR, linha 1
```

Figura 4: saída

2.3 Análise Semântica

Para que seja possível realizar a etapa da análise semântica uma tabela de símbolos foi criada utilizando um algoritmo de Hashmap, essa tabela permite o armazenamento de símbolos como palavras reservadas (if/for/int/print) e também de variáveis assim que elas forem declaradas. A partir dela é possível saber se uma variável já foi declarada ou se está sendo atribuído a ela um valor condizente com o seu tipo, entre outros erros semânticos que o usuário pode cometer.

Todas as funções envolvendo a tabela de símbolos foram feitas no arquivo “hash.c” com o header “hash.h”. A tabela é feita no array nomeado chain, este possui uma lista de data para armazenar colisões de hash. Cada data armazenada nessa tabela tem:

Informações da variável (Se possuir):

a. Tipo de variável (int = ‘i’, float = ‘f’, booleano = ‘b’);
b. Valor da variável;
c. Nome da variável;
d. Verificação se é constante (Se for = ‘1’, Se não = ‘0’);

Nome do símbolo (Se possuir);

As linhas que foi mencionada a data;

O tipo da informação (Símbolo = ‘s’, Variável ‘v’, Função = ‘f’).

Informações da função (Se possuir):

a. Nome da função
b. Parâmetros (Se possuir)
c. Número de Parâmetros

Ao final da execução do programa todos os símbolos armazenados são mostrados com suas devidas informações no arquivo “tabela_de_simbolos.out” e caso um erro semântico seja identificado pelo compilador uma mensagem será exibida como a visível nas Figuras 5 e 6. Um exemplo da tabela preenchida com alguns dados pode ser visto a seguir.

Tipo da informação	Nome do Símbolo	Tipo de variável	Valor da variável	Nome da variável	Verificação se é constante	As linhas que foi mencionada	Nome da Função	Parâmetros	Número de Parâmetros
s	(	NULL	NULL	NULL	NULL	1, 2	NULL	NULL	NULL
v	NULL	i	5	a	0	4, 5, 6	NULL	NULL	NULL
f	NULL	NULL	NULL	NULL	NULL	5, 6, 9	percorre Vetor	a, b	2

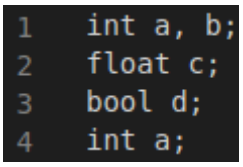

Figura 5: entrada

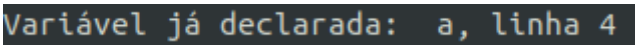

Figura 6: saída

3. Geração de Código

Na geração de código foram utilizados os arquivos feitos nas partes anteriores do trabalho para gerar um arquivo chamado de “bytecode.j”.
Este arquivo foi realizado com as regras de códigos da linguagem Jasmin que é basicamente um *Java Assembler,* isto é, converte um código para a linguagem binária de java com a extensão .class. Todas as regras de código foram encontradas com o auxílio da fonte (JVM... 2021), assim que todo o código foi gerado ele é executado dentro da *JVM - Java Virtual Machine.*

Alguns detalhes importantes sobre o funcionamento do programa, as funções devem ser declaradas no início do código, toda variável é única, ou seja, pode ser declarada apenas uma vez. O for é por padrão incrementado de um em um, porém caso seja necessário modificar a incrementação é possível por meio da manipulação dos parâmetros de entrada. Outra questão é que não é possível colocar booleanos dentro da comparação das estruturas do if e do while, apenas valores para serem analisados. Além do já citado, caso não se deseja

alterar o valor de uma variável na chamada da função, é necessário passar ela como parâmetro, pois todas as variáveis são globais.

Abaixo estão alguns exemplos de entradas e saídas executadas pelo programa, além do bytecode gerado para execução do Jasmim.

	Código	**Saída**	**Bytecode**
1	int i:180; print(i);	180	.method public static main([Ljava/lang/String;)V .limit stack 1000 .limit locals 1000 ldc 180 istore 1 getstatic java/lang/System/out Ljava/io/PrintStream; iload 1 invokevirtual java/io/PrintStream/println(I)V return .end method

2	int inicio:0, limite: 10;
	for(inicio, limite){
	  print(inicio);
	};
		0
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
			.method public static main([Ljava/lang/String;)V
			  .limit stack 1000
			  .limit locals 1000
			
			  ldc 0
			  istore 1
			  ldc 10
			  istore 3
			  iload 1
			  iload 3
			CmpLabel3:
			  swap
			  if_icmplt Label3
			  getstatic java/lang/System/out Ljava/io/PrintStream;
			  iload 1
			  invokevirtual java/io/PrintStream/println(I)V
			  iload 1
			  ldc 1
			  iadd
			  istore 1
			  iload 1
			  iload 3
			  goto CmpLabel3
			Label3:
			  ldc 0
			  istore 1
			
			  return
			.end method

3	int inicio:0, fim: 10;
	if(inicio < fim){
	  print(1);
	  for(inicio, fim){
	  print(2);
	  while(inicio == 10){
	  inicio : inicio + 1;
	  print(3);
	  };
	  };
	};
		1
		2
		2
		2
		2
		2
		2
		2
		2
		2
		2
		3
			.method public static main([Ljava/lang/String;)V
			  .limit stack 1000
			  .limit locals 1000
			
			  ldc 0
			  istore 1
			  ldc 10
			  istore 3
			  iload 1
			  iload 3
			  if_icmplt EQ0
			  ldc 0
			  goto END0
			EQ0:
			  ldc 1
			END0:
			  ifeq ELSE0
			  getstatic java/lang/System/out Ljava/io/PrintStream;
			  ldc "1"
			  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
			  iload 1
			  iload 3
			CmpLabel3:
			  swap
			  if_icmplt Label3
			  getstatic java/lang/System/out Ljava/io/PrintStream;
			  ldc "2"
			  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
			CmpWhile0:
			  iload 1
			  ldc 10
			  if_icmpeq EQ1
			  ldc 0
			  goto END1
			EQ1:
			  ldc 1
			END1:
			  ifeq While0
			  iload 1
			  ldc 1
			  iadd
			  istore 1
			  getstatic java/lang/System/out Ljava/io/PrintStream;
			  ldc "3"
			  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
			  goto CmpWhile0
			While0:

			iload 1 ldc 1 iadd istore 1 iload 1 iload 3 goto CmpLabel3 Label3: ldc 0 istore 1 goto ENDIF0 ELSE0: ENDIF0: return .end method

4	fun int func(int a, int b){
	int retorno: a + b;
	return retorno;
	};
	
	int valor: func(1,2);
	print(valor);
		3	.method public static main([Ljava/lang/String;)V
			.limit stack 1000
			.limit locals 1000
			
			ldc 1
			ldc 2
			invokestatic java_class.func(II)I
			istore 11
			getstatic java/lang/System/out Ljava/io/PrintStream;
			iload 11
			invokevirtual java/io/PrintStream/println(I)V
			
			return
			.end method
			
			.method public static func(II)I
			.limit stack 1000
			.limit locals 1000
			iload 0
			istore 5
			iload 1
			istore 7
			iload 5
			iload 7
			iadd
			istore 9
			iload 9
			
			ireturn
			.end method

5	int valorint;
	float seno, cosseno, logaritmo;
	
	seno: sin(77);
	print(seno);
	cosseno: cos(77);
	print(cosseno);
	logaritmo: log(77);
	print(logaritmo);
			.method public static main([Ljava/lang/String;)V
			.limit stack 1000
			.limit locals 1000
			
			ldc 0
			istore 1
			dconst_0
			dstore 3
			dconst_0
			dstore 5
			dconst_0
			dstore 7
			ldc 77
			i2d
			invokestatic java/lang/Math.sin(D)D
			dstore 3
			getstatic java/lang/System/out Ljava/io/PrintStream;
			dload 3
			invokevirtual java/io/PrintStream/println(D)V
			ldc 77
			i2d
			invokestatic java/lang/Math.cos(D)D
			dstore 5
			getstatic java/lang/System/out Ljava/io/PrintStream;
			dload 5
			invokevirtual java/io/PrintStream/println(D)V
			ldc 77
			i2d
			invokestatic java/lang/Math.log(D)D
			dstore 7
			getstatic java/lang/System/out Ljava/io/PrintStream;
			dload 7
			invokevirtual java/io/PrintStream/println(D)V
			
			return
			.end method

4. Destaque Para Compilação/Ambiente/Linguagem

Ambiente:

a. Linux com Ubuntu 20.04;
b. pacotes instalados : gcc, flex, bison, default-jre;

Compilação:

Foi elaborado um arquivo Makefile para organização de compilação, então utilizando o comando 'make all' é o suficiente para compilar e executar todos os arquivos necessários para o programa, desde que todos os arquivos enviados se encontrem na mesma pasta.

Makefile:

```
all:
   bison trabalho_bison.y -d
   flex trabalho_flex.l
   gcc -o trab hash.c jasmin.c trabalho_bison.tab.c lex.yy.c -lm
   ./trab
   clear
   java -jar jasmin-2.4/jasmin.jar bytecode.j
   java java_class
   make clean
```

- bison trabalho_bison.y -d : compila o arquivo bison nomeado de trabalho_bison.y;
- flex trabalho_flex.l: compila o arquivo flex nomeado de trabalho_flex.l;
- gcc -o trab hash.c trabalho1.tab.c lex.yy.c -lm: compila o arquivo C no arquivo binário 'trab' utilizando os arquivos gerados pelas compilações anteriores;
- ./trab: executa o código c compilado e gera o arquivo bytecode.j;
- clear: limpa o terminal para melhor visualização dos resultados;
- java -jar jasmin-2.4/jasmin.jar bytecode.j: executa o arquivo jar jasmin.jar contido dentro da pasta do trabalho para compilar o arquivo bytecode.j feito na geração de código;
- java java_class: executa o arquivo binário gerado pelo Jasmin;
- make clean: remove arquivos utilizados durante a execução que não são mais necessários.

5. Conclusão

Os objetivos solicitados foram alcançados com sucesso, porém algumas limitações de processamento ainda são encontradas, como exemplo dependendo do tamanho do código gerado pode haver falta de memória para a pilha da main. As duas maiores dificuldades encontradas foram para realizar a análise semântica e a geração de código. No primeiro caso o desenvolvimento da tabela de símbolos para que seja possível encontrar esse tipo

de erro foi mais complexo e exigiu mais da equipe. A outra dificuldade foi estruturar o equivalente em Jasmin na devida sequência de execução e tratar todas as possíveis possibilidades de código que o usuário pudesse digitar.

Gerar estruturas dentro de estruturas como um for dentro de outro for também foi uma adversidade, porém a equipe conseguiu realizar essa implementação. De modo geral, apesar dos obstáculos encontrados, todas as estruturas solicitadas foram implementadas e quase todas as possibilidades de códigos escritos pelo usuário são devidamente estruturadas.

6. Referências

Oracle. “Chapter 6. The Java Virtual Machine Instruction Set”, https://docs.oracle.com/javase/specs/jvms/se12/html/jvms-6.html. Acesso em: 9 abr. 2021.

Desconhecido. “Java bytecode”, https://en.wikipedia.org/wiki/Java_bytecode. Acesso em: 9 abr. 2021.

Desconhecido. “Java bytecode instruction listings”, https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings. Acesso em: 11 abr. 2021.

Desconhecido. “Arithmetic and Logic”, http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/co/jvm/jasmin/arithmetic.htm. Acesso em: 10 abr. 2021.

Desconhecido. “JVM and Jasmin Tutorial”, https://saksagan.ceng.metu.edu.tr/courses/ceng444/link/f3jasmintutorial.html. Acesso em: 04 abr. 2021.

Meyer, J. (1996) “Jasmin User Guide”, http://jasmin.sourceforge.net/guide.html. Acesso em: 03 abr. 2021.

Martins, F. (2021) “RicardoFM”, https://www.ricardofm.me/index.php/pt/. Acesso em: 02 abr. 2021.

Delamaro, M.E. (2004) “Como construir compiladores utilizando ferramentas Java”, Ed. Novatec, São Paulo.

Branco, G. A. J. and Tamae, R. Y. (2008) “Uma Breve Introdução ao Estudo e Implementação de Compiladores”, Revista Científica Eletrônica de Sistema de Informações, Garça-SP.

Price, A. M. A. and Toscani, S. S. (2001), Implementação de Linguagens de Programação: Compiladores, Sagra Luzzatto, segunda edição.

Desconhecido. “Jasmin Examples”, http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/co/jvm/jasmin/demos/demos.html. Acesso em: 01 abr. 2021.

Paxson, V. “Flex”, http://dinosaur.compilertools.net/flex/manpage.html. Acesso em: 25 fev. 2021.

Desconhecido. (2018) “xspdf”, https://www.xspdf.com/resolution/53145011.html#:~:text=Bison%20shift%2Freduce%20conflict,get%20bison%20to%20produce%20an%20. Acesso em: 26 fev. 2021.

Desconhecido. “Tabela de Símbolos”, https://erinaldosn.files.wordpress.com/2011/03/aula-5-tabelas-de-sc3admbolos.pdf. Acesso em: 26 fev. 2021.

Júnior, C. O. “Linguagens de Programação”, http://docs.fct.unesp.br/docentes/dmec/olivete/lp/arquivos/Aula4.pdf. Acesso em: 25 fev. 2021.

Drifter1. (2019) “Writing a simple Compiler on my own - Generating Code for Assignments (part 1)”, https://steemit.com/utopian-io/@drifter1/writing-a-simple-compiler-on-my-own-generating-code-for-assignments-part-1. Acesso em: 26 fev. 2021.

Log2base2. “Open hashing or separate chaining”, https://log2base2.com/algorithms/searching/open-hashing.html. Acesso em: 27 fev. 2021.