Documentação

Linguagem de Programação T++

Universidade Tecnológica Federal do Paraná

Leonardo Pontes Baiser

Sobre a Linguagem

Este documento especifica o detalhamento da implementação de um compilador para uma nova linguagem de programação, definida para complementação da teoria vista em sala na disciplina de Compiladores no curso de Ciência da Computação, ministrada pelo M.e Rodrigo Hübner.

O título dado a linguagem é T++, as raízes deste nome pode ser encontrado em características da linguagem de programação TINY, C++ e outras especificidades impostas pelo professor.

A linguagem é inteiramente escrita em português do Brasil, ou seja suas palavras reservadas, mensagens de alerta e erro. A linguagem é tipada, permitindo apenas dois tipos, o tipo inteiro e o tipo flutuante.

Tabela de Marcações da Linguagem t++

Tipo		Marca	Conotação	Expressão Regular
Palavra-chave		repita	Repetição	repita
		se	Condição	se
		então		então
		senão		senão
		fim		fim
		inteiro	Conjunto dos números inteiros	inteiro
		flutuante	Tipo numérico ponto flutuante	flutuante
		retorna	Retorno de funções	retorna
		até		até
		leia	Entrada de dados	leia
		escreve	Saída de dados	escreve
Símbolo	Operador	+	Operador soma	\+
		-	subtração	\-
		*	multiplicação	*
		/	divisão	/
	Comparador	=	igualdade	=
		<	menor que	<
		>	maior que	>
		<=	menor ou igual que	<=
		>=	maior ou igual que	>=
	Delimitador	(	abre parêntesis	\(
		)	fecha parêntesis	\)

	Outro	:	dois pontos	:
		,	vírgula	,
		:=	atribuição	:=
Outro		<comentários>		\{.*\}
		<números>		[0-9]+(\.[0-9]+)?
		<identificadores>	variáveis e nomes de função	[a-zA-Zá-ñÁ-Ñ][a-zA-Zá-ñÁ-Ñ0-9]*
		<nova linha>		\n+

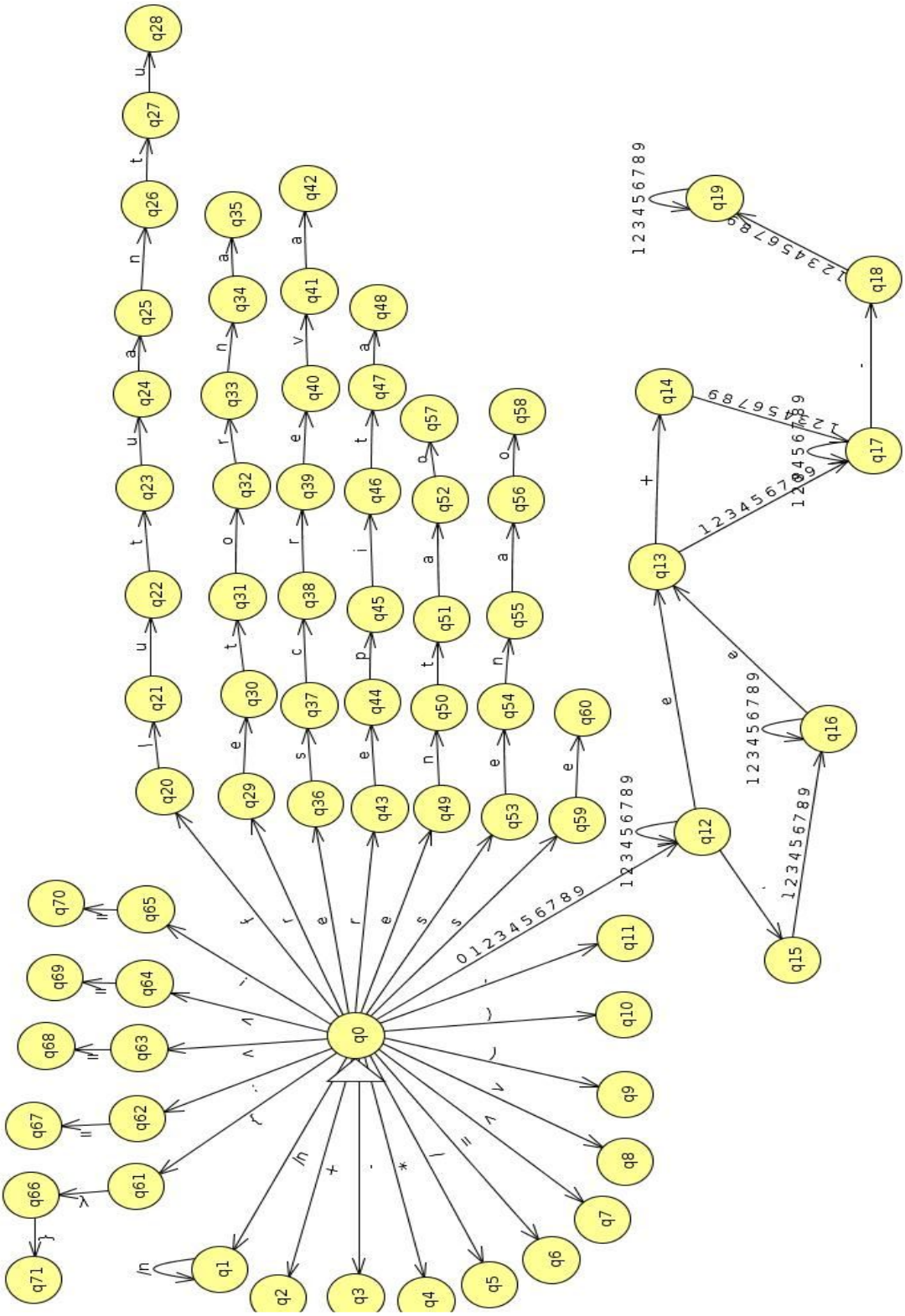

Figura 1. Autômato da Linguagem

Implementação

Para implementação foi escolhida a linguagem de programação Python na versão 3. Utilizando-se das bibliotecas RE(*Regular Expression*) e PLY (*Python Lex-Yacc*), a biblioteca de expressões regulares foi utilizada para o *parsing* do código fonte, tendo como saída um conjunto de *tokens*.

Análise Léxica

Na fase de análise léxica todas a linhas de códigos são analisadas encontrando os tokens gerados pela linguagem regular ou seja pelo autômato. Todo o caso não especificado pelo autômato casa com um estado *trap* levantando um erro léxico.

Análise Sintática

Na fase de análise sintática temos como entrada os pares (token, valor) oriundos da análise léxica e como saída temos um árvore sintática gerada a partir do casamento dos tokens conforme esperado na estrutura do código. Todo caso que quebre as regras estabelecidas no módulo do analisador sintático fará que ocorra uma mensagem de erro e a execução é interrompida. O formato de análise é o LALR, formato padrão do YACC.

Segue a representação da gramática na forma Backus-Naur

```
<top>                       ::= ""
                                | <procedimento>
                                | <top> <procedimento>

<procedimento>              ::= <declaracao>
                                | <funcao>

<declaracao>                ::= <tipo> : <identificador>
<funcao>                    ::= <prototipo> <corpo> fim
<prototipo>                 ::= <tipo> <identificador> ( <declaracao-argumentos> )

<declaracao-argumentos>     ::= ""
                                | <declaracao>
                                | <declaracao> , <declaracao_args>

<corpo>                     ::= ""
                                | <composicao> <corpo>

<composicao>            ::= <atribuicao>
                                | <declaracao>
                                | <chamada>
```

| <condicional>
| <repetição>
| <retorna>
| <leia>
| <escreva>

<atribuicao> ::= <identificador> := <expressao>

<chamada> ::= <identificador> (<expressao_args>)

<expressao_args> ::= ""
| <expressao>
| <expressao> , <expressao_args>

<repeticao> ::= repita <corpo> até <expressao>

<condicao> ::= se <expressao> então <corpo> fim
| se <expressao> então <corpo> senão <corpo> fim

<tipo> ::= inteiro
| flutuante

<expressao> ::= <expressao_binaria>
| <chamada>
| <expressao_unaria>
| <expressao_numerica>
| <expressao_identificador>
| (<expressao_parenteses>)

<expressao_binaria> ::= <expressao> = <expressao>
| <expressao> >= <expressao>
| <expressao> > <expressao>
| <expressao> <= <expressao>
| <expressao> < <expressao>
| <expressao> + <expressao>
| <expressao> - <expressao>
| <expressao> * <expressao>
| <expressao> / <expressao>

<expressao_unaria> ::= - <expressao>

<retorna> ::= retorna (<expressao>)

<leia> ::= leia (<expressao>)

<escreva> ::= escreve (<expressao>)

Análise Semântica

Na fase da semântica percorremos a árvore sintática abstrata (AST) gerada pelas fases de análise léxica e sintática, para realizar a análise sensível ao contexto e a geração da tabela de símbolos relevante para a próxima fase dada como geração de código. O princípio da análise semântica é verificar se existem erros de contexto. Caso existam estes erros de contexto o analisador deve levantar *warnings* e erros durante a análise. Segue descrito quais warnings e erros foram implementados para serem captados nesta análise:

1. Verificado se a quantidade de parâmetros reais de uma chamada de procedimento é igual a quantidade de parâmetros formais da sua definição.
2. Um *warning* é mostrado quando uma variável for declarada mais de uma vez.
3. Um *warning* é mostrado quando uma variável for declarada mas nunca utilizada.
4. Um warning é mostrado quando ocorrer uma coerção implícita de tipos (inteiro<->flutuante).
5. Um *warning* é mostrado quando uma função não foi utilizada.
6. Um warning é mostrado quando tipos diferentes são passados por parâmetros em uma chamada de função.
7. Um warning é mostrado quando a função principal não foi declarada.
8. Um erro é mostrado quando uma variável não foi declarada, tanto no escopo global quanto local.
9. Um erro é mostrado quando uma variável já foi declarada e é declarada novamente no código.
10. Um erro é mostrado quando existe uma tentativa de declarar uma variável com o mesmo nome de uma função.
11. Um erro é mostrado quando uma função já foi declarada.
12. Um erro é mostrado quando uma variável não foi inicializada.
13. Um erro é mostrado quando uma função não foi declarada.
14. Um erro é mostrado quando o número de parâmetros passados é diferente do número de parâmetros esperado.

Para que a próxima etapa do compilador possa ser concretizada na tabela de símbolos são guardados as seguintes informações para variáveis e funções:

variável
#classe, nome variavel, utilizada, atribuida, tipo

função
#classe, nome_func, argumentos, utilizada, tipo

parâmetro
#classe, nome, argumentos, utilizada, tipo, escopo