Análise Léxica da Linguagem C-IPL

Johannes Peter Schulte - 150132662

Universidade de Brasília, Brasília-DF, Brasil
johpetsc@gmail.com

1 Motivação

O trabalho apresentado tem como finalidade a criação de um analisador léxico, analisador sintático, analisador semântico e gerador de código intermediário da linguagem C-IPL. A linguagem C-IPL é baseada em um subconjunto da linguagem C, adicionando novos tipos de dados e operações com o intuito de facilitar operações com listas e fornecer um ambiente de desenvolvimento com listas mais prático e ágil. Listas são um tipo abstrato de dados que representam uma sequência e, diferente de vetores, possuem a vantagem de serem expandidas ou comprimidas. Dessa forma, a estrutura é de grande importância para várias aplicações e a linguagem C-IPL visa aprimorar sua implementação.

2 Análise Léxica

O analisador léxico foi implementado seguindo a descrição da linguagem C-IPL, onde alguns *tokens* fazem parte da linguagem C e o restante está relacionado às operações, operadores e tipos de dados de listas. Foram criadas expressões regulares referentes às definições mais simples, como dígitos e letras, das quais então, são construídas as definições mais complexas da linguagem, como os identificadores aceitos, tipos de listas e *strings*. Sempre que o analisador encontra um identificador, a tabela de símbolos deve ser consultada e feita uma busca pelo mesmo, e caso não encontrado, é adicionado em uma nova linha da tabela.

A nova linguagem possui, além dos tipos de dados para lista, operadores que retornam o primeiro e último elemento de uma lista, operadores como ":" e "%" que funcionam como construtores e destrutores para listas, e por fim funções para filtrar elementos e aplicar uma função aos elementos de uma lista.

O programa executa linha por linha, da primeira até o fim do arquivo. Para cada linha, é analisado caractere por caractere, onde este, ou o conjunto destes, é atribuído à expressão regular que os descreve na linguagem. Caso não seja encontrada nenhuma definição para tal, é considerado que há um erro e que aquilo não faz parte da linguagem.

Durante a análise, o programa procura sempre o maior conjunto de caracteres possíveis que se encaixam em alguma expressão regular definida, de forma que todos os componentes da linguagem podem ser encontrados durante a execução. Um exemplo disto é o caractere ">", que poderia ser confundido com a operação relacional, mas quando repetido, ">>", representa uma função para lista da nova linguagem.

3 Arquivos de Teste

Com a finalidade de testar o analisador léxico, foram criados quatro arquivos em linguagem C seguindo a nova linguagem a ser analisada. Cada arquivo de teste tem uma finalidade diferente, onde dois devem apresentar uma execução sem erros. Os testes podem ser encontrados no subdiretório *tests*.

- Arquivo de teste 1: O primeiro arquivo, **test1.c**, é um programa simples com o objetivo de testar, principalmente, operações e estruturas de controle de fluxo.
- Arquivo de teste 2: O segundo arquivo, **test2.c**, que foi dado como exemplo na descrição da linguagem, testa todos os lexemas, com foco nos relacionados a nova linguagem.
- Arquivo de teste 3: O terceiro arquivo, **test3.c**, tem como objetivo apresentar erros durante a análise. Seu código é igual ao primeiro, adicionando caracteres que não pertencem à linguagem. Erros são encontrados nas linhas 18, 16, 15, 14, 13 e 2.
- Arquivo de teste 4: O quarto arquivo, **test4.c**, também tem como objetivo apresentar erros. O código é igual ao terceiro, adicionando caracteres que não pertencem à linguagem e testando *strings* com formatação incorreta. Erros são encontrados nas linhas 50, 43, 30, 19, 18, 13, 12, 11, 6, 4, 2 e 1.

4 Compilação e Execução

O ambiente de desenvolvimento e suas ferramentas possuem as seguintes características:

- Sistema Operacional: Ubuntu, versão 20.04
- Flex: 2.6.4
- gcc: 9.3.0

Para compilar o analisador léxico e executar os arquivos de teste, no repositório principal, executar os comandos:

- $ flex -o src/lex.yy.c src/lex_analyser.l && gcc src/lex.yy.c -o tradutor
- $ ./tradutor tests/[arquivo de teste].c

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Pearson, second edition, 2006.

A
Gramática da Linguagem

A gramática foi construída seguindo o exemplo do livro "Compilers - Principles, Techniques and Tools" [ALSU06], utilizando os *tokens* descritos na Tabela 1. A gramática é inicializada pelo programa que está sendo executado (*program*) e precisa necessariamente iniciar pela função *main*. Para cada função existem blocos, declarações e expressões. Um bloco de função se faz de um conjunto de declarações. Declarações são delimitadas pelo ponto e vírgula (;) e são constituídas de expressões. A partir das expressões geramos os *tokens* da linguagem para efetuar operações, operadores, declarações, atribuições, chamadas de funções, controle de fluxo e os *tokens* terminais.

program → *main_dclr*
main_dclr → `type id` *func*
func_dclr → `type id` *func*
func_call → *func_dclr params*
func_call → *func_dclr*
func → *params* `brackets` *block* `brackets`
params → `brackets` *declar* `brackets`
block → *statement block*
block → *statement*
statement → *expr* `end`
statement → `line_comment | block_comment |` ϵ
expr → *block*
expr → *expr expr*
expr → *declar*
expr → `ass_op | ari_op | log_op | rel_op`
expr → `list_op | list_func`
expr → `flow_ctr` *expr*
expr → `brackets` *expr* `brackets`
expr → *val*
expr → `input | output`
`flow_ctr` → `if | else | for | return`
`input` → `read`
`read` → `brackets` *val* `brackets`
`output` → `write | writeln`
`write` → `brackets` *val* `brackets`
`writeln` → `brackets` *val* `brackets`
`ari_op` → *expr* `+` *val*
`ari_op` → *expr* `-` *val*
`ari_op` → *expr* `*` *val*
`ari_op` → *expr* `\` *val*
`log_op` → `!`*val*
`log_op` → *expr* `&&` *val*
`log_op` → *expr* `||` *val*

rel_op → *expr* > *val*
rel_op → *expr* < *val*
rel_op → *expr* >= *val*
rel_op → *expr* <= *val*
rel_op → *expr* != *val*
rel_op → *expr* == *val*
ass_op → *val* = *expr*
list_op → ? list
list_op → ! list
list_op → % list
list_op → *val* : list
list_func → func_call << list
list_func → func_call >> list
val → id
val → inumber | fnumber
val → NIL
val → literal
type → int | float
declar → type id
declar → separator | *val* | ϵ
declar →*declar declar*
declar → type list id
brackets → { | } | (|) | [|]
end → ;
separator → ,

Table 1. Léxico da linguagem.

Token	Definição regular	Exemplo de Lexemas
delim	[\s\v\t]	” ”, ”\t”
line_break	\n	”\n”
letter	[A-Za-z]	”a”, ”b”, ”c”
digit	[0-9]	”1”, ”2”, ”0”
id	{letter}({letter}|{digit}|_)*	”abc”, ”ab2”, ”ab_c”
inumber	{digit}+	”100”, ”101”, ”123”
fnumber	{digit}+(\.{digit}+)	”1.5”, ”2.3”, ”1.337”
types	int|float	”int”, ”float”
list	list	”list”
list_op	`[?!%:]`	”?”, ”!”, ”%”, ”:”
list_func	`[>]{2}[<]{2}`	”>>”, ”<<”
nil	NIL	”NIL”
brackets	`[\[\]\{\}\(\)]`	”[”, ”(”, ”{”
end	`[;]`	”;”
ari_op	`[+*/-]`	”+”, ”-”, ”*”
log_op	`[!]|[&]{2}|[|]{2}`	”!”, ”&&”, ””
rel_op	`[>]|[<]|(>=)|(<=)|(==)|(!-)`	”<”, ”>=”, ”!=”
ass_op	`(=)`	”=”
flow_ctr	if|else|for|return	”for”, ”if”, ”return”
input	read	”read”
output	write|writeln	”write”, ”writeln”
dquot	`[\"]`	” ”
separator	,	”,”
literal	`{dquot}(\\[^\n]|[^\\"\n])*{dquot}`	”teste”, ”read_list”, ”’string’ ”
line_comment	”//”.*`[^n]`	”//comment”
block_comment	”/*”`((\*+[^/*])|([^*]))**`”*/”	”/*comment*/”
rest	.	”@”, ”#”, ”.”