Análise 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 inserido o novo símbolo na próxima posição disponível, caso faça parte de uma declaração.

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.

Os *tokens* são criados pelo analisador léxico e enviados para a análise sintática. Sua estrutura deve apresentar o lexema associado ou *token* e sua localização no código, incluindo linha, coluna e escopo. O analisador léxico não interage diretamente com a tabela de símbolos, os identificadores declarados são inseridos pelo analisador sintático.

3 Análise Sintática

O analisador sintático tem a função de apresentar a árvore sintática abstrata e a tabela de símbolos, além de detectar os erros sintáticos da linguagem C-IPL, apresentando uma descrição do erro e sua localização no código.

Os *tokens* são recebidos do analisador léxico, numa estrutura contendo o lexema e sua localização no código. A análise sintática então é feita com a ferramenta de gerador de análise sintático Bison, seguindo a gramática da linguagem apresentada no apêndice A. O padrão seguido foi *bottom-up parser* LR(1) canônico.

As estruturas de dados criadas foram uma árvore, onde será armazenada a árvore sintática abstrata, e um vetor de *struct* representando os símbolos, formando a tabela de símbolos. A árvore foi implementada de forma que uma *struct* representa cada nó da árvore e seu valor, e aponta para quatro outros nós que dão origem às subárvores durante a análise sintática. A árvore então é formada por um vetor que armazena cada um dos nós criados pelo analisador. Cada símbolo da tabela de símbolos contém informações sobre o identificador, sua localização no código e sua forma de declaração.

As estruturas são preenchidas ao longo da análise sintática, onde para cada variável declarada, ou para cada função declarada, um novo símbolo é adicionado na próxima posição livre da tabela de símbolos. O analisador não possui nenhuma forma de verificação de declarações repetidas, cada declaração é inserida na tabela independente do seu identificador.

Durante a análise, a raiz da árvore inicializa junto ao *program* da gramática. Para cada derivação, é criado uma nova subárvore, onde seu nó possui o nome da derivação, e aponta para todas as derivações geradas a partir dela. Sempre que o analisador encontra um terminal da gramática, este representa uma folha da árvore, onde um novo nó é criado com o nome do terminal, que não aponta para nenhum outro nó.

A política de erros segue o padrão do analisador léxico, sempre que um erro é encontrado, a descrição e localização deste devem ser apresentadas, e o analisador deve prosseguir a análise para procurar por outros erros sintáticos. Em alguns casos é possível que o analisador não consiga se recuperar de um erro, por exemplo, caso não reconheça as declarações de funções, assim o conteúdo da função não será analisado.

4 Arquivos de Teste

Com a finalidade de testar o analisador sintático, 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, tem como objetivo testar as novas funcionalidades da 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 erros sintáticos nas estruturas de controle de fluxo e operações. Erros são encontrados na linha 5 coluna 10, linha 12 coluna 11, linha 17 coluna 9, linha 19 coluna 10, linha 22 coluna 13 e linha 24 coluna 18.
- Arquivo de teste 4: O quarto arquivo, **test4.c**, também tem como objetivo apresentar erros. O código é igual ao segundo, adicionando erros sintáticos nas declarações de variáveis e funções. Erros são encontrados na linha 1 coluna 5, linha 4 coluna 5, linha 7 coluna 11, linha 18 coluna 5, linha 22 coluna 8 e linha 24 coluna 9.

5 Compilação e Execução

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

- Sistema Operacional: Ubuntu, versão 20.04
- Flex: 2.6.4
- Bison 3.5.1
- Valgrind 3.15.0
- gcc: 9.3.0

Para compilar e executar os analisadores léxico e sintático, no repositório principal, executar os comandos:

Compilar e Executar:

- $ make

Compilar:

- $ make compile

Executar:

- $ make run
- ou
- $ ./tradutor tests/nome_arquivo.c

Valgrind: Compilar e Executar:

- $ make debug

Referências

[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 procura pelas declarações de variáveis e funções no escopo inicial. 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 → *program program_block* | *program_block*
program_block → *declar* `END` | *func_dclr*
func_dclr → *func* `LP` *params* `RP LB` *block* `RB`
func_dclr → *func* `LP RP LB` *block* `RB`
func_call → *id* `LP` *func_params* `RP`
func_call → *id* `LP RP`
func_params →*func_params* `SEPARATOR` *id* `id`
func → `TYPE ID`
func → `TYPE LIST ID`
params → *params* `SEPARATOR` *declar* | *declar*
declar → `TYPE ID`
declar → `TYPE LIST ID`
block → *block statement*
block → *statement*
statement → *expr* `END`
statement → *ass_op* `END`
statement → `LB` *block* `RB`
statement → *flow_ctr*
expr → *operation*
expr → *declar*
expr → *input*
expr → *output*
expr → *list_op*
expr → *list_func*
flow_ctr → *if_else* | *for* | *return* `END`
list_op → *list_con*
list_op → *list_oper*
if_else → `IF LP` *operation* `RP` *statement*
if_else → `IF LP` *operation* `RP` *statement* `ELSE` *statement*
for → `FOR LP` *ass_op* `END` *operation* `END` *ass_op* `END` *statement*
return → `RETURN` *expr*
ass_op → *id* `ASS_OP` *expr*
list_con → *expr* `RLIST_OP` *id*
list_oper → `LLIST_OP` *expr*

list_func → *id* `LIST_FUNC` *expr*
operation → *log_op*
input → `IN LP` *id* `RP`
output → `OUT LP` *val* `RP`
log_op → *log_op* `LLOG_OP` *ulog_op*
log_op → *ulog_op*
ulog_op → `RLOG_OP` *rel_op*
ulog_op → *rel_op*
rel_op → *rel_op* `REL_OP` *ari_op*
rel_op → *ari_op*
ari_op → *ari_op* `SS_OP` *md_op*
ari_op → *md_op*
md_op → *md_op* `MD_OP` *val*
md_op → `SS_OP` *val*
md_op → *val*
val → *id* `| INT | FLOAT | NIL | LITERAL`
val → *func_call*
val → `LP` *operation* `RP`
id → `ID`

Tabela 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	.	"@", "#", "."