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. A estrutura que representa o *token* apresenta o identificador do lexema 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, uma sequência de *structs* representando os símbolos, formando a tabela de símbolos, e uma pilha que que contém os escopos durante a execução da análise. 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 estrutura de cada nó apresenta o identificador e um tipo, que será utilizado pra verificação de tipos na análise semântica. 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 pelo *program* da gramática. A árvore então é formada seguindo a gramática da linguagem apresentada no apêndice A, onde os nós são criados pelos tipos da gramática e as operações representadas seguindo as precedências especificadas pela linguagem.

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 Análise Semântica

A análise semântica verifica se o programa segue as regras semânticas da linguagem e é realizada durante a criação da árvore sintática. Os erros são detectados durante a execução mas não interrompem a análise.

A estrutura da árvore utilizada pela análise sintática foi alterada para conter um valor de tipo, onde cada nó terminal possui o tipo de sua declaração, e nós não terminais verificam os tipos de cada subárvore, atualizando seu valor baseado nos tipos das subárvores ou conversões de tipo necessárias para operações. A tabela de símbolos também foi alterada para armazenar a quantidade de parâmetros de uma função, para que seja feita a verificação em chamadas de função.

O escopo é tratado pela pilha de escopo e armazenado na tabela de símbolos para cada identificador. A pilha utiliza uma variável auxiliar para saber qual o próximo valor de escopo que será inserido. Sempre que o analisador passa por uma *bracket* esquerda o valor da variável de escopo é incrementado e esse valor é inserido no topo da pilha. Quando passa por uma *bracket* direita o valor no topo da pilha é removido.

Operações entre *int* e *float* são permitidas pelo analisador, onde o resultado é tratado como *float*, seguindo a hierarquia $float- > int$. Atribuições também são permitidas, convertendo o valor para o tipo recebendo a atribuição. Os tipos de listas não realizam operações aritméticas com *int* e *float*, apenas operações de listas, como a inserção de novo elemento à lista. Os tipos de lista seguem a mesma hierarquia, onde elementos de tipos diferentes são convertidos para o tipo da lista quando inseridos.

O analisador semântico verifica as seguintes regras:

- Função main: Durante a análise sintática, sempre que uma função é declarada, seu identificador é comparado com "main", e caso positivo altera o valor de uma variável para 1. Caso a execução termine e o valor da variável ainda esteja em 0, um erro é emitido.
- Verificação do tipo de retorno: O analisador verifica se o tipo retornado pela função é o mesmo de sua declaração. A verificação é feita pela árvore, onde o bloco de função analisa a subárvore com a expressão contendo o tipo de retorno, e é feita a comparação com o tipo declarado da função.
- Verificação de declarações: Sempre que uma função ou variável é declarada, o analisador busca na tabela de símbolos se existe alguma declaração de função com o mesmo identificador, ou declaração de variável com o mesmo identificador num escopo presente na pilha de escopos.
- Parâmetros de função: Quando há uma chamada de função, a quantidade de argumentos na chamada é comparada com a quantidade de parâmetros da função sendo chamada. Para cada argumento, uma variável é incrementada e seu valor é posteriormente comparado com a quantidade de parâmetros da função na tabela de símbolos.
- Tipos dos parâmetros: Em chamadas de função com a quantidade correta de argumentos, os tipos de cada argumento são verificados com os tipos de cada parâmetro da função. A verificação é feita pela tabela de símbolos, onde

os parâmetros são armazenados com um identificador de parâmetro. Dessa forma, durante as chamadas de funções, a tabela de símbolos é verificada por parâmetros no escopo da função sendo chamada, e os tipos são comparados.
- Chamada de função: É verificado na tabela de símbolos se o nome da função sendo chamada já foi declarada anteriormente.
- Verificação de tipo: Todas as operações e atribuições passam por uma verificação de tipos, onde os tipos das subárvores são comparados e caso alguma conversão seja necessária, o novo tipo é armazenado no nó da operação.

5 TAC

TAC é um interpretador de Código de Três Endereços turing-completo. [San21] Após passar pela análise semântica, um programa pode ser considerado correto caso não tenha apresentado erros léxicos, sintáticos e semânticos, e com isso transformado para código intermediário. O código intermediário gerado pelo analisador deve ser capaz de ser interpretado pelo TAC. O código é gerado ao longo da análise e, caso não ocorram erros, gera um arquivo com extensão **.tac**.

Para isso, foram utilizadas duas variáveis, *tacTable* e *tacCode*, onde a primeira armazena o código para a seção **.table** e a segunda para a seção **.code**. Essas variáveis são vetores de caracteres onde os comandos de código intermediário são adicionados ao longo da criação da árvore sintática abstrata.

A seção da tabela de símbolos é gerada para declarações de variáveis e literais usados para comandos de saída ao longo do código. Sempre que uma variável é declarada, seu tipo e identificador são inseridos em *tacTable*. Os literais usados pelas funções de saída são armazenados como vetores de char e atribuídos um nome de string que é utilizada na seção de código.

A seção de código é composta por todas as instruções necessárias para a execução correta do programa. Para que o programa possa ser interpretado em código de três endereços, é necessário o uso de registradores para armazenar valores, *jumps* e *branches* para estruturas de controle de fluxo. Dessa forma, a árvore foi modificada para conter o registrador (caso necessário) com o valor armazenado da operação realizada naquele nó. O analisador também conta com contadores globais de registradores ($1, $2, $3, etc.) e controle de fluxo (L1, L2, L3, etc.). O código é gerado da seguinte forma:

- Operações aritméticas, relacionais e lógicas: As operações são realizadas entre dois valores e o resultado armazenado em um novo registrador. Operações lógicas usam **and** e **or**, operações relacionais usam **sleq**, **slt** e **seq** com o comando **not** no registrador em casos onde o resultado seria o inverso, operações aritméticas usam **add**, **sub**, **mul** e **div**.
- Atribuição: O valor é atribuído para uma variável utilizando a instrução **mov**.
- Entrada: Valores de entrada são lidos pelas instruções **scani** para inteiros e **scanf** para flutuantes.

- Saída: Para saídas de um único caractere ou valores numéricos, apenas um **print** é utilizado, mas para saída de literais é criado um **branch** que repete a operação para cada caractere da saída.
- Estrutura de repetição: Para cada *for* um novo valor de controle de fluxo é criado, o qual é acessado por uma instrução **brnz** enquanto a condição de parada não é alcançada.
- Estrutura condicional: Uma estrutura *if else* é composta por vários pulos e um **brnz**. Um valor de controle de fluxo é criado para o caso *if*, caso *else* e para o fim das condições. Inicialmente é feito um **jump** para o final onde a condição é testada em um **brnz** que direciona para o valor de controle de fluxo equivalente ao resultado.
- Retorno: O retorno de uma função é feito pela instrução **return**.
- Funções e parâmetros: Funções podem ser chamadas utilizando a instrução **call** seguida do nome da função e a quantidade de parâmetros. Os parâmetros são declarados anteriormente à chamada pela instrução **param**, onde os valores podem ser acessados pela função chamada.

6 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 correto com o objetivo de testar, principalmente, operação aritméticas e relacionais, assim como funções e entrada e saída.
- Arquivo de teste 2: O segundo arquivo, **test2.c**, é um programa correto com o objetivo de testar, principalmente, estruturas de fluxo de controle, como condições, repetições e chamadas de funções.
- Arquivo de teste 3: O terceiro arquivo, **test3.c**, tem como objetivo apresentar erros durante a análise. Erros semânticos ocorrem na linha 3 coluna 8, tipo de retorno diferente da declaração da função, linha 13 coluna 5, quantidade incorreta de argumentos na chamada de função, e erro de função main não encontrada, que não apresenta localização no código. Erros sintáticos ocorrem na linha 4 coluna 11, operação de leitura sem parâmetros, linha 6 coluna 13, retorno de função sem valor especificado.
- Arquivo de teste 4: O quarto arquivo, **test4.c**, também tem como objetivo apresentar erros. Erros semânticos ocorrem na linha 18 coluna 9, variável já declarada no mesmo escopo, linha 19 coluna 8, argumento com tipo incorreto na chamada de função, linha 20 coluna 10, erro de tipo em operação relacional entre inteiro e lista, linha 21 coluna 17, variável declarada em outro escopo. Erros sintáticos ocorrem na linha 6 coluna 12, operação de atribuição incorreta, linha 10 coluna 6, declaração de função incorreta.

7 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:

- $ bison -o src/sin_analyser.tab.c -d src/sin_analyser.y -v
- $ flex -o src/lex.yy.c src/ lex_analyser.l
- $ gcc -g src/sin_analyser.tab.c src/lex.yy.c src/data_structures.c src/intermediate_code.c -o tradutor -I lib -I src -Wextra

Executar:

- $ ./tradutor tests/nome_arquivo.c

Valgrind:

- $ make debug

TAC:

- $ ./tac nome_arquivo.tac

Referências

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

[San21] Luciano Santos. Simple three address code virtual machine. . https://github.com/lhsantos/tac, 2021 (acessado 25 de Outubro, 2021).

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 operações, operadores, declarações, atribuições, chamadas de funções, controle de fluxo e 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` *operation* | *operation*
func → `TYPE ID`
func → `TYPE LIST ID`
params → *params* `SEPARATOR` *param* | *param*
param → `TYPE ID`
param → `TYPE LIST ID`
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*
flow_ctr → *if_else* | *for* | *return* `END`
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` *operation*
input → `IN LP` *id* `RP`
output → `OUT LP` *operation* `RP`
ass_op → *id* `ASS_OP` *operation*
operation → *log_op*
log_op → *log_op* `LLOG_OP` *rel_op*

log_op → *rel_op*
rel_op → *rel_op* `REL_OP` *list_op*
rel_op → *list_op*
list_op → *list_op* `RLIST_OP` *ari_op*
list_op → *list_op* `LIST_FUNC` *ari_op*
list_op → *ari_op*
ari_op → *ari_op* `SS_OP` *md_op*
ari_op → `SS_OP` *md_op*
ari_op → *md_op*
md_op → *md_op* `MD_OP` *un_op*
md_op → *un_op*
un_op → `UN_OP` *val*
un_op → *val*
val → *id* `| inumber | fnumber | 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	`[\{\}]`	"{", "}"
parentheses	`[\(\)]`	"(", ")"
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	.	"@", "#", "."