UNIOESTE – Universidade Estadual do Oeste do Paraná
Ciência da Computação
Disciplina de Compiladores
Profª. Camile Bordini

Relatório - Trabalho 3

Bruno Brezolin

1. Introdução

Este relatório tem como intuito apresentar o processo de desenvolvimento de um compilador para uma linguagem baseada em C, chamada C.

A linguagem escolhida para o desenvolvimento foi a linguagem Lua.

2. Análise Léxica

Todo o processo de análise léxica é feito pelo código presente no arquivo $lexer.lua$. O compilador suporta um total de 31 classes de *tokens*, sendo elas:

keyword	identifier	int_literal	float_literal
string_literal	char_literal	comment	preprocessor
lparen	rparen	lbrace	rbrace
plus	minus	times	div
assign	not	and	or
bitwise_and	bitwise_or	less	greater
less_equal	greater_equal	equal	not_equal
comma	semicolon	eof	

a. Expressões regulares
- ID ::= [a-zA-Z_][a-zA-Z0-9_]*
- INTEGER ::= [0-9]+
- FLOAT ::= [0-9]*\.[0-9]+
- CHAR ::= '[a-zA-Z0-9]'
- STRING ::= "[a-zA-Z0-9]*"

b. Erros léxicos
- Unknown token (Token não identificado)

3. Análise Sintática

O processo de análise sintática foi implementado no arquivo $parser.lua$. O método utilizado foi de análise descendente recursiva, implementada através de um grafo de análise sintática feito com técnicas de programação funcional. Como exemplo, demonstra-se a função que faz a análise sintática de uma lista de argumentos de uma função:

```
function parse_argument_list(state)
  local arguments = {}

  while true do
    local current_token = get_current_token(state)

    if not current_token or current_token.value == ')' then
      break
    end

    local expression
    state, expression = parse_expression(state)
    table.insert(arguments, expression)

    local comma
    state, comma = expect(state, 'comma')
    if not comma then
      break
    end
  end

  return state, {
    type = 'argument_list',
    arguments = arguments
  }
end
```

Abaixo está descrita a gramática em notação BNF.

```
program ::= declaration_list
declaration_list ::= declaration | declaration_list declaration
declaration ::= variable_declaration | function_declaration
variable_declaration ::= type_specifier identifier ';'
function_declaration ::= type_specifier identifier '(' [parameter_list] ')' compound_statement
parameter_list ::= parameter | parameter_list ',' parameter
parameter ::= type_specifier identifier
compound_statement ::= '{' statement_list '}'
statement_list ::= statement | statement_list statement
statement ::= expression_statement | return_statement | variable_declaration | assignment_statement |
if_statement | while_statement | for_statement
if_statement ::= 'if' '(' expression ')' statement | 'if' '(' expression ')' statement 'else' statement
while_statement ::= 'while' '(' expression ')' statement
for_statement ::= 'for' '(' [expression] ';' [expression] ';' [expression] ')' statement
expression_statement ::= expression ';'
return_statement ::= 'return' expression ';'
assignment_statement ::= identifier '=' expression ';'
expression ::= logical_or_expression
logical_or_expression ::= logical_and_expression | logical_or_expression '||' logical_and_expression
logical_and_expression ::= equality_expression | logical_and_expression '&&' equality_expression
equality_expression ::= relational_expression | equality_expression '==' relational_expression |
equality_expression '!=' relational_expression
relational_expression ::= additive_expression | relational_expression '<' additive_expression |
relational_expression '>' additive_expression | relational_expression '<=' additive_expression |
relational_expression '>=' additive_expression
additive_expression ::= multiplicative_expression | additive_expression '+' multiplicative_expression |
additive_expression '-' multiplicative_expression
multiplicative_expression ::= unary_expression | multiplicative_expression '*' unary_expression |
multiplicative_expression '/' unary_expression
unary_expression ::= primary_expression | '-' unary_expression | '!' unary_expression | function_call
function_call ::= identifier '(' [argument_list] ')'
argument_list ::= expression | argument_list ',' expression
primary_expression ::= identifier | literal | '(' expression ')'
literal ::= INTEGER | FLOAT | CHAR | STRING
identifier ::= ID
type_specifier ::= 'int' | 'float' | 'char' | 'void'
```

4. Etapa Semântica

A análise semântica foi incorporada junto ao código de análise sintática, facilitando a extração de algumas informações. Por consequência, esta está presente no arquivo $parser.lua$.

Para auxiliar no gerenciamento de contextos, foram criados três novos tipos sendo estes $Context$, que representa um contexto (ou escopo), $Function$ que representa a entrada de uma função em um contexto e $Variable$ que representa uma entrada de variável em um contexto. A definição destes tipos ficou da seguinte maneira:

Context		
name	string	Nome do contexto
parent	Context	Contexto pai do contexto atual. *Nil* para o global
children	Context[]	Lista de contextos filhos do atual
content	(Variable | Function) []	Conteúdo do contexto
Function		
name	string	Nome da função
type	“function”	Auxilia na diferenciação com variável
return_type	var_type	Tipo do retorno da função
params	Variable[]	Parâmetros da função
Variable		
name	string	Nome da variável
type	“variable”	Auxilia na diferenciação com função
var_type	var_type	Tipo de dado da variável

O gerenciamento de contexto foi feito através de uma nova variável no estado do parser chamada de $context$. Essa variável aponta para o atual contexto da pilha de contextos, e através das funções $pushContext$ e $popContext$, é possível manipular os contextos do programa, criando novos e retornando para contextos superiores.

O contexto global, sendo o mais superior do programa, é diferenciado por não possuir um “pai”. Quando um símbolo é utilizado, este é procurado no contexto atual, caso não seja encontrado, sobe-se um contexto e reinicia a busca. Se a busca não encontrar o símbolo ao chegar no contexto global, considera-se um símbolo não declarado. Foi implementado o conceito de “shadowing”, onde uma variável de mesmo nome pode ser declarada em um contexto inferior, escondendo a variável que foi criada no contexto superior.

a. Gramática de atributos

As seguintes regras foram definidas para a gramática de atributos:

- Declaração de variáveis

```
variable_declaration ::= type_specifier identifier ';' ///
[variable_declaration.var_type = type_specifier, variable_declaration.name =
identifier]
```

Implementada no seguinte segmento de código:

```
if existsCurrent(new_state.context, identifier.value) ~= nil then --- verifica se
já existe
  print('Identificador `' .. identifier.value .. '` já existe no contexto atual')
  os.exit(1)
  return state, nil
end

-- se não existe, é inserido no contexto atual
new_state.context.content[identifier.value] = {
  name = identifier.value,
  type = "variable",
  var_type = type_specifier.value
}
```

- Declaração de funções

```
function_declaration ::= type_specifier identifier '(' [parameter_list] ')'
compound_statement /// [function_declaration.return_type = type_specifier,
function_declaration.name = identifier, function_declaration.parameters =
parameter_list]
```

Implementada no seguinte segmento de código:

```
if existsCurrent(new_state.context, identifier.value) then --- verifica se já
existe
  print('Identificador `' .. identifier.value .. '` já existe no contexto atual')
  os.exit(1)
  return state, nil
end

local params = extractArgs(parameter_list) --- coleta os parametros
new_state.context.content[identifier.value] = { --- insere no contexto atual
  name = identifier.value,
  type = "function",
  return_type = type_specifier.value,
  params = params
}
```

Com estes atributos semânticos, foi possível implementar alguns erros, como:

- Número errado de parâmetros em uma chamada de função
- Parâmetros com tipo errado em uma chamada de função
- Redeclaração de variáveis e funções
- Utilização de variáveis e funções não declaradas
- Tentativa de chamar uma variável como função

5. **Processamento de código fonte**

   O seguinte código foi utilizado para validar o processamento sem erros.

```
#include <stdio.h>

int soma(int x, int y) {
  int resultado;
  resultado = x + y;
  return resultado;
}

int main(int argc) {
  int a;
  int b;
  for(a = 0; a < 10 ; ++a) {
    b = a + 1;
    if(a && 1 == 0) {
      printf("Soma de %d com %d eh %d (par)\n", a, b, soma(a, b));
    } else {
      printf("Soma de %d com %d eh %d (impar)\n", a, b, soma(a, b));
    }
  }
}
```

Tabela de Símbolos

Contexto	Identificador	Tipo	Retorno	Parâmetros
global	soma	function	int	int x, int y
global	printf	function	int	------
global	main	function	int	int argc
soma	resultado	variable	int	
main	b	variable	int	
main	a	variable	int	

Árvore sintática

Devido à extensividade da árvore sintática, esta foi movida para o arquivo $arvore.json$, presente junto com o código fonte.

6. Referências

ALFRED, V. A.; MONICA, S. L.; JEFFREY, D. U. Compilers Principles, Techniques & Tools. [S.l.]: pearson Education, 2007