Relatório 2 Compiladores

Eliton Machado e Igor Froehner

Agosto 2021

1 Linguagem

Nomeada *Verb*, a linguagem tem como objetivo ser menos verbosa que as linguagens populares, para isso a linguagem desenvolvida nessa etapa do trabalho tem como característica que todas as palavras reservadas são um caractere maiúsculo, ela possibilitará tanto o uso de código que não está em funções quanto, definição e chamada de funções. A definição dos blocos é dada como em C, com abertura e fechamento de colchetes ({}), e, assim como em C, a abertura de bloco só é necessária em if's, *while*'s, for's, *switch*'s quando há mais de uma instrução no corpo do comando. Ela é fortemente tipada, portanto, na criação de uma variável ou função é necessário informar qual o tipo do dado. As condicionais dos blocos devem ser delimitadas por parenteses.

Como a proposta é de cada palavra ter um caractere, é necessário ter um lista de comandos para um programador que está tendo seu primeiro contato saber qual o significado de cada comando:

1.1 Tipos

- I: *int*
- D: *double*
- S: *string*

1.2 Comandos de Fluxo

- $? = if$

- \$ = *else if*
- : = *else*
- # = *switch*
- O = *do*
- W = *while*
- F = *for*

Com isso já é possível ter uma visão geral sobre a linguagem, para finalizar essa introdução a seguir está um código de exemplo que calcula uma média ponderada de três valores:

```
D a = input();
D b = input();
D c = input();

D media = (2*a + 3*b + 5*c) / 10;

print("MEDIA = %f\n", media);
```

Há comentários de linha que são determinados com //, e comentários de bloco que são começados com /* e terminados com */.

2 Gramática

Sabendo a cara da linguagem tratada pelo analisador sintático pode-se desenvolver a gramática que fará a análise sintática:

```
0 $accept: program $end

1 program: block

2 block: %empty
3      | statement block
4      | flux block
```

```
5       | function block

6  statement: declaration ';'
7           | assignment ';'
8           | expr ';'
9           | error ';'

10 optional_block: statement
11                | '{' block '}'
12                | '{' error '}'

13 type: 'I'
14     | 'D'
15     | 'S'

16 value: INTEGER
17      | FLOAT
18      | STRING

19 expr: value
20     | call
21     | expr '<' expr
22     | expr '>' expr
23     | expr BOOLOP expr
24     | expr '|' expr
25     | expr '^' expr
26     | expr '&' expr
27     | expr CMPOP expr
28     | expr BITSHIFTOP expr
29     | expr '+' expr
30     | expr '-' expr
31     | expr '*' expr
32     | expr '/' expr
33     | expr '%' expr
34     | '-' expr
35     | '!' expr
36     | '~' expr
37     | expr EXPOP expr
```

```
38     | '(' expr ')'
39     | '(' error ')'

40 declaration: type ID
41            | type ID '=' expr

42 assignment: ID '=' expr
43           | ID ATTOP expr

44 call: ID
45     | ID UNARYOP
46     | UNARYOP ID
47     | ID '(' ')'
48     | ID '(' expr_list ')'
49     | ID '(' error ')'

50 expr_list: expr
51          | expr ',' expr_list

52 declaration_list: declaration
53                 | declaration ',' declaration_list

54 assignment_list: assignment
55                | assignment ',' assignment_list

56 flux: if
57     | switch
58     | while
59     | do
60     | for

61 if: '?' '(' expr ')' optional_block elseif else
62   | '?' '(' error ')' optional_block elseif else

63 elseif: %empty
64       | '$' '(' expr ')' optional_block elseif
65       | '$' '(' error ')' optional_block elseif
```

```
66 else: %empty
67     | ':' optional_block

68 switch: '#' '{' switch_body '}'
69       | '#' '{' error '}'

70 switch_body: %empty
71            | value ':' statement switch_body

72 while: 'W' '(' expr ')' optional_block else
73      | 'W' '(' error ')' optional_block else

74 do: 'O' '{' block '}' while

75 for: 'F' '(' expr ')' optional_block else
76    | 'F' '(' INTEGER ';' expr ';' INTEGER ')' optional_block else
77    | 'F' '(' declaration_list ';' expr ';' assignment_list ')'
        optional_block else
78    | 'F' '(' error ')' optional_block else

79 function: type ID '(' declaration_list ')' optional_block
80         | type ID '(' error ')' optional_block
```

A gramatica foi desenvolvida sem nenhuma ambiguidade de shift/reduce, optou-se por impedir conflitos como o *Dangling else problem* obrigando o usuário a escrever o código sem ambiguidade.

3 Implementação

A implementação foi feita usando o framework *flex/bison*, o *flex* trata da léxica identificando os tokens e os informando ao *bison*, que por sua vez trata da parte sintática.

Não foi implementada uma estrutura de dados complexa para a tabela de símbolos pois os autores não viram necessidade da tabela para a analise léxica que é o objetivo deste trabalho.

Com o objetivo de tratar erros léxicos e semânticos, utilizou-se as diretivas:

```
%locations
%define api.pure full
%define parse.error custom
%define parse.lac full
%param { user_context* uctx}
```

Estas possibilitaram o rastreio das posições dos erros, a construção de mensagens de erro completamente customizadas, o controle dos dados passados entre o analisador léxico e sintático e a utilização da estrutura *user_context* definida pelos autores para o armazenamento da linha do erro.

```
1 I a = ;
2 I A = 0;
3 F() {
4     a 9;
5     a = 0 + ++2 + 5;
6 }
7 I a 1234567890;
```

Acima à um arquivo com diversos erros e abaixo a saída do compilador ao ler o arquivo. Todos as mensagens de erro são customizadas e escritas pelos autores.

```
Compiling ../tests/error.ve
1.7: syntax error before ';'
    1 | I a = ;
      |       ^
2.3: Lexical Error, Unexpected 'A'
4.3: syntax error before ')'
    4 | F()
      |   ^
5.7: syntax error before INTEGER
    5 |     a 9
      |       ^
6.15: syntax error: expected ID before INTEGER
```

```
    6 |     a = 0 + ++2
      |               ^
9.5-15: syntax error: expected '=' or ';' or '(' before INTEGER
    9 | I a 1234567890
      |     ^~~~~~~~~~
```

Pode-se perceber a presença de *error recovery* pois o programa continua a analise mesmo depois do encontro de erros.

4 Árvore de Derivação

Foi criada uma estrutura para implementar a árvore de decisão, onde cada nó é tem um nome, que indica qual o tipo da regra a ser derivada ou, se for um terminal, qual é este terminal; e dois ponteiros para outros nós, um nó aponta para a direita, ou seja, para uma regra ou terminal que está no mesmo nível da derivação, e se for uma regra, um ponteiro para o próximo nível da derivação.

Com tal estrutura foi possível manter a guardar a árvore de derivação de forma simples, para dessa forma, percorre-lá imprimindo-a do jeito que se desejar. E nesse trabalho a árvore foi impressa para um código *.dot* para posteriormente ser gerada uma imagem da árvore de derivação. Aplicando tal abordagem ao código de exemplo a baixo:

```
I a = input();
I cont = 0;

F (a) {
    ?(a > 0) a--;
    : a++;
    cont++;
}

print(a);
```

A árvore de derivação fica como segue, cada nó em verde é um terminal da gramática.

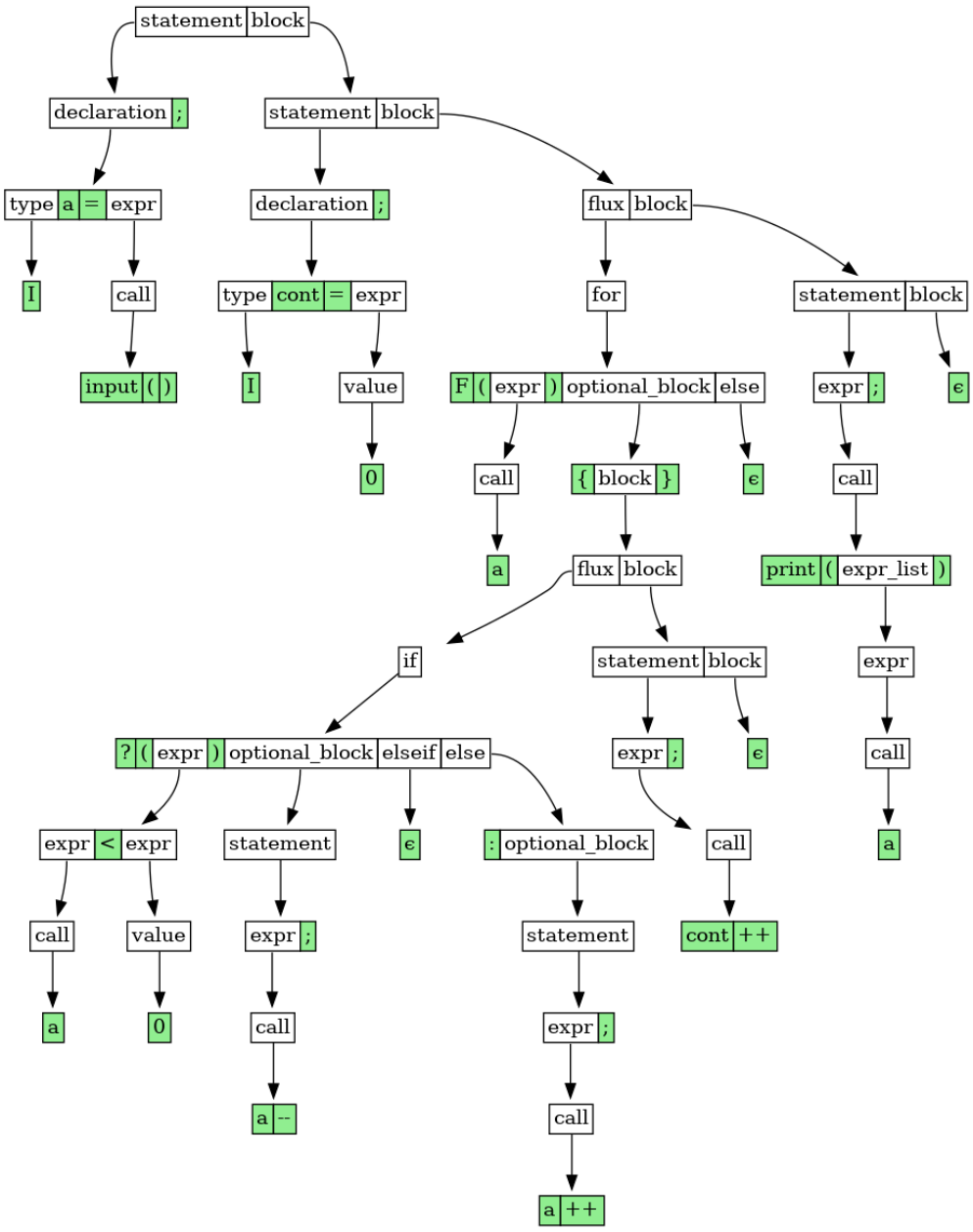

Figure 1: Árvore de Derivação do Código cont.ve

Nos arquivos do projeto seguem diversos códigos exemplo na pasta *tests*

5 Execução

O projeto foi desenvolvido e testado em um máquina Ubuntu 20.04, gcc 9.3.0, make 4.2.1, flex 2.6.4 e bison 3.7 baixado manualmente do site bison

O compilador está localizado dentro do arquivo zip na pasta *verb*. Para a criação do arquivo executável utiliza-se o comando *make*.

5.1 Parâmetros de execução

```
./verb myfile.ve        // compilar o meu arquivo
./verb -g myfile.ve     // compilar e gerar a árvore de derivação
./verb -p myfile.ve     // compilar e mostrar o passo a passo da derivação
```