RELATÓRIO COMPILADOR 2021.1

Alunos: Vinicius Martins Bezerra
Vinicius Santos de Almeida

1. Gramática

Foi realizado a construção de uma gramática LL(1), ou seja, uma gramática fatorada à esquerda, sem recursão à esquerda, na forma de Backus-Naur (BNF)

Dados

<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<type> ::= "int" | "bool"

<number> ::= <digit> {<digit>}*

<bool_value> ::= "True" | "False"

<id> ::= <letter> (<letter> | <number>)*

<op_bool_value> ::= == | != | > | < | >= | <=

<op_aritmetic> ::= + | - | * | /

Variaveis

<var_declaration> ::= <type> <id> ;

<var_atrib> ::= <id>_<call_func> | <bool_value> | <number> | <id> | <call_op>

Função & Procedimento

<func_declaration> ::= func <type> <id> (<params>) { <block_func> <return_declaration>};

<call_func> ::= call <id> (<params_call>);

<proc_declaration> ::= proc <id> (<params>) { <block_func> };

<call_proc> ::= call proc <id> (<params_call>);

<params> ::= <type> <id> (, <params> | Ø)*

<params_call> ::= <id> (, <params_call> | Ø)*

<return_declaration> ::= return <return_declaration_aux> ;

<return_declaration_aux> ::= <id> | <number> | <bool_value>

Operações

<call_op> ::= <number> <op_aritmetic> <call_op> | <id> <op_aritmetic> <call_op>

Condicional

<expression> ::= <id> <op_bool_value> <expression_aux> | <number> <op_bool_value> <expression_aux>

<expression_aux> ::= <expression> | <id> | <number>

<if_declaration> ::= if(<expression>){<block_if>} endif <else_declaration>

<else_declaration> ::= else { <block_if> } endelse | Ø

<while_declaration> ::= while(<expression>){<block_whihle>}endwhile

<if_declaration_flow> ::= if(<expression>){<block_whihle>} endif <else_declaration_flow>

<else_declaration_flow> ::= else { <block_whihle> } endelse | Ø

<statement_flow> ::= continue ; | break ;

Print

<print_declaration> ::= print (<params_print>) ;

<params_print> ::= <id> | <call_func> | <call_op> | <bool_value> | <number>

Blocos

<block_main> ::= <var_declaration> | <var_atrib> | <func_declaration> | <proc_declaration> | <call_func> | <call_proc> | <print_declaration> | <if_declaration> | <while_declaration> | <id> | Ø

<block_func> ::= <var_declaration> | <var_atrib> | <print_declaration> | <if_declaration> | <while_declaration> | <id> | Ø

<block_whihle> ::= <var_declaration> | <var_atrib> | <call_proc> | <call_func> | <id> | <if_declaration_flow> | <while_declaration> | <statement_flow> | <print_declaration>

<block_if> ::= <var_declaration> | <var_atrib> | <call_proc> | <call_func> | <id> | <if_declaration> | <while_declaration> | <print_declaration>

Obs: *statement_flow somente pode está dentro de um WHILE, e no IF/ELSE dentro do WHILE
*block_whihle é o bloco que contém break/continue que só pode ser chamado dentro de um while e não pode declarar função e procedimento dentro
*block_if é o bloco do if/else, que não pode declarar função e procedimento dentro
*e chamado somente dentro do while, pois dentro dele pode ter BREAK E CONTINUE
*if_declaration_flow e lse_declaration_flow são chamado somente dentro do while, pois dentro dele pode ter BREAK E CONTINUE (block_whihle)

2. Analisador Léxico

O analisador léxico resulta em uma cadeia de tokens contendo o código do símbolo identificado, o conteúdo do símbolo e a linha da ocorrência do símbolo. Os tokens possuem o seguinte formato: **token+código+posição na tabela+_+conteúdo.** A seguir é dada a lista dos tokens possíveis nessa linguagem:

Lista de tokens

OPERADORES - código 1		Delimitadores - código 2
token100 _.	token106_!=	token200_;
token101_+	token107_>	token201_,
token102_-	token108_>=	token202_(
token103_*	token109_<	token203_)
token104_/	token110_=	token204_{
token105_==		token205_}
Números código 3	Constantes código 4	Identificadores código 5
token300_num	Token400_cc	token500_id
Palavras reservadas - código 6		
token600_main	token605_endif	token613_break
token601_end	token606_else	token614_cont
token602_func	token606_endelse	token609_int
token603_retur	token607_while	token610_bool
token604_call	token607_endwhile	token611_boolValue
token605_if	token608_print	token615_proc

*O código 4 foi removido posteriormente durante a implementação

Abaixo pode ser visto a representação token no programa onde é armazenado o seu tipo, lexema e a linha onde foi encontrado.

```
class Token:
    def __init__(self, tipo, lexema, linha):
        self.tipo = tipo
        self.lexema = lexema
        self.linha = linha
```

3. Analisador Sintático

O analisador sintático construído foi do tipo Descendente Preditivo Recursivo. Para cada símbolo não-terminal da gramática, uma nova função foi construída. As produções da gramática foram representadas por chamadas sucessivas dessas funções.

Abaixo segue algumas das características da linguagem criada que devem ser respeitada durante a analise sintática

- É uma linguagem inspirada em C;
- Linguagem estruturada;
- Toda variável deve ser declarada,contendo o tipo e o nome da variável seguido por um ponto e virgula, mas não deve ser inicializada. A sua inicialização deve ser feita na linha abaixo da qual foi declarada (opcionalmente), respeitando seu tipo;
- As variáveis devem ser do tipo inteiro ou booleano;
- As variáveis podem ser locais, devendo ser declaradas no início do bloco da função, sendo seu escopo, o corpo da função onde as variáveis forem declaradas;
- Variáveis podem ser usados em atribuições, expressões, retornos de funções e parâmetros em chamadas de funções;
- O corpo principal do programa é um bloco iniciado pela palavra reservada “main”. Quando um programa nessa linguagem é executado, é este bloco que é executado;
- Um programa nesta linguagem pode possuir várias funções;
- As funções podem ter parâmetros dos tipos primitivos e podem retornar valores ou não (procedimentos);

- Toda função deve iniciar com a palavra reservada func, seguida do tipo de retorno seguida do identificador de nome da função, seguido pela lista de parâmetros formais e o corpo da função delimitado por { e }endfunc;

- Da mesma forma todo procedimento deve iniciar com a palavra reservada proc seguido do identificador de nome do procedimento, seguido pela lista de parâmetros formais e o corpo do procedimento delimitado por { e }endproc ;

- Delimitadores ‘’{‘’ e ‘’}endmetodo’’ marcam início e fim de blocos (comandos, declarações, funções), respectivamente;

- São permitidas expressões relacionais e aritméticas.

- Comandos:

  - Comando If/Else:
    - Na condição do comando If só serão permitidas expressões relacionais.
    - O fim do bloco é representado pela sequencia endif.
    - A parte Else será opcional e o seu fim é representado por endelse.

  - Comando while:
    - Na condição do comando while só serão permitidas expressões relacionais.
    - Podem utilizar as palavras reservadas ‘’break’’ e ‘’continue’’ para alterar o fluxo do programa.
    - O fim do bloco while é representado por endwhile.

  - Comando print:
    - O comando iniciará com a palavra print e o que deverá ser escrito entre parênteses, finalizando com ponto e vírgula.
    - O comando pode imprimir: Números, variáveis, e expressões.

Quando um erro sintático é encontrado, uma mensagem de erro é indicada no console informando a linha onde o erro foi encontrado e o token que causou o erro.

Após indicar algum erro o analisador sintático pode encontrar erros em cascata devido ao erro inicial. Por isso caso esteja presente mais de um erro no programa, é recomendado ao usuário tentar consertar os erros iniciais antes dos posteriores.

Uma mensagem de sucesso é mostrada no terminal indicando o reconhecimento sintático completo da cadeia de entrada fornecida pelo arquivo programa.txt.

4. Analisador Semântica & Código de três endereços

Foi construido um analisador semântico para a linguagem de programação até então elaborada, de acordo com as especificações abaixo:

- Atribuição deve ser do mesmo tipo que foi declarado
- Operações aritméticas e relacionais devem ser feitas entre operadores de mesmos tipos
- As funções não vêem o escopo global, elas só vêem o que é passado para elas por parâmetro
- Chamadas de funções devem ser feitas com o número e ordem de parâmetros corretos
- Retorno de funções deve ser do mesmo tipo declarado
- Retorno de método só suporta uma variável ou valor primitivo. Chamada de função, ou operação aritmética devem ser feitas antes e atribuídas a uma variável que então será atribuída ao retorno.
- Operações + - / * são compatíveis apenas com operandos inteiro
- Operações relacionais com operadores == != > < >= <= podem ser feitas com inteiro desde que ambos operandos sejam do mesmo tipo

Após indicar algum erro o analisador semântico pode encontrar erros em cascata devido ao erro inicial. Por isso caso esteja presente mais de um erro no programa, é recomendado ao usuário tentar consertar os erros iniciais antes dos posteriores.

Uma mensagem de sucesso é mostrada no terminal indicando o reconhecimento semântico completo da cadeia de entrada fornecida pelo arquivo programa.txt.

Quanto ao código de 3 endereços o mesmo foi projetado para ser executado paralelamente a analise sintática. Ao final da analise semântica é imprimido no console o código de três endereços

Na imagem abaixo pode ser observado um exemplo de trecho de código que é aceitado no nosso compilador, respeitando cada atribuição construida na gramática.

```
main {

    int i;
    bool b;
    i = 1;
    b = True;

    func int f1(int t1, int t2){
        return t1;
    }endfunc

    func bool f2(bool t1, bool t2){
        return True;
    }endfunc

    proc p1(int t, int t2){
        print(t1);
    }endproc

    int cont;
    cont = 0;
    while(cont < 10){
        if(cont == 5){
            break;
        }endif
        else{
            continue;
        }endelse
    }endwhile

    int teste1;
    teste1 = call f1(2, 1);
    call p1(2, 4);
 } endmain

```