

LINGUAGEM ORION

Contents

1	Introdução	1
2	Semântica Informal da Orion	2
2.1	Identificadores, Constantes e Comentários	2
2.1.1	Identificadores	2
2.1.2	Constantes	2
2.1.3	Comentários	2
2.1.4	Declarações e Definições	2
2.1.5	Arranjos	3
2.1.6	Escopos de Nomes	3
2.1.7	Alocação de Memória	3
2.1.8	Expressões	3
2.1.9	Funções e Arranjos	4
2.1.10	Condições	4
2.1.11	Comandos	4
2.1.12	Atribuição	4
2.1.13	Leitura	5
2.1.14	Escrita	5
2.1.15	Comando if	6
2.1.16	Comando while	6
2.1.17	Comando repeat	6
2.1.18	Comando return	6

2.1.19 Comando exit	6
2.1.20 Chamada de Procedimento	6
2.2 Disciplina de tipo	7
2.3 Sugestões	7
3 Sintaxe da Linguagem Orion	8
3.1 Gramática do Mini-Pascal	8
4 Compilador para Orion	11
4.1 Metodologia para o Desenvolvimento do Compilador	11
4.2 Ferramentas de Apoio	11
4.2.1 Geradores de Analisadores Léxicos	11
4.2.2 Geradores de Analisadores Sintáticos	11
4.3 Gramática de Orion modificada para a Implementação	12
4.4 Interpretador TAM	15
4.4.1 Como utilizar o Interpretador TAM	15
4.4.2 Exemplo	16
4.5 Resultados Desejados	18

6.4 Interpretador	23
-----------------------------	----

7 Bibliografia	24
-----------------------	-----------

Linguagem Orion

1 Introdução

A linguagem de alto nível para a qual você deve projetar e implementar um compilador é chamada Orion. Ela é um mini-pascal. O propósito destas notas é apresentar uma definição informal da Orion.

Orion é uma linguagem de programação com estruturas de blocos que contém construções que vieram do ALGOL 60 e Pascal. Tal como no Algol 60, os procedimentos, subrotinas e funções, de Orion podem ser recursivos; e como em Pascal, novos tipos de dados podem ser criados. Orion possui 3 tipos básicos: **integer**, **boolean** e **char**, os quais provêm as bases para criação de estruturas de tipos mais complexas, tais como o **array**. As regras que governam o escopo e visibilidade de identificadores em programas Orion são exatamente as do ALGOL 60.

Orion é uma linguagem estruturada que possui vários tipos de comandos: **comando de atribuição**, **entrada/saída**, **condicional**, **repetitivo**, **blocos**, **chamada de procedimentos**, **retorna** e **abandonar**. Expressões em Orion pode ter os operadores usuais de somar, multiplicar, dividir, subtrair e elevar a uma potência. Além disto, operadores lógicos e relacionais podem ser usados.

A sua tarefa é construir um **compilador** para Orion. Para executar os programas em Orion será usado a máquina abstrata TAM (*Triangle Abstract Machine*), cuja especificação completa pode ser encontrada em [5]. É imprescindível a leitura deste documento. TAM é implementada por um interpretador. Portanto, o seu compilador deve gerar código para a linguagem intermediária de quádruplas da máquina abstrata TAM para ser lida pelo interpretador. TAM foi projetada especificamente para dar suporte às implementações de linguagens de alto-nível, e, em particular, às técnicas de administração de memória em tempo de execução.

O código do interpretador e o relatório técnico com a descrição de TAM estão disponíveis na página do curso sob o nome **Interpretador** e **TAM** [5], respectivamente. Não é preciso implementar o interpretador. Sua tarefa será executá-lo com os programas Orion dos testes da Seção ???. A Seção 4.4 mostra como usar o interpretador, apresenta um exemplo que ilustra a compilação de um programa fonte em Orion para as instruções de TAM, e, em seguida, mostra o resultado de sua execução. Caso você ache que as informações fornecidas em [5] não são suficientes, consulte [3]. Para facilitar o trabalho, na falta do livro [3], uma cópia do mesmo estará disponível para os interessados com o professor. Não é preciso comprá-lo.

2 Semântica Informal da Orion

Uma gramática livre do contexto para Orion é dada na Seção 3.1.

Nesta seção, uma definição informal da semântica de Orion é dada e tem a intenção de ser apenas uma definição parcial das condições que um compilador Orion deve satisfazer.

2.1 Identificadores, Constantes e Comentários

2.1.1 Identificadores

Em Orion, um identificador é uma letra seguida ou não de letras e/ou dígitos decimais. O comprimento máximo de um identificador é dependente da implementação.

As palavras chaves de Orion são escritas em letras minúsculas. As palavras chaves de Orion serão consideradas reservadas.

```
begin  boolean          char    do     else   end      false
endif   endwhile         exit    if     integer  procedure  program  reference
        repeat           read   return  then   true    type      until   value
write  while
```

2.1.2 Constantes

As constantes podem ser **inteiros, lógicas ou um caractere**. As constantes inteiras possuem 16 bits, podendo variar de -32768 a 32767.

2.1.3 Comentários

Comentários em Orion são textos delimitados por /* e */.

2.1.4 Declarações e Definições

Toda variável em Orion tem que ser declarada antes de ser usada. O mesmo vale para definições de tipos e declarações de procedimentos. Note que esta restrição elimina a possibilidade de se ter definições de tipos mutualmente recursivas. Entretanto, procedimentos podem ser mutualmente recursivos desde que declarações extras (e sem o corpo) de alguns dos procedimentos sejam acrescentadas antes das declarações

dos procedimentos. Uma declaração de procedimento sem o respectivo corpo tem o mesmo efeito da construção **FORWARD** do Pascal. Ela apenas anuncia que o procedimento será declarado mais adiante no texto.

As declarações devem ser feitas na seguinte ordem:

1. Definições de tipos
2. Declarações de variáveis
3. Declarações de procedimentos

2.1.5 Arranjos

O valor do limite inferior de qualquer dimensão de um arranjo não pode ser menor que zero, como indica a gramática de Orion. Além disso, o valor limite superior deve ser maior ou igual ao valor do limite inferior.

Programas que violam limites de arranjos devem ser considerados ilegais, isto é, a menos que se consiga determinar durante a compilação que os limites não serão violados, o compilador Orion deve gerar um código tal que erros deste tipo sejam detectados durante a execução caso eles existam.

2.1.6 Escopos de Nomes

O escopo de nomes em Orion é regido pelas mesmas regras de escopo do Pascal, isto é, dentro de um bloco, os nomes acessíveis são aqueles declarados ou definidos no início do bloco mais aqueles declarados ou definidos em blocos envolventes, desde que estes nomes não sejam redeclarados no bloco em questão.

2.1.7 Alocação de Memória

Orion adota um mecanismo de alocação dinâmica de memória da seguinte forma: memória para variáveis de um procedimento é alocada quando o procedimento é ativado e liberada no momento do retorno.

2.1.8 Expressões

Expressões Orion podem ter tipo **integer**, **boolean**, **char** ou qualquer outro tipo definido pelo usuário. Entretanto, as operações aritméticas, lógicas e de relações só podem ser efetuadas se os operandos têm tipos não-estruturados, isto é, **integer**, **boolean** ou **char**. Nas operações de relação, os operandos não podem ser do tipo **boolean**.

Os operadores em ordem crescente de precedência são:

1. booleano |
2. booleano &
3. booleano **not**
4. relacionais: <, >, =, \leq , \geq , not=
5. aritméticos: +, -
6. aritméticos: *, /
7. exponenciação **
8. aritmético (unário) -

Os operadores associativos associam-se à esquerda, exceto o "<<" , que se associa à direita.

2.1.9 Funções e Arranjos

Em Orion, parênteses são usados para delimitar tanto os índices de um arranjo como os parâmetros de uma função. **Portanto**, a gramática de Orion é ambígua porque a sintaxe de chamada de funções e referência a elementos de arranjos é a mesma. Note-se, entretanto, que esta ambiguidade pode ser facilmente resolvida via consultas às declarações em vigor em um dado programa.

Em uma chamada de função, os tipos dos argumentos têm que ser os mesmos dos parâmetros formais correspondentes.

2.1.10 Condições

O fluxo de controle de comandos condicionais e repetitivos é governado por expressões booleanas chamadas condições. Estas expressões podem conter operadores de relação, lógico ou aritméticos. Entretanto, o tipo do resultado tem que ser sempre **booleano**.

2.1.11 Comandos

Orion é uma linguagem estruturada no sentido de que ela somente tem construções de controle estruturados (**if-then-else**, **while**, **repeat**, etc) e não tem construções perigosas como o **goto**. Os comandos de Orion são sumariamente descritos a seguir.

2.1.12 Atribuição

O identificador do lado esquerdo do ":" não pode ser nome de procedimento.

O tipo da variável do lado esquerdo tem que ser equivalente ao da expressão do lado direito. O método de equivalência adotado é da "eqüivalência por nome" , isto é, dois tipos são considerados equivalentes se eles têm exatamente o mesmo nome.

2.1.13 Leitura

Muito simples. Apenas a palavra **read** seguida de uma variável. O próximo valor do meio de entrada é lido, convertido para o tipo da variável dada e armazenado nesta variável. O tipo da variável deve ser **integer**, **boolean** ou **char**.

Dada a sintaxe do comando **read**: comando_read ::= READ variável

na geração do código podem ser realizadas as seguintes ações:

- se a variável for do tipo **boolean** ou **integer** gera-se a instrução **getint** que obtém o próximo inteiro da entrada e o armazena no endereço que está no topo da pilha de dados. Considera-se que 1 (um) é **true** e 0 (zero) **false** para uma variável booleana.
- Se a variável for do tipo **char** gera-se a instrução **get** que obtém o próximo caractere da entrada.

2.1.14 Escrita

O comando para imprimir consiste da palavra **write** seguido de uma expressão que deve ser avaliada e convertida para o formato aceitável pelo meio de saída. O tipo da expressão pode ser **integer**, **char** ou **boolean**.

Dada a sintaxe do comando **write**: comando_write ::= WRITE expr

na geração de código podem ser realizadas as seguintes ações:

- se a expressão for uma variável, carrega-se o conteúdo da variável usando o endereço que está no topo da pilha que corresponde ao endereço da variável.
- Se a expressão for do tipo **boolean** ou **integer** gera-se a instrução **putint** que escreve o valor no topo da pilha para a saída. Se for **boolean** será impresso 1 (um) caso a expressão seja verdadeira e 0 (zero) caso seja falsa.
- Se a expressão for do tipo **char** gera-se a instrução **put** que imprime o caractere que está no topo da pilha.

2.1.15 Comando if

Orion tem dois comandos **if**. O **if-then** e o **if-then-else**. Ambos são terminados pelas palavras chaves **endif**. O comando **if** tem a semântica usual. A geração de código para o comando **if** pode ser feita de acordo com [1] página 505, ou [2].

2.1.16 Comando while

A lista de comandos no corpo do comando **while** é executada enquanto a expressão booleana que está entre as palavras **while** e **do for** verdadeira.

2.1.17 Comando repeat

A lista de comandos no corpo do comando **repeat** é executada até que a expressão booleana que está depois da palavra **until** se torne verdadeira. Note que a condição é avaliada após cada execução da lista de comandos.

2.1.18 Comando return

Só pode aparecer dentro de um procedimento do tipo função. O tipo da expressão após o **return** tem que ser equivalente ao tipo do valor a ser retornado pela função, isto é, o tipo especificado na declaração do procedimento função.

2.1.19 Comando exit

Este comando faz com que o controle deixe a execução do comando cujo rótulo é o especificado após a palavra **exit**. É claro que o comando **exit** tem que estar no corpo do comando que tem o rótulo especificado. Após a execução de **exit**, o controle passa para o comando que textualmente segue o comando que tem o rótulo especificado.

2.1.20 Chamada de Procedimento

Os tipos dos argumentos têm que ser exatamente os mesmos dos parâmetros formais correspondentes. Isto é, os nomes dos tipos de cada par correspondente têm que ser o mesmo.

Quanto ao modo de passagem de parâmetros têm-se as seguintes possibilidades: **value** e **reference**.

A geração de código para a chamada de procedimento pode ser feita da seguinte forma:

- Obtêm-se os atributos da função sendo chamada da tabela de símbolos.
- Verifica se o número e o tipo dos argumentos passados estão de acordo com os parâmetros formais correspondentes.
- Para cada parâmetro formal: se o modo de passagem for por **reference** empilha-se o endereço da variável de argumento, **não sendo permitido passar uma expressão**. Se o modo for **value** empilha-se, na pilha de dados, o valor da expressão de argumento.
- Gera-se a instrução para chamar a função.

2.2 Disciplina de tipo

Orion é uma linguagem com "strong typing" , isto é, todo valor em Orion somente tem um tipo de tal modo que, o tipo de qualquer expressão em qualquer contexto pode ser determinado a partir dos tipos dos elementos e operadores da expressão.

Além disto, dois tipos são considerados equivalentes *se e somente se* eles têm o mesmo nome. Note que os tipos abaixo não são vistos como equivalentes.

```
type X = (1:10) integer;
```

```
type Y = (1:10) integer
```

2.3 Sugestões

Para que se possa verificar se o número e tipos dos parâmetros que são usados em uma chamada de procedimento estão compatíveis com os que aparecem na sua declaração, as informações sobre os parâmetros formais de um procedimento devem ser salvas (na própria tabela de símbolos) no momento em que se termina a compilação do procedimento. Uma sugestão é nunca liberar o espaço na tabela de símbolos que é ocupado pelo procedimento.

3 Sintaxe da Linguagem Orion

3.1 Gramática do Mini-Pascal

```
program ::= program declaracoes block
block ::= begin lista_comandos end
declaracoes ::= declaracoes declaracao ;
               |
               | vazio
declaracao ::= decl_de_var
              |
              | def_de_tipo
              |
              | decl_de_proc
decl_de_var ::= tipo : lista_de_ids
tipo ::= integer
          |
          | boolean
          |
          | char
          |
          | tipo_definido
def_de_tipo ::= type nome_do_tipo = definicao_de_tipo
nome_do_tipo ::= identificador
definicao_de_tipo ::= ( limites ) tipo
limites ::= inteiro : inteiro
tipo_definido ::= identificador
decl_de_proc ::= proc_cab : proc_corpo
proc_cab ::= tipo_retornado procedure nome_do_proc
            espec_de_parametros
proc_corpo ::= declaracoes bloco
               |
               | vazio
lista_de_parametros ::= parametro
                      |
                      | lista_de_parametros , parametro
tipo_retornado ::= integer
                  |
                  | boolean
                  |
                  | char
                  |
                  | vazio
parametro ::= modo tipo : identificador
modo ::= value
        |
        | reference
nome_do_proc ::= identificador
```

```

lista_de_comandos ::= comando
                   | lista_de_comandos ; comando
lista_de_ids ::= identificador
                | lista_de_ids , identificador
vazio ::= ε
espec_de_parametros ::= ( lista_de_parametros )
                      | vazio
comando ::= comando_de_atribuicao
           | comando_while
           | comando_repeat
           | comando_if
           | comando_read
           | comando_write
           | comando_return
           | comando_exit
           | chamada_de_procedimento
           | rotulo : comando

comando_de_atribuicao ::= variavel := expr
comando_while ::= while expr do lista_de_comandos endwhile
comando_repeat ::= repeat lista_de_comandos until expr
comando_if ::= if expr then lista_de_comandos endif
              | if expr then lista_de_comandos
                else lista_de_comandos endif
comando_read ::= read variavel
comando_write ::= write expr
comando_return ::= return expr
comando_exit ::= exit identificador
rotulo ::= identificador
variavel ::= identificador
            | chamada_ou_indexacao
chamada_ou_indexacao ::= indices )
chamada_de_proc ::= identificador
                   | chamada_ou_indexacao
indices ::= identificador ( expr
                           | indices , expr
expr ::= expr | termo_a

```

```

    | termo_a
termo_a ::= termo_a & fator_a
    |
    | fator_a
fator_a ::= not primario_a
    |
    | primario_a
primario_a ::= expr_a oprel expr_a
    |
    | expr_a
expr_a ::= expr_a opad termo_b
    |
    | termo_b
termo_b ::= termo_b opmul fator_b
    |
    | fator_b
fator_b ::= - primario_b
    |
    | primario_b
primario_b ::= variavel
    |
    | constante
    |
    | (expr)
constante ::= int_ou_char
    |
    | booleano
int_ou_char ::= inteiro
    |
    | char
opad ::= + | -
opmul ::= * | /
oprel ::= = | < | ≤ | > | ≥
inteiro ::= digito
    |
    | inteiro digito
booleano ::= true
    |
    | false
digito ::= 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
identificador ::= letra
    |
    | letra mais_id
mais_id ::= letra
    |
    | digito
    |
    | letra mais_id
    |
    | digito mais_id
letra ::= A | B | ... Z
char ::= ' letra '

```

4 Compilador para Orion

4.1 Metodologia para o Desenvolvimento do Compilador

O compilador da linguagem Orion pode ser implementado da seguinte forma: para a análise léxica é permitido o uso de uma das ferramentas relacionadas em 4.2.1. A tarefa de construir tabelas sintáticas LR(1) ou LALR(1) [1, 2] é uma tarefa muito simples de ser automatizada, de modo que ela raramente é implementada de outra forma que não por meio do uso de ferramentas geradoras de analisadores sintáticos. Para esta etapa do compilador as ferramentas disponíveis estão enumeradas em 4.2.2. Para a implementação da tabela de símbolos estão disponíveis na página do curso um pacote de rotinas para a linguagem C. A descrição deste pacote pode ser encontrado em [7], disponível na página do curso sob o nome **Tabela de Símbolos**. Para a geração do código intermediário **use** [5] que contém informações sobre o conjunto de instruções, registradores, primitivas, etc de TAM. Para consulta, use [1] ou [2], [3] e, também, as notas de aula [6].

4.2 Ferramentas de Apoio

As principais ferramentas disponíveis na internet que podem ser usadas no projeto do analisador léxico e sintático são:

4.2.1 Geradores de Analisadores Léxicos

- Lex: gerador de analisadores léxicos.
URL: <http://www.jflex.de/download.html>
- JFlex: gerador de analisadores léxicos.
URL: <http://www.jflex.de/download.html>
- Flex
URL: <http://www.eos.ncsu.edu/software/gnu/pages/flex.html>
- JLex: gerador de analisadores léxicos para JAVA (1997).
URL: <http://www.cs.princeton.edu/appel/modern/java/JLex>.

4.2.2 Geradores de Analisadores Sintáticos

- YACC: gerador de analisador sintático LALR para a linguagem C [4].
- GNU: gerador de analisador sintático LALR para a linguagem C.

URL:http://sr.d.yahoo.com/srst/84677/%bbison+/1/109/*http://www.cs.cmu.edu/afs/cs/project/atk-ftp/bison/

- CUP: gerador de analisadores sintáticos LALR para JAVA (1998).

URL: <http://www.cs.princeton.edu/appel/modern/java/CUP>

4.3 Gramática de Orion modificada para a Implementação

Esta seção mostra a gramática apresentada na Seção 3.1 pronta para ser usada na implementação do compilador. Note que a gramática está ambígua para as expressões. Portanto para evitar conflitos e para que as expressões produzam resultados corretos é necessário que antes sejam estabelecidas as precedências e associatividades de seus operadores. Em alguns casos, como por exemplo, para o operador unário é necessário também associar a precedência na produção.

As precedências e associatividades são:

```
precedence left OR;
precedence left AND;
precedence left NOT;
precedence left NE, EQ, LE, LT, GT, GE;
precedence left PLUS, MINUS;
precedence left MULT, DIV;
precedence right EXP;
precedence left UMINUS;
```

onde AND, NOT, NE, EQ, LE, LT, GT, GE, PLUS, MINUS, MULT, DIV, EXP e UMINUS são constantes representando os operadores: |, &, not, not=, =, <, ≤, >, ≥, +, -, *, /, , ** e - unário, respectivamente.

Os nomes: IDENT, CONST_CHAR e NUMBER que aparecem na BNF da gramática de Orion representam, respectivamente, os identificadores, constantes literais e as constantes retornadas pelo analisador léxico.

```
program ::= program M2 declaracoes M0 block
block ::= begin lista_comandos M0 end
declaracoes ::= declaracoes M0 declaracao ;
               |
               vazio
declaracao ::= decl_de_var
              |
              def_de_tipo
```

```

    | decl_de_proc
decl_de_var ::= tipo : lista_de_ids
    tipo ::= integer
        | boolean
        | char
        | tipo_definido
    M0 ::= vazio
    M1 ::= vazio
    M2 ::= vazio
    def_de_tipo ::= type nome_do_tipo M0 = M1 definicao_de_tipo
    nome_do_tipo ::= identificador
    definicao_de_tipo ::= ( limites ) tipo
        limites ::= inteiro : inteiro
    tipo_definido ::= identificador
    decl_de_proc ::= proc_cab proc_corpo
        proc_cab ::= tipo_retornado procedure M0 nome_do_proc
                    espec_de_parametros
        proc_corpo ::= : declaracoes M0 bloco emit_return
            | emit_return
        emit_return ::= vazio
    lista_de_parametros ::= parametro
        | lista_de_parametros , parametro
    tipo_retornado ::= integer
        | boolean
        | char
        | vazio
    parametro ::= modo tipo : identificador
        modo ::= value
        | reference
    nome_do_proc ::= identificador
    lista_de_comandos ::= comando
        | lista_de_comandos ; M0 comando
    lista_de_ids ::= identificador
        | lista_de_ids , identificador
    vazio ::= ε
    espec_de_parametros ::= ( lista_de_parametros )
        | vazio

```

```

comando ::= comando_de_atribuicao
          | comando_while
          | comando_repeat
          | comando_if
          | comando_read
          | comando_write
          | comando_return
          | comando_exit
          | chamada_de_procedimento
          | rotulo : comando

comando_de_atribuicao ::= variavel := expr
comando_while ::= while M0 expr do M0 lista_de_comandos endwhile
comando_repeat ::= repeat M0 lista_de_comandos until M0 expr
comando_if ::= if expr then M0 lista_de_comandos endif
              | if expr then M0 lista_de_comandos M1
                else M0 lista_de_comandos endif
comando_read ::= read variavel
comando_write ::= write expr
comando_return ::= return expr
comando_exit ::= exit identificador
rotulo ::= identificador
variavel ::= identificador
           | chamada_ou_indexacao
chamada_ou_indexacao ::= indices )
chamada_de_proc ::= identificador
                   | chamada_ou_indexacao
indices ::= variavel2 ( expr
                     | indices , expr
variavel2 ::= identificador
expr ::= expr | M0 expr
       | expr & M0 expr
       | not expr
       | expr not = expr
       | expr < expr
       | expr > expr
       | expr >= expr

```

```

|   expr  <=  expr
|   expr  +  expr
|   expr  -  expr
|   expr  *  expr
|   expr  /  expr
|   expr  **  expr
|   -  expr %prec UMINUS
|   variavel
|   constante
|   (expr)
constante ::= int_ou_char
            |
            | booleano
int_ou_char ::= inteiro
            |
            | CONST_CHAR
inteiro ::= NUMBER
booleano ::= true
            |
            | false
identificador ::= IDENT

```

4.4 Interpretador TAM

A implementação do interpretador TAM é baseada na descrição dada em [3] e [5]. O código listado em [3] foi ligeiramente modificado e reescrito para a linguagem Java [5]. Duas classes foram implementadas: **RuntimeOrganization** que define algumas constantes usadas pelo interpretador e **Main** que contém o interpretador propriamente dito.

4.4.1 Como utilizar o Interpretador TAM

Para executar o interpretador é necessário o interpretador Java JDK que pode ser obtido no *site* www.java.sun.com. Com este interpretador instalado em seu computador, para executar o interpretador TAM, digite na linha de comando:

java interpretador.Main arqIn,
onde **arqIn** é o nome do arquivo a ser executado.

Veja um exemplo. Suponha que o arquivo gerado pelo seu compilador chama-se **codigo.out**. Para executá-lo, digite:

```
java interpretador.Main codigo.out
```

4.4.2 Exemplo

Dado o exemplo na linguagem Orion apresentado na Seção ??,

```
program
    integer : i;
begin
    i := 20;
    while (i > 10) do
        write(i+10);
        i := i - 1
    endwhile;
    write i
end
```

o arquivo **codigo.out** correspondente ao exemplo teria o seguinte código gerado para o interpretador TAM:

nro	op	r	n	d	mnemonico
0:	10	0	0	1	; PUSH 1
1:	3	0	0	3	; LOADL 3
2:	13	0	0	0	; JUMPI
3:	1	4	0	0	; LOADA 0[SB]
4:	3	0	0	20	; LOADL 20
5:	0	5	1	-2	; LOAD(1) -2[ST]
6:	5	0	0	1	; STOREI 1
7:	11	0	0	1	; POP(0) 1
8:	1	4	0	0	; LOADA 0[SB]
9:	3	0	0	10	; LOADL 10
10:	0	5	1	-2	; LOAD(1) -2[ST]
11:	2	0	0	1	; LOADI 1
12:	11	0	0	1	; POP(0) 1
13:	0	5	1	-2	; LOAD(1) -2[ST]
14:	6	2	0	16	; gt
15:	11	0	1	2	; POP(1) 2
16:	14	0	1	19	; JUMPIF(1) 19[CB]
17:	3	0	0	43	; LOADL 43
18:	13	0	0	0	; JUMPI
19:	1	4	0	0	; LOADA 0[SB]
20:	3	0	0	10	; LOADL 10
21:	0	5	1	-2	; LOAD(1) -2[ST]
22:	2	0	0	1	; LOADI 1
23:	11	0	0	1	; POP(0) 1
24:	0	5	1	-2	; LOAD(1) -2[ST]
25:	6	2	0	8	; add
26:	11	0	1	2	; POP(1) 2
27:	6	2	0	26	; putint
28:	1	4	0	0	; LOADA 0[SB]
29:	1	4	0	0	; LOADA 0[SB]
30:	3	0	0	1	; LOADL 1
31:	0	5	1	-2	; LOAD(1) -2[ST]
32:	2	0	0	1	; LOADI 1
33:	11	0	0	1	; POP(0) 1
34:	0	5	1	-2	; LOAD(1) -2[ST]
35:	6	2	0	9	; sub
36:	11	0	1	2	; POP(1) 2
37:	0	5	1	-2	; LOAD(1) -2[ST]
38:	5	0	0	1	; STOREI 1
39:	11	0	0	1	; POP(0) 1
40:	3	0	0	8	; LOADL 8
41:	13	0	0	0	; JUMPI
42:	11	0	0	1	; POP(0) 1
43:	1	4	0	0	; LOADA 0[SB]
44:	2	0	0	1	; LOADI 1
43:	11	0	0	1	; POP(0) 1
44:	6	2	0	26	; putint
45:	15	0	0	0	; HALT

cujo resultado da execução é:

30
29
28
27
26
25
24
23
22
21
10

4.5 Resultados Desejados

Além da listagem do programa, exibindo compilação e execução de exemplos, **a documentação do projeto também deverá ser produzida.**

“A documentação do programa é como um pequeno artigo que explica o que o programa faz, como faz, e apresenta conclusões obtidas sobre o trabalho. A documentação é um documento à parte e não deve ser escrita no programa fonte”.

A documentação deverá, **no mínimo**, incluir o seguinte:

Descrição sucinta sobre o desenvolvimento do trabalho

Uma explicação sobre as decisões de implementação tomadas, uma visão geral do funcionamento do programa, comentários sobre os testes executados, etc.

Descrição das estruturas de dados utilizada

Voce pode fazer esta descrição utilizando desenhos ou escrevendo.

Descrição do formato de entrada dos dados

Uma descrição simples e clara dizendo quais são os dados de entrada e como o programa irá recebê-los.

Descrição do formato de saída dos dados

Uma descrição simples e clara dizendo como o programa apresentará os resultados ao usuário. Por exemplo:

“O programa irá gerar uma seqüência de caracteres representado os *tokens* reconhecidos pelo analisador léxico, como: palavras reservadas, operadores, identificadores. Eles serão apresentados em três colunas, onde na primeira terá o próprio *token* ou seja o *lexema*, na segunda o *token* e na terceira a sua classe ou valor.

7 Bibliografia

References

- [1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986.
- [2] Aho, A.V.; Sethi, R.; Ullman, J.D. *Compiladores – Princípios, Técnicas e Ferramentas*, (Compilers Principles, Techniques, and Tools), Ed. Guanabara Koogan S.A., 1995, Tradução de: Daniel de Ariosto Pinto.
- [3] Watt, David A., *Programming Language Processors*, C.A.R. Hoare Series Editor, Prentice Hall International Series in Computer Science, 1993.
- [4] Johnson, S. C., *YACC - Yet another compiler compiler*, Computing Science Technical Report 32, AT & T, Bell Laboratories Murray Hill,N J , 1975.
- [5] Reuber G. Duarte, Mariza A. S. Bigonha, *Implementação do Interpretador TAM*. LLP013/99.
- [6] Roberto S. Bigonha, Mariza A. S. Bigonha, Transparências do Curso de Compiladores, Capítulo 8 - Geração de Código Intermediário.
- [7] Silva, Yêdda A. D. e Bigonha, Mariza Andrade da Silva, “Tabela de Símbolos: Implementação e Avaliação das Principais Operações em Diferentes Paradigmas de Programação”, *Relatório Técnico do Laboratório de Linguagens de Programação LLP04/99*, Departamento de Ciência da Computação, UFMG, março/1999.