Projeto do Compilador

A Linguagem

Convenção Léxica

1. Palavras revervadas (**keywords**) são as seguintes:

```
else   if   int   return   void   while
```

Não podem ser usadas como nomes de variáveis/funções, e devem ser escritas em minúsculo.

2. Símbolos especiais são:

```
+  -  *  /  <  <=  >  >=  ==  !=  =  ;  ,  (  )  [  ]  {  }  /*  */
```

3. Outros lexemas são **ID** e **NUM**, definidas pelas seguintes expressões regulares:

```
ID = letter (letter | digit)*
NUM = digit digit*
letter = a | .. | z | A | .. | Z
digit = 0 | .. | 9
```

Há distinção entre letras em maiúsculo ou minúsculo.

4. Espaços em branco consistem em caracter branco (' '), quebra-de-linha ('\n'), e tabs ('\t'). Devem ser ignorados exceto quando for necessário separar ID's e NUM's, e palavras reservadas.

5. Comentários são escritos da mesma forma que na linguagem C usando notações /* ... */. Podem ser inseridos em qualquer parte do código onde é permitido colocar espaços em branco; isto é, comentários não podem ser inseridos dentro de tokens, e podem incluir mais de uma linha. Comentários não podem ser aninhados.

Sintaxe

A gramática em BNF (Backus–Naur form) é descrita abaixo:

```
<program> ::= <declaration-list>
```

```
<declaration-list> ::= <declaration-list> <declaration> | <declaration>
<declaration> ::= <var-declaration> | <fun-declaration>

<var-declaration> ::= <type-specifier> ID ; | <type-specifier> ID [ NUM ] ;
<type-specifier> ::= int | void

<fun-declaration> ::= <type-specifier> ID ( <params> ) <compound-stmt>
<params> ::= <param-list> | void
<param-list> ::= <param-list> , <param> | <param>
<param> ::= <type-specifier> ID | <type-specifier> ID [ ]

<compound-stmt> ::= { <local-declarations> <statement-list> }

<local-declarations> ::= <local-declarations> <var-declaration> | empty
<statement-list> ::= <statement-list> <statement> | empty

<statement> ::= <expression-stmt> | <compound-stmt> | <selection-stmt>
              | <iteration-stmt> | <return-stmt>
<expression-stmt> ::= <expression> ; | ;

<selection-stmt> ::= if ( <expression> ) <statement>
                   | if ( <expression> ) <statement> else <statement>

<iteration-stmt> ::= while ( <expression> ) <statement>

<return-stmt> ::= return ; | return <expression> ;

<expression> ::= <var> = <expression> | <simple-expression>
<var> ::= ID | ID [ <expression> ]

<simple-expression> ::= <additive-expression> <relop> <additive-expression>
                      | <additive-expression>
<relop> ::= <= | < | > | >= | == | !=

<additive-expression> ::= <additive-expression> <addop> <term> | <term>
<addop ::= + | -
<term> ::= <term> <mulop> <factor> | <factor>
<mulop> ::= * | /

<factor> ::= ( <expression> ) | <var> | <call> | NUM

<call> ::= ID ( <args> )
<args> ::= <arg-list> | empty
<arg-list> ::= <arg-list> , <expression> | <expression>
```

MATA61 Compiladores

Semântica

Para cada conjunto de regras da gramática temos uma breve explicação da semântica associada a elas:

```
<program> ::= <declaration-list>
<declaration-list> ::= <declaration-list> <declaration> | <declaration>
<declaration> ::= <var-declaration> | <fun-declaration>
```

Um programa consiste de uma lista (ou sequência) de declarações, que podem ser funções ou variáveis, em qualquer ordem. É necessário haver ao menos uma declaração. Restrições semânticas são descritas a seguir (e não ocorrem em C). Todas as variáveis e funções devem ser declaradas antes de serem usadas (evitando *backpatching* de referências). A última declaração em um programa precisa ser uma declaração da forma **`void main(void)`**. Note que não existem protótipos nessa linguagem, então não existe distinção entre declaração e definição (ao contrário de C).

```
<var-declaration> ::= <type-specifier> ID ; | <type-specifier> ID [ NUM ] ;
<type-specifier> ::= int | void
```

Uma declaração de variável declara uma variável de tipo inteiro ou um vetor ao qual o tipo base é um inteiro em que os índices ficarão na faixa `0 .. NUM-1`. Note que os únicos tipos básicos são `int` e `void`. Em uma declaração de variável somente o especificador de tipo `int` pode ser usado. `Void` é apenas para declarações de funções (como explicado abaixo). Note, também, que apenas uma variável pode ser declarada por declaração.

```
<fun-declaration> ::= <type-specifier> ID ( <params> ) <compound-stmt>
<params> ::= <param-list> | void
<param-list> ::= <param-list> , <param> | <param>
<param> ::= <type-specifier> ID | <type-specifier> ID [ ]
```

Uma declaração de função consiste no especificador de tipo de retorno, um identificador, uma lista de parâmetros entre parênteses e separados por vírgula, seguido de um bloco de instruções compostas. Se o tipo de retorno da função é `void`, então a função não retorna qualquer valor (i.e. é um procedimento). Parâmetros de uma função são `void` (i.e. não há parâmetros) ou uma lista representando seus parâmetros. Parâmetros seguido de colchetes são parâmetros de vetores ao qual o tamanho é variável. Inteiros são passados por valor. Vetores são passados por referência (i.e. como ponteiros) e devem corresponder a uma variável do tipo vetor durante a chamada de função. Note que não existem parâmetros do tipo função. Os parâmetros de uma função tem escopo igual ao bloco de instruções compostas da função, e cada invocação de função tem seu próprio conjunto de parâmetros. Funções podem ser recursivas (ao limite que declarações antes de uso

permitem).

MATA61 Compiladores

```
<compound-stmt> ::= { <local-declarations> <statement-list> }
```

Um bloco de instruções compostas consiste de um par de chaves entre uma série de declarações e instruções. A execução de um bloco desse tipo se dá pela execução da sequência de instruções na ordem em que foram especificadas. A declaração de variáveis tem escopo igual ao do conjunto de instruções entre chaves e substituem a visibilidade de variáveis globais.

```
<local-declarations> ::= <local-declarations> <var-declaration> | empty
<statement-list> ::= <statement-list> <statement> | empty
```

Note que tanto declarações quanto lista de instruções podem ser vazios (O não-terminal *empty* significa uma palavra vazia, também referida como ε).

```
<statement> ::= <expression-stmt> | <compound-stmt> | <selection-stmt>
              | <iteration-stmt> | <return-stmt>
<expression-stmt> ::= <expression> ; | ;
```

Uma instrução do tipo expressão é uma expressão opcional seguida de um ponto e virgula. Tais expressões são normalmente avaliadas afim de obter efeitos colaterais. Portanto, esse tipo de instrução é usado para atribuições e chamadas de função.

```
<selection-stmt> ::= if ( <expression> ) <statement>
                   | if ( <expression> ) <statement> else <statement>
```

Uma instrução do tipo `if` tem as semânticas usuais: A expressão é avaliada; Um valor diferente de zero causa a execução da primeira instrução; Um valor de zero causa a execução da segunda instrução, se esta existe. Essa regra resulta na ambiguidade clássica do *else pendente*, que é resolvida da maneira padrão: A parte do `else` é analisada como uma subestrutura imediata do `if` atual.

```
<iteration-stmt> ::= while ( <expression> ) <statement>
```

A instrução de tipo `while` é a única instrução de iteração da linguagem. A execução desta se dá pela repetida execução de uma expressão, seguido da execução de uma instrução caso a expressão for avaliada como diferente de zero, acabando somente quando a expressão for avaliada como zero.

```
<return-stmt> ::= return ; | return <expression> ;
```

Uma instrução de retorno pode ou não retornar um valor. Funções não declaradas como `void` devem retornar valores. Funções declaradas como `void` não devem retornar valores. Uma instrução de retorno transfere o controle de execução de volta para o chamador (ou a terminação do programa caso a função atual seja o `main`).

```
<expression> ::= <var> = <expression> | <simple-expression>
<var> ::= ID | ID [ <expression> ]
```

Uma expressão é uma expressão simples, ou uma referência a uma variável seguido de um símbolo de atribuição e uma expressão. A atribuição tem as semânticas habituais de armazenamento: O local de uma variável representada por `var` é encontrado, então a sub expressão a direita da atribuição é avaliada, e o valor desta é armazenado no local encontrado. Este valor é também retornado como o valor da expressão como um todo. Uma variável é ou uma variável simples, de tipo inteiro, ou a indexação de uma variável vetor. Uma indexação negativa causa a parada do programa (diferentemente de C). No entanto, o limite superior de indexações não é verificado.

Variáveis apresentam ainda mais uma restrição nessa linguagem quando comparada a C. Em C, o alvo de uma atribuição deve ser um valor-l , e tais valores são endereços que podem ser obtidos através de diversas operações. Nesta linguagem, entretanto, o único valor-l são aqueles obtidos pela produção *var*, e portanto esta categoria é checada semanticamente, e não durante a verificação de tipos, como acontece em C. Por consequência, aritmética de ponteiro não é permitido nessa linguagem.

```
<simple-expression> ::= <additive-expression> <relop> <additive-expression>
                      | <additive-expression>
<relop> ::= <= | < | > | >= | == | !=
```

Uma expressão simples consiste de operadores relacionais que não se associam (isto é, uma expressão sem parênteses pode conter apenas um operador relacional). O valor de uma expressão simples é o valor de sua expressão aditiva, se não há operadores relacionais na expressão simples, ou valor 1 caso o operador relacional seja avaliado como verdadeiro, ou valor 0 caso avaliado como falso.

```
<additive-expression> ::= <additive-expression> <addop> <term> | <term>
<addop ::= + | -
```

```
<term> ::= <term> <mulop> <factor> | <factor>
<mulop> ::= * | /
```

Expressões aditivas e termos representam a associatividade e precedência típica de operadores aritméticos. O símbolo / representa uma divisão inteira; isto é, qualquer resto é truncado.

```
<factor> ::= ( <expression> ) | <var> | <call> | NUM
```

Um fator é uma expressão entre parênteses; ou uma variável, que é avaliada como o valor da variável; ou uma chamada de função, que é avaliada como o retorno desta função; ou um ***NUM***, o qual o valor é computado pelo scanner. Uma variável vetor precisa ser indexada, exceto no caso de expressões contendo um único **ID**, usadas juntamente em chamadas de funções com parâmetros de tipo vetor (mais detalhes seguem abaixo).

```
<call> ::= ID ( <args> )
<args> ::= <arg-list> | empty
<arg-list> ::= <arg-list> , <expression> | <expression>
```

Uma chamada de função consiste de um **ID** (o nome da função) seguido de parênteses em volta de seus argumentos. Argumentos são ou vazios ou consistem de uma lista de expressões separadas por vírgula, representando os valores a serem atribuídos aos parâmetros da função durante a chamada. Funções precisam ser declaradas antes de serem chamadas, e o número de parâmetros na declaração deve ser igual ao número de argumentos na chamada. Um parâmetro de tipo vetor precisa casar com uma expressão consistindo de um único identificador, representando a variável vetor a ser passada para a função.

Ademais, as regras acima não definem qualquer instrução de entrada e saída. Devemos, portanto, incluir essa funcionalidade na definição da linguagem, dado que, ao contrário de C, não temos semânticas para interação com bibliotecas ou unidades de compilação. Iremos, portanto, considerar que existem duas funções predefinidas no espaço global do programa, como se tivessem a seguinte declaração:

```
int input(void)  {...}
void println(int x) {...}
```

Os detalhes de implementação dessas funções está descrito nas notas de implementação do projeto.

Exemplos de código

```
int sum(int x){
  if(x>0)
    return x + sum(x-1);
```

```
    return x + sum(x-1);
return 0;
}

void main(void){
  println(sum(10));
}



/* A program to perform Euclid's
   Algorithm to compute gcd. */

int gcd (int u, int v)
{
    if (v == 0) return u ;
    else return gcd(v,u-u/v*v);
    /* u-u/v*v == u mod v */
}

void main(void)
{
    int x;
    int y;
    x = input();
    y = input();
    println(gcd(x,y));
}



/* A program to perform selection sort on a 10 element array */
int x[10];
int minloc(int a[], int low, int high) {
    int i;
    int x;
    int k;
    k = low;
    x = a[low];
    i = low + 1;
    while (i < high) {
        if (a[i] < x) {
            x = a[i];
            k = i;
        }
        i = i + 1;
```

```
    }
    return k;
}
void sort(int a[], int low, int high) {
    int i;
    int k;
    i = low;
    while (i < high - 1) {
        int t;
        k = minloc(a, i, high);
        t = a[k];
        a[k] = a[i];
        a[i] = t;
        i = i + 1;
    }
}
void main(void) {
    int i;
    i = 0;
    while (i < 10) {
        x[i] = input();
        i = i + 1;
    }
    sort(x, 0, 10);
    i = 0;
    while (i < 10) {
        println(x[i]);
        i = i + 1;
    }
}
```