Apêndice A

Projeto de Compilador

Definimos aqui uma linguagem de programação denominada **C–** (pronuncia-se "C menos"), que é uma linguagem apropriada para um projeto de compilador por ser mais complexa que a linguagem TINY, pois inclui funções e matrizes. Ela é essencialmente um subconjunto de C, mas sem algumas partes importantes, o que justifica seu nome. Este apêndice é composto por cinco seções. Na primeira, listamos as convenções léxicas da linguagem, incluindo uma descrição dos marcadores da linguagem. Na segunda, apresentamos uma descrição em BNF de cada construção da linguagem, juntamente com uma descrição em português da semântica associada. Na terceira seção, apresentamos dois programas de exemplo em C–. Na quarta, descrevemos um ambiente de execução TINY para C–. Na última seção, descrevemos alguns projetos de programação usando C– e TM, adequados para um curso de programação.

A.1 CONVENÇÕES LÉXICAS DE C–

1. As palavras-chave da linguagem são as seguintes:

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

   Todas as palavras-chave são reservadas e devem ser escritas com caixa baixa.
2. Os símbolos especiais são os seguintes:

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

3. Há, ainda, os marcadores *ID* e *NUM*, definidos pelas expressões regulares a seguir:

```
ID = letra letra*
NUM = dígito dígito*
letra = a|..|z|A|..|Z
dígito = 0|..|9
```

   Existe diferença entre caixa baixa e caixa alta.

4. Espaço em branco é composto por brancos, mudanças de linha e tabulações. O espaço em branco é ignorado, exceto como separador de **IDs**, **NUMs** e palavras-chave.
5. Comentários são cercados pela notação usual de C `/*...*/`. Os comentários podem ser colocados em qualquer lugar que possa ser ocupado por um espaço em branco (ou seja, comentários não podem ser colocados dentro de marcadores), e podem incluir mais de uma linha. Comentários não podem ser aninhados.

A.2 SINTAXE E SEMÂNTICA DE C–

Uma gramática em BNF para C– é apresentada a seguir:

1. *programa* → *declaração-lista*
2. *declaração-lista* → *declaração-lista declaração* | *declaração*
3. *declaração* → *var-declaração* | *fun-declaração*
4. *var-declaração* → *tipo-especificador* **ID** `;` | *tipo-especificador* **ID** `[` **NUM** `]` `;`
5. *tipo-especificador* → `int` | `void`
6. *fun-declaração* → *tipo-especificador* **ID** `(` *params* `)` *composto-decl*
7. *params* → *param-lista* | `void`
8. *param-lista* → *param-lista* `,` *param* | *param*
9. *param* → *tipo-especificador* **ID** | *tipo-especificador* **ID** `[]`
10. *composto-decl* → `{` *local-declarações statement-lista* `}`
11. *local-declarações* → *local-declarações var-declaração* | *vazio*
12. *statement-lista* → *statement-lista statement* | *vazio*
13. *statement* → *expressão-decl* | *composto-decl* | *seleção-decl* | *iteração-decl* | *retorno-decl*
14. *expressão-decl* → *expressão* `;` | `;`
15. *seleção-decl* → `if` `(` *expressão* `)` *statement* | `if` `(` *expressão* `)` *statement* `else` *statement*
16. *iteração-decl* → `while` `(` *expressão* `)` *statement*
17. *retorno-decl* → `return` `;` | `return` *expressão* `;`
18. *expressão* → *var* `=` *expressão* | *simples-expressão*
19. *var* → **ID** | **ID** `[` *expressão* `]`
20. *simples-expressão* → *soma-expressão relacional soma-expressão* | *soma-expressão*
21. *relacional* → `<=` | `<` | `>` | `>=` | `==` | `!=`
22. *soma-expressão* → *soma-expressão soma termo* | *termo*
23. *soma* → `+` | `-`
24. *termo* → *termo mult fator* | *fator*
25. *mult* → `*` | `/`
26. *fator* → `(` *expressão* `)` | *var* | *ativação* | **NUM**
27. *ativação* → **ID** `(` *args* `)`
28. *args* → *arg-lista* | *vazio*
29. *arg-lista* → *arg-lista* `,` *expressão* | *expressão*

Para cada uma dessas regras gramaticais, apresentamos uma breve explicação da semântica associada.

1. *programa* → *declaração-lista*
2. *declaração-lista* → *declaração-lista declaração* | *declaração*
3. *declaração* → *var-declaração* | *fun-declaração*

Um programa é composto por uma lista (ou seqüência) de declarações, que podem ser de funções ou de variáveis, em qualquer ordem. Deve haver pelo menos uma declaração.

As restrições semânticas são as seguintes (elas não ocorrem em C): todas as variáveis e funções devem ser declaradas antes do uso (isso evita referências para ajustes retroativos). A última declaração em um programa deve ser uma declaração de função, da forma `void main(void)`. Observe que em C– não existem protótipos, assim não são feitas distinções entre declarações e definições (como em C).

4. *var-declaração* → *tipo-especificador* **`ID`** `;` | *tipo-especificador* **`ID`** `[` **`NUM`** `]` `;`
5. *tipo-especificador* → `int` | `void`

Uma declaração de variável declara uma variável simples de tipo inteiro ou uma matriz cujo tipo básico é inteiro, e cujos índices variam de 0 . . **`NUM`** – 1. Observe que em C– os únicos tipos básicos são inteiro e vazio. Em uma declaração de variável, apenas o especificador de tipos `int` pode ser usado. `Void` é usado em declarações de função (ver a seguir). Observe também que apenas uma variável pode ser declarada em cada declaração.

6. *fun-declaração* → *tipo-especificador* `ID(` *params* `)` *composto-decl*
7. *params* → *param-lista* | `void`
8. *param-lista* → *param-lista* `,` *param* | *param*
9. *param* → *tipo-especificador* **`ID`** | *tipo-especificador* **`ID`** `[]`

Uma declaração de função é composta por um especificador de tipo de retorno, um identificador e uma lista de parâmetros entre parênteses separados por vírgulas, seguida de uma declaração composta contendo o código da função. Se o tipo de retorno da função é `void`, a função não retorna nenhum valor (ou seja, é um procedimento). Os parâmetros de uma função são `void` (ou seja, a função não tem parâmetros) ou uma lista que representa os parâmetros da função. Os parâmetros seguidos por colchetes são matrizes cujo tamanho pode variar. Parâmetros de inteiros simples são passados por valor. Parâmetros de matriz são passados por referência (ou seja, como ponteiros), e devem casar com uma variável de tipo matriz durante a ativação. Observe que não existem parâmetros de tipo "função". Os parâmetros de uma função têm escopo igual ao da declaração composta na declaração de função, e cada ativação de uma função tem um conjunto separado de parâmetros. Funções podem ser recursivas (na medida permitida pela declaração antes do uso).

10. *composto-decl* → `{` *local-declarações statement-lista* `}`

Uma declaração composta consiste de chaves envolvendo um conjunto de declarações. Uma declaração composta é executada com base na ordem em que aparecem as declarações entre as chaves.

As declarações locais têm escopo igual ao da lista de declarações da declaração composta e se sobrepõem a qualquer declaração global.

11. *local-declarações* → *local-declarações var-declaração* | *vazio*
12. *statement-lista* → *statement-lista statement* | *vazio*

Observe que tanto as declarações como as listas de declarações podem ser vazias. (O não-terminal *vazio* identifica a cadeia vazia, às vezes denotada como ε.)

13. *declaração* → *expressão-decl*
| *composto-decl*
| *seleção-decl*
| *iteração-decl*
| *retorno-decl*
14. *expressão-decl* → *expressão* `;` | `;`

Uma declaração de expressão tem uma expressão opcional seguida por um ponto-e-vírgula. Essas expressões são, em geral, avaliadas por seus efeitos colaterais. Assim, essa declaração é usada para atribuições e ativações de funções.

15. *seleção-decl* → `if` (*expressão*) *statement* | `if` (*expressão*) *statement* `else` *statement*

A declaração *if* tem a semântica usual: a expressão é avaliada; um valor diferente de zero provoca a execução da primeira declaração; um valor zero provoca a execução da segunda declaração, se ela existir. Essa regra resulta na clássica ambigüidade do *else* pendente, que é resolvida da maneira padrão: a parte *else* é sempre analisada sintática e imediatamente como uma subestrutura do *if* corrente (a regra de eliminação de ambigüidade do "aninhamento mais próximo").

16. *iteração-decl* → `while` (*expressão*) *statement*

A declaração *while* é a única declaração de iteração em C–. Ela é executada pela avaliação repetida da expressão e em seguida pela execução da declaração se a expressão receber valor diferente de zero, terminando quando a expressão receber valor zero.

17. *retorno-decl* → `return ;` | `return` *expressão* `;`

Uma declaração de retorno pode retornar ou não um valor. Funções que não sejam declaradas como `void` devem retornar valores. As funções declaradas como `void` não devem retornar valores. Um retorno transfere o controle de volta para o ativador (ou termina o programa se ele ocorrer dentro de `main`).

18. *expressão* → *var* `=` *expressão* | *simples-expressão*
19. *var* → `ID` | `ID` `[` *expressão* `]`

Uma expressão é uma referência de variável seguida por um símbolo de atribuição (sinal de igual) e uma expressão, ou apenas uma expressão simples. A atribuição tem a semântica de armazenamento usual: a localização da variável representada por *var* é identificada, a subexpressão à direita da atribuição é avaliada, e o valor da subexpressão é armazenado na localização dada. Esse valor também é retornado como o valor de toda a expressão. Uma *var* é uma variável inteira (simples) ou uma variável de matriz indexada. Um índice negativo leva à interrupção do programa (diferentemente de C). Entretanto, os limites superiores dos índices não são verificados.

As variáveis representam uma restrição adicional de C– em comparação a C. Em C, o alvo de uma atribuição deve ser um **l-valor**, e os l-valores são endereços que podem ser obtidos por diversas operações. Em C–, os únicos l-valores são os dados pela sintaxe *var*, e portanto essa categoria é verificada sintaticamente, em vez de durante a verificação de tipos como em C. Assim, a aritmética de ponteiros não é permitida em C–.

20. *simples-expressão* → *soma-expressão relacional soma-expressão* | *soma-expressão*
21. *relacional* → `<=` | `<` | `>` | `>=` | `==` | `!=`

Uma expressão simples é composta por operadores relacionais que não se associam (ou seja, uma expressão sem parênteses pode ter apenas um operador relacional). O valor de uma expressão simples é o valor de sua expressão aditiva se ela não contiver operadores relacionais, ou 1 se o operador relacional for avaliado como verdadeiro, ou ainda zero se o operador relacional for avaliado como falso.

22. *soma-expressão* → *soma-expressão soma termo* | *termo*
23. *soma* → `+` | `-`
24. *termo* → *termo mult fator* | *fator*
25. *mult* → `*` | `/`

Expressões
operadores aritm

26. *fator* →

Um fator é
valor; uma ativa
NUM, cujo valor
indexada, exceto
ativação de funçã

27. *ativação*
28. *args* →
29. *arg-lista*

Uma ativaç
argumentos ent
separadas por ví
ativação. As fun
parâmetros em u
Um parâmetro de
por um único ide

Finalmente,
mos incluir essas
pilação em separ
como **predefinid**

```
int input
void outp
```

A função i
entrada padrão (
valor é impresso
mudança de linh

A.3 PROGRAMA

O programa a seg

```
/* Um prc
   segund

int gcd (
{ if (v =
  else re
  /* u-u/
}

void main
{ int x;
  x = inp
  output(
}
```

Expressões e termos aditivos representam a associatividade e a precedência típicas dos operadores aritméticos. O símbolo / representa a divisão inteira; ou seja, o resto é truncado.

26. *fator* → (*expressão*) | *var* | *ativação* | **NUM**

Um fator é uma expressão entre parênteses, uma variável, que é avaliada como o seu valor; uma ativação de função, que é avaliada como o valor retornado pela função; ou um NUM, cujo valor é computado pelo sistema de varredura. Uma variável de matriz deve ser indexada, exceto no caso de uma expressão composta por um único ID e usada em uma ativação de função com um parâmetro de matriz (ver a seguir).

27. *ativação* → **ID** (*args*)
28. *args* → *arg-lista* | *vazio*
29. *arg-lista* → *arg-lista* , *expressão* | *expressão*

Uma ativação de função é composta por um ID (o nome da função), seguido por seus argumentos entre parênteses. Os argumentos são vazio ou uma lista de expressões separadas por vírgulas, representando os valores atribuídos aos parâmetros durante uma ativação. As funções devem ser declaradas antes de serem ativadas, e a quantidade de parâmetros em uma declaração deve igualar a quantidade de argumentos em uma ativação. Um parâmetro de matriz em uma declaração de função deve casar com uma expressão composta por um único identificador que represente uma variável de matriz.

Finalmente, as regras anteriores não dão declarações de entrada nem de saída. Precisamos incluir essas funções na definição de C–, pois, diferentemente de C, em C– não há compilação em separado nem recursos de vinculação. Portanto, consideramos duas funções como **predefinidas** no ambiente global, como se elas tivessem as declarações indicadas:

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

A função **input** não tem parâmetros e retorna um valor inteiro do dispositivo de entrada padrão (em geral, o teclado). A função **output** recebe um parâmetro inteiro, cujo valor é impresso no dispositivo de saída padrão (em geral, o monitor), juntamente com uma mudança de linha.

A.3 PROGRAMAS DE EXEMPLO EM C–

O programa a seguir recebe dois inteiros, computa o máximo divisor comum e o imprime:

```
/* Um programa para calcular o mdc
   segundo o algoritmo de Euclides. */

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();
  output(gcd(x,y));
}
```

O programa a seguir recebe uma lista de dez inteiros, ordena esses inteiros por seleção e os imprime de volta:

```
/* Um programa para ordenação por seleção de
   uma matriz com dez elementos. */
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[], in 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)
    { output(x[i]);
      i = i + 1; }
}
```

A.4 AMBIENTE

A descrição a seg
e o entendiment
(diferentemente
ser baseado em
pela pilha logo a
nem alocação dir
de ativação (ou q

Aqui, fp é o **pon**
acesso. O ofp (p
controle, conforn
(deslocamento de
cada quantidade
mento dos parâm
tem ponteiros de
utilizarão o fp, c

Por exempl

```
int f(int
{ int z;
  ...
}
```

então **x**, **y** e **z** pr
início da geração
x, **y** e **z** e duas lo
camentos de **x**, **y**

Referências
Ainda assim, de
essas variáveis p
trador fixo, que
simulador TM ar