UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA

DIM00164 – Compiladores

Martin A. Musicante

Especificação da linguagem a ser implementada por cada grupo[1]

Índice

[1] Adaptado de `https://www.uio.no/studier/emner/matnat/ifi/INF5110/v24/obligs/oblig2/languagespec/compila.html`.

1 Introdução

Este documento especifica e descreve a sintaxe e a semântica estática das linguagens Comp25.x, sendo $x \in \{1, 2, 3, 4, 5\}$. As diferentes variantes 1 a 5 da linguagem, correspondem às tarefas atribuidas a cada grupo da turma.

A semântica dinâmica, ou seja, a descrição do comportamento da linguagem ao ser executada, deve ser razoavelmente clara, mesmo sem especificação formal explícita.

1.1 Convenções de notação e sintaxe deste documento

Na descrição da gramática abaixo, usamos letras maiúsculas para não terminais. Como meta-símbolos para a gramática, usamos o seguinte:

```
→ , | , ( , ) , { , } , [ , ] , ”
```

As vírgulas na linha anterior são usadas como "meta-meta símbolos" na enumeração dos meta-símbolos.

Ao escrever a gramática em alguma variante de EBNF, $\{\dots\}$ representa iteração de zero ou mais vezes, $[\dots]$ representa cláusulas opcionais. Todo o resto, escrito como sequências contíguas, são símbolos terminais. Aqueles com apenas letras minúsculas são palavras-chave reservadas da meta-linguagem.

Note que os símbolos terminais das linguagens Comp25.x são escritos em `tipo texto` (quando o nome tiver mais de um caractere) ou entre aspas duplas "α" (quando o nome for apenas um caeactere) para distingui-los dos símbolos da meta-linguagem. Alguns símbolos terminais específicos são escritos em maiúsculas e sem aspas. Esses são:

- *NAME*,
- *INT_LITERAL*,
- *FLOAT_LITERAL* e
- *STRING_LITERAL*.

A próxima seção trata sobre aspectos léxicos para saber o que esses símbolos terminais representam.

2 Aspectos léxicos

2.1 Identificadores e literais

- *NAME* deve começar com uma letra, seguida por uma sequência (possivelmente vazia) de caracteres numéricos, letras e caracteres de sublinhado; o sublinhado não pode ocorrer no final. Letras maiúsculas e minúsculas são consideradas diferentes.
- Todas as palavras-chave das linguagens são escritas com letras minúsculas. As palavras-chave não pode ser usada para identificadores padrão.
- *INT_LITERAL* contém um ou mais caracteres numéricos.
- *FLOAT_LITERAL* contém um ou mais caracteres numéricos, seguidos por um ponto decimal, que é seguido por um ou mais caracteres numéricos. Opcionalmente, a sequencia de caracteres pode continuar com o símbolo `e`, seguido por *+* ou *-* e seguido de dois dígitos decimais.

- *STRING_LITERAL* consiste em uma sequência de caracteres, entre aspas ("). A sequência não pode conter mudança de linha, nova linha, retorno de carro ou similar. O valor semântico de a *STRING_LITERAL* é apenas a sequência em si, as aspas não fazem parte do valor da sequência em si.

2.2 Comentários

O Comp25.x suporta comentários de uma ou várias linhas .

Comentários de linha única começam com `//` e se estendem até o final dessa linha (como em, por exemplo, Java, C++ e na maioria dos dialetos C modernos). Comentários multilinha começam com `(*` e terminam com `*)`.

A última forma não pode ser aninhada. A primeira pode ser "aninhada" (no sentido de que uma linha comentada pode conter outra `//` ou os delimitadores de comentários de várias linhas, que são então ignorados).

3 Tipos de dados

3.1 Tipos de dados primitivos

As linguagens Comp25.x têm quatro tipos primitivos, além de possuir tipos definidos pelo usuário:

- Tipos primitivos:
  1. Números de ponto flutuante (`"float"`),
  2. Inteiros (`"int"`),
  3. Strings (`"string"`), e
  4. booleanos (`"bool"`).
- Tipos definidos pelo usuário:
  1. Cada (nome de um) registro representa um tipo.
  2. Tipos de referência, representando referências a elementos dos tipos especificados. O construtor do tipo de referência pode ser aninhado.

3.2 Registros

As linguagens suportam registros. Para pessoas vindas de Java ou C++, registros podem ser vistos como uma forma (muito) simples de classes, contendo apenas variáveis de instância como membros, mas que não suportam métodos, nem herança, nem construtores explicitamente programáveis. "Variáveis de instância" são mais comumente chamadas de *campos de registro* ou apenas *campos* ao lidar com registros. Registros suportam instanciação, por meio da palavra-chave `new`. Outro aspecto que se assemelha a classes como em Java é que variáveis do tipo registro contêm um apontador para um elemento ("objeto") daquele tipo de registro ou o valor especial do apontador `null`.[2]

[2] Os registros às vezes também são chamados de "`structs`".

3.3 Referências

As linguagens Comp25.x permitem que variáveis sejam declaradas como sendo do tipo de referência ("apontadores´´ ou "ponteiros´´, se preferir) escrevendo, por exemplo,

```
var x:  ref(int);
```

Variáveis, como `x`, podem receber valores que são referências, então, por exemplo, o seguinte é permitido:

```
var y:  int;
e := 42;
x := ref(y);
```

Da mesma forma, pode-se "seguir´´ uma referência `r` usando `deref(r)`, de modo que, dadas as definições anteriores, o seguinte é legal:

```
y := deref(x);
```

Expressões com `deref` também podem ser usadas como *L-values*, para que possamos atribuir valores ao local para o qual eles estão apontando, por exemplo:

```
deref(x) := y;
```

Veja também mais adiante o exemplo do procedimento de *swap* no contexto de passagem de parâmetros por referência.

4 Sintaxe

4.1 Gramática

As regras de produção seguintes, em EBNF, descrevem a sintaxe da linguagem de base. As variantes para cada grupo são definidas mais adiante, com base nessa gramática. Para precedências e associatividade de várias construções, veja mais adiante.

PROGRAM	→	`program` *NAME* `begin` [*DECL* {";" *DECL*}] `end`
DECL	→	*VAR_DECL* | *PROC_DECL* | *REC_DECL*
VAR_DECL	→	`var` *NAME* ":" *TYPE* [`:=` *EXP*] | `var` *NAME* `:=` *EXP*
PROC_DECL	→	`procedure` NAME "(" [*PARAMFIELD_DECL* { ","*PARAMFIELD_DECL* }] ")" [":" TYPE] `begin` [[*DECL* {";" *DECL*}] `in`] *STMT_LIST* `end`
REC_DECL	→	`struct` *NAME* "{" [*PARAMFIELD_DECL* { ";" *PARAMFIELD_DECL* }] "}"
PARAMFIELD_DECL	→	*NAME* ":" *TYPE*
STMT_LIST	→	[*STMT* { ";" *STMT*}]

EXP → *EXP LOG_OP EXP*
| `not` *EXP*
| *EXP REL_OP EXP*
| *EXP ARITH_OP EXP*
| *LITERAL*
| *CALL_STMT*
| `new` *NAME*
| *VAR*
| *REF_VAR*
| *DEREF_VAR*
| "(" *EXP* ")"

REF_VAR → `ref` "(" *VAR* ")"

DEREF_VAR → `deref` "(" *VAR* ")" | `deref` "(" *DEREF_VAR* ")"

VAR → *NAME* | *EXP* "." *NAME*

LOG_OP → `&&` | `||`

REL_OP → "<" | "<=" | ">" | ">=" | "=" | "<>"

ARITH_OP → "+" | "−" | "∗" | "/" | "^"

LITERAL → *FLOAT_LITERAL* | *INT_LITERAL* | *STRING_LITERAL* | *BOOL_LITERAL* | `null`

BOOL_LITERAL → `true` | `false`

STMT → *ASSIGN_STMT*
| *IF_STMT*
| *WHILE_STMT*
| *RETURN_STMT*
| *CALL_STMT*

ASSIGN_STMT → *VAR* ":=" *EXP* | *DEREF_VAR* ":=" *EXP*

IF_STMT → `if` *EXP* `then` *STMT_LIST* [`else` *STMT_LIST*] `fi`

WHILE_STMT → `while` *EXP* `do` *STMT_LIST* `od`

RETURN_STMT → `return` (*EXP*)

CALL_STMT → *NAME* "(" [*EXP* { "," *EXP* }] ")"

TYPE → `float` | `int` | `string` | `bool` | *NAME* | `ref` "(" *TYPE* ")"

4.2 Precedência

A precedência das seguintes construções é ordenada como a seguir (da precedência mais baixa para a mais alta):

1. `||`

2. `&&`
3. `not`
4. Todos os símbolos relacionais
5. `+` e `-`
6. `*` e `/`
7. `^` (exponenciação)
8. `.`("ponto", para acessar campos de um registro).

4.3 Associatividade

As operações binárias `||`, `&&`, `+`, `-`, `*`, e `.` são associativas à esquerda, mas a exponenciação é associativa à direita. Símbolos de relação não são associativos. Isso significa que por exemplo `a < b + c < d` é ilegal. É legal escrever `not not not b` e significa `(not (not (not b)))`.

5 Passagem de parâmetros

Ao descrever os mecanismos de passagem de parâmetros da linguagem, distinguimos (como é comumente feito) entre

- parâmetros reais e
- parâmetros formais.

Os parâmetros reais são as expressões (que incluem, entre outras coisas, variáveis) como parte de uma chamada de procedimento. Os parâmetros formais são as variáveis mencionadas como parte da definição do procedimento. A linguagem suporta apenas o mecanismo de passagem de parâmetros chamada por copia-valor:

5.1 Chamada por copia-valor

Os parâmetros formais são variáveis locais na definição do procedimento. Quando um procedimento está sendo chamado, os valores dos parâmetros locais são copiados para os parâmetros formais correspondentes.

5.2 Chamada por referência

A passagem de parâmetros por referência é implementada no estili da linguagem C, na qual, na hora da chamada, é copiado o endereço da variável, no lugar do valor nela contido.

5.3 Pequeno exemplo para chamada por referência

```
program swapexample
  begin
    procedure swap (a :  ref(int), b : ref(int))
    begin
      var tmp : int
    in
      tmp      := deref(a);
      deref(a) := deref(b);
      deref(b) := tmp
    end;
    procedure main ( )
      begin
        var x : int;
        var y : int;
        var xr : ref (int);
        var yr : ref (int)
       in
        x := 42; y := 84;
        xr := ref (x); yr := ref(y);
        swap (xr,yr)
      end
  end
```

6 Biblioteca padrão

A linguagem de programação vem com uma biblioteca padrão que oferece uma série de procedimentos de E/S. Toda leitura, ou seja, entrada, é feita a partir da entrada padrão ("stdin"). Toda escrita, ou seja, saída, é para a saída padrão ("stdout")

`proc readint():  int`	leia um número inteiro
`proc readfloat():  float`	leia um número de ponto flutuante
`proc readchar():  int`	leia um caractere e retorne seu valor ASCII. Retorne -1 para EOF
`proc readstring():  string`	leia uma string (até o primeiro espaço em branco)
`proc readline():  string`	leia uma linha
`proc printint(i:int)`	escreva um número inteiro
`proc printfloat(f:float)`	escreva um número de ponto flutuante
`proc printstr(s:string)`	escreva uma string
`proc printline(s:string)`	escreva uma string, seguida de uma "nova linha"

7 Semântica estática / tipagem / avaliação

7.1 Vinculação (*binding*) de nomes

A ocorrência normal de um identificador (sem um ponto precedente) é vinculada a uma declaração. Essa associação do uso de um identificador a uma declaração ("*binding*") pode ser descrita informalmente como segue: Examine o bloco ou escopo que envolve a ocorrência do identificador (onde o bloco se refere ao corpo do procedimento ou programa). A ocorrência de vinculação correspondente é a primeira declaração encontrada dessa maneira. Se nenhuma ocorrência de vinculação for encontrada dessa maneira, o programa é errôneo. Parâmetros formais contam como declarações locais para o corpo do procedimento.

Ocorrências de um nome precedido por um ponto que correspondem à cláusula *EXP* "."*NAME* na regra de produção para o não terminal *VAR* na gramática. Esses nomes são vinculados ao tipo da expressão *EXP* (que precisa ser um tipo registro) e procurar o campo com nome *NAME* naquele registro. É um erro, se *EXP* não for do tipo registro ou então, não houver tal campo naquele registro.

7.2 Tipo de construções compostas

expressões: expressões precisam ser verificadas quanto à correção de tipo da maneira óbvia. A expressão completa (caso seu tipo possa ser verificado) terá o seu tipo atribuído.

atribuições: Ambos os lados de uma atribuição devem ser do mesmo tipo. Nota: é permitido atribuir aos parâmetros formais de um procedimento. Isso se aplica tanto aos parâmetros de chamada por valor quanto aos parâmetros de chamada por referência. Claro, o efeito de uma atribuição nesses dois mecanismos é diferente.

condicionais e loop while: a condição (ou seja, expressão) na construção condicional deve ser do tipo bool. O mesmo para a condição no loop while.

seleção de campo de um registro: a expressão que está na frente de um ponto deve ser do tipo registro. o nome que fica depois de um ponto é o nome de um campo/atributo do tipo de registro da expressão na frente do ponto. O tipo da expressão de seleção de campo (se o tipo verificar) é o tipo conforme declarado para o campo do registro.

7.3 Tipos e conversão implícita de tipos

É permitido atribuir uma expressão do tipo inteiro a uma variável do tipo `float`. A situação inversa não é permitida. Não há operador de conversão de tipo. Se uma expressão aritmética tiver pelo menos um operando do tipo float, a operação é avaliada usando aritmética de ponto flutuante e o resultado é do tipo float. A exponenciação é sempre considerada feita com aritmética de ponto flutuante e o resultado é do tipo float.

7.4 Inferência de tipos

A linguagem requer uma forma muito simples de inferência de tipos (tão simples que nem se pode chamar de inferência de tipos!). As variáveis do programa precisam ser declaradas junto com o seu tipo usando a palavra-chave `var`. A declaração pode ser combinada com uma definição de um valor inicial (usando a sintaxe como `var x : int := 42`). Quando fornecido com uma expressão inicial, o tipo pode ser omitido (usando uma sintaxe como var x := 42), em cujo caso um tipo apropriado precisa ser inferido.

7.5 Avaliação de curto-circuito

Os operadores lógicos `&&` e `||` usam a chamada *avaliação de curto-circuito*. Isso significa que se o valor da expressão lógica puder ser determinado após alguém ter avaliado a primeira parte, apenas, o resto da expressão não é avaliado.

8 Procedimentos

- Em um procedimento, todas as declarações devem ocorrer antes do código executável (instruções). Declarações de variáveis, procedimentos e registros locais a um procedimento são

permitidas.

- Procedimentos chamados dentro de uma expressão devem ter um tipo de retorno definido. Esse tipo será o tipo da chamada em uma expressão.
- Sobre o número e os tipos de parâmetros de um procedimento: eles devem coincidir comparando a declaração/definição do procedimento e o uso de um procedimento. Esse requisito se aplica também ao mecanismo de passagem de parâmetros (ou seja, se a variável resp. parâmetro real é marcado como por referência).
- Declarações de retorno:
  - Um comando *return* é permitido somente dentro de subprogramas (`procedure`). Tal comando marca que o procedimento termina (e retorna). Além disso, `return` pode fornecer uma expressão para o valor a ser retornado ao chamador.
  - Se um procedimento for declarado sem tipo de retorno, o procedimento não precisa de uma declaração return. Nesse caso, o procedimento retorna (sem um valor de retorno) quando a última declaração no corpo do procedimento tiver sido executada.
  - Se um procedimento declarou um tipo de retorno, seu corpo precisa ter um comando `return` (com expressão correspondente do tipo correto). Esse comando precisa ser o último comando no corpo do procedimento.

9 Outras observações

- As declarações devem ser únicas por bloco. Duas declarações (dentro de um bloco) de um procedimento, um registro ou uma variável com o mesmo nome são consideradas declarações duplas, que são proibidas .
- O nome de um parâmetro formal não deve colidir com nomes de declarações locais dentro do procedimento. Além disso, os nomes de todos os parâmetros formais de um procedimento devem ser distintos.
- Todos os nomes utilizados devem ser declarados.
- Cada programa deve ter um procedimento chamado main. Este procedimento é o chamado no início do programa.

10 Variantes da linguagem

Comp25.1

É exatamente a linguagem descrita até aqui, com a adição de primitivas de iteração das formas:

FOR_STMT → `for` VAR "=" *EXP* `to` *EXP* `step` *EXP* `do` *STMT_LIST* `od`

DO_UNTIL_STMT → `do` *STMT_LIST* `until` *EXP* `od`

Comp25.2

É a linguagem descrita atá aqui, substituindo a estrutura *WHILE_STMT* por

do_STMT → `do` *STMT_LIST* `od`

STMT → ... ⟨ Todos os comandos ja definidos, menos o While ⟩
| `exit when` *EXP*

Comp25.3

É exatamente a linguagem descrita até aqui, com a adição de tipos array:

TYPE → ... ⟨ Todos os tipos ja definidos ⟩
| `array` [*INT_LITERAL*] `of` *TYPE*

VAR → ... ⟨ Todos os acessos a veriáveis ja definidos ⟩
| *EXP* "[" *EXP* "]"

Comp25.4

É exatamente a linguagem descrita até aqui, com a adição de tipos enumerados:

DECL → ... ⟨ Todas as declarações ja definidas ⟩
| `enum` *NAME* "=" "{" *NAME* { "," *NAME* } "}" `of` *TYPE*

Deve ser possível comparar elementos desses tipos, sendo a ordem definida pela sequencia da sua definição. É possível declarar e atribuir variáveis com esses tipos.

Comp25.5

É exatamente a linguagem descrita até aqui, com a adição de novas estruturas de seleção:

IF_STMT → ... ⟨ Todos os condicionais ja definidos ⟩
| `unless` *EXP* `do` *STMT_LIST* [`else` *STMT_LIST*] `od`
| `case` *EXP* `of` *CASE* { ";" *CASE* } "}" [`otherwise` *STMT_LIST*] `esac`

CASE → *INT_LITERAL* ["." "." *INT_LITERAL*] { "," *INT_LITERAL* ["." "." *INT_LITERAL*] } ":" *STMT_LIST*