CBesta

Bianca Cesarino Daniel Guerra João Pedro de Amorim
Max William Filgueira Paulo Medeiros

9 de Dezembro de 2020

1 Introdução

A linguagem *CBesta* tem como domínio de aplicação o ensino de programação para principiantes, enfatizando a importância e compreensão de conceitos como tipos, funções, procedimentos, ponteiros e afins.

2 Design de Implementação

2.1 Nível Léxico

A transformação do código-fonte em unidades léxicas foi realizada através da ferramenta `Alex`. Neste nível, foram definidos operadores, tipos, nomes de comandos, delimitadores, separadores, literais, palavras reservedas para a criação de subprogramas e registros, entre outros.

Algumas decisões de design foram feitas já no nível léxico, como por exemplo, a diferenciação entre nomes de tipos (que devem começar com letra maiúscula) e nomes de variáveis (que devem começar com letra minúscula). Além disso, a linguagem ignora espaços em branco e *tabs*.

2.2 Nível Sintático

Para realizar o *parsing* de unidades léxicas em sentenças de código, foi utilizada a linguagem de programação `Haskell` estendida pela biblioteca `Parsec`.

Em *CBesta*, as sentenças precisam ser separadas por um token (`;`). Além de sentenças simples, a linguagem também pode apresentar **sentenças compostas**, que são aquelas que podem conter outras sentenças em seu corpo (por exemplo, estruturas condicionais, de repetição, subprogramas e registros). Nesse caso, como existe a presença de delimitadores, o token de separação não é utilizado.

Outra decisão de design deste nível é que subprogramas não podem ser definidos dentro de subprogramas, ou seja, na linguagem não há suporte para subprogramas aninhados.

2.2.1 Estruturas condicionais

Em *CBesta*, temos apenas uma estrutura condicional, cujas regras sintáticas funcionam no seguinte formato, onde `<cond>` é uma expressão, e `<stmts1>` e `<stmts2>` são conjuntos de sentenças, possivelmente vazios:

```
if (<cond>) {
  <stmts1>
} else {
  <stmts2>
}
```

É importante ressaltar que a estrutura do `else` é opcional. Além disso, ainda há a possibilidade de, ao chegar no `else`, a estrutura de condição se repetir:

```
if (<cond1>) {
  <stmts1>
} else if (<cond2>) {
  <stmts2>
}
```

2.2.2 Estruturas de repetição

A linguagem suporta dois tipos de estruturas de repetição: `while` e `for`.

```
while (<cond>) {
  <stmts>
}
```

Nesse caso, mais uma vez `<cond>` é uma expressão e `<stmts>` é um conjunto possivelmente vazio de sentenças da linguagem.

```
for (<binding1>); <cond>; <binding2>) {
  <stmts>
}
```

Nessa estrutura, `<binding1>` e `<binding2>` são campos opcionais, correspondendo a uma inicialização ou atribuição de valor a uma variável; `<cond>` é uma expressão.

Mais uma vez, `<stmts>` é um conjunto possivelmente vazio de sentenças.

2.2.3 Estruturas de subprogramas

CBesta explicita a diferença entre funções e procedimentos, havendo uma palavra-chave distinta para a definição de cada um desses. No caso de funções, essa palavra-chave é `func`, e no caso de procedimentos, `proc`.

```
func <ret> <name> (<args>) {
  <stmts>
}

proc <name> (<args>) {
  <stmts>
}
```

Aqui, `<ret>`, presente apenas na definição da função, representa o seu tipo de retorno. Já `<name>` e `<args>` representam o nome do subprograma e uma lista de argumentos (possivelmente vazia), respectivamente. Mais uma vez, `<stmts>` é um conjunto possivelmente vazio de sentenças da linguagem.

Essa lista de argumentos tem a seguinte forma (em EBNF):

```
<args> -> <type> <name> (, <args>)
```

Sendo `type` um tipo e `name` um nome de variável.

2.2.4 Estruturas de registros

```
struct <name> {
  <declrs>
}
```

Nessa estrutura, `<name>` indica o nome do registro e `<declrs>`, em seu corpo, é um conjunto não-vazio de declarações.

2.3 Nível Semântico

2.3.1 Tabela de Símbolos

O estado da linguagem *CBesta* é composto por sete elementos: as tabelas de memória, de subgprogramas e de tipos; uma pilha de escopos; uma *flag booleana* que indica se estamos em tempo de execução; um contador de subprogramas ativos; e uma pilha de *booleanos* referentes a *loops*. A seguir, falamos um pouco de cada um desses:

- **Memória:** É uma tupla cujo primeiro elemento é uma lista de variáveis ativas e o segundo um contador indicando quantos elementos já foram alocados no escopo *heap*. O último é necessário pois é usado como identificador único de variáveis *heap*.

  Variáveis são também tuplas com os seguintes campos:

  - **Nome**: O identificador da variável.
  - **Escopo**: O escopo em que a variável foi declarada.

  - **Pilha de valores**: Uma pilha com os valores armazenados naquela variável. É necessário dado que nossa linguagem dá suporte a recursão. Dessa forma, cada ativação de subprograma que use essa variável (nome e escopo) terá um correspondente valor na pilha. Cada elemento da pilha será então o valor em si e o identificador da ativação do subprograma em que aquele valor foi inicializado.

- **Subprogramas:** Lista de subprogramas declarados. Um subprograma é composto de:
  - **Nome**: O identificador do subprograma.
  - **Tipo de retorno**: O tipo de retorno. Caso seja um procedimento, esse será nulo.
  - **Lista de argumentos formais**: Uma lista de tuplas com os nomes e tipos dos argumentos formais do subprograma.
  - **Corpo**: Uma lista de código (*tokens*) indicando o que deve ser executado na chamada do subprograma.

- **Tipos:** Lista de tipos, que contém os tipos definidos pelo usuário.

- **Pilha de escopos:** Uma pilha de *strings* indicando os escopos ativos na execução. É necessário dado que, numa chamada de subprograma, um novo escopo é criado enquanto o corrente ficará latente. Assim, uma pilha é ideal para essa modelagem. O tamanho dessa pilha será sempre igual ao *contador de subprogramas ativos* acrescido de uma unidade.

- **Flag de execução:** Indica se estamos em momento de execução. Sendo esse o caso, além do *parsing* sintático, também há alterações no estado, como a inserção, atualização e remoção de elementos nas tabelas que o compõem.

- **Contador de ativações de subprogramas:** Indica quantas ativações de subprogramas estão correntes.

- **Pilha de estados de *loop*:** Indica os estados dos *loops* correntes. É modelada como uma pilha pela mesma razão apresentada na pilha de escopos. Um estado pode ter um dos seguintes valores:
  - `OK`: Indica que o *loop* está em estado normal de execução.
  - `CONTINUE`: Indica que foi executado o comando `continue` dentro do *loop*.
  - `BREAK`: Indica que foi executado o comando `break` dentro do *loop*.
  - `RETURN`: Indica que foi executado o comando `return` dentro do *loop*.

2.3.2 Expressões

Em *CBesta*, não há conversão implícita de tipos (é uma linguagem fortemente tipada), sendo necessário o uso de *cast* para essa finalidade. Além disso, as expressões respeitam a precedência de operações definida na linguagem *C*.

Outro ponto a ser considerado é que a linguagem não permite expressões soltas no código, apenas quando compõem outros elementos da gramática, como atribuições.

2.3.3 Estruturas condicionais

Relembrando a sintaxe de uma estrutura condicional da linguagem, temos um exemplo de formato a seguir:

```
if (<cond1>) {
  <stmts1>
} else if (<cond2>) {
  <stmts2>
} else {
  <stmts3>
}
```

Quando uma estrutura condicional é executada na linguagem (ou seja, quando é *parseada* com a *flag* de execução ligada), a expressão em `<cond1>` é avaliada, de forma que seja esperado um valor `true` ou `false`, interrompendo a execução caso seja qualquer outro.

Caso o valor avaliado seja `true`, o conjunto `<stmts1>` é *parseado* ainda com a *flag* de execução ligada. Depois disso, essa *flag* é desligada até que toda a estrutura seja *parseada* apenas sintaticamente.

Caso o valor avaliado seja `false`, o conjunto `<stmts1>` é avaliado com a *flag* de execução desligada, até chegar no `else`, onde a *flag* é ligada novamente e todo esse processo se repete para o segundo `if`. No caso em que `<cond2>` seja `false`, como o próximo passo da estrutura é um `else` incondicional, ele sempre será executado com a *flag* de execução ligada. Similarmente, caso `<cond2>` seja `true`, o `else` será *parseado* sem ser executado.

Ao fim da estrutura, a linguagem assegura-se de que a *flag* de execução esteja ligada novamente, exceto em casos específicos, como o uso de `return`, `continue` ou `break`.

2.3.4 Estruturas de repetição

Assim como as estruturas condicionais, as estruturas de repetição podem aparecer tanto em um momento puramente sintático quanto em tempo de execução.

```
for (<binding1>); <cond>; <binding2>) {
  <stmts>
}
```

No caso em que a *flag* de execução está desligada, a estrutura só é *parseada* sintaticamente. Caso esteja ligada, ela é desligada temporariamente para que os `<binding1>`, `<cond>`, `<binding2>` e `<stmts>` sejam *parseados* apenas sintaticamente e seus *tokens* sejam extraídos.

Depois desse primeiro momento, a *flag* de execução volta a ficar ligada e o `<binding1>` é executado. O próximo passo é a execução da expressão `<cond>`, que precisa retornar um valor `true` ou `false`, interrompendo a execução do programa caso contrário.

Assim que o *loop* se inicia, adicionamos o estado `OK` à pilha de estados de *loop*. No caso em que `<cond>` é `true`, então o conjunto `<stmts>` é executado até que termine ou que alguma das seguintes palavras-chave seja encontrada nesse conjunto:

- `continue`: a *flag* de execução é desligada, o estado da pilha de estados de `loop` é alterado para `CONTINUE` e a *flag* de execução só é ligada novamente ao fim de `stmts`.
- `break`: a *flag* execução é desligada, o estado da pilha de estados de `loop` é alterado para `BREAK`. Então saímos do laço e a *flag* de execução é ligada mais uma vez.
- `return`: a *flag* execução é desligada, o estado da pilha de estados de `loop` é alterado para `RETURN`. Então saímos do laço, mas a *flag* de execução continua desligada.

Nos casos de `OK` (ou seja, quando nenhuma desses comandos esteve presente em `<stmts>`) ou de `CONTINUE`, `<binding2>` é executado e `<cond>` é checada mais uma vez. Caso seja `true`, o estado volta para `OK` e o laço é executado mais uma vez.

Isso ocorre até que `<cond>` seja avaliada como `false`, o que faz com que a *flag* de execução seja desligada e o corpo do laço seja *parseado* apenas sintaticamente.

Além disso, ao ser finalizado o laço em qualquer uma das condições, a pilha de estados de *loop* é atualizada, tendo seu primeiro elemento removido.

```
while (<cond>) {
  <stmts>
}
```

Já a estrutura do `while` é processada como a do `for`, mas com os campos de `<binding>` vazios.

2.3.5 Estruturas de subprogramas

Subprogramas sempre devem ser no escopo raiz do programa, evitando a possibilidade de se definir subprogramas aninhados.

Para processar essas definições, a *flag* de execução é desligada e o corpo do subprograma é *parseado* apenas sintaticamente.

Durante o *parse* exclusivamente sintático d corpo do subprograma (`<stmts>`), tem-se a preocupação em avaliar se existe um `return` em cada fluxo possível de execução. Para isso, cada sentença retorna um valor *booleano* que indica se esse comando está presente ali.

Ao ser invocado em tempo de execução, o subprograma finalmente é processado semanticamente (e novamente sintaticamente). É importante ressaltar que todo subprograma deve ter um comando de retorno em cada caminho possível em seu corpo, independente de ser função ou procedimento.

Temos a seguir o formato de uma definição e de uma chamada de uma função, respectivamente (`<a>` e `<b>` são outros elementos da sintaxe do programa com os quais não nos preocupamos neste momento).

```
func <ret> <name> (<args1>) {
  <stmts>
}

<a> <name>(<args2>) <b>
```

O primeiro passo é avaliar o valor de `<args2>`. Em seguida, o registro da função invocada é buscado na lista de subprogramas declarados. Então, um novo escopo é criado na lista de escopos e a função é adicionada a esse e também incrementa-se em seguida o contador de subprogramas ativos.

Em seguida, no escopo atual, os argumentos em `<args1>` são declarados e inicializados, tendo como atribuição seus respectivos valores em `<args2>`. Essa operação pode indicar erro quando a quantidade de argumentos em `<args1>` e `<args2>` diferem.

O corpo do subprograma (`<stmts>`), então, é processado semanticamente, executando seu corpo de comandos. Ao término do processamento semântico de `<stmt>`, ou seja, ao ser efetuado o comando `return`, o valor a ser retornado é armazenado em uma variável temporária em um escopo especial. A *flag* de execução é desligada. As sentenças restantes da função são *parseadas* apenas sintaticamente.

Ao fim de `stmts`, a *flag* de execução volta a ficar ligada e o valor retornado tem então seu tipo comparado com o tipo de retorno esperado, interrompendo o programa caso sejam diferentes. O escopo da função é retirado da pilha de escopos.

```
proc <name> (<args>) {
  <stmts>
}
```

O procedimento funciona de forma similar à da função, diferindo apenas em seu tipo de retorno, que é nulo.

2.3.6 Estruturas de registros

```
struct <name> {
```

```
  <declrs>
}
```

Assim como subprogramas, registros sempre devem ser definidos no escopo raiz do programa. Após a leitura de `<name>`, o nome do novo tipo é adicionado à tabela de tipos do estado da linguagem, tendo sua lista de campos inicialmente vazia. Ao entrar no corpo do registro, a *flag* de execução é desligada e `<declrs>` é processado sintaticamente. Então, a *flag* de execução é ligada novamente e a lista de campos do tipo é atualizada no estado. Essa preocupação em inserir um tipo ainda vazio deve-se ao suporte que *CBesta* tem a registros recursivos.

3 Instruções de uso

Para executar o analisador léxico:
`alex lexer.x`

Para compilar o analisador sintático:
`ghc parser.hs lexer.hs -i terminals/*.hs -i non_terminals/*.hs -i state/*.hs -i execution/*.hs`

Para executar o analisador:
`./parser.exe <nome_do_programa>`