Universidade Federal do Rio Grande do Norte

DIM0437

Linguagens de Programação: Conceitos e Paradigmas

OWLS

Discentes:	*Matrícula:*
Ana Caroline	2014028310
Gustavo Alves	2014026460
Luísa Rocha	2014042202
Vítor Godeiro	2014028197

14 de Junho de 2017

Conteúdo

1 Introdução

OWLS é uma linguagem desenvolvida seguindo o paradigma imperativo para o domínio de programação de sistemas focado em sistemas de grande porte que necessitem de uma garantia de confiabilidade maior.

A nossa linguagem desenvolvida deve ser de rápida execução porém tem de fornecer ao programador uma alta confiabilidade. Portanto, em uma situação ideal a nossa linguagem deveria ser híbrida, onde o interpretador é usado para depurar programas e, em seguida, após um estado (relativamente) livre de erros atingido, os programas são compilados para aumentar a velocidade de execução. Isso ajudaria aos programadores a encontrarem erros com maior clareza em seus códigos e manter a eficiência necessária para os sistemas do domínio proposto. Porém devido ao fator tempo a nossa linguagem é apenas interpretada, logo a mesma não é tão eficiente computacionalmente quanto desejávamos inicialmente.

2 Design da Implementação

O interpretador da linguagem foi separado em três principais módulos: Lexer, Parser e Interpreter.

O Lexer, em conjunto com o módulo Tokens, é responsável por ler o código fonte e transformá-lo para unidades léxicas (tokens), com auxílio da biblioteca Alex, do Haskell. No módulo Tokens são definidos cada um dos tipos de tokens, e então no Lexer são implementadas função de parsing para cada token individual.

A partir da lista de tokens retornada, o módulo Parser realiza a análise sintática, utilizando a biblioteca Parsec. O resultado dele é uma árvore sintática, descrita no módulo ProgramTree. Essa árvore irá conter cada atributo de cada sub-estrutura do código, como por exemplo: a declaração de uma função, os atributos de cada instrução (como a expressão booleana de um if, ou a variável a ser modificada por uma atribuição), etc.

O Interpreter, por fim, utiliza o resultado do Parser para então executar o código. Nessa etapa é que são feitas as checagens de erros de semântica estática.

3 Funcionalidades disponíveis na linguagem

Nossa linguagem tem os seguintes recursos disponíveis: condicional (if-else), laços (for e while), cálculo de expressões booleanas e numéricas. É possível realizar avaliação curto-circuito para expressões booleanas. Adicionalmente é possível declarar funções e procedimentos mesmo aninhados, subprogramas podem ser passados com argumentos. O usuário também pode criar seus próprios tipos e damos suporte a ponteiros.

Além disso, são oferecidas funções de leitura e escrita de dados para entrada e saída padrão. Há funções de leitura para naturais, inteiros e reais. Há também uma função que lê caracteres e retorna uma cadeia de caracteres. Note que as funções de leitura exigem que um valor seja inserido por linha.

A função de escrita recebe qualquer valor primitivo ou cadeira de caracteres e imprime.

Sintaxe resumida:

- Declaração de variável primitiva e tipo do usuário: `nome : tipo = valor;`
- Declaração de tipo: `struct nome {<declarações>}`

  As declarações são separadas por ponto e vírgula.
- Declaração de função: `func nome (<declarações>) : retorno {...}`

  As declarações são separadas por vírgula.
- Declaração de procedimento: `proc nome (<declarações>) {...}`
- Declaração de ponteiro: `nome : @tipo = $var;`
- Acesso a valor de ponteiro: `@nome`
- Função passada como argumento deve ser definido como a seguir: `nome : func (<tipos>)[retorno]`

  A lista deve ser separada por vírgulas e o retorno é o tipo de retorno que a função deve ter.
- Procedimento passado como argumento deve ser definido como a seguir: `nome : proc (<tipos>)`

  A lista deve ser separada por vírgulas. Por exemplo, `nome : proc (int, int, real)` .
- Funções de leitura: `read()`, `readReal()`, `readInt()`, `readNat()`

- Função de escrita: `write(argumento)`

  Esse argumento pode ser tanto caracteres (palavras) entre aspas duplas, como também expressões de tipo numérico, por exemplo.

3.1 Representação de símbolos, tabela de símbolos e funções associadas

No Lexer, cada símbolo é representado por um valor do tipo Token. Exemplos de Tokens seriam: `Func`, `Proc`, `Assign` (símbolo =), `Nat(n)` - onde n é um número -, entro vários outros que podem ser encontrados no arquivo "Tokens.x".

No Interpreter, a tabela de símbolos é representada dentro da estrutura OWLS-tate. O state armazena uma lista de tipos definidos pelo usuário e uma pilha de escopos. Cada escopo é descrito por uma tupla $< ID, PID, table >$, onde ID é o identificador único do escopo atual, PID é o escopo ancestral (que compartilha sua tabela de símbolos com os filhos), e *table* é a tabela de símbolos do escopo. A tabela de símbolos do escopo é uma lista com todas as variáveis declaradas no tal escopo e seu valor atual. Cada variável isolada é identificada pela chave (SID, N), em que SID é o ID do escopo dela e o N é o nome dela.

3.1.1 Funcionamento da criação e atualização de variáveis

Quando o Parser retorna a estrutura programa com nossa representação interna, inicia-se a execução do Interpreter, nesse momento é criado um estado inicial que não tem nenhuma declaração e possui apenas a lista com os tipos definidos pelo usuário.

O programa está dividido em instruções, que serão percorridas e quando necessário atualizamos o estado para que este reflita como está o programa no dado momento. Para que seja possível a manipulação do estado, temos algumas funções auxiliares importantes: updateVar e addVarDec. Elas são responsáveis, respectivamente, por atualizar o valor de uma variável e criar uma variável. Note que essas duas funções funcionam recebendo um estado, e retornando um novo estado com a tabela de símbolos alterada. A maior parte das funções relacionadas a manipulação do estado para adicionar variável, modificar variável, adicionar escopo, retirar escopo estão no módulo ProgramState.

3.2 Tratamento de estruturas condicionais e de repetição

Durante a fase de execução, cada instrução devolve um "resultado de instrução", que podem ser: `return`, `break`, `continue` ou `finish`. A primeira indica o comando de retorno, usado para tratar funções. A segunda é usada para tratar laços. A terceira é para indicar que a instrução seguinte pode ser executada. A última indica o final de um procedimento.

No caso de condicionais, a expressão de condição é avaliada e, caso seja verdadeira, o bloco de instruções dentro do if é executado. Caso contrário, pode ocorrer a execução do bloco else caso exista ou então a continuação no fluxo normal do código. Além disso, dentro do if é criado um novo escopo local e quando as instruções terminam esse escopo é desalocado.

Para estruturas de repetição, a expressão de condição é reavaliada ao fim da execução do bloco. Além disso, também é checado se o resultado da instrução é um `break`, para que a execução do bloco seja interrompida. Antes de iniciar a execução do corpo do loop, um novo escopo é criado. Esse escopo é destruído no final da iteração. O escopo será criado novamente caso a condição ainda seja avaliada como verdadeira. A principal diferença da implementação entre for e while é que o for força a execução da instrução de incrementação. Em relação ao escopo, o for irá criar um escopo extra (para a variável recém-declarada) antes de executar o corpo. Isso acontece para que, quando o escopo do corpo for deletado ao fim da iteração, a variável declarada não seja deletada também. Desse modo, evita-se erros de loop infinito ou mesmo de variável não declara/fora do escopo.

Em ambos os casos, os escopos criados são filhos do escopo atual, dessa forma é possível criar variáveis locais, mas também permanece o acesso às variáveis dos escopos ancestrais.

3.3 Tratamento de subprogramas

Dois tipos de sub-programas foram implementados: procedimentos e funções.

Os procedimentos são um tipo de instrução que cria um novo escopo cujo ancestral foi definido quando o procedimento foi declarado. O corpo do procedimento é então executado até o final ou até uma instrução de retorno seja encontrada. Ao final da execução do bloco, o fluxo volta para o bloco original que chamou o procedimento. As funções são implementadas de forma similar ao procedimento, com a diferença de que, caso chegue ao final do corpo sem uma instrução de retorno, é emitido um erro.

As funções são executadas durante a avaliação de expressões, visto que retornam um valor que deve ser usado. Caso o usuário tente chamar uma função sem atribuir seu valor a uma variável, será emitido um erro. Note que consideramos instruções (na execução do passo a passo do programa): condicional, laços, atribuições e a chamada de procedimentos. Por outro lado, a chamada de uma função está inserida na expressão que deve ser avaliada para uma atribuição ser feita.

3.4 Verificações realizadas

As verificações de tipos são sempre feitas ao executar operações e ao atribuir valores às variáveis. As únicas conversões automáticas feitas são as de `nat` para `int`, `nat` para `real` e `int` para `real`. As conversões entre outros tipos devem ser implementadas manualmente pelo usuário ou feitas através das funções `floor` e `ceil`, no caso de `real` para `int`.

Também é feita a verificação de declaração em duplicidade para variáveis com o mesmo nome no mesmo escopo. Caso o usuário tenha feito duas declarações de tipos com o mesmo nome, isso ocasiona um erro. Também ocorre erro quando no tipo declarado há dois campos com o mesmo nome.

No caso da verificação de intervalos, nossa linguagem está representando todos os valores numéricos como double em Haskell, logo os limites de valores são os mesmo do Haskell.

4 Instruções de uso do interpretador

É necessário ter instalado o compilador de Haskell e a biblioteca Parsec. Para compilar o projeto (dados os arquivos fornecidos pelo grupo), usar o `make` a partir da pasta raiz. Então, para executar um programa da linguagem, execute o arquivo `main` criado na pasta `bin` pelo terminal e passando como parâmetro o diretório do arquivo. Por exemplo:

```
./bin/main exercises/problema1.owls
```