RELATÓRIO DO TRABALHO 2
ANÁLISE SINTÁTICA E INTERPRETADOR

BCC328 - Construção de compiladores I

Autores:
21.1.4111 - Caio Silas de Araujo Amaro
21.1.4025 - Henrique Dantas Pighini
6° Período

28/01/2024

1 Introdução

Neste projeto, foi proposta a elaboração de um analisador sintático e do interpretador para a linguagem Lang, que foi concebida pelo professor Rodrigo Ribeiro.

O objetivo principal deste trabalho é desenvolver um interpretador capaz de executar uma variedade de programas simples escritos na linguagem Lang. Buscamos não apenas implementar um analisador sintático eficiente, mas também garantir que o interpretador seja capaz de compreender e executar corretamente os diferentes aspectos da linguagem.

2 Desenvolvimento

Começando o desenvolvimento do projeto, utilizamos um modelo de exemplo que foi fornecido utilizava monadas para registrar a posição dos elementos (Bytes, lin, colun).

Porém, quando fomos para a parte de construir o interpretador percebemos que sua complexidade iria ser um desafio muito grande no trabalho então decidimos utilizar um modelo mais básico e que conseguia fornecer também a posição dos tokens usando o wrapper do Alex.

Assim utilizando o modelo de exemplo da linguagem Imp que o professor Rodrigo disponibilizou nós continuamos o desenvolvimento do trabalho.

3 Árvore de Sintaxe

A estrutura dessa árvore de sintaxe representa a gramática de uma linguagem de programação fictícia. Vamos analisar as principais partes dessa estrutura:

3.1 Variáveis (Var) e Valores (Value):

Var: Representa uma variável na linguagem.
Value: Representa valores possíveis, como inteiros, floats, booleans, caracteres ou Null.

3.2 Expressões (Exp):

EValue: Representa um valor.
EVar: Representa uma variável.
Operações aritméticas: Add, Mul, Sub, Div, Mod.
Operações lógicas e relacionais: Conj, Not, Equal, NEqual, Relac.
Negate: Representa a negação de uma expressão.
ExpFunc: Representa a chamada de uma função.
New: Representa a criação de um novo objeto.

3.3 LValues (LValue):

LValue: Representa uma variável.
VLValue: Representa um índice ou posição em uma lista.
AccReg: Representa o acesso a um registro.

3.4 Tipos (Ty):

Tipos básicos: TSInt, TSFloat, TSBool, TSChar, TSId. Listas de tipos: TypeList. Vetor de tipos: TSVector.

3.5 Programa (Program), Estrutura (Structure), Bloco (Block), Registro (Reg), Parâmetro (Param) e Declarações (Stmt):

Program: Representa o programa como um todo, contendo um bloco.
Structure: Representa estruturas de dados ou funções.
Block: Representa um bloco de código, que pode conter uma lista de declarações (Stmt).
Reg: Representa um registro.
Param: Representa um parâmetro de função.
Stmt: Representa declarações ou comandos no programa, como atribuição, estruturas condicionais (If, IfElse), impressão (Print), leitura (CRead), iteração (Iterate), chamada de função (CallFunc), retorno (Return), declaração de registro (DeclReg) e definição de variáveis (Def).

4 Analisador Léxico

Durante o desenvolvimento, o analisador léxico passou por modificações para aceitar números decimais (**float**) e caracteres (**char**). Houve uma alteração no tipo do wrapper para " posn" com o objetivo de possibilitar o retorno da linha em que ocorreu o erro.

Foram implementadas funções específicas para identificar números e variáveis. Além disso, foi criado um tokenizador que separa o token de sua criação, melhorando a estrutura e organização do analisador léxico.

Essas melhorias proporcionam uma maior capacidade de lidar com diferentes tipos de dados, tornando o analisador léxico mais robusto e capaz de identificar erros de forma mais precisa. A introdução de funções especializadas para a identificação de números e variáveis contribui para uma análise mais eficiente e específica dos elementos no código-fonte.

5 Analisador sintatico:

Durante o desenvolvimento do trabalho prático de computação, realizamos a definição dos tokens recebidos do Analisador Léxico (Alex). Em seguida, estabelecemos a precedência de operadores, contribuindo para a correta interpretação das expressões no contexto da linguagem em questão.

Além disso, procedemos à definição das regras da gramática, delineando as estruturas sintáticas aceitas pelo analisador. Este processo foi crucial para a construção de um parser eficiente capaz de analisar corretamente o código fonte.

Ao longo do desenvolvimento, enfrentamos desafios relacionados à manipulação de estruturas condicionais do tipo " if-else". Devido a problemas específicos de shift-reduce, foram necessárias alterações no fluxo dessas estruturas. Para superar essas dificuldades, implementamos uma solução que envolveu forçar um shift em operações específicas, permitindo assim a resolução adequada dessas condições.

Essas modificações foram essenciais para garantir a coerência e a precisão na análise sintática do código fonte. O aprimoramento na manipulação de estruturas condicionais contribuiu significativamente para a robustez e eficácia do analisador sintático desenvolvido.

6 Interpretador:

Durante a elaboração do trabalho prático, utilizamos como ponto de partida o interpretador apresentado em sala para a linguagem IMP, fornecido pelo professor. No entanto, encontramos desafios significativos durante a definição das regras de gramática, o que demandou mais tempo do que inicialmente previíamos.

Inicialmente, tentamos seguir a abordagem utilizando monadas, mas percebemos que a compreensão e implementação estavam se tornando excessivamente complexas. Diante dessa dificuldade, optamos por uma abordagem mais simplificada, inspirada no exemplo fornecido para a linguagem IMP. Essa mudança de estratégia revelou-se mais acessível e facilitou a implementação do interpretador.

Além disso, enfrentamos obstáculos na definição da sintaxe das árvores, que se tornou a parte mais desafiadora do trabalho. Foi necessário refazê-la diversas vezes, incorporando novos tipos à medida que progredíamos. A introdução de funções adicionou mais camada de complexidade, exigindo uma cuidadosa reflexão sobre como seria feita a chamada dos blocos de código.

As lições aprendidas ao longo dessas dificuldades foram valiosas para o aprimoramento do nosso entendimento sobre a construção de interpretadores. A flexibilidade e adaptabilidade tornaram-se fundamentais, evidenciando a importância de ajustar abordagens conforme a complexidade do problema enfrentado.

No enunciado do trabalho, foi solicitada a inclusão de novos tipos de dados, como float e char, na linguagem. Para incorporar o tipo float, além de definir seus tokens nas partes léxica e sintática, foi necessário realizar alterações nas funções do interpretador. Isso incluiu a adição de casos base para sobrecarga de funções e para as funções de leitura (read) e impressão (print) que agora suportam o novo tipo float.

7 Seções não implementadas

Nosso trabalho reproduz a maioria das sintaxes da linguagem Lang. Porém, não conseguimos implementar alguns tópicos como Registro, Arranjo e chamadas de função.

Registro seriam necessários para definir um novo tipo de dados. Funções precisam de Arranjos para retornar valores. Arranjos utilizariam os anteriores para definir vetores de tipos diferentes e para definir os vetores de retorno das funções.

No nosso desenvolvimento começamos tentando definir os Arranjos, e definimos os tokens e sua gramática sintática. Entretanto, chegamos na parte sintática e não conseguimos resolver os problemas gerados à tempo, então decidimos não manter essas alterações no código.

8 Exemplos implementados

Com o código que implementamos criamos 3 códigos para testes.

Foi implementado um código ops.lang que testa todas as operações que podem ser realizados com inteiros e floats.

```
main(){
Int n = 0;
read n ;

print n + 1.1;
print n * 1.1;
print n - 1.1;
print n / 2;
print n % 2;
print -n;
}
```

Foi implementado um código factorial.lang que realiza o calculo de fatorial sendo passado um valor n inteiro lido pelo teclado.

```
main() {
  Int n = 0;
  read n ;
  Int fact = 1;
  iterate (0 < n) {
    fact = fact * n ;
    n = n - 1 ;
  }
  print fact ;
}
```

Foi implementado um código fib.lang que calcula o n número da sequencia Fibonacci sendo n um inteiro lido pelo teclado.

```
main(){
    Int n = 0;
    read n;
    n = n + 1;

    Int a = 0;
    Int b = 1;
    Int i = 2;
    Int nextTerm = 0;

    iterate(i < n){
        nextTerm = a + b;
        a = b;
        b = nextTerm;

        i = i + 1;
    }

    print nextTerm;
}
```

9 Diretivas de compilação

A partir do diretório base do GitHub, é necessário entrar na pasta " tp2" para executar o código. Para compilar o código, é preciso utilizar o Cabal. Na construção do código, foram empregadas a versão,Utilizamos as ferramentas Alex e Happy para análise léxica e sintática:
3.0 do Cabal
4.18.0.0 do GHCi.
1.20.1.1 Happy
3.4.0.0 Alex

Para compilar o código, execute o comando " cabal new-build". Já para executar o código, utilize o comando " cabal new-run tp2 – ./test/nomedoarquivo.imp". Na pasta " test", encontram-se diversos casos de teste para a linguagem. Basta selecionar um deles para realizar os testes.

10 Dados de Entrada

Para a leitura da entrada utilizamos parte do código fornecido pelo professor em atividades passadas, de modo a conseguir utilizar comandos no terminal.

10.1 Função Principal (main :: IO ())

```
main :: IO ()
main = do
  args <- getArgs
  runInterpreter args
```

A função main é a principal função do programa. Aqui estão as etapas executadas:

args ¡- getArgs: Obtém os argumentos da linha de comando e armazena na variável args. runInterpreter args: Chama a função runInterpreter passando os argumentos obtidos. Essa abordagem permite que o programa seja executado a partir da linha de comando, recebendo argumentos que podem incluir o nome de um arquivo contendo código-fonte a ser interpretado.

10.2 Função de Mensagem de Erro (printErrorMessage :: IO ())

```
printErrorMessage :: IO ()
printErrorMessage
  = do
       putStrLn "Usage"
       putStrLn "tp2 <file>"
```

A função printErrorMessage imprime mensagens de erro indicando o uso correto do programa. Ela exibe as seguintes mensagens:

"Usage" "tp2 ¡file¿" Essas mensagens orientam o usuário sobre como utilizar o programa, indicando que um arquivo deve ser fornecido como argumento na linha de comando.

10.3 Função de Execução do Interpretador (runInterpreter::[String]-IO ())

```
runInterpreter :: [String] -> IO ()
runInterpreter [arg1]
  = do
      content <- readFile arg1
      let ast = langParser content
      _ <- interpProgram ast
      return ()
runInterpreter _
  = printErrorMessage
```

A função runInterpreter é responsável por interpretar o código-fonte fornecido como argumento da linha de comando. Ela segue os seguintes passos:

Caso com um argumento:

content ¡- readFile arg1: Lê o conteúdo do arquivo especificado pelo primeiro argumento e armazena em content. let ast = langParser content: Utiliza o parser (langParser) para gerar a Árvore Sintática Abstrata (AST) a partir do conteúdo do arquivo. ¡- interpProgram ast: Chama a função interpProgram para interpretar e executar o programa representado

pela AST. return (): Retorna vazio, indicando que a execução foi concluída com sucesso. Caso sem argumento ou com mais de um argumento:

printErrorMessage: Chama a função printErrorMessage para informar ao usuário sobre o uso correto do programa.