Disciplina Compiladores - DCOMP/UFS
2022.2
Profa. Beatriz Trinchão Andrade

Especificação da Linguagem Elipses

1. Introdução

Indo contra o padrão de arquitetura de computadores e idiomático vigentes na computação, a linguagem **Elipses** é uma linguagem funcional e em português que contém características – muito simplificadas e aleatórias - de ML, Scheme, LISP e Haskell.

Ela apresenta as seguintes características principais:

Tipos primitivos: inteiro, real, booleano;
Tipos compostos: não possui;
Estruturas de controle: se-senao ternário, recursão;
Escopo das constantes: escopo do bloco função;
Funções: há suporte a funções padrão, lambda e de alta ordem;
Comentários: bloco e linha.

Elipses, obviamente, é uma linguagem experimental, então esta especificação é passível de adaptações. Em caso de modificações na especificação, versões atualizadas serão postadas via SIGAA (no tópico de aula "Definições do Projeto") e notificações serão enviadas aos alunos.

As Seções 2, 3, 4 e 5 deste documento apresentam a especificação da linguagem. A Seção 6 contém informações sobre a avaliação e entregas.

2. Léxico

Regras para identificadores:

- Identificadores podem conter um ou mais letras ou underlines, em qualquer ordem.
- Não são permitidos espaços em branco, acentos e caracteres especiais (ex.: á, @, $, ç, +, -, ^, % etc.).
- Identificadores não podem ser iguais às palavras reservadas ou operadores da linguagem.

Tipos primitivos:

- A linguagem aceita os tipos inteiro, real e booleano.
- Números inteiros podem ser escritos em decimal ou em binário (começando com 0b).
- A parte decimal dos números reais deve ser separada por uma vírgula.
- Booleanos podem assumir dois valores: `verdade` e `falso`.

Tipos compostos:

- A linguagem não possui.

Blocos:

- Blocos são delimitados por parênteses.

Comentários:

- A linguagem aceita comentários de linha, indicados pelo símbolo # no início da linha.
- A linguagem aceita comentários de bloco (possivelmente de múltiplas linhas) delimitados por {-- e --}.
- O funcionamento dos comentários de bloco em **Elipses** são similares aos da linguagem C. Exemplo: o que acontece se você compilar /* */ */ em C? E isso: /* /* */? Estudem como um compilador C reconhece o fim de um comentário de bloco.

Estruturas de controle:

- laço: recursão
- condição: se/senão ternário

Operadores:

- Operadores aritméticos: +, -, *, /, %
- Operadores relacionais: >, <, =
- Operadores booleanos: ‘`e`’ ‘`nao`’ ‘`ou`’.
- Operador ternário `se`. Este operador só existe quando é usado como expressão.
- A precedência dos operadores é igual à de C, e pode ser alterada com o uso de parênteses.

Subrotinas:

- Funções, sempre com retorno de valor

Vide Seção 3 para mais informações sobre a sintaxe dos comandos e sobre as classes de tokens a serem implementadas.

3. Sintático

A gramática abaixo foi escrita em uma versão de E-BNF seguindo as seguintes convenções:
1 - variáveis da gramática são escritas em letras minúsculas sem aspas.
2 – lexemas que correspondem diretamente a tokens são escritos entre aspas simples
3 - símbolos escritos em letras maiúsculas representam o lexema de um token do tipo especificado.
4 - o símbolo | indica produções diferentes de uma mesma variável.
5 - o operador [ ] indica uma estrutura sintática opcional (similar à ?).
6 - o operador { } indica uma estrutura sintática que é repetida zero ou mais vezes (fecho).

```
programa :  { dec_funcao }

dec_funcao : ['entrada'] '(' tipo ID '(' parametros ')' ':'
     '(' exp ')' ')'

tipo : 'inteiro' | 'booleano' | 'real'

parametros : Ɛ | parametro { '|' parametro }
parametro : tipo ID | assinatura
assinatura : tipo ID '(' parametros_assinatura ')'

//os parâmetros na assinatura de uma função passada como parametro não têm
//identificador
parametros_assinatura : Ɛ | parametro_assinatura{ '|' parametro_assinatura}
parametro_assinatura : tipo | assinatura

exp : NUMERAL_INT | NUMERAL_REAL | 'verdadeiro' | 'falso'
     | bloco_exp
     | chamada
     | lambda
     | '-' exp
     | 'se' '(' exp ')' 'entao' exp 'senao' exp
     | exp '+' exp
     | exp '-' exp
     | exp '*' exp
     | exp '/' exp
     | exp '%' exp
     | exp '=' exp
     | exp '<' exp
     | exp '>' exp
     | 'nao' exp
     | exp 'e' exp
     | exp 'ou' exp
bloco_exp : '(' { dec_cons } exp ')'
dec_cons : '(' 'const' tipo ID '(' exp ')' ')'

chamada : ID '(' lista_exp ')'
lambda :  '(''lambda' '(' lista_ids ')' ':' '('exp')' '[' lista_exp ']'')'
lista_ids : Ɛ | ID { '|' ID }
lista_exp : Ɛ | exp { '|' exp }
```

4. Semântico:

- A execução de um programa consiste na execução de uma função com modificador “entrada”. A inexistência de um ponto de entrada não impede a compilação, mas gera um *warning*.

- A precedência dos operadores é a matemática.

- Em operações entre os tipos inteiro e real, os valores inteiros devem ser convertidos para real.

- É possível criar funções lambda com tipo implícito. Exemplo:

```
lambda (x | y) : (x*y) [2 | 7]   #  expressão que vale 14
```

Obs. O tempo de vida do identificador é o mesmo da função lambda.

- Funções de alta ordem em Elipses: uma função pode receber uma ou mais funções como parâmetros. Exemplo:

```
( inteiro max (inteiro x | inteiro y) :
       (se (x>y) entao x senao y)
)

( inteiro aplica_operador (inteiro x | inteiro y | inteiro op(inteiro | inteiro) ) :
       (op(x, y) # função de alta ordem
)

entrada ( inteiro principal () :
       (aplica_operador(7 | 5 | max) )
```

O arquivo exemplo.elip (anexo à aula do SIGAA) contém este exemplo como um arquivo com formato `.`elip.

O que deve ser verificado na análise semântica:

- Se as entidades criadas pelo usuário são inseridas na tabela de símbolos - com os atributos necessários - quando são declaradas;
- Se uma entidade foi declarada e está em um escopo válido no momento em que ela é utilizada;
- Se entidades foram definidas (inicializadas) quando isso se fizer necessário;
- Checar a compatibilidade dos tipos de dados envolvidos nas **declarações** e **expressões.**

6. Desenvolvimento do Trabalho

Trabalhos devem ser desenvolvidos em quartetos (preferencialmente), trios, duplas ou individualmente. Os grupos devem se cadastrar na planilha correspondente à turma dos seus componentes:

TURMA 01:
https://docs.google.com/spreadsheets/d/11SvjM7_qMUxn_i7zOaEYglH33h7wZ4APgb1cNshtnXQ/edit?usp=sharing

Serão abertos fóruns no SIGAA para a discussão sobre as etapas. Em caso de dúvida, verifique inicialmente no fórum se ela já foi resolvida. Se ela persiste, consulte a professora.

6.1. Ferramentas

- Implementação com SableCC, linguagem Java.
- IDE Java (recomendação: Eclipse).
- Submissão das etapas do projeto via SIGAA. Haverá uma tarefa para cada etapa.

6.2. Avaliação

- A avaliação será feita com base nas etapas entregues e em entrevistas feitas com os grupos durante as aulas de acompanhamento do projeto.
- O valor de cada etapa está definido no plano de curso da disciplina.
- O cumprimento das requisições de formato também será avaliado na nota de cada etapa.

6.3. Etapas

Etapa 1. Códigos Elipses
- **Prazo: 01/02/2023**
- **Atividade:** escrever três códigos em **Elipses** que, unidos, usem todas as alternativas gramaticais (ou seja, todos os recursos) da linguagem.
- **Formato de entrega:** arquivo comprimido contendo três códigos, onde cada código deve estar escrito em um arquivo de texto simples, com extensão "**.elip**".

Etapa 2. Análise Léxica
- **Prazo: 13/02/2023**
- **Atividade:** implementar analisador léxico da linguagem com o SableCC, fazendo a impressão dos lexemas e tokens reconhecidos ou imprimindo erro quando o token não for reconhecido.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos).
  O nome do pacote a ser gerado pelo sablecc deve se chamar **Elipses** (em letras minúsculas).

Etapa 3. Análise Sintática
- **Prazo: 03/04/2023**
- **Atividade:** implementar analisador sintático da linguagem com o SableCC, fazendo impressão da árvore sintática em caso de sucesso ou impressão dos erros caso contrário.

- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos).

O nome do pacote a ser gerado pelo sablecc deve se chamar **Elipses** (em letras minúsculas).

Etapa 4. Análise Sintática Abstrata

- **Prazo: 17/04/2023**
- **Atividade:** implementar analisador sintático abstrato em SableCC, com impressão da árvore sintática.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos).
  O nome do pacote a ser gerado pelo sablecc deve se chamar **Elipses** (em letras minúsculas).

Etapa 5. Tabela de Símbolos e Análise Semântica

- **Prazo: 06/05/2023**
- **Atividade:** implementação da tabela de símbolos, validar escopo, declaração e definição de identificadores. Implementar verificação de tipos.
- **Formato de entrega:** projeto completo, incluindo obrigatoriamente: o arquivo .sable; todas as classes java escritas pelo grupo ou geradas automaticamente; e arquivos .**elip** que demonstrem o que foi feito nesta tarefa.
  Também é obrigatória a entrega de um pdf contendo uma breve explicação sobre o que foi implementado nesta etapa e como.

Etapa 6. Geração de código (extra: 2 pontos na segunda unidade):

- **Prazo: o mesmo da Etapa 5**
- Compilação de código Elipses com geração de código em linguagem alvo (C ou Haskell)

Entregas após o prazo sofrem penalidade de metade da nota da etapa por dia de atraso.

- Trabalhos entregues com atraso devem ser submetidos na Tarefa ‘Entrega após prazo’, no SIGAA, que ficará aberta durante todo o período. Arquivos enviados por e-mail não serão considerados.

Bom trabalho!