Mandatory Assignment 2
Context Free Grammars

Fariha Nazmul (s094747)

Presentation # 52

Collaborators:

Phan Anh Dung (s094745)

Sameer K. C. (s094746)

Q1 CFG for WHILE language

*Program -> Inp Stmt ';' Outp
*Inp -> 'input' '(' Ident ')' ';' *Outp -> 'output' '(' Ident ')'
*Stmt -> 'skip' | Assign | Sequence | If | While
*Assign -> Var ':=' Arith *Sequence -> '(' Stmt ';' Stmt ')'
*If -> 'if' Bool 'then' Stmt 'else' Stmt 'fi
*While -> 'while' Bool 'do' Stmt 'od'
*Arith -> Term | '(' Arith $Arith_1$
$Arith_1$ -> '+' Arith ')' | '-' Arith ')' | '' Arith ')'
*Bool -> 'true' | 'false' | '(' $Bool_2$
*$Bool_2$ -> Bool $Bool_1$ | '!' Bool ')' | Term $Term_1$
*$Bool_1$ -> '&&' Bool ')' | '||' Bool ')'
*$Term_1$ -> '=' Term ')' | '>' Term ')' | '<' Term ')'
*Ident -> Lower $Ident_1$
*$Ident_1$ -> Lower $Ident_1$ | Upper $Ident_1$ | Num $Ident_1$ | ϵ
*Var -> Ident *Term -> Num | Ident
*Lower -> 'a'| 'b'| 'c'| 'd'| … | 'z' *Upper -> 'A'| 'B'| 'C'| …. | 'Z'
*NonNull -> '1'|'2'|'3'|…|'9'
*Num -> '0' | NonNull Num_1 * Num_1 -> '0' Num_1 | NonNull Num_1 | ϵ

(Here all terminals are indicated in single quotes)

Q2b Constraint of arguments in comparison

The rule for admissible arithmetic expressions in comparisons was changed from general arithmetic expressions.

The original rule that did not constraint the kind of arguments that could occur in a comparison would have allowed **Arith** in **Bool**. In that case both the rules **Bool** and **Arith** would have began with **‘(‘ Arith** . This will create ambiguity in the grammar.

It also will create a non LL recursion which in turn will create problem in LL parsing.

Q3 Parser for WHILE program

The context-free grammar that is described in the answer of Q1 is used in the ANTLR tool. The grammar is defined in the file while.g using the syntax of ANTLR.

Mainly, I had to define the rules for Statement and write codes for each rules. Each rule in the grammar will create the instance of appropriate Intermediate Representation objects with appropriate arguments.

Building the while.g file in the ANTLR tool creates the lexer and parser code for the WHILE program.

Q4 Implementation of function $\mathcal{A}$

A new class file named “DerivedFunctions” is created in the package “ConFreeG.Functions”. In this class a method is created with the name “functionA( )” that takes an arithmetic expression as argument.

- This function is defined recursively using the given definition.
- It checks the instance type of the input argument and recursively calls the function itself with appropriate argument.
- The function only returns the string “?v” where v is the name of a variable or returns empty string when the input is a Num.

Since I have to change the string representation to regular expression later, I’ve made the function to return the regular expression of type RegExp using the package of assignment 1.

Q5 Recursive definition of function $\mathcal{B}$

$\mathcal{B}$(True) = ϵ

$\mathcal{B}$(False) = ϵ

$\mathcal{B}$(V) = ?v

$\mathcal{B}$(N) = ϵ

$\mathcal{B}$(!B) = $\mathcal{B}$(B)

$\mathcal{B}$(A1 = A2) = ($\mathcal{B}$(A1) $\mathcal{B}$(A2))

$\mathcal{B}$(A1 > A2) = ($\mathcal{B}$(A1) $\mathcal{B}$(A2))

$\mathcal{B}$(A1 < A2) = ($\mathcal{B}$(A1) $\mathcal{B}$(A2))

$\mathcal{B}$(B1 && B2) = ($\mathcal{B}$(B1) $\mathcal{B}$(B2))

$\mathcal{B}$(B1 || B2) = ($\mathcal{B}$(B1) $\mathcal{B}$(B2))

- The function $\mathcal{B}$ is defined recursively similar to function $\mathcal{A}$.
- It recursively calls the function itself whenever its operands are also **Bool**.
- The function calls function $\mathcal{A}$ whenever its operands are **Arith.**
- The function only returns the string "?v" where v is the name of the variable or returns empty string when the input is a **Num**, **True** or **False**.
- From the definition, we know that the result of function $\mathcal{A}$ is a regular expression over the alphabet △' := {?v |v ϵ Var}. Function $\mathcal{B}$ also acts like funtion $\mathcal{A}$ and it is defined using itself and funtion $\mathcal{A}$ only. So, the result of function $\mathcal{B}$ is always a regular expression over the alphabet △ := {?v |v ϵ Var} ∪ {!v |v ϵ Var}.

Q6,7 Implementation of function $\mathcal{B}, \mathcal{S}, \mathcal{P}$

The functions $\mathcal{B}$, $\mathcal{S}$, $\mathcal{P}$ are defined in the "DerivedFunctions" class file as "functionB( )", "functionS( )" and "functionP( )" respectively and as argument they take a boolean expression, a statement and a program respectively.

- These functions are defined recursively using their definition.
- Each function checks the instance type of the input argument and recursively calls the function itself or any of the these four functions $\mathcal{A}, \mathcal{B}, \mathcal{S}, \mathcal{P}$ with appropriate argument.

Since I have to change the string representation to regular expression later, I've made the functions to return the regular expression of type RegExp using the package of assignment 1.

Q8 Implementation of function h_v

- The function h_v is also defined in the "DerivedFunctions" class file as "functionHomomorphism( ) according to the given definition.

- It takes a RegExp ***exp*** and a variable name v as input and returns $h_v(exp)$ which is a regular expression over the alphabet set $\Sigma = \{ r, w, n\}$.

Q9

- The decision algorithm to decide the following problem

  "Given a program P; does $\exists v \in$ **Var** :

  $h_v(\mathcal{P}(P)) \not\subseteq n^*(w(r+w+n)^* + \epsilon)$ hold? "

  is already partially implemented in assignment 1.
- In assignment1, we can determine the property that a given regular expression **rexp** $\subseteq n^*(w(r+w+n)^* + \epsilon)$ using the Prop.check( ) function.

- The given problem can be solved in the following steps:
  - Parse the given program P
  - Input the parsed result to function $\mathcal{P}$
  - Create the list of variables **Var** of the program P using the function functionVariableList( ).
  - For each variable $v \in$ **Var**
    - call the function $h_v(\mathcal{P}(P), v)$ and get the regular expression **rexp** for that variable.
    - Call the funciton Prop.check( ) with **rexp**
    - Negate the result