utbm
université de technologie
Belfort-Montbéliard

DA53: Compilation and Language Theory - Final Exam A2024

Duration: 2h00. No document, no electronic device allowed.
English recommended, French accepted.

Part 1: Introduction (5 points)

Question 1.1:

Consider the language Bash that is usually used in Unix/Linux shells. Is this programming language type-safe? Explain why.

Question 1.2:

What is a Syntax-Directed Definition (SDD)? Explain briefly the concept of SDD and it content.

Question 1.3:

How the runtime environment is implementing and managing in the virtual machine a function call that is usually defined in a high level language? Explain briefly the applied method and the data structure involved.

Part 2: Tiny-PROLOG (10 points)

In this part, you must consider a simplified version of the logic-programming language PROLOG, named Tiny-PROLOG, that is explained below. In Tiny-PROLOG, a program is a set of logic predicates. Each predicate is defined as:

`<head> :- <body>.`

where:

- `<head>` is true if `<body>` is true,
- `<head>` is defined as `<name>(<var`$_1$`>, ..., <var`$_n$`>)`, where `var`$_n$ represents a variable name. The list of variables is optional and the parentheses could be removed if there is not variable.

- <body> is a sequence of calls to other PROLOG predicates (or heads), including their names and arguments, e.g., `N(X, Z)`.
- The predicates in the <body> are separated by:
  - the character "`,`" for representing a conjunction (AND logic),
  - the character "`;`" for representing a disjunction (OR logic).
- The predicate "`\+`" is predefined and represents the negation operator, e.g., `\+X` is true if `X` is false.
- The predicate "<name> `is` <expression>" is predefined and corresponds to the computation of the given <expression> and the assignment of the result to the variable with the <name>.
- <expression> is an arithmetic expression that may contain:
  - Constant value are numbers (integer or floating points),
  - Binary arithmetic addition, with operator +,
  - Binary arithmetic multiplication, with operator *.

Reminder:

You may answer on your paper sheet to the three following questions in a single Syntax Directed Definition (SDD) that is containing both the grammar rules in Backus-Naur Form (BNF) and the SDD semantic rules.

Question 2.1:

Write the grammar rules (using the Backus Naur Form - BNF) for recognizing the nonterminal <expression> for the Tiny-PROLOG language. Your grammar will be analyzed with LL(0) approach. So that, you should avoid left-recursions and remove ambiguities from the rules.

Question 2.2:

Extend the previous BNF grammar for recognizing the total Tiny-PROLOG syntax with all the constructs that are explained above.

Question 2.3:

Write the semantic rules of the Syntax Direction Definition (SDD) to interprete the code and compute the boolean values (true or false) for each nonterminal of the grammar. The goal of the SDD rules is to execute the Tiny-PROLOG operational semantic and reply the computed boolean value. To do so, you must follow the points:

1. List the attributes for each nonterminal. Are they synthesized or inherited?
2. Write the semantic rules for each nonterminal that is computing the values of the attributes defined in the first point.

Part 3: Code Generation (5 points)

In this part, you must focus on the generation of the form of three-address code that is named “quadruple.” The language to translate is the one described in the part 2.

Reminder:

In the generated three-address code, you may not define the addresses themselves, but use symbolic names in place.

Reminder:

You could define you own set of three-address instructions, in addition to those that are explained in the DA53 slides, that are required to defined your SDD in this part.

Question 3.1:

In this question, you must assume that you **don't have a syntax tree** generated from the parsing stage. The generation of the quadruples must be directly put in the semantic rules of the SDD.

1. Define the additional attributes that are required to generate the "quadruples".
2. Write the SDD to generate the "quadruples" for the nonterminal `<expression>` and its accessible nonterminals in the BNF defined in Part 2. Optimization of quadruple code is not mandatory.