G52CMP, Compilers
Assessed Coursework, 2002/2003

School of Computer Science and IT
University of Nottingham

January 23, 2003

Abstract

This document details the assessed coursework for the module G52CMP in the academic year 2002/2003. The goal of the coursework is to extend a supplied partial implementation of a contextual analyzer for the MiniTriangle language in such a way that conventional precedence is given to all operators, and exponentiation (taking powers) is added. The coursework is split into two parts: the first part aims to familiarise you with the supplied code, whilst the second part is the actual implementation of the extension.

1 Introduction

Watt and Brown's textbook, Programming Language Processors in Java, uses the Mini-Triangle language to illustrate various aspects of compiler construction. This coursework is about the initial stages (syntax analysis and contextual analysis) in the construction of a compiler for the language. The coursework will give you experience of working with a medium-sized program, designed and written by someone other than yourself. You will also get in-depth knowledge of "abstract syntax trees" and the use of the "visitor pattern", one of the so-called "design patterns" used for the development of object-oriented programs. The coursework will be supported by laboratory sessions to be held between 9am and 11am on 14th February, 21st February, 7th March, 14th March and 21st March.

It is possible to complete this coursework in a group of at most 3 members. See section 6 for further details.

Files mentioned below in `teletype` font can be obtained from the web site for this coursework:

`http://www.cs.nott.ac.uk/~pni/G52CMP`

2 Part I: Familiarisation

Although there is no separate deadline for this part of the coursework, you should aim to complete it by the end of February. Only in this way will you obtain full benefit from attending the lectures.

2.1 Supplied Code

From the web site for this module you can download the file `MiniTriangle.zip`. This contains an implementation of a lexical analyzer, a syntax analyzer and a contextual analyzer for the MiniTriangle language.

Task I.0: Download the file. Study the directory structure and compile the main program.

2.2 Lexical Analysis

The lexical analysis phase of the compiler converts sequences of characters into tokens. In the supplied code, the relevant classes and methods are `SyntacticAnalyzer/Scanner` and `SyntacticAnalyzer/Token`. You should read these files to make sure you know what they do.

Task I.1: In the file `Token.java`, a table of strings, called `tokenTable`, is defined and initialised. This table contains the spellings for most token classes defined in the lines above it.

1. What is the role of this table, and how does it relate to the token classes?
2. Why are spellings for the token classes `INTLITERAL`, `IDENTIFIER` and `OPERATOR` not included?
3. The spellings for the token classes `Token.BECOMES` and `Token.COMMA` are also not in the table (positions 15 and 16). In what way does this affect the scanner? Does it affect the parser? (Hint: run the MiniTriangle compiler on the `CourseworkPart1` test file after replacing the occurrences of `~` and `:=` by the symbol `=`. Do the error messages make sense?)

2.3 Syntax Analysis

Figure 1 specifies the syntax of MiniTriangle using BNF. The specification uses "::=" to separate the left and right sides of each production and "|" to separate alternative right sides. The grammar does not specify the syntax of identifiers or integer literals because this part of the syntax is handled by the lexical analyser.

Figure 3 specifies the syntax of MiniTriangle using extended BNF. A superscript "$*$" denotes iteration an arbitrary number of times (including zero). Bold parentheses "**(**" and "**)**" are *meta*symbols used for grouping; non-bold parentheses "(" and ")" are symbols of the *object* language, MiniTriangle. It is this syntax that has been used in the supplied implementation of a syntax analyser for the language.

Task I.2: Below is an example of a MiniTriangle program. Draw a derivation tree for this program based on the syntax specified in fig. 1. As a deterrent against plagiarism, your tree should be drawn by hand. **Computer-drawn solutions are not allowed.** What would be the output from this program if it were executed? What would be the output if the parentheses on the righthand side of the assignment were omitted? (That is, if the assignment were `n := m - 2 * m + 1`.) *Justify your answers.*

```
! This is a comment.  It continues to the end of the line.
let
   const m ~ 7;
   var n: Integer
in
   begin
   n := m - 2 * ( m + 1);
   putint(n)
   end
```

2.4 Abstract Syntax Trees

Figure 2 describes the abstract syntax corresponding to the extended BNF description of MiniTriangle. The abstract-syntax-tree class and all its subclasses can be found in the directory `AbstractSyntaxTrees` of the supplied code. The `Parser` method (in the directory `SyntacticAnalyzer`) constructs an abstract syntax tree representing a MiniTriangle program as it parses the program.

Task I.3: Draw an abstract syntax tree for the above MiniTriangle program. Indicate on your diagram in which order the nodes of the tree are added. (You can do this by tracing the execution of the program, recording the order in which ***new*** is executed to create a new instance of an abstract syntax tree.) As a deterrent against plagiarism, your tree should be drawn by hand. **Computer-drawn solutions are not allowed.** Explain briefly how the nodes are created, and why the order of creation is as it is.

3 Modifying the Supplied Code

You should begin this part of the coursework by the beginning of March.

3.1 Parser

As explained in the lectures, the grammar does not distinguish between any of the operators; there is thus no precedence between operators.

Task II.1: Modify the grammar so that operators are divided into four categories, *UnaryOp* denoting unary operators (`\`, denoting logical negation, is the only unary operator in MiniTriangle), *MulOp* denoting (binary) arithmetic multiplication operators (`*` and `/`), *AddOp* denoting (binary) arithmetic addition operators (+ and `-`) and *RelOp* denoting (binary) relational operators (`=`, `<` and `>`). The grammar should give precedence to the operators in the order given (that is, unary operators first and relational operators last). Add an arithmetic power operator, denoted by the symbol `^` (so that, for example, `m^2` denotes m^2). This operator should have precedence higher than the multiplication operators, but lower than the unary operators. The exponent may be an arbitrary arithmetic expression.

Task II.2: Modify the supplied parser (the `Parser` method, to be found in the directory `SyntacticAnalyzer`) so that it parses expressions according to the syntax you have specified as your solution to task II.1.

To complete this task, you should execute the supplied code with the option "`-nocheck`" set. This will turn off contextual analysis. You will need to execute the code normally to complete the remainder of the coursework.

3.2 Checker

The `Checker` method in the directory `ContextualAnalyzer` is a partial implementation of a contextual analysis of MiniTriangle programs. It creates a standard environment containing declarations of identifiers like "*false*" and "*putint*", it checks that all identifiers are declared before their use, and that the lefthand side of assignment statements are variables (and not, for example, constants). It also checks that every expression is meaningful in the sense of being correctly typed. (For example, adding two boolean values would not be allowed.) It checks that the tests in a conditional or while statement denote boolean values, and the right and left sides of an assignment have identical types. Finally, the arguments of the procedures `putint` and `getint` are checked to ensure they are integers, and the argument of `getint` is a variable and not a constant.

Task II.3: Identify all places in `Checker` where a contextual error in a MiniTriangle program can be flagged. Construct *one* MiniTriangle program that will flag all the errors. Include a comment line before each error to indicate which error should be flagged.

Task II.4: Extend the current implementation of the parser so that the abstract syntax trees that are constructed reflect the precedence of the operators detailed in section 3.1.

Evaluation of operators of the same precedence should be from left to right, except in the case of powers, where evaluation is from right to left. (So, for example, the structure of the abstract syntax tree for `1 - 2 - 3` should reflect the fact that its value is `-4`, and the structure of the abstract syntax tree for `2 ^ 3 ^ 2` should reflect the fact that its value is `516` (that is, 2^9, not 8^2).

4 Deadline

The deadline for your solution to all parts of the coursework is Friday, 21st March 2003.

5 Assessment

The coursework counts for 25% of the total mark. Questions in the written examination will test your understanding of the coursework. It is very important that you use the coursework to get a full understanding of the material in this module[1].

It is not possible to complete this coursework at the last minute. You will need to spread the work out over the term. It is recommended that you complete the first part of the coursework by the end of February. **No extensions to the deadline will be allowed except in exceptional extenuating circumstances supported by written evidence.**

This coursework will be assessed by a careful (human) reading of your solutions, including the code and the documentation. All answers should be accompamied by clear explanations, zero marks being awarded if no justification is given for an answer. It is therefore vital that you put much effort into making your solutions, code and documentation as readable as possible. Marks will also be awarded for evidence of extensive testing of the code.

6 Group Work and Plagiarism

Plagiarism is claiming that someone else's work is your own. The School has a strict policy regarding plagiarism and, if plagiarism is indeed discovered, this policy will be applied. Note that punishments apply also to anyone assisting another to commit plagiarism (for example by knowingly allowing someone to copy your code).

Plagiarism is different from group work in which a number of individuals share ideas on how to carry out the coursework. I would strongly encourage you to work in small groups, and will certainly not penalise anyone for doing so. This means that you may work together on the program. What is important to me is that you have a full understanding

[1] NB. If you are in the unfortunate circumstance of having to resit the module, your coursework mark will not count. In the examination in May/June, you will be asked to answer 3 questions, each counting for 25%. If you resit in September or next year you will be required to answer 4 questions in the same amount of time.

of all aspects of the completed program. In order to allow proper assessment that this is indeed the case you should adhere to the following requirements.

1. Include a list of the names of anyone with whom you have collaborated when submitting your work. The lists for the individuals in a group should all correspond. Groups larger than three *will not be permitted*. If you have not worked with others this should also be clearly stated.

2. *All documentation should be your own individual work.* Any evidence of copying of documentation will be treated as plagiarism. (Recall that all solutions will be read carefully.)

As a deterrent against plagiarism, a number of individuals, selected randomly, will be asked to discuss their work in detail, including demonstrating their program, after the submission deadline has passed.

7 What to Submit and When

Hard copy of the complete documentation, grammar, program code etc. should be submitted to room B33 of the Computer Science Building in printed form by 4pm on the relevant deadline. Additionally your completed program should be submitted electronically (by the same date and time). The program documentation should include a description *in your own words* of the techniques used to construct the program, written to make it easy for anyone wanting to add additional operators to the grammar

For further details of what to submit, and how, consult the web page for the coursework:

`http://www.cs.nott.ac.uk/~pni/G52CMP`

8 Getting Help

The main point of contact for help is the laboratory sessions in the terminal room A32 between 9am and 11am on 14th February, 21st February, 7th March, 14th March and 21st March. At these sessions, a number of assistants will be present in order to answer questions and provide support. Please use these sessions wisely. Note that it is likely to be difficult to obtain last-minute help.

In addition, questions can be posted on the coursework website. (See above for the link.) Do not expect immediate answers to such questions; they will only be answered at irregular intervals.

9 Appendix

```
Program              ::= single-Command

Command              ::= single-Command
                       | Command ; single-Command

single-Command       ::= Identifier := Expression
                       | Identifier ( Expression )
                       | if Expression then single-Command
                         else single-Command
                       | while Expression do single-Command
                       | let Declaration in single-Command
                       | begin Command end

Expression           ::= primary-Expression
                       | Expression Operator primary-Expression

primary-Expression   ::= Integer-Literal
                       | Identifier
                       | Operator primary-Expression
                       | ( Expression )

Declaration          ::= single-Declaration
                       | Declaration ; single-Declaration

single-Declaration   ::= const Identifier ~ Expression
                       | var Identifier : Type-denoter

Type-denoter         ::= Identifier
```

Figure 1: MiniTriangle BNF Grammar

Program	::=	Command	*Program*
Command	::=	Identifier := Expression	*AssignCommand*
	|	Identifier `(` Expression `)`	*CallCommand*
	|	Command ; Command	*SequentialCommand*
	|	`if` Expression `then` Command `else` Command	*IfCommand*
	|	`while` Expression `do` Command	*WhileCommand*
	|	`let` Declaration `in` Command	*LetCommand*
Expression	::=	Integer-Literal	*IntegerExpression*
	|	Identifier	*IdentifierExpression*
	|	Operator Expression	*UnaryExpression*
	|	Expression Operator Expression	*BinaryExpression*
Declaration	::=	`const` Identifier ~ Expression	*ConstDeclaration*
	|	`var` Identifier : Type-denoter	*VarDeclaration*
	|	Declaration ; Declaration	*SequentialDeclaration*
Type-denoter	::=	Identifier	*SimpleTypeDenoter*

Figure 2: MiniTriangle Abstract Syntax

```
Program             ::=  single-Command

Command             ::=  single-Command (; single-Command)*

single-Command      ::=  Identifier ( := Expression | (Expression) )
                       |  if Expression then single-Command
                            else single-Command
                       |  while Expression do single-Command
                       |  let Declaration in single-Command
                       |  begin Command end

Expression          ::=  primary-Expression (Operator primary-Expression)*
primary-Expression  ::=  Integer-Literal
                       |  Identifier
                       |  Operator primary-Expression
                       |  ( Expression )

Declaration         ::=  single-Declaration (; single-Declaration)*

single-Declaration  ::=  const Identifier ~ Expression
                       |  var Identifier : Type-denoter

Type-denoter        ::=  Identifier
```

Figure 3: MiniTriangle left-factorised EBNF grammar