
CM30171: Translation of TLL to three-address code and of three-address code to stack machine code

By Neil Munday

1 - Table of Contents

1 - Table of Contents	1
2 - Introduction.....	3
3 - Requirements	4
3.1 Functional requirements	4
3.2 Non-functional requirements.....	4
4 - The Three-Address Code	5
5 - The Zero-Address Code	7
6 - Design	9
6.1 Design considerations	9
6.2 Implementation overview.....	9
The TLL parser.....	9
The three-address code parser	10
6.3 Compilation of TLL to three-address code	11
6.4 Compilation of three-address code to zero-address code.....	15
7 - User Guide	18
7.1 Location of implementation	18
7.2 How to use the <i>TLLParser</i>	18
7.3 How to use the <i>TACParser</i>	18
8 - Testing.....	19
8.1 TLL to three-address code.....	19
Test 1	19
Test 2.....	19
Test 3	19
Test 4	20
Test 5	21

Test 6 22

Test 7 23

Test 8 24

Test 9 25

8.2 Three-address code to zero-address code 25

 Test 1 26

 Test 2 26

 Test 3 26

 Test 4 27

 Test 5 28

9 - Conclusion 30

 9.1 Critical analysis 30

 9.2 Further improvements 30

2 - Introduction

The aim of this project is to implement two of the common phases of a compiler. The high-level language that the compiler is to compile is TLL (This Little Language). The two phases to be implemented are: the generation of machine independent three-address code for an infinite register machine, and then to translate from this three-address code into the instruction set of a zero-address virtual machine implemented as a byte code interpreter. Two programs are to be developed – one for each phase.

This document describes the three-address code and zero-code address code implemented, the design of the two programs, a detailed description of the two translation phases, a user guide that details the location of the software developed on the BUCS system and how to use them, example tests of both programs, a critical analysis and directions for further improvements. The source code is provided at the end of this document.

3 - Requirements

Below are the requirements for the two programs implemented.

3.1 Functional requirements

- The two programs developed should read from standard input and write to standard output.
- The first program should accept a program written in TLL and output its translation to three-address code.
- The second program should accept a program written in three-address code and output its translation to the zero-address code accepted by the byte-code interpreter.
- The three-address code is designed by the implementer.
- The three-address code intermediate representation can use an infinite number of registers.

3.2 Non-functional requirements

- The programs may be written in any programming language.
- The programs should work correctly on the BUCS Unix systems.

4 - The Three-Address Code

Below is a table detailing the syntax and semantics of the three-address code developed.

Notes:

- Temporaries only appear once on the left hand side of any three-address code instructions.
- All variables are stored in temporaries.
- All variables are prefixed with a "\$" to distinguish them from temporaries.
- There is a stack that allows values to be pushed onto it and popped from it.
- The arguments for functions are pushed onto the stack in reverse order before a function is called, and the return value of a function is pushed onto the stack.

Instruction	Associated meaning	Description
tx	A temporary value	A temporary that holds the value of a variable or constant. A constant may be a string or integer value. x must be an integer. Example: $t1$.
$labelx$	A label	Defines a label that can be jumped to. A label can only be declared once. x must be an integer. Example: $label2$
$x = y$	Assignment	Stores the value of y in x . x can be a temporary, or a variable . y can be a temporary, integer, variable, or a string.
$x = y + z$	Signed integer addition	Stores the result of adding y to z in x . x is a temporary. y and z can be temporaries or integer values.
$x = y - z$	Signed integer subtraction	Stores the result of subtracting z from y in x . x is a temporary. y and z can be temporaries or integer values.

$x = y * z$	Signed integer multiplication	Stores the result of multiplying y and z in x . x is a temporary. y and z can be temporaries or integer values.
$x = y / z$	Signed integer division	Stores the result of dividing y by z in x . x is a temporary. y and z can be temporaries or integer values.
pushstack x	Pushing an item onto the stack	Pushes x onto the stack. x can be a temporary or an integer.
$x =$ popstack	Removes an item from the stack	Removes the top item on the stack and stores the value in x . x is a temporary.
call x	Calls function a function	Causes the program to branch to the function named by x . x must be the name of a function.
sub x	Begins the definition of a function	Defines the beginning of function x . When the <i>call</i> instruction is executed, the program will transfer control to this instruction.
endsub	Ends the definition of a function	
if x <i>op</i> y goto z	Conditional jump	If the result of evaluating x <i>op</i> y is true, then jump to z . x and y can be a temporaries or variables. <i>op</i> must be a relational operator defined by TLL. At the time of writing only the equality operator existed. z must be a label.
goto x	Unconditional jump	Jumps to x . x must be a label.

5 - The Zero-Address Code

Below is a table detailing syntax and semantics of the zero-address generated from the three-address code.

Notes:

- The zero-address code used was that provided by bci.lex.
- It was necessary to add two new instructions, BRAT and TEST to accommodate conditional statements as the original specification of the zero-address code only provided BNE (branch not equal to zero) and BLT (branch less than).
- Not all the instructions provided by bci.lex were used as the functional nature of TLL meant that some were not needed, e.g. LOC, LOC!, ARG! etc.

Instruction	Associated meaning	Description
+	Signed integer addition	Takes the first two items on top of the stack and adds them together, by popping each one off the stack and then saving the result back to the stack.
-	Signed integer subtraction	Takes the first two items on top of the stack and subtracts the top item from the second, by popping each one off the stack and then saving the result back to the stack.
*	Signed integer addition	Takes the first two items on top of the stack and multiplies them together, by popping each one off the stack and then saving the result back to the stack.
/	Signed integer division	Takes the first two items on top of the stack and divides the second item by the top item on the stack, by popping each one off the stack and then saving the result back to the stack.
=	Returns the top item on the stack	Pops the top item from the stack.
BRA	Unconditional branch	Pops the top item off the stack (which should be a label) and transfers control to the label. Unlike function calls, (see CALL), the BRA instruction does not create a stack frame that stores the environment at the time the BRA instruction was executed.

TEST	Equality test	Pops the top two items off the top of the stack and checks to see if the two items are equal. If so, a flag in the zero-address code machine is set to true (1), if not it is set to false (0).
BRAT	Conditional branch	Pops the top item off the top of the stack (which should be a label). It then examines the flag mentioned in the TEST instruction above. If it is set to true, then execution is transferred to the label, otherwise execution continues as normal.
CALL x	Function call	x must be an integer that corresponds to the number of arguments that the function to be called takes. The function to be called should be on top of the stack before the CALL instruction is executed. When a CALL instruction is executed, a new stack frame is created that stores the environment at the time the instruction was executed.
<code>:def body;</code>	Function definition / label	The colon defines the beginning of a function definition. A function definition may also serve as a label to be jumped to as well. <i>def</i> is the name of the definition and <i>body</i> is the set of instructions to be executed. The semi-colon defines the end of the definition and transfers control back to where the definition was called or branched to. Arguments to a function must be pushed onto the stack in reverse order before the function is called.
ARG x	Returns the value of argument x .	x must be an integer value that represents one of the arguments to a function. The ARG instruction must occur inside a function definition. ARG puts the value of the argument specified by x onto the top of the stack.

6 - Design

6.1 Design considerations

Having had experience using *JFlex* and *Java*, it was decided to use *JavaCC* to implement a parser for TLL and the generated three-address code. *JavaCC* generates top-down recursive descent parsers as opposed to *YACC* parser tools which generate bottom up parsers. By default *JavaCC* generates LL(1) parsers, but there may be areas of a grammar that are not LL(1). Should this occur, *JavaCC* uses semantic and syntactic look ahead locally to resolve shift-shift ambiguities at such points.

JavaCC also provides a tool called *JJTree* which is a powerful tree building pre-processor. It inserts tree building code inside the *JavaCC* specification. This tool can be also be used to create abstract syntax trees (AST) that support the visitor pattern.

To create a parser using *JJTree*, the user defines the grammars and lexical specifications in a *jjt* file as they would normally in a *JavaCC* file (*jj* file). In addition, the user can specify directives for *JJTree* and specify which non-terminals are to be used in the construction of the abstract syntax tree. The user can also specify what actions to take when nodes in the tree are created by embedding *Java* code into the grammar. *JJTree* will automatically generate *Java* classes for the nodes in the tree that have not been implemented by the user.

Once the *jjt* file has been successfully compiled by *JJTree*, the resulting *jj* file can be compiled by *JavaCC*. If successful, a set of *Java* class files will be generated, which can then be compiled into *Java* byte code.

6.2 Implementation overview

The TLL parser

Using the *LEX* and *YACC* specifications provided it was possible to create a parser for TLL (called *TLLParser*) using *JavaCC* and *JJTree* that generates an abstract syntax tree supporting the visitor pattern. This was done by creating a *JJTree* specification, *TLLParser.jjt* (see page **Error! Bookmark not defined.**) from which it was possible to create *TLLParser.java* by using *JJTree* and *JavaCC*.

In order to be able to store information at the abstract syntax tree nodes to be used later in the generation of three-address code, it was necessary to add extra functionality to the following nodes of the abstract syntax tree generated by *JJTree*: *SimpleNode* and its following sub-classes: *ASTVariable*, *ASTString*, *ASTPrimop* and *ASTInteger*. The extra functionality added to these classes was as follows:

`getType()` – returns a string corresponding to the type of the node.

`toString()` – returns a string containing the type of the node and if applicable, it's value.

The following methods were added to *ASTVariable*, *ASTString*, *ASTPrimop* and *ASTInteger*:

`getValue()` – returns a string containing the value of this node.

`setValue(String)` – sets the value held by this node. This method is used during the construction of the annotated syntax tree.

It was decided to use a visitor to generate the three-address code because *JJTree* will automatically generate an abstract syntax tree supporting the visitor pattern and the visitor pattern has its own advantages: it allows the programmer to define a new operation without changing the classes of the elements on which it operates; a visitor allows related operations to be defined in one class; it is possible to define a new operation over a structure by simply creating a new visitor; related behaviour of classes is not spread over the classes that define the object structure as it is defined in the visitor; and visitors can accumulate state as they visit each element of the object structure.

To walk the annotated syntax tree created by the parser and to create three-address code as each node is visited, the *TLLCodeVisitor* class (see page **Error! Bookmark not defined.**) was created. This visitor visits each node in the abstract syntax tree and generates three-address code for each of the nodes.

To support the *TLLCodeVisitor*, the following objects were created: *Stack*, *TACItem*, *Symbol*, *SymbolTable* and *Logger*.

Stack – This object models a stack, allowing *TACItem* objects to be pushed and popped from a *Stack* instance. As each node of the abstract syntax tree is visited, the result of evaluating a node (a *TACItem*) is pushed onto an instance of *Stack*.

TACItem – Objects of this type are used to store information about generated three-address codes. An instance of *TACItem* is treated as “complex” or “simple”. A complex *TACItem*, is an instruction in three-address code that references other three-address code items and which must not be removed from the stack. A simple *TACItem* is an element of three-address code that does not comprise of any other three-address code items, for example, an integer value.

Symbol – Objects of this type store information about variables and functions used in the three-address code. A *Symbol* object stores the type of the symbol, its value and if applicable, the temporary that stores the value of this symbol.

SymbolTable – This object models a symbol table. Objects of type *Symbol* can be added and removed from an instance of *SymbolTable* and it is also possible to search for *Symbols* held by an instance of *SymbolTable*. An instance of *SymbolTable* is used to store variables that have been used in the three-address code so that the temporaries associated with them can be looked-up later.

Logger – An instance of this class is used to output the generated three-address code to an output file.

The three-address code parser

To generate the zero-address code from the three-address code generated by the *TLLParser* it was decided to again use *JavaCC* to implement a parser for the three-address code. However, this time it was not necessary to create an AST as the three-address code is linear unlike TLL where the language has a tree structure, and *JavaCC* allows *Java* source code to be embedded

in the grammars. Therefore, as the three-address code is parsed, it is possible to generate the necessary zero-address code instructions.

A *JavaCC* specification *TACParser.jj* (see page **Error! Bookmark not defined.**) was created with the necessary *Java* code required to generate the zero-address code. From this specification, *TACParser.java* was created using *JavaCC*.

Once again, the *SymbolTable*, *Symbol* and *Logger* objects described above were used to support the generation of the zero-address code. A method called `add(String)` was implemented to append new zero-address code instructions to a string instance variable called *zac*, which stores previously generated zero-address code. This method formats the generated zero-address code so that it is easier for the user to read. So for example, definitions appear on new lines.

6.3 Compilation of TLL to three-address code

Firstly *TLLParser* creates an abstract syntax tree from the TLL source code which is read in from a file. If the source code is valid, then the parser will create a new instance of *TLLCodeVisitor* and instruct it to visit the root node of the tree. Once each node has been visited, the resulting three-address code is output to a file in order that it can be read by the *TACParser* later (if necessary).

The following table describes how *TLLCodeVisitor* evaluates the abstract syntax tree created by *TLLParser*.

Notes:

- As mentioned above, *TLLCodeVisitor* has access to an instance of *Stack* and *SymbolTable*.
- *TLLCodeVisitor* has four class variables that are used to indicate if the TLL code being evaluated is within the scope of another TLL instruction. They are passed down the abstract syntax tree. The class variables are: FUNCTION, LET, FUNC_DEF, and APPLICATION.
- The result of evaluating a node in the abstract syntax tree is pushed onto the stack.
- Once each node has been visited, the resulting three-address code will be on the stack in reverse order.

TLL	Evaluation description
Integer value	A new simple <i>TACItem</i> is created with the value of the integer, and is pushed onto the stack.
String	A new temporary is created to store the string, and a complex <i>TACItem</i> is created with the following form: <code>tx = string</code>

	<p>Where <i>tx</i> is the new temporary, and <i>string</i> is the value of the string.</p> <p>The <i>TACItem</i> is then pushed onto the stack.</p>
Primop	<p>A new simple <i>TACItem</i> is created with the value of the primop, and is pushed onto the stack.</p>
Variable	<p>If the scope that the variable is in has not been specified, then the symbol table is searched to find if the variable has already been assigned to a temporary.</p> <p>If it has, then a new simple <i>TACItem</i> with the temporary that holds the variable is pushed onto the stack.</p> <p>If the variable is not in the symbol table, then a new temporary is created to hold the variable, and a complex <i>TACItem</i> is created with the following form:</p> <p><i>tx</i> = <i>variable</i></p> <p>Where <i>tx</i> is the new temporary, and <i>variable</i> is the variable.</p> <p>The <i>TACItem</i> is then pushed onto the stack and a new entry is created in the symbol table for the variable.</p> <p>If the scope the variable is in is within a let binding or an application, then a simple <i>TACItem</i> is created to hold the variable. This <i>TACItem</i> is then pushed on the stack.</p> <p>If the variable is the name of a function being declared, then a simple <i>TACItem</i> is created to hold the variable. This <i>TACItem</i> is then pushed on the stack and a new symbol of type function is added to the symbol table.</p> <p>If the variable is within the body of a function, then if the variable exists in the symbol table, a simple <i>TACItem</i> with the temporary value associated with the variable is pushed onto the stack. Otherwise a new temporary is created and a new entry for the variable is created in the symbol table. A new simple <i>TACItem</i> is also created with the value of the new temporary and pushed onto the stack.</p>
Application	<p>If the application is a primop, then the primop child node is visited first in order to store the value of the operation for use later. Its value is retrieved by popping a <i>TACItem</i> from the stack once the node has been visited.</p> <p>The primop is then applied to each of the other children of the application node by visiting each node in turn. Complex <i>TACItems</i> are created for each application of the primop to the children and are</p>

	<p>pushed onto the stack.</p> <p>If the application is a function, then each of the arguments to the function are visited in reverse order. As each node is evaluated, pushstack three-address code instructions are generated to push the result of evaluating each node onto the stack of the three-address code machine. A <i>TACItem</i> for each pushstack instruction is pushed onto the stack.</p> <p>The first child is then visited in order to evaluate the name of the function with the scope set to APPLICATION. The symbol table is searched to see if the function name has an alias. This can occur if the function is bound to a variable in a let-binding. If it has an alias, its true name is retrieved from the symbol table. A call three-address code instruction is then generated for the function and stored in a complex <i>TACItem</i> that is pushed onto the stack.</p> <p>Finally, a <i>TACItem</i> is created to store a popstack three-address code instruction to store the return value of the function. This <i>TACItem</i> is then pushed onto the stack.</p>
Recursive function	<p>The first child node is visited to determine the name of the function with the scope set to FUNC_DEF. A new complex <i>TACItem</i> to store a sub three-address code instruction is then pushed onto the stack.</p> <p>Each argument node of the function is then visited with the scope set to FUNCTION and a popstack three-address code instruction is generated and stored in a complex <i>TACItem</i> for each one, which is then pushed onto the stack.</p> <p>The body of the function is then visited with the scope set to FUNCTION and an endsub three-address code instruction is generated and stored in a complex <i>TACItem</i> that is pushed onto the stack. The id of this <i>TACItem</i> is set to the name of the function.</p>
Let binding	<p>If the second child is not a recursive function definition, then the first child node is visited to determine the name of the variable to be used within the let binding. The second child is then visited, and the result of visiting the node is stored. A three-address code instruction in the following form is generated and stored in a complex <i>TACItem</i> that is pushed onto the stack:</p> <p><i>tx = variable</i></p> <p>Where <i>tx</i> is a new temporary and <i>variable</i> is the variable to be used within the let binding.</p> <p>The following three-address code is then generated and stored in a new <i>TACItem</i> which is pushed onto the stack:</p>

	<p><i>variable</i> = value of second child</p> <p>The symbol table is then searched to determine if the variable has been declared already. If it has, then it is removed temporarily from the symbol table, and a new symbol is added to the symbol table with its temporary value set to tx (see above).</p> <p>The third child is then visited. The following three-address code is then generated and stored in a new <i>TACItem</i> which is pushed onto the stack:</p> <p><i>variable</i> = value of <i>variable</i> before third child visited</p> <p>Any references to <i>variable</i> in the symbol table are then removed and any previous symbol table entries for <i>variable</i> that were removed earlier are restored in the symbol table.</p> <p>If however, the second child is a recursive function definition then the first child is visited to find out the name of the variable. The “\$” is removed from the variable name, as it will be bound to a function name.</p> <p>The second child node is then visited to determine the function name. Next the symbol table is search to see if the function has been defined before. If so it is removed from the symbol table temporarily so it can be added back later. A new function symbol is then created and added to the symbol table, specifying that the variable of the let binding is an alias for the function defined by the second child node.</p> <p>The third child is then visited. After this, the function symbol added earlier is removed and the original symbol put back (if necessary).</p>
Conditional	<p>The generated three-address code has the following form:</p> <pre> if true goto labelx result of visiting third child goto labely labelx result of visiting second child labely </pre> <p>Where <i>x</i> and <i>y</i> are integers.</p> <p>The first child node is visited to determine the condition to be tested. The three-address code for the if statement is then generated and stored in a new simple <i>TACItem</i> that is pushed onto the stack.</p> <p>The third child node is then visited.</p> <p>A goto instruction is then generated to jump out of the if statement</p>

	<p>to <i>labely</i>. This is stored in a new simple <i>TACItem</i> that is pushed onto the stack.</p> <p>Next a simple <i>TACItem</i> to store <i>labelx</i> is created and pushed onto the stack.</p> <p>The second child node is then visited. After this a simple <i>TACItem</i> to store <i>labely</i> is created and pushed onto the stack.</p> <p>If the scope is set to FUNCTION when the second and third child nodes are visited, then a pushstack three-address code instruction is generated. The argument for the pushstack instruction is determined by examining the top <i>TACItem</i> currently on the stack. This instruction is stored in a new simple <i>TACItem</i> that is pushed onto the stack.</p>
Definition	<p>A definition in TLL has the form: (<i>def variable term</i>)</p> <p>The second child node is visited first to determine the value of the <i>term</i>. The first child node is the visited to determine the value of the <i>variable</i>.</p> <p>The generated three-address code will have the form:</p> <p><i>variable</i> = result of evaluating the second child</p> <p>This three-address code is stored in a new <i>TACItem</i> that is pushed onto the stack.</p>

6.4 Compilation of three-address code to zero-address code

TACParser reads in the three-address code to be parsed from a file. As it parses the file, zero-address code is generated and stored in a string instance variable called *zac*. If the file is successfully parsed, then the resulting zero-address code is output to the screen and output to a file.

The following table describes how the *TACParser* generates the zero-address code from the three-address code.

Notes:

- *TACParser* has access to an instance of *SymbolTable*.
- *TACParser* stores the generated zero-address code in a string instance variable called *zac*.
- *TACParser* has three class variables that are used to check if the three-address code being parsed is within the scope of certain instructions. These are mainly used for zero-address code that is being generated within the scope of a function definition.

Three-address code	Evaluation description
Integer value	The integer is appended to <i>zac</i> .
Temporary, e.g. <i>tx</i>	The symbol table is searched to see if <i>tx</i> exists in the symbol table. If it does, then its corresponding temp value is returned. The temp value will be an ARG instruction. Temporaries and there corresponding ARG instructions are only added to the symbol table during the parsing of a function.
Assignment of a value to a temporary, e.g. <i>t1 = x</i>	If <i>x</i> is a variable, then the “\$” character is removed from the variable name and the resulting string is added to <i>zac</i> followed by a CALL instruction to retrieve the value of the variable. Otherwise <i>x</i> is added to <i>zac</i> .
Assignment of a value to a variable, e.g. <i>a = x</i> .	A colon is added to <i>zac</i> followed by the variable name without the “\$” character, and then the value of <i>x</i> is added to <i>zac</i> .
Math operation, e.g. <i>t1 = x op y</i>	If <i>x</i> is not a temporary, then its value is added to <i>zac</i> . Then if <i>y</i> is not a temporary, its value is added to <i>zac</i> . Finally, <i>op</i> is added to <i>zac</i> .
Conditional, e.g. <i>if x = y goto label</i>	The value of <i>x</i> is added to <i>zac</i> followed by the value of <i>y</i> and then a TEST instruction. Next <i>label</i> is added to <i>zac</i> followed by a BRAT instruction.
<i>tx = popstack</i>	If the instruction occurs at the beginning of a function, then its corresponding ARG instruction is generated and the resulting symbol is added to the symbol table.
<i>pushstack x</i>	If <i>x</i> is an integer, then it is added to <i>zac</i> . Regardless of what type of token <i>x</i> is, the argument counter instance variable (<i>args</i>) is incremented.
<i>goto label</i>	<i>label</i> is added to <i>zac</i> and then a BRA instruction is added to <i>zac</i> .
<i>label</i>	A semi-colon followed by colon is added to <i>zac</i> followed by <i>label</i> .
<i>call sub</i>	<i>sub</i> is added to <i>zac</i> followed by a CALL instruction with the number of arguments (e.g. CALL3). The number of arguments is determined by the <i>args</i> instance variable.
<i>sub x body</i> <i>endsub</i>	A colon followed by <i>x</i> is added to <i>zac</i> . Any <i>popstack</i> three-address code instructions are then parsed, followed by the body of the function. Upon

	parsing the endsub instruction, a semi-colon is added to <i>zac</i> .
--	---

7 - User Guide

7.1 Location of implementation

On the BUCS system the source code for the *TLLParser* and the *TACParser*, and its supporting classes can be found in the following directory:

```
~ma0nm/compilers/source
```

The *TLLParser* filename is:

```
~ma0nm/compilers/source/TLLParser.java
```

The *TACParser* filename is:

```
~ma0nm/compilers/source/TACParser.java
```

The *Java Documentation* for both programs can be found in the following directory:

```
~ma0nm/compilers/javadoc
```

`~ma0nm/compilers/javadoc/index.html` is the entry point for the *Java Documentation*.

7.2 How to use the *TLLParser*

At the command prompt type:

```
java TLLParser File
```

Where `File` is the name of the file containing the TLL source code to be compiled. If successful, the generated three-address code will be output to the command window and a file called `File.tac` will be created.

The user can see the abstract syntax tree as well as the generated three-address code by appending “debug” to the command line like so:

```
java TLLParser File debug
```

7.3 How to use the *TACParser*

At the command prompt type:

```
java TACParser File
```

Where `File` is the name of the file containing the three-address code to be compiled. If successful, the generated zero-address code will be output to the command window and a file called `File.zac` will be created.

8 - Testing

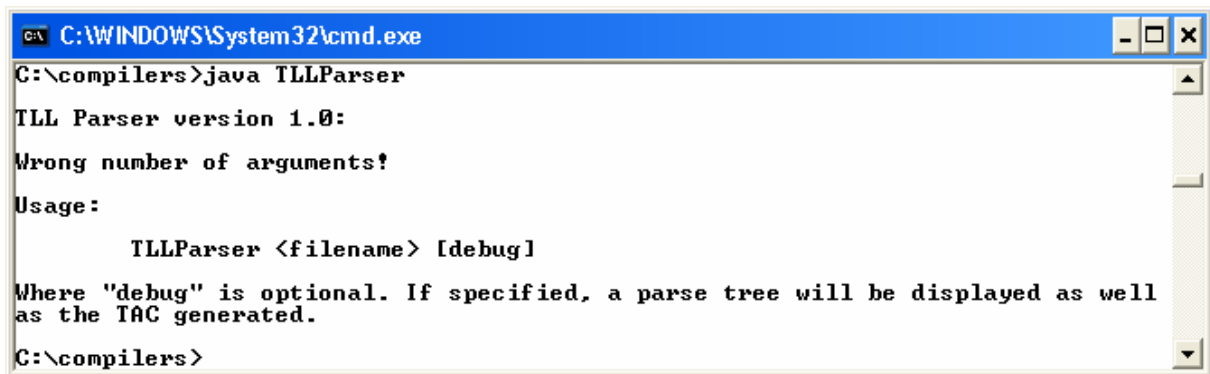
8.1 TLL to three-address code

The following tests demonstrate the *TLLParser*.

Note: All TLL source code was input by using a file called “test.txt” containing the source code.

Test 1

This test shows what happens when the parser is not passed a file to parse:



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser
TLL Parser version 1.0:
Wrong number of arguments!
Usage:
    TLLParser <filename> [debug]
Where "debug" is optional. If specified, a parse tree will be displayed as well
as the TAC generated.
C:\compilers>
  
```

Test 2

This test shows what happens when the parser is passed a file that does not exist:



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser afile
TLL Parser version 1.0:
Reading from file afile...
File afile not found!
C:\compilers>_
  
```

Test 3

Input: (+ (- 9 8) 3 (* 6 (/ 4 2) 7)), the debug option was also turned on.

Output:

The contents of test.tac was as follows:

```

t1 = 9 - 8
t2 = t1 + 3
t3 = 4 / 2
t4 = 6 * t3
t5 = t4 * 7
t6 = t2 + t5
  
```

```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt debug
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Beginning parse tree trace...
Start
Application
  Primop +
    Application
      Primop -
        Integer 9
        Integer 8
      Integer 3
    Application
      Primop *
        Integer 6
    Application
      Primop /
        Integer 4
        Integer 2
      Integer 7
Generating TAC...
t1 = 9 - 8
t2 = t1 + 3
t3 = 4 / 2
t4 = 6 * t3
t5 = t4 * 7
t6 = t2 + t5
TAC generation complete!
TAC output to file: test.tac
C:\compilers>

```

Test 4

Input:

```

(def n 2)
(def k 3)
(if (= n 1) (if (= k 0) (* 2 1 6) (+ 3 n)) (- (* 4 n) (* 9 8 5)))

```

Output:

The contents of test.tac was as follows:

```

$n = 2
t1 = $n
$k = 3
t2 = $k
if t1 = 1 goto label1
t3 = 4 * t1
t4 = 9 * 8
t5 = t4 * 5
t6 = t3 - t5
goto label2
label1
if t2 = 0 goto label3
t7 = 3 + t1

```

```
goto label4
label3
t8 = 2 * 1
t9 = t8 * 6
label4
label2
```

```
C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
$n = 2
t1 = $n
$k = 3
t2 = $k
if t1 = 1 goto label1
t3 = 4 * t1
t4 = 9 * 8
t5 = t4 * 5
t6 = t3 - t5
goto label2
label1
if t2 = 0 goto label3
t7 = 3 + t1
goto label4
label3
t8 = 2 * 1
t9 = t8 * 6
label4
label2
TAC generation complete!
TAC output to file: test.tac
C:\compilers>
```

Test 5

Input:

Recursive factorial:

```
(let fact (rec f (n) (if (= n 0) 1 (* n (f (- n 1))))) (fact 10 ))
```

Output:

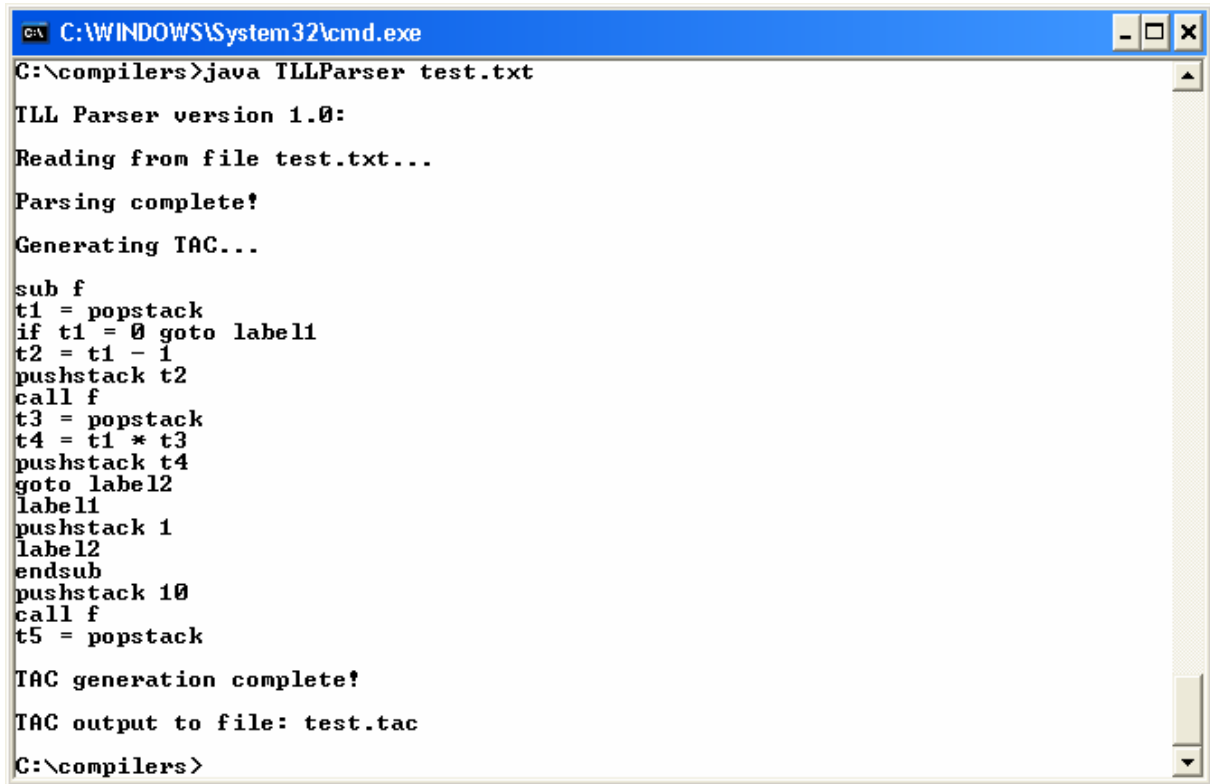
The contents of test.tac was as follows:

```
sub f
t1 = popstack
if t1 = 0 goto label1
t2 = t1 - 1
pushstack t2
call f
t3 = popstack
t4 = t1 * t3
pushstack t4
goto label2
```

```

label1
pushstack 1
label2
endsub
pushstack 10
call f
t5 = popstack

```



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
sub f
t1 = popstack
if t1 = 0 goto label1
t2 = t1 - 1
pushstack t2
call f
t3 = popstack
t4 = t1 * t3
pushstack t4
goto label2
label1
pushstack 1
label2
endsub
pushstack 10
call f
t5 = popstack
TAC generation complete!
TAC output to file: test.tac
C:\compilers>

```

Test 6

Input:

Recursive Fibonacci:

```
(rec f (n) (if (= n 0) 1 (if (= n 1) 1 (+ (f (- n 1)) (f (- n 2))))))
```

Output:

The contents of test.tac was as follows:

```

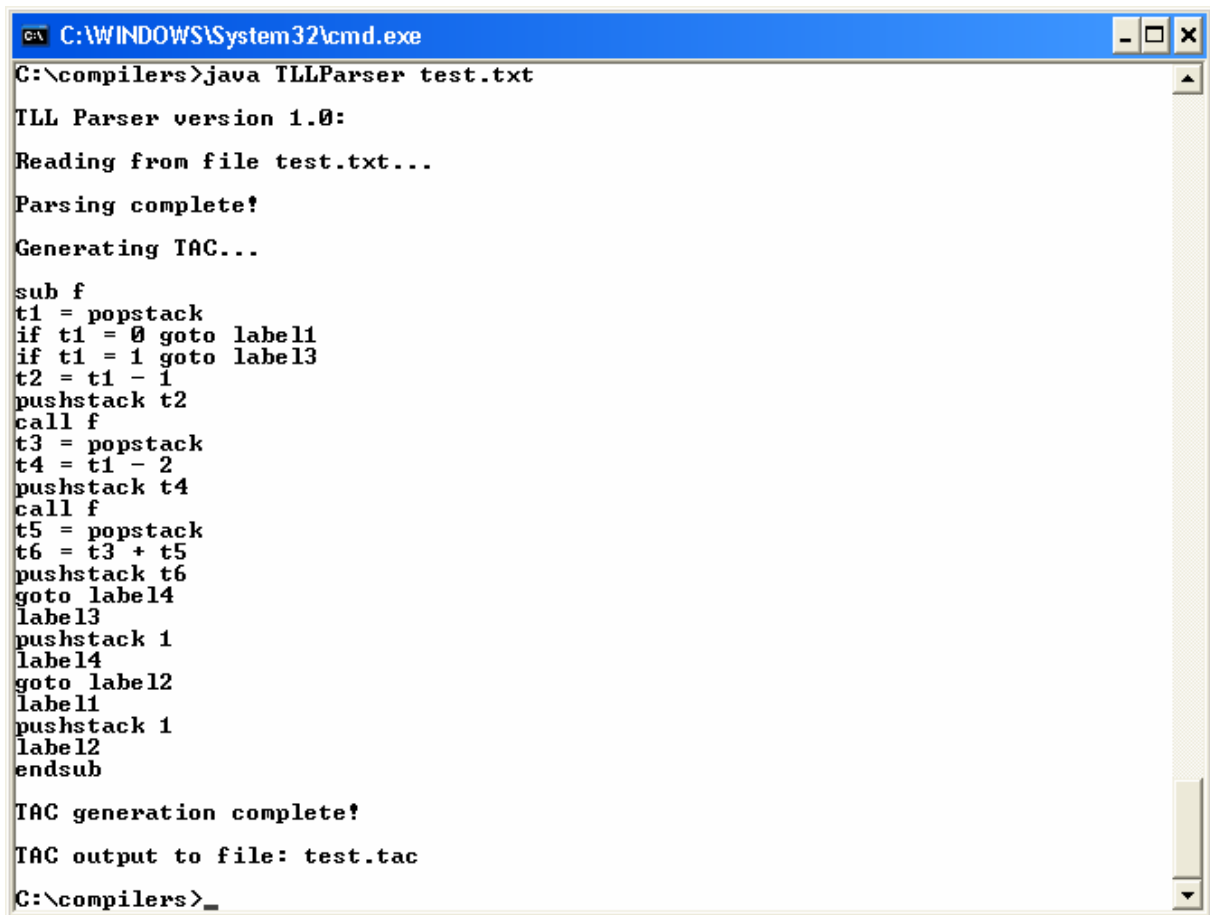
sub f
t1 = popstack
if t1 = 0 goto label1
if t1 = 1 goto label3
t2 = t1 - 1
pushstack t2
call f
t3 = popstack
t4 = t1 - 2
pushstack t4
call f

```

```

t5 = popstack
t6 = t3 + t5
pushstack t6
goto label4
label3
pushstack 1
label4
goto label2
label1
pushstack 1
label2
endsub

```



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
sub f
t1 = popstack
if t1 = 0 goto label1
if t1 = 1 goto label3
t2 = t1 - 1
pushstack t2
call f
t3 = popstack
t4 = t1 - 2
pushstack t4
call f
t5 = popstack
t6 = t3 + t5
pushstack t6
goto label4
label3
pushstack 1
label4
goto label2
label1
pushstack 1
label2
endsub
TAC generation complete!
TAC output to file: test.tac
C:\compilers>_

```

Test 7

Input: (let a (+ a c) (+ a b))

Output:

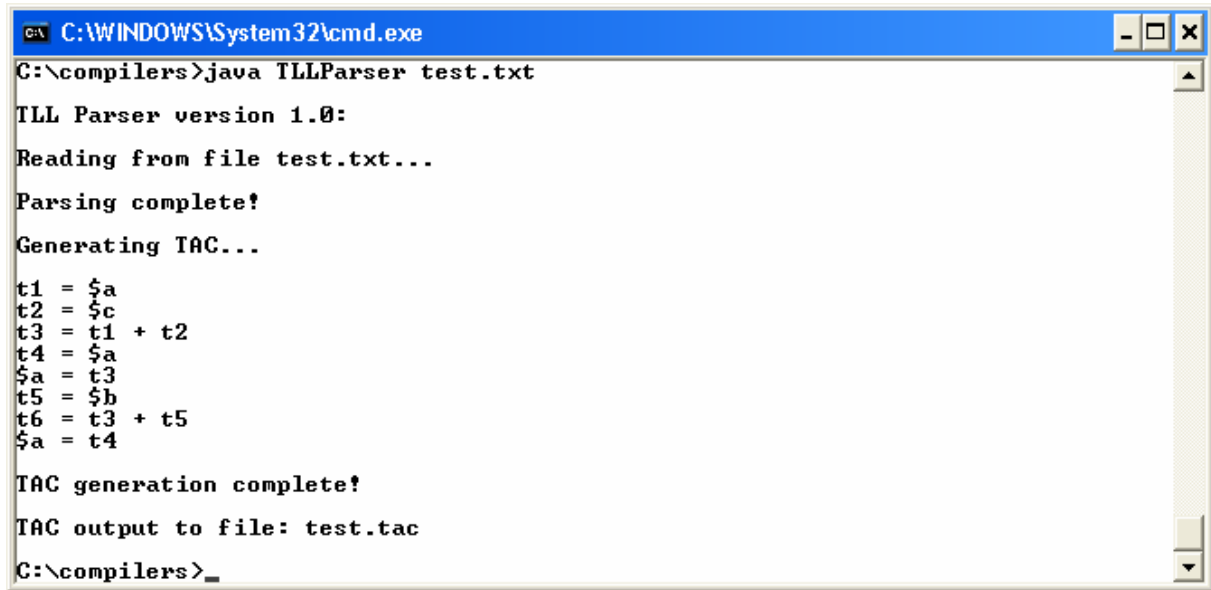
The contents of test.tac was as follows:

```

t1 = $a
t2 = $c
t3 = t1 + t2
t4 = $a
$a = t3
t5 = $b

```

```
t6 = t3 + t5
$a = t4
```



```
C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
t1 = $a
t2 = $c
t3 = t1 + t2
t4 = $a
$a = t3
t5 = $b
t6 = t3 + t5
$a = t4
TAC generation complete!
TAC output to file: test.tac
C:\compilers>_
```

Test 8

Input: (def a (+ (* 2 3) (+ 4 5 6) 11))

Output:

The contents of test.tac was as follows:

```
t1 = 2 * 3
t2 = 4 + 5
t3 = t2 + 6
t4 = t1 + t3
t5 = t4 + 11
$a = t5
t6 = $a
```

```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
t1 = 2 * 3
t2 = 4 + 5
t3 = t2 + 6
t4 = t1 + t3
t5 = t4 + 11
$a = t5
t6 = $a
TAC generation complete!
TAC output to file: test.tac
C:\compilers>

```

Test 9

Input: (rec f (n a) (if (= n 0) a (f (- n 1) (* a n))))

Output:

The contents of test.tac was as follows:

```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TLLParser test.txt
TLL Parser version 1.0:
Reading from file test.txt...
Parsing complete!
Generating TAC...
sub f
t1 = popstack
t2 = popstack
if t1 = 0 goto label1
t3 = t2 * t1
pushstack t3
t4 = t1 - 1
pushstack t4
call f
t5 = popstack
pushstack t5
goto label2
label1
pushstack t2
label2
endsub
TAC generation complete!
TAC output to file: test.tac
C:\compilers>_

```

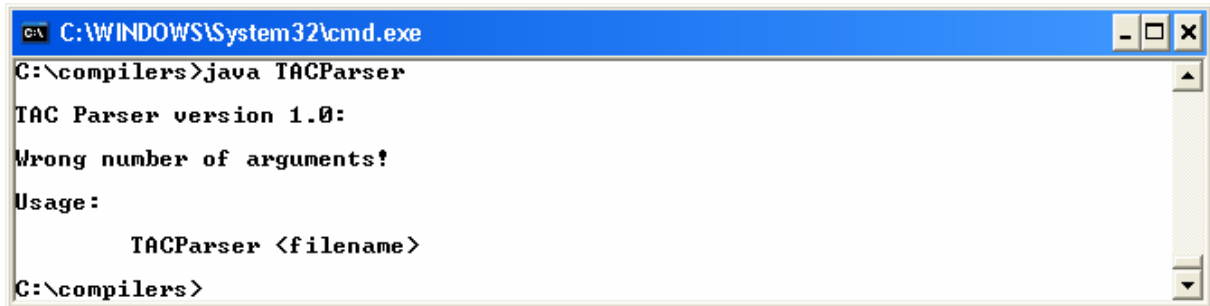
8.2 Three-address code to zero-address code

The following tests demonstrate the *TACParser*.

Note: All three-address code was input by using a file called “test.tac” containing the three-address code.

Test 1

This test shows what happens when the parser is not passed a file to parse:



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TACParser
TAC Parser version 1.0:
Wrong number of arguments!
Usage:
    TACParser <filename>
C:\compilers>

```

Test 2

Input:

From *Test 3* in the previous section on page 19:

```

t1 = 9 - 8
t2 = t1 + 3
t3 = 4 / 2
t4 = 6 * t3
t5 = t4 * 7
t6 = t2 + t5

```

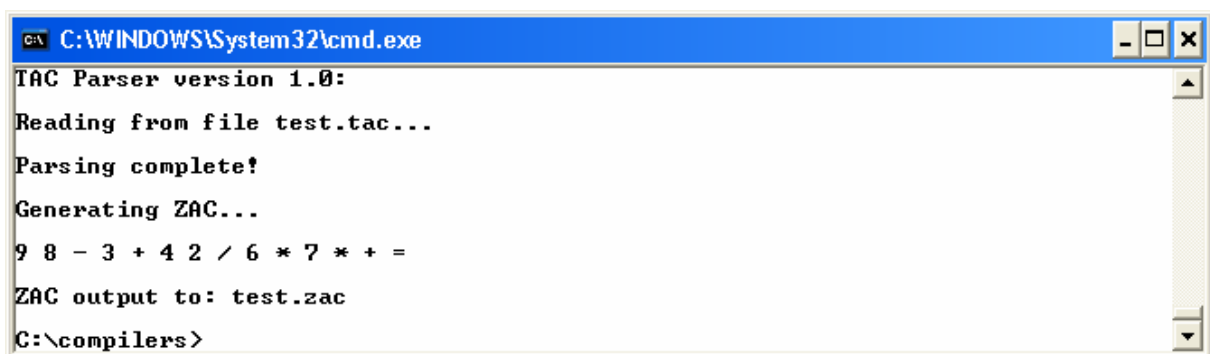
Output:

The contents of test.zac was as follows:

```

9 8 - 3 + 4 2 / 6 * 7 * + =

```



```

C:\WINDOWS\System32\cmd.exe
TAC Parser version 1.0:
Reading from file test.tac...
Parsing complete!
Generating ZAC...
9 8 - 3 + 4 2 / 6 * 7 * + =
ZAC output to: test.zac
C:\compilers>

```

Test 3

Input:

From *Test 5* (recursive factorial) in the previous section on page 21:

```

sub f
t1 = popstack
if t1 = 0 goto label1
t2 = t1 - 1
pushstack t2
call f
t3 = popstack
t4 = t1 * t3
pushstack t4
goto label2
label1
pushstack 1
label2
endsub
pushstack 10
call f
t5 = popstack

```

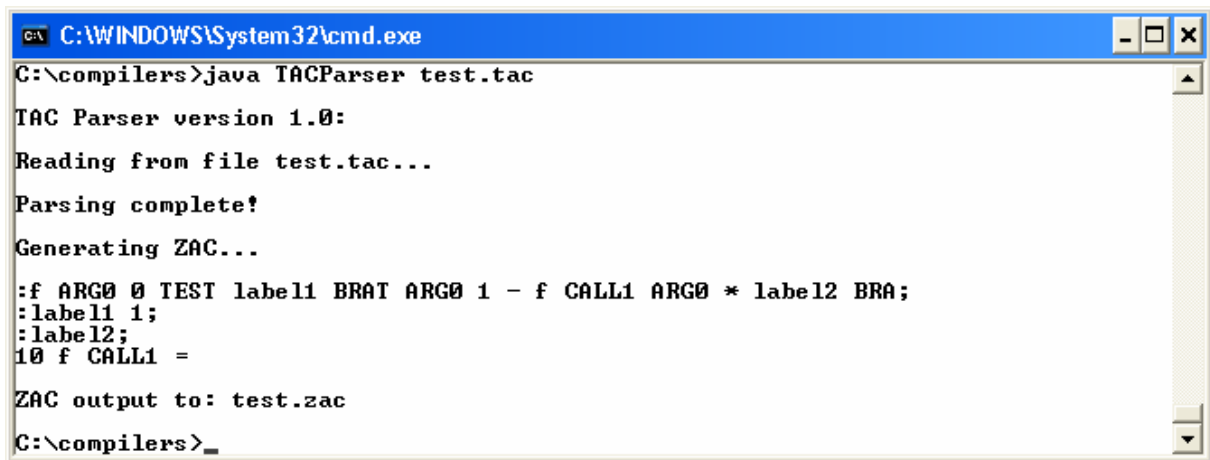
Output:

The contents of test.zac was as follows:

```

:f ARG0 0 TEST label1 BRAT ARG0 1 - f CALL1 ARG0 * label2 BRA;
:label1 1;
:label2;
10 f CALL1 =

```



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TACParser test.tac
TAC Parser version 1.0:
Reading from file test.tac...
Parsing complete!
Generating ZAC...
:f ARG0 0 TEST label1 BRAT ARG0 1 - f CALL1 ARG0 * label2 BRA;
:label1 1;
:label2;
10 f CALL1 =
ZAC output to: test.zac
C:\compilers>_

```

Test 4

Input:

From *Test 6* (recursive Fibonacci) in the previous section on page 22:

```

sub f
t1 = popstack
if t1 = 0 goto label1
if t1 = 1 goto label3
t2 = t1 - 1
pushstack t2
call f
t3 = popstack

```

```

t4 = t1 - 2
pushstack t4
call f
t5 = popstack
t6 = t3 + t5
pushstack t6
goto label4
label3
pushstack 1
label4
goto label2
label1
pushstack 1
label2
endsub

```

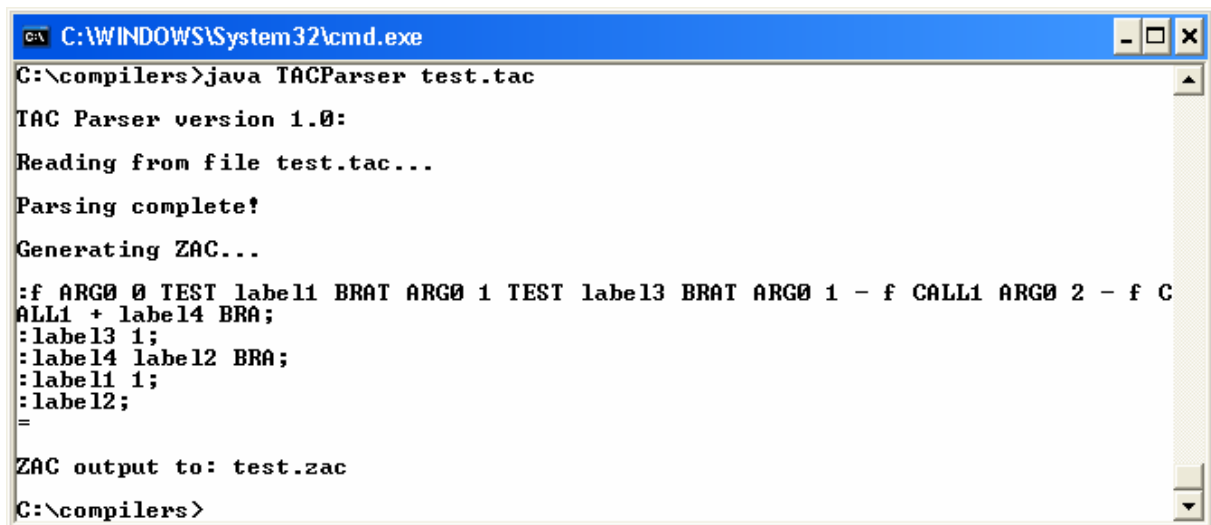
Output:

The contents of test.zac was as follows:

```

:f ARG0 0 TEST label1 BRAT ARG0 1 TEST label3 BRAT ARG0 1 - f CALL1 ARG0 2
- f CALL1 + label4 BRA;
:label3 1;
:label4 label2 BRA;
:label1 1;
:label2;
=

```



```

C:\WINDOWS\System32\cmd.exe
C:\compilers>java TACParser test.tac
TAC Parser version 1.0:
Reading from file test.tac...
Parsing complete!
Generating ZAC...
:f ARG0 0 TEST label1 BRAT ARG0 1 TEST label3 BRAT ARG0 1 - f CALL1 ARG0 2 - f C
ALL1 + label4 BRA;
:label3 1;
:label4 label2 BRA;
:label1 1;
:label2;
=
ZAC output to: test.zac
C:\compilers>

```

Test 5

Input:

From Test 9 in the previous section on page 25:

```

sub f
t1 = popstack
t2 = popstack
if t1 = 0 goto label1
t3 = t2 * t1

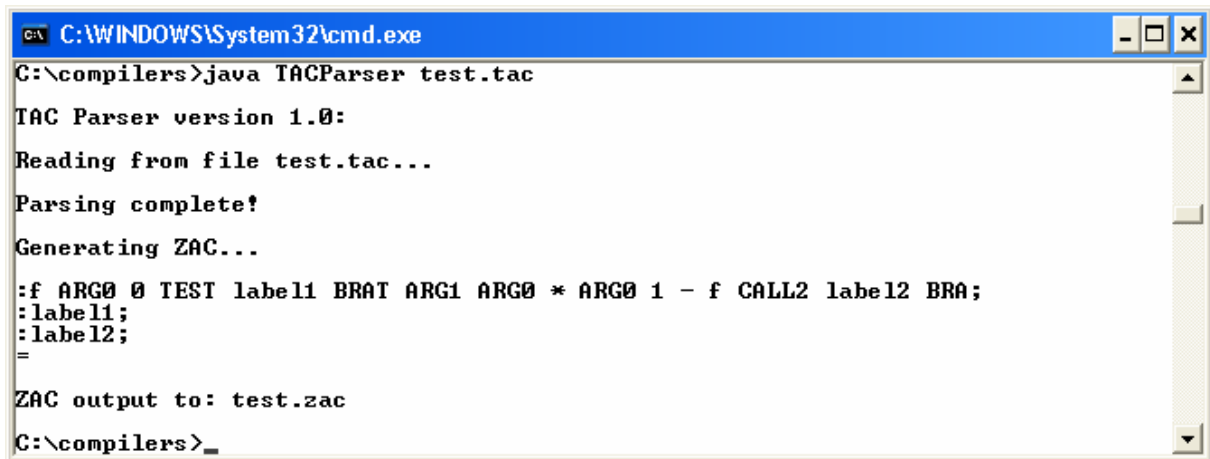
```

```
pushstack t3
t4 = t1 - 1
pushstack t4
call f
t5 = popstack
pushstack t5
goto label2
label1
pushstack t2
label2
endsub
```

Output:

The contents of test.zac was as follows:

```
:f ARG0 0 TEST label1 BRAT ARG1 ARG0 * ARG0 1 - f CALL2 label2 BRA;
:label1;
:label2;
```



```
C:\WINDOWS\System32\cmd.exe
C:\compilers>java TACParser test.tac
TAC Parser version 1.0:
Reading from file test.tac...
Parsing complete!
Generating ZAC...
:f ARG0 0 TEST label1 BRAT ARG1 ARG0 * ARG0 1 - f CALL2 label2 BRA;
:label1;
:label2;
=
ZAC output to: test.zac
C:\compilers>_
```

9 - Conclusion

9.1 Critical analysis

By using *JavaCC* it was relatively simple to implement a parser for TLL by using the provided *TLL.lex* and *TLL.yacc* files. The use of the visitor pattern was particularly useful as it aided my understanding and allowed me to see the advantages of using a visitor pattern for myself as a programmer. Also by using *JavaCC* it means that the parser can be used on any platform with a *Java* virtual machine.

As far as I can tell, the *TLLCodeVisitor* generates valid three-address code for all TLL with the exception of definitions that assign a variable to function and the lexical scoping of nested let-bindings may not be entirely correct.

In hindsight it would have been better to implement the three-address code in the following way:

```
opcode arg1 arg2 arg3
```

This is because it would make generating zero-address code much simpler as every three-address code instruction would have the same format.

The *TACParser* in most cases will generate valid zero-address code. The handling of three-address code generated from TLL definitions needs improvement, but this is mainly due to the way the *TLLCodeVisitor* generates three-address code for definitions. The *TACParser* may also not handle user variables and strings properly in some cases. However, for functions the *TACParser* will always generate the correct zero-address code.

If the two programs were not de-coupled, it would have made generating zero-address code easier as it would have been possible to access the internal state of the *TLLCodeVisitor* and therefore have access to the *TACItems* it had generated. It was also necessary (as explained earlier) to add two extra instructions to the byte-code interpreter's instruction set in order to implement conditional statements.

9.2 Further improvements

At present the only optimisation that the *TLLCodeVisitor* implements is the loading of variables into temporaries. It would be desirable to implement further optimisations such as constant folding, common sub-expression elimination, copy propagation, algebraic transformation and in particular dead code elimination as the three-address code generated often has redundant statements, for example multiple labels that are not always required are generated with nested conditional statements.

Also, by using the symbol table of the *TLLCodeVisitor* it would be possible to pick-up some syntax errors regarding variable use and function naming. At present the user can potentially call a function that has not been defined for example, and the same is true of variables that have not been assigned a value. It would be quite simple to implement by using the existing symbol table.

As mentioned above, the three-address code could be re-designed so that it uses a more machine friendly format. At present, the three-address code is quite human readable, but not very machine readable. The *TACParser* also needs to be improved so that it can compile correctly any three-address code generated by the *TLLCodeVisitor*.

It should also be possible to implement another visitor for the AST generated by the *TLLParser* that would compile TLL straight to zero-address code. This may in some ways be more efficient than generating three-address code and then generating from this zero-address code.

A final improvement would be to extend TLL's instruction set. This is because at present TLL only has one relational operation (equality), and the language could easily be extended to include the other relational operators. This would make the language more powerful and useful.