Abstract
We present in this paper an ambiguous and context-free grammar (AFEG) for a deterministic parsing phase in a Queue-Java compiler, which is based on Queue programming paradigm and is targeted for a novel Queue-Java computing system.

We shall show how the AFEG provides a clear and precise description of those strings that make up Java and Queue-Java programming. For every item, we will propose more than one rule and we will investigate, then, their usage rate. Selected rules were used in AFEG and implemented for our QJAVAC compiler that embed Queue Syntax tree Algorithm (QST).

1. Introduction
Java is steadily increasing in popularity in the embedded and network chips arena, fueled primarily by the convenience and elegance of its "write once and run anywhere" motto. The Java virtual machine is stack based and provides the advantage of dense code 1.8 bytes per instruction on average as compared to other CISC or RISC machines. No source or destination register identifiers need to be assigned for the instructions, making the instruction small size.

The Java technology achieves its platform independence by compiling Java source code into machine independent "byte codes" that are executed on the JVM. These byte codes are typically executed by an Interpreter, or a Just-In-Time (JIT) compiler, or executed directly by specialized Java processors. In addition, it was widely recognized that each current execution mode for java has its own strength and weakness. While software emulation is easy to implement, it incurs a high overhead during execution of certain Java workloads. A Java processor eliminates the software layer. However, a direct execution of byte codes on stack-based processors is invariably constrained by the limitations of the stack architecture for accessing operands. This inherent the dependence severely limits the instruction level parallelism.

In order to improve the instruction level parallelism in java byte code, we proposed a new parallel Java execution model named queue-based Java execution model (QJVM). The above new computing model is based on the Queue computing scheme and uses a first in first out (FIFO) queue data structure as the underlying control mechanism for the manipulation of operands and results. we had realized the implementation and the evaluation of a Queue-Java compiler (QJAVAC), which is a part of whole research project at Sowa Laboratory.

In the QJAVAC, we propose a new type of syntax tree-queue syntax tree-that is used for an optimized QJVM instruction generation. In this paper, the grammar items solving method will be given with queue syntax tree. In some grammar, more than one solution is proposed. As a result, we will show that the QJVM implemented in a pipelined arithmetic/logic unit (ALU) is more efficient than the stack based JVM. That is, the QJVM has the potential to take advantage of a pipelined ALU when the operand queue is not empty. On the other hand, the Stack based JVM cannot exploit pipelined ALU since the result of one operation must be returned to the top of the stack before they can become the operands of the next operation. In other words, the QJVM can take better advantage of overlapped fetch-execute stage.

2. QJAVAC Background
2.1 Java Language:
The Java programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features.

The Java programming language is strongly typed. This specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

2.2 The Queue Execution Model Java

The Queue Java is an extension of the Queue machine execution model and Java Platform. The Language and Virtual Machine Specification compose it. The syntax and semantics of Queue Java take no change with conventional Java System released from Sun Microsystems, the difference only occurs in the Virtual Machine.

As this well known, the conventional Java Virtual Machine uses stack based execution model, the queue based execution mechanism in a pipelined arithmetic/logic unit (ALU) is more efficient than the stack based execution mechanism. It can exploit more parallelism in fetch and execution than stack execution mechanism, so Queue Java Virtual Machine which uses queue based execution model, is expected to have a better performance than conventional Java.

3 Queue Syntax Tree for QJAVAC

Queue Syntax Tree is a new type of syntax tree optimum for Queue Execution Model Compiler, not only for qjavac.

To compare with the conventional Syntax tree, let us review the character using Fig 1

The character of it is that it has a leaf for each terminal and an internal node for each no terminal. The Relation (a line in the tree) presented in conventional syntax tree is only between children (the node) and parent (the node). For stack execution model, the instruction is emitted in the order

From above, it is very clear that the conventional syntax trees provide enough information for stack execution model's compiler. With the Relation, The stack compiler can easily jump to the children and back to parent to produce instructions in stack execution order. From some viewpoint, we can call a conventional syntax tree as stack syntax tree.

![Fig. 1 Conventional java syntax tree](image1)

But the problem occurs when we still use conventional syntax trees to deal with the queue execution model, the compiler, in order to find the deepest and shallowest nodes, must traverse the entire tree at first, remember the position of the nodes, and then load the tree again to emit instructions. Also, because there is not Relation existing between same-level nodes, finding same level nodes in instructions issue stage is very difficult and its algorithm becomes very complex and huge, also time-consume.

Here we introduce Queue Syntax Tree. Its characteristics are:

1. node still means one or more instructions that should been emitted in here
2. the Relation is still appeared between two nodes
3. the Relation not only means the parent-and-child relation also brother-and-brother one

Thus, the queue syntax of \( a \times c + (c - d) \div e \) becomes as shown in Fig 3.

![Fig. 2 Queue syntax tree for a × c + (c − d) ÷ e](image2)
nodes in the expression are kept, the Relation connects the brother-and-brother (same-level) nodes, the necessary connections among some parent-and-child nodes are preserved, but some no useful connection are cut off.

We can follow the next rules to get queue order instruction with queue syntax tree:
1. Enter a node, check whether it is a parent node.
2. If it is, flow to his child node.
3. If it is not, emit the instruction(s) belonging to of this node
4. Check it for whether it has right side-brother node.
5. If it has, flow to its right side-brother node(back up to step 1)

Under our experiment, with the number of nodes in a expression increasing, the consumed time of instruction emitting a queue syntax tree can be reduced a lot comparing with conventional syntax tree.

3. Language Grammar Items in QJAVAC

Unfortunately, there is no a parser that can recognize sequences of tokens according to java grammar and generate Queue Syntax Trees. Also our research core work is not to create it, so we have to choose some other solution to get Queue Syntax Trees. One is to convert the existing Stack Syntax Tree into Queue Syntax Tree.

3.1 Multiplicative, Additive, Shift and Bitwise Grammar Items in JAVAC

The Multiplication, Division, Remainder, Additive, Shift, and Bitwise items have a common character-when they act as a node in stack syntax tree, they must be a parent-node and have two children and those nodes only emit one Queue Java Virtual Machine Instruction.

Converting those Stack Syntax trees to Queue one is as follows:
1, the Relation between parent and right-child is removed
2, the Relation between parent and left-child may be removed or preserved: if it is not in most-left side of stack syntax tree, it will be cut off, if it is just there, it will not be cut
3, a new Relation is inserted between two children nodes because they are in same-level.

The flowchart of this conversion is shown in Fig 3 and In program, it is as follow:

```javascript
function ConvertFunction(stack.node, queue.node){
  if(CheckForHasChild(stack.node)){
    if(CheckForHasRightChild(stack.node)){
      ConvertFunction(stack.node.left-child, queue.node.child);
    }else{
      if(CheckForHasLeftChild(stack.node)){
        InsertNewQueueChildNode(queue.node);
      }else{
        InsertAsRightBrother(stack.node,queue.node);
      }
    }
  }else{
    if(CheckForHasRightChild(stack.node)){
      ConvertFunction(stack.node.right-child, queue.node.child);
    }else{
      InsertAsRightBrother(stack.node,queue.node);
    }
  }
}
```

With this algorithm, a stack syntax tree is converted to a queue syntax tree, then by using the Instruction Emitting Algorithm we had talked about, we can very easily get a queue exactation model instruction sequence.

(Fig. 2 → *SN=stack syntax node QN=queue syntax node)

3.2 Postfix Increment and Decrement Grammar Items

With the Unary Postfix Increment Item (++) at run time, the value 1 is added to the value of the variable and the sum is stored back into the variable. The value of the postfix increment expression is the value of the variable before the new value is stored.

Unary Postfix Decrement Item (--), at run time, the value 1 is subtracted from the value of the variable and the difference is stored back into the variable. The value of the postfix decrement expression is the value of the variable before the new value is stored.

An expression content Unary Postfix Item can be represented with f(a) + (b++) + f(c); it also can been divide into two parts: f(a)+b+f(c) and b=b+1

We can convert two Stack Syntax Trees independently, change one expression into two ones, but it will lead to a loss of parallelism, so after dividing, we can unite the two parts again, because [store b] in the second part must be later than the [load b] in the first part, so we raise the second part to an upper level than the first part.
The Algorithm of this conversion is shown in Fig 3.

With this algorithm, a stack postfix operator syntax tree is converted to a normal queue syntax tree, then using the Instruction Emitting Algorithm we had talked about, we can very easily get a queue execution model instructions sequence.

![Algorithm Diagram]

(Fig. 3 SN=stack syntax node, QN=queue syntax node)

3.3 Relational (Comparison) Items

The Relational Items (<, <=, >, >=) are issued by following next rules:

1. The value produced by the < is true if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is false.
2. The value produced by the <= operator is true if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is false.
3. The value produced by the > operator is true if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is false.
4. The value produced by the >= operator is true if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is false.

Here, the false means 0; the true means 1.

The expression is f(a)>f(b) or f(a)>=f(b) or f(a)<f(b) or f(a)<=f(b).

The Relational Items Syntax Tree sometimes is very similar to a binary item, its parents have two children. For integer numeric comparison, there are several instructions for different comparison case in Queue Java Virtual, so it is very easy. But for long and double numeric comparison, there is one comparison instruction, so one time comparison operation has those steps:

1. First, get all of three relational-status of two numbers (less '1', equal '0', greater '1')
2. Second, according to the operator type, load 1 or 0 to queue
3. Third, compare the result we had got and the 1(0) that we load in Step 2, according the operator type, load the correct value to queue.

With this, the Long type numeric comparison becomes a common integer type numeric comparison.

The Algorithm of this conversion is shown is Fig 4.

3.4 Conditional-And and Conditional-Or Items

The operand of the Conditional-And (&&) Items and Conditional-Or (||) Items must be of type Boolean. For an expression f(a) && f(b), if any of f(a) and f(b) is false, then the value of the conditional-and expression is false; only if all of f(a) and f(b) are true, the value of the conditional-and expression is true for an expression f(a) || f(b), if any of f(a) and f(b) is true, then the value of the conditional-or expression is true; only if all of f(a) and f(b) are false, the value of the conditional-or expression is false.

There are two solutions for Conditional-And and Conditional-Or Items, when an expression has two or more operators like this:

f(a) && f(b) && f(c).

The first:

We evaluate all operands of an expression like a binary tree, from in left-to-right, evaluate the whole expression stepping two values of operands by two.

The conversion algorithm is presented here below:

1. Get a stack syntax tree's node and a queue syntax tree's node
2. Check whether the stack syntax's node
whether be a conditional operator
2.1 Take the left-child node of current stack syntax's node and queue node's child.
2.2 Append the left-child node to queue node's child in most right brother.
2.3 Append the right-child node to queue node's child in most right brother.
2.4 Append the current node to current queue syntax's node as its most right brother.

The problem is: although the parallelism is very high, but there would be the no-using computing, for example, in Conditional-And a&&b&&c&&d, if b had been evaluated, to be false, the expression value is false, there is no need to evaluate the value of c and d. In this solution we can not avoid it. The result of this is shown in Fig 5. (QCN: queue condition node)

The second
We evaluate operands of an expression one by one left-to-right, if anytime a false value (in Conditional-Or, is value true) is returned, we can terminate the following computing at once and return false (in Conditional-Or, return true) as the whole expression value.

The Algorithm of this conversion is:
1. Get a stack syntax tree's node and a queue syntax tree's node
2. Check whether the stack syntax's node is conditional operator
3. If it is,
   3.1 Check whether the stack syntax's node has left-child node
   3.2 If it has, then get the left-child node and queue node and go to step 1
   3.3 check whether the stack syntax's node has right-child node
   3.4 if it has, then get the right-child node and queue node and go to step 1
4. Append the current stack syntax's node to current queue syntax's node as its right-brother.
5. Append a checker node to current queue syntax's node as its right-brother.

The problem is: Although we lose useful computing, at the same time we lose the parallelism, because the computation becomes sequential, the result of this is shown in Fig 5. (VCN: value check node)

3.6 Question-Mark Items

The conditional operator ? : uses the Boolean value of one expression to decide which of two other expressions should be evaluated.

Expression ? Expression: Expression
The conditional operator has three operand expressions: the ? appears between the first and second expressions, and : appears between the second and third expressions.

The first expression must be the type of Boolean.

At run time, the first operand expression of the conditional expression is evaluated first; its Boolean value is then used to choose either the second or the third operand expression:
1. If the value of the first operand is true, then the second operand expression is chosen.
2. If the value of the first operand is false, then the third operand expression is chosen.

For example, expression f(a) ? f(b) : f(c) + f(d) + f(e), if f(a) is greater than 0 then the value of expression is f(b) + f(b) + f(c), else the value is f(c) + f(d) + f(e),

its Stack Syntax is shown in the figure 4:

Here the ϵ() means that it is a Stack Syntax Tree.

Fig. 4 Stack syntax tree of question mark operator

In Stack Java Virtual Machine, if, in an expression, Question-Mark Operator is not the most-left part, it executes well as normal way. From above, we can see, because the result of left part of the question-mark is pushed to the stack, and the value of select-controller is also saved on the top of stack, the jump-direction can be decided at once.

The conversion of the stack syntax tree to queue tree normally will cause failure of evaluating the expression, because the results of the left part of question-mark is lined in the front of it, the select-controller can get value to decide which direction should jump to. So we choose a speculation model to deal this case:

Solution 1

Solution 2

Fig 5
three expressions are evaluated in parallel, at last, when the three results come the head of queue together, we can select one correct result, if the first value in the queue-head is true, it means the first expression’s result is true, and we should select the second expression’s result as the correct result, the selection step as follow:
1. duplicate one time ,copy the correct result to the tail of the queue
2. pop two times, to discard the correct(it had been saved) and incorrect result if the first value in the queue-head is false, it means the first expression’s result is false, and we should select the third expression’s result as the correct result, the selection step as follow:
1. pop one time, to discard the incorrect(the result of second expression)
2. duplicate one time ,copy the correct result(the result of third expression) to the tail of the queue
3. pop one time again, to discard the correct(it had been saved)

The explanation in “The conditional operator ‘?’ ” should not be “… should be evaluated.”, but is “… should be selected.”

The Algorithm of this conversion is that:
1. get a stack syntax tree’s node and a queue syntax tree’s node
2. check whether the stack syntax’s node is question mark operator
3. if it is,
   (1) get the child-level node of current queue syntax’s node
   (2) append the first expression into the most right end of the child node of queue node
   (3) append the second expression into the most right end of the child node of queue node
   (4) append the third expression into the most right end of the child node of queue node

   (5) append a select node to current queue syntax’s node as its right-brother.

In program, it is like this:

```cpp
function ConvertFunction(stack. node, queue. Node)
if(CheckForIsaAssignment(stack. node)) {
    node = GetTheChildOf(queue. node)
    InsertAsRightBrother(stack. node.first, node)
    InsertAsRightBrother(stack. node.second, node)
    InsertAsRightBrother(stack. node.third, node)
}
```

Fig. 6 converting the stack syntax tree to queue syntax tree in question mark operator

Although the parallelism is very high in speculation model, it can cause no useful computing, for example, because we only use one of results of f(a) and f(b), so the part of calculation must be no useful computing. But, after all we can execute the question mark correctly.

4. Conclusions and Future Work

The work in this thesis has shown that a queue machine model is as powerful as a conventional stack machine in terms of evaluating arbitrary expressions. Furthermore, it was shown that instruction sequences for the QEM has more parallelism than SEM instruction sequences. An important observation is that a QEM machine is better than a SEM machine at utilizing a pipelined arithmetic/logic unit. There still remain many unresolved compiler implementation issues. The current compiler dose not completely decompose any loop bodies and conditionals for execution in separate contexts. In many cases, this decomposition is excessive and actually increases computational throughput. Work needs to be done to develop a
more intelligent partitioning of the source program. A queue machine processing element architecture has been proposed and specified in some detail. Work still remains to specify the control logic for the processing element and possibly to implement a prototype in silicon.

References
1. Sowa Laboratory, the University of ElectroCommunications: http://www.sowa.is.uec.ac.jp
2. Wang L. and Sowa M. “High Parallelism Java Compiler with Queue Architecture”.
8. SUN MICROSYSTEM , “picoJava-I microprocessor core architecture” http://www.sun Microsystems