SFU CMPT 379
Compilers
Fall 2016
Assignment 2

Assignment due Tuesday, October 25, by 11:59pm.

For this assignment, you are to expand your Pika-1 compiler from assignment 1 to handle Pika-2. Project setup and acceptable submissions are the same as for assignment 1 (for example, submit a java eclipse project).

Pika-2 is backwards compatible with Pika-1. All restrictions and specifications from Pika-1 are still in force, unless specifically noted. (For example, there are still at most 32 characters in an identifier, even though I no longer note this on the Pika-2 description.)

Language Pika-2

Tokens:

integerConstant → (**+**|**-**)? [**0..9**]+ // has type "integer"
floatingConstant → (**+**|**-**)? ([**0..9**]* **.** [**0..9**]+) (**E** (**+**|**-**)? [**0..9**]+)? // has type "floating"
booleanConstant → **_true_** | **_false_** // has type "boolean"
stringConstant → " [^"**\n**]* "
// **\n** denotes newline. has type "string"
// (pika does not interpret _n_ in a string constant as a newline.)
characterConstant → **^α^**
// **α** is any printable ascii character (encoding is decimal 32 to 126)
// starts and ends with circumflex. **^α^** has type "character"
identifier → [**a..zA..Z_**][**a..zA..Z_$0..9**]*
//at most 32 characters in an identifier

punctuator → *operator* | *punctuation*
operator → *arithmeticOperator* | *comparisonOperator* | *booleanOperator* | *otherOperator*
arithmeticOperator → **+** | **–** | ***** | **/**
comparisonOperator → **<** | **<=** | **==** | **!=** | **>** | **>=**
booleanOperator → **&&** | **||** | **!**
otherOperator → **//** | **///** | **////** // yes, they're literally two to four slashes. They deal with rationals.

punctuation → **;** | **,** | **.** | **{** | **}** | **(** | **)** | **[** | **]** | **|** | **#**| **:=**

comment → **#** [^**#\n**]* (**#** | **\n**)

Grammar:

S → **exec** *blockStatement*
blockStatement → **{** *statement** **}**

statement → *declaration*
assignmentStatement
ifStatement
whileStatement
printStatement
blockStatement
releaseStatement

declaration → **const** *identifier* **:=** *expression* **.** // immutable "variable"
var *identifier* **:=** *expression* **.** // mutable variable

assignmentStatement → *target* **:=** *expression* **.** // Reassignment. target and expr must have same type.
target → *expression* // must be a targetable expression.

printStatement → **print** *printExpressionList* **.** // print the expr values
printExpressionList → *printExpression* ⋈ (**,** | **;**)
printExpression→ *expression* | **_n_** | **_t_** | ε

ifStatement → **if (***expression***)** *blockStatement* (**else** *blockStatement*)? // expression must be boolean
whileStatement → **while (***expression***)** *blockStatement* // expression must be Boolean
releaseStatement → **release** *expression* **.** // expression must be reference type

expression → *expression operator expression* // all binary operations left-associative
! *expression* // Boolean negation
(*expression* **)** // targetable iff the expression is.
[*expression* **|** *type* **]** // casting
expression **[** *expression* **]** // array indexing. Targetable if array's elements are.
length *expression* // expression must be array type
arrayExpression
literal

type → *primitiveType* | *arrayType*
primitiveType → **bool** | **char** | **string** | **int** | **float** | **rat** // rat is rational type
arrayType → **[** *type* **]** // an array of the given type

arrayExpression → **new** *arrayType* **(** *expression* **)**
[*expressionList* **]** // expressions must all be promotable to same type
clone *expression* // expression must be array type

literal → *integerConstant* | *floatingConstant* | *booleanConstant* | *characterConstant* | *stringConstant*
identifier // identifier can be targetable.

1. **Boolean expressions**

The boolean-and operator **&&** has the signature (boolean, boolean) -> boolean.
The boolean-or operator **||** also has the signature (boolean, boolean) -> boolean.

Both of these operators perform short-circuit evaluation of their operands from left to right. This means that if the outcome of the operation is known after evaluating the left operand, then the right operand will not be evaluated.

The prefix boolean-not operator **!** has the signature boolean -> boolean.

2. Control flow

The control-flow statements (**if** and **while**) work as they do in most block-oriented languages (such as java and C++,) but note that they only take blockStatements as their clauses. In other words, one has to use the curly-braces **{** and **}** around the subordinate code, even if it is only one statement.

The **while (** *condition* **)** *body* loop checks the condition before entering the loop body, and may therefore execute the body zero times (if the condition is false upon statement entry).

3. Targetable expressions

An expression is called *targetable* if it may appear as the target of an assignment statement. Syntactically, an identifier is targetable and an array indexing expression (*expression*[*expression*]) is targetable. Also, a parenthesized expression is targetable if the expression inside the parentheses is. Any other target expression generates a syntax error.

Semantically, an identifier must be declared with **var** to be targetable. Using any other identifier as a target results in a semantic error. All array elements are targetable.

In the code generator, targetable expressions will conveniently make *address code* rather than *value code*. In C++, the concept of *lvalue* corresponds with Pika's *targetable*.

4. Type system

We now define a *type system* for Pika: a way to write down types and some rules about them. Type systems are a broad concept: they are used to write about, reason about, and specify features of languages and programs. They often contain features that are not simply the types used within a language. For instance, in Pika, the type system will include the *signatures* of the operators, which one cannot specify within the language Pika itself.

Be very careful with types. The notation described below **is not used <u>in</u> Pika programs.** It is used by people talking about Pika programs. Only where it refers to explicit type specification or Pika-syntax representation are we talking about the things a Pika programmer can use in a program.

4.1. Primitive types

Primitive types are the basic built-in types in a type system, which cannot be further decomposed.

There are six primitive types in Pika_2: the five types from Pika_1: boolean, character, floating, integer, and string, as well as a new type, *rational*. In our type system, we will denote these types using their corresponding keywords **bool**, **char**, **float**, **int**, **rat**, and **string,** or, when brevity is desired, **b**, **c**, **f**, **i**, **r**,

and **s**, respectively. The bold type on these notations in the previous sentence is to make them evident in that sentence; it is not necessary to embolden them in practice.

4.2. Compound types

A compound type is a type somehow composed from another type or types. Records, arrays, and objects are examples of compound types.

Pika_2 introduces compound array types. Array is not itself a type, but **array [** *type* **]** is, for any valid type. (Here and henceforth we consider types and their denotation in the type system to be identical.) We may abbreviate array[*type*] as **a[***type***]** or simply **[***type***]**.

Thus, the following are all valid Pika_2 types:

array[bool]
array[rat]
array[array[f]]
a[a[a[a[c]]]]

Although Array is not a type, we use the phrase "array type" to stand in for "some type **array[**T**]**" where **T** can be any type".

array[ ] is an example of a *type constructor*. It is an operator we can apply to types in our type system to create a new type.

4.3. Value types and reference types

Variables in the source program are associated with memory locations in a running program. If the memory location holds a value, the variable is a *value variable*, and if it holds a reference (pointer) to where the value is, the variable is called a *reference variable*. Typically the choice between keeping a variable as a value variable or reference variable is made based on its type. Thus, we will classify types as *value types* or *reference types* if their variables are implemented as value variables or reference variables, respectively.

Primitive types are often value types, and we will mainly follow this convention in Pika. The rational type will be a value type. Strings are an exception, being a reference type; we have already noted this in Pika-1. The compound *array* types will be reference types. Reference variables (variables whose type is a reference type) in Pika will each consume 4 bytes, which is the size of a pointer on the ASM.

If *T* is a reference type, we will make a distinction between a *variable of type T*, which evaluates to a pointer, and a *record of type T*, which is a section of memory that (1) the pointer of a variable of type *T* points at, and (2) holds the values associated with the referenced structure. For instance, a variable of type array[float] holds a pointer, and that pointer should point at a record of type array[float], which is a hunk of memory holding a sequence of floating-point numbers (along with some extra array-control information; see below).

In java, primitive types are value types, and all objects are reference types. In C++, all types are value types, but there are compound types for pointers and references.

4.4. A signature of an operator

An n-ary operator is said to have a *signature* of

$$(type_1, type_2, \ldots type_n) \rightarrow type$$

if it accepts operands of types $type_1, type_2, \ldots type_n$ in order, and from them produces a result of type *type*. Sometimes the commas between the operand types are replaced with the Cartesian product symbol ×:

$(type_1 \times type_2 \times \ldots type_n) \rightarrow type$
in which case the parentheses are optional.

For instance, the binary operator ***** has a signature of (int, int) → int, as it accepts two integer operands and from them produces an integer result. It also has a signature of (float, float) → float.

As a further example, the unary operator **!** has a signature of (bool) → bool.

4.5. The type of operators

The *type* of an operator is the set of all of signatures that it accepts. For instance, ***** has a type of

{ (int, int)→int, (float, float)→float }

We sometimes call the type of an operator the *signatures* of the operator, for obvious reasons. (Note the plural usage here is distinct from the singular usage of section 4.4).

If an operator has only one signature, we sometimes shorten the notation by omitting the set braces. For instance, the operator **!** has a type of

(bool) → bool.

Or, equivalently and more formally,

{ (bool) → bool }.

4.6. Operators with *type* operands

Pika_2 has two operators that take *type* (rather than expression) as one of their operands. These are the operators **[|]** (casting) and **new** (array creation). There are several ways we can notate and think about these operators.

4.6.1. Type operands as ordinary operands

The first way is to treat the operand as we do other operands, giving it an effective type of whatever type it represents. For instance, we could consider the cast **[68 | bool]** to have an integer first operand and a boolean second operand. (Note that for it to really have a boolean second operand, it would have to be something like **[68 | _true_]**). However, if we set the types of type expressions this way in the semantic analyzer, we can use signatures where (int, bool)->bool is a signature of casting. This is effective but not really clear.

4.6.2. Type operands as type literals

Another way is to treat the type operand as a *type literal* of its given type. This just means that the program text is literally a type. We'll use the notation **type T** for a type literal for the type **T**. Using this convention, we can speak of casting as having the signature

(int, type bool) → bool

With the example

[42 | bool]

matching that signature…this expression thus evalutates to a boolean value.

Note that we never have a type literal as the result of an operation; they can only be operands.

4.6.3. Type operands as constant parts of the operator

The third way is to treat the type operand as built into the operator. For instance, rather than considering **[|]** an operator, we consider **[|float]** as an operator. (And **[|int]**, **[|bool]**, **[|[int]]**, etc.) In this way, we can say that **[|float]** has a type of

{ (int) → float }

and that **[| int]** has a type of

{ (float) → int, (char) → int }

The operator **[|bool]** has a type of

∅.

This approach is not really viable for large type systems, as the number of cases multiplies quickly.

4.7. Type variables

Oftentimes it is more effective to write the signatures of an operator using *type variables*. Type variables are denoted using capital letters near T. A type variable is a stand-in for "any type" unless its range is restricted. For instance, the array indexing operator **[]** has a signature of:

(array[T], int)→T

That is, it takes an array of any type as its first argument, and an integer as its second, and produces a result of the type that the array is made from. This is normal array indexing: if A is of type array[float], and you write something like "A[4]", you get back a float.

Type variables can be used inside type literals. For instance, we may consider the casting operator **[|]** to have a signature of:

(int, type T)→T

This is because it will take an integer and an explicit type T as operands, and give a result of type T. For example,

```
[68 | char]
```

Gives the result

```
D
```

However, this is not entirely true; if T is an array type, then casting does not have that signature. That is,

[68 | [bool]]

is a semantic error. So we would have to restrict the range of the variable, and say something like:

casting has a signature of (int, type T)→T for primitive types T.

Ideally, we would like to give the type of casting as

{ (S, type T) → T }

That is, casting should take an expression of any type S, and an explicitly specified type T, and convert the expression to the specified type. However, it does not do this for all pairs of types (S, T). One way to deal with this is to use the "such that" bar that is a standard part of set notation:

{ (S, type T) → T | (S, T) ∈{(int, float), (int, char), (float, int), (char, int)} }

If the signatures are stated this way, then they are generally listed out (e.g. in FunctionSignatures.java). Type variables are more useful when dealing with an operator that takes any type rather than a specific subset of types.

5. Promotion: Implicit type conversions

Pika-2 introduces implicit (unstated) type conversions, called *promotions*. Promotions are performed when the source has an operator whose operands do not match any signature of the operator.

A single operand may be promoted:

(1) from **char** to **int**,
(2) from **int** to **float**,
(3) from **int** to **rat**,

or using any sequence of the above conversions. (Here, the sequences are simply (1)(2), yielding **char** to **float**, and (1)(3), yielding **char** to **rat**.) This promotion (even if it is a sequence) is considered a single operation. (That is, the sequence (1)(3), for instance, is referred to as **a** promotion.)

The promotion digraph is the digraph with vertices corresponding to **char, int, float,** and **rat**, and with the three edges (1)-(3) above. A vertex **v** in the digraph is considered a *predecessor* of a vertex **w** if it is possible to reach **w** by a path of 0 or more directed edges in the digraph.

When dealing with an k-ary operator, we first check if the operands match a signature. If not, then we check if promoting one argument will allow a signature to match. If not, then we check if promoting two arguments will allow a match. If not, then there is no match (we do not proceed to consider three or more promotions).

Let us call a (or a set of) promotion(s) *matching* if applying them allows gives operand types that match a signature of the operator.

To check if there is one matching promotion, we check first if there is a matching promotion for the first argument. If there is more than one such promotion, we issue an error. If there is exactly one such promotion, that promotion is applied and the operands are considered a match. Otherwise, we continue.

Next we check if there is a matching promotion for the second argument. Then the third, etc. These are checked in the manner described above for promotion of the first argument.

If no single-operand promotion works, then we check for double-operand promotion. If there is more than one double-operand promotion that works, then issue an error. We can summarize this as:

Promotion	**Promotion(s)**
1	None
2	First operand
3	Second operand
	…
k+1	k^{th} operand
k+2	Any two operands

Where we proceed to check each promotion level from 1 to k+2 in turn. At any level we encounter,

1) If there are two or more matches to signatures:
   a) If there is one signature where all operands are predecessors (in the promotion digraph) to the corresponding operands in all the other matching signatures, we use that signature.
   b) Otherwise we issue an error.
2) If there is exactly one match, then we stop checking and use that promotion (or those promotions).
3) If there are no matches, we go on to the next level.

As an example of 1a), consider a situation where we have actuals of types **c** and **i**, and an operator with signatures **{(i, i) → r, (f, i) → r, (r, i) → r }**. Promotion of argument 1 can yield matches to all three of these signatures. However, **i** is a predecessor of **f** and **r**, so **(i, i)** has all operands predecessors to the corresponding operands in **(f, i)** and **(r, i)**. Therefore we would use the **(i, i) → r** signature.

For the purposes of promotion, assignment is considered an operator. Only expressions are promoted. Type literals are not promoted.

6. Rational numbers

6.1. Storage and manipulation

Rational numbers are a new primitive type in Pika. A rational number takes up eight bytes on the ASM; four for an integer numerator followed by another four for an integer denominator. When you "put a rational number on the accumulator", you put two integers: first the numerator, then the denominator. An operator with two rational operands will expect four integers on the accumulator. Calculating with these numbers requires temporary storage; this storage can be allocated statically, as it should not be needed during a recursive call. Another (slower) option is to use the *memory manager* to allocate temporary storage.

Any operation that yields a rat must at the end of the operation convert the numerator and denominator to lowest terms by dividing them by their gcd (greatest common divisor). Use any standard gcd algorithm to do this. You probably should have an ASM subroutine for converting to lowest terms, which can be accessed with the **Call** and **Return** opcodes. The gcd is performed to help avoid overflow on either of the integers.

6.2. Normal creation

There are no rational constants. Instead, rationals are created from two integers by the **//** operator. We call this operator "over". Be careful about using the terms *division* or *divided by* for **/** and *over* for **//**). Over has only one signature, which is **(int, int) → rat**. For instance,

```
3 // 4
```

Denotes a rational number with numerator 3 and denominator 4. The **//** operator has the same precedence as the normal division operator.

6.3. Existing operators

The operators **+**, **-**, **/**, and ***** all have the new signature **(rat, rat) → rat**.

Rational numbers can be cast to themselves, to floating, and to integer. The cast to floating should be accomplished by converting the numerator and the denominator to floating, and then doing floating division. The cast to integer truncates the result towards zero (use integer division for this).

Characters and integers can be cast to rational by using their value as the numerator and giving them denominator 1. Casting a float *f* to rational is equivalent to performing *f* **////**d, where d is the magic-value denominator **223092870**, which is the product of the first nine primes. (The **////** operator is defined below.) No other type can be cast to rational.

Admittedly, the definition of cast to rational is a joke, and careful pika programmers will avoid using it, preferring to choose their own denominator for **////**.

6.4. New operators

Along with over, two other new operators have been introduced for working with rationals: **///** (the "express over" operator) and **////** (the "rationalize" operator). Both have the same precedence as the other multiplicative operators.

The **///** operator is the "express over" operator. It has signatures **(rat, int) → int** and **(float, int) → int** . The result of *f* **///** *d* is the integer that would "best" express the number *f* when it is used as a numerator over the denominator *d*. More technically, the result is the largest-magnitude integer that is closer to zero than *n* in the equation:

$f = n / d$ (f is rational or floating)

That is,

$n = [d*f \mid \text{int}]$ (f is rational or floating)

where the cast is the normal float-to-int or rat-to-int conversion.

The expression f **////** d is a shorthand for

(f /// d) // d,

but where d is evaluated just once. As such this "rationalize" operator has the signatures **(rat, int) → rat** and **(float, int) →rat**. Note that the **//** does not merely put d in the denominator, it puts d in the denominator and then converts to lowest terms.

6.5. Printing a rational

To print a rational stored as n/d, print the whole part followed by an underscore followed by the fractional part expressed as a numerator followed by a slash followed by the denominator. An example should make this clear: suppose we have the rational with n=7 and d=4. Then we print this as:

1_3/4

and we read the underscore as "and". Be sure that negative numbers work as expected. Thus n=-40 and d=13 would be printed as

-3_1/13

When the fractional part is 0, then we print the integer part only. If n=44 and d=4, we would print

11

When the integer part is 0, then we print an underscore followed by the fractional part. So if n=7 and d=8, we print

_7/8

When one of n and d is negative, we print the minus sign before any of the rest of the number. So we would print

-_7/8 , -11, -3_1/13

7. Arrays

Arrays are the new compound type in Pika_2. The term *length,* when referring to an array, means the number of elements in the array.

7.1. Array expressions

There are three ways to create array records in Pika; these are the three options for *arrayExpression.*

7.1.1. Populated array creation

The first way to create an array record is to list the members of an array between square brackets:

[*expressionList* **]**

All expressions in the *expressionList* must be promotable to the same type. If there is more than one type that they are all promotable to, then promote the least amount possible to get to a common type. Currently, this means that if they are all promotable to int, rat, and float, then promote them all to int.

If there are n expressions in the list, each promoted to type T, then this syntax creates an array record with length n with elements of type T. The values of the expressions are assigned to the elements of the array, in order.

For example, the expression

```
[^s^, ^a^, ^i^, ^d^]
```

creates a 0-based array record with four characters, with the first character (index 0) being **s**, the second character (index 1) being **a**, etc.

One may not create a zero-length array with populated array creation:

```
var word = [];
```

is **not** legal Pika. (This is because we would not know the type of the array at this definition of it.)

Array elements are always mutable.

7.1.2. Empty array creation

The second way to create an array record is to give its type and length:

new *arrayType* **(** *expression* **)**

Here the expression must be of type int. This form creates an array record of the given type and length. The expression gives the length of the array, and indexing is 0-based.

For example,

```
new [char](14)
```

creates a character array record of length 14 with lower index 0 (cf. java "new char[14]").

If the length given to an empty array creation expression is negative, then a runtime error is issued. Zero-length arrays are permitted with empty array creation:

```
create[char](0)
```

Creates a zero-length array of characters with lower index 0.

7.1.3. The *clone* expression

The third way to create an array record is to make a copy of another array:

clone *expression*

Here the expression must have array type. This creates a new array record of the given type and length, copying the information from the result of the *expression*. For instance:

```
const B := clone A.
```

Makes an array record for B that is a copy of the array record for A.

In this example, the variable B is immutable, meaning it is a pointer that will always point at the record that is assigned to it in this statement. The record itself may change, but the pointer will not.

7.2. Array variables

Arrays are reference types, so a variable of type array[T] (remember, this is type-system notation, not Pika syntax notation) is a pointer to an array record, and thus the variable occupies 4 bytes. It does ***not*** occupy 16 + length*size(T) bytes.

Array variables, and any future reference-type variables, obey semantics much like java objects (which are themselves reference types):

Array variables declared with **const** do not change their pointer; however, the elements in the record they point at may change.

```
const R := [7, 5].
```

```
R[0] := 4.
```

is valid Pika-2 and results in R pointing at an array record that contains the elements 4 and 5. Assignment or initialization of arrays with other arrays is simply a pointer copy.

```
const R := [7, 5].
const S := R.
```

is valid Pika-2 and results in R and S being two pointers to the same record. If this is followed by

```
R[0] := 4.
```

then not only will R[0] be 4, but S[0] will, as well. Contrast this situation with **clone**.

Array variables declared with **var** may change their pointer as well as the elements in the record.

```
var R := [7, 5].
R[0] := 2.
…
R := new [int](5).
```

is valid Pika-2.

However, array variables may not be assigned an array of a different type.

```
const intArray := [7, 5].
const charArray := [`a, `z].

var R := intArray.
R := charArray.
```

is ***not*** valid Pika-2. The **var** declaration sets R's type as array[int], and the assignment tries to update it with an array[char], so this should generate a typechecking error.

7.3. Array indexing

The *array indexing expression* is of the form:

expression$_1$ **[** *expression*$_2$ **]**

Here, *expression*$_1$ must have type array[T] for some T, and *expression*$_2$ must be of integer type. The result of this expression is the *n*-th element of the array *expression*$_1$, where *expression*$_2$ evaluates to *n*. The type of the result is T. Thus **[ ]** (array indexing) has the signature **(array(T), int) → T** for any type **T**.

Whenever an array is indexed, the index must be checked for validity (i.e. it must be between 0 and the highest index in the array, inclusive). If the index is not valid, a runtime error is issued.

Array indexing expressions are targetable. In the code generator, generate *address code* for them.

7.4. Array length

The *length expression* is of the form:

length *expression*

The *expression* must have array type, and the result is the integer length (number of elements) of the array. Length thus has the signature **array(T) → int** for any type T.

7.5. Array printing

If an expression of array type is in the expressionList of a printing statement, then the array is printed as the following sequence:

[
A print of the first element
,
a space
a print of the next element
,
a space
…

A print of the last element
]

This is a quite natural way to print an array. For example, the array [1.23, 2.79, 5.41] is printed as

```
[1.23, 2.79, 5.41]
```

Note that there are only two spaces printed here, one located right after each comma.
The "print of the *n*th element" parts are done recursively. It is your problem to figure out when that recursion is done—at compile time or at run time.

7.6. Multidimensional arrays

Pika does not have true multidimensional arrays. Instead (like java), one must use arrays of arrays. An array with array elements must have all of those array elements of the same type, but they need not be of the same size. For instance, the following is legal:

```
const numSets := [ [1, 2, 3], [4, 5], [6, 7, 8], create[int](0)].
```

This makes numSets have type array[array[int]], with four elements. For empty array creation, the following is the proper idiom for a rectangular array:

```
const width := 4.
const height := 7.
const matrix := new [[float]](width).

var x := 0.
while(x < width) {
      matrix[x] := new[float](height).
      x := x + 1.
}
```

7.7. Array equality and inequality

The **==** and **!=** operators work on array types, provided the left and right operands are the same array types. More formally, **==** and **!=** both have the signature (array[T], array[T])→bool, for any type T. Two arrays are

equal iff they have the same record (i.e. equal pointers to the record), and they are inequal otherwise. This is the same as the java **==** and **!=** as they apply to objects.

7.8. Array casting

Array types may not be cast to other types, including other array types. No other type may be cast to an array type, although array types may be cast to themselves.

8. Records

In a running pika program, we will often have many pieces of heap memory that we need to process and keep track of. In order to do this, we will enforce a simple record format, as follows:

Type identifier (4 bytes)	Status (4 bytes)	Rest of record (?? bytes)

The type identifier is an integer describing what type is being stored in this record. Currently there are only two "types" with records: **string** and **array( )**. (Remember that **array( )** is not by itself a type.) String is given type identifier 6, and array is given type identifier 7.

The status field currently holds only four bits of data, in the lowest bits.

- The datum in bit 0 is the *immutability status* of the elements of the record (1 is immutable, 0 is mutable).
- The datum in bit 1 is the *subtype-is-reference status* of the array; this indicates whether the subtype T of the array is a reference type (i.e. if the subtype is itself an array or string type
- The datum in bit 2 is the *is-deleted status* of the record, which indicates that this record has been given to the memory deallocator.
- The datum in bit 3 is the *permanent status* of the record. Records with this bit set won't be released. If an attempt is made to release a permanent record, it is silently ignored. (It doesn't issue a runtime error).

8.1. String Records

The record for a string is changed in Pika-2 from the simple C-style characters & null record of Pika-1 to a record with some header information. The record is as shown below.

Type identifier (4 bytes)	Status (4 bytes)	Length (4 bytes)	Characters *length* bytes	Null character (1 byte)

The *type identifier* for a string is the integer 6. The status currently holds only four bits of data, in the lowest bits. When creating a static string (i.e. a string for a string constant), set

- The *immutability status* to 1. (Strings are immutable.)
- The *subtype-is-reference status* to 0. (There is no subtype for a string.)
- The *is-deleted status* of the record to 0.
- The *permanent status* to 1. We don't want to call the deallocator with a constant record.

Please note that this will necessitate some changes in the string duplication elimination part of your optimizer, and in the printing code for a string. Are there any other changes that need to be made? What operators take string arguments or give string results?

8.2. Array records

An *arrayExpression* results in the allocation of a new block of memory—a record—from the heap at runtime. The record for the type array[T] (informally, an *array record*) has the following format:

Type identifier (4 bytes)	Status (4 bytes)	Subtype size (4 bytes)	Length (4 bytes)	Elements ((subtypeSize * length) bytes)

The *type identifier* for an array is the integer 7. When creating an array, set

- The *immutability status* to 0. (Array elements are mutable.)
- The *subtype-is-reference status* according to the subtype of the array.
- The *is-deleted status* to 0.
- The *permanent status* to 0. Array records can be deleted.

The subtype size is the number of bytes consumed by a variable of type T; we will refer to this as size(T). The *length* is the number of elements in the array. (Note that the highest index in the array is *length*-1).

Over the lifetime of this record, only the elements and possibly the status flags may change. The other values will not.

To create an Array record, you will need to call the memory manager. This means you must enable the memory manager. This involves uncommenting one line (in makeASM( ) in the code generator), and adding another. Look at the public methods in MemoryManager.java to figure out what the added line should be (and then you must decide exactly awhere it should go). You do not need to understand the MemoryManager in detail; you only need to figure out its API.

8.3. Release of records

When an expression of reference type is the object of a **release** statement, then the record that the expression points to is subject to what I call an *explicit release*. For this assignment, this is the only way that allocated memory gets recycled.

If a record that is released has type array[T] for some T, then the elements in the record are recursively subject to explicit release. If T is an a value type, then this means nothing. If T is a reference type, then recurse on the elements if it is. (This is the purpose of keeping the *subtype-is-reference* status bit.)

When releasing, check that the *is-deleted-status* and *permanent-status* bits are zero before changing the record. If they are not both zero, abort the releasing of this record. This (heuristically) helps prevent the runtime system from double-freeing records.

To release, or recycle, a block of memory, first set its *is-deleted-status* bit. Then give the memory block back to the memory manager by calling its deallocate subroutine with the pointer to the block on the top of the stack.

This is a clumsy way of handling memory; it offloads the work to the pika programmers.

9. Operator precedence

The precedence of operators is

Highest precedence	parentheses, populated array creation empty array creation casting	**( ) [ ] new[]() [|]**
	array indexing	**[ ]**
(prefix unary operators are right-associative)	not, copy, length,	**! clone length**
	multiplicative operators	*** / // /// ////**
	additive operators	**+ –**
	comparisons	**< > <= >= == !=**
	and	**&&**
Lowest precedence	or	**||**

These are all left-associative operators, except as noted.

10. Optimizer

10.1. Basic blocks

In Pika-2, Your first task in the optimizer is to divide the instructions into *basic blocks*. A basic block is a maximum contiguous sequence of instructions such that:

- There is no jump or branch from anywhere in the code (including within the block) to any instruction in the block except for the first one. (The only way into the block is through the first instruction.)
- Every instruction of the block gets executed (in order) each time the first instruction gets executed.

This means that a basic block will start at the first instruction, at any jumped-to instruction, or any instruction after a branch. A basic block will always end with a jump, a branch, or a halt.

We'll modify this definition a little, to allow a Jump opcode after a branch opcode at the end of a block. Essentially, these two opcodes together specify the places to jump to depending on some condition being true or not.

Keep Label directives with the basic block that follows them.

For example, the ASM code of figure 1 has been divided into basic blocks according to our (modified) definition. The code in figure 1 is the (abbreviated) ASM for the pika-2 code of figure 2. In figure 1, each basic block has been numbered, from BB1 through BB17.

10.2. Control Flow Graph

Once you have the basic blocks, you should construct the *control flow graph (CFG)* for them. Here, each basic block becomes a node in the graph. The edges of the graph are the transitions out of the basic blocks. If a basic block has only a Jump (without a preceding branch) out, then it gets one outedge to the basic block that the jump leads to. If it has no jump or branch at the end, then it gets one outedge to the block that follows it. If it has a branch at the end (with or without a following Jump), then it gets two outedges, one to the basic block starting with the branch target label, and one to either the following Jump target or to the next block if there is no following jump. Also, the

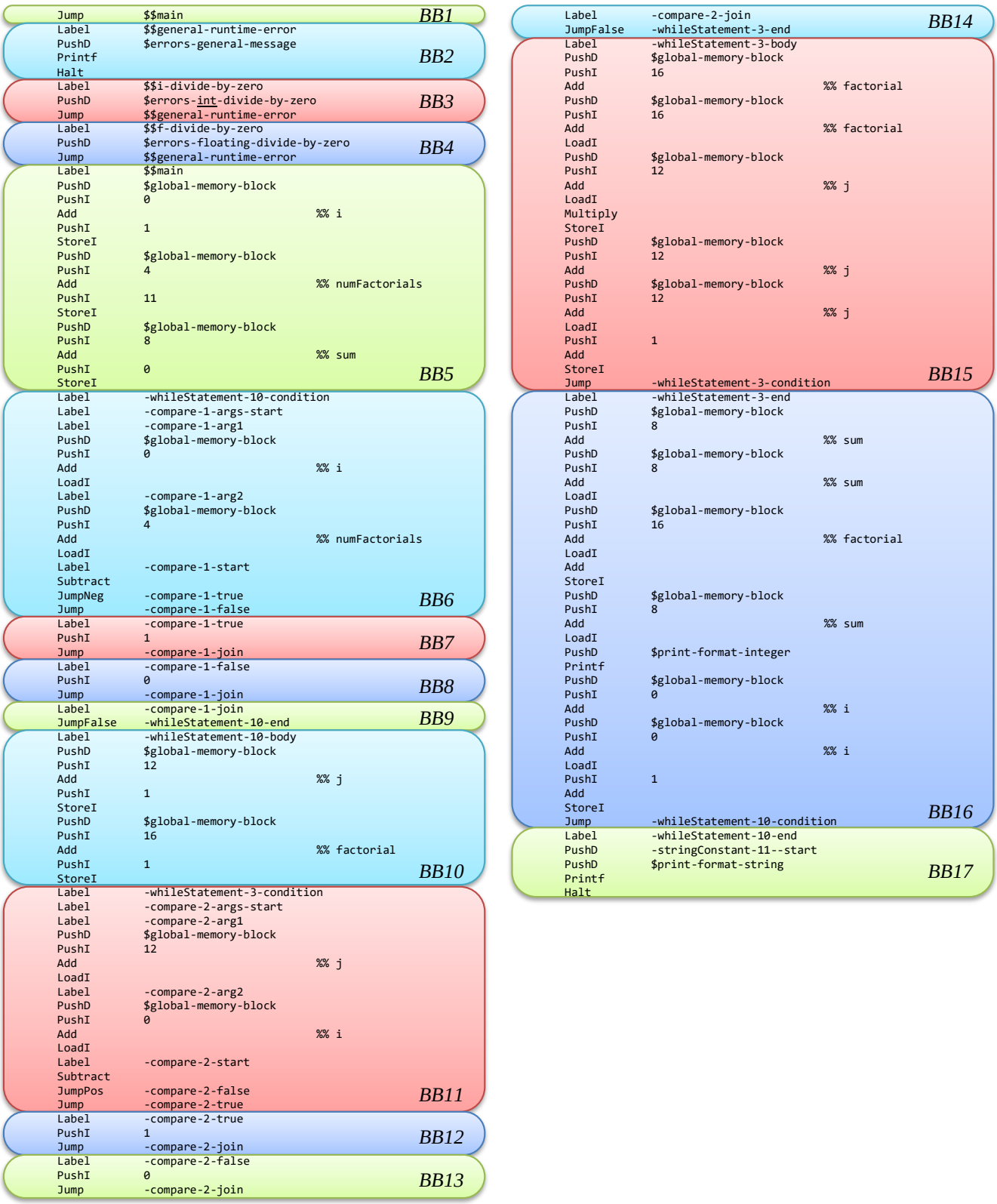


Figure 2. ASM code divided into basic blocks.

```
exec {
	var i := 1.
	const numFactorials := 11.
	var sum := 0.

	while (i < numFactorials) {
		# compute the factorial of i
		var j := 1.
		var factorial := 1.

		while (j <= i) {
			factorial := factorial * j.
			j := j + 1.
		}

		# ... and add it to sum
		sum := sum + factorial.

		# print "sum of first "; i; " factorials is "; sum, _n_.
		i := i + 1.
	}
	print "done", _n_.
}
```

Figure 2. Pika source for ASM code shown in Figure 1.

condition under which the branch is taken is noted. (For example, if the branch were a JumpFZero, then "on floating zero" would be recorded. Figure 3 shows the CFG for the blocks in Figure 1.

Knowing the directed edges and branch conditions on the CFG, one can trim each basic block by removing any Label directives at the front and any Jump and/or branch instructions at the end. For instance, basic block 6 in the example would become just:

```
PushD	$global-memory-block
PushI	0
Add			%% i
LoadI
Label	-compare-1-arg2
PushD	$global-memory-block
PushI	4
Add			%% numFactorials
LoadI
Label	-compare-1-start
Subtract
```

In the optimizer, you should create an object for each basic block and its node in the CFG. It should hold the in-neighbors and out-neighbors of the CFG node, the condition if there is more than one outedge from the node, and either all instructions/directives in the block or the trimmed form above. Also note which basic block is the starting block for the program (as we have done with the inarrow on BB1 in the example). You probably also want an object that represents the entire CFG (it could, for instance, hold a list of the basic blocks and know the starting block).

10.3. Unreachable code

One optimization that you can do on a CFG is to eliminate *unreachable* code. This is code that is not reachable from the start of the CFG in the standard directed graph sense. To accomplish this, one can simply do a depth-first search in the CFG from the start block. Any node (basic block) that this depth-first search doesn't encounter is unreachable and can be removed.

In the example, the basic blocks BB2, BB3, and BB4 are unreachable. These were the blocks that held the code for reporting and handling runtime errors; as this particular code had no possibility of encountering those runtime errors, the code was unnecessary.

Eliminating unreachable code is a basic optimization that may be run several times during optimization of a single file. This is because other optimizations that restructure the CFG may create unreachable blocks. Be sure to create your unreachable code eliminator in a way that makes it easy to invoke several times.

10.4. Block merging

A useful simplification on a CFG is to merge two nodes X and Y when X has exactly one outneighbor (Y) and Y has exactly one inneighbor (X). If there is a Jump at the end of X, then eliminate it in the merged node. Similarly, if there are any labels at the beginning of Y, they can be eliminated. In our example, BB1 and BB5 are the only pair that can be merged. Write a block merging optimization; again, this is often invoked more than once during optimization.

10.5. Cloning to simplify conditionals

In our text, there is a section on postional encoding for booleans. In pika, we're not doing that in the code generator, but we will accomplish the same effect in the optimizer.

Suppose that we have an empty block (empty *trimmed* block) X with two outneighbors, and the branch at the end of the block is on true or false. Furthermore, X has two or more inneighbors Y_1, Y_2, … Y_k, and all of these inneighbors have one outneighbor. This situation arises in loops and in conditionals when dealing with boolean values. An easy optimization here is to make a clone (copy) X_i of X for each inneighbor Y_i. Then let Y_i have outneighbor X_i. X_i has the same outneighbors as X did, and same condition. The situation is shown on the left of Figure 5, and the transformed code is shown on the right.

The vast majority of the time, k, the number of inneighbors of X, will be two. This case is illustrated in Figure 4, with the transform from part (a) to part (b).

This may not seem like much, and in fact may seem to be creating code bloat, but like many "optimizations," this one is performed mainly to *expose* other chances to optimize. For instance, in the resulting code we may merge (section 10.4) each Y_i with its corresponding X_I (see Figure 4(c).) Then we may apply branch simplification (section 10.6) to the resulting blocks, and if any block simplifies, then we have eliminated some branching (see Figure 4(d)), which oftentimes leads to faster code. Finally, after eliminating branching, we may be able to merge the resulting blocks, as shown in Figure 4(e). (In fact, this should usually happen at the implementation of pika conditionals—why?)

There are two opportunities to apply this optimization in our example: one where BB9 plays the role of X, and one where BB14 plays that role.

You should write code to do this cloning, and ensure that your merging and simplification passes are called often enough to reduce conditionals as in Figure 4.

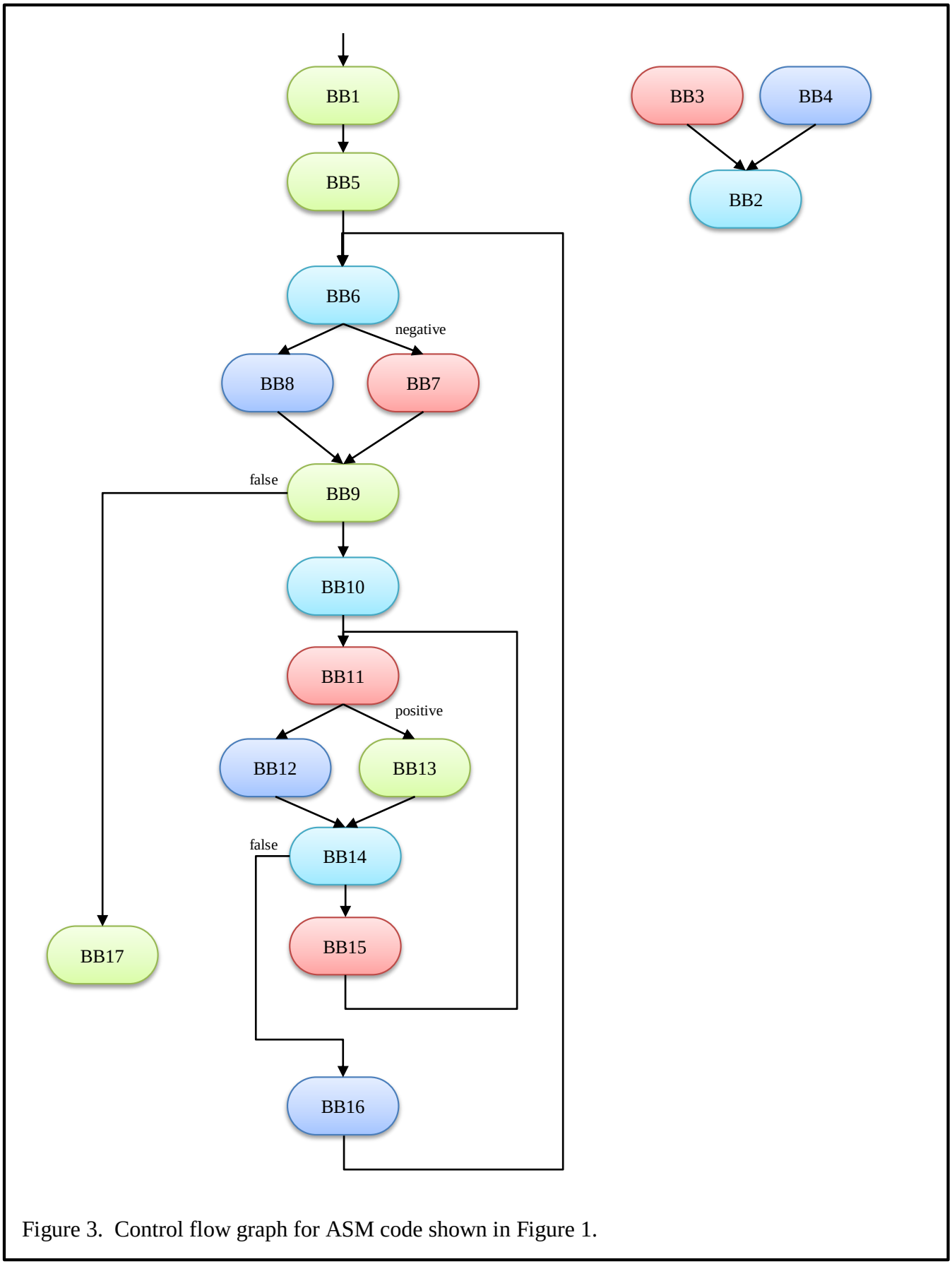


Figure 3. Control flow graph for ASM code shown in Figure 1.

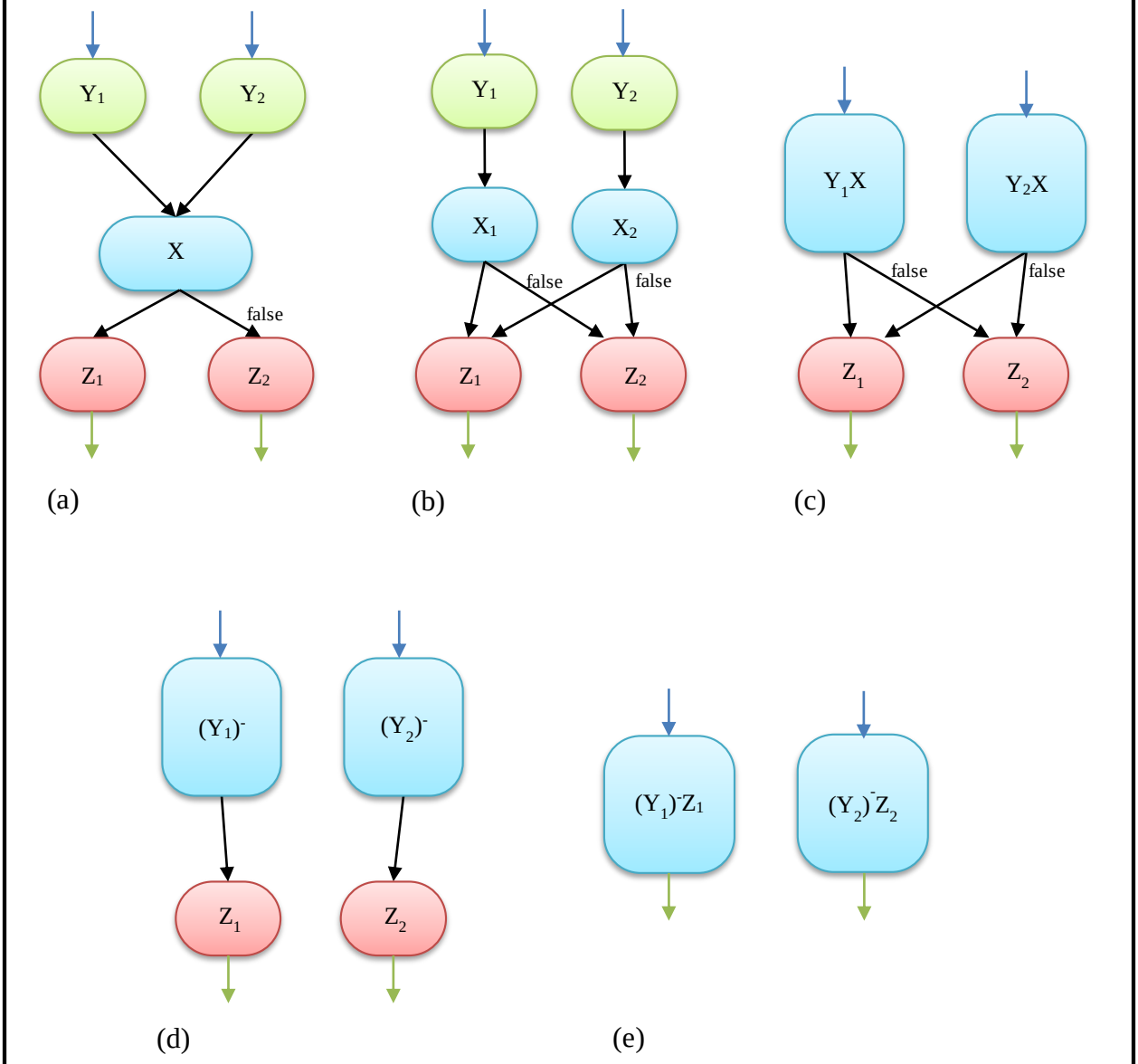


Figure 4. Cloning an empty decision block. (a) original structure. (b) after cloning X. (c) after merging Y_1 with X_1 and Y_2 with X_2. (d) structure when the branches simplify. (e) merging nodes after (d).

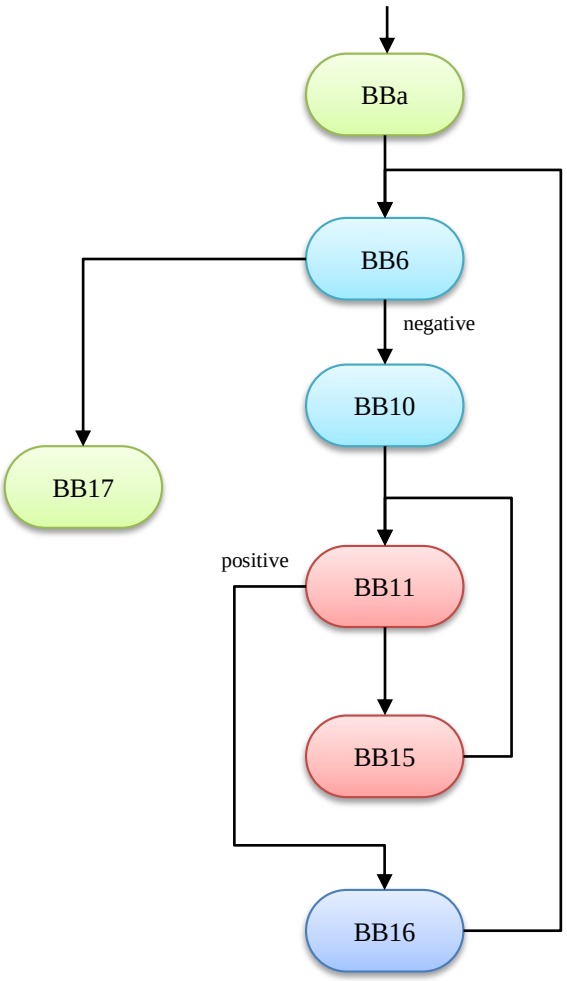


Figure 5. CFG of Figure 1 after optimization.

10.6. Branch simplification/elimination

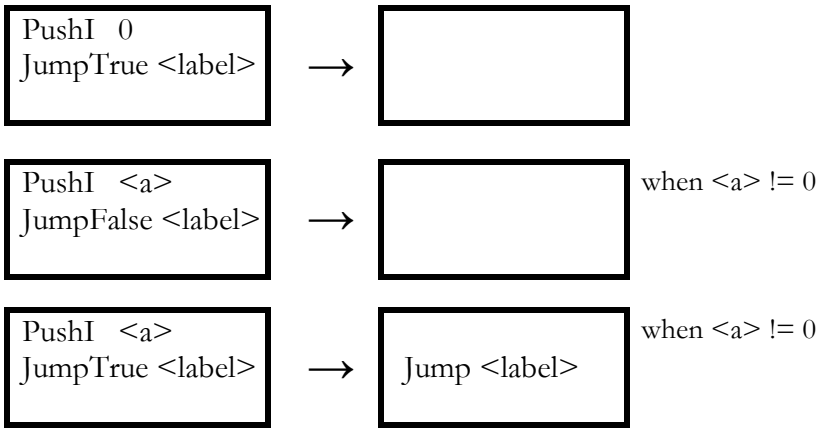
The main idea here is this: if we come to a branch (conditional jump), and we know which of the two options the program will take, then we convert the branch to a jump to that option. Here I want you to only consider those branches with a PushI or PushF instruction immediately before them.

We can consider these to be peephole optimizations; for instance

```
PushI  0
JumpFalse <label>
```

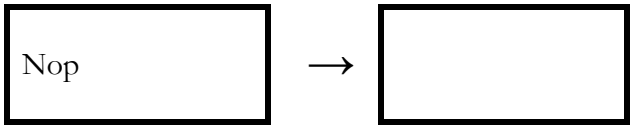
→

```
Jump <label>
```



The empty right-hand boxes in the second and third reductions indicate that the instructions in the left-hand box are simply deleted; they have no effect.

Consider every branch instruction and either PushI or PushF before it as appropriate; I won't list out all of the options here...that's your job. Oh, and implement this one little peephole optimization that you should apply liberally whenever you like:



The branch-simplification optimizations are probably best done when the code is in CFG form, but they can be done on the usual linear form of ASM. If done on the linear form, you may have to convert back and forth between linear and CFG forms a few times.

10.7. Code layout

Eventually, the optimizer will be done transforming the CFG and will need to convert the final CFG into linear code. There are techniques for trying to get a good such conversion (or *layout*) and you can look into that if you'd like but I won't require it in your code. For layout, use any technique you'd like, but you should relabel the basic blocks "basicBlock-#" counting up starting at 1. (For example, in Figure 5, we would rename the blocks basicBlock-1, basicBlock-2, … basicBlock-7, and use these labels (and these labels only) in the optimized ASM. I will be checking how many basic blocks your optimizer yields on some inputs.