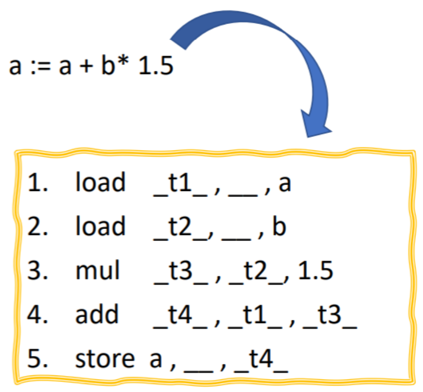
PPL 1/27

Reagan Shirk

January 27, 2020

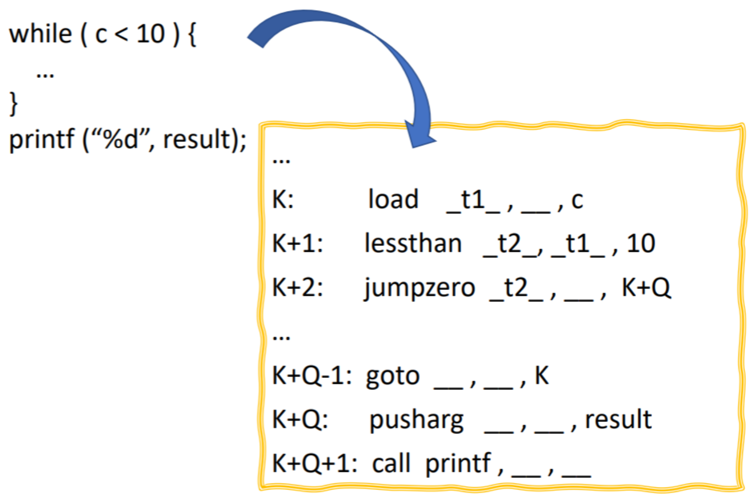
Intermediate Code Generation

- Sorry I was **so** zoned out
- The goal is to construct an intermediate representation of the program
  - Kinda looks like the assembly we did in computer organization
- Representation is machine/target independent
- Compiler can have multiple intermediate representations (IRs)
- You want to optimize a bit of the code at the end of the day
- It's interleaved with scanning, parsing, and semantic analysis
- Uses a symbol table and creates temporary variables
- High-level program constructs decomposed into simpler operations, like `for` = `JUMP` with operations



```
a := a + b* 1.5
```

```
1.  load   _t1_ , __ , a
2.  load   _t2_, __ , b
3.  mul    _t3_ , _t2_, 1.5
4.  add    _t4_ , _t1_ , _t3_
5.  store  a , __ , _t4_
```

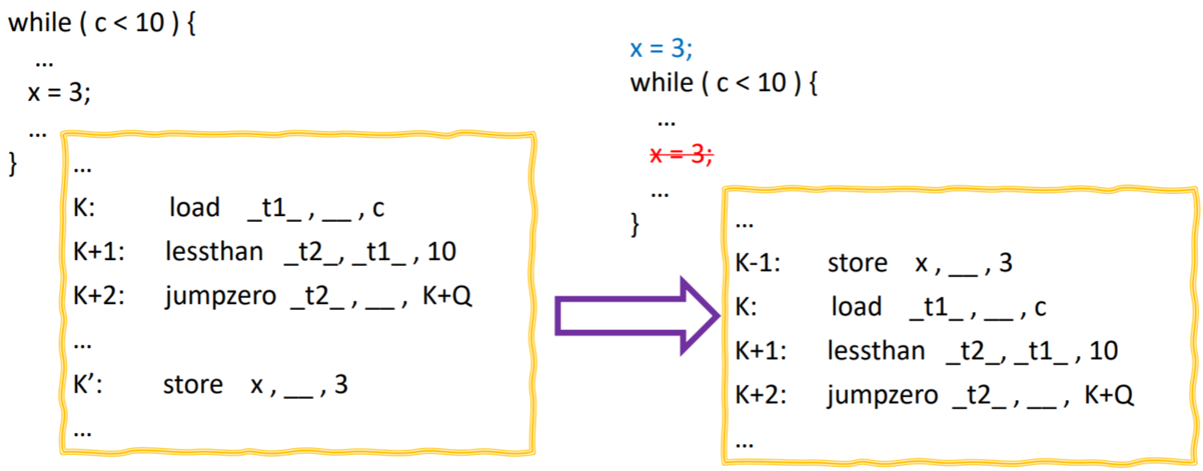


```
while ( c < 10 ) {
   ...
}
printf (“%d”, result);
```

```
...
K:        load     _t1_ , __ , c
K+1:      lessthan   _t2_, _t1_ , 10
K+2:      jumpzero  _t2_ , __ ,  K+Q
...
K+Q-1:  goto  __ , __ , K
K+Q:      pusharg   __ , __ , result
K+Q+1:  call  printf , __ , __
```

Machine Independent Optimizations

- Performing transformations that are independent of the target machine
  - removal of redundant stores and such
  - arithmetic simplification
- Overall the goal is to:
  - Avoid redundant work
  - Use cheaper operations
  - Eliminate unused code



```
while ( c < 10 ) {
  ...
  x = 3;
  ...
}
```

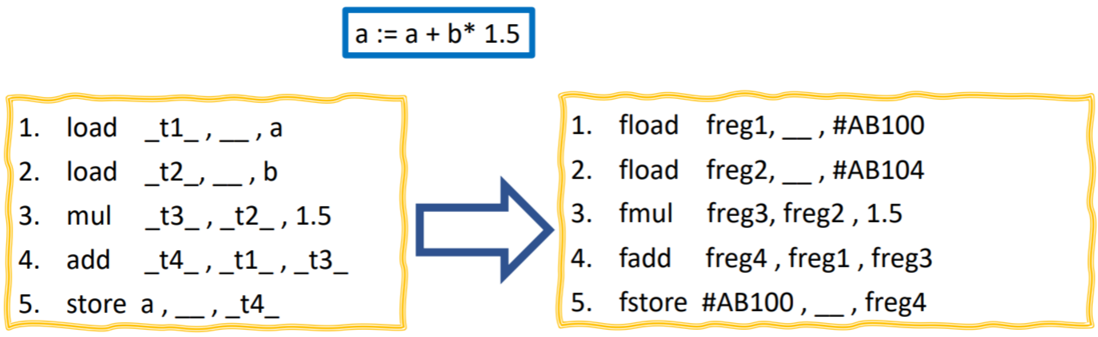
```
...
K:      load      _t1_ , __ , c
K+1:    lessthan  _t2_, _t1_ , 10
K+2:    jumpzero  _t2_ , __ ,  K+Q
...
K':     store   x , __ , 3
...
```

```
x = 3;
while ( c < 10 ) {
  ...
  ~~x = 3;~~
  ...
}
```

```
...
K-1:    store    x , __ , 3
K:      load     _t1_ , __ , c
K+1:    lessthan  _t2_, _t1_ , 10
K+2:    jumpzero  _t2_ , __ ,  K+Q
...
```

Machine Code Generation

- Machine specific
- Takes intermediate code generation into specific machine code



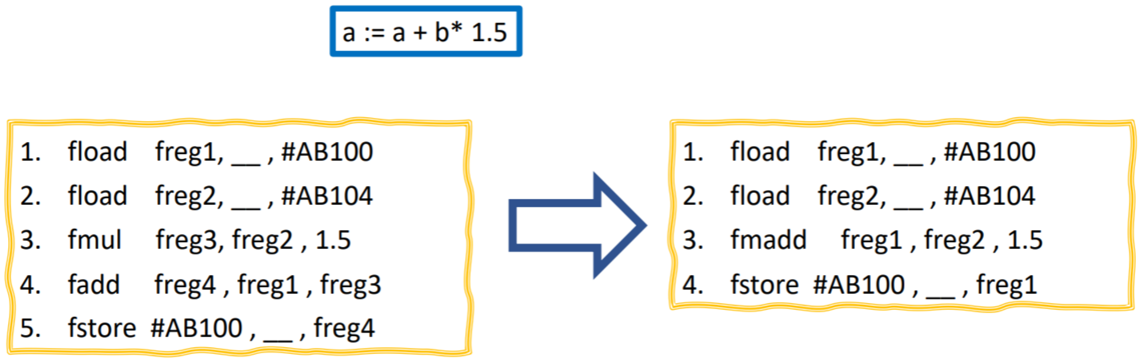
```
a := a + b* 1.5
```

```
1.  load   _t1_ , __ , a
2.  load   _t2_, __ , b
3.  mul    _t3_ , _t2_ , 1.5
4.  add    _t4_ , _t1_ , _t3_
5.  store  a , __ , _t4_
```

```
1.  fload   freg1, __ , #AB100
2.  fload   freg2, __ , #AB104
3.  fmul    freg3, freg2 , 1.5
4.  fadd    freg4 , freg1 , freg3
5.  fstore  #AB100 , __ , freg4
```

Machine Specific Optimization

- Removes redundancy from the machine code generation
- Goal is to generate more efficient code



```
a := a + b* 1.5
```

```
1.  fload   freg1, __ , #AB100
2.  fload   freg2, __ , #AB104
3.  fmul    freg3, freg2 , 1.5
4.  fadd    freg4 , freg1 , freg3
5.  fstore  #AB100 , __ , freg4
```

```
1.  fload   freg1, __ , #AB100
2.  fload   freg2, __ , #AB104
3.  fmadd    freg1 , freg2 , 1.5
4.  fstore  #AB100 , __ , freg1
```

Lexical Analysis

- This deals with “low level” syntactic structure
  - For the first quiz, I always remembered it as dealing with anything that wasn’t machine/hardware specific
- Strings with the same structure are put into the same class
- Implemented as a scanner that is invoked by a parser
- Can be written by hand or by using a scanner generator with regular expressions
- Lexical Analysis’ Job: to assemble an arbitrary stream of characters into strings (called lexemes) recognizable by the language
  - This is called source tokenization
- It removes comments
- Stores the actual values of:
  - identifiers
  - numbers
  - literal strings
- Recording location information is used for possible error reporting

Regular Expressions

- I had greek guy so I mostly understand this stuff yay
- A regular expression is:
  - A character
  - The empty string, ϵ
  - The concatenation of two regular expressions
  - Two regular expressions separated by **or**
  - A regular expression followed by the Kleene star *
- The whole point of regular expressions is to be able to identify strings