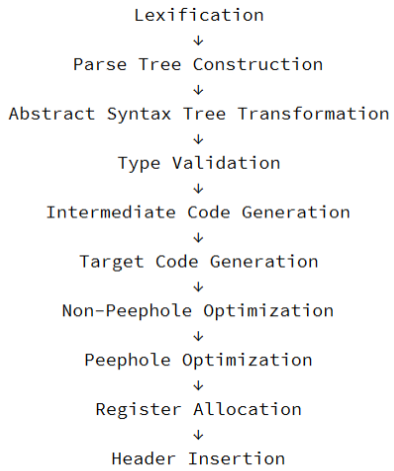
CSSE404 Presentation

Matthew Lyons & Karl Reese

Data Flow



Lexification

- Each Token type has a corresponding Matcher
- Match tokens using regular expressions
- Shown below: CommentMatcher

```
@Override
public boolean match(String _tokenString)
{
    return _tokenString.matches("//.*|(?s)/\\*.*?\\*/");
}
```

Parse Tree Construction

Test.java

```
Root
 ExitLabel
  Exit
   class
   Test
   {
   public
   static
   void
   main
   (
   String
   [
   ]
   args
   )
   {
   Stmt
    System.out.println
    (
    Exp1
     Exp2
      Exp3
       Exp4
        Exp5
         Exp6
          Factor
           int 2
          Exp6Prime
         Exp5Prime
          int +
          Exp6
           Factor
            int 13
           Exp6Prime
          Exp5Prime
        Exp4Prime
       Exp3Prime
      Exp2Prime
     Exp1Prime
    )
    ;
   }
   }
 ClassDeclList
```

- Top-down table-driven parser using an LL(1) grammar
- Read in grammar table from .txt file on initialization
- Create symbol tables
  - Cheap scoping: each symbol table has a pointer to its parent table
- Panic button error recovery

https://mikedevice.github.io/first-follow/

http://jsmachines.sourceforge.net/machines/ll1.html

—

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  -
  -
  -
  -
  -
- Puts tree into a form that is easier to generate intermediate code for

—

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  - “addLastChildTo”
  -
  -
  -
  -
- Puts tree into a form that is easier to generate intermediate code for

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  - “addLastChildTo”
  - “murderIfs”
  -
  -
- Puts tree into a form that is easier to generate intermediate code for

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  - “addLastChildTo”
  - “murderIfs”
  - “gentlyRemoveStatementListsAndFriends”
- Puts tree into a form that is easier to generate intermediate code for

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  - “addLastChildTo”
  - “murderIfs”
  - “gentlyRemoveStatementListsAndFriends”
  - “ultraviolentlySlaughterizeArgListInParticular”
- Puts tree into a form that is easier to generate intermediate code for

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    +
     2
     13
```

Abstract Syntax Tree Transformation

- Deep Copy the parse tree on initialization
- Run a sequence of many small but mighty transformation functions on the parse tree
  - “addLastChildTo”
  - “murderIfs”
  - “gentlyRemoveStatementListsAndFriends”
  - “ultraviolentlySlaughterizeArgListInParticular”
  - “superUltraOmegaviolentlyMurderSlaughterizifyArgListPrimeInParticularAndSpecifically”
- Puts tree into a form that is easier to generate intermediate code for

Type Validation

Test.java

```
ExitLabel
 Exit
  Stmt
   System.out.println
    int +
     int 2
     int 13
```

- Expressions and subexpressions
  - Postorder traversal

```
+       |   +           |   int +
    a   |       int a   |           int a
    3   |       int 3   |           int 3
```

- Function parameters
  - Look up types in symbol table

Intermediate Code Generation

Test.java

```
load [2] [R0]
load [13] [R1]
+ [R0, R1] [R2]
print [R2] []
```

- Postorder traversal of AST
  - Creates an instruction for most nodes
- Extremely naive register and label allocation
  - Numbers assigned sequentially, starting from 0
- Also inserts some “non-instructions”
  - PRECALL/POSTCALL
  - PROLOG/EPILOG
  - nope

Target Code Generation

Test.java

```
mov     R0, 2
mov     R1, 13
mov     EBX, R0
add     EBX, R1
mov     R2, EBX
push    R2
call    printInt
add     ESP, 4
jmp     EXIT
EXIT:
```

- Convert instructions to work with x86 (32-bit)
- Still assumes an infinite number of registers
  - We use x86's EBX, ESI, ETC registers to store temporary values

Non-Peephole Optimization

- AST construction inherently optimizes to some extent
- After code generation, we implement one more non-peephole optimizer
- Removes instructions where the destination register is never used as an input
  - "UnusedDestDestroyer"
  - Removes ***29*** instructions from LinkedList.java

Peephole Optimization

- Variety of peephole optimizers
- Sliding window of 3
- Removes ***90*** instructions from LinkedList.java

```
private void populatePeepholeOptimizers()
{
    this.peepholeOptimizers.add(new PingpongOptimizer());
    this.peepholeOptimizers.add(new RedundantMovRemover());
    this.peepholeOptimizers.add(new OpImmediateOptimizer( instName: "add"));
    this.peepholeOptimizers.add(new OpImmediateOptimizer( instName: "sub"));
    this.peepholeOptimizers.add(new OpImmediateOptimizer( instName: "or"));
    this.peepholeOptimizers.add(new OpImmediateOptimizer( instName: "and"));
    this.peepholeOptimizers.add(new MulImmediateOptimizer());
    this.peepholeOptimizers.add(new XorifierOptimizer());
    this.peepholeOptimizers.add(new IdentityOpRemover( instName: "add", constant: "0"));
    this.peepholeOptimizers.add(new IdentityOpRemover( instName: "sub", constant: "0"));
    this.peepholeOptimizers.add(new IdentityOpRemover( instName: "mul", constant: "1"));
    this.peepholeOptimizers.add(new CrisscrossOptimizer( instName: "add"));
    this.peepholeOptimizers.add(new CrisscrossOptimizer( instName: "sub"));
    this.peepholeOptimizers.add(new CrisscrossOptimizer( instName: "or"));
    this.peepholeOptimizers.add(new CrisscrossOptimizer( instName: "and"));
}
```

Register Allocation

Test.java

```
push    EBP
mov     EBP, ESP
sub     ESP, 8
mov     dword [EBP - 4], 2
mov     EBX, [EBP - 4]
add     EBX, 13
mov     dword [EBP - 8], EBX
push    dword [EBP - 8]
call    printInt
add     ESP, 4
jmp     EXIT
EXIT:
leave
```

- Haha, registers are a lie
- Everything lives on the stack
  - Uses architectural registers occasionally since memory-to-memory is impossible
- dword

```
this.ruthlesslyslaughterCivilianInstructions();
this.dword();
```

Header Insertion

Test.java

```
%include "library.asm"

global start
section .text
start:
    push    EBP
    mov     EBP, ESP
    sub     ESP, 8
    mov     dword [EBP - 4], 2
    mov     EBX, [EBP - 4]
    add     EBX, 13
    mov     dword [EBP - 8], EBX
    push    dword [EBP - 8]
    call    printInt
    add     ESP, 4
    jmp     EXIT
EXIT:
    leave
    call    exit
```

- Designed to work with NASM
- Imports assembly “library” containing print, malloc, and exit routines
- Default supports macOS syscalls, but can be easily extended to other environments

Fibonacci

- Fib.ComputeFib(40) runs in 1.123 seconds
- Fib.ComputeFib(44) runs in 7.642 seconds
- Fib.ComputeFib(45) runs in 12.374 seconds

Questions?

Too bad!