

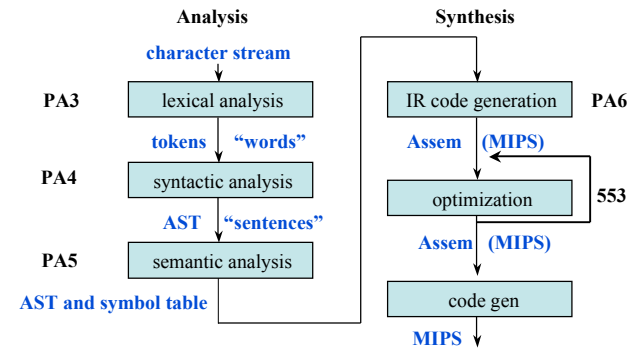
CS 453: Compiler Construction Review

Phases of the compiler

- lexicographical analysis, or scanning (regular expressions)
- syntactic analysis, or parsing (context free grammars)
 - building the abstract syntax tree (syntax-directed translation)
- building the symbol table (visitor design pattern)
- semantic analysis, or type checking (visitor design pattern)
- code generation (visitor design pattern)
 - 3-address code
 - Assem(MIPS)

How would adding floats to the MiniJava compiler affect each phase?

Structure of the MiniJava Compiler



Specifying Tokens with JFlex

JFlex example input file:

```
package mparser;
import java_cup.runtime.Symbol;

%%
%line
%char
%cup
%public

%eofval{
    return new Symbol(sym.EOF, new
    TokenValue("EOF", yyline, yychar));
%eofval}
```

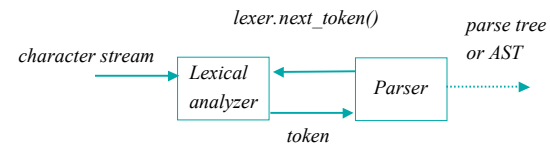
```
LETTER=[A-Za-z]
DIGIT=[0-9]
UNDERSCORE="_"
LETT_DIG_UND={LETTER}|{DIGIT}|{UNDERSCORE}
ID={LETTER}{LETT_DIG_UND}*

%%
"&&" { return new Symbol(sym.AND, new
    TokenValue(yytext(), yyline, yychar)); }

"boolean" {return new
    Symbol(sym.BOOLEAN,...

{ID} { return new Symbol(sym.ID, new ...
```

Interaction Between Scanning and Parsing



Specifying Grammar with JavaCUP

JavaCUP example input file:

```
package mparser;
import java_cup.runtime.*;
import ast.node.*;
...
terminal AND, ASSIGN, INT;
terminal mparser.TokenValue
NUMBER;
...
non terminal Program program;
...
non terminal List<ClassDecl>
class_decl_list;
non terminal MainClass
main_class;
```

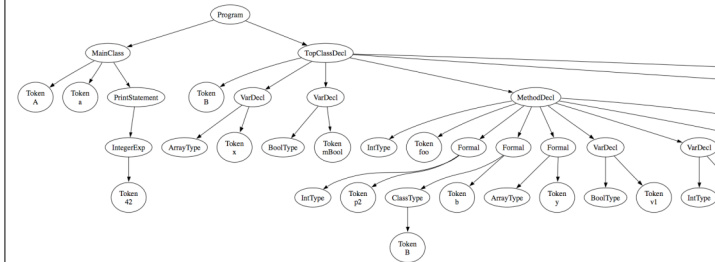
start with program;

```
program ::=
main_class:m class_decl_list:l
{: RESULT = new Program(m,l); :}
;

exp ::=
| NUMBER:n
{: Token token = new Token(n.text,
n.line, n.pos);
RESULT = new IntegerExp( token );
:}
...

```

Abstract Syntax Tree for Memory Layout Example



Semantic Analysis

Determine whether source is meaningful

- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
 - Relate variable uses to their declarations

Example errors (from C)

```
function1 = 3.14159;
x = 570 + "hello, world!"
scalar[i]
```

Compiler Data Structures

Symbol Tables

- Compile-time data structure
- Holds names, type information, and *scope* information for variables

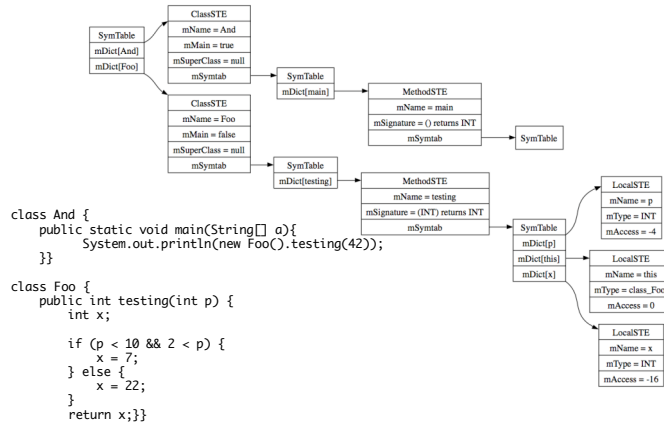
Scopes

- A name space
 - e.g.*, In Pascal, each procedure creates a new scope
 - e.g.*, In C, each set of curly braces defines a new scope
- Can create a separate symbol table for each scope
- What are the scopes in MiniJava?

Using Symbol Tables

- For each variable declaration:
 - Check for symbol table entry
 - Add new entry; add type info
- For each variable use:
 - Check symbol table entry

Example Symbol Table



CS453 Lecture

Final Review

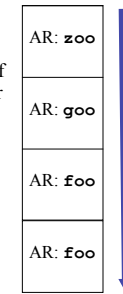
9

Compiling Procedures

Properties of procedures

- Procedures/methods/functions define scopes
- Procedure lifetimes are nested
- Can store information related to **dynamic invocation** of a procedure on a call stack (*activation record* or AR or stack frame):
 - Space for saving registers
 - Space for passing parameters and returning values
 - Space for local variables
 - Return address of calling instruction

higher addresses



Stack management

- Push an AR on procedure entry (caller or callee)
- Pop an AR on procedure exit (caller or callee)
- Why do we need a stack?

lower addresses

stack

CS453 Lecture

Final Review

10

Stack Frame for MiniJava Compiler

```

int foo(int x,int y,int *z) {
    int a;
    a = x * y - *z;
    return a;
}
void main() {
    int x;
    x = 2;
    cout << foo(4,5,&x);
    cout << "\n";
}
    
```

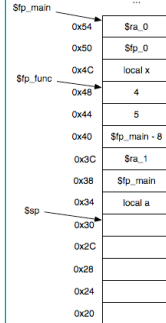
```

.text
.foo:
sw $ra, 0($sp) #PUSH
subu $sp, $sp, 4
sw $fp, 0($sp) #PUSH
subu $sp, $sp, 4
addu $fp, $sp, 20
subu $sp, $sp, 24
...
lw $t0, -20($fp)
move $v0, $t0
lw $ra, -12($fp)
move $t0, $fp
lw $fp, -16($fp)
move $sp, $t0
jr $ra
    
```

```

.text
-globl main
main:
sw $ra, 0($sp) #PUSH
subu $sp, $sp, 4
sw $fp, 0($sp) #PUSH
subu $sp, $sp, 4
addu $fp, $sp, 8
subu $sp, $fp, 12
li $t0, 2
sw $t0, -8($fp)
li $t0, 4
sw $t0, 0($sp) #PUSH
subu $sp, $sp, 4
li $t0, 5
sw $t0, 0($sp) #PUSH
subu $sp, $sp, 4
sw $t0, 0($sp) #PUSH
subu $sp, $sp, 8
sw $t0, 0($sp) #PUSH
subu $sp, $sp, 4
jal .foo
move $a0, $v0
...
lw $ra, 0($fp)
move $t0, $fp
lw $fp, -4($fp)
move $sp, $t0
jr $ra
    
```

Stack frame for funcCall1.c



return value is put in \$v0
\$sp is set to current \$fp before return

Final Review

Assem intermediate representation

Assem.Instr

- "assembly language instruction without register assignments"

OPER(String assem, List<Temp> dst, List<Temp> src, List<Label> jumps)

- contains a string with holes for registers indicated by `d#` and `s#` and holes for labels indicated by `j#`
- dst and src are lists of Temps whose register assignment should fill holes
 - first entry in src is associated with `s0`, second with `s1`, etc.
 - first entry in dst is associated with `d0`, etc.
- jumps is a list of labels for filling in label holes

CS453 Lecture

Final Review

12

Assem intermediate representation cont ...

LABEL(String assem, Label label)

- a label statement in the target code

MOVE(String assem, Temp dst, Temp src)

- similar to OPER in that assem string contains holes, but ..
 - no jumps
 - only one src and dst Temp

CJUMP(String a, Temp.Temp src1, RELOP op, Temp.Temp src2, Temp.Label t, Temp.Label f)

- similar to OPER in that assem string contains holes, but ..
 - only jumps to true and false target
 - only two source Temps for comparison
 - explicit conditional operation, which enables later changes in code layout

Instruction selection for x86-64

Registers

- 16 64-bit registers
 - RSP, the stack pointer register
 - RBP, the frame pointer register
- RBP, the frame pointer register
- 32-bit register names used to access lower 32-bits of corresponding 64-bit register: `eax, edx, ecx, ebx, esi, esi, edi, esp` and `ebp`

Representations

- Constants prefixed with '\$', for example \$3, \$4, \$-5, etc
- Registers prefixed with '%', for example %rsp, %rbp, etc.

Some Instructions

- `movl %eax, -12(%rbp)` // `M[%rbp-12] = %eax`
- `addl -20(%rbp), %eax` // `%eax = M[%rbp-20] * %eax`
- `cmpl -4(%rbp), %eax`
- `jge .L2` // if (`M[%rbp-4] >= %eax`) goto .L2

x86-64 example

```
.file "funcCall2.c"
.text
.globl foo
.type foo, @function
foo:
pushq %rbp          # %rsp = %rsp-8; M[%esp] = %rbp
movq %rsp, %rbp    # %rbp = %rsp
movl %edi, -20(%rbp) # storing parameters to stack
movl %esi, -24(%rbp)
movl -24(%rbp), %eax # accessing x
addl -20(%rbp), %eax # adding y to x and storing in %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
.size foo, .-foo
```

Example continued...

```
.globl main
.type main, @function
main:
pushq %rbp          # %rsp = %rsp-8; M[%esp] = %rbp;
pushes onto stack
movq %rsp, %rbp    # %rbp = %rsp
subq $16, %rsp     # %rsp = %rsp - 16
movl $5, %esi      # %esi = 5
movl $4, %edi      # %edi = 4
call foo
movl %eax, -4(%rbp) # M[%rbp-4] = %eax
movl -4(%rbp), %eax # %eax = M[%rbp-4]
leave
ret
```