Rudimentary Compilation (Objectives)

- Given a simple language of expressions, the student will be able to write a simple compiler for those expressions in order to give semantics to the language.

1

Compilers and Syntax

- Compilers give meaning to syntax.
  - What does the following syntax mean?

```
x = 1 + 2 *3;
```

- To start, we will write a compiler for a simple calculator language.

$$
\begin{array}{lcl}
Calc & \rightarrow & Calc;E \\
 & | & E \\
E & \rightarrow & E+T \\
 & | & E-T \\
 & | & T
\end{array}
\qquad
\begin{array}{lcl}
T & \rightarrow & T*F \\
 & | & T/F \\
 & | & F \\
F & \rightarrow & \mathbf{num} \\
 & | & (E) \\
 & | & -F
\end{array}
$$

2

Bison Specification of Calc

```
%%
Calc    : Calc  ';'  Expr
        | Expr
        ;

Expr    : Expr  '+'  Term
        | Expr  '-'  Term
        | Term
        ;

Term    : Term  '*'  Factor
        | Term  '/'  Factor
        | Factor
        ;

Factor  : num
        | '('  Expr ')'
        | -  Factor
        ;
%%
```

What is the meaning of a Calc Program?

- Does the Calc program below have meaning?

```
1+2 ; -7
```

  - If so, what is it?
  - If not, why and how can we give it meaning?

4

Meaning of **num**

- Let's consider of a subset of the Calc language

$$\begin{aligned} Calc &\rightarrow Calc; E \\ &\mid E \\ E &\rightarrow T \\ T &\rightarrow F \\ F &\rightarrow \textbf{num} \end{aligned}$$

How do we give it meaning in our compiler?

Regression – x86-64 Assembler

- We will express the meaning of a high-level language with x86-64 assembler
- X86-64 Architecture
  - two-address instructions
  - 16 64-bit general purpose integer registers
    - Reserved: %rsp, %rbp
    - Not preserved: %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15
    - 32-bit portions:
      - %eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp
      - %r8d, %r9d, %r10d, %r11d, %r12d, %r13d, %r14d, %r15d

x86 Assembler Continued

- Load/store instructions
  - Load immediate
    ```
    movl $5, %eax
    movq $5,  %rax
    ```
  - Load word from memory (register indirect)
    ```
    movl  (%rbx), %eax     # load 32-bit integer from memory address of %rbx to register %eax
    movq  (%rbx), %rax     # load 64-bit long integer from memory address of %rbx to register %rax
    ```
  - Store word to memory (register indirect)
    ```
    movl %eax, (%rbx)    # store 32-bit integer from register %eax to memory address of %rbx
    movq %rax, (%rbx)    # store 64-bit integer from register %rax to memory address of %rbx
    ```
  - Move a register
    ```
    movl %eax, %ebx      # %ebx = %eax
    movq %rax, %rbx     # %rbx = %rax
    ```
- Arithmetic instructions
  - Addition (addl, addq), subtraction (subl, subq), multiplication (imull, imulq)
    ```
    addl %eax, %ebx    # %ebx = %ebx + %eax
    subl %eax, %ebx    # %ebx = %ebx - %eax
    imull %eax, %ebx   # %ebx = %ebx * %eax
    ```
  - Division is special. For 32-bit division, "idivl" divide %edx : %eax by operand and store quotient in %eax, remainder in %edx. For example, 10/5 can be implemented as below
    ```
    movl  $5,  %ecx
    movl  $10, %eax
    cdq
    idivl %ecx
    ```


x86 Assembler Continued

- Logical instructions
  - Bit-wise and, or, xor, not (andl, andq, etc.)
    ```
    andl %eax, %ebx
    orl  %eax, %ebx
    xorl %eax, %ebx
    notl %eax
    ```
- Comparison
  - cmpl %eax, %ebx # Compare %ebx with %eax
    - set flags
- Conditional move
  - cmovg, cmovge, cmovl, cmovle, cmove, cmovne

    ```
    cmovg %eax, %ebx  // move %eax to %ebx of the flag is "greater than" which is set by cmpl
    ```

I/O: write

- Use printf.
  - declare formats

```
.int_wformat        .string     "%d\n"
```

  - Move the format label to %edi (first integer argument)
  - Move the value to print to %esi (second integer argument)
  - Example

print the integer 5

```
movl  $5, %esi,
movl  $.int_wformat, %edi
 movl  $0, %eax       #number of float arguments
call printf
```

9

I/O: read

- Use scanf
  - declare formats

```
.int_rformat        .string     "%d"
```

  - Move the format address to %edi (first integer argument)
  - Move the address to store the value to %esi (second integer argument)
  - Example

read into address in %ebx

```
movl  %ebx, %esi,
movl  $.int_rformat, %edi
movl  $0, %eax       #number of float arguments
call scanf
```

10

Meaning of addition and subtraction

- How do we determine the meaning of the following syntax?

$$E \rightarrow E + T \mid E - T$$

11

Practice Problem

- Finish the compiler for *T* and *F*.

12

What is missing?

- Even in simple calculations, we can encounter repeated calculations
  - Solution → variables
- Syntax for variable declarations

$$
\begin{array}{lcl}
P & \rightarrow & Vars\ Calc \\
Vars & \rightarrow & \textbf{var}\ IdList; \\
IdList & \rightarrow & IdList, Id \\
 & | & Id
\end{array}
$$

- What is the meaning of a variable declaration?

13

Binding Time

- Binding – an association between a name and what it names (e.g, variable and memory location).
- Times when bindings occur
  - Language design time
    - types, keywords, etc.
  - Language implementation time
    - Left to the implementation
    - Precision of operations, evaluation order of parameters, etc.
  - Program writing time
    - Variable names, etc.
  - Compile time
    - Memory layout
  - Load time
    - Machine addresses
  - Run time
    - Values to variables, method invocation

14

Binding

- Static binding
  - Refers to binding that occurs before run time
    - Statically scoped variables
    - Functions, some methods
- Dynamic binding
  - Refers to bind that occurs at run time
    - Dynamically scoped variables
    - Some virtual methods
      - Smalltalk

15

Object Lifetime

- Where should a variable be stored?
  - It depends on its lifetime
- Object lifetime – the period of time between the creation and destruction of an object (an object is a piece of data)
- Storage locations
  - Static data area – objects whose lifetime is the entire execution of a program
    - Global variables
    - String constants
  - Stack – objects whose lifetime consist of a procedure call
    - Allocated on entry to and deallocated on exit from a procedure
    - E.g., local variables
  - Heap – objects whose lifetime vary depending on execution
    - Pointers
    - Requires garbage collection

16

Allocating Space

- Where should variables in the calc language be allocated?
- Once we allocate a variable to a memory location, we need to retain that information for variable references.
  - Where should that information be stored?
    - Symbol table (hash table is one organization)

17

Symbol Tables

- What items should be entered in a symbol table?
  - variable names
  - literal constants and strings
  - source text labels

18

Information in a Symbol Table

- character string for each name
- data type
- storage class (base address, static data area, heap, stack)
- offset in storage area

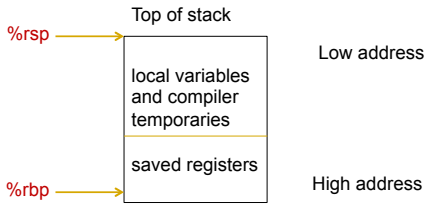
19

Symbol Table Organization

- How should the table be organized?
- Linear list
  - O(n) probes per lookup
  - easy to expand, no fixed size
  - one allocation per insertion
- Binary tree
  - O(log n) probes per lookup (balanced tree)
  - easy to expand, no fixed size
  - one allocation per item
- Hash table
  - O(1) probes per lookup (expected)
  - expansion costs vary with collision resolution scheme

20

X86-64 Stack Frame



- %rbp points to the old top of stack
  - %rbp must be set in assembler explicitly
- %rsp points to location at the top of the stack
  - %rsp must be 16-byte aligned

21

Adding Variable References

- Add the following syntax to a Calc program

$$Factor \rightarrow Id$$

- Update the compiler to handle variable declarations and references

22

Example (3.add.cm)

```
int i,j,k,l;

int main () {
        write(10+20);
        i=1;  k=3; l=4;
        j = i + k + l;
        write(j);
}
```

23

Example (3.add.cm): prologue + globals

```
        .section        .rodata
.int_wformat: .string "%d\n"
.str_wformat: .string "%s\n"
.int_rformat: .string "%d"
        .comm _gp, 16, 4 #space for globals

        .text
        .globl main
        .type main,@function
main:   nop
        pushq %rbp
        movq %rsp, %rbp
```

```
int i, j, k, l;

int main()
{
```

24

Example (3.add.cm): write(10+20);

```
movl $10, %ebx
movl $20, %ecx
addl %ecx, %ebx              # 10+20
movl %ebx, %esi              # 2nd argument of printf
movl $0, %eax                # number of float arguments
movl $.int_wformat, %edi     # 1st argument of printf
call printf
```

25

Example (3.add.cm): i:=1; k:=3; l:=4

```
movq $_gp,%rbx     #starting pointer of globals
addq $0, %rbx      # add offset of i
movl $1, %ecx      # value of rhs
movl %ecx, (%rbx)  # store; i=1
movq $_gp,%rbx
addq $8, %rbx
movl $3, %ecx
movl %ecx, (%rbx)
movq $_gp,%rbx
addq $12, %rbx
movl $4, %ecx
movl %ecx, (%rbx)
```

26

Example (3.add.cm): j := i+k+l;

```
movq $_gp,%rbx
addq $4, %rbx
movq $_gp,%rcx
addq $0, %rcx
movl (%rcx), %r8d
movq $_gp,%rcx
addq $8, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
movq $_gp,%rcx
addq $12, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
movl %r8d, (%rbx)
```

27

Example (3.add.cm): write(j); + epilogue

```
movq $_gp,%rbx
addq $4, %rbx
movl (%rbx), %ecx            #load j
movl %ecx, %esi              # 2nd argument of printf
movl $0, %eax                # number of float arguments
movl $.int_wformat, %edi     # 1st argument of printf
call printf

leave                        #epilogue
ret
```

28