Code Generation

Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Thailand

Approach

- Beginning with Parse Tree
- Abstract Interpretation

  Interpreter (take a parse tree and "run" it)
- Machine Abstraction

  Machine Instructions (what kind of machine to "run" the abstract program)
- Code Generation

Parse Tree

data structure: list

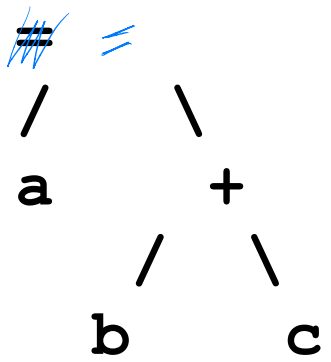
consists of operator and operand*

Operator
  binary, unary, n-ary, function call

Operand
  constant, local variable, global variable (include vector)

Illustrative example

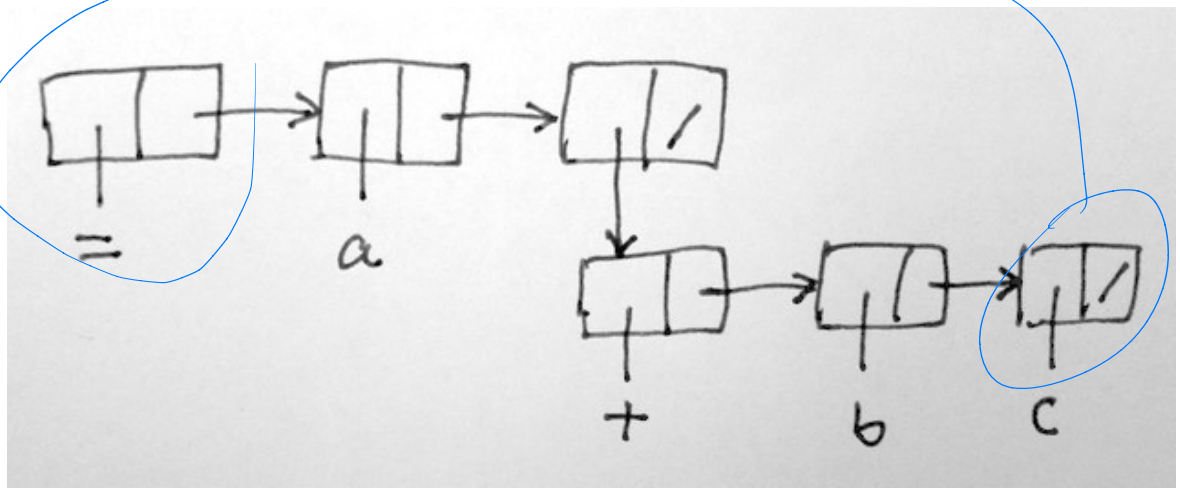
Source language: a = b + c



Printed parsed tree: (= a (+ b c))

Data structure

- This data structure is built from a number of cells.
- A cell contains two fields, head and tail.
- There are two types of cell: atom and dot-pair.
- An atom is the end of a branch (a kind of leaf node).
- A dot-pair is a pointer to another list.



a program to copy a parse tree

```
copy(a)
 if(isatom(a)) return newatom(copyof(a))
 else return cons(copy(head(a)),copy(tail(a)))
```

Interpreter

traverse a parse tree and execute the operator.

```
eval(e)
 if (e == nil) stop
 if isatom(e) do_atom(e)
 //  then e is a list  (operator operand*)
 a = head(e)
 b = tail(e)
 switch typeof(a)
   PLUS: ret eval(arg1(b)) + eval(arg2(b))
   MINUS:
   ...
   IF:   if( eval(arg1(b)) )
           ret eval(arg2(b))
         ret nil
   ...
```

Interpreter (2)

```
do_atom(a)
 switch typeof(a)
   NUM:
   GLOBAL:
   LOCAL:
   ...
```

Machine Instructions (S-code)

- A stack-based instruction set.
- S-code is zero-address instruction. The main working storage is a stack and is comparable to the registers in a processor.
- A stack has two operations: push and pop.
- It is not necessary to "address" the stack (hence the name "zero-address").

S-code

stack-based, fix 32-bit width

instruction format: argument:24-bit opcode:8-bit

```
arith:      add, sub, mul, div, mod
logic:      band, bor, bxor, shl, shr, not,
           eq, ne, lt, le, gt, ge
data:       ld, st, ldx, stx, lit, get, put
control:    call, ret, case, fun, jt, jf,
jmp
other:      inc, dec, callt, sys
```

Example: local var a,b,c

source : a = b + c

S-code :

```
get.2
get.3
add
put.1
```

global var a,b,c

source: a = b + c

S-code:

bad

```
ld.b
ld.c
add
st.a
```

control: local var a,b,c

source: if(a == b) c = 1 else c = 2

S-code:

```
    get.1
    get.2
    eq
    jf xx
    lit.1
    put.3
    jmp yy
:xx
    lit.2
```

function call

source: mysum(a,b){ return a + b; }

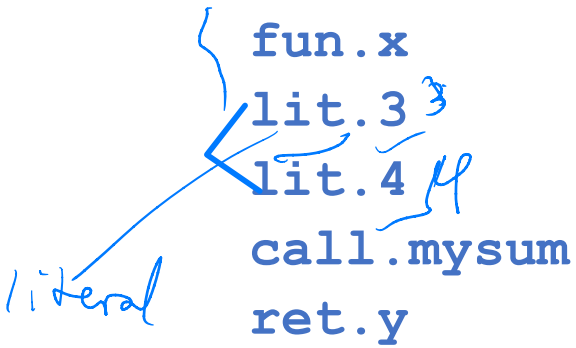
S-code:

```
fun.x
get.1
get.2
add
ret.y
```

function call (2)

source: main(){ mysum(3,4); }

S-code:



Code Generator

- The code generator works very much like an interpreter.
- Instead of "run" the parse tree, it "generates" the machine code that will give the same result as "run".

```
eval(e)
 if (e == nil) stop
 if isatom(e) do_atom(e)
 //  then e is a list
 (operator operand*)
 a = head(e)
 b = tail(e)
 switch(typeof(a))
   PLUS:
     eval(arg1(b))
     eval(arg2(b))
     out(ADD)
```

Example: generate if statement

source: if (a == b) return b;

parse tree

```
(if (== #1 #2 ) (return #2 ))
```

machine code

```
        get.1
        get.2
        eq
        jf.L18
        get.2
        ret.3
    :L18
```

Example of code generation

Source

```
sum(n, m){
 if( n == m ) return m;
 else return n + sum(
n+1, m);
}

main(){
 print(sum(1,10));
```

Parse Tree

```
(fun main
  (print (call sum 1 10 )))
(fun sum
  (if-else
      (== #1 #2 )
      (return #2 )
      (return (+ #1 (call sum (+ #1 1 )#2 )))))
```

Parse Tree

```
(fun main (print (call sum 1 10 )))
(fun sum
  (if-else (== #1 #2 )
     (return #2 )
     (return (+ #1 (call sum (+ #1 1 )#2 )))))
```

Machine code

```
:main
   fun.1
   lit.1
   lit.10
   call.sum
   sys.1
   ret.1
```

```
:sum
   fun.1
   get.2
   get.1
   eq
   jf.L18
   get.1
   ret.3
   jmp.L26
:L18
```

```
get.2
   get.2
   lit.1
   add
   get.1
   call.sum
   add
   ret.3
:L26
   ret.3
```

Summary

- Begin with abstract interpretation of a parse tree
- (interpreter)
- then modify it to output "sequence of machine code"