Topic 15 – Code Generation: Pointers

Key Ideas

- lvalue
- MIPS register conventions (for CS 241)

References

- CS241 – WLP4 Programming Language Specification
- CS241 Assignment 10: P1- P4

Review

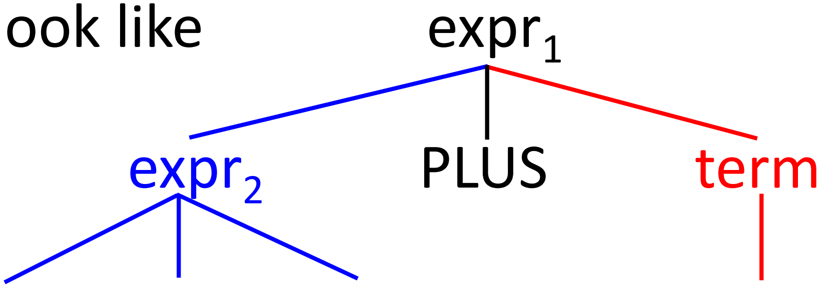
From Topic 13: Context-sensitive Analysis / Assignment 8

- we created symbol table(s) to track variable and procedure declarations and types
  - a global symbol table plus one for each procedure
- we ensured that *variables/procedures are declared exactly once*
  - 1st check for multiple declarations,
  - 2nd check for undeclared variables or procedures.
- we *checked the types of expressions*
- we augmented (or decorated) our parse tree nodes with useful information (rule, tokens, children, type, what it does) so we know, for example, which children will generate code

Review

From Topic 14: Code Generation / Assignment 9

- use syntax-directed translation, i.e. create a translation function for each syntactic category, *e.g.* for loops, if-then-else statements, comparisons, arithmetic expressions
- have a recursive function code(parse-tree-node) → MIPS assembly language code
- recursively call code() on some of the node's children to generate some code and then do some additional operations
- e.g. for rule: $expr_1 \rightarrow expr_2 + term$, part of the parse tree would look like



Review

From Topic 14: Code Generation / Assignment 9

- we would recursively call code($expr_2$) and code(term) to generate assembly language code for these two subtrees and then generate code that would add the two results
- use a few strategies to make the task easier
  - *use $3 to store the results* of evaluating a subtree
  - *use the stack to store* arguments and local variables
  - *use $29 as a frame pointer*, i.e. it points to the beginning of the stack frame for the current procedure
  - *use symbol table(s)* to store the location of these variables (relative to the frame pointers, $29)
  - for complex calculations, *push intermediate results onto the stack* (from $3) and when needed, pop it off into $5

Preview of A10

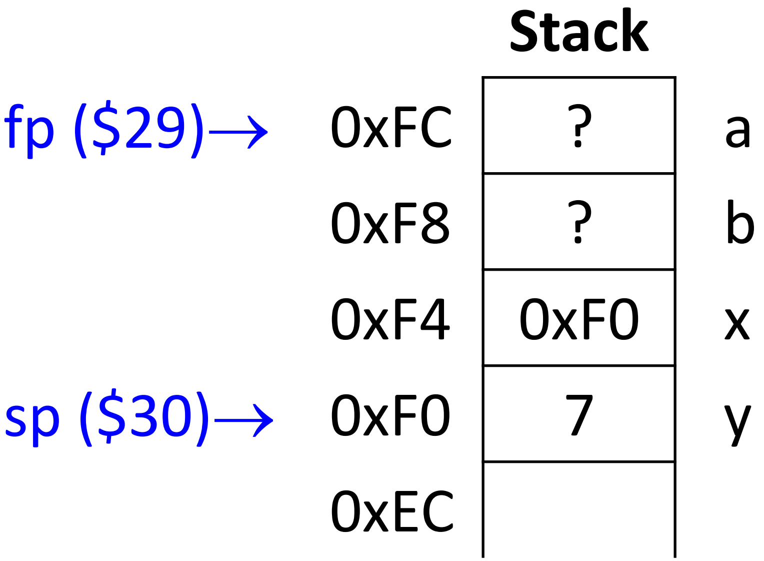
Recall

- *Our Goal:* generate a MIPS assembly language program that is equivalent to the WLP4 version (same input → same output and return value)
- We have two flavours of loaders
  - mips.twoints
  - mips.array
- WLP4 allows *arrays to be declared, initialized, dynamically allocated and destroyed*
  - represent an array as an **`int*`** that points to the first element of the array
- can also use pointers on their own (without involving arrays)

An Example

Pointers

```
int wain(int a, int b) {
  int *x = NULL;
  int y = 7;
  x = &y;
  return (*x);
}
```



Stack

fp ($29)→ 0xFC ? a

0xF8 ? b

0xF4 0xF0 x

sp ($30)→ 0xF0 7 y

0xEC

- What does this program do?
- How do we do it in MIPS?
- *Hint:* let our grammar rules be our guide, i.e. syntax-directed translation

Pointer Specifications

Specifications

- *The WLP4 compiler must support:*
  - Dynamically allocating and deallocating (heap) memory
  - Assignment through pointers
  - Dereferencing (*) and address-of (&) operators
  - pointer arithmetic
  - pointer comparisons
  - the NULL pointer

Example: A10 P1

Dereferencing a Pointer

- $\text{factor}_1 \rightarrow \text{STAR factor}_2$
- Example:

  *p
- Solution:
  - here you are dereferencing the pointer p
  - i.e. returning the contents of the address stored in p
  - generate the code for factor_2, then interpret the results (which is in $3) as an address and load the contents of that address into $3

```
code(factor₁) = code(factor₂)
                lw $3, 0($3)
```

Example: A10 P1

Code for NULL

- factor→ NULL
- Requirements:

  dereferencing a NULL pointer crashes the MIPS machine
- Solution: make NULL = 0x01,
  - not word aligned , i.e. address is not divisible by 4
  - any attempt to use this address (with the lw or sw MIPS instruction) will crash the machine
  - implementation: move 0x01 into register $3, the results register

    code(NULL) = add $3, $0, $11

Example: A10 P1

Lvalues

- Informally, there are two ways to think about *lvalues*

1) An lvalue is something that can appear on the left hand side of an assignment, i.e. it *can be assigned a value*.

These are Correct	These are Incorrect
a = 0;	0 = a;
int *p = NULL;	NULL = *p;
a = b - (c + 2);	b - (c + 2) = a;
p = &a;	&a = p;

Here a, p and *p are lvalues.

0, NULL, b-(c+2) and &a are not lvalues.

Example: A10 P1

Lvalues

2) An lvalue is an expression that *gives the location* and the type stored at that memory location, i.e. a location value, e.g.
   - a=1; means store the value 1 in the location specified by a
   - p=&a means p now refers to the same location as a refers to

- In different programming languages, lvalues can have slightly different meanings
- Even in the same language, it can mean different things in different standards:
  - In C89 the meaning is closer to version 2) above
  - Recognizing that a variable can be declared **`const`** in C, C99 is closer to a combination of versions 1) and 2).

Example: A10 P1

Lvalues

- In WLP4, lvalue appears in five production rules
  1) statement → lvalue BECOMES expr SEMI
     you can assign to it
  2) factor→ AMP lvalue
     it has a address
  3) lvalue → ID
     it can be an ID
  4) lvalue → STAR factor
     it can be a dereferenced factor
  5) lvalue → LPAREN lvalue RPAREN
     putting parenthesis around an lvalue is still an lvalue

Example: A10 P1

Code for Address-of

- factor→ AMP lvalue
- lvalue has an address
- it cannot be “NULL” or “3”
- the rule is factor→ AMP lvalue rather than factor→ AMP factor in order to prohibit patterns like “&NULL” or “&3”
- What directly derives from an lvalue?
- Ans: 3 cases

1. lvalue → ID	e.g. a=b
2. lvalue → STAR factor	e.g. *p = a
3. lvalue → LPAREN lvalue RPAREN	e.g. (a) = b

Example: A10 P1

Code for Case 1: Address-of

- factor→ AMP lvalue
- lvalue → ID
- the statement "`&y`" is asking for the address where the variable `y` is stored, so look it up in the symbol table
- the value is stored as an offset from the frame pointer ($29) so get the actual address by adding the offset to $29
- use "lis $3" and the ".word" directive
- Implementation:

```
code(factor) =  lookup offset of ID in the symbol table
                lis $3
                .word offset
                add $3, $3, $29
```

Example: A10 P1

Program

```
int wain(int a, int b) {
  int *x = NULL;
  int y = 7;
  x = &y;
  return (*x);
}
```

E.g. for the statement “&y”

- y’s offset is -0xC
- &y = $29 + y’s offset from fp
- &y = 0xFC + (-0×C) = 0xF0

```
lis $3
.word  -0xC
add $3, $3, $29
```

Stack

fp ($29)→	0xFC	?	a
	0xF8	?	b
	0xF4	0xF0	x
sp ($30)→	0xF0	7	y
	0xEC		

Symbol Table

Name	Type	Offset from fp
a	int	0x0
b	int	-0x4
x	int*	-0x8
y	int	-0xC

Example: A10 P1

Code for Case 2: Address-of

- $\text{factor}_1 \rightarrow$ AMP lvalue
- lvalue $\rightarrow$ STAR factor_2
- we will say "**`&(*y)`**" = "**`y`**"
- Solution:
  $\text{code}(\text{factor}_1) = \text{code}(\text{factor}_2)$

Code for Case 3: Address-of

- factor $\rightarrow$ AMP lvalue_1
- $\text{lvalue}_1 \rightarrow$ LPAREN lvalue_2 RPAREN
- "&**`(y)`**" = "&**`y`**"
- Solution:
  $\text{code}(\text{lvalue}_1) = \text{code}(\text{lvalue}_2)$

Example: A10 P1

Assignment to a Pointer

- lvalue → STAR factor
- recall what happens in A9 for
  statement → lvalue BECOMES expr SEMI

```
code(statement) = code(expr)
                  sw $3, ID_offset($29)
```

- i.e. store the value of the expression in the address of the variable, i.e. frame pointer ($29) plus variable's offset
- works if expr is type int and lvalue is an int variable
- must now handle case were the lvalue is an int * variable
- e.g. *p = 2;

Example: A10 P1

Assignment to a Pointer

- statement → lvalue BECOMES expr SEMI
- lvalue → STAR factor
- calculate the value of lvalue: what is the address?
- then store the result of expr at that address.
- Solution
  - calculate the code for expr and push the result onto the stack
  - calculate the code for lvalue (an address) and leave in $3
  - pop stack into $5 and store the results at the address in $3

```
code(statement) = code (expr)
                  push($3)
                  code(lvalue)
                  $5 = pop()
                  sw $5, 0($3)
```

Background for A10 P1

A Simple Array

```
int wain(int *a, int n) {
  return *x;
}
```

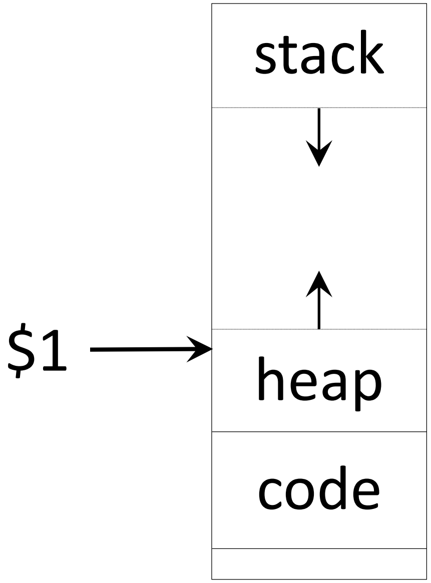
- E.g. format for mips.array loader
- What does the program do
  - Answer: return the first element of the array
- How do we do this in MIPS?
  - *find the base address for the array* (in $1)
  - lw $3, 0($1)
- What is mips.array actually doing?

Background for A10 P1

A Simple Array

```
% cat ex1.wlp4 | wlp4scan | wlp4parse > ex1.wlp4i
% ./wlp4gen < ex1.wlp4i > ex1.mips
% java mips.array ex1.mips
Enter length of array: 3
Enter array element 0: 10
Enter array element 1: 11
Enter array element 2: 12
```

- What is mips.array actually doing?
- *It allocates memory on the heap* and then calls **`wain`** with the location of the array ($1) and its size ($2) as parameters



Background for A10 P1

Another Simple Array

```
int wain(int *a, int n) {
  return *(a+1);
}
```

- What does this program do?
  - Answer: it returns the 2nd element of the array
  - a[1] = *(a+1) = *(1+a)
  - the size of each element in the array (an int) is 4 bytes so we are actually adding 4 to the base address to get the address of the 2nd element

Example: A10 P2

Dynamic Memory Allocation

- factor → NEW INT LBRACK expr RBRACK
- statement → DELETE LBRACK RBRACK expr SEMI
- we (CS241) provide the library routines that handles memory management
- you must include the following directives
  - **`.import init`**
  - **`.import new`**
  - **`.import delete`**
  - call **`init`** to initialize the heap
    - see assignment for details on parameters for **`init`**
  - link in alloc.merl (which we provide for you) as the last object file to link in.

Example: A10 P2

Dynamic Memory Allocation

- factor → NEW INT LBRACK expr RBRACK
- statement → DELETE LBRACK RBRACK expr SEMI
- `init` initializes the data structures within the dynamic memory module
- `new` allocates memory from the heap
  - $1 is the size of the array
  - it returns the addr of first element (*base address*) if successful
  - it returns 0 in $3 if memory is exhausted
- `delete` frees up the memory
  - $1 is the base address of the array
  - must delete the whole array (not part of it)
  - it does not check if $1=NULL

Example: A10 P3

Pointer Arithmetic: PLUS

- $expr_1 \to expr_2$ PLUS term
- **if** type($expr_2$) == int and type(term) == int
- **then** do as you did in A9: evaluate $expr_2$, push on stack, evaluate term, pop stack into $5 and include instruction add $3, $3, $5
- **else if** type($expr_2$) == int* and type(term) == int

```
code(expr1) = code(expr2)
              push($3)
              code(term)
              mult $3, $4  // $4 = 4, i.e. the size of one word
              mflo $3
              pop($5)
              add $3, $5, $3
```

Example: A10 P3

Pointer Arithmetic: PLUS

- **else if** type($expr_2$) == int and type(term) == int*
  - left as an exercise

- Notes:
  - *you must know the types of the children* $expr_2$ and term
  - typically you would *store type info in the parse tree nodes*
  - much of the code for “int*, int” is the same as for “int, int”

Example: A10 P3

Pointer Arithmetic: MINUS

- $expr_1 \rightarrow expr_2$ MINUS term
- **if** type($expr_2$) == int and type(term) == int
- **then** do as you did in A9: eval, push, eval, pop, sub $3, $5, $3
- **else if** type($expr_2$) == int* and type(term) == int

```
code(expr_1) = code(expr_2)
               push($3)
               code(term)
               mult $3, $4  // $4 = 4, i.e. the size of a word
               mflo $3
               pop($5)
               sub $3, $5, $3
```

Example: A10 P3

Pointer Arithmetic: MINUS

- **else if** type($expr_2$) == int* and type(term) == int*

  same as regular subtraction but divide result by 4

```
code(expr$_1$) =  code(expr$_2$)
                  push($3)
                  code(term)
                  pop($5)
                  sub $3, $5, $3
                  div $3, $4               // divide result by 4
                  mflo $3
```

Example: A10 P4

Pointer Comparisons

- test → expr_1 LT expr_2
- since the code has already successfully passed through the context-sensitive analysis phase before reaching the code generation phase, the types of expr_1 and expr_2 match
- What needs to change if type(expr_1) == *int ?
  - for A9 you used the command slt $3, $5, $3
  - for pointers use the command sltu $3, $5, $3
  - addresses / pointers are unsigned integers
  - they can range from 0 to 2^{32}-4