Type Checking Quiz

Well-typed Expressions

- Given the following declarations

```
int i;
int *p;
```

- Which of the following assignments violate WLP4's type rules?

```
i = i + i;
i = i + p;
i = p + i;
i = p + p;
i = p - p;
```

```
p = i + i;
p = i + p;
p = p + i;
p = p + p;
p = p - p;
```

Assignment 8

Symbol Table

- For A8 you will only have *one procedure*, i.e. this rule
  procedures → main
  never this rule
  procedures → procedure procedures
  which generates addition procedures.
- So you only need to create *a single symbol table* for this single procedure not a global one (containing signatures) plus one for each procedure

Type Checking for A8

- first, type check all expressions: subtrees with root *expr* or *lvalue*
- next, check that all subtrees with root *statement* or *test* are well-typed

Topic 14 – Code Generation

Key Ideas

- syntax-directed translation
- frame pointer (fp)
- MIPS register conventions (for CS 241)

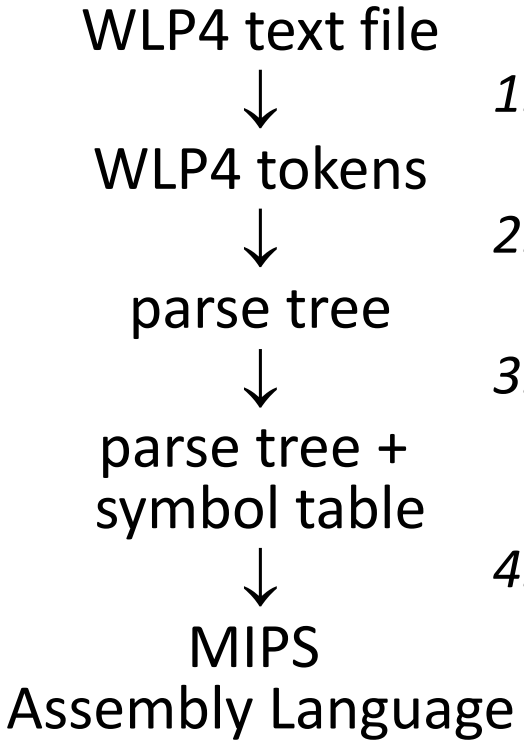
References

- *Basics of Compiler Design* by Torben Ægidius Mogensen section 7.4
- CS241 – WLP4 Programming Language Specification
- CS241 Assignment 9

Code Generation: Overview

Recall: Basic Compilation Steps

The steps in translating *a program from a high level language to an assembly language program* are:



1. *scanning*: identify the tokens
2. *syntax analysis*: parse
3. *context-sensitive analysis:* create a symbol table and perform type checking
4. *code generation*

Code Generation: Overview

Overview

- *Input:*
  - a parse tree + a symbol table
- *Preconditions:*
  - the program is semantically valid, *i.e.* types rules have been followed (it is well-typed), variable and procedures are properly declared, scope has been utilized
- *Output:*
  - MIPS assembly language program that is equivalent to the WLP4 version (same input $\rightarrow$ same output and return value)
  - many possible answers, *i.e.* append "`add $1,$1,$0`" to the program any number of times

Code Generation: Overview

Key Issues

- *Correct*
  - compiler must create an equivalent program
  - compiler must be correct for all valid inputs (i.e. programs)
- *Ease of (or simplicity of) writing the compiler*
  - especially for CS 241 students
- *Efficiency of the compiler:*
  - how long does it take to compile a program
- *Efficiency of the compile code:*
  - how quickly does it run
  - code optimization is only touched on in this course e.g. how to assign registers effectively

Code Generation: Overview

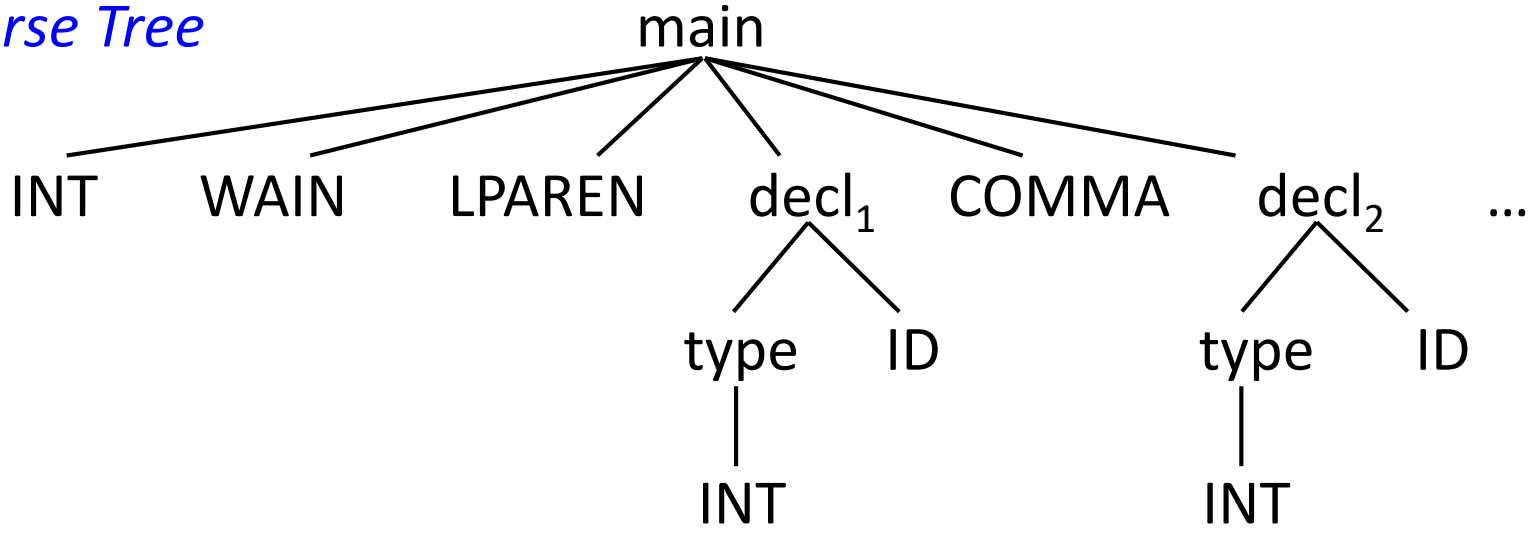
Approach

- *Syntax-directed Translation*
  - create a translation function for each syntactic category, *e.g.* for loops, if-then-else statements, assignment, procedures
    - e.g. a code() function for each grammar rule/production
  - translation closely follows the syntactic structure of the code (*i.e.* parse tree) with some additional information as needed (*i.e.* symbol table)
  - recursively traverse the parse tree to gather the needed information

Code Generation: Overview

Approach

Parse Tree



Code Generation

- for rule $\alpha \rightarrow \beta$, (e.g. expr $\rightarrow$ term) then code(α) = code(β)
- code(main) = code($decl_1$) + code($decl_2$) + code(statements) + ...
- here "+" means concatenation and code() means generate the code for this expression

Storing Variables: A9P1

Example a)

```
int wain(int a, int b) { return a; }
```

Output

```
add $3, $1, $0     ; move 1st parameter to register
jr $31             ; that holds the return value and
                   ; return control to OS
```

Conventions

- $1 and $2 hold the parameters for the **`wain`** function
  - think of loaders mips.twoints and mips.array
- $3 holds the return value

Storing Variables: A9P1

Example b)

```
int wain(int a, int b) { return b; }
```

Output

```
add $3, $2, $0     ; move 2nd parameter to register
jr $31             ; that holds the return value and
                   ; return control to OS
```

Observation

- Examples a) and b) both have the same parse tree.
- How do we differentiate the programs?
- Add a Location column to the Symbol Table

Symbol Table		
Symbol	**Type**	**Location**
a	int	$1
b	int	$2

Storing Variables

Option A: Store Variables in Registers

- Idea : each variable gets own register
- Problem: what if there are more that 32 variables

Option B: Store Variables in RAM

- Idea : Store variables in RAM usings .word directive
  - each variable x gets its own label “x” in MIPS
- Problems
  - costly to access variables
  - must be able to differentiate local from gobal variables if both share the same label

Storing Variables

Option C: Store Variables in Stack

```
int wain(int a, int b) {
  int c = 0;
  return a;
}
```

- store parameters *a*, *b* and local variable *c* on the stack

Symbol Table		
Symbol	**Type**	**Location**
a	int	8
b	int	4
c	int	0

```
;;; prolog
lis $4
.word 4
sw $1,-4($30)   ; push a
sub $30,$30,$4
sw $2,-4($30)   ; push b
sub $30,$30,$4
;;; body
sw $0,-4($30)   ; push c
sub $30,$30,$4
lw $3,8($30)    ; return a
;;; epilog
add $30,$30,$4 ; pop c
add $30,$30,$4 ; pop b
add $30,$30,$4 ; pop a
jr $31
```

Storing Variables

Frame Pointer

- Problem
  - we don't know the offsets to the parameters and variables until all the variables (for that function) have been defined
  - cannot use stack for temporary storage after that because we'll change the value of the stack pointer and so the offsets in the symbol table will all need to be updated
- Solution: *frame pointer (fp)*
  - reserve $29 to point to the first element of the stack (for this procedure)
  - offsets in symbol table will be relative to the frame pointer
  - frame pointer does not change value as variables are declared

Storing Variables

Option C: Store Variables in Stack

```
int wain(int a, int b) {
  int c = 0;
  return a;
}
```

- store *a*, *b* and *c* on the stack with location offset relative to the frame pointer, $29

Symbol Table		
Name	**Type**	**Location**
a	int	0
b	int	-4
c	int	-8

```
;;; prolog
lis $4
.word 4
sub $29,$30,$4 ; push a
sw $1, -4($30)
sub $30,$30,$4
sw $2, -4($30) ; push b
sub $30,$30,$4
;;; body
sw $0, -4($30) ; declare c
sub $30,$30,$4
lw $3, 0($29)  ; return a
;;; epilog
add $30,$29,$4 ; pop c,b,a
jr $31
```

Storing Variables

Option C: Store Variables in Stack

- Prolog: have the frame pointer ($29) point to the next available stack location ($30 - 4).
- Body: refer to parameters and variables based on the frame pointer ($29) which does not change value during the function
- Epilog: have the stack pointer ($30) point to its previous value before the function call ($29+4)

```
;;; prolog
lis $4
.word 4
sub $29,$30,$4 ; push a
sw $1, -4($30)
sub $30,$30,$4
sw $2, -4($30) ; push b
sub $30,$30,$4
;;; body
sw $0, -4($30) ; declare c
sub $30,$30,$4
lw $3, 0($29)  ; return a
;;; epilog
add $30,$29,$4 ; pop c,b,a
jr $31
```

Storing Variables

Some Conventions (for CS 241)

- We will use the following conventions in CS 241

$0	always 0 (or false)
$1	value of 1st parameter / argument (a1)
$2	value of 2nd parameter / argument (a2)
$3	result (and intermediate results) of calculations
$4	constant 4, useful in pushing on and popping off the stack
$5	store previous intermediate results
$11	always 1
$29	frame pointer (fp)
$30	stack pointer (sp)
$31	return address (ra)

Storing Variables

Some Conventions (for CS 241)

- We will use the following conventions in CS 241

$0	always 0 (or false)
$1	value of 1st parameter / argument (a1)
$2	value of 2nd parameter / argument (a2)
$3	result (and intermediate results) of calculations
$4	constant 4, useful in pushing on and popping off the stack
$5	store previous intermediate results
$11	always 1
$29	frame pointer (fp)
$30	stack pointer (sp)
$31	return address (ra)

Storing Variables

Some Conventions (for CS 241)

- Prolog
  - push return address ($31) on stack
  - push arguments on stack
- Generated Code
  - put local variables on stack
  - generate code for body of function
- Epilog
  - pop frame (local variables and arguments) off stack
  - restore previous return address to $31

Some Examples: A9P2

Code

```
int wain(int a, int b) {
   return (a);
}
```

Output

```
;; same prolog
lw $3, 0($29) ; load a from stack(based on fp)
;; same epilog
```

Rationale

- recall: for rule $\alpha \rightarrow \beta$, use code(β) to build code(α)
- for the rule : factor $\rightarrow$ LPAREN expr RPAREN

  code(factor) = code(LPAREN) + code(expr) + code(RPAREN)
  = code(expr)

Some Examples: A9P3

Code

```
int wain(int a, int b) {
   return a+b;
}
```

Output

```
add $3, $1, $2
```

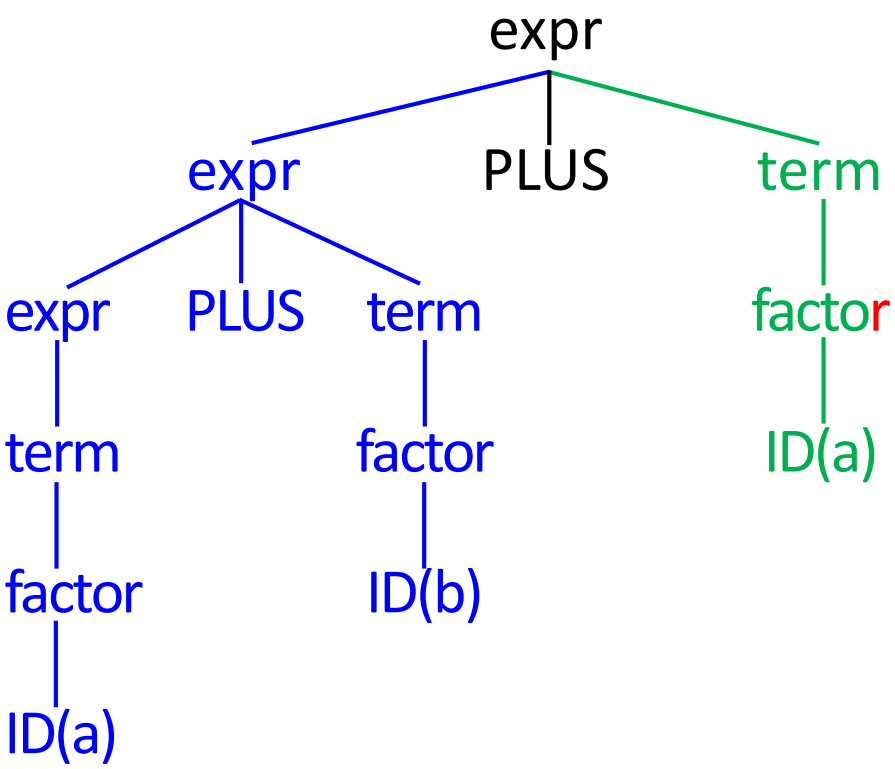
Does this approach always work?

- What about a+b+a?
- What about a+b+c+d ?
- What about a+b+c+d+e? and so on ...

Some Examples: A9P3

Parse Tree

```
int wain(int a, int b) {
    return a+b+a;
}
```



Problem

- If we assign $3 to the value of the left subtree (expr) what register do we assign to the right subtree (term)?
- If our plan is to use another register, and if there are many nested subexpressions, we will run out of registers .
- Our idea needs to work for an *arbitrary* number of expressions.

Some Examples: A9P3

Aproach

- for rule: $expr_1 \rightarrow expr_2 + term$ the code is

Output

```
;; code(expr1) =
code(expr2)      ; $3 = expr2
push($3)         ; pseudocode to push $3 onto stack
code(term)       ; $3 = term
$5 = pop()       ; $5 = expr2, pseudocode to pop stack
add $3, $5, $3   ; $3 = expr2 + term
```

Rationale

- use $3 for all intermediate (as well as final) results
- use stack and $5 to store and then use the results of previous intermediate calculations
- only need one other register, $5.