Lecture 18

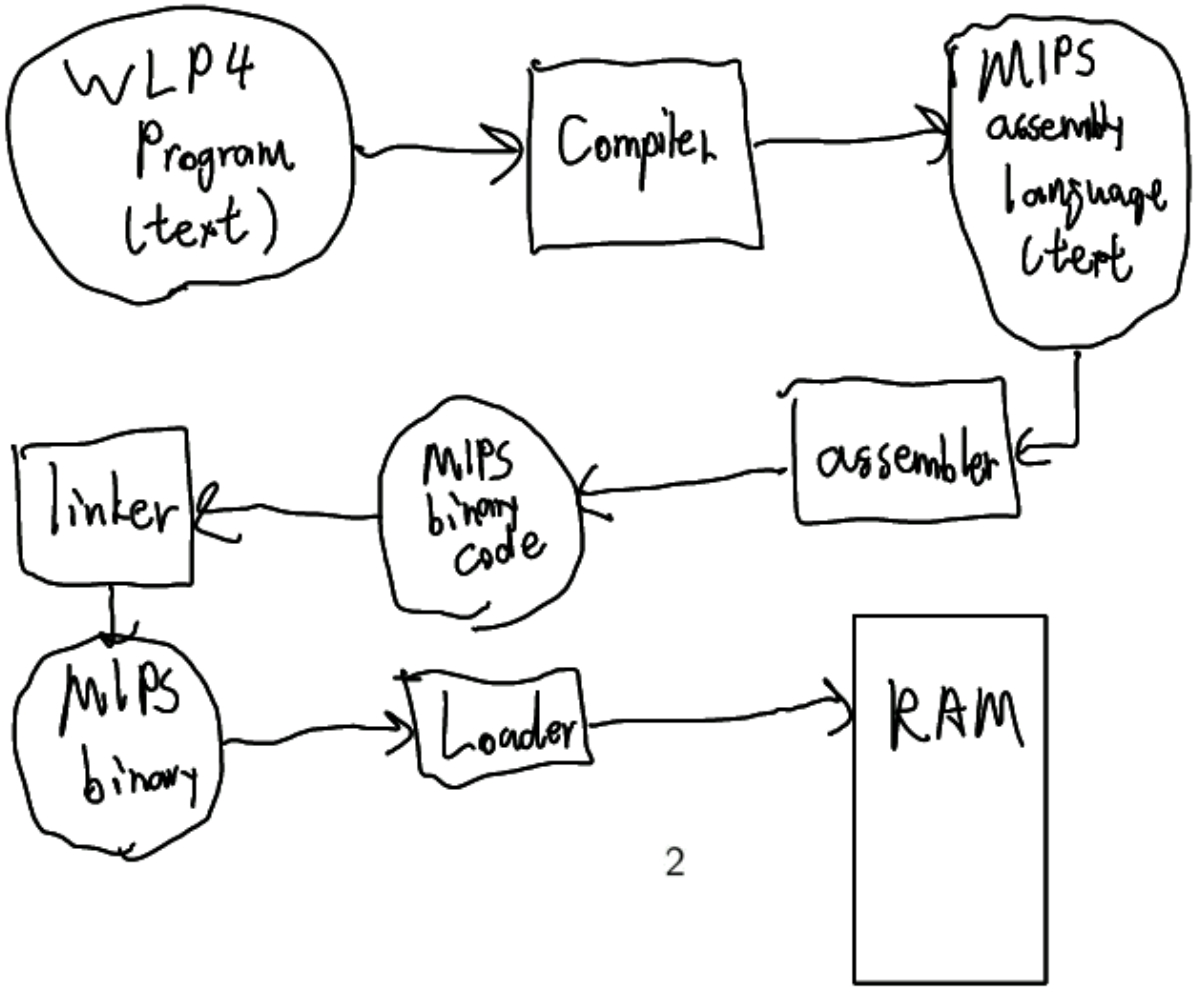
Semantic Analysis

CS 241: Foundations of Sequential Programs
Fall 2014

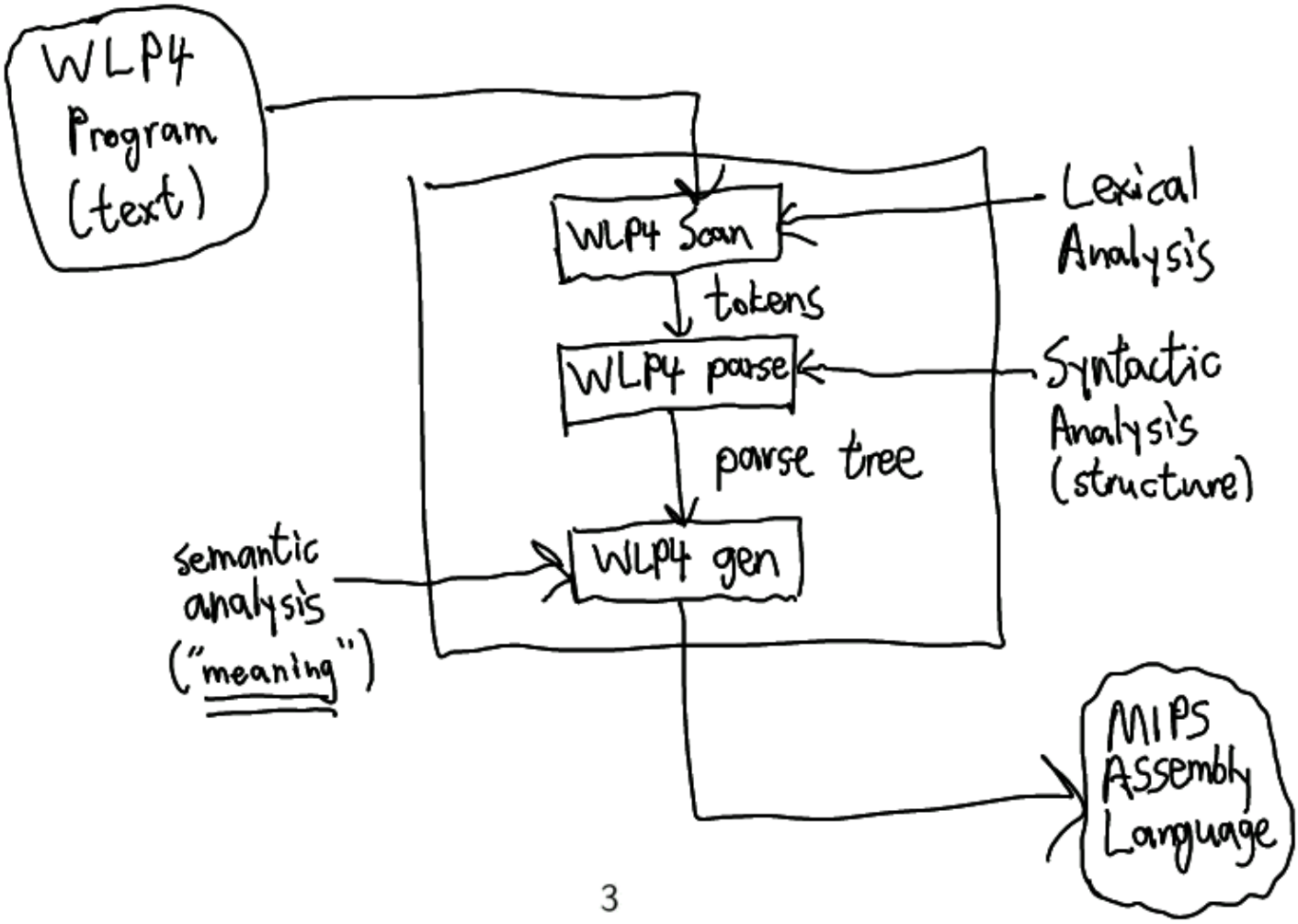
Troy Vasiga et al
University of Waterloo

Big Picture of WLP4 Compilation

From WLP4 to MIPS



Zooming In On Compiler



Example

Input file:

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

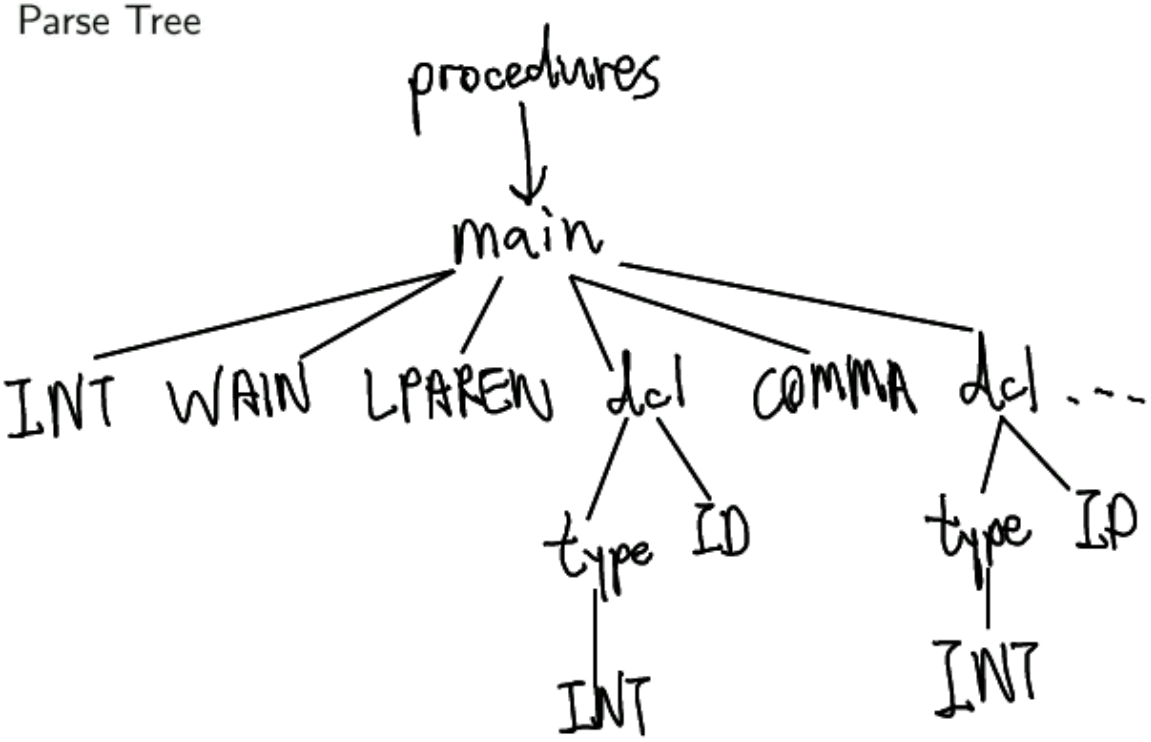
Example

Tokens from Scanner

INT, WAIN, ID, COMMA, LPAREN

Example

Parse Tree



Parse Tree

Write `wlp4gen.rkt` or `wlp4gen.cc`

- structure
- traversals

Process a `.wlp4i` file and generate a parse tree.

Hint: You have dealt with constructing a tree in Assignment 3.

Next steps

- further error checking (context-sensitive analysis)
- code generation

Context-Sensitive Analysis

↳ where this statement occurs matters.

- Input: parse tree
  (∴ program is syntactically valid)

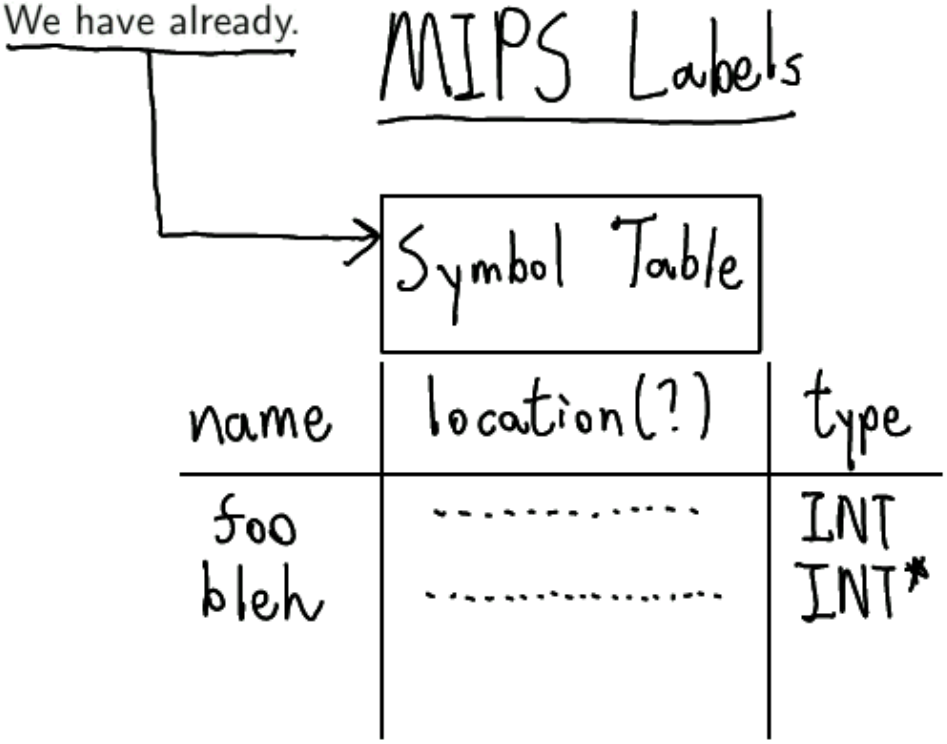
- Output:
  - ERROR, if parse tree represents a semantically invalid program
  - Parse tree + something, if parse tree is valid

Possible errors

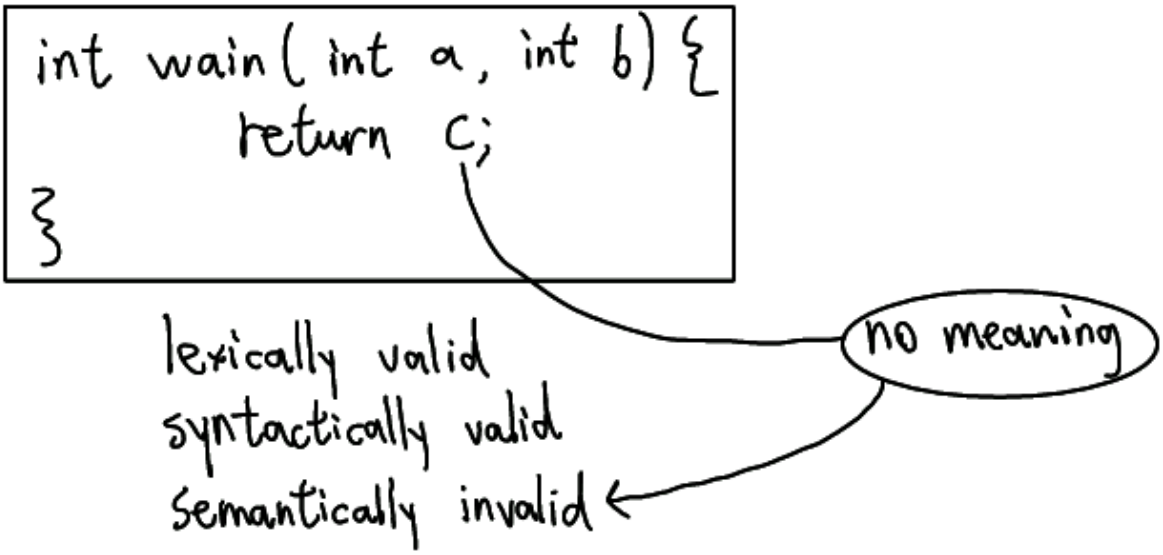
If a program is syntactically valid, what else can go wrong?

- variables: undeclared
- variables: duplicate
- variables: types
- procedures: undeclared
- procedures: duplicate
- procedures: type of return value
- procedures and variables: type of calling matches the type of declaration
- variables and procedures: scope of variables in/out of procedures

Solving the "variable declaration" problems



Solving the "undeclared variable" problem



When using a variable, look in symbol table to ensure it exists.

Solving the "duplicate variable" problem

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

When declaring, check that the variable is not in the symbol table already.

Procedures bite us

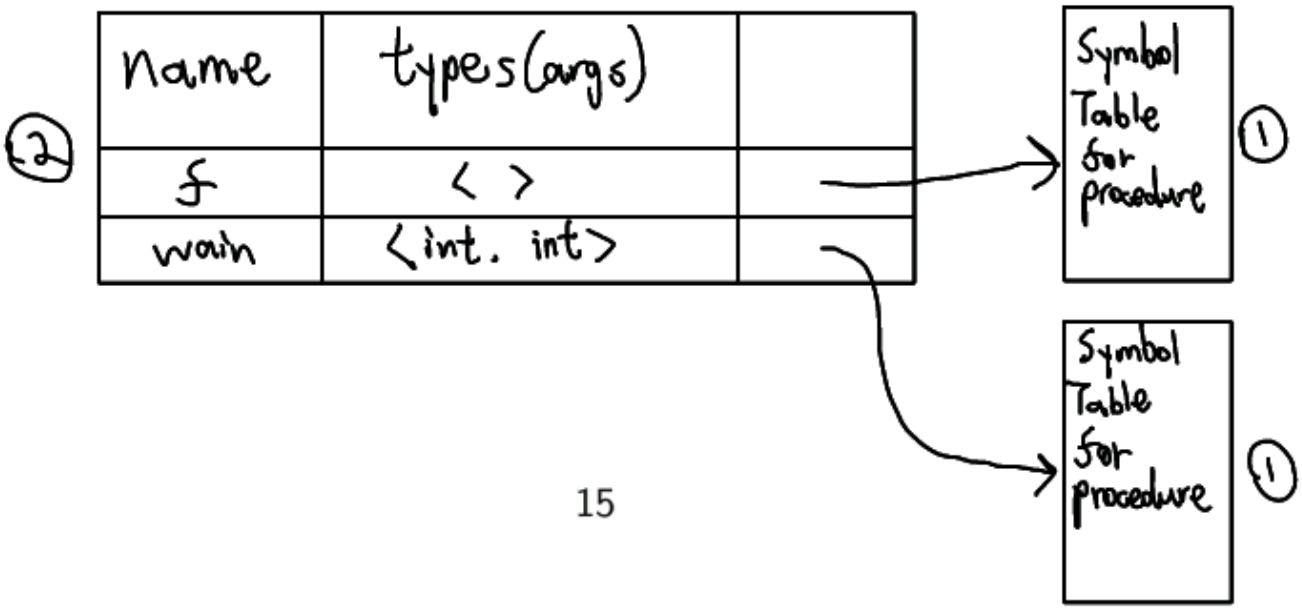
```
int f() {
    int a = 0;
    return 1;
}
int wain(int x, int y) {
    int a = 0;
    return 1;
}
```

not duplicates!!!

Solving "duplicate/undeclared" procedures and variables at the same time

① Separate symbol tables for each procedure

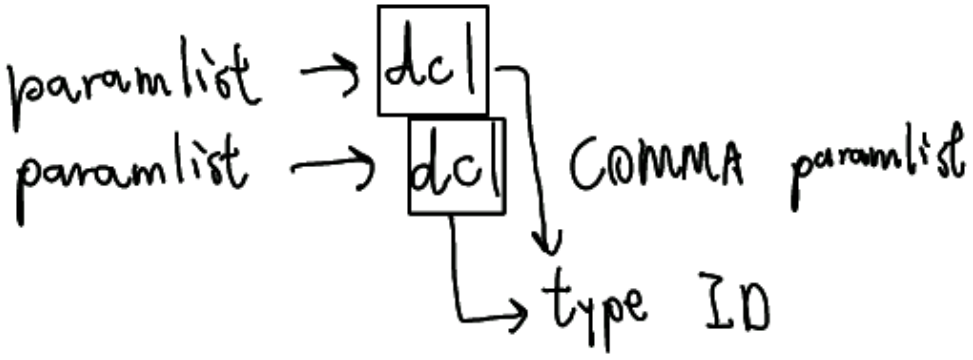
② Global symbol table for procedure



Obtaining signatures

procedures have:

- names ← easy to extract
- return types ← easy: INT in WLP4
- parameters of specific types.



Why types matter

Recall: looking at bits will not tell us what they represent.

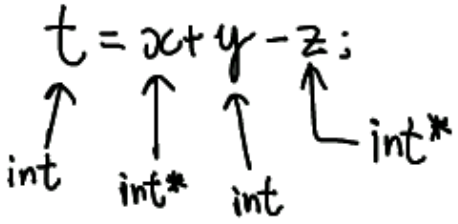
Types help us remember what a variable means.

ex: `int* a = NULL;`
`a = 7;` ← should be prevented

Solving the "type mismatch" problems

See the "WLP4 semantic rules" handout. ← A8

Notation: $\frac{\text{assumptions}}{\text{consequences}}$



To type-check:

- make sure all rules in the above handout are met when computing the type of an expression
- make sure the left-hand side type (`lvalue`) is the same as the right-hand side type (`expr`)

Hint: pointers make your life even more interesting

$\frac{E : \text{int}}{\&E : \text{int*}}$ If E is a type of int, &E is a type of int*

Types of identifiers

look in symbol table

Simple production rules

expr → term
term → factor
factor → ID

typeOf(LHS) ← typeOf(RHS)

Basic Pointer Types: & *

& address of
* pointer dereference

Addition and subtraction are hard

multiplication/division are ez

+ : int + int → int
int* + int → int*
int + int* → int*
int* + int* → INVALID

− : int − int → int
int* − int → int*
int* − int* → int
int − int* → INVALID

Functions, comparisons and other statements

Refer to handouts

More advanced context-sensitive analysis

"Real" languages add more complexity.

- scope: declarations within loops/ifs
- nested pointer rules: int*****
- other types: booleans, strings, ...
- user defined types
- objects

CS442

Testing

- Make a script!
- Store expected output in a file and automatically `diff` against it.
- Start making many, small test cases now.
- Write a "wrapper" script to test all your tests as you progress along.