Topic 13 – Context-sensitive Analysis

Key Ideas

- variable and procedure declarations
- scope
- type checking
- well-typed expressions

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 4.1- 4.2, 6.1-6.7
- WLP4 Language Spec and Type rules

  https://www.student.cs.uwaterloo.ca/~cs241/wlp4/WLP4.html

  https://www.student.cs.uwaterloo.ca/~cs241/wlp4/typerules.pdf

What is Next?

From Source Code to Running Program

WLP4 program (text)

↓ *1.* *compiler*

MIPS assembly
language (text)

↓ *2.* *assembler*

MERL file (binary)

↓ *3.* *linker*

MERL file (binary)

↓ *4.* *loader*

process (binary) i.e.
running program in RAM

What is Next?

Basic Compilation Steps

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

↓ *1. WLP4 program*

(A6) WLP4 Scan: lexical analysis / regular languages

↓ *2. tokens*

(A7) WLP4 Parse: syntactic analysis / context-free grammars

↓ *3. parse tree*

(A8) WLP4Gen: semantic analysis

↓ *4. symbol table*

(A9-A10) code generation

↓ *5. MIPS assembly language*

What is Next?

Basic Compilation Steps

WLP4 Input file:

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

$\xrightarrow{\text{A6}}$

Tokens from WLP4 Scan:

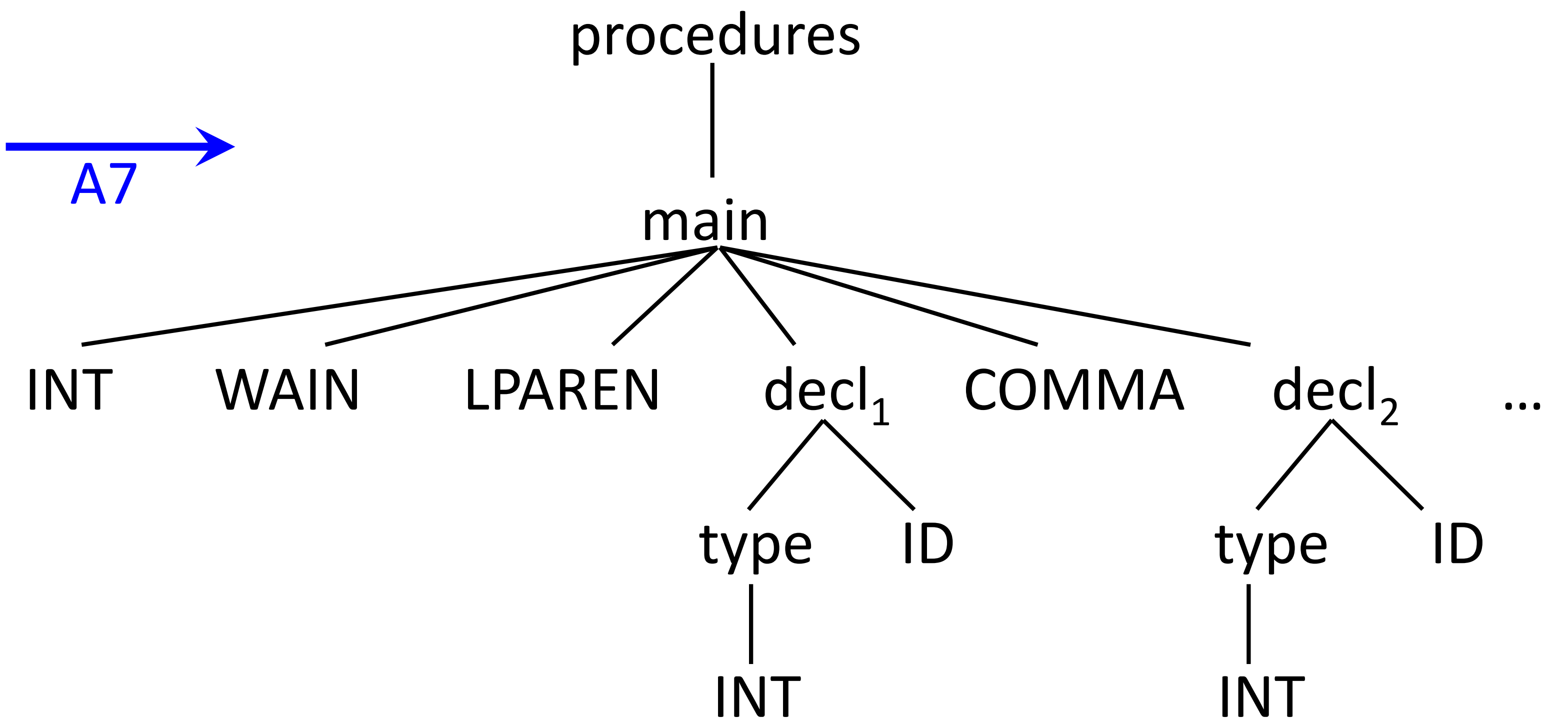
INT
WAIN
LPAREN
INT
ID
COMMA
INT
ID
RPAREN
…

$\xrightarrow{\text{A7}}$

What is Next?

Basic Compilation Steps

Parse Tree



- unlike a binary tree, nodes can have more than two children

Context-Sensitive Analysis

Syntax vs. Semantics

- *Context-free:* where it occurs *does not* matter,
  - e.g. it does not matter if a variable is used before it is declared
- *Context-sensitive:* where it occurs *does* matter
  - e.g. it is an error if a variable is used before it is declared
- Input:
  - a parse tree
- Precondition:
  - the program is syntactically valid
- Output:
  - **if** it is semantically valid **then** output a augmented parse tree
    else output ERROR

Context-Sensitive Analysis

Errors that a Context-Sensitive Analysis Finds

- If a program is syntactically valid, what else can go wrong?
  - *variables* can
    - be undeclared, used before they were declared
    - have multiple declarations
  - *procedures* can
    - be undeclared, used before they were declared
    - have multiple declarations
  - *types*
    - return value of procedures
    - parameter lists
    - operators
  - *scope*
    - scope of variables in and out of procedures

Variable Declaration Issues

How to Solve Variable Declaration Issues

- Answer: *A Symbol Table*
  - similar to the one we did our assembler and MIPS Labels
  - track: *Name* and *Location*
    - which we also did for MIPS
  - also track: *Type* (e.g. INT and INT*)
    - did not track this information with our MIPS assembler
    - programming languages generally have many more types, long, char, float, double, etc.

Variable Declaration Issues

How to Solve Variable Declaration Issues

- e.g. test001.wlp4

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

- *When using a variable*, make sure it is in the symbol table
  - i.e. it exists
- "return c;" is
  - lexically valid,
  - syntactically valid,
  - but is semantically invalid (i.e. a semantic error) if **c** has not been declared somewhere, i.e. if we do not know what location **c** represents

Variable Declaration Issues

How to Solve Variable Declaration Issues

- e.g. test002.wlp4

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

- *When declaring a variable*, make sure it is not already in the symbol table

Checking Variable Declarations

First Check for Multiple Declarations

- *recursively traverse* the parse tree and track any declarations
- *search* for nodes with rule *decl* $\rightarrow$ TYPE ID
  - extract the name (e.g. a) and the type (e.g. int)
  - *check* **if** the name is already in the symbol table **then** ERROR
  - else *add* name and type to symbol table

Next Check for Undeclared Variables

- *recursively traverse* the parse tree and track the use of variables
- *search* for nodes with the rules
  - *factor* $\rightarrow$ ID
  - *statement* $\rightarrow$ ID BECOMES *expr* SEMI
- *check* if ID's name is in symbol table, if not then ERROR

Checking Variable Declarations

Scope

- must also consider the concept of *scope*
- both functions can declare and use the local variable **a**

```
int f() {
  int a=0;
  return a;
}

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

- clearly we need a more sophisticated version of a symbol table

Variable Declaration Issues

How to Solve Variable Declaration Issues

- have a *global symbol table* for procedures
- have separate symbol tables for each procedure

Name	**Types (args)**	
f	< >	•
wain	<int, int>	•

Name	**Type**
a	int

Name	**Type**
x	int
y	int
a	int

Variable Declaration Issues

Obtaining Signatures

- Procedures have *signatures,* i.e.
  - *names* (which must be extracted)
  - *return types* (which is always `int` in WLP4)
  - *parameters lists* with a mixture of `int` and `int *` types

- *Finding procedures* in the parse tree
  - *traverse* the parse tree and *search* for procedures declarations i.e. nodes with one of these two rules
    - procedure $\rightarrow$ INT ID LPAREN params RPAREN…
    - main $\rightarrow$ INT WAIN LPAREN dcl COMMA dcl RPAREN…

Variable Declaration Issues

Obtaining Signatures

- once you have found one of these rules
  - *procedure* $\rightarrow$ INT ID LPAREN …
  - *main* $\rightarrow$ INT WAIN LPAREN …
- if the procedure name is not already in the global symbol table then add it and create a new symbol table for that procedure
- for procedures we store its signature in the symbol table.
  - these are captured by the following production rules

    decl $\rightarrow$ TYPE ID

    paramlist $\rightarrow$ decl

    paramlist $\rightarrow$ decl COMMA paramlist

Type Checking

Why Types Matter

- Recall: looking at a pattern of bits will not tell us what they represent
- in WLP4 there are only two types: *int* and *int **
- Types help us
  - remember what a variable means
  - interpret the pattern of 0's and 1's
  - delimit how a value can be used
  - catch if we have used the value improperly (sometimes)
  - e.g.

```
int *aPtr = NULL;  // aPtr is pointer
aPtr = 7;          // trying to store an int
                   // ERROR
```

Type Checking

Working with Type Rules

- See “WLP4 semantic rules” handout
- Notation: $\dfrac{\text{assumptions}}{\text{consequences}}$ or $\dfrac{\text{preconditions}}{\text{postconditions}}$
- 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, the *lvalue* type, is the same as the right-hand side, the *rvalue* type, e.g. for rules

    expr $\rightarrow$ term
    term $\rightarrow$ factor — type(LHS) = type(RHS)
    factor $\rightarrow$ ID
    factor $\rightarrow$ NUM

Type Checking

Working with Type Rules

- *must check if types are being used properly*
- we'll introduce the variable τ to represent a type
- i.e. τ can be *int* or τ can be *int ** (for WLP4)
- we'll need τ to talk about types without mentioning a specific type, e.g.
  - "E_1 is of type τ and E_2 is of type τ" means that E_1 and E_2 have the same type, i.e. they are either
    - both *int* or
    - both *int **
- pointers create a challenge here, i.e. we have to track if the type is *int* or *int **

Type Checking

Working with Type Rules

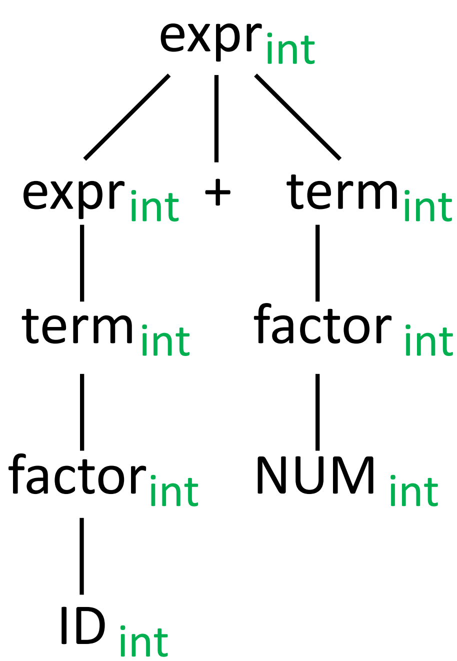
- To type-check:
  - decorate the parse tree with types
  - propagate from the leaves up
  - ensure that rules are followed

    $expr \rightarrow term$

    $term \rightarrow factor$

    $factor \rightarrow ID$

    $factor \rightarrow NUM$
  - e.g int + int is an int
  - *we need a method to specify type rules*



Type Checking

Working with Type Rules

- Rule: $\dfrac{<\mathit{id.name},\ \tau> \in \mathit{decl}}{\mathit{id.name} : \tau}$
- Meaning:
  - **if** *id.name* was declared to have type τ
  - **then** *id.name* has type τ
  - true if $\tau = \mathit{int}$ or $\tau = \mathit{int}\ *$
- Rules: $\dfrac{}{\text{NUM} : \mathit{int}}$ $\qquad \dfrac{}{\text{NULL} : \mathit{int*}}$ $\qquad \dfrac{\text{E} : \tau}{\text{(E)}: \tau}$
- Meaning:
  - NUM is always of type *int* (no assumptions are needed)
  - NULL is always of type *int** (no assumptions are needed)
  - putting parenthesis around an expression preserves its type

Type Checking

Type Rules for Pointer Types

- Rules: $\dfrac{E : \mathit{int}}{\&E : \mathit{int}*}$ $\dfrac{E : \mathit{int}*}{*E : \mathit{int}}$
- Meaning:
  - when you take the address of an *int*, you get an *int* pointer
  - when you dereference an *int* pointer (put a * in front of it) , you get an *int.*

- Rule: $\dfrac{E : \mathit{int}}{\text{new } \mathit{int}[E] : \mathit{int}*}$
- Meaning:
  - when you create a new array (of size E) you get a pointer to an *int* (i.e. a pointer to the first element in the array)

Type Checking

Type Rules for Arithmetic Operations

- Rules: $\dfrac{E_1 : \textit{int} \quad E_2 : \textit{int}}{E_1 * E_2 : \textit{int}}$ $\dfrac{E_1 : \textit{int} \quad E_2 : \textit{int}}{E_1 / E_2 : \textit{int}}$ $\dfrac{E_1 : \textit{int} \quad E_2 : \textit{int}}{E_1 \% E_2 : \textit{int}}$
- Meaning:
  - if E_1 and E_2 are *ints* then the result of multiplying them, dividing them or finding the remainder is also an *int*.

- Rule: $\dfrac{E_1 : \textit{int} \quad E_2 : \textit{int}}{E_1 + E_2 : \textit{int}}$ $\dfrac{E_1 : \textit{int*} \quad E_2 : \textit{int}}{E_1 + E_2 : \textit{int*}}$ $\dfrac{E_1 : \textit{int} \quad E_2 : \textit{int*}}{E_1 + E_2 : \textit{int*}}$
- Meaning:
  - When you add two *ints*, the sum is an *int.*
  - When you add an *int* and a pointer to an *int*, the sum is a pointer to an *int.* You *cannot add two pointers* to *ints,* i.e. there is no rule for this operation.

Type Checking

Type Rules for Arithmetic Operations

- Rules: $\dfrac{E_1 : \mathit{int} \quad E_2 : \mathit{int}}{E_1 - E_2 : \mathit{int}}$ $\dfrac{E_1 : \mathit{int*} \quad E_2 : \mathit{int}}{E_1 - E_2 : \mathit{int*}}$ $\dfrac{E_1 : \mathit{int*} \quad E_2 : \mathit{int*}}{E_1 - E_2 : \mathit{int}}$
- Meaning:
  - When you subtract two *ints*, or two *int**'s, the difference is an *int.*
  - An *int** minus an *int* is an *int**.
  - You *cannot subtract an int* from an int*

- Rules: $\dfrac{< f, () > \ \in \mathit{decl}}{f() : \mathit{int}}$ $\dfrac{< f, (E) > \ \in \mathit{decl} \ E : \tau}{f(E) : \mathit{int}}$
- Meaning:
  - If a function with 0 or 1 parameters has been declared, its return type is *int.*

Type Checking

Well-typed Expressions

- For more complicated structures, such as while loops, in statements, or the body of a procedure we check that the structure is *well-typed.*

- Rules: $$\frac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\, E_1 == E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\, E_1 < E_2)}$$

- Meaning:
  - If E_1 and E_2 are of the same type, then the comparisons $E_1 == E_2$ and $E_1 < E_2$ are well-typed.
  - We allow "less than" comparisons of pointers.
  - These are referred to as *tests.*

Type Checking

Well-typed Expressions

- Rules: $$\frac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\ E_1 = E_2)}$$
- Meaning:
  - When you assign a value to a variable, then the types must match.
  - This is referred to as *assignment.*

- Rules: $$\frac{E : \textit{int*}}{\textit{well-typed}(\text{ delete [] E})}$$
- Meaning:
  - You can deallocate memory if it is a pointer to an *int*

Type Checking

Well-typed Expressions

- Rules:

$$\frac{\textit{well-typed}(\mathrm{T}) \quad \textit{well-typed}(\mathrm{S}_1)}{\textit{well-typed}(\ \textit{while}\ (\mathrm{T})\ \{\mathrm{S}_1\}\ )}$$

- Meaning:
  - *Here T is a test and S_1 are statements.*
  - The while loop is well-typed if you have
    - the *while* keyword,
    - followed by the left parenthesis,
    - followed by a test,
    - follow by a right parenthesis,
    - followed by a left brace,
    - followed by statements,
    - followed by a right brace.

Type Checking

Well-typed Expressions

- Rules: $$\frac{\textit{well-typed}(\mathrm{T}) \quad \textit{well-typed}(\mathrm{S}_1) \quad \textit{well-typed}(\mathrm{S}_2)}{\textit{well-typed}(\ \textit{if}\ (\mathrm{T})\ \{\mathrm{S}_1\}\ \textit{else}\ \{\ \mathrm{S}_2\}\ )}$$
- Meaning:
  - *Here T is a test, S_1 and S_2 are statements.*
  - The if statement is *well-typed* if you have
    - the *if* keyword,
    - followed by the left parenthesis,
    - followed by a test,
    - follow by a right parenthesis,
    - followed by a left brace,
    - followed by statements,
    - followed by …