What did Parsing check

- wain function has been defined
- return (appears only once in a procedure as the last statement)
- every return type is an integer in WLP4

Rules 2 check:

1. Type rules are not violated
2. We cannot declare more than one variable with the same name in the same scope.
3. A variable cannot be used before it has been declared.

WLP4, CSA

We use a code-based solution. We **traverse** the **parse-tree** generated by the parser

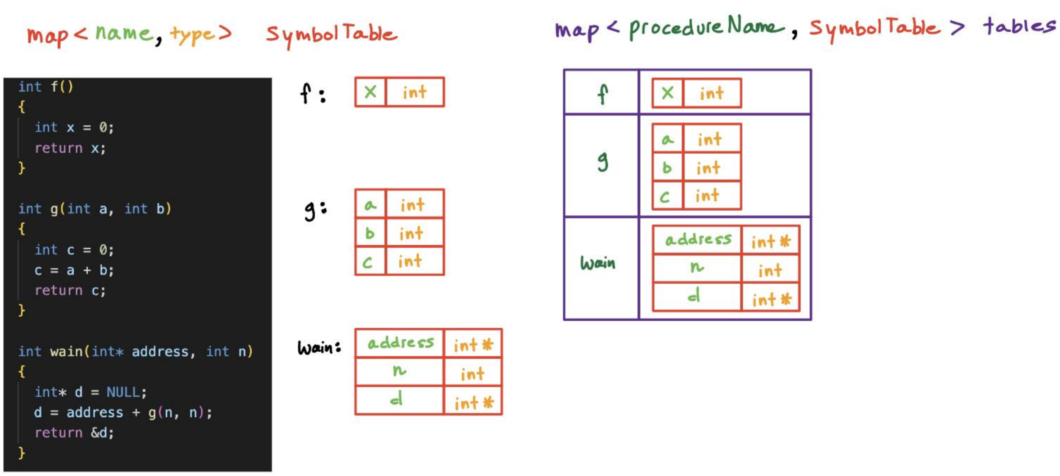
1) Duplicate Identifiers

Identifier Types

1. Variables
2. Procedures

Symbol Table

Separate symbol table for each procedure
Maps variable's to their type



Examples	

<pre>int f() {
  int x = 0;
  return x;
}
int wain(int a, int b) {
  int x = 0;
  return x;
}</pre>
	Valid
	
	Can't have duplicates within a procedure

<pre>int f() {
  int x = 0;
  return x;
}
int wain(int a, int b){
  return x;
}</pre>
	Error
	
	x is not defined in procedure wain

<pre>int f() {
  int x = 0;
  return x;
}
int f() {
  int x = 0;
  return x;
}
int wain(int x, int y){
  return f() + x;
}</pre>
	Error
	
	All procedure names must be unique

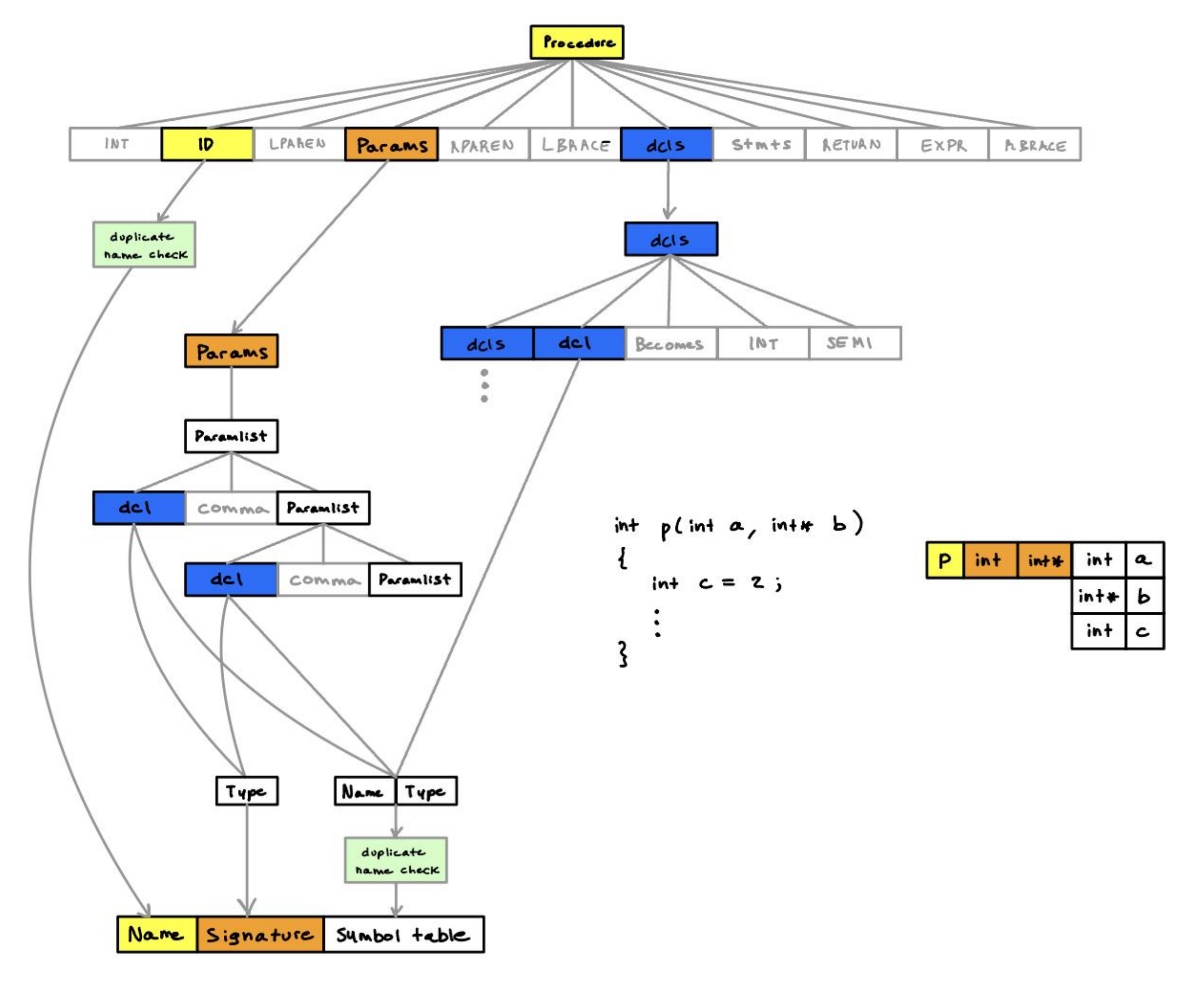
<pre>int f() {
 int f = 1;
 return f + 1;
}
int g(int g) {
  return g - 1;
}
int wain(int a, int b){
  return a;
}</pre>
	It is legal to have variables that have the same name as any procedure
	
	Note: All occurrences of the identifier refer to the variable and not the procedure

<pre>int p(int p) {
  return p(p);
}</pre>
	Error
	All occurrences of "p" are treated as a variable and not a procedure even though a procedure with that name exists.

Implementation (Tree Traversal)

Step 1

1. Duplicate identifier (procedure, variable) checks
2. Build SymbolTable



2) Defined Before Use

- Variables are always declared (in **dcl's**) before they are used (in **statements/RETURN**)
- SymbolTable of the procedure is complete by the time the traversal reaches **statements**

Variables

Traverse the **statements** subtree and the **return** expression. Look for:

- factor → ID

- lvalue → ID

check that **ID** lexeme exists in the procedure's symbolTable

3) Checking Procedure Calls

during the creation of **tables**

1) Name Defined

Traverse the **statements** subtree and the **return** expression. Look for these rules:

1. factor → ID LPAREN RPAREN
2. factor → ID LPAREN arglist RPAREN

Check if **ID** has been declared in **tables**

Not found = ERROR

2) signature match

Signature:

1. # arguments
2. types of each argument

We should store the **signature** when a new procedure declaration is discovered

Obtaining Signature

params subtree, either:

1. params → .EMPTY
2. params → paramlist
    a. Traverse children of **paramlist**. Either:
        i. paramlist → dcl
        ii. paramlist → dcl COMMA paramlist

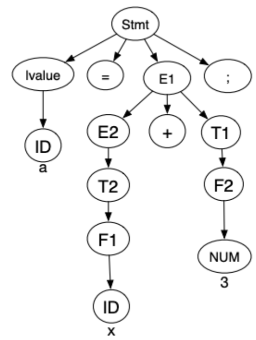
When our traversal reaches a use of an **procedure call**

1. checking that the procedure had already been declared
2. check that the signature matches

Type Errors

<table>
<tr><td colspan="2">Example: How to catch a type error</td></tr>
<tr><td>a = x + 3

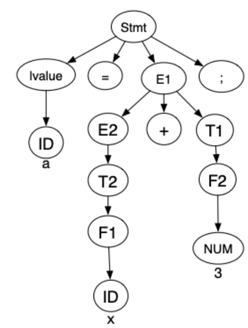
required: type(a) = type(x+3)

Must determine the types for the left and right expressions.</td><td>Parse tree:

</td></tr>
<tr><td>Type of LHS</td><td>Determined by traversing the leftmost child of Stmt
1. Determine the type for the lvalue
2. type of lvalue = type of only child, ID
3. type of ID = symbolTable[a]</td></tr>
<tr><td>Type of RHS</td><td>Determined by traversing the third child(E1)
1. compute the type of the Expr (E2)
  a. type(E2) = type(T2)
  b. type(T2)=type(F1)
  c. type(F1)=type(ID)
  d. type(ID)=symbolTable[x]
2. compute the type of term (T1)
  a. type(T1) = type(F2)
  b. type(F2) = type(NUM)
  c. type(NUM) = int
3. Apply the type system rule for addition
  a. int if type(x)=int
  b. int* if type(x)=int*</td></tr>
<tr><td colspan="2">Hence, type(LHS)=type(RHS)</td></tr>
</table>

Type Inference Algorithm:

```
void typeOf ( Tree & tree )
{
 for each child c of tree
 {
   typeOf ( c );      // recursively compute type of each child subtree
 }
// refer to the relevant type system rule for this tree node
// use the computed types of children to determine if the rule is violated
// if it is not violated , store the computed type in the tree node
}
```

<table>
<tr><td colspan="2">a = x + 3</td></tr>
<tr><td>Type-System-Rule(=): type(a) = type(x+3)

Type-System-Rule(+):
- type(x) = int, type(3) = int
- type(x)=int*, type(3) =in</td><td>
</td></tr>
</table>

3) Type Checking

2 Types

1. int
2. int*

<table>
<tr><td>typeOf(tree)</td></tr>
<tr><td>for each child (c) of tree :
{
    //recursively assign each child a type
    typeOf(c)
}</td></tr>
</table>

//refer to the **type system rule** for this tree node
//use the children types to determine if the rule is violated
//if not violated, store the type in this tree.

Type System/Inference Rules

Premise	Result	Notation
ID is declared with type τ	τ	$\frac{<ID.name,\, \tau> \epsilon\ declarations}{ID.name : \tau}$
Number	int	$\frac{}{NUM : int}$
NULL	int*	$\frac{}{NULL : int*}$
Parentheses	type same	$\frac{E : \tau}{(E) : \tau}$
Taking the address of an int	int*	$\frac{E : int}{\&E : int*}$
Dereferencing an int*	int	$\frac{E : int*}{*E : int}$
new int[E]	int*	$\frac{E : int}{new\ int[E] :\ int*}$
int * int	int	$\frac{a : int\ ,\ b : int}{a*b : int}$
int / int	int	$\frac{a : int\ ,\ b : int}{a/b : int}$
int % int	int	$\frac{a : int\ ,\ b : int}{a\%b : int}$
int + int ⇒ int	int	$\frac{a : int\ ,\ b : int}{a+b : int}$
int* + int ⇒ int*	int*	$\frac{a : int*\ ,\ b : int}{a+b : int*}$
int - int ⇒ int	int	$\frac{a : int\ ,\ b : int}{a-b : int}$
int* - int ⇒ int*	int*	$\frac{a : int*\ ,\ b : int}{a-b : int*}$
int* - int* ⇒ int	int	$\frac{a : int*\ ,\ b : int*}{a-b : int}$
signature matches	int	$\frac{<f, t1, ..., tn> \epsilon\ declarations,\ E1 : t1, ...\ ,\ En : tn}{f(E1, ...\ , En) : int}$

We specify only what is allowed, what is not allowed is everything not specified.

Type Checking Statements

Expressions	Statements (contain expressions)
produce values (have a type)	Do not produce values (no type)
We **infer** the types of expressions	Can't **infer** a type

well-typed statement: components are well-typed

An **expression** is well-typed if a type can be inferred	$\frac{E : \tau}{E \textit{ is well type}}$

Statement	Well Typed If and only if
println	parameter has type int: $\frac{E : int}{println(E) \textit{ is well type}}$
delete	Parameter has type int*: $\frac{E : int^*}{delete[]\ E \textit{ is well typed}}$
assignment: a = b	type(LHS)=type(RHS) $\frac{a : \tau,\ b : \tau}{a{=}b \textit{ is well typed}}$ *no need to check that the LHS is an lvalue (already enforced by CFG)
empty sequence of statements	always well typed
sequence of statements is well-typed	each statement in the sequence is well-typed: $\frac{S1 \textit{ is well typed}, S2 \textit{ is well types}}{S1\ S2 \textit{ is well typed}}$
test is well-typed	operands for the comparison are of the same type: $\frac{a : \tau,\ b : \tau}{a{<}b \textit{ is well typed}}$
if statement	components of the statement are well-typed:

	$\dfrac{well{-}typed(test), well{-}typed(S1), well{-}typed(S2)}{well{-}typed(if\ (test)\ \{S1\}\ else\ \{S2\})}$
while statement	components of the statement are well-typed: $\dfrac{well{-}typed(test), well{-}typed(S)}{well{-}typed(while\ (test)\ \{S\})}$
empty sequence of declarations	well typed
Variable that is declared to be an integer	it is initialized with an integer value $\dfrac{well{-}typed(dcls),\ \langle ID.name, int\rangle \in declarations}{well{-}typed(dcls\ int\ ID = NUM;)}$
Variable that is declared to be a pointer	it is initialized with a NULL value $\dfrac{well{-}typed(dcls),\ \langle ID.name, int*\rangle \in declarations}{well{-}typed(dcls\ int*\ ID = NULL;)}$
Procedure	Declarations and statements are well-typed and returns an integer: $\dfrac{well{-}typed(dcls), well{-}typed(S), E : int}{well{-}typed(int\ ID(dcl_1, \ldots, dcl_n)\{dcls\ S\ return\ E;\})}$
wain procedure	second parameter is an INT the declarations and statements are well-typed and the procedure returns an integer $\dfrac{dcl2 : int, well{-}typed(dcls), well{-}typed(S), E : int}{well{-}typed(int\ wain(dcl1, dcl2)\{dcls\ S\ return\ E;\})}$

Note:

- WLP4 does require that all variables be initialized before they are used but does so through the context-free grammar; a variable when declared must be initialized to a value.
- The only way to return from a function is through the last statement of the function.