**Context-Sensitive Languages**

* **Context-sensitive analysis**
* Which properties of valid C programs still cannot be enforced by CFGs?
  + Declaration before use
  + Const correctness
  + Type correctness
  + Multiple-declared variables
* Context-sensitive language
  + E.g. dA → dab; d provides context
* Declaration errors
  + Multiple or missing declarations
  + Solution: construct a symbol table
    - E.g. map<string (name), string (type)>
  + i.e. traverse parse tree to collect all declarations
  + for each node with rule “dcl → TYPE ID”
    - extract name and type
    - if name already in symbol table, ERROR
    - else add (name, type) to symbol table
* Variable use w/o declaration
  + Solution: traverse parse tree and look for rules with RHS “ID”
    - If ID name is not in symbol table, ERROR
* These two passes of the parse tree can be merged
* Procedures
  + How do we implement scopes – i.e. forbid duplicate definitions in the same procedure, but allow in different procedures?
    - Solution: separate symbol table for each procedure & a top-level symbol table that holds procedure names
    - E.g. map<string (procedure name),

map<string, string> (procedure symbol table) >

* + - For each node with rule
      * “procedure → INT ID LAREN …” or “main → INT WAIN …”
      * If name already in symbol table, ERROR
      * Else add procedure to symbol table
    - Also need a global variable to keep track of current procedure
  + There is type information associated with procedures, i.e. procedure signature
    - Since all procedures in WLP4 return int, the signature is just the sequence of parameter types
    - Revised symbol table:
      * map<string (procedure name),

pair<vector<string> (signature),

map<string, string> (procedure symbol table) > >

* + - To compute the signature, traverse nodes with rule
      * “paramlist → dcl” or “paramlist → dcl COMMA paramlist”
      * If paramlist → ε then signature is empty
* Types
  + Purpose: the computer does not know exactly what each byte in its memory is used for
  + Types remember what kind of data is stored at each location in the memory
  + Ensures that the memory location is always used that way
  + Only 2 types in WLP4: int and int\*
  + To determine the type of a node in the parse tree:

String typeof(Tree &t) {

For each c in t.children

Compute typeof(c)

Use t.rule to decide which type rules are relevant

Combine rule & types of children to get type for t

If not possible, ERROR

}

* + ID
    - Get type from symbol table
    - Use notation for natural deduction inference rule:

<id.name, T> ∈ symbol table ← premise(s)

id : T ← conclusion

* + - i.e. if ID was declared to have type T, then it has type T
  + Singleton production (e.g. expr → term, term → factor)
    - Type of LHS = type of RHS
  + NUM

\_\_\_\_\_\_\_\_\_\_\_ ← no premise

NUM : int ← conclusion (i.e. this is an axiom)

* + NULL

\_\_\_\_\_\_\_\_\_\_\_

NULL : int\*

* + Parenthesized expressions

E : T

(E) : T

* + - i.e. parenthesized expression has the same type of the expression itself
  + Address operator (AMP)

E : int

&E : int\*

* + - If E is a variable value (int), then &E is its address (a pointer)
  + Dereference operator (STAR)

E : int\*

\*E : int

* + - If E is a pointer, then \*E is a variable value (int)
  + Memory allocation (NEW)

E : int

new int [E] : int\*

* + Multiplication/division

E1 : int E1 : int E1 : int

E2 : int E2 : int E2 : int

E1 \* E2 : int E1 / E2 : int E1 % E2 : int

* + Addition

E1 : int E1 : int E1 : int\*

E2 : int E2 : int\* E2 : int

E1 + E2 : int E1 + E2 : int\* E1 + E2 : int\*

* + - Int\* + int\* is not valid
  + Subtraction

E1 : int E1 : int\* E1 : int\*

E2 : int E2 : int E2 : int\*

E1 – E2 : int E1 – E2 : int\* E1 – E2 : int

* + - Int – int\* is not valid
  + Function calls

<f, <T1 … Tn> > ∈ symbol table

E1 : T1 … En : En

f(E1 … En) : int

* + - Make sure f & its signature is declared in symbol table
    - Make sure signature of f matches with provided params
    - All functions return int in WLP4
  + Loops & ifs
    - i.e. while T { S }
    - i.e. if (T) { S1 } else { S2 }
    - T should be boolean – but no boolean type in WLP4
      * WLP4 grammar ensures that T is a comparison (e.g. expr COMPARE expr)
      * Make sure exprs are well-typed and the comparison makes sense
    - Every element (e.g. T, S) should be well-typed (verify recursively)
  + Tests

E1 : T E1 : T E1 : T

E2 : T E2 : T E2 : T

well-typed(E1 == E2) well-typed(E1 != E2) well-typed(E1 < E2)

* + - Where:

E : T

well-typed(E)

* + - Means E satisfies the type rules, but might not actually have a type
  + BECOMES

E1 : T

E2 : T

well-typed(E1 = E2)

* + PRINTLN

E : int

Well-typed(println(E))

* + Memory deallocation (DELETE)

E : int\*

Well-typed(delete [] E)

* + Sequences

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (empty sequence) well-typed(S1)

Well-typed(ε) well-typed(S2)

Well-typed(S1 S2)

* **Code generation**
* Source → lexical analysis (scanning)
  + → [tokens] → context-free analysis (parsing)
  + → [parse tree] → context-sensitive analysis (semantic analysis)
  + → [parse tree, symbol table] → code generation
* At this point, we have guaranteed for the source program to be free of compile-time errors
* Now we need to generate equivalent MIPS code
* Ex: int wain(int a, int b) { return a; }
  + Conventions: params in $1, $2; output passed in $3
  + MIPS:

Add $3, $1, $0

Jr $31

* Ex: int wain(int a, int b) { return b; }
  + MIPS:

Add $3, $2, $0

Jr $31

* These two programs have the same parse tree – how to tell one from another?
  + Use symbol table
  + Add column to table for where each symbol is stored (register location)
  + When traversing the parse tree for code generation, when ID is countered → lookup in symbol table to get location
  + For local variables & params of wain – allocate space on the stack
  + Reference local variables by the amount of offset from $30
  + But offsets change as other things are added to/removed from the stack
* Ex: int wain(int a, int b) { return a; }
  + MIPS:

sw $1, -4($30) //store a, b

sw $2, -8($30)

lis $4

.word 8

sub $30, $30, $4 //adjust SP

lw $3, 4($30) //output a (a is stored 4 from SP)

//this offset will change if more items are pushed onto stack

add $30, $30, $4 //adjust SP

jr $31

* Introduce conventions:
  + $4 always contains 4 – do not change it
  + $29 points to the bottom of stack frame (frame pointer) – location of local variables will be calculated using offsets to $29, which remain constant
* Ex: int wain(int a, int b) { int c = 0; return a; }
  + MIPS:

lis $4 //$4 = 4

.word 4

sw $1, -4($30) //store a, b, c & adjust SP

sub $30, $30, $4

add $29, $30, $0 //set up SP to point at first local variable stored on stack

sw $2, -4($30)

sub $30, $30, $4

sw $0, -4($30)

sub $30, $30, $4

//symbol table:

//a = 0 from $29; b = -4 from $29; c = -8 from $29

lw $3, 0($29) //output a

add $30, $29, $4 //restore SP value = FP + 4

* More complicated program:
* In general, for each grammar rule A → γ, build code(A) from code(γ)
* Introduce conventions:
  + Use $3 for output of all expressions
    - But if there are multiple pending operations – where to store those results without overwriting $3?
* Ex: int wain(int a, int b) { return a + b; }
  + MIPS: