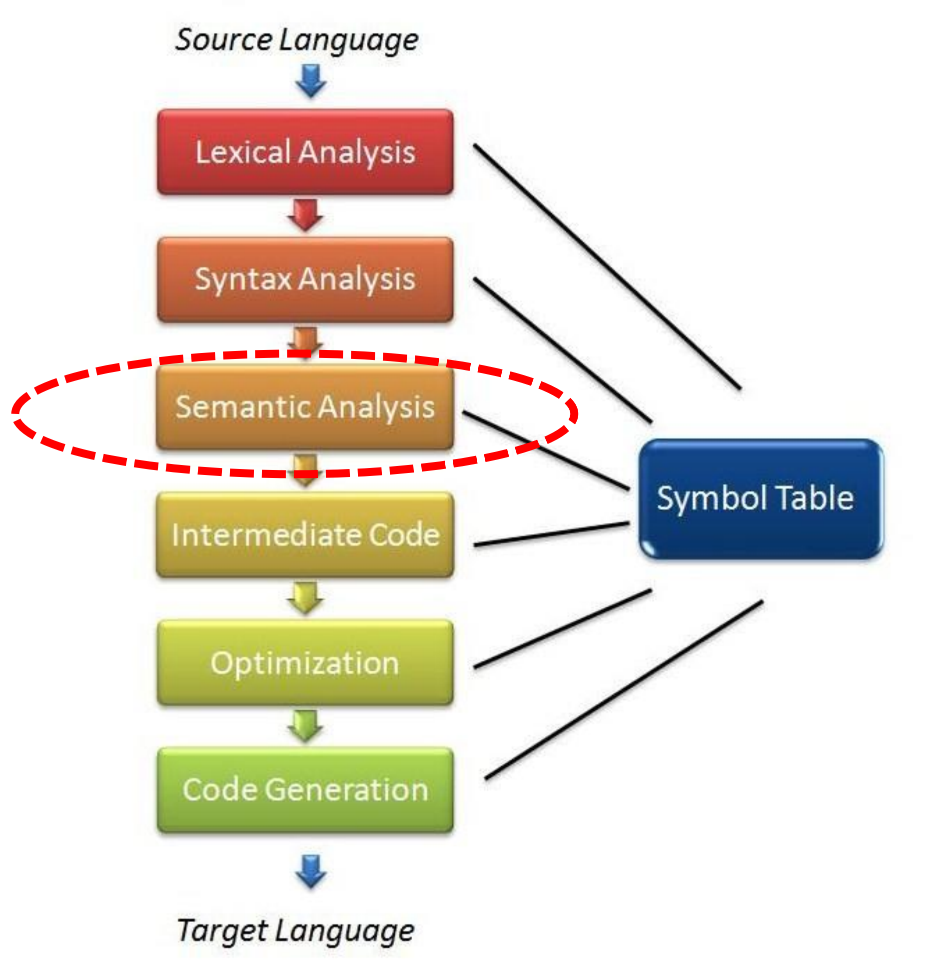
Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Compiler Structure



The Compiler So Far: error detection

- Lexical analysis
  - Detects inputs with illegal tokens
    - e.g.: `main$ ();`
- Syntactic analysis
  - Detects inputs with ill-formed parse trees
    - e.g.: missing semicolons
- Semantic analysis
  - Catches all remaining errors

Beyond Syntax

What's wrong with this code?

(Note: it parses perfectly)

```
foo(int a, char * s){ return 0; }

int bar() {
  int f[3];
  int i, j, k;
  char *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 5;
  j = i + k;
  printf("%s,%s.\n",p,q);
  goto label23;
}
```

Examples of semantic errors

- Undeclared identifiers
  - Variable not in scope
- Identifiers declared multiple times
- Index out of bounds (compile/run time)
- Wrong number/types of arguments in a function call
- Incompatible types for some operation
- Break statement outside switch/loop
- Goto with no/non-existing label
- .....

Inlined TypeChecker

You can type check as part of semantic actions:

Bison/Yacc example:

```
Expr : Expr '+' Expr
       {
          if ($1.type == $3.type &&
             ($1.type == IntType || $1.type == RealType)) {
            $$.type = $1.type;
          } else {
            error("operator + applied to wrong type!");
          }
          generateAdd($1, $3, $$);
       }
```

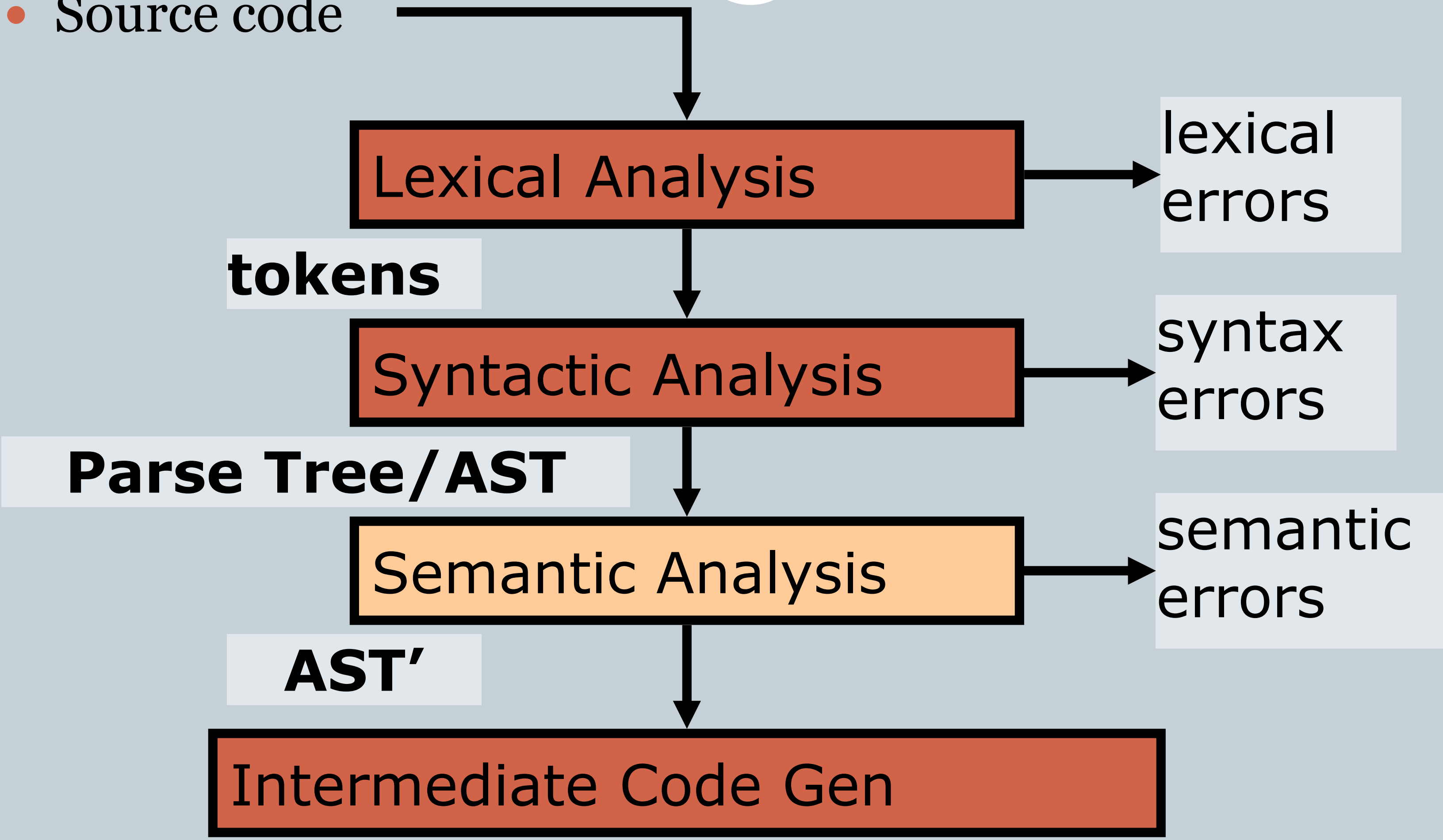
Problems

- Difficult to read
- Difficult to maintain
- Compiler must analyze program in the same order as the parser

- Instead … in most compilers the tasks are split up

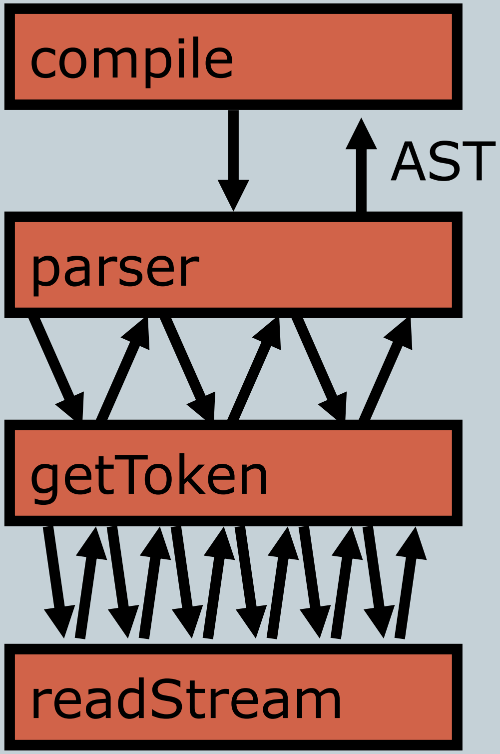
Semantic Analysis

- Source code



Compiler ‘main program’

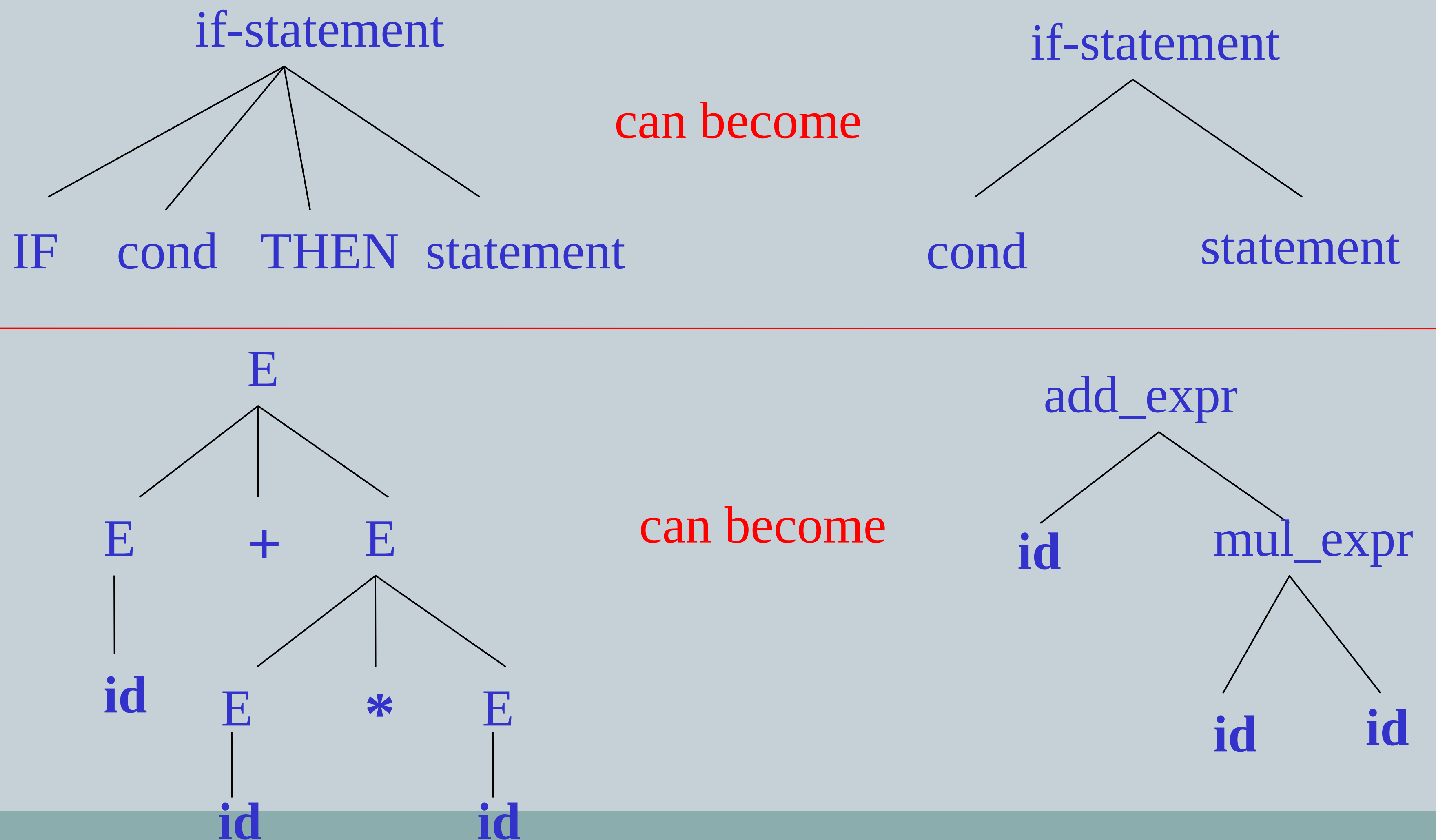
```
void compile() {
  AST tree = parser(program);
  if (typeCheck(tree)) {
    IR ir = GenIntermedCode(tree);
    emitCode(ir);
  }
}
```



Abstract Syntax Tree (AST)

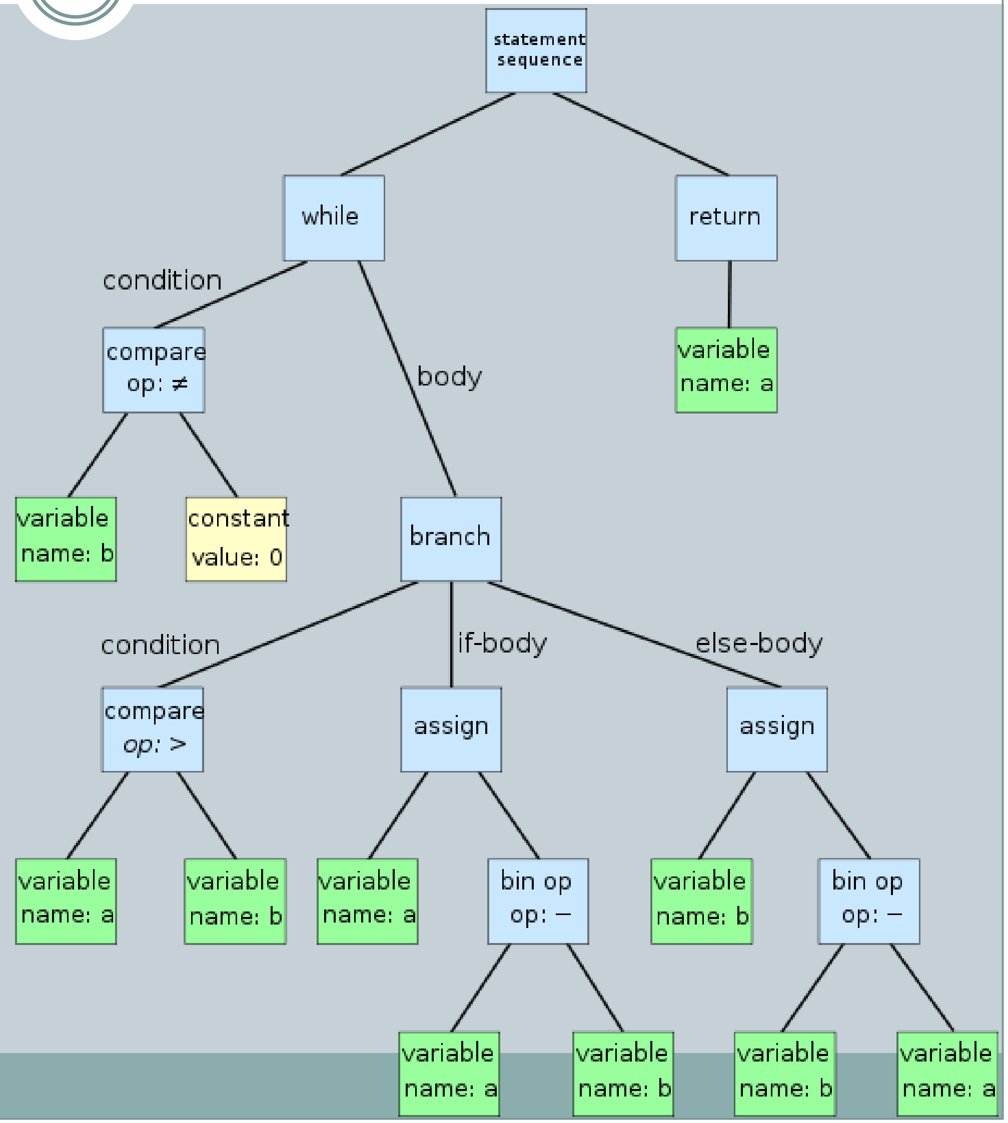
- The parse tree contains too much detail
  - e.g. unnecessary terminals such as parentheses
  - depends heavily on the structure of the grammar (e.g. intermediate non-terminals)
- Idea:
  - strip the unnecessary parts of the tree, simplify it.
- AST
  - Encodes the syntactic structure of the program while providing abstraction of the original grammar.
  - Can be easily annotated with semantic information (attributes) such as type, numerical value, etc.

Abstract Syntax Tree



Abstract Syntax Trees (AST)

```
while (b != 0) {
   if (a > b)
      a = a - b;
   else
      b = b - a;
}
return a;
```



Abstract Syntax Trees (AST)

- How to build ASTs?
  - Augment the parser with actions

- $T \rightarrow T * F$ { make mult tree node; left,right subtrees are ASTs of T, F}
- $T \rightarrow F$ { return AST of F }
- $F \rightarrow (E)$ { return AST of E }
- $F \rightarrow$ number { make number leaf node; }
- $F \rightarrow$ ident { make identifier leaf node; }

Beyond Syntax Analysis

- An identifier named x has been recognized.
  - Is x a scalar, array or function?
  - How big is x?
  - If x is a function, how many and what type of arguments does it take?
  - Is x declared (before being used)?
  - Is the expression x + y type-consistent?

- Semantic analysis collects information about the types of identifiers and checks for type related errors.

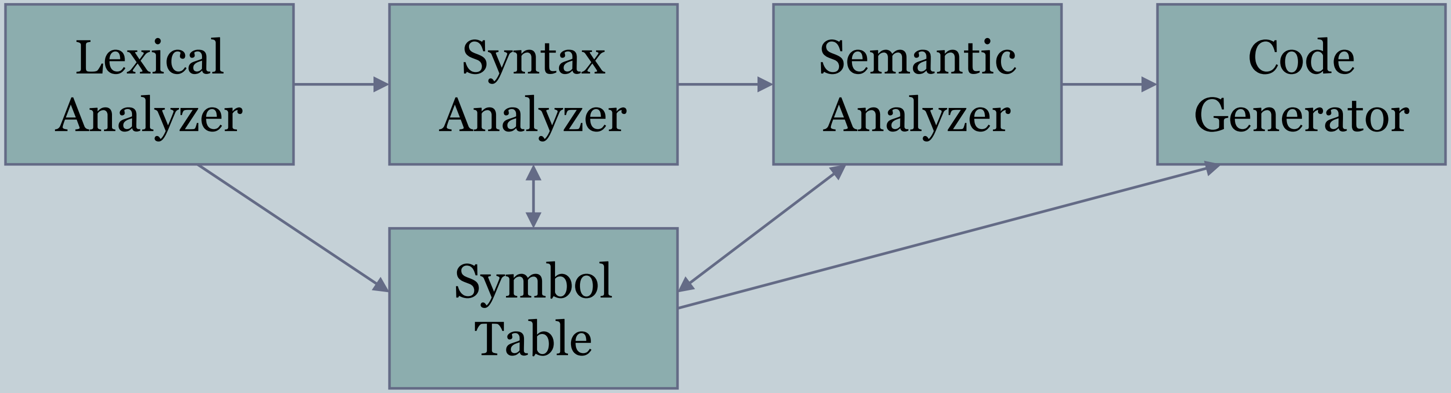
identifiers

- Type of identifiers (C like languages):
  - Simple variables: local/global
  - Type/struct names
  - Function names
  - Formal parameters
  - Struct fields
  - …

The Symbol Table

- A *symbol table* is a datastructure that
  - tracks the *current* bindings of identifiers.
  - holds all relevant information about identifiers.

- When identifiers are found, they are entered into the symbol table.
  - Typically by the lexer



Symbol Table

- When a symbol is first encountered by the lexer, we do not yet know its type.
  - That is determined later by the parser.

- For example, we could encounter the symbol `count` in any of the following contexts.

```
int count;
int count(int n, int a[]);
struct count{…};
```

Symbol Table Entries

- Symbol information is stored in a type called **`IdEntry`**.
  - Usually it is a pointer to a **`struct idEntry`**.
- The information may not all be known at once.
- We may begin by knowing only the name (in the lexer).
- A bit later (in the parser), we know
  - the data type
  - other info, like the block level, and scope

Symbol Table ADT

- The basic symbol-table functions are:
  - `IdEntry install(String s[, int blkLev])`
  - `IdEntry idLookup(String s)`
  - `void remove(String s)`

- The **`install()`** function:
  - creates an IdEntry struct and stores it in the table.
  - returns a handle/pointer to the **`idEntry`** struct.

Hash Tables

- A *hash table* is a ‘database’ in which each member is accessed through a *key*.
- The key is used to determine a location (index) where to store the member in the table.
- The function that produces a location from the key is called the *hash* function.
- For example, for a hash table of strings, a simple/naive hash function might compute the sum of the ASCII values of the characters of the string, modulo the size of the table.

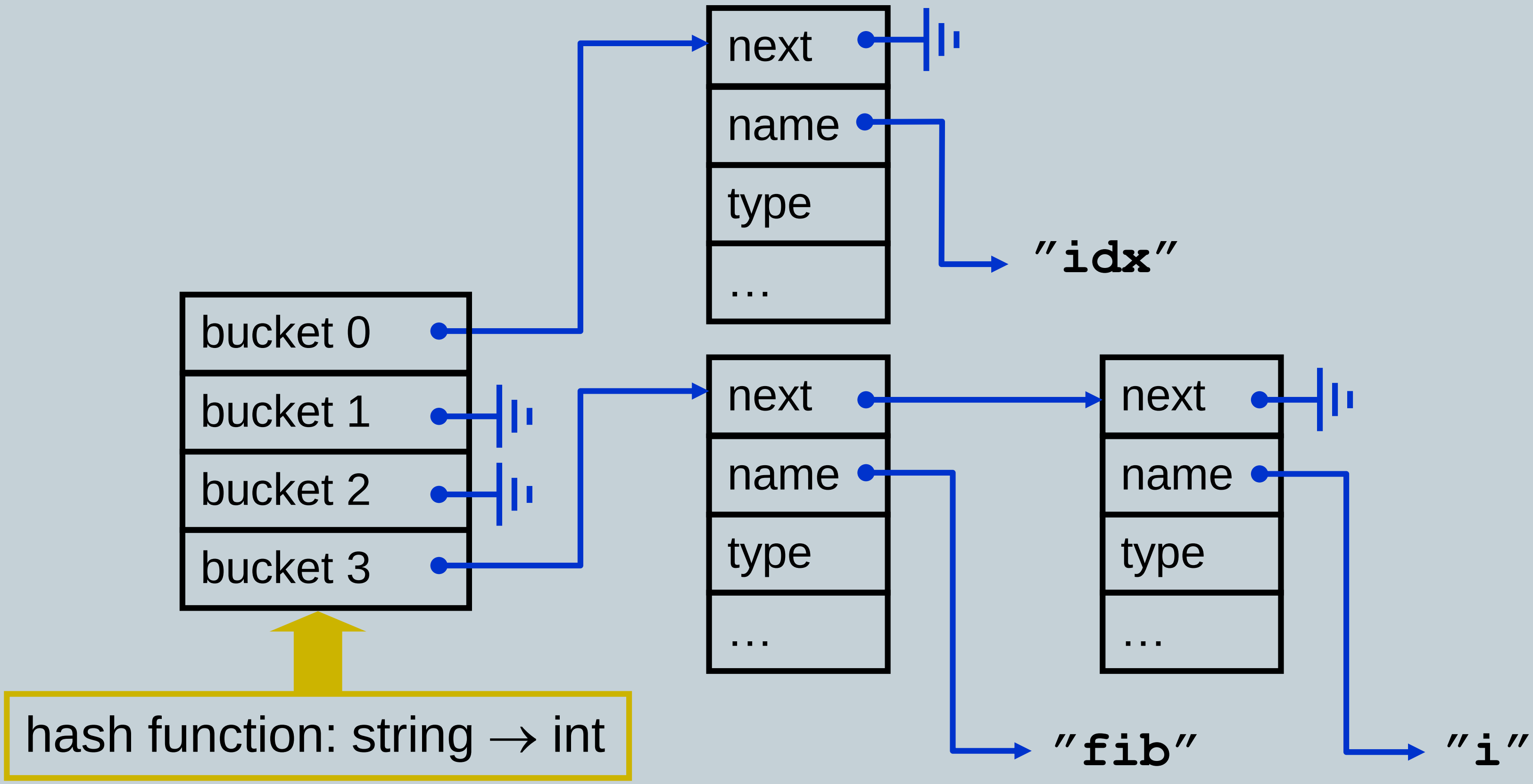
Clashes and Buckets

- Clearly, there is the possibility of a clash: two members have the same hashed key.

- Several solutions:
  - hash table contains lists (called “buckets”) of values that map to the same location.
  - linear probing
  - secondary hashing
  - ...

Symbol table implemented as a hash table

extensible string-indexable array



Looking up a Symbol

- If an identifier is declared both globally and locally, which one will be found when it is looked up?
- If an identifier is declared only globally and we are in a function, how will it be found?
- These ‘problems’ can be solved using a block level based symbol table.

Scope of variables

- Scope of an identifier= portion of the program in which the identifier is “visible”
  - Identifier refers to ‘closest’ enclosing definition

```
int x; /* global x */

int main () {
  int x;   /* another local x */
  for (x=0; x < 10; x++) {
    int x; /* another one (don't try this at home!) */
    ...
  }
  ...
}
```

Symbol Table Entries

- Typically, the following information about identifiers is stored.
  - The name (as a string).
  - The data type.
  - The block level.
    - Its scope (global, local, or parameter).
  - Its offset from the base pointer (for local variables and parameters only).

Block Levels

- Global variables, and local variables are stored at different block levels.
- In C, blocks are delimited by braces { }.
  - Exception: globals
- In C, variables local to a block must be declared at the beginning of the block.
- Every time we enter a block, the block level is increased by 1 and every time we leave a block, it is decreased by 1.

Keeping Track of variables

We need a way to keep track of all identifier types in scope

```
{
    int i, n = …;
    for (i=0; i < n; i++) {
        float b = …
    }
}
```

i → int
n → int

i → int
n → int
b → float

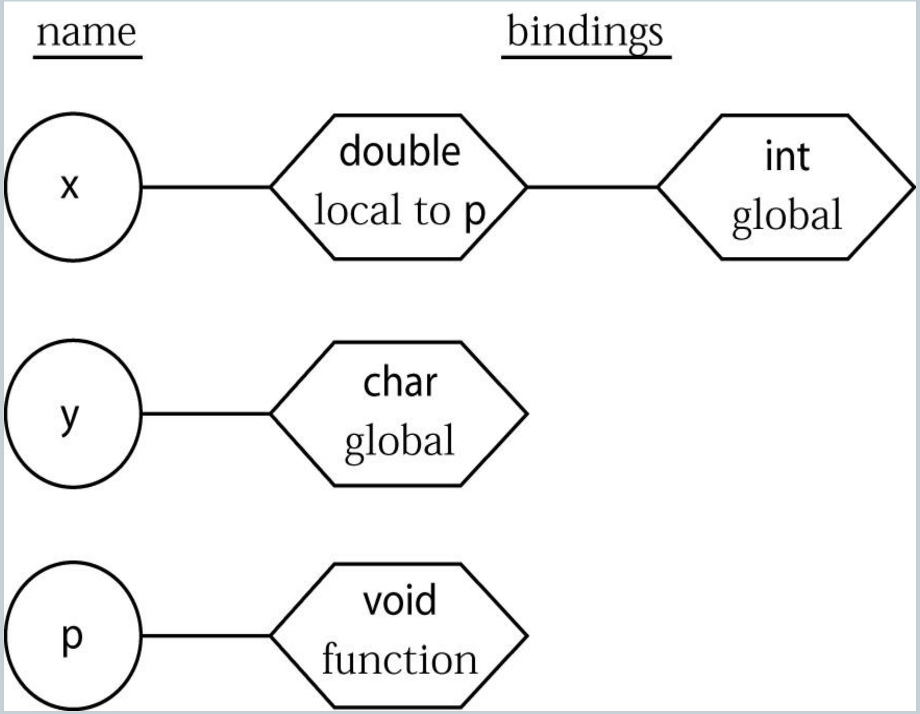
?

Block level structured Symbol Table

- Each symbol has a block level.
  - (Block level 0 = Keywords)
  - Block level 1 = Global variables.
  - Block level 2 = Parameters and local variables.
  - Block level 3,... = local variables in nested scope.

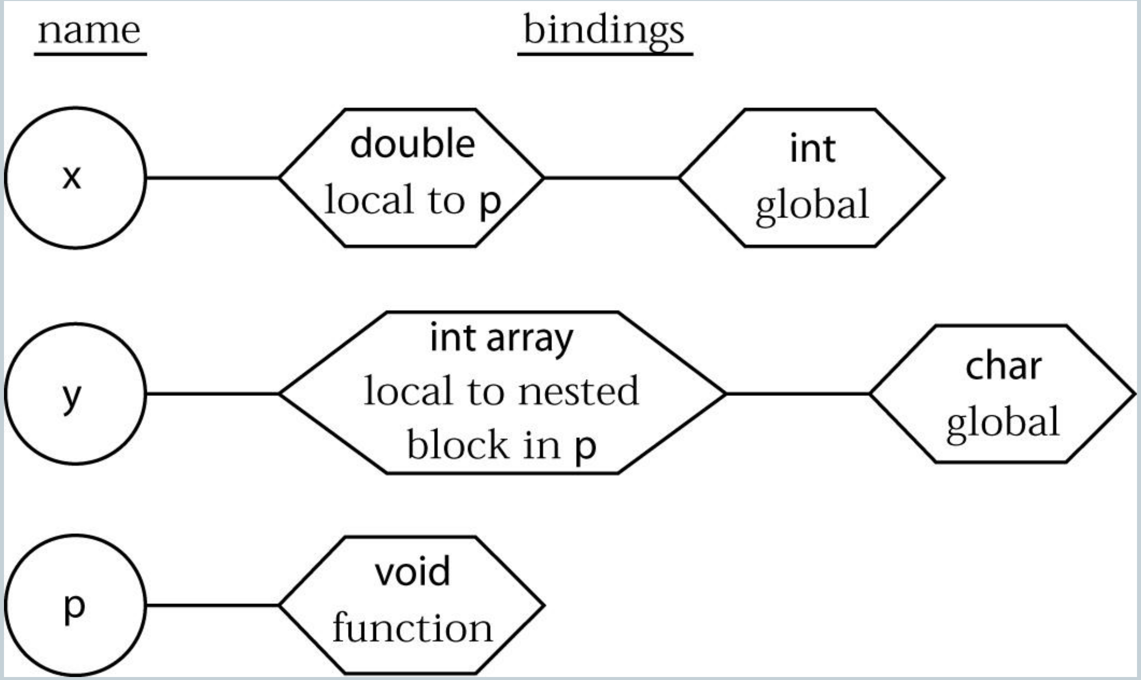
Scope

```
int x;
char y;

void p(void) {
 double x;
    …
 while (…) {
   int y[10];
   …
 }
 …
}

void q(void){
 int y;
 …
}

main() {
  char x;
  …
}
```



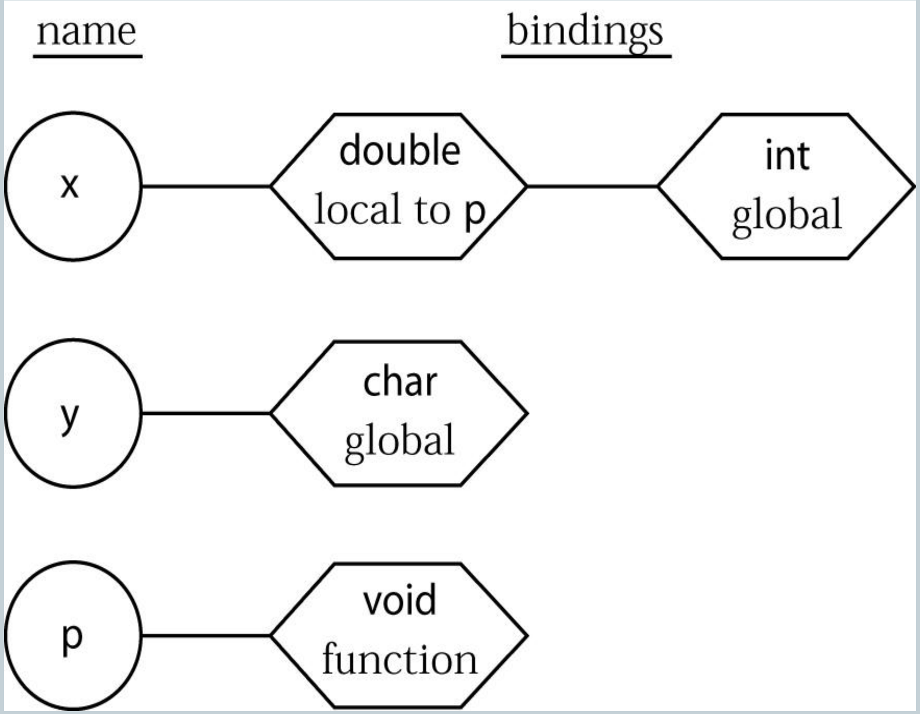
Scope

```
int x;
char y;

void p(void) {
 double x;
    ...
 while (...) {
   int y[10];
   ...
 }
 ...
}

void q(void){
 int y;
 ...
}

main() {
  char x;
  ...
}
```



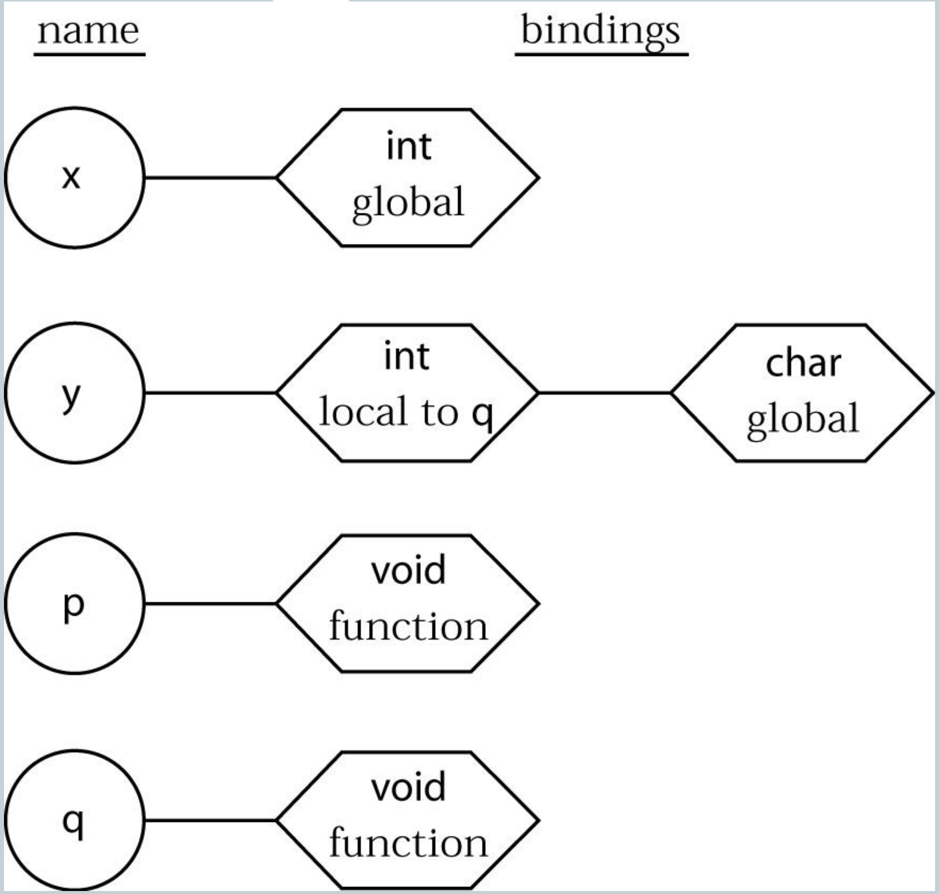
Scope

```
int x;
char y;

void p(void) {
 double x;
    …
 while (…) {
   int y[10];
   …
 }
 …
}

void q(void){
 int y;
 …
}

main() {
  char x;
  …
}
```



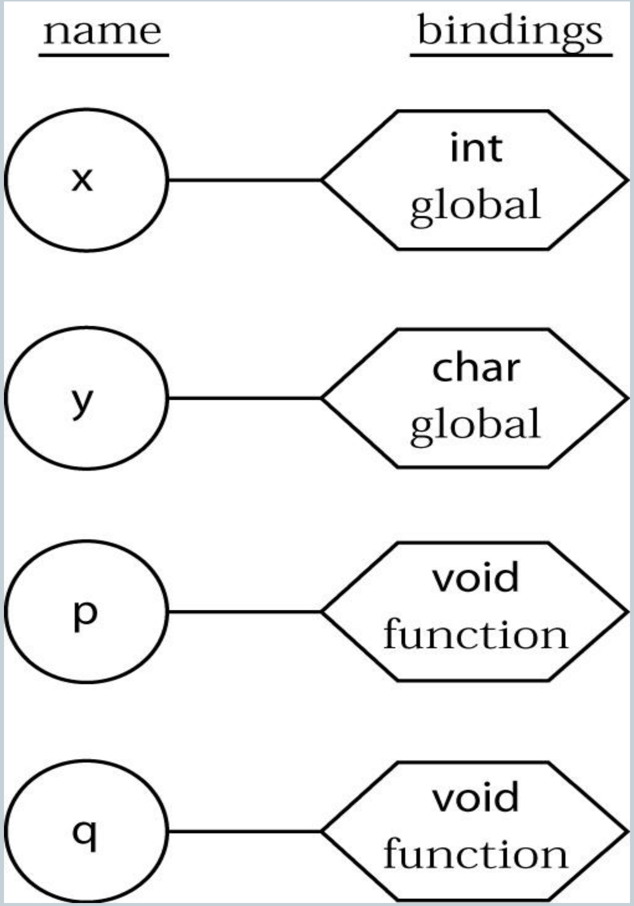
Scope

```
int x;
char y;

void p(void) {
 double x;
   ...
 while (...) {
   int y[10];
   ...
 }
 ...
}

void q(void){
 int y;
 ...
}

main() {
  char x;
  ...
}
```



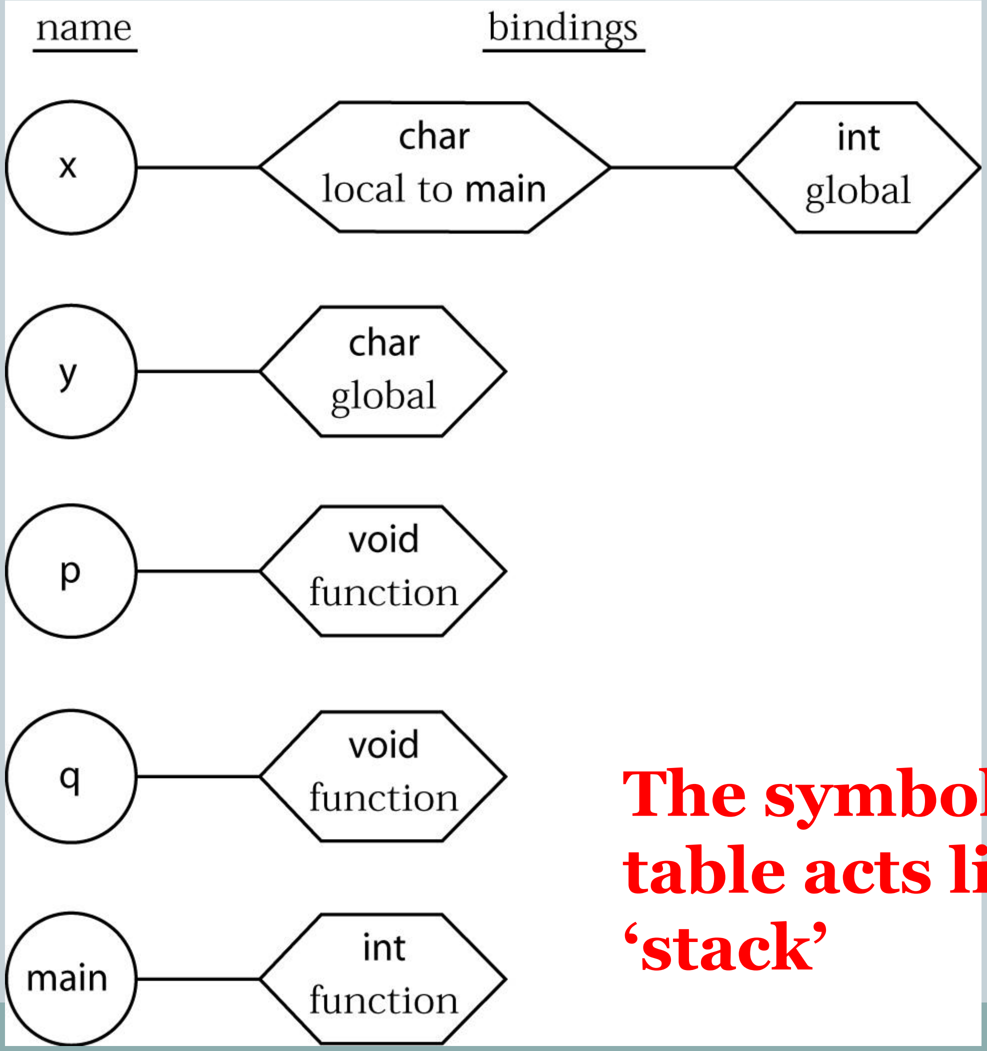
Scope

```
int x;
char y;

void p(void) {
 double x;
    ...
 while (...) {
   int y[10];
   ...
 }
 ...
}

void q(void){
 int y;
 ...
}

main() {
  char x;
  ...
}
```



Scope

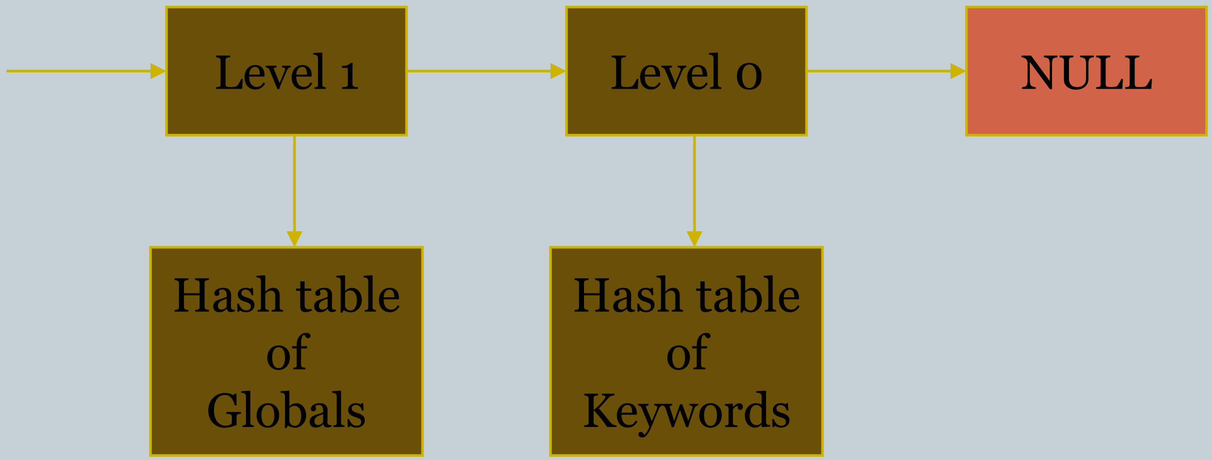
```
int x;
char y;

void p(void) {
 double x;
    …
 while (…) {
   int y[10];
   …
 }
 …
}

void q(void){
 int y;
 …
}

main() {
  char x;
  …
}
```



The symbol table acts like a 'stack'

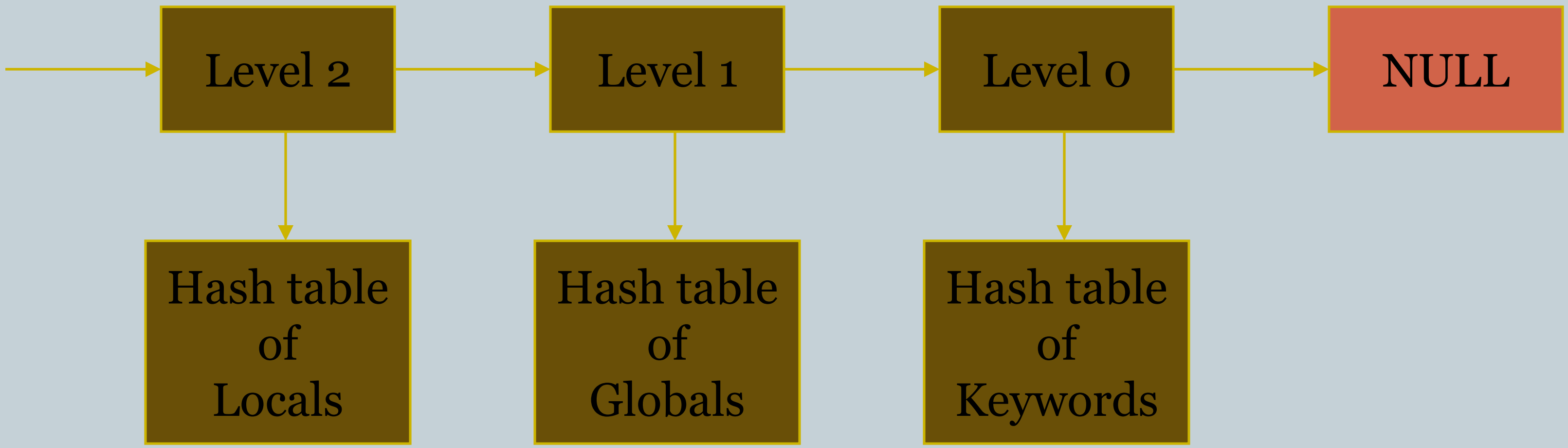
Block level structured Symbol Table

- A possible implementation of the symbol table is a linked list of *hash tables*, one hash table for each block level.
  - In fact, it is used as a stack of hash tables



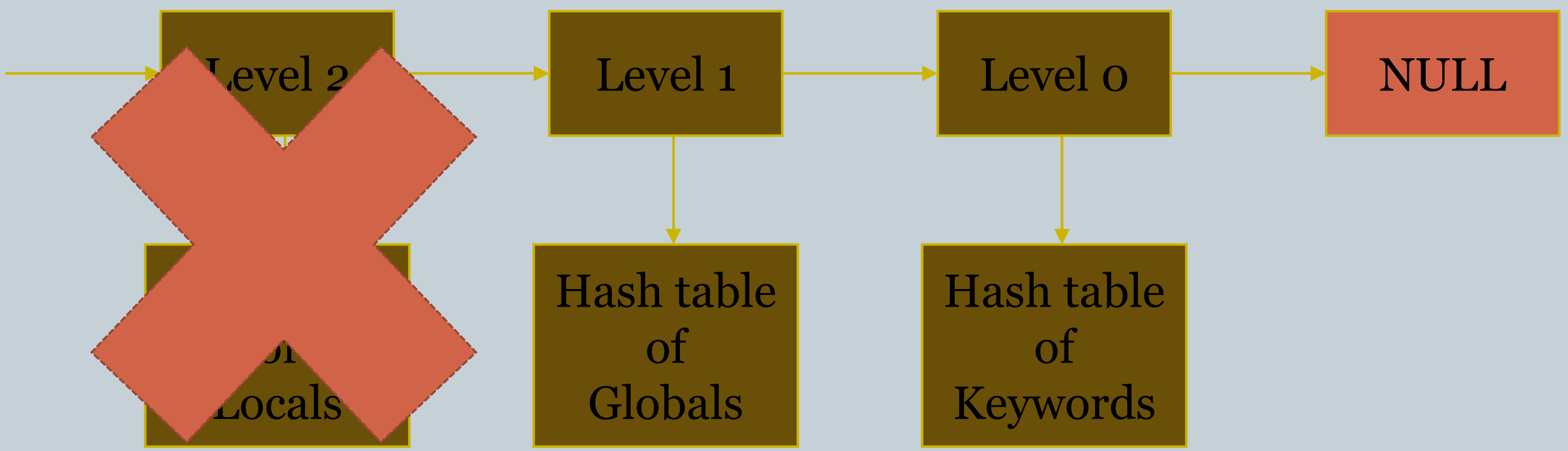
Structure of the Symbol Table

- When the parser enters a new block, it creates a level 2 (or higher) hash table and stores parameters and local variables there.



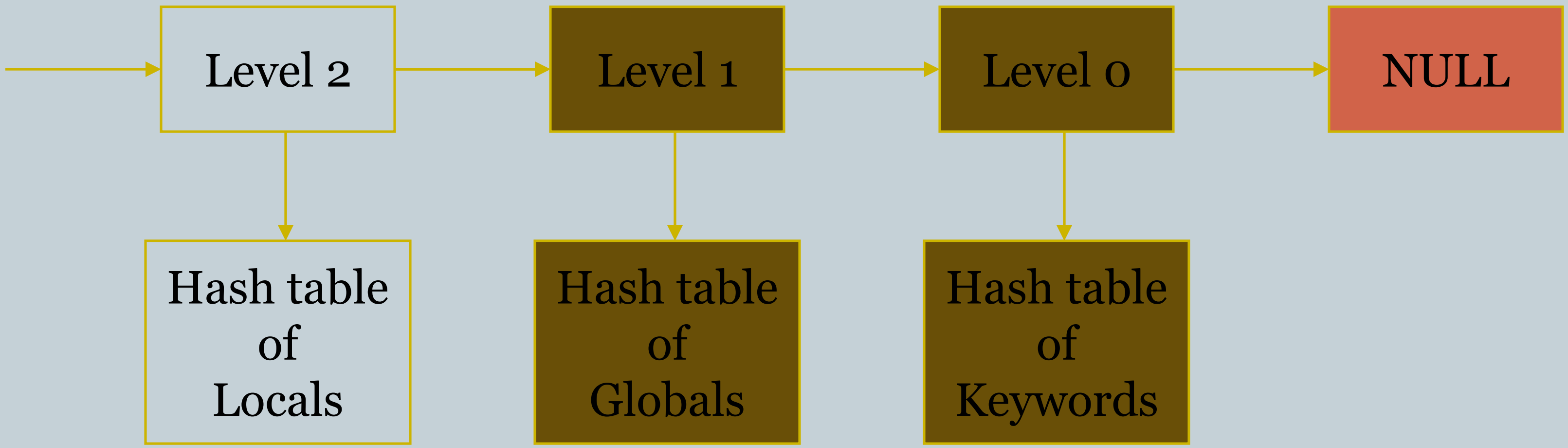
Structure of the Symbol Table

- When the parser accepted a block level (i.e. reached the closing }), the top hash table of local variables is removed from the list.



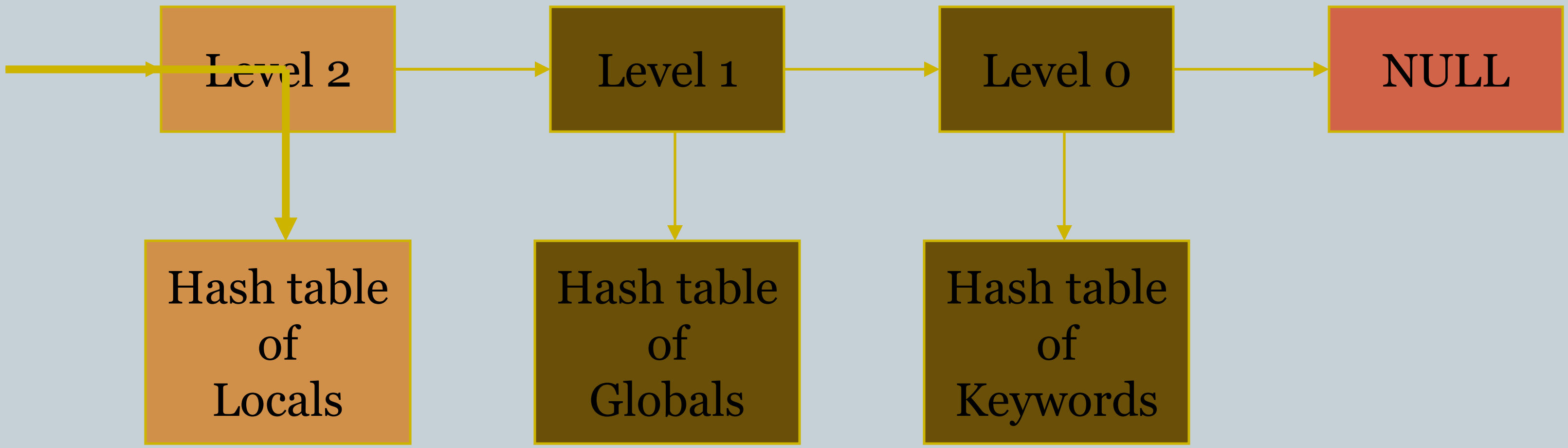
Locating a Symbol

- If the parser enters another function, a new level 2 hash table is created.



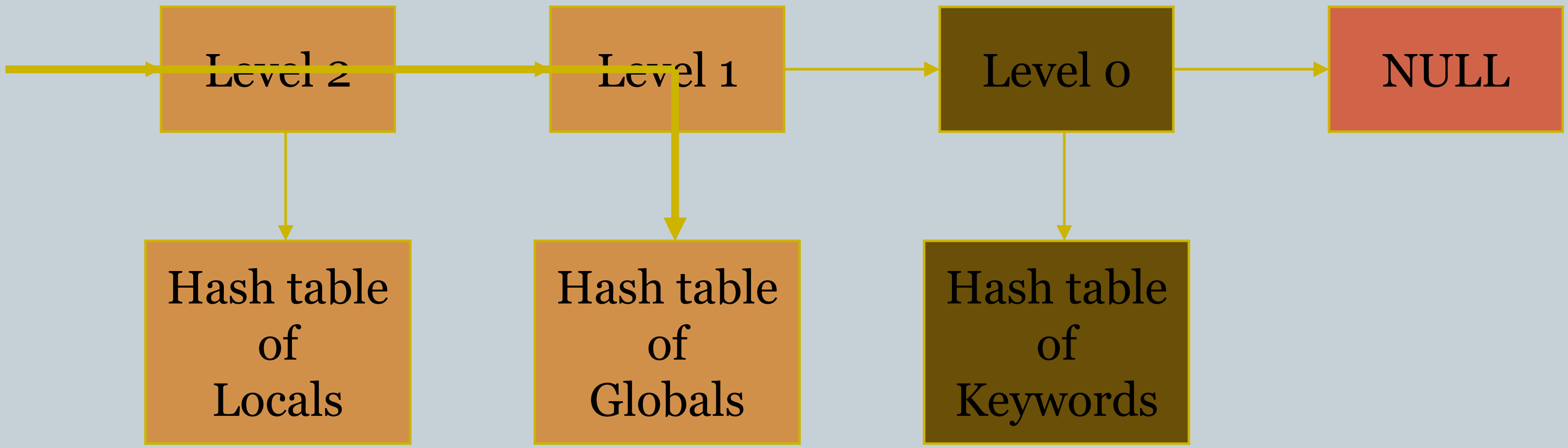
Locating a Symbol

- When we look up an identifier, we begin the search at the head of the list.



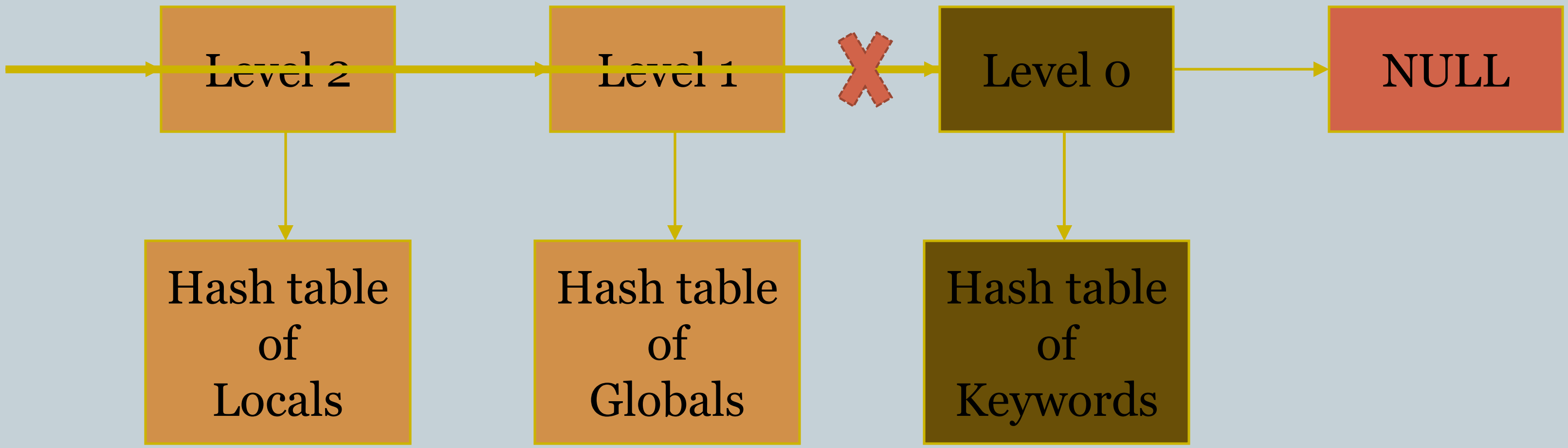
Locating a Symbol

- If it is not found there, then the search continues at the lower levels.



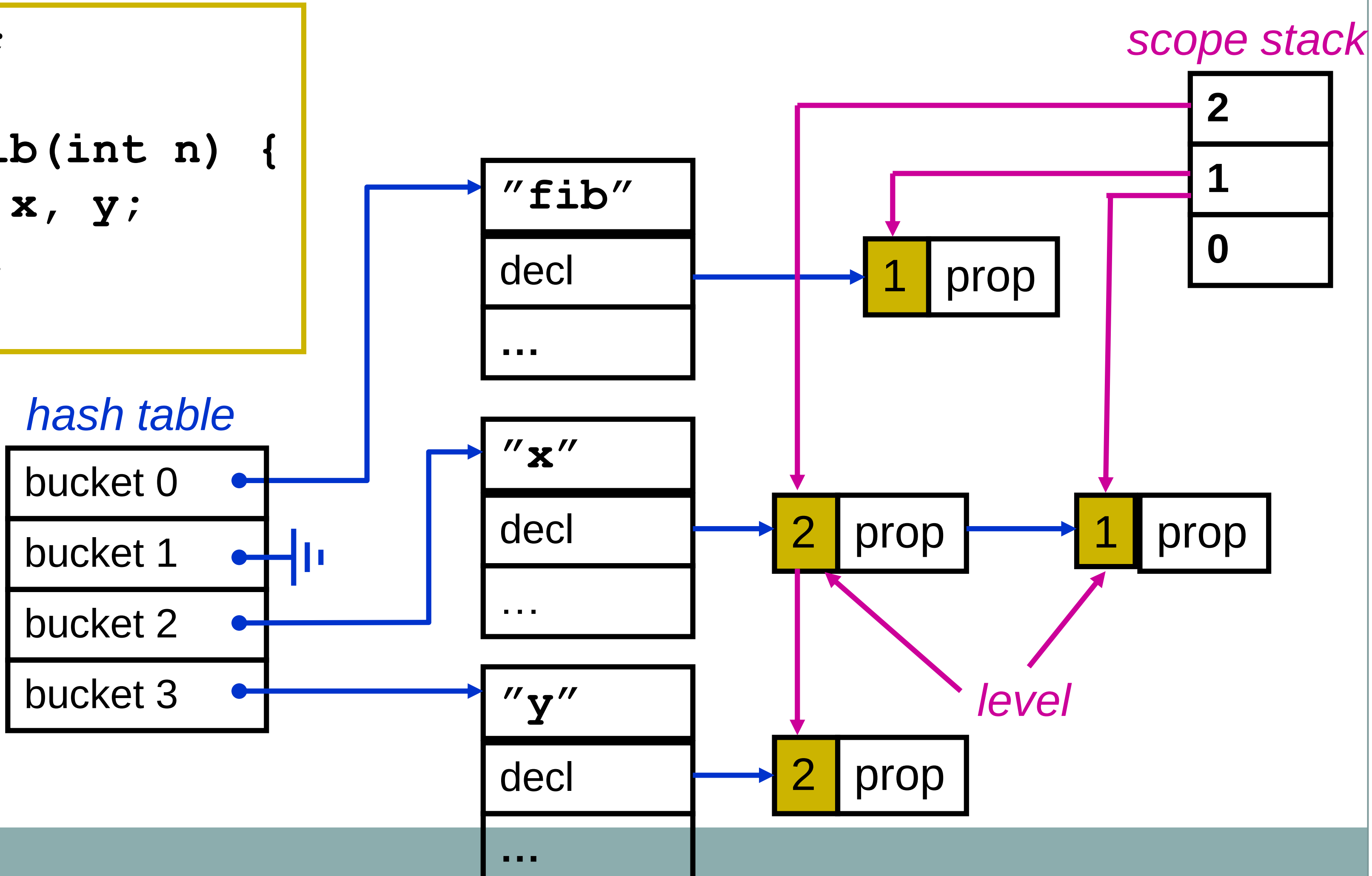
Locating a Symbol

- Keywords are not a problem!
  - The lexer only inspects the level 0 table.



Another scope hash-based symbol table

```
int x;

int fib(int n) {
  int x, y;
  ....
}
```



Declare before use

- A variable must be declared before it is used:
  - If the parser is not parsing a declaration, then it expects `idLookup()` to return non-null.
  - Anything else is an error.

Use before declaration

- In some languages, it is possible to use an identifier before it is declared.

  - A typical example is the use of a field in a C++ class

```
class A {
  A() { x = 42;}
private:
  int x;
}
```

- Solution: two-pass compiler:
  - Pass 1: gather all class names + types/names fields
  - Pass 2: do the real checking

String Tables

- Compilers often use a table of strings.
- These strings are the lexemes of the identifiers, and other strings used in the program.
- Strings get inserted, but they get never removed.
- Thus, if the same string is used for several different identifiers, the string will be stored only once in the string table.
  - Besides, it saves doing a lot of malloc's and copying
- Symbol table entries contain indexes (not pointers because of realloc!) to the string table.
  - Instead of actual strings.

Type checking/inference

- *Type checking* is the process of verifying fully typed programs.
  - Typical for compilers!
  - Weakly/Untyped languages are hard to compile => interpreters

- *Type inference* is the process of deducing type information.
  - E.g.: If e1:int and e2:int then e1+e2:int

Difference between checking and inference

- Consider the expression **`x%y`**. *Type checking* is the process that verifies that the arguments are of the type **`int`**.

- Consider the expression **`x+y`**, where **`x`** is an **`int`** and **`y`** is a **`float`**. *Type inference* is the process that infers that this expressions has type **`float`**.

Type checking

- Three typing systems:
  - Static: all checking is done as part of the compilation process (C, C++, Pascal, Haskell, …)
    - ‘Real’ compilers
  - Dynamic: all checking is done as part of program execution (Python, scheme, …)
    - Hard to compile, mostly interpreters
  - Untyped: No type checking at all (assembly)

Why type checking?

- It prevents the programmer from making 'obvious' mistakes.
- It does not make sense (in C) to add a function pointer and an integer.
- It does make sense to add a char pointer and an integer (in fact, that is array indexing).
- Both have the same assembly language implementation!

Type checking assignments

- The assignment **`x = E`** is sound if
  - type(x) and type(E) are of the same 'class' (i.e. numeric)
  - (size of) type(E) <= type(x), otherwise warning/error

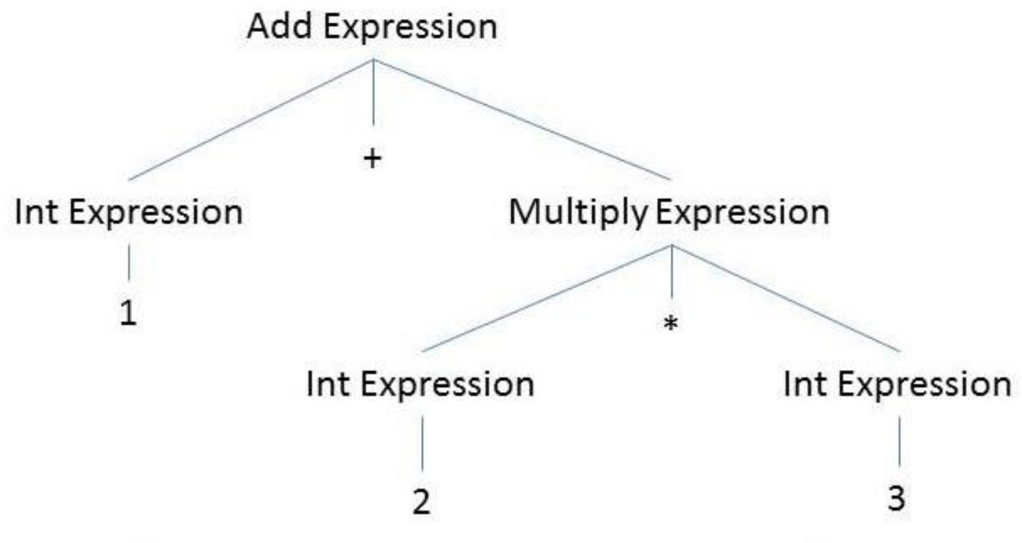
  - What is the type of **`(b ? E0 : E1)`** ?
    - smallest supertype of **`E0`** and **`E1`**

Type checking (in expressions)

- This is typically done by a recursive descent of an AST (Abstract Syntax Tree)

- For checking node n:
  - Recurse: check the children of n
    - Types are passed up the tree, i.e. from child to parent
  - On return:
    - Error in any of the children: error
    - No error in children: check node n, infer and return type (or error)

Type checking (in expressions)



Abstract Syntax Tree (AST) for 1 + 2 * 3

- Children return [ok,int]
- + node: left, right ok, both int => + node returns [ok, int]
- * node: idem

- **`3/"hello"+2`** will fail at the level of the operator **`/`**
- **`3/("hello"+2)`** will fail at the level of the operator **`+`**

Attributed grammars

- Grammar annotated with attribute rules
- Example: Grammar of signed binary numbers

Production	Semantic rule
$N \to S\ L$	printf((S.neg ? "-%d\n" : "%d\n"), L.val);
$S \to +$	S.neg = 0
$S \to -$	S.neg = 1
$L \to L_1\ B$	$L.val = 2*L_1.val+B.val$
$L \to B$	L.val = B.val
$B \to 0$	B.val = 0
$B \to 1$	B.val = 1

Attributed grammars

- Grammar annotated with attribute rules
- Each rule defines a set of dependencies
  - An attribute its value often depends on the values of other attributes.
- Some attribute values flow *upward*
  - The attributes of a node depend on those of its children
  - We call those *synthesized attributes*.
- Some attribute values flow *downward*
  - The attributes of a node depend on those of its *parent* or *siblings*.
  - We call those *inherited attributes*.

Attribute grammars

- We are only interested in two kinds of attr. grammars:
  - *S-attributed grammars*
    - All attributes are synthesized (flow up)
  - *L-attributed grammars*
    - Attributes may be synthesized or inherited, AND
    - Inherited attributes of a non-terminal only depend on the parent or the siblings to the left of that non-terminal.
      - This way it is easy to evaluate the attributes by doing a depth-first traversal of the parse tree.

Embedding Rules in Productions

- Synthesized attributes depend on the children, so they are evaluated after the children have been parsed.
- Inherited attributes that depend on the left siblings of a non-terminal must be evaluated after the left siblings have been parsed.

L.in is inherited and evaluated after parsing T but before L

T.type is synthesized and evaluated after parsing int

$D \rightarrow T$ {L.in = T.type} L
$T \rightarrow$ int {T.type = integer}
$T \rightarrow$ char {T.type = character}
$L \rightarrow$ id {L.action = addtype (id.index, L.in)}

Top-Down parsing of attributed grammars

- Recall the structure of a recursive descent parser:
  - There is a routine to recognize each lhs, which contains calls to routines that recognize the non-terminals or match the terminals on the rhs of the corresponding production.
  - We can pass the attributes as parameters (for inherited) or return values (for synthesized).
  - Example:
    $D \rightarrow T$ {L.in = T.type} L
    $T \rightarrow$ int {T.type = integer}
    - The routine for T will return the value T.type
    - The routine for L, will have a parameter L.in
    - The routine for D will call T(), get its value and pass it into L()

LL(1) version of the Expression interpreter

```
%token NUMBER, PLUS, MINUS, TIMES, DIVIDE,
       LEFT_PARENTHESIS, RIGHT_PARENTHESIS, SEMICOLON;

%start LLparse, Input;

Input: [Expression SEMICOLON]*
     ;

Expression: Term [[PLUS | MINUS] Term]*
          ;

Term: Factor[[TIMES | DIVIDE] Factor]*
    ;

Factor: NUMBER
      | MINUS Factor
      | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS
      ;
```

```
%token NUMBER, PLUS, MINUS, TIMES, DIVIDE,
        LEFT_PARENTHESIS, RIGHT_PARENTHESIS, SEMICOLON;

%start LLparse, Input;

Input {double e;}: [ Expression(&e) {printf("%f\n", e);}
                     SEMICOLON
                   ]*
                 ;

Expression(double *f) {double t; int sign;}:
      Term(f)
      [
       [PLUS {sign=1;} | MINUS {sign=-1;}]
        Term(&t) {*f += sign*t;}
      ]*
    ;

Term(double *t) {double p; int op;} :
      Factor(t)
      [
       [TIMES Factor(&p) | DIVIDE Factor(&p) {p = 1/p;}]
       {*t *= p;}
      ]*
    ;
```

```
Factor(double *f):
      NUMBER { *f = atof(yytext); }
    | NEG Factor(f) { *f = -(*f); }
    | LEFT_PARENTHESIS Expression(f) RIGHT_PARENTHESIS
    ;

{ /* the following code is copied verbatim */

  void LLmessage(int token) {
    printf("Syntax error....abort\n");
    exit(-1);
  }

  int main() {
    LLparse();
    return 0;
  }
}
```

Bottom-up parsing of attributed grammars

- Example: expression evaluator using an LR parser

Production	Semantic rule
$L \to E$ '\n'	print(E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$ digit	F.val = token.value

Bottom-up parsing of attributed grammars

- S-attributed grammars only
  - All attributes are synthesized
  - Rules are evaluated bottom-up

```
%token NUMBER PLUS MINUS TIMES DIVIDE POWER
%token LEFT_PARENTHESIS RIGHT_PARENTHESIS SEMICOLON

%left  PLUS MINUS TIMES DIVIDE NEG
%right POWER

%start Input
%%

Input: /* Empty */
     | Input Line
     ;
```

LALR(1) version of the Expression interpreter (including exponentiation)

```
Line:  Expression SEMICOLON{ printf("Result: %f\n",$1); }
       ;

Expression:
         NUMBER                              { $$=$1; }
       | Expression PLUS Expression          { $$=$1+$3; }
       | Expression MINUS Expression         { $$=$1-$3; }
       | Expression TIMES Expression         { $$=$1*$3; }
       | Expression DIVIDE Expression        { $$=$1/$3; }
       | MINUS Expression %prec NEG          { $$=-$2; }
       | Expression POWER Expression         { $$=pow($1,$3); }
       | LEFT_PARENTHESIS
          Expression RIGHT_PARENTHESIS       { $$=$2; }
       ;
```


Happy Holidays!