Compilers

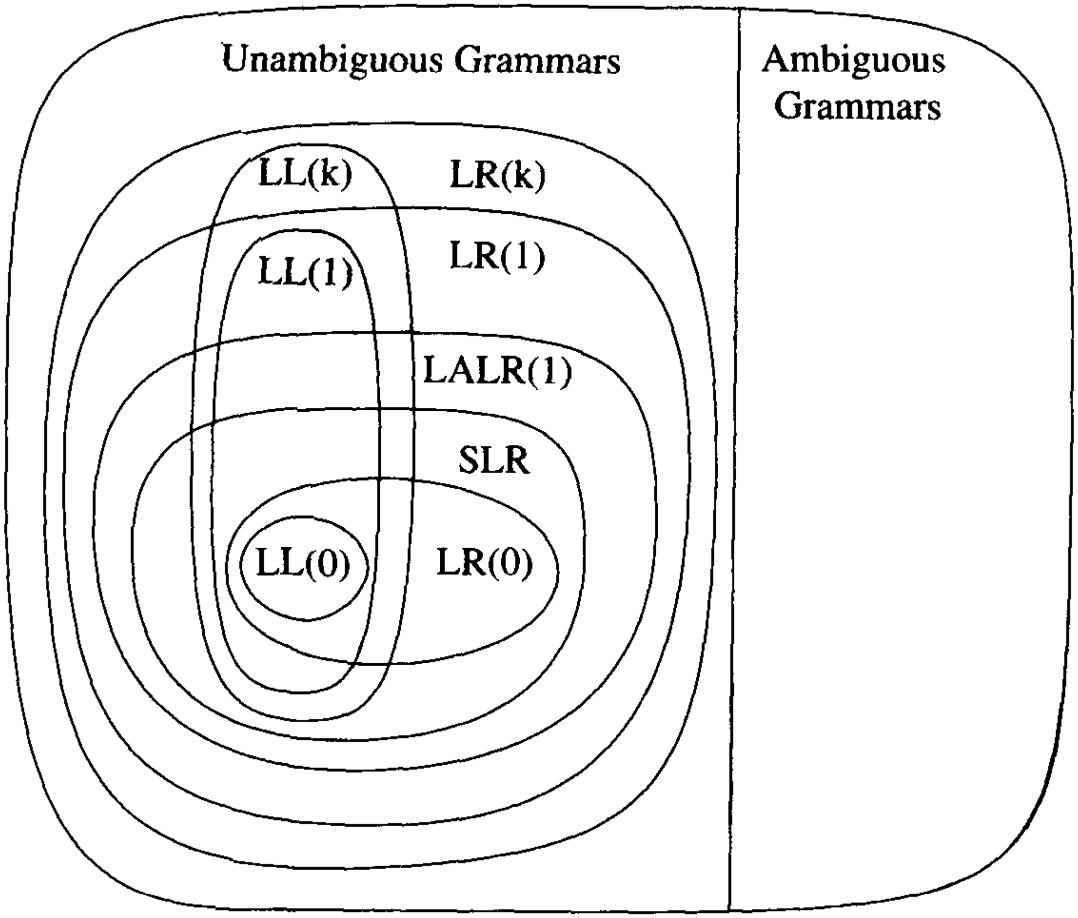
Type checking

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Summary of parsing

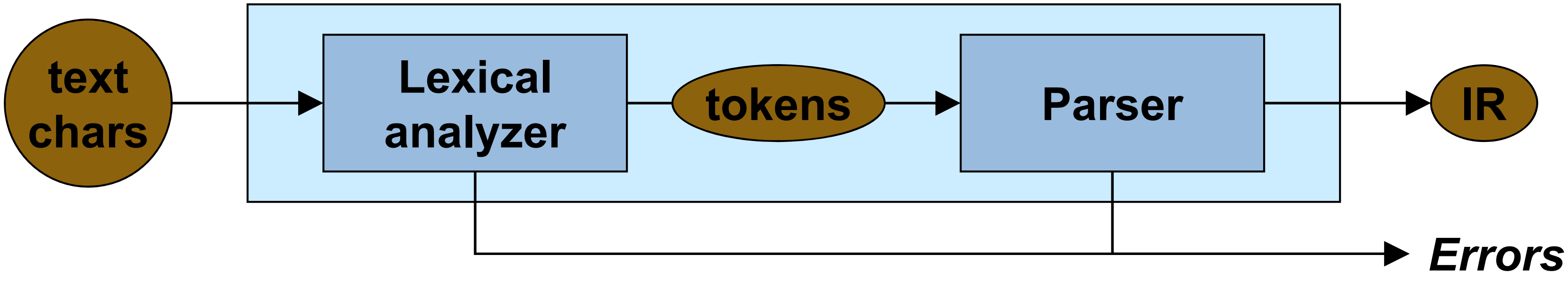
- Parsing
    - A solid foundation: context-free grammars
    - A simple parser: LL(1)
    - A more powerful parser: LR(1)
    - An efficiency hack: LALR(1)
    - LALR(1) parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"

Roadmap



- Parsing
  - Tells us if input is syntactically correct
  - Gives us derivation or parse tree
  - But we want to do more:
    - Build some data structure – the IR
    - Perform other checks and computations

Syntax-directed translation

- In practice:
    - Fold some computations into parsing
    - Computations are triggered by parsing steps

    ➡ ***Syntax-directed translation***
- Parser generators
    - Add action code to do something
    - Typically build the IR
- How much can we do during parsing?

Syntax-directed translation

- General strategy
  - Associate values with grammar symbols
  - Associate computations with productions

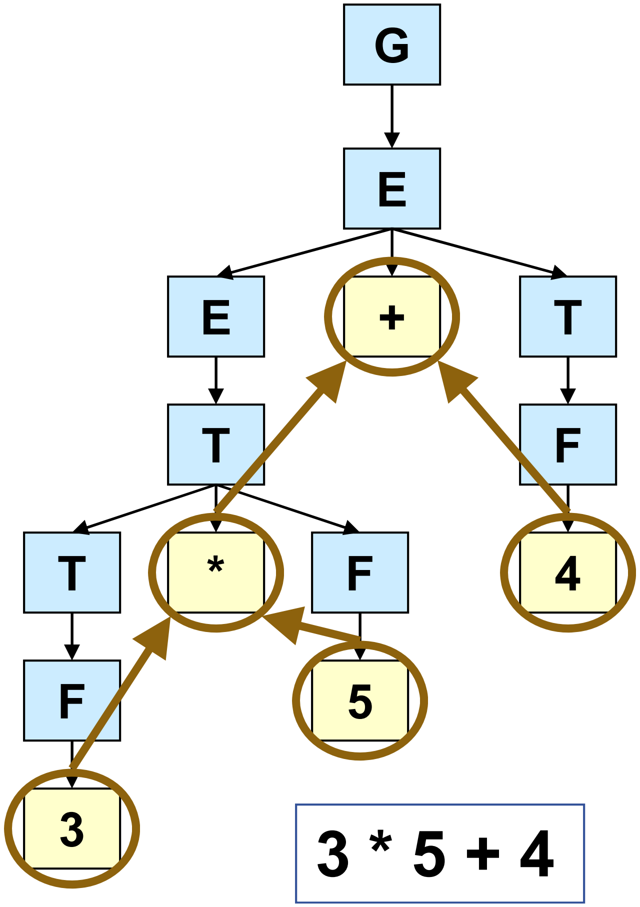
- Implementation approaches
  - **Formal**: attribute grammars
  - **Informal**: ad-hoc translation schemes

- Some things cannot be folded into parsing

Example

- Desk calculator
  - Expression grammar
  - Build parse tree
  - Evaluate the resulting tree

#	***Production rule***
1	*G* → *E*
2	*E* → *E*$_1$ + *T*
3	*E* → *T*
4	*T* → *T*$_1$ * *F*
5	*T* → *F*
6	*F* → **(** *E* **)**
7	*F* → **num**

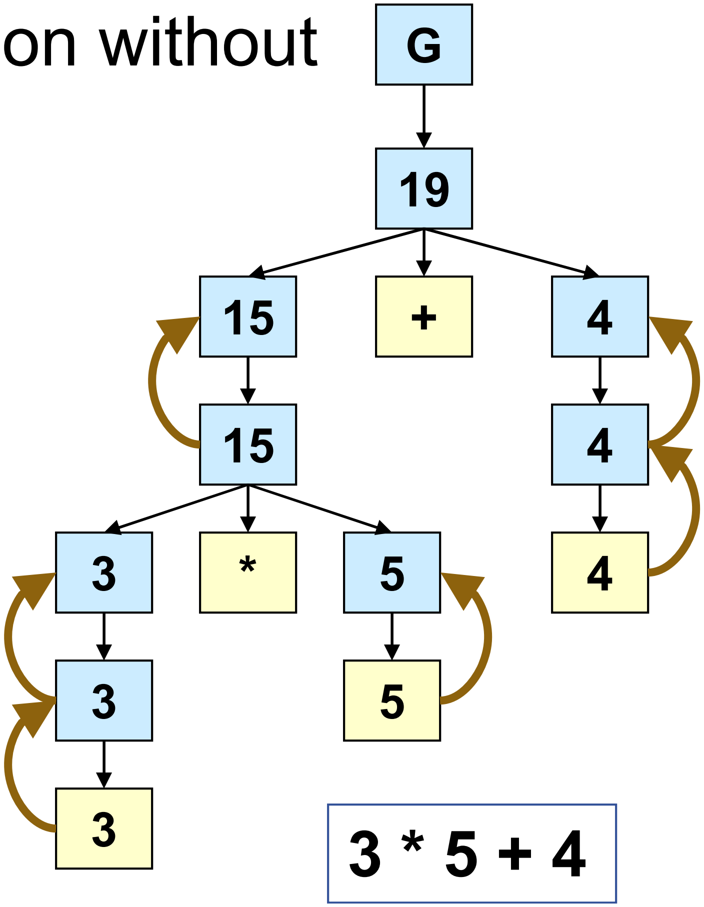


Example

- Can we evaluate the expression without building the tree first?

  "Piggyback" on parsing

#	*Production rule*
1	$G \rightarrow E$
2	$E \rightarrow E_1 + T$
3	$E \rightarrow T$
4	$T \rightarrow T_1 * F$
5	$T \rightarrow F$
6	$F \rightarrow$ <u>(</u> E <u>)</u>
7	$F \rightarrow$ <u>num</u>

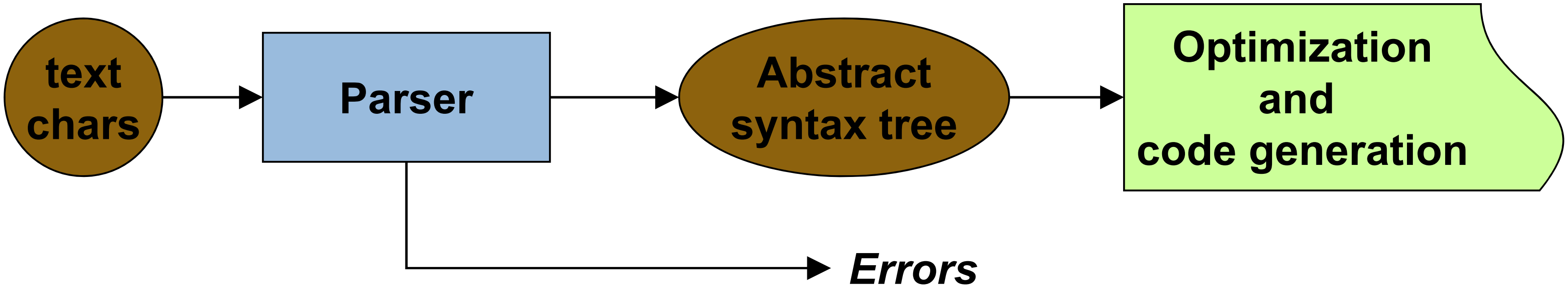


Example

- Codify:
  - Store intermediate values with non-terminals
  - Perform computations in each production

#	*Production rule*	*Computation*
1	$G \rightarrow E$	print(E.val)
2	$E \rightarrow E_1 + T$	E.val ← E_1.val + T.val
3	$E \rightarrow T$	E.val ← T.val
4	$T \rightarrow T_1 * F$	T.val ← T_1.val * F.val
5	$T \rightarrow F$	T.val ← F.val
6	$F \rightarrow \underline{(}\ E\ \underline{)}$	F.val ← E.val
7	$F \rightarrow \underline{\text{num}}$	F.val ← valueof($\underline{\text{num}}$)

Where are we…



- Parsing complete
  - Syntax is correct
  - Built an internal representation

    (usually an abstract syntax tree)
  - Now what?

Beyond syntax

- What’s wrong with this code?

  (Note: it parses perfectly)

```
foo(int a, char * s){ … }

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

Errors

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of args to call
- Incompatible types for operation
- Break statement outside switch/loop
- Goto with no label

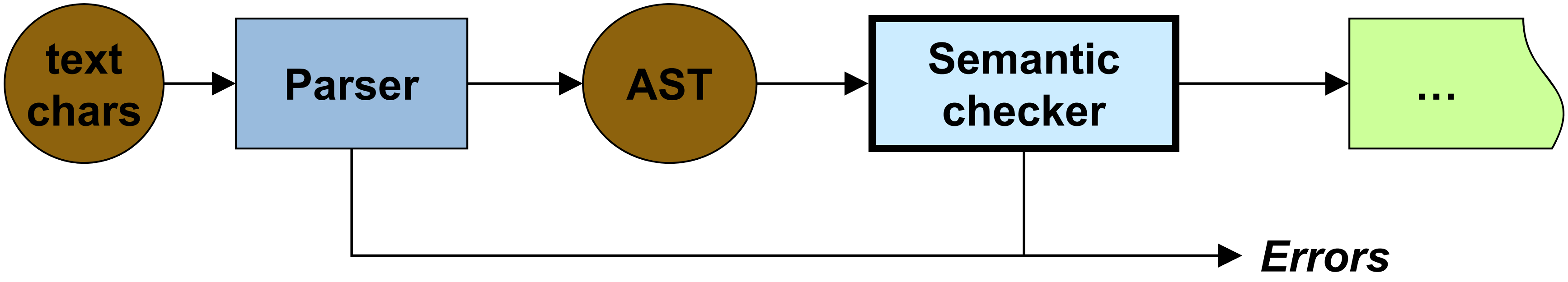
Program checking

Why do we care?

- Obvious:
  - Report mistakes to programmer
  - Avoid bugs: *f[6] will cause a run-time failure*
  - Help programmer verify intent

- How do these checks help compiler?
  - Allocate right amount of space for variables
  - Select right machine operations
  - Proper implementation of control structures

Program checking



- *Semantic* checking
  - Beyond syntax: hard to express directly in grammar
  - Requires extra computation, extra data structures
  - **Goals**:
    - Better error checking – “deeper”
    - Give back-end everything it needs to generate code

Program checking

When are checks performed?

- ***Static*** checking
  - At compile-time
  - Detect and report errors by analyzing the program
- ***Dynamic*** checking
  - At run-time
  - Detect and handle errors as they occur

- What are the pros and cons?

  Efficiency? Completeness? Developer vs user experience? Language flexibility?

- What is the role of the compiler?

Kinds of static checks

- Uniqueness checks
  - Certain names must be unique
  - Many languages require variable declarations
- Flow-of-control checks
  - Match control-flow operators with structures
  - Example: break applies to innermost loop/switch
- Type checks
  - Check compatibility of operators and operands
  - Example: does 3.5 + “foobar” make sense?
- What kind of check is “array bounds”?

Uniqueness checks

- What does a ***name*** in a program denote?
  - Variable
  - Label
  - Function name
- Information maintained in ***bindings***
  - A binding from the name to the entity
  - Bindings have ***scope*** –
    the region of the program in which they are valid
- Uniqueness checks:
  - Analyze the bindings
  - Make sure they obey the rules
- Closely tied to ***procedures***

Procedures

- What is a ***procedure***/***function***/***method***?
- Does it exist at the machine code level?
  - Not really – it's an abstraction created by the compiler
  - Components
    - Name space abstraction
    - Control abstraction
    - Interface
- Today: name space abstraction
  - Defines scoping and binding rules
- Later: look at how abstraction is implemented

Procedures as name spaces

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names hide identical non-local names (***shadowing***)
- Local names cannot be seen outside the procedure
- We call this set of rules & conventions ***lexical scoping***
- Scopes may be ***nested***

Examples

- C has global, static, local, and block scopes
  Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes
  Procedure scope (typically) contains formal parameters

Procedures as name spaces

- Why introduce lexical scoping?
  - Flexibility for programmer
  - Simplifies rules for naming & resolves conflicts
- Implementation:

  The compiler responsibilities:
  - At point *p*, which “x” is the programmer talking about?
  - At run-time, where is the value of *x* found in memory?
- Solution:

  Lexically scoped symbol tables

Examples

- In C++ and Java

```
{
  for (int i=0; i < 100; i++) {
   ...
  }

  for (Iterator i=list.iterator(); i.hasNext();) {
   ...
  }
}
```

- This is actually useful!

Dynamic vs static

- Static scoping
  - Most compiled languages – C, C++, Java, Fortran
  - Scopes only exist at ***compile-time***
  - We'll see the corresponding ***run-time*** structures that are used to establish addressability later.
- Dynamic scoping
  - Interpreted languages – Perl, Common Lisp

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

Lexically-scoped Symbol Tables

- Compiler job
  - Keep track of names (identifiers)
  - At a use of a name, find its information (like what?)
- The problem
  - Compiler needs a distinct entry for each declaration
  - Nested lexical scopes admit duplicate declarations
- The symbol table interface
  - **enter()** – enter a new scope level
  - **insert(*name*)** – creates entry for *name* in current scope
  - **lookup(*name*)** – lookup a name, return an entry
  - **exit()** – leave scope, remove all names declared there

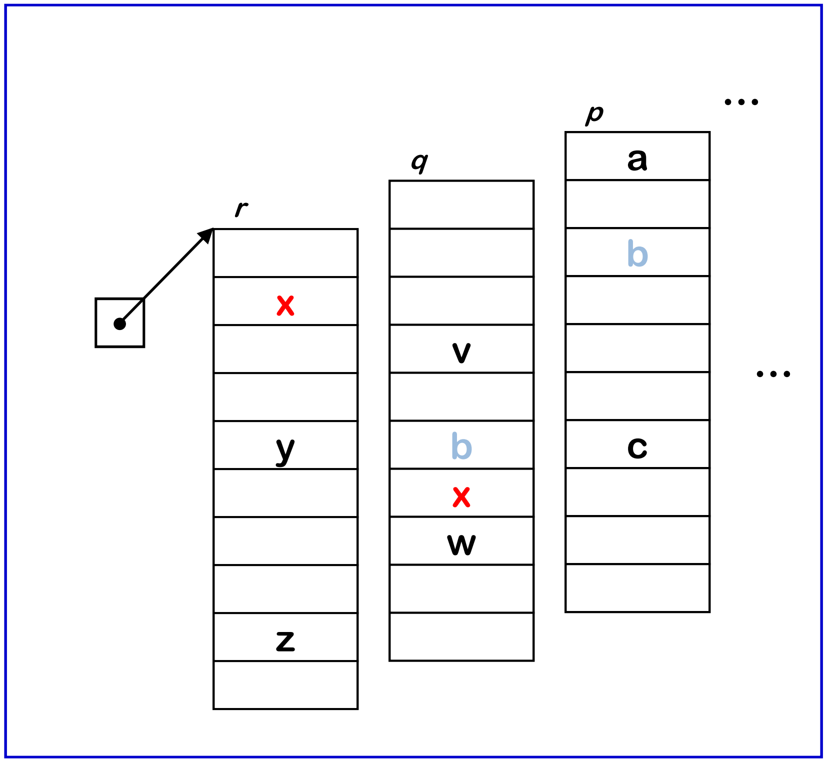
Example

```
class p {
   int a, b, c
   method q {
      int v, b, x, w
      for (r = 0; ...) {
         int x, y, z
         ….
      }
      while (s) {
         int x, a, v
         …
      }
      … r … s
   }
  … q …
}
```

```
L0:{
     int a, b, c
L1:  {
       int v, b, x, w
L2a:   {
         int x, y, z
          ...
       }
L2b:   {
         int x, a, v
         ...
       }
     }
   }
```

Chained implementation

- Create a new table for each scope, chain them together for lookup

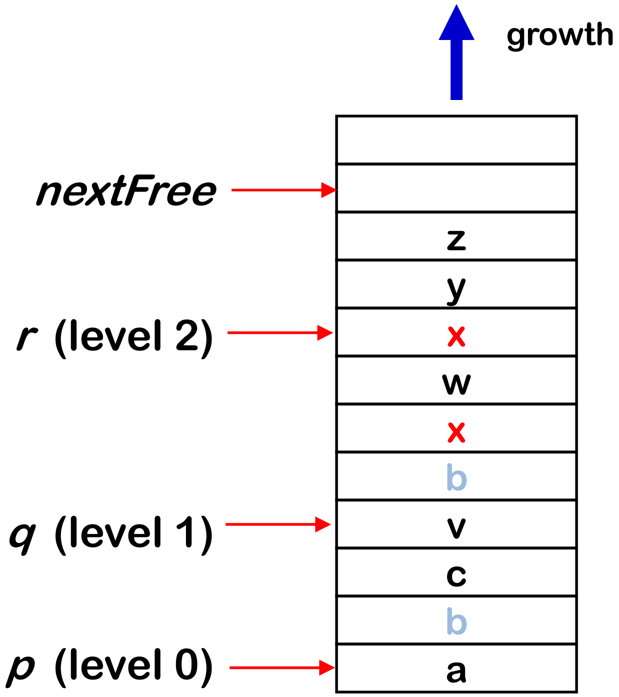


"***Sheaf of tables***" implementation

- ***enter*()** creates a new table
- ***insert*()** adds at current level
- ***lookup*()** walks chain of tables & returns first occurrence of name
- ***exit*()** throws away table for level *p*, if it is top table in the chain

How would you implement the individual tables?

Stack implementation



Implementation

- **enter()** puts a marker in stack
- **insert ()** inserts at nextFree
- **lookup ()** searches from nextFree–1 forward
- **exit ()** sets nextFree back to the previous marker.

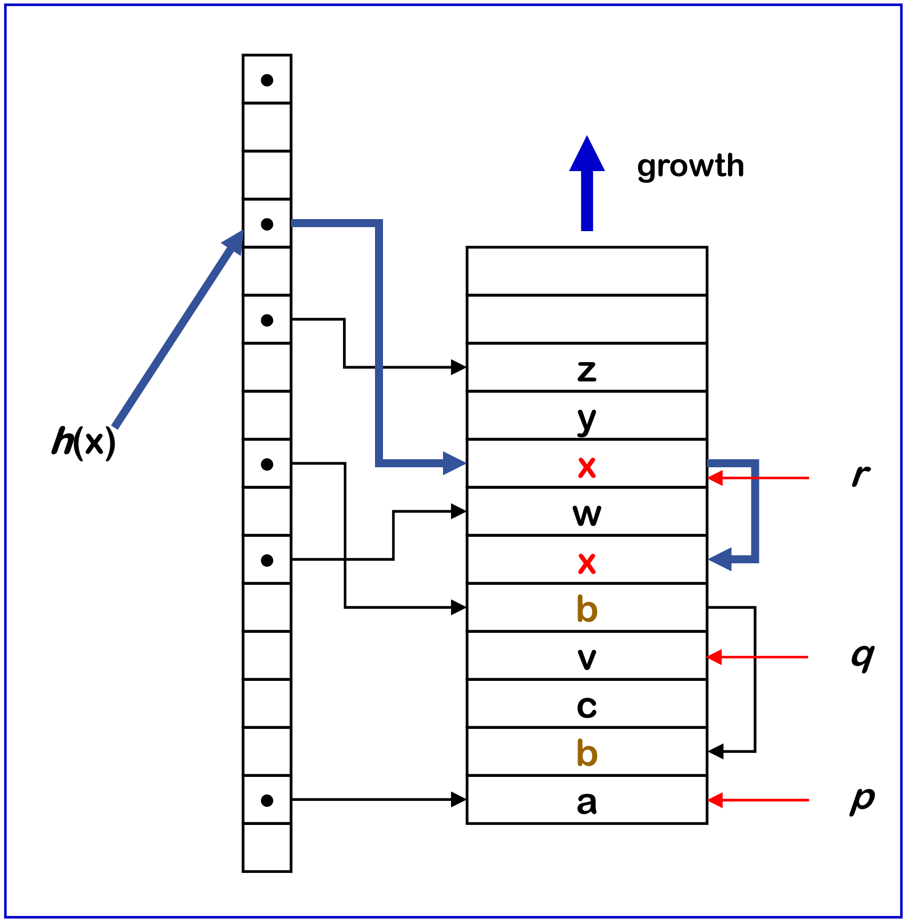
Advantage

- Uses less space

Disadvantage

- Lookups can be expensive

Threaded stack implementation



Implementation

- ***insert ()*** puts new entry at the head of the list for the name
- ***lookup ()*** goes direct to location
- ***exit ()*** processes each element in level being deleted to remove from head of list

Advantage

- lookup is fast

Disadvantage

- exit takes time proportional to number of declared variables in level

Symbol tables in C

- ***Identifiers***
  - Mapping from names to declarations
  - Fully nested – each '{' opens new scope
- ***Labels***
  - Mapping from names to labels (for goto)
  - Flat table – one set of labels for each procedure
- ***Tags***
  - Mapping from names to struct definitions
  - Fully nested
- ***Externals***
  - Record of extern declarations
  - Flat table – redundant extern declarations must be identical

In general, rules can be very subtle

Examples

- Example of typedef use:

```
typedef int T;
struct S { T T; }; /* redefinition of T as member name */
```

- Example of proper declaration binding:

```
int;       /* syntax error: vacuous declaration      */
struct S;  /* no error: tag is defined, not elaborated */
```

- Example of declaration name spaces
  - Declare "a" in the name space before parsing initializer

    ```
    int a = sizeof(a);
    ```

  - Declare "b" with a type before parsing "c"

    ```
    int b, c[sizeof(b)];
    ```

Uniqueness checks

- Which ones involve uniqueness?
- What do we need to do to detect them?

```
foo(int a, char * s){ … }

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

Next: type checking

- Big topic
  - Type systems
  - Type inference
  - Non-standard type systems for program analysis
  - Theory of type systems
- Focus
  - Role of types in compilation
  - Imperative and object-oriented languages
- What is a type?

  Def:

  A ***type*** is a collection of values and
  a set of operations on those values

Purpose of types

- Identify and prevent errors
  - Avoid meaningless or harmful computations
  - Meaningless: (x < 6) + 1 – “bathtub”
  - Harmful?
- Program organization and documentation
  - Separate types for separate concepts
  - Type indicates programmer intent
- Support implementation
  - Allocate right amount of space for variables
  - Select right machine operations
  - Optimization: e.g., use fewer bits when possible
- **Key idea**: types can be ***checked***

Type errors

- **Problem**:
  - Underlying memory has no concept of type
  - Everything is just a string of bits:

```
0100 0000 0101 1000 0000 0000 0000 0000
```

    - The floating point number 3.375
    - The 32-bit integer 1,079,508,992
    - Two 16-bit integers 16472 and 0
    - Four ASCII characters: @ X NUL NUL
- Without type checking:
  - Machine will let you store 3.375 and later load 1,079,508,992
  - Violates the intended semantics of the program

Type system

- **Idea**:
  - Provide clear interpretation for bits in memory
  - Imposes constraints on use of variables, data
  - Expressed as a set of rules
  - Automatically check the rules
  - Report errors to programmer
- Key questions:
  - What types are built into the language?
  - Can the programmer build new types?
  - What are the typing rules?
  - When does type checking occur?
  - How strictly are the rules enforced?

When are checks performed?

- What do you think the choices are?
  - Static and dynamic
  - ***Statically typed*** languages
    - Types of all variables are determined ahead of time
    - Examples?
  - ***Dynamically typed*** languages
    - Type of a variable can vary at run-time
    - Examples?

- Our focus?
  - Static typing – corresponds to compilation

Expressiveness

- Consider this Scheme function:

```
(define myfunc (lambda (x)
           (if (list? x) (myfunc(first x))
                         (+ x 1))
```

- What is the type of x?
  - Sometimes a list, sometimes an atom
  - Downside?
- What would happen in static typing?
  - Cannot assign a type to x at compile time
  - Cannot write this function
  - Static typing is ***conservative***

Types and compilers

- What is the role of the compiler?

  Example: we want to generate code for

  ```
  a = b + c * d;
  ```

  ```
  arr[i] = *p + 2;
  ```

  - What does the compiler need to know?
- Duties:
  - Enforce type rules of the language
  - Choose operations to be performed

    Can we do this in one machine instruction?
  - Provide concrete representation – bits

    Next time: where is the storage?
  - What if can't perform the check at compile-time?

Type systems

From language specifications:

> “The result of a unary & operator is a pointer to the object referred to by the operand. If the type of the operand is “T”, the type of the result is “pointer to T”.

> “If both operands of the arithmetic operators addition, subtraction and multiplication are integers, then the result is an integer”

Properties of types

These excerpts imply:

- Types have structure

  "Pointer to T" and "Array of Pointer to T"

- Expressions have types

  Types are derived from operands by rules

- **Goal**: determine types for all parts of a program

Type expressions

(Not to be confused with types of expressions)

- Build a description of a type from:
  - Basic types – also called “primitive types”

    Vary between languages: int, char, float, double
  - Type constructors

    Functions over types that build more complex types
  - Type variables

    Unspecified parts of a type – polymorphism, generics
  - Type names

    An “alias” for a type expression – **`typedef`** *in C*

Type constructors

- **Arrays**
  - If T is a type, then *array*(T) is a type denoting an array with elements of type T
  - May have a size component: *array*(I,T)
- **Products** or **records**
  - If T_1 and T_2 are types, then $T_1 \times T_2$ is a type denoting pairs of two types
  - May have labels for records/structs

    (“name”, char *) × (“age”, int)

Type constructors

- Pointers
  - If T is a type, the *pointer*(T) denotes a pointer to T

- Functions or function ***signatures***
  - If D and R are types then D → R is a type denoting a function from domain type D to range type R
  - For multiple inputs, domain is a product
  - Notice: primitive operations have signatures

    Mod % operator: int × int → int

Example

- Static type checker for C
  - Defined over the structure of the program
- Rules:

Expression	*Type rule*
$E_1 + E_2$	if type(E_2) is int and type(E_1) is int result type is int else …other cases…

- **Question**:

  How do we get declared types of identifiers, functions?

More examples

- More interesting cases
- Rules:

Expression	*Type rule*
E_1 [E_2]	if type(E_2) is int and type(E_1) is *array*(T) result type is T else error
* E	if type(E) is *pointer*(T) result type is T else error

Example

- What about function calls?
  - Consider single argument case

Expression	***Type rule***
E_1 (E_2)	**if type(E_1) is D → R and type(E_2) is D result type is R else error**

- How do we perform these checks?
  - What is the core type-checking operation?
  - How do I determine if “type(E) is D”?

    “If two type expressions are equivalent then…”

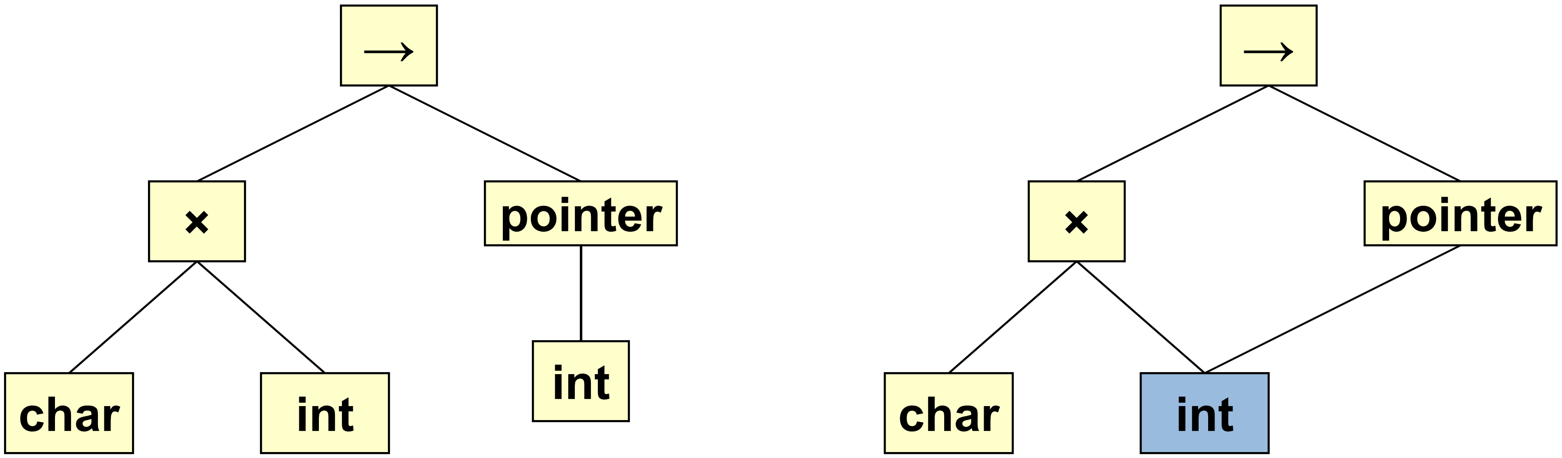
Type equivalence

- Implementation: ***structural equivalence***
  - Same basic types
  - Same set of constructors applied
- Recursive test:

```
function equiv(s, t)
    if s and t are the same basic type
        return true
    if $s$ = pointer($s_1$) and $t$ = pointer($t_1$)
        return equiv($s_1$,$t_1$)
    if $s = s_1 \times s_2$ and $t = t_1 \times t_2$
        return equiv($s_1$,$t_1$) && equiv($s_2$,$t_2$)
…etc…
```

Representation

- Represent types as graphs
  - Node for each type
  - Often a DAG: share the structure when possible



Function: (char × int) → int *

Structural equivalence

- Efficient implementation
  - Recursively descend DAG until common node
- Many subtle variations in practice
  - Special rules for parameter passing
    - C: array T[] is compatible with T*
    - Pascal, Fortran: leaving off size of array
    - Is “size” part of the type?
  - Type qualifiers: **`const`**, **`static`**, etc.

Expr	*Type rule*
E_1 = E_2 ;	**if type(E_1) == type(E_2) result type is E_1 else error**

Notions of equivalence

- Different way of handling type names
- ***Structural equivalence***
  - Ignores type names
  - **`typedef int * numptr`** means **`numptr ≡ int *`**
  - Not always desirable
  - Example?
- ***Name equivalence***
  - Types are equivalent if they have the same name
  - Solves an important problem: recursive types

Recursive types

- Why is this a problem?

```
struct cell {
    int info;
    struct cell * next;
}
```

- Cycle in the type graph!
- C uses structural equivalence for everything *except* structs (and unions)
  - The name "**`struct cell`**" is used instead of checking the actual fields in the struct
  - Can we have two compatible struct definitions?

Java types

- Type equivalence for Java

```
class Foo {       class Bar {
  int x;            int w;
  float y;          float z;
}                 }
```

- Can we pass Bar objects to a method taking a type Foo?
  - No
  - Java uses name equivalence for classes
  - What can we do in C that we can’t do in Java?

Type checking

- Consider this case:

  What is the type of `x+i`   if `x` is `float` and **i** is `int`

- Is this an error?

- Compiler fixes the problem
  - Convert into compatible types
  - Automatic conversions are called ***coercions***
  - Rules can be complex
    - in C, large set of rules for called ***integral promotions***
    - Goal is to preserve information

Type coercions

- Rules
  - Find a common type
  - Add explicit conversion into the AST

Expression	*Type rule*
E_1 + E_2	**if type(E_1) is int and type(E_2) is int** **result type is int** **if type(E_1) is int and type(E_2) is float** **result type is float** **if type(E_1) is float and type(E_2) is int** **result type is float** **…etc…**

Implementing type checkers

Expression	***Type rule***
$E \rightarrow E_1 [E_2]$	if type(E_2) is int and type(E_1) is *array*(T) type(E) = T else error
$E \rightarrow * E$	if type(E) is *pointer*(T) type(E) is T else error

- Does this form look familiar?
  - Type checking fits into syntax-directed translation

Interesting cases

- What about printf?
    - printf(const char * format, …)
    - Implemented with varargs
    - Format specifies which arguments should follow
    - Who checks?
- Array bounds
    - Array sizes rarely provided in declaration
    - Cannot check statically (in general)

      There are fancy-dancy systems that try to do this
    - Java: check at run-time

Overloading

- “+” operator
  - Same syntax, same “semantics”, multiple implementations
  - C: `float` versus `int`
  - C++: arbitrary user implementation
    - Note: cannot change parser – what does that mean?
- How to decide which one?
  - Use types of the operands
  - Find operator with the right type signature
- Complex interaction with coercions
  - Need a rule to choose between conversion and overloading

Object oriented types

```
class  Foo  {     ….      }
class  Bar  extends  Foo  {    …    }
```

- What is relationship between **`Foo`** and **`Bar`**?
  - **`Bar`** is a ***subtype*** of **`Foo`**
  - Any code that accepts a **`Foo`** object can also accept a **`Bar`** object
  - We'll talk about how to implement this later
- Modify type compatibility rules
  - To check an assignment, check subtype relationship <=
  - Also for formal parameters

Expr	***Type rule***
$E_1 = E_2$;	**if type(E_2) <= type(E_1) result type is E_1 else error**

Java arrays

```
class  Foo  {     ….     }
class  Bar  extends  Foo  {    …    }
Foo[] foo_array;
Bar[] bar_array;
```

- **Question**: is `bar[]` a subtype of `foo[]`?
  - Answer: ***yes***
  - Consequences?

```
void storeIt(Foo f, Object [] arr)
{
    arr[0] = f;
}
```

  - How do we perform this check?