Abstract Syntax - 2

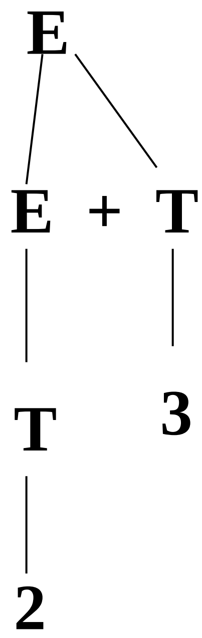
- ***Abstract syntax (parse) trees*** **- an intermediate program representation**
- ***Symbol tables*** **- keeping information about identifiers**
- **Hashing**

Abstract Syntax Trees

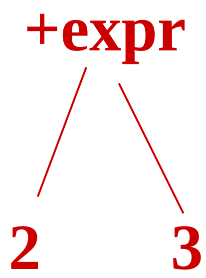
- **Allow separation of parsing from semantic checking (e.g., type checking)**
- **Shows the abstract syntax of the language (only keeps nonterminals which have some meaning attached)**
  - **Different from concrete syntax with punctuation, trivial productions (e.g., E → T → F → *id*)**
  - **Compilers can manipulate the abstract syntax once concrete syntax has been checked**

Example

2 + 3



Parse tree



Abstract syntax tree

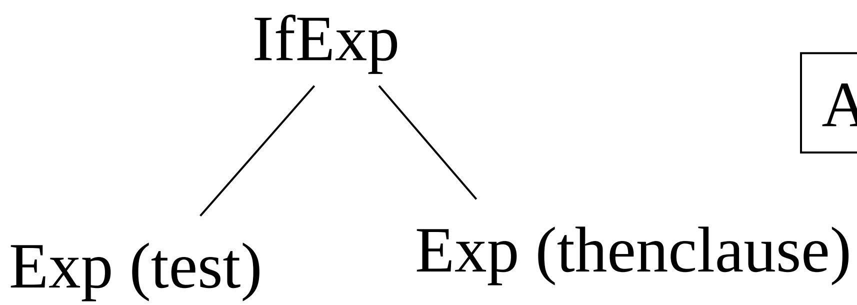
More examples in Appel, Chp 4

Abstract Syntax Trees

- **Should maintain some pointer back into the input (line number in file + character position in line) corresponding to tree**
- **Scanner passes beginning and ending positions of each token**
- **Internal tree nodes can figure out their own *position* as a function of *positions* of leaf nodes beneath them**

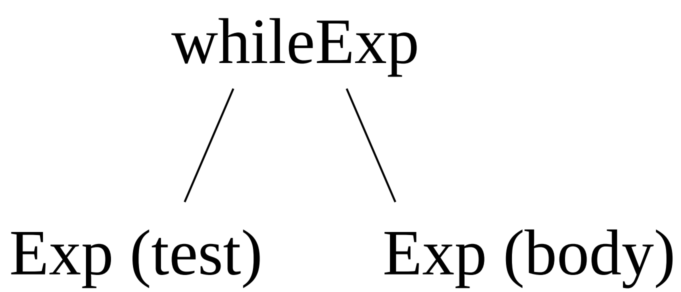
Tiger Abstract Syntax, E.G.s

IfExp (int pos, Exp test, Exp thenclause)



Appel, Chp 4.2

whileExp (int pos, Exp test, Exp body)



Tiger Abstract Syntax

- **Boolean expressions translated in optimized form as ifs**
- **SeqExp(null) is empty statement**
- **See abstract syntax defined for all Tiger constructs, Appel p 103**

Design of Abstract Syntax

- ***Syntax separate from interpretations* (SSI) style of programming**
  - **E.g., Appel, program 1.5, used *instanceof* and public class variables to access abstract syntax**
- ***Object-oriented* (OO) style**
  - **No public instance variables (need observer methods)**
- **Choice of style affects modularity**
- ***Kinds* of objects (e.g., assignments, prints, if statements)**

Design of Abstract Syntax

- **Types of *interpretations* (or analyses) (e.g., type checking, code generation)**
- **SSI finds it easy to add another interpretation but hard to add another kind**
  - **Impacts all previous interpretations which now need to handle new kind**
- **OO finds it easy to add another kind, but difficult to add another interpretation**
  - **Impacts all previous classes that now need to add a method for the new interpretation**

Environment

- ***Environment* - a set of bindings denoted as pairs: <id $\rightarrow$ value>**
- **Environment embodies scoping of identifiers**
- **In *lexical scoping*, you build environments from previous ones by adding bindings that supercede any previous ones for the same identifier or deleting bindings no longer current**

Nested Environments

$\sigma 0$

function f (int x, int y) (

$\sigma 1$

let var z := 1

$\sigma 2$

in print_int (x+z)

end;

$\sigma 1$

)

$\sigma 0$

$\sigma 0$: **defining environment**

$\sigma 1 = \sigma 0 + \{x \rightarrow \text{int}; y \rightarrow \text{int}\}$

$\sigma 2 = \sigma 1 + \{z \rightarrow \text{int}\}$

Environments

- **Can handle environments either in functional style (always make copies) or in imperative style (do destructive updates)**
- **Choice is independent of whether the language being parsed is functional or imperative**

Symbol Table

- ***Maps from identifier to meanings*** **(e.g., types, function signatures, array bounds)**
- **Two functions need efficient implementation:**
  - **Add new entries**
  - **Search for existing entries**
- **Entries are usually of uniform size**
  - **Some data kept in auxiliary store and pointed to from table**

Open Addressing Hash Tables

- **All elements stored in the hash table itself**
- **Complexity**
  - **O(1) expected time for search or insertion**
  - **O(m) space for size m table**
- **In case of collision, have to offer a way to resolve collisions**
  - ***Linear resolution* f(key)=k, try k-1, k-2, etc until find empty entry**
    - **Simple, but forms long chains**

Open Addressing Hash Tables

- *Add the hash rehash*, f(key) =k, try 2*f(key), 3*f(key), etc. until find empty entry
  - Reduces clustering found in linear rehash
- *Quadratic rehash*, f(key)=k, (f(key)+1) mod m, (f(key)+2^2) mod m, (f(key)+3^2) mod m, etc

- Sometimes use 2 level table, top level for indices (sparse), bottom level for entries of varying size pointed to by top level entries.

Bucket Hash Tables

- **Fixed-size array of *m* buckets (linked lists)**
  - **Combines sparse index with list of items**
- **Use stack discipline (LIFO) when adding to or searching a bucket**
- **Lookup in constant time to correct bucket and then linear in number of bucket entries**
  - **Worst case assumed very unlikely**
  - **Average search time 1/2*(n/m)**
  - **Average insertion time n/m**
  - **O(n+m) space needed**

Bucket Hash Tables

- **Lookup and insertion:**
  - **Hash to index**
  - **If table[index] empty**
    - **Lookup fails**
    - **Insertion adds into bucket at index**
  - **if table[index] full**
    - **Match in bucket implies lookup succeeds; otherwise, fails**
    - **Insert at head of bucket list**

Hash Functions

- **Try to map *n* items uniformly into *n* of *m* distinct entries in table; use a *mod size* calculation**
- **Usually pick *m* to be a large prime number**
- ***Collision* - when f(key1) = f(key2) or two keys hash to the same entry**
- **Desirable that f(key) is cheap to evaluate and *randomizing* (i.e., similar names map to different indices)**

Hash Functions

- **How hard is it to design a good hash function?**
  - **Say have 31 keywords and want a 41 element hash table; Have 41^{31} 10^{50} choices of hash mappings; 41*40*39*…*11 of them will map each key to a distinct entry (1 out of 10^{7})**
  - **Good hash functions can be hard to find**

Collisions

- **If there are more than 24 people at a birthday party then probability that at least 2 will have the same birthday is more than 50%**
  - **You didn't try to invite people with the same birthday,**
- **Conclusion: collisions will happen in hash tables**

Hash Functions

- **How to build a good hash function?**
  - **Select only certain characters from the key**
    - **Not evenly distributed**
  - **Partition the key into parts and then combine them**
    - **E.g., take a function of each byte**
  - **Convert key to integer by modulo arithmetic using a very large prime number (spreads the keys fairly uniformly about) f(k)=k mod M**

Symbol Table

- **Operations**
  - **Insert: make new entry**
  - **Delete: remove most recently created entry**
  - **Lookup: find most recently created entry of name**
- **If use buckets with LIFO strategy, then dealing with nested lexical scopes is easy**
- **Conceptually think of 1 symbol table per scope, but actually use same table with LIFO to accomplish (see *Symbol* package in project)**

Symbol package, Appel Chp 5

- **Map strings to symbol objects so can compare more easily**
- ***Symbol* package contains classes *Table* and *Symbol***
  - ***Table* creates bucket hash table with scope entry and exit methods to mark and process the buckets**
  - ***Symbol* uses *String* instance method *intern()* to return a unique value for any string; this value is encapsulated in the *Symbol* object created and used for comparison**