Abstract Syntax

Can write entire compiler in ML-YACC specification.
- Semantic actions would perform type checking and translation to assembly.
- Disadvantages:
  1. File becomes too large, difficult to manage.
  2. Program must be processed in order in which it is parsed. Impossible to do global/inter-procedural optimization.
Alternative: Separate parsing from remaining compiler phases.

Parse Trees

- We have been looking at concrete parse trees.
  - Each internal node labeled with non-terminal.
  - Children labeled with symbols in RHS of production.
- Concrete parse trees inconvenient to use! Tree is cluttered with tokens containing no additional information.
  - Punctuation needed to specify structure when writing code, but
  - Tree structure itself cleanly describes program structure.
Parse Tree Example

\[
\begin{align*}
P & \rightarrow ( S ) \\
S & \rightarrow S ; S \\
S & \rightarrow ID := E
\end{align*}
\]

\[
E \rightarrow ID \\
E \rightarrow NUM \\
E \rightarrow E + E \\
E \rightarrow E \ast E \\
E \rightarrow E / E
\]

\[
( \ a \ := \ 4 \ ; \ b \ := \ 5 \ )
\]

\[
\begin{array}{c}
P \\
( \ S ) \\
\rightarrow \\
S \\
ID("a") := E \\
ID("b") := E \\
NUM(4) \\
NUM(4)
\end{array}
\]

Type checker does not need "(" or ")" or ",".

Parse Tree Example

Solution: generate abstract parse tree (abstract syntax tree) - similar to concrete parse tree, except redundant punctuation tokens left out.

Semantic Analysis: Symbol Tables

- Semantic Analysis Phase:
  - Type check AST to make sure each expression has correct type
  - Translate AST into IR trees
- Main data structure used by semantic analysis: symbol table
  - Contains entries mapping identifiers to their bindings (e.g. type)
  - As new type, variable, function declarations encountered, symbol table augmented with entries mapping identifiers to bindings.
  - When identifier subsequently used, symbol table consulted to find info about identifier.
  - When identifier goes out of scope, entries are removed.
Symbol Table Example

```plaintext
function f(b: int, c: int) =
    (print_int(b+c);
     let
        var j := b
        var a := "x"
    in
        print(a)
        print(j)
    end
    print_int(a)
)
```

σ₀ = {a → int}
σ₁ = {b → int, c → int, a → int}

Symbol Table Implementation

- Imperative Style: (side effects)
  - Global symbol table
  - When beginning-of-scope entered, entries added to table using side-effects. (old table destroyed)
  - When end-of-scope reached, auxiliary info used to remove previous additions. (old table reconstructed)
- Functional Style: (no side effects)
  - When beginning-of-scope entered, new environment created by adding to old one, but old table remains intact.
  - When end-of-scope reached, retrieve old table.

Imperative Symbol Tables

Symbol tables must permit fast lookup of identifiers.

- **Hash Tables** - an array of buckets
- **Bucket** - linked list of entries (each entry maps identifier to binding)

Suppose we wish to lookup entry for id \( i \) in symbol table:
1. Apply hash function to key \( i \) to get array element \( j \in [0, n-1] \).
2. Traverse bucket in table[\( j \)] in order to find binding \( b \).
   (table[x]: all entries whose keys hash to \( x \))
Hash tables not efficient for functional symbol tables. Insert \( a \mapsto \text{string} \Rightarrow \text{copy array, share buckets:} \)

Old Symbol Table Array

| i | a \mapsto \text{int} |

New Symbol Table Array

| i | a \mapsto \text{string} |

Not feasible to copy array each time entry added to table.

Better method: use binary search trees (BSTs).

- Functional additions easy.
- Need “less than” ordering to build tree.
  - Each node contains mapping from identifier (key) to binding.
  - Use string comparison for “less than” ordering.
  - For all nodes \( u \in L \), key\((u) < \text{key}(l) \)
  - For all nodes \( u \in R \), key\((u) \geq \text{key}(l) \)

Functional Symbol Table Example

Lookup:
Insert:

insert z ← i.r.t. create node z, copy all ancestors of z: