Today

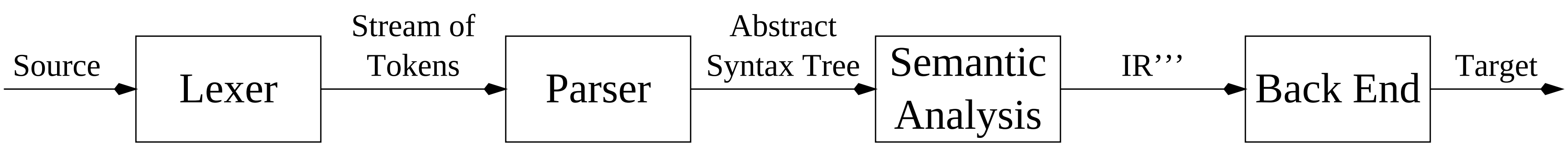
- Abstract Syntax
- Chapter 4

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.

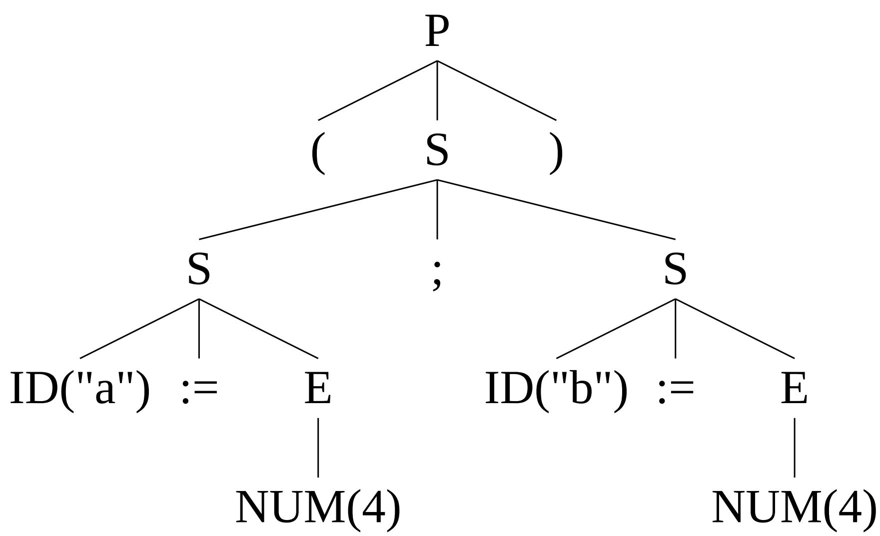
Parse Tree Example

$P \rightarrow (S)$
$S \rightarrow S ; S$
$S \rightarrow \text{ID} := E$

$E \rightarrow \text{ID}$
$E \rightarrow \text{NUM}$
$E \rightarrow E + E$

$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E / E$

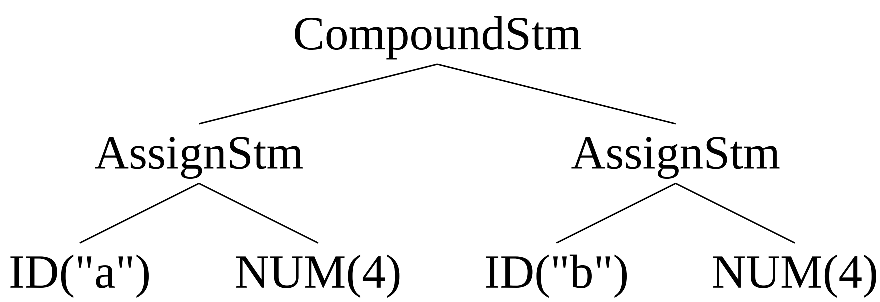
```
( a := 4 ; b := 5 )
```



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.



Abstract Syntax Tree Description

$P \rightarrow$ (S)
$S \rightarrow S$; S
$S \rightarrow$ ID := E

$E \rightarrow$ ID
$E \rightarrow$ NUM
$E \rightarrow E + E$

$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E / E$

Can describe abstract syntax tree structure using data types in ML:

```
type id = string
datatype binop = PLUS | MINUS | TIMES | DIV
datatype var = ID of id
datatype sym = CompoundStm of stm * stm
             | AssignStm of var * exp
     and exp = var
             | NUM of int
             | OpExp of exp * binop * exp
```

Abstract Syntax for Tiger

Positions

In order to report semantic errors, need to annotate each AST node with source file position of character(s) from which node was derived.

- ML-Lex specification: annotated each token-type with beginning and end positions of token.
- ML-Yacc specification: these positions are available in semantic actions.

  X$<n>$: returns attribute of nth occurrence of X.

  X$<n>$left: returns left-end position of token corresponding to X.

  X$<n>$right: returns right-end position of token corresponding to X.

Abstract Syntax for Tiger

```
structure A = Absyn
%%
%nonterm exp of A.exp | ...
%%
exp : INT               (A.IntExp(INT))
exp : exp PLUS exp      (A.OpExp {left = exp1,
                                  oper = A.PlusOp,
                                  right = exp2,
                                  pos = PLUSleft})
ty  : ARRAY OF ID       (A.ArrayTy(Symbol.symbol(ID),
                                   ARRAYleft))
```

- Identifiers in AST required to have *symbol* values.
- Lexer returns ID tokens with string values.

```
Symbol.symbol = fn:  string -> symbol
Symbol.name = fn:  symbol -> string
```

Examples

```
( a := "hi"; b)

A.SeqExp[(A.AssignExp{
              var = A.SimpleVar(Symbol.symbol("a"), 2),
              exp = A.StringExp("hi", 7),
              pos = 4}, 2),
         (A.VarExp(A.SimpleVar(Symbol.symbol("b"), 13)),
         13)]
```

Examples

```
let
  var a:= 5
  function f1():int = f2()
  function f2():int = f1()
in
  f1()
end
```

`A.FunctionDec` represents function declarations

- It takes *list* of function decs, not just one.
- List contains maximal consecutive sequence of function decs to simplify processing of mutually recursive functions.

Examples

```
A.FunctionDec[{name = Symbol.symbol("f1"),
               params = nil,
               result = SOME(Symbol.symbol("int"), _),
               body = A.CallExp{func = Symbol.symbol("f2"),
                                args = nil,
                                pos = _},
               pos = _},
              {name = Symbol.symbol("f2"),
               params = nil,
               result = SOME(Symbol.symbol("int"), _),
               body = A.CallExp{func = Symbol.symbol("f1"),
                                args = nil,
                                pos = _},
               pos = _}]
```

Tiger Language

- Simple control constructs:
  - if-then, if-then-else
  - while-loops, for-loops
  - function calls
- two basic types: int, string
  - facility to define record and array types
  - facility to define mutually-recursive types
- Supports nested functions, mutually-recursive functions.

Let-In-End Expressions

A Tiger program is expression. One important expression is *let-in-end*:

```
let
  <type declarations>
  <variable declarations>
  <function declarations>
in
  <sequence of expressions, separated by ';'>
end
```

Scope extends to end of expression sequence.

Type Declarations

```
type t1 = int
type t2 = string
type rec1 = {f1:int, f2:t2}
type intArray = array of int
```

Array lengths are not specified until creation.

Variable Declarations

```
var v1 := 4
var v2:string := "a"
var v3 := rec1 {f1 = 4, f2 = "b"}
```

- Field names must all be specified, must be in order.
- Record not allocated on *stack*, allocated on *heap* (malloc)
- v3 is a pointer to record structure on *heap*.

```
var z := 5 + v3.f1
var v4 := intArray[10] of 1
var w := z + v4[5]
```

- Accessed as `v4[0]` through `v4[9]`.
- Array allocated on *heap*, v4 pointer to heap object

Heap-Allocation

Heap-allocation is a run-time system issue.

- Programmer must “free” heap-allocated data, or
- Run-time system must do garbage collection.

Let’s not worry about this until after spring break.

Function Declarations

- Parameters are passed by value:

```
function add1(x:int):int = x + 1;
function changeRec(r:rec1) =
           (r.f1 := r.f1 + 10; r.f2 := "z")
```

- Function declarations can be nested:

```
function f1(y:int):int =
let
  var z := 5
  var w := 10
  function f2():int  = z + w * y
in
  f2()
end
```

Nested functions can access local variables or parameters of outer functions.

Mutual Recursion

Functions and types can be mutually recursive

- Mutually-recursive types must be declared *consecutively* with no intervening variable or function declarations.
- Each recursion cycle in a type definition must pass through a record or array type.
- Mutually-recursive functions must be declared *consecutively* with no intervening type or variable declarations.

Examples

```
type intList = {val:int, rest:intList}
```

{ 1, 2, 3 }

```
intList{val = 1,
        rest = intList{val = 2,
                       rest = intList{val = 3,
                                      rest = nil}}}
```

`nil` is a reserved word belonging to every record type.

- Essentially a NULL pointer.
- If record var has value `nil`, then field from variable cannot be selected.

Examples

Valid:

```
type rec2 = {a:int, b:rec3}
type rec3 = {c:string, d:rec2}
```

Invalid:

```
type rec4 = {f1:int, f2:rec5}
var z := 10
type rec5 = {f3:rec4, f4:string}
```

No intervening variable declarations allowed.

Invalid:

```
type t1 = t2
type t2 = t3
type t3 = t1
```

Recursion cycle does not pass through record or array.

Examples

Valid:

```
function isEven(n:int):int =
     if n = 0 then 1
     else isOdd(n-1)
function isOdd(n:int):int =
     if n = 0 then 0
     else isEven(n-1)
```

Invalid:

```
function f() = g()
function g() = h()
type a = array of string
function h() = f()
```

No intervening type declarations allowed.

Record/Array Distinction

Each declaration of record or array type creates *new* type, incompatible with all other record/array types.

```
let
  type a1 = array of int
  type a2 = array of int
  var v1 := a1[10] of 1
  var v2 := a2[10] of 1
in
  v1 := v2
end
```

Record/Array Distinction (continued)

Incompatible array types. Change to:

```
let
  type a1 = array of int
  var v1 := a1[10] of 1
  var v2 := a1[10] of 1
in
  v1 := v2
end
```

Name Spaces

2 different name spaces: one for *types*, one for *variables/functions*

- Can have type ”t” and variable ”t” in scope at same time.
- Cannot have variable ”t” and function ”t” in scope simultaneously.

Valid:

```
let
  type t = {s:int, t:int}
  var t := t{s = 4, t = 5}
in
  t
end
```

Name Spaces (continued)

```
let
  type t = int
  var t:t := 5
  function t():t = t + 10
in
  t()
end
```

Function `t` hides variable `t`.