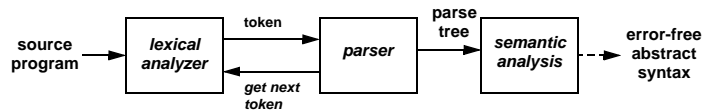


Compiler Front-End

- *Almost all compilers and interpreters contain the same front-end --- it consists of three components:*

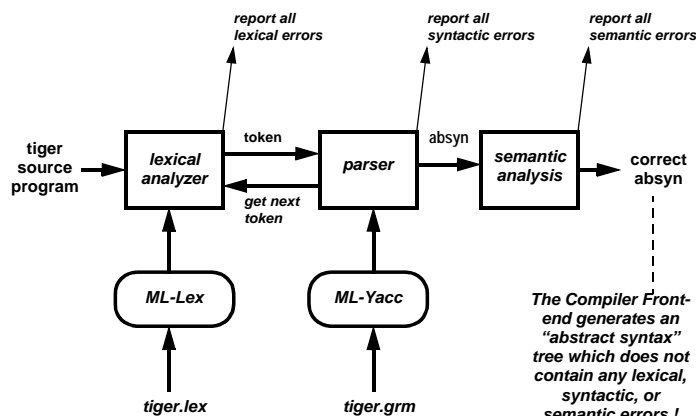
1. **Lexical Analysis** --- report lexical errors, output a list of tokens
2. **Syntax Analysis** --- report syntactic errors, output a parse tree
3. **Semantic Analysis** --- report semantic errors (e.g., type-errors, undefined identifiers, ...) --- generate a clean and error-free "abstract syntax tree"



“Concrete” vs. “Abstract” Syntax

- *The grammar specified in “tiger.grm” (for Yacc) is mainly used for parsing only ----- the key is to resolve all ambiguities. This grammar is called **Concrete Syntax**.*
- **Abstract Syntax (Absyn)** is used to characterize the essential structure of the program ----- the key is to be as simple as possible; **Absyn** may contain ambiguities.
- *The grammar for **Abstract Syntax** is defined using ML **datatypes**.*
- **Traditional Compilers:** do semantic analysis on Concrete Syntax --- implemented as “actions” in Section 3 of “tiger.grm” file (for Yacc)
- **Modern Compilers:** “tiger.grm” constructs the Abstract Syntax tree; the semantic analysis is performed on the Absyn later after parsing!

Tiger Compiler Front End



Tiger Program and Expression

- *A Tiger program prog is just an expression exp*
- *An expression can be any of the following:*

l-value	foo, foo.bar, foo[l]
Nil	nil
Integer literal	34
String literal	"Hello, World\n"
Sequencing	(exp; exp; ...; exp)
Function call	id(), id(exp{,exp})
Arithmetic expression	exp arith-op exp
Comparison expression	exp comp-op exp
Boolean operators	exp & exp, exp exp
Record creation	ty-id {id = exp, ...}, {}
Array creation	ty-id [exp ₁] of exp ₂
Assignment	lvalue := exp

Tiger Expression and Declaration

- *More Tiger expressions:*

```

If-then-else      if exp1 then exp2 else exp3
If-then          if exp1 then exp2
While-expression while exp1 do exp2
For-expression   for id:=exp1 to exp2 do exp3
Break-expression break
Let-expression   let decsq in {exp} end
  
```

- *A Tiger declaration sequence is a sequence of type, variable, and function declarations:*

```

dec -> tydec | vardec | fundec
decsq -> decsq dec | ε
  
```

Tiger Type Declaration

- *Tiger Type declarations:*

```

tydec -> type id = ty
ty -> id | { tyfields } | array of id
tyfields -> ε | id : type-id {,id: type-id}
  
```

- *You can define mutually-recursive types using a **consecutive sequence** of type declarations*

```

type tree = {key : int, children : treelist}
type treelist = {hd : tree, tl : treelist}
  
```

recursion cycle must pass through a record or array type !

Variable and Function Declaration

- *Tiger Variable declarations:*

```

short-form:      vardec -> var id := exp
long-form:      vardec -> var id : type-id := exp
  
```

“var x := 3” in Tiger is equivalent to “val x = ref 3” in ML

- *Tiger Function declarations:*

```

procedure:      fundec -> function id (tyfields) := exp
function:      fundec -> function id (tyfields):type-id := exp
  
```

- *Function declarations may be mutually recursive --- must be declared in a **sequence of consecutive** function declarations! Variable declarations **cannot** be mutually recursive !*

Tiger Absyn “Hack”

- *When translating from **Concrete Syntax** to **Abstract Syntax**, we can do certain syntactic transformations*

```

MINUS exp      ==>      0 MINUS exp
exp1 & exp2 ==>      if exp1 then exp2 else 0
exp1 | exp2 ==>      if exp1 then 1 else exp2
  
```

This can make **Abstract Syntax** even simpler.

Toy does not support Macros. If the source language supports macros, they can be processed here.

Tiger Semantics

- `nil` --- a value belong to every record type.
- Scope rule --- similar to PASCAL, Algol ---- support nested scope for types, variables, and functions; redeclaration will hide the same name.

```
function f(v : int) =
  let var v := 6
  in print(v);
  let var v := 7 in print(v) end;
  print(v);
  let var v := 8 in print(v) end;
  print(v)
end
```

- Support two different **name space**: one for types, and one for functions and variables. You can have a type called `f00` and a variable `f00` in scope at same time.

An Example

```
(* A program to solve the 8-queens problem, see Appel's book *)

let
  var N := 8

  type intArray = array of int
  var row := intArray [ N ] of 0
  var col := intArray [ N ] of 0
  var diag1 := intArray [N+N-1] of 0
  var diag2 := intArray [N+N-1] of 0

  function printboard() =
    (for i := 0 to N-1
     do (for j := 0 to N-1
        do print(if col[i]=j then " 0" else " .");
           print("\n"));
       print("\n"))

  function try(c:int) =
    (* for i:= 0 to c do print("."); print("\n"); flush(); *)
    if c=N then printboard()
    else for r := 0 to N-1
         do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
            then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                 col[c]:=r; try(c+1);
                 row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

in try(0)
end
```