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"

```
source program -> lexical analyzer -> parser -> semantic analysis -> error-free abstract syntax
```

"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[1]
  - 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 cmp-op exp
  - Boolean operators: exp & exp, exp | exp
  - Record creation: ty-id {id = exp, ...}, {}
  - Array creation: ty-id [exp] of exp
  - Assignment: lvalue := exp
Tiger Expression and Declaration

- More Tiger expressions:
  - If-then-else
    
    ```
    if exp_1 then exp_2 else exp_3
    ```
  - If-then
    
    ```
    if exp_1 then exp_2
    ```
  - While-expression
    
    ```
    while exp_1 do exp_2
    ```
  - For-expression
    
    ```
    for id:=exp_1 to exp_2 do exp_3
    ```
  - 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 Expression and Declaration

- 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 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}
    ```

    A recursion cycle must pass through a record or array type!

Tiger Absyn “Hack”

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

  ```
  MINUS exp ===> 0 MINUS exp
  exp_1 & exp_2 ===> if exp_1 then exp_2 else 0
  exp_1 | exp_2 ===> if exp_1 then 1 else exp_2
  ```

  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.

```tiger
function f(v : int) =
    let var v := 6 in print(v);
    let var v := 7 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 `foo` and a variable `foo` in scope at the same time.

```tiger
function printboard() =
    (for i := 0 to N-1 do (for j := 0 to N-1 do print(if col[i]=j then " O" 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
```

An Example

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

```tiger
let var N := 8
    type intArray = array of int
    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [2*N-1] of 0
    var diag2 := intArray [2*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 " O" 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
```