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 
                  |                  |                  
         token       |                  |                   
                get next token |
```

```
lexeme         parser tree       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

```
source program ---+--- lexical analyzer ---+--- parser 
                  |                  |                  
         token       |                  |                   
                get next token |
```

```
lexeme         parser tree       semantic analysis  error-free abstract syntax
```

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 comp-op exp
  - Boolean operators: exp & exp, exp | exp
  - Record creation: ty-id {id = exp, ...}, {}
  - Array creation: ty-id [exp1] of exp2
  - Assignment: lvalue := exp

```
ML-Lex  ML-Yacc

ML-Lex

ML-Yacc

The Compiler Front-end generates an "abstract syntax" tree which does not contain any lexical, syntactic, or semantic errors!

```
fio, foo.bar, foo[1] nil
34 "Hello, World\n"
(exp; exp; ...; exp)
id(), id(exp{,exp})
exp arith-op exp
exp comp-op exp
exp & exp, exp | exp
ty-id {id = exp, ...}, {}
ty-id [exp1] of exp2
lvalue := exp
Tiger Expression and Declaration

• More Tiger expressions:

If-then-else
\[
\text{if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } \text{exp}_3
\]

If-then
\[
\text{if } \text{exp}_1 \text{ then } \text{exp}_2
\]

While-expression
\[
\text{while } \text{exp}_1 \text{ do } \text{exp}_2
\]

For-expression
\[
\text{for } \text{id} := \text{exp}_1 \text{ to } \text{exp}_2 \text{ do } \text{exp}_3
\]

Break-expression
\[
\text{break}
\]

Let-expression
\[
\text{let } \text{decsq} \text{ in } \{ \text{exp} \} \text{ end}
\]

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

\[
\text{dec } \rightarrow \text{tydec} | \text{vardec} | \text{fundec}
\]

\[
\text{decsq } \rightarrow \text{decsq } \text{dec} | \text{\varepsilon}
\]

Tiger Type Declaration

• Tiger Type declarations:

\[
\text{tydec } \rightarrow \text{type id } \rightarrow \text{ty}
\]

\[
\text{ty } \rightarrow \text{id} | \{ \text{tyfields} \} | \text{array of id}
\]

\[
\text{tyfields } \rightarrow \text{\varepsilon} | \text{id : type-id \{,id : type-id\}}
\]

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

\[
\text{type tree } = \{ \text{key : int, children : treelist} \}
\]

\[
\text{type treelist } = \{ \text{hd : tree, tl : treelist} \}
\]

recursion cycle must pass through a record or array type!

Variable and Function Declaration

• Tiger Variable declarations:

short-form: \[
\text{vardec } \rightarrow \text{var id } := \text{exp}
\]

long-form: \[
\text{vardec } \rightarrow \text{var id : type-id } := \text{exp}
\]

“\text{var } x := 3” in Tiger is equivalent to “\text{val x } := \text{ref 3}” in ML

• Tiger Function declarations:

procedure: \[
\text{fundec } \rightarrow \text{function id (tyfields) } := \text{exp}
\]

function: \[
\text{fundec } \rightarrow \text{function id (tyfields):type-id } := \text{exp}
\]

• 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

\[
\text{MINUS } \text{exp} \rightarrow 0 \text{ MINUS } \text{exp}
\]

\[
\text{exp}_1 \text{ | } \text{exp}_2 \rightarrow \text{if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } 0
\]

\[
\text{exp}_1 \text{ & } \text{exp}_2 \rightarrow \text{if } \text{exp}_1 \text{ then } \text{exp}_2 \text{ else } \text{exp}_2
\]

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);
    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 foo and a variable foo in scope at same time.

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
```

Console output:
```
    .O.
    .O.
    .O.
    .O.
    .O.
    .O.
    .O.
    .O.
```