Introduction

Compiler is a program that can read a program in one language — the source language — and translate it into an equivalent program in another language — the target language

For implementation we use Standard ML of New Jersey for the Standard ML. Strict, statically typed functional programming language with modular structure.

What we have done ?

Compiler Front-End (partially)

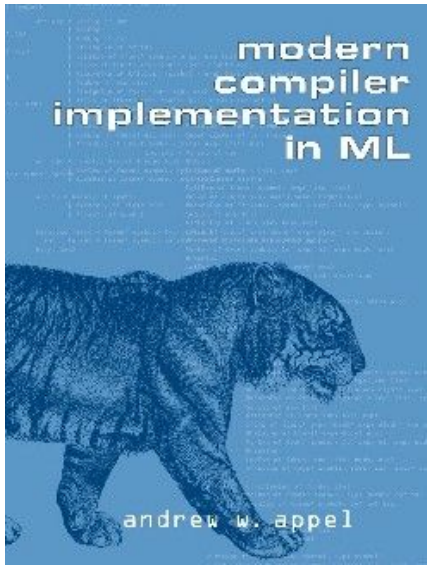
1. Lexical Analysis --- report lexical errors, output a list of tokens

2. Syntax Analysis --- report syntactic errors, output a parse tree

For this project, we relied on Andrew Appel's book

"Modern Compiler Implementation in ML".

This book gave us code presets and clear information in its exercises.



All skeleton files were uploaded from open source official site:

https://www.cs.princeton.edu/~appel/modern/ml/project.html

Lectures from professor Zouev were also a fundamental references especially for AST tree implementation

Tiger language: main principles

- Tiger program is just an expression
- 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
Assignment lvalue	:= 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

- Two named types *int* and *string* are predefined. Additional named types may be defined or redefined by type declarations records, arrays)
- Only integer numbers, simple arithmetic produce integer result.
- Comparison between strings are supported, produce boolean result 1 or 0)

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

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

Lexical analyzer

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks transformed into tokens
It is also discards white space and comments between them.

For Lexer we used a Lexical Analyzer Generator ML-lex

It uses regular expressions to describe classes of words.

This information is given to Lex as a specification.

As the output, we get the token and its position.

Set of Tokens:

"," => COMMA
":" => COLON
";" => SEMICOLON
"(" => LPAREN
")" => RPAREN
"[" => LBRACK
"]" => RBRACK
"{" => LBRACE
"}" => RBRACE
"." => DOT
"+" => PLUS
"-" => MINUS

"*" => TIMES
"/" => DIVIDE
"=" => EQUAL
"<>" => NEQUAL
"<" => LT
"<=" => LE
">" => GREATE
">=" => GREATEOREQ
"&" => AND
"|" => OR
":=" => .ASSIGN

"var" => VAR
"while" => WHILE
"for" => FOR
"to" => TO
"break" => BREAK
"let" => LET
"in" => IN
"end" => END
"function" => FUNCTION

"type" => TYPE
"array" => ARRAY
"if" => IF
"then" => THEN
"else" => ELSE
"do" => DO
"of" => OF
"nil" => NIL

Regular expressions :

escape_sequence = \\(n|t|[0-9]{3}|\"|\\);
ignored_sequence = [\ \t\f];
id = [a-zA-Z][a-zA-Z0-9_]*;
digit = [0-9];
whitespace = [\ \t];

Comments handling and string proceed

- Perform as state machine < initial> <comment>
- Nested and multiline comments are processed

```
/* This is example
 of multiline
 /*and nested  */ comment
 in  Tiger Language
*/
```

\n	A character interpreted by the system as end-of-line.
\t	Tab.
\^c	The control character c, for any appropriate c.
\ddd	The single character with ASCII code ddd (3 decimal digits).
\"	The double-quote character (").
\\	The backslash character (\).
\f___f\	This sequence is ignored, where f___f stands for a sequence of one or more formatting characters (a subset of the non-printable characters including at least space, tab, newline, formfeed). This allows one to write long strings on more than one line, by writing \ at the end of one line and at the start of the next.

Examples

Tiger programm:

```
/* define valid mutually recursive procedures */
let
	function do_nothing1(a: int, b: string)=
			do_nothing2(a+1)

	function do_nothing2(d: int) =
			do_nothing1(d, "str")

in
	do_nothing1(0, "str2")
end
```

Result of lexer:

LET

FUNCTION ID(*do_nothing1*) LPAREN ID(a) COLON ID(*int*) COMMA ID(*b*) COLON ID(*string*) RPAREN EQUAL
ID(do_nothing2) LPAREN ID(*a*) PLUS INT(*1*) RPAREN

FUNCTION ID(*do_nothing2*) LPAREN ID(*d*) COLON ID(*int*) RPAREN EQUAL ID(*do_nothing1*) LPAREN ID(*d*) COMMA STRING(*"str"*) RPAREN

IN

ID(*do_nothing1*) LPAREN INT(*0*) COMMA STRING(*"str2"*) RPAREN

END

Tiger programm:

```
let
	type myint = int
	type  arrtype = array of myint

	var arr1:arrtype := arrtype [10] of 0
in
	arr1
end
```

Result of lexer:

```
LET

	TYPE ID(myint)  EQUAL  ID(int)
	TYPE  ID(arrtype)  EQUAL ARRAY  OF ID(myint)

	VAR ID(arr1) COLON  ID(arrtype) ASSIGN ID(arrtype) LBRACK
	INT(10) RBRACK OF INT(0)

IN

	ID(arr1)

END
```

Tiger programm:

```
/* define valid recursive types */
let
	/* define a list */
	type intlist = {hd: int, tl: intlist}

	/* define a tree */
	type tree ={key: int, children: treelist}
	type treelist = {hd: tree, tl: treelist}

	var lis:intlist := intlist { hd=0, tl= nil }
in
	lis
end
```

Result of lexer:

```
LET

	TYPE ID(intlist) EQUAL LBRACE ID(hd) COLON ID(int) COMMA
	ID(tl) COLON ID(intlist) RBRACE

	TYPE ID(tree) EQUAL LBRACE ID(key) COLON ID(int) COMMA
	ID(children) COLON ID(treelist) RBRACE

	TYPE ID(treelist) EQUAL LBRACE ID(hd) COLON ID(tree) COMMA
	ID(tl) COLON ID(treelist) RBRACE

	VAR ID(lis) COLON ID(intlist) ASSIGN ID(intlist) LBRACE ID(hd)
	EQUAL INT(0) COMMA ID(tl) EQUAL NIL RBRACE

IN

	ID(lis)

 END
```

```
val it = true : bool
TIMES   114
DIVIDE   115
:7.1:comment was not closed
EOF   149
val it = () : unit
- 
```

```
/* simple nested comment /* */ */

/* complex nested comment /* /* */ /* */ */ */

/* illegal nested comment */ */

/* /* illegal nested comment */
~
```

Tiger language

- Procedures do not return result values but functions do.
- Functions may be recursive. Functions hide variables of the same name, and vice versa. No two functions in a sequence of mutually recursive functions may have the same name.
- If – then , If – then – else conditional statements
- While do ,for to do loops

Parser: grm for Tiger

DATA TYPES

The syntax of types and type declarations in Tiger is

tydec → **type** *type-id* = *ty*

ty → *type-id*
→ { *tyfields* } *(these braces stand for themselves)*
→ **array of** *type-id*

tyfields → ϵ
→ *id* : *type-id* {, *id* : *type-id*}

VARIABLES

vardec → **var** *id* := *exp*
→ **var** *id* : *type-id* := *exp*

DECLARATIONS

A declaration-sequence is a sequence of type, value, and function declarations; no punctuation separates or terminates individual declarations.

decs → {*dec*}

dec → *tydec*
→ *vardec*
→ *fundec*

FUNCTIONS

fundec → **function** *id* (*tyfields*) = *exp*
→ **function** *id* (*tyfields*) : *type-id* = *exp*

Parser: grm for Tiger

```
%term
EOF
 | ID of string
 | INT of int | STRING of string
 | COMMA | COLON | SEMICOLON | LPAREN | RPAREN | LBRACK | RBRACK
 | LBRACE | RBRACE | DOT
 | UMINUS
 | PLUS | MINUS | TIMES | DIVIDE | EQ | NEQ | LT | LE | GT | GE
 | AND | OR | ASSIGN
 | ARRAY | IF | THEN | ELSE | WHILE | FOR | TO | DO | LET | IN | END | OF
 | BREAK | NIL
 | FUNCTION | VAR | TYPE
 | LOWER_THAN_ELSE
```

```
%nonterm
exp
| program
| decs | dec | tydec | vardec | fundec
| typeid | ty | tyfields | lvalue | expseq |
args
| recfields
```

```
%right OF
%nonassoc FUNCTION VAR TYPE THEN DO ASSIGN
%nonassoc ELSE
%left AND OR
%nonassoc EQ NEQ LT LE GT GE
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS
```

```
decs : dec          ()
     | dec decs     ()

dec : tydec  ()
    | vardec ()
    | fundec ()
```

```
tydec : TYPE ID EQ ty ()

ty : ID                        ()
   | LBRACE tyfields RBRACE ()
   | ARRAY OF ID               ()

tyfields :                                ()
         | ID COLON ID                    ()
         | ID COLON ID COMMA tyfields ()
```

```
vardec : VAR ID ASSIGN exp          ()
       | VAR ID COLON ID ASSIGN exp ()

fundec : FUNCTION ID LPAREN tyfields RPAREN EQ exp          ()
       | FUNCTION ID LPAREN tyfields RPAREN COLON ID EQ exp ()
```

```
lvalue: ID  ()
      | lvalue_not_id ()

lvalue_not_id : lvalue DOT ID ()
              | ID LBRACK exp RBRACK ()
              | lvalue_not_id LBRACK exp RBRACK ()

args :                  ()
     | exp              ()
     | exp COMMA args ()

recfields :                                ()
          | ID EQ exp                      ()
          | ID EQ exp COMMA recfields ()
```

```
exp : lvalue                                    ()
    | lvalue ASSIGN exp                         ()
    | NIL                                       ()
    | INT                                       ()
    | STRING                                    ()
    | LPAREN expseq RPAREN                      ()
    | MINUS exp %prec UMINUS                    ()
    | ID LPAREN args RPAREN                     ()
    | exp PLUS exp                              ()
    | exp TIMES exp                             ()
    | exp MINUS exp                             ()
    | exp DIVIDE exp                            ()
    | exp EQ exp                                ()
    | exp NEQ exp                               ()
    | exp LT exp                                ()
    | exp LE exp                                ()
    | exp GT exp                                ()
    | exp GE exp                                ()
    | exp AND exp                               ()
    | exp OR exp                                ()
```

```
expseq :                                        ()
       | exp                                    ()
       | exp SEMICOLON expseq                   ()
```

```
    | ID LBRACE recfields RBRACE                ()
    | ID LBRACK exp RBRACK OF exp               ()
    | IF exp THEN exp                           ()
    | IF exp THEN exp ELSE exp                  ()
    | WHILE exp DO exp                          ()
    | FOR ID ASSIGN exp TO exp DO exp           ()
    | BREAK                                     ()
    | LET decs IN expseq END                    ()
```

Result:
compliance with grammatical rules

```
3   let
4       var a:= 0
5   in
6       i:=0 to 100 do (a:=a+1;())
7   end
```

```
./TigerProgramms/test.tig:6.2:syntax error: inserting  FOR
```

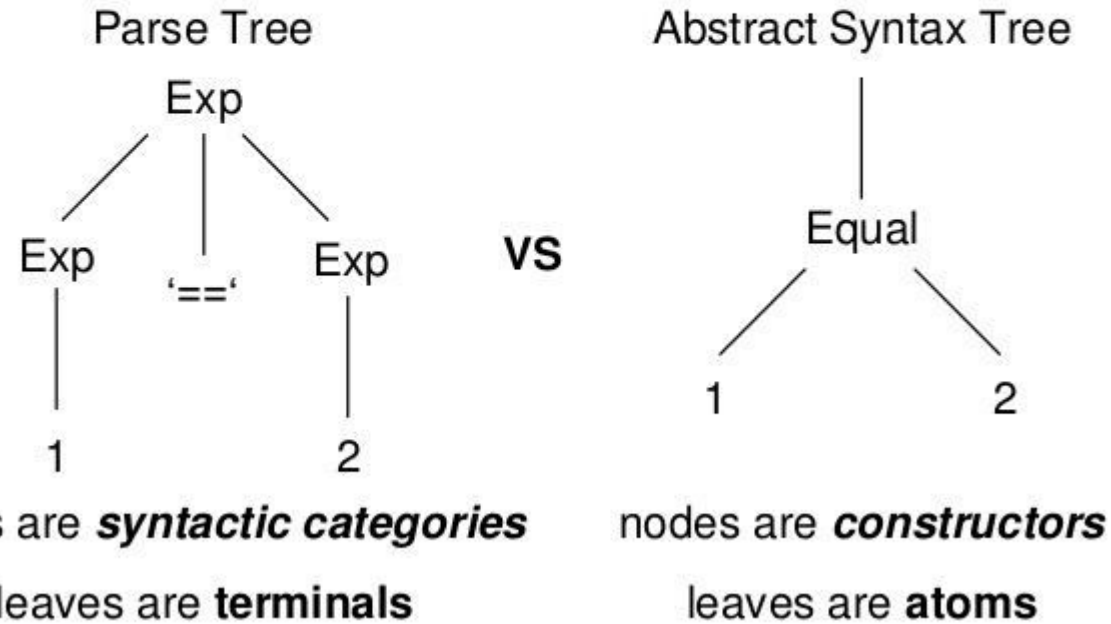
```
2   let
3       var a:= 0
4       for i:=0 to 100 do (a:=a+1;())
5   end
```

```
../TigerProgramms/test.tig:4.2:syntax error: inserting  IN
val it = () : unit
```

```
2   let
3       val a:= 0
4   in
5       for i:=0 to  100 do (a:=a+1;())
6   end
```

```
- Parse.parse("../TigerProgramms/test.tig");
../TigerProgramms/test.tig:3.6:syntax error: replacing  ID with  VAR
val it = () : unit
```

Contrasting Abstract and Concrete Syntax



nodes are ***syntactic categories***

leaves are **terminals**

nodes are ***constructors***

leaves are **atoms**

AST

```
datatype var = SimpleVar of symbol * pos
         | FieldVar of var * symbol * pos
         | SubscriptVar of var * exp * pos

and exp = VarExp of var
       | NilExp
       | IntExp of int
       | StringExp of string * pos
       | CallExp of {func: symbol, args: exp list, pos: pos}
       | OpExp of {left: exp, oper: oper, right: exp, pos: pos}
       | RecordExp of {fields: (symbol * exp * pos) list,
                  typ: symbol, pos: pos}
       | SeqExp of (exp * pos) list
       | AssignExp of {var: var, exp: exp, pos: pos}
       | IfExp of {test: exp, then': exp, else': exp option, pos: pos}
       | WhileExp of {test: exp, body: exp, pos: pos}
         | ForExp of {var: symbol, escape: bool ref,
                lo: exp, hi: exp, body: exp, pos: pos}
       | BreakExp of pos
       | LetExp of {decs: dec list, body: exp, pos: pos}
       | ArrayExp of {typ: symbol, size: exp, init: exp, pos: pos}
```

```
and dec = FunctionDec of fundec list
       | VarDec of {name: symbol,
                  escape: bool ref,
                  typ: (symbol * pos) option,
                  init: exp,
                  pos: pos}
       | TypeDec of {name: symbol, ty: ty, pos: pos} list
       | StartOfDecList of unit

and ty = NameTy of symbol * pos
      | RecordTy of field list
      | ArrayTy of symbol * pos

and oper = PlusOp | MinusOp | TimesOp | DivideOp
        | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp
```

Examples:

```
/* an array type and an array variable */
let
    type  arrtype = array of int
    var arr1:arrtype := arrtype [10] of 0
in
    arr1
end
```

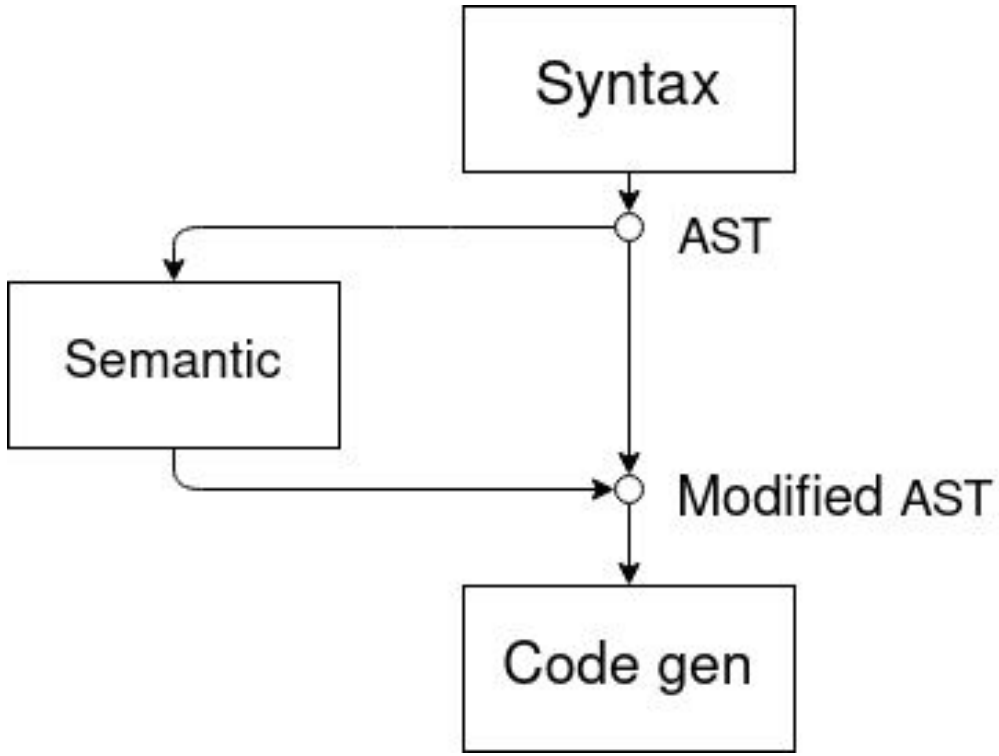
```
LetExp([
 TypeDec[
  (arrtype,
   ArrayTy(int))],
 VarDec(arr1,true,SOME(arrtype),
  ArrayExp(arrtype,
   IntExp(10),
   IntExp(10)))],
 SeqExp[
  VarExp(
   SimpleVar(arr1))])
val it = () : unit
```

```
/* error : procedure returns value  and procedure is used in arexpr */
let

/* calculate n! */
function nfactor(n: int) =
        if  n = 0
            then 1
            else n * nfactor(n-1)

in
    nfactor(10)
end
```

```
LetExp([
 FunctionDec[
  (nfactor,[
   (n,true,int)],
NONE,
   IfExp(
    OpExp(EqOp,
     VarExp(
      SimpleVar(n)),
     IntExp(0)),
    IntExp(1),
    OpExp(TimesOp,
     VarExp(
      SimpleVar(n)),
     CallExp(nfactor,[
      OpExp(MinusOp,
       VarExp(
        SimpleVar(n)),
       IntExp(1))])))]],
 SeqExp[
  CallExp(nfactor,[
   IntExp(10)])])
val it = () : unit
-
```

Code generation

Development process



Code generation

Success

- For loops
- if statements
  - multiple condition
- variable assignment
- function declaration, call

Failures

- Array declaration
- Global frame usage
- implicit type

Code generation

Compiler call

./demi tiger_file_name [executable_file_name]

```
sml ubuntu_launch.sml $1
if [$2 = ""]
then
    str="tig_program"
else
    str=$2
fi

echo "gcc compile start: "
gcc c_code.c -o $str
echo "Tiger program executable: $str"
```

```
  void try(int c){
  if( c == N){
     printboard();
    }else{
     for( int r =  0; r <=  N - 1; r++){
     if( row[ r] == 0 || diag1[ r + c] == 0 || diag2[ r + 7 - c] == 0){
        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;

      }
      }
     }

  }
int main()
{
  try( 0);
  return 0;
}
```