

Lecture #35

Compilers

Introduction

- Ref. Beck chapter 5
- Compiler = a kind of translator
 - high-level language --> machine code
- Translation “gap” is larger than assembly
 - sophisticated data structures
 - arrays, records, classes, ...
 - sophisticated control structures
 - if, while, switch, function calls, nested scopes, ...

The “High-level” Language

- Two aspects to language definition:
 1. Syntax
 - what are *legal* programs?
 - *i.e.*, what is accepted by the compiler?
 2. Semantics
 - what does a legal program *mean*?
 - *i.e.*, into what machine code is it translated?

Modular Decomposition

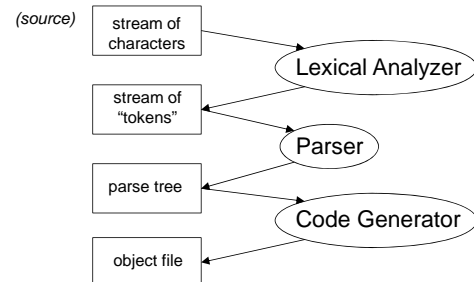
- View input as a stream of characters

Prog_____ORIG\nX_____FILL_____



- Compiler must give this stream *structure* in order to perform the translation

Coarse-Grained Decomposition



Lexical Analysis

- First step of compilation process
- Also called:
 - “scanner”, “tokenizer”, “lexer”
- Scans program (often stripping comments)
- Recognizes:
 - keywords, operators, identifiers, ints, floats, ...
- All of these called “tokens”

Tokens

- A token is defined by:
 1. Type (e.g., integer)
 2. Value (e.g., 312)
- Keywords (e.g., “while”) often have their own token type (no associated value)
- Example:
 - MEAN := SUM DIV 100;

Tokens - Example

- Result of tokenizing:

<u>Line</u>	<u>Token Type</u>	<u>Token Value</u>
13	<i>id</i>	MEAN
	<i>:=</i>	
	<i>id</i>	SUM
	DIV	
	<i>int</i>	100
	<i>;</i>	

Token Definition

- How do we define what is and is not a token?
- Some token types seem to be simple
 - e.g., keywords (defined by a string, e.g., “while”)
- Other token types, however, are more difficult
 - e.g., what is an “id”? what is an “int”?
- Also, language syntax rules can add complexity
 - line continuation characters
 - white space between, within, tokens
- A general notation is needed for defining all token types

Regular Expressions

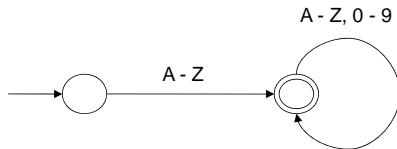
- Examples
 - *label* :: [A - Z] [A - Z, 0 - 9] {0, 5}
 - a label is a capital letter followed by 0 to 5 characters that may be capital letters or numbers
 - *int* :: 0 | [1 - 9] [0 - 9]*
 - an int is either a 0 or a digit in range 1 to 9, followed by any number (0 or more) digits
- Regular expressions are equivalent to...

Finite State Automata (FSA)

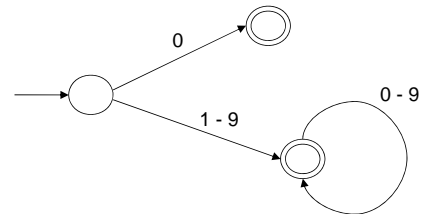
- Definition:
 - a finite collection of nodes (“states”)
 - directed arcs (“transitions”) between nodes
 - arcs are labeled
 - special nodes:
 - 1 start
 - at least 1 final (or “ending” or “accepting”)
- An FSA “accepts” a string iff it can read the string and end up in a “final” state

Example: LongLabel

longlabel :: [A - Z] [A - Z, 0 - 9]*



Example: Int



Exercise

- Consider a token type: LongLabelUS
 - same rules as for LongLabel (for letters/no.s), but underscores are also permitted
 - no “_” at start
 - no “_” at end
 - no 2 “_”s in a row
- Should the following be accepted?
BUFFER1 BUFF_SIZE BUFF__S
T_B_SIZE BUFF_ 1SIZE
- Write an FSA for this token type

Exercise: Solution

Lexical Analysis

- We could write code to recognize LongLabel directly
 - see Beck figure 5.10
 - but this is hard to read, modify, maintain, ...
- Much easier to read and understand FSA!
- Scanners are often built *automatically* from FSA descriptions!

Step #2: Parsing

Grammar

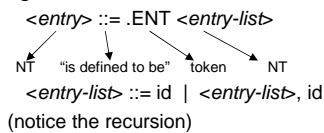
- Defines syntax of language
- Given as a collection of rules
 - “transformations”
 - e.g., $(X) \rightarrow *(TX)*$
 - maps string on left into string on the right
- One particular (and important) kind of grammar: “Context-Free Grammar” (CFG)

CFG

- Two kinds of symbols:
 - terminals
 - non-terminals
- Each non-terminal has an associated rule
 - single non-terminal is the *only* thing on the left
 - e.g., $p \rightarrow \epsilon \mid (p) \mid pp$
- application: $p \rightarrow pp \rightarrow (p)p \rightarrow (pp)p$
 $\rightarrow ((p)p)p \rightarrow ((())p)p \rightarrow \dots \rightarrow (())(())(())$

BNF (Backus-Naur Form)

- A common notation for CFGs
- Invented to define the syntax of ALGOL60
- Terminals are... tokens!
- e.g.,



BNF II

- One special “start” symbol
 - e.g., $\langle program \rangle ::= id \langle origin \rangle \langle body \rangle \langle end \rangle$
- Notice trade-off between scanner & parser
 - tokenizer could return “smaller” tokens
 - then rules in parser become more complicated
 - e.g., $\langle read \rangle ::= 'R' 'E' 'A' 'D' \dots$
 vs $\langle read \rangle ::= READ (\langle id-list \rangle) \dots$
- Example: see PASCAL BNF p. 228

Parse Trees

- Record the application of BNF rules
 - root: the start symbol
 - internal nodes: non-terminal symbols
 - leaves: terminals (i.e., tokens)
- Example: using PASCAL BNF, what is the parse tree for MEAN := SUM DIV 100 ?

Parse Trees - Example

Parse Trees - Exercise

- Exercise: FOR I := 1 TO 10 DO
READ (TEMP)
- Exercise:
 - $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle - \langle \text{exp} \rangle \mid \text{int}$
 - parse $3 + 6 - 2$
 - answer?
- Grammar that allows more than one parse tree to be formed for the same token sequence is called “_____”

Lecture #36

Algorithm

- How do we calculate a parse tree?
- Two approaches:
 - bottom-up (start at leaves)
 - top-down (start at root)

Operator Precedence Parsing

- An early bottom-up technique
- Define binding priority between “operators”
 - e.g., $A + B * C - D$
 - priority: $'+' < '*'$
 - resulting parse tree:
- Operators are *tokens*
 - binding priority defined between *terminals* only
- Grammar (implicitly) defines a matrix of binding priorities
 - see Beck fig 5.11
 - note 1: not all pairs defined!
 - note 2: ordering is not antisymmetric!

Example Application

- Reductions: Parse Tree Creation:
- ... BEGIN READ (id) ; ...
 $\langle \quad = \quad \rangle \rangle$
 - ... BEGIN READ (nt1) ; ...
 $\langle \quad = \quad = \quad \rangle$
 - ... BEGIN nt2 ; ...
 $\langle \quad$
- Notes:
- Each reduction adds an internal node to parse tree
 - Internal node names don't matter!

Shift-Reduce Parsing

- Generalizes idea of operator precedence
- Scan tokens, placing them on a stack } “shift”
- Group tokens at top of stack:
 - pop them all off
 - push corresponding non-terminal
 } “reduce”
- Repeat until done
 - should be left with _____

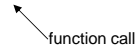
Shift-Reduce Parsing II

- Grammar must be “LR”
 - “left-to-right scan of the input, producing a right-most derivation”
 - symbols to be reduced always appear at *top* of stack (never inside it)
- Need to “look ahead” to decide how/when to reduce
 - if we only need to look ahead 1 token: LR (1) grammar

Recursive Descent

- Top-down approach
- Each rule has associated function
 - scan forward
 - try to identify string matching this rule
- Function may have to call other functions
 - see Figure 5.16, example for `find <read>`:


```
find "READ";
find "(" ;
find <id-list>;
find ")"
```



Recursive-Descent - Problem

- Subtle potential problem: “left-recursion”
 - the left-most (first) symbol in the BNF rule is the same non-terminal (recursive)
 - e.g. `<id-list> ::= id | <id-list>, id`
- If we want to expand 2nd alternative, first call ourselves! (i.e., infinite recursion)
- Solution: change notation slightly
 - `<id-list> ::= id { , <id-list> }`
 - function *always consumes* a token before recursion

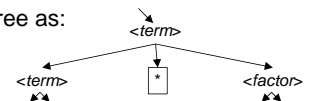
Step #3: Code Generation

Introduction

- Use a collection of routines
- 1 routine / rule in the grammar
 - called “semantic” or “code-generating” routines
- 2 approaches:
 - create entire tree
 - then “walk” the tree, generating code
 - generate code as we go
 - when a grammar rule is recognized, call the corresponding code-generating routine

Example

- Consider: `<term> ::= <term> * <factor>`
- Occurs in parse tree as:



- Generate code as we come up the tree
 - keep track of where (which registers) results of lower nodes are stored
 - generate code for * operation
 - keep track of where result is placed

Optimization

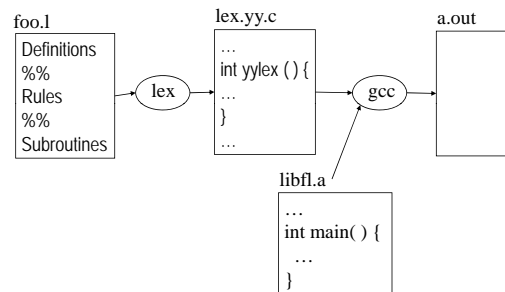
- An optimizing compiler tries to generate the most “efficient” object code
 - time (fast execution times)
 - space (small object files)
- Requires sophisticated analysis
- Often uses an “intermediate form” of code
 - not executed directly
 - analyzed for deciding register allocation, instruction ordering, branch shadows, etc...

Lecture #37

Lex & Yacc

- Unix tools for building compilers
 - lex: “lexical analyzer”
 - yacc: “yet another compiler compiler”
- Lex
 - input: regular expressions (plus actions with each rule)
 - output: a **lexical analyzer** (a C function, `yylex()`)
- Yacc
 - input: context free grammar (plus actions with each rule)
 - output: a **parser** (a C function, `yyparse()`, that calls `yylex()`)
- Link it all together
 - **compiler**
- “The asteroid to kill this dinosaur is still in orbit”

Lex Process



Lex Input File (Extension .l)

- Definitions include:
 - convenient short-hands for REs. e.g., `digit [0-9]`
 - code to include at top of lexer (enclosed in `%{ ... %}`)
- Rule syntax: pattern action
 - pattern is a RE to be recognized. e.g., `[- +](digit)+`
 - action is a statement to execute when RE is recognized
- Subroutines: user-provided functions
 - defaults are provided, but can be over-ridden here
 - e.g., `int yywrap()`, `int main()`
 - main would typically call `yylex()`

Lex Example: Counting Identifiers

```
digit [0-9]
letter [A-Za-z]
%{
  int count=0;
  %}
%%
{letter}({letter}({digit})*) count++;
%%
int main(void) {
  yylex();
  printf("# of id = %d\n", count);
  return 0;
}
```

Create lexer:

- `lex id_count.l`
- `gcc lex.yy.c -fl`

Resulting executable:

- reads entire input (with one call to `yylex()`)
- recognizes occurrences of RE, consuming those characters and executing corresponding action
- unrecognized characters copied to `stdout`
- input comes from `stdin`

Count Id Example: Sample Run

```
alpha% cat prog.pas
begin
if (x > 0) then
begin
y = 10
end
else
begin
y = 20
end
end
alpha%
```

```
alpha% cat prog.pas | a.out
. . . (-> 0). °
. . . °
. . . . . = 10°
. . . °
. . . °
. . . °
. . . °
. . . . . = 20°
. . . °
. . . °
. . . °
# of ident = 12
alpha%
```

Counting Identifiers, Improved

```
digit [0-9]
letter [A-Za-z]
%{
int count;
}%
%{
(letter)((letter)|(digit))* count++;
[ \t\n] ; /*ignore whitespace*/
}%
int main(int argc, char *argv[]){
if (argc > 1)
yyin = fopen(argv[1],"r");
else yyin = stdin;
yylex();
printf("# of id = %d\n", count);
return 0;
}
```

Changes:

- recognizes white space
 - no action (still not copied)
- takes input from file

Rerun on prog.pas:

```
alpha% a.out prog.pas
(>0)=10=20# of id = 12
alpha%
```

INSERT: Pascal Lex Example

Input file for simple Pascal syntax (pascal.l)
Result: a 701-line C program!

Pascal Example: Sample Run

```
alpha% ls
pascal.l prog.pas
alpha% lex pascal.l
alpha% ls
lex.yy.c pascal.l prog.pas
alpha% wc lex.yy.c
701 1858 13957 lex.yy.c
alpha% gcc lex.yy.c -fl
alpha%
```

```
alpha% a.out prog.pas
A keyword: begin
A keyword: if
Unrecognized character: (
An identifier: x
Unrecognized character: >
An integer: 0 (0)
...
An integer: 20 (20)
A keyword: end
A keyword: end
alpha%
```

Yacc Input File (Extension .y)

- Definitions include:
 - token declarations (e.g., %token INTEGER)
 - code to include at top of parser (enclosed in %{ ... %})
- Rule syntax: BNF
 - each rule of grammar also has an action
 - action is a C statement to execute when rule recognized
- Subroutines: override default implementations
 - int yyerror()
 - called if syntax error detected
 - int main()
 - calls (generated) yyparse()

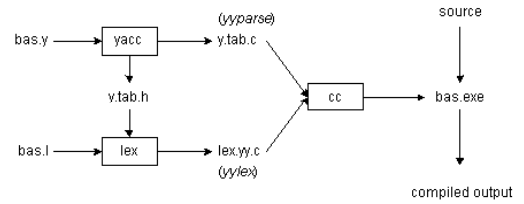
Yacc Example: Expressions

```
%token INTEGER
%%
prog:
prog expr '\n' {printf("%d\n", $2);}
| ;
expr:
INTEGER { $$ = $1;}
| expr '+' expr { $$ = $1 + $3;}
| expr '-' expr { $$ = $1 - $3;}
;
%%
int main(void) {
yyparse();
return 0;
}
```

Using Lex and Yacc Together

- yacc-generated `yyparse()` calls lex-generated `yylex()`
- Token type declarations
 - yacc generates header file `y.tab.h`
 - this file `#included` into lexer
- Token type and value identification
 - occurs in rules of lexer
 - set global value `yylval` to token value
 - return int representing the token type

Overview of Files



```
yacc -d bas.y # create y.tab.h, y.tab.c
lex bas.l # create lex.yy.c
cc lex.yy.c y.tab.c -o bas.exe # compile/link
```

Lecture 38

Decompression
Review