The Phases of a Compiler

- Lexical Analysis – we will cover
- Syntax Analysis – we will cover
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generator

Backus-Naur Form (BNF)

- Also called context free grammar
- Has 4 components
  - A set of tokens known as terminal symbols
  - A set of non-terminals.
  - A set of productions where each production consists of a non-terminal, called the left side of the production, an arrow, and a sequence of tokens and/or non-terminals, called the right side of the production
  - A designation of one of the non-terminals as the start symbol.

Tokens and non-terminals

- Are keywords, semicolons, and other lexical elements
- non-terminals are variables that may represent a sequence of tokens
- The start symbol is normally just the first line in the grammar

A Grammar

- 9+5-2
- The following grammar describes the syntax of the above expression:
  - list -> list + digit
  - list -> list – digit
  - list -> digit
  - digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

What are?

- The tokens?
- The nonterminals?
- The start symbol?

Once we have a Grammar

- We can create a parse tree from it!
- Parse trees pictorially show how the start symbol of a grammar derives a string in the language.
- Finding a parse tree for a given string of tokens is called “parsing” that string.
Context Free Grammar

In a context free grammar, a parse tree is a tree with the following properties:
- The root is labeled by the start symbol.
- Each leaf is labeled by a token or E (empty)
- Each interior node is labeled by a nonterminal.
- If A is the nonterminal labeling some interior node and X₀, X₁, X₂, …, Xₙ are the labels of the children of that node from left to right, then A → X₀X₁X₂…Xₙ is a production.

Ambiguity

- Some grammars have more than 1 parse tree for a given string
- Example:

```
string -> string + string | string – string
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Associativity

- By convention: 9+5 + 2 is equivalent to (9+5)+ 2
- This is left associative. Left associative trees grow to the left and right associative trees grow to the right.
- Some operators are right associative
  - In C, a=b=c is treated like a=(b=c)
  - a=b=c may be generated by the following grammar:
    ```
    right -> letter = right | letter
    letter -> a | b | ... | z
    ```

Precedence

- 9+5*2 can be evaluated as (9+5)*2 or as 9+(5*2). Associativity doesn’t resolve this ambiguity!
- * has a higher precedence if * takes its operands before +
- We can create 2 nonterminals expr and term for the two levels of precedence and an extra nonterminal factor for generating basic units in expressions.

The Grammar

```plaintext
factor -> digit | ( expr )
term -> term * factor
      | term / factor
      | factor
expr -> expr + term
      | expr – term
      | term
```

For the language statements shown...

- What are the terminals used?
- Write the parse tree for the terms
- Come up with a grammar that explains the following statements and allows for
  - multiple digits in a row
  - Precedence where appropriate

```
(setf x (* 1 2))
(setf x (+ x 5))
:int x = 1*2;
:int x = x + 5;
$x = 1 * 2$;
$x = x + 5$;
```
The Java Grammar

Java snippet:
```
int x = 1 * 2;
x = x * 5;
```

Grammar:
```
statement -> declaration assignment
declaration -> primitiveType id ;
| primitiveType id = expr ;
assignment -> id = expr ;
primitiveType -> int
eexpr -> eexpr + term | expr – term | term
term -> term * factor | term / factor | factor
factor -> digits | id | ( expr )
digits -> digit | digits digit
```

Terminals used:
- id
- int
- digit
- * / ( ) ; =

Perl Grammar

Perl snippet:
```
$x = 1 * 2;
$x = $x + 5;
```

Grammar:
```
assignments -> assignments assignment | assignment
assignment -> scalarId = expr ;
scalarId -> $ id
eexpr -> expr + term | expr – term | term
term -> term * factor | term / factor | factor
factor -> digits | scalarId | ( expr )
digits -> digit | digits digit
```

Terminals used:
- id
- digit
- * / ( ) ; =

Lisp Grammar

Lisp snippet:
```
(setf x (* 1 2))
(setq x (+ x 5))
```

Grammar:
```
functions -> functions function | function
function -> ( operator operand operand )
operator -> setf | * | - | / | +
operand -> symbol | digits | function
digits -> digit | digits digit
```

Terminals used:
- symbol
- setf
- digit
- * / ( )

Can you find a problem with this?

The Real Grammar
Takes More Into Account

From the Java language specs:
- Identifier: IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral
- IdentifierChars: JavaLetter | JavaLetterOrDigit
- JavaLetter: any Unicode character that is a Java letter (see below)
- JavaLetterOrDigit: any Unicode character that is a Java letter-or-digit (see below)

More Java Comment Defs

- Comment: TraditionalComment | EndOfLineComment | DocumentationComment
- TraditionalComment: / * NotStar CommentTail
- EndOfLineComment: // CharactersInLineopt LineTerminator
- DocumentationComment: / * * CommentTailStar
- CommentTail: * CommentTailStar | NotStar CommentTail
- CommentTailStar: / * CommentTailStar | NotStarNotSlash | CommentTail
- NotStar: InputCharacter but not * | LineTerminator
- NotStarNotSlash: InputCharacter but not * or / | LineTerminator
- CharactersInLine: InputCharacter | CharactersInLine
- InputCharacter

Lexical Analysis

- The goal of the lexical analyzer is to read input characters and produce as output a sequence of tokens that the parser may use for syntax analysis
- flex is a lexical analyzer
Issues in Lexical Analysis

- There are several reasons for separating lexical analysis and parsing:
  - Simpler design
  - Compiler efficiency is improved as the separation allows a more specialized module for the task
  - Compiler portability is enhanced. Input anomalies may be restricted to the lexical analyzer.

Keywords

- Grammars typically use keywords to recognize statements
- Example from Pascal (id represents an identifier):
  ```
  stmt -> id := expr
  | if expr then stmt
  | if expr then stmt else stmt
  | while expr do stmt
  | begin opt_stmts end
  ```
- Is this ambiguous?

Question

- Where did the keywords/id come from?
  - Answer: The scanner
  - The scanner often uses regular expressions in order to help tokenize the program. The tokens that come out of the scanning phase are the terminals used in the BNF parsing phase.
  - Flex is the “fast lexical scanner” and bison is the parser we will be using

Regular Expressions in Flex

- [see the handout]
- How do I:
  - Create an id for Java?
  - Create a real number?
  - Create the keyword “for”?
  - Create a leet X?

The Format of Flex Files

```
%{
  .h files, variables, and c-constructs for the program go here
%
Definitions go here
%
Rules go here
%
User subroutines go here
%
}
```

An Example – What’s it do?

```
%
%include "calc.tab.h"
%
%{
  [a-z]   { yylval = *yytext - 'a'; return VARIABLE; }
  [0-9]+  { yylval = atoi(yytext); return INTEGER; }
  :=      return ASSIGN;
  \+      return PLUS;
  \*      return TIMES;
  \n      return NEWLINE;
  \s      ;
  \$\|\|\|\} yyerror("Invalid character");
%
int yywrap() {
  return 1;
}
```
Can You Write?

- A flex program that
  - ignores all white space and tabs
  - Prints "%s is a verb" for all verbs (you can think of)
  - Prints "%s is not a verb" for all other words and symbols

Bison

- A parser that parses BNF
  - It may be used to do semantic analysis, but this is not normally what’s done with a larger language. Normally, the parser would create a symbol table and parse tree and semantic analysis would be separate.

A Simple Bison Example

```c
%{
#include <stdio.h>
%
%token INTEGER PLUS TIMES
expr: INTEGER            { printf("INTEGER\n"); }
| expr PLUS expr   {printf("PLUS\n");}
| expr TIMES expr  {printf("TIMES\n");}
; 
%
int yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
  return 0;
}
int main() {
  yyparse();
  return 0;
}
```

Ambiguity

- Suppose we have the following ambiguous grammar:
  1. E -> E + E
  2. E -> id

  Let’s parse x + y + z.

A reduce-reduce conflict

- If the grammar is:
  - E -> T
  - E -> id
  - T -> id

  There are two different reductions possible with id on the stack. In that case, Bison will take the first rule in the listing.
  - Of course it’s better to avoid ambiguity. Often, you do not have to rewrite the whole grammar. You can specify in Bison whether operators are left or right associative, and you can define their precedence.

An Example

```c
%{
#include <stdio.h>
int sym[26];
%
%token VARIABLE ASSIGN INTEGER PLUS MINUS NEWLINE
%left PLUS
%left TIMES
```
Bison Ex. Cont’d

```
program: program statement
| " empty "
| statement: expr NEWLINE  { printf("%d\n", $1); }
| VARIABLE ASSIGN expr NEWLINE
|    { sym[$1] = $3; } 
| expr: INTEGER            { $$ = $1; }
| VARIABLE        { $$ = sym[$1]; }
| expr PLUS expr   { $$ = $1 + $3; }
| expr TIMES expr  { $$ = $1 * $3; }
| %
```

RIT Computer Science Dept.

Bison Ex. Cont’d

```
%%
int yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main() {
    yyparse();
    return 0;
}
```

RIT Computer Science Dept.

Lecture topics

- context free grammars
- compiler intro (copy p. 13 from compiler book)
- types, values, and binding
- Flex and tokenizing
- Bison and parsing
- regular expressions
- Group work
  - come up with regular expressions for certain items
  - Create the tokens and syntax trees for code from beers
  - Create the grammars needed to represent the tokens