CS 490 Compilers: Notes

Fall 2018

Professor Douglas Comer

Ian Renfro

Week 1

Introduction: Will be focusing on Parsing and Lexical Analysis instead of code optimization

Week 2

Parsing: The Fundamental Idea

- Take a Program, i.e. just a text file
- Figure out what it means

Translation: the movement between two languages
Compilation: The translation from a high level language to machine code

Lexical Analysis:

How to build a Lexical Analyzer:
During initialization get the first character. When a token is requested then look at the saved character and decide where to go next once you have read the following character. Repeat until you have an identified token or there is an error.

How should we handle Keywords?
We can handle keywords either in the parser or in the lexical analyzer. If done in the parser an extra "if" statement is needed to identify if the token is of the type "identifier/variable" then it will call a keyword matcher, this is bad because this set of lines is run for every token which will add up. If done in the lexical analyzer then we will know the type of the token and there will be no need to check what the type of the token is which will save time during execution. The ladder is the preferred method for us to use.

There is no need for dumb lookahead in the lexical analyzer

Hierarchy:

- Parser
  - The purpose of the Parser is to retrieve tokens and use the keyword matcher
- Keyword Matcher("if", "else", "while", "program")
  - Returns a more specific token than just an identifier token
  - Usually embedded in the lexical analyzer
- Lexical Analyzer
  - Retrieves characters from the program and creates tokens for the Parser

When creating the lexical analyzer, always save the last character read. A good thought is to have an UNUSED token to give errors

Week 3

Topic: How to implement a Lexical Analyzer?

Every call to the lexical analyzer returns a token, with the EOF token being returned at the end of the file. Please spend some time thinking about how you would design such a function.
The best way to implement a lexical analyzer is by using a Finite State Machine. FSMs are a good way to build any function that requires states, but you need to be careful because implementing them can be tricky. The finite state machine will give an action and a transition for each character that is encountered. We will implement it in such a way that we will always be looking at the next character so that we know where we need to go when we come back.

Keywords:

Should we include keywords in the finite state machine? This creates a big burden on the finite state machine and a much better implementation would be to have the get token function or parser take care of that. This would save a lot of overhead in the finite state machine as well as be easier in the parser since we can just check the correct subset of tokens for keywords.

Overall Structure:

The Parser will call the get token function, which in turn calls the lexical analyzer. The keyword check should be done in get token so that we can easily check tokens only if they are identifiers.

Finite State Machine - An Efficient Implementation

Never do in code what you can do in data!

- Create the Finite State Machine as a 2D Array of structs
- One column per character (256 + 1 for EOF) and one row per state in the Finite State Machine
- Initialize everything as an error then start making the transitions

Week 4

Review about Lexical Analysis

For example look at the case of multi-line comments, i.e. /*.....*/. How should this work? In one row of the array all of the characters lead to the same state to eat up the input until there is a "*" followed by a "/" then we go back to the initial state. For next week try to create the finite state machine in code.

Things to remember when creating the Finite State Machine

- Do not use for loops with "c='a'; c<'*';c++". Initialize the individual characters so as to be more clear with what the code is doing.
- Start small to make sure it works
- Add small number of states at a time and make sure that they work correctly
- Another option is to write a script for creating the code, this is a good route it takes longer but ends up being more verbose.

No matter what you choose, make sure to write it down.

Documentation is not just for the next guy that comes along, it is for the future you. It is there so that when you look back after a few months or years, you still know what it is doing and how you got there. Make sure to include all the assumptions that you are making.

Week 5

This week Comer went over his solution to the Lexical Analyzer

He had a specification that was a description of the finite state machine which was read in by an awk script. The script generates two files, the first of which is fsm-defs. This file contains the state names in C-style define statements as well as the total number of states. The second file that is generated is fsm-code. This file contains all of the initialization code for the finite state machine.

Important Functions and Actions:

- gettok
  - This function gets the next token and places it in the global variable so that the lexical analyzer and parser can use it
- errmsg
  - uses a global line number and character number to produce nice error messages when something unexpected is encountered
  - this functions also exits so there is no need to exit from the caller
- getch
  - calls the UNIX getchar
  - checks to see if the read char was an EOF and in that case returns the defined EOF char instead of -1
- FSM_ACT_SRET
  - Save and return, mostly used for single character tokens
- FSM_ACT_ARET
  - Append and return, mostly used for two character tokens, or tokens that we know are finished at the current character
- fsmentry struct
  - this is the struct that is inside all the indices of the finite state machine
  - Contains the following information
    - the next state to transition to
    - the action to perform
    - the type of token that will be returned

There is something wrong with everything we are doing! In lexinit we run pages and pages of code in order to create the finite state machine. This is a terrible approach. Instead we should write a script that produces an array initialization of the finite state machine instead of a code

Week 5 continued on next page...

based initialization. By doing this we can pre-process all of the finite state machine initialization at compile time and not have to run all of the code each time lexinit is called.

String Constants:

Some questions to think about on the topic of string constants are: "How do we store these string constants?", "Are they going to be handled differently than our other tokens?". We need to think about how long we want our string constants to be because they sometimes need to be significantly longer. We also need to think about how we will handle the escaped characters. While it can be better to handle these earlier we need to keep these things in mind:

- Error messages in the parser can lead to actual whitespace characters being written
- Debugging becomes very excruciating when trying to figure out all the characters that are in the string
- At the end of the day it is a design choice and it is up to us
  - Comer decided to store literal backslashes and he had it in such a way that the lexical analyzer would check if it is a valid string

Week 6

Parsing

We are starting on parsing now. The lexical analyzers should not only be done but they should also be correct. We will be building on top of them so it is important for them to be correct.

Parsers vs Lexical Analyzers and what they should handle

Parsers should only deal with tokens and never with characters. Lexical Analyzers should only deal with characters.

Problems

There is a problem because we want to make users happy. Users do not operate in the world of tokens but rather in the world of characters. If we encounter an error then we want a line and character number for the user, but if the error is inside of a token then it is difficult. In short if the lexical analyzer encounters an error in getting tokens then it will be very accurate but if a parsers finds an error then the error will be off by one token. This is because it takes us one token to know if we encountered something undefined or not.

Parsers!!

Things to think about are: "Is the set of tokens valid?", "If so then pass it off to the code generator".
How should we write a compiler?
Two thought processes:

- Bottom-Up (most real compilers - also claim that they can build faster compilers)
  - Starts by gathering tokens in a pile
  - Once it recognizes what a set of tokens is then it puts it aside
  - Shift-Reduce
    - Shift tokens in until you get the tail end
    - Reduce it to something else that you can identify
  - Generally when a language has a weird syntax, the parser is behind it
- Top-down (Us)
  - Parses from left to right
  - Need to make sure that each statement in unambiguous from the start, this is why many languages have each statement start with a keyword

Grammars are at the center of Parsers

Many people will say that "Once you write a grammar, then the compiler is trivial!" This is not true. It is possible, many beginners do it, to write grammars that are inefficient or very difficult to translate into code. You need to learn how to use grammar before you can make a grammar that is easily translated into code.

Week 6 continued on next page...

Recognizers:

Given a sequence of tokens, i.e. a program, a recognizer will announce if it is a valid or invalid sequence.
A good example of a recognizer is a Turing Machine. Mathematicians created Push-Down-Automata where if you are given a grammar then you can generate a PDA for the grammar and vice-versa.

Ad Hoc Parsers:

these were some of the first parsers and were essentially glorified if-else statements which was truly terrible.

Donald Knuth started working on a practical parser that we could remove from context-free grammar. He was able to do this by constricting context-free grammar and using a subset which would allow him to easily generate parsers.

- LL(K), LL1 (Us), LL2, ...
- LR(K), LR1, LR2, LR3...
- We are using LL1 (Top-Down with K=1)
- K is the number of things that you need to lookahead at to make sense of what you are currently seeing
  - As K increases the more powerful the grammar becomes but the harder it is to write a compiler for it. It will also take the compiler more time to run

Bell Labs came along next and thought that Knuth's definition was still to general and wanted to restrict it more. They created LA and LR.

What does a grammar look like?

- upper case: Nonterminal
- lower case: terminal

Classic context-free grammar problem is how to generate well nested parenthesis, such as: () or () () or ((())) or ((((()))))) (()) ((()))

- P -> ()
- P -> P P
- P -> (P)

Grammars use recursion to generate complicated languages.

Week 7

Grammars - The Practical Side of Things

Some grammar rules:
"OR": LEFT -> X — Y
"One or more": LEFT -> [stuff]+
"Zero or more": LEFT -> [stuff]*
We can use these tools to create a grammar for f9.
program -> pgm "EOF"
pgm -> "program" "{" decls stmts "}"
decls -> ["type" (LEXINT), "name" ["," "name"]* ";"]*
stmts -> stmt [stmt]*
stmt -> "if" if-body | "while" while-body | ...

In f9 there are 4 main types of statements:

- Assignment
- Conditional
- Iterative
- Function Call

How to turn a grammar into code:

Each non-terminal corresponds to a function.

```
void program (void) {
        pgm();
        if (nexttok.toktyp != LEX_EOF) {
                error( expected EOF );
                exit(1);
        }
        return;
}

void pgm() {
        if (nexttok.tok_typ != LEXPROG) {
                error( expecting    program     );
        }
        gettoken();
        if (nexttok.tok_typ != LEXLBR) {
                error( expected left brace );
        }
        gettoken();
```

```
        decls();
        stmts();
        if (nexttok.tok_typ != LEXRBR) {
                error( expected right brace );
        }
        gettoken();
        return;
}

// EXAMPLE WITH STARS:
// decls -> [type name ; ] *
void decls(void) {
        // Body of the while loop encapsulates (type name ;)
        //          with the * being the while
        while(nexttok.tok_typ == LEX_TYPE) {
                gettok();
                if (nexttok.tok_typ != LEX_IDENTIFIER) {
                        error( expecting identifier );
                }
                // Found an identifier
                // TODO: Save this id in a symbol table somewhere
                gettok();
                // Expect to find a semicolon!
                if (nexttok.tok_typ != LEX_SEMICOLON) {
                        error( expecting semicolon );
                }
                gettok();
        }/* end while (AKA the * in our expression) */
        return;
}
```

Do not write ambiguous right hand sides because they are hard to translate into code.

When writing grammars be sure to think about how I will be able to translate this into code. This is directly tied to which each statement starts with a keyword. It makes it easier to make progress at each step of the grammar. If we can write compilers that are LL1 then they are super fast because they only have to look at one token ahead of where they are.

Week 8

Grammar

```
pgm     ->  "program" "{" decls stmts "}"

stmts   ->  stmt [stmt]*
```

Another approach is to put the ending curly brace at the end of stmts. For example:

```
pgm     ->  "program" "{" decls stmts

stmts   ->  stmt [stmt]* "}"
```

Why would it be good to do this:

```
stmts   ->  stmt [stmt]* "}"
```

This allows you to identify when a set of statements is ending, i.e. you can iterate while the next token is not a "}". Otherwise, you need to iterate while the next token is equal to "if | while | func | identifier". Putting the "}" in the stmts makes the while loop a lot easier. Do not build redundant code. Comer put the "}" in pgm but he left the while loop as while nexttoken is not equal to "}" because every set of statements ends with a "}" anyways.

Compiler Architecture

There are two ways in which to approach building a compiler.

1. Build a parse tree of the entire program
    - Generate code by walking the tree after the parsing has completed.
    - This allows for a lot of optimization
        - Can walk through the tree and assign variables to specific registers
        - Can replace common sub expressions with their values
        - Constant sub expressions are computed at compile time so there no need to generate code for that in our parse tree
    - An optimizer can also identify if it will run out of registers for the variables and decide which variables or sub expressions should be loaded into memory versus registers
2. Generate code "on the fly", i.e. while parsing
    - Less opportunity for optimization because you cannot look at the entire parse tree
    - Ease of programming, it is much easier to generate the code as you parse as opposed to when you are walking the tree
    - We will be using this method but most compilers use method 1 and spend enormous amounts of time on optimization

Parsing Expressions

```
LOGOP = || or &&
CMPOP = < or > or <= or >= or == or !=
ADDOP = + or -
MULOP = * or /

expr    ->  comp [ LOGOP comp ]*

comp    ->  addn [ CMPOP addn ]*

addn    ->  muln [ ADDOP muln ]*

muln    ->  term [ MULOP term ]*

term    ->  NAME | INTEGER | '(' expr ')'
```

Top Down Parser $\equiv$ LL(1) $\equiv$ Recursive Descent
The non-terminal *term* is recursive! If a '(' is found in *term* then we call expr. This is beautiful because in this order captures precedence i.e. for *addn* we call LHS multiply then we find a "+" and we call RHS multiply. This is really cool, no need to bind precedence into the parser, it does it automatically when parsing! The precedence is encoded into the grammar!!
Note: Read is special because it is not part of expressions. When it happens it can only appear on the RHS of the assignment.

Week 9

Parsing Parsing Expressions should be done by now. We are going to take a look at the parsing of control structures. For review this is *parsestmt*

```
void parsestmt(void) {

    char fname[MAXTOK];
    switch(nexttok.tok_type) {
        case LEXIF:
            gettoken(); //move past if stmt
            parseIF();
            break;
        case LEXWHILE:
            gettoken() //move past while stmt
            parseWHILE()
            break;
        case LEXNAME:
            // do not remove the name before calling parseASSIGN
            parseASSIGN();
            break;
        case LEXFCN:
            strcpy(fname, nexttok.tok_type);
            parseFCN(fname);
            break;
        case LEXSEMI:
            errormsg("Did not expect ';'...")
        default:
            errmsg("Expecting a statment"); // could also print token information
    }
}

void parseIF(void) {
    // can put this if statment into its own method i.e. match(LEXLP, "error messag
    if(nexttok.tok_type != LEXLP) {
        errmsg("Expecting left paren");
    }
    parseEXPR();
    match(LEXRP, "Expecting right paren");
    match(LEXLB, "Expecting left brace");
    parseSTMTS(); //parses statments until the next right brace
    match(LEXRB, "Expecting right brace");
    if(nexttok.tok_type == LEXELSE) {
        gettoken();
```

```
        match(LEXLB, "Expecting left brace");
        parseSTMTS();
        match(LEXRB, "Expecting right brace");
    }
}

void parseWHILE(void) {
    match(LEXLP, "Expecting left paren");
    parseEXPR();
    match(LEXRP, "Expecting right paren");
    match(LEXLB, "Expecting left brace");
    parseSTMTS(); //parses statments until the next right brace
    match(LEXRB, "Expecting right brace");
}
```

Note: we could create something like parsebody where we parse '{' stmts '}', however this might not be best because it makes it more complicated to read.

Parsing Assignments

We use a symbol table to store all of the variables in the program. A declaration is an insert into the table and a reference is a lookup in the table.

```
Scheme
name    |       type        |       size
-----------------------------------------
xyz     |       int         |       0 => scalar vs aggregate i.e. arrays
```

One thing that is not considered in the scheme is scope. Most compiler's symbol tables are more complicated because of the storage of scope. Here is an example of what that might look like.

```
scope id     |   name    |          type      |       size
-------------------------------------------------------------------------------------
unique id    |   xyz     |           int      |       0 => scalar
                                                      vs aggregate i.e. arrays
```

A global scope has id 0. Also your entries in the symbol table will probably look similar to the following.

```
struct sym_entry {
    char sym_name[MAXNAME | MAXTOK]; // Should pick either maxname or maxtok here
    int sym_type; //defines for these
    int size;
}

struct sym_entry symtab[MAXSYMS];
int nsyms = 0;
```

Week 10

Functions

Note: Next week we will start looking at code generation.
A few functions were left out from last time. We classified them as LEXFCN so that they both went to the same function. When a function occurs you have to call parse function.

```
parseFCN(char* fname) {
    match(LEXLP, "Expecting '('");
    if(!strcmp(fname, "exit")) {
        // handle exit
        parseEXPR()
    } else {
        // handle print
        // see if list is empty
        if(nexttok.toktyp == LEXRP) {
            errmsg("empty list")
        }
        moreargs = 1;
        argscount = 0;
        while(moreargs > 0) {
            argcount++;
            if(argscount >= MAXARGS) {
                errmsg("too many args, quitting");
            }
            if(nexttok.toktyp == LEXSTR) {
                gettoken();
            } else {
                parseEXPR();
            }
            if(nexttok.toktyp == LEXCOMMA) {
                gettoken
            } else {
                moreargs = 0;
            }
        }
    }
    match(LEXRP, "Expected ')'");
    return;
}
```

This does not handle things like double commas or a comma after the last argument. Parseing Assignments is another function that we have not done yet.

```
void parseASSIGN() {
```

```
    // current token is the identifier
    int symindex;
    char temptok[MAXTOK];

    symindex = symlookup(nexttok.tok_str);
    strcpy(temptok, nexttok.tok_str);
    gettoken();
    match(LEXANSOP, "Expecting assignment operator");
    gettoken();
    if(nextok.tok_typ == LEXREAD) {
        parseREAD();
    } else {
        parseEXPR();
        match(LEXSEMI, "Expecting semicolon");
    }
}
```

The last function that we are looking at this week is parsing read.

```
void parseREAD() {
    int varindex;
    int morevars;
    gettoken();
    match(LEXLP, "Expecting '('");
    morevars = 2;
    int argcount = 0;
    while(morevars > 0) {
        argcount++;
        if(nexttok.tok_typ != LEXNAME) {
            errmsg("Expecting var name");
        }
        varindex = symlookup(nexttok.tok_str);
        gettoken();
        if(nexttok.tok_typ == LEXCOMMA) {
            gettoken();
        } else {
            morevars = 0;
        }
    }
    match(LEXRP, "Expecting ')'");
    return;
}
```

Week 11

Code Generation

There are two ways to generate code. You either generate the code while you are parsing or you create a parse tree and walk the tree afterwards. We will be generating code while parsing, even though creating the parse tree and then generating the code while walking through it is what most compilers do. The parser will generate either assembly or high level code, then it will be run through an assembler, such as gas or gcc, which then produces the object code which is passed on to the linker, this is where outside libraries get pulled in as well, then finally an executable is made. We can pick which on that we want to do but Comer suggests generating C code because it will be easier and more straightforward than generating assembly code. He passed out a copy of his parser to everyone. He embedded the code generation into his parser by just using printf statements. This is a very elegant way of approaching this because we are parsing left to right and that is also the way that most code is written.

Assembly Code Generation

Assembly language generation is interesting because you need to generate different assembly language for each system. Comer says its very hard because you need to learn all of the commands for all languages and then you need to learn what hardware does what instructions better so that you can provide better optimization. In the f9cc that comer provides you can run *f9cc -S x.f9* which will output x.S, this is the assembly language version of x.f9 so that you can see what code needs to be generated.
We need to think about changes between generating C code and Assembly code, also a not on instructions the asm instruction *branch* is for things that are close by because branch will use a number of lines forward or backward to go to whereas *jump* takes a memory address. For example how would we generate a while loop in assembly? We would want a label generator, otherwise we will be stuck trying to figure out what the next label is and such. A while loop would look something like the following.

```
;       while(expr) {
;           stmts
;       }
; depending on the assembler you may need a directive after the colon
L0001:
    # asm for expr
    bzero L0002   ; if the expr does not hold
    ; nothing for LEXLB
    ; asm for stmts
    br L0001
L0002:
```

Essentially all control statements will need labels.