Any computable thing can be described by this machine:

Diagram

Description automatically generated

For any computable problem, you can draw a state machine like below:

Diagram

Description automatically generated

Not very effective for what we are doing today.

**VON NEUMANN MODEL**

Diagram

Description automatically generated

2 types of machines:

* Fixed program computer
  + Sth that does 1 function like “calculator”. You don’t see these things much today bc stored program computers are inexpensive these days.
* Stored program computer
  + Carry out many different tasks, you can store the program.

Diagram

Description automatically generated

You can do 1 thing at a time. I’m gonna read from memory, write to memory, arithmetic operation or logic operation whatever. Writing to and reading from memory is bottleneck for this model.

**VON NEUMANN PROGRAMMING LANGUAGES**

Programming languages at the high level abstraction are based on von neumann architectures are called von neumann programming languages.

PLs that are high level abstract isomorphic copies of von neumann architectures

The isomorphism between von neumann programming languages and architectures is in the following manner:

* program variables ⬄ computer storage cells
* control statements ⬄ computer test-and-jump instructions
* assignment statements ⬄ fetching, storing instructions
* expressions ⬄ memory reference and arithmetic instructions

C has one-to-one matching for these things.

In purely functional PLs there are no assignment statements.

**SYNTAX AND SEMANTICS**

Syntax: The symbols used to write a program

* With 200 characters that I wrote a program, I can write different versions of a program. There are 256 characters in the ASCII table but language may not have all those characters in its syntax. Also not every combination of letters is also not acceptable for a language. For example “for” is acceptable but “12xxx” is not acceptable by C as a word. Syntax defines this. Syntax limits the possibilities that I can write a program with 200 characters with not different versions.

Semantics: The actions that occur when a program is executed

* For example “five” in natural language can mean many different things as money, hour, etc. so this is the semantic.
* Semantic will allow us to look at or analyze or tell what the program is gonna do when it is executed, what those symbols are gonna cause the machine to do.
* For example “1+2” and “(+ 1 2)” mean same things in different ways, addition in infix and prefix editions. Combination of symbols may convey different things and I need to define that through semantics.
* We are gonna analyize syntax with divide and conquer.

Programming language implementation: Syntax 🡪 Semantics

* Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

**INTERPRETER VS. COMPILER**

Diagram

Description automatically generated

Interpreter is actually a program.

Typical Compiler

Diagram

Description automatically generated with medium confidence

Lexical Analyzer (Tokenization):

* Allowed words, punctuations, characters, sequences of characters

Syntax Analyzer:

* Allowed sequence of things are not necessarily just comes together in an arbitrary way, there is a limitation on how things can be
* Lexical analysis can say “123” is a number and “145” is another number and “xy” is keyword. Then syntax analysis will say the keyword and 2 numbers can come together in this way or that way.

Semantic Analyzer:

* I know what I have is a valid instance of a language in terms of the symbols coming together but are they gonna do sth right? For example “int x;” I said before “int” “x” and “;” are okay. As a sequence of actions is this good enough for me? What does this mean semantically and is it acceptable to me, can I check it? For example “x int;” is not semantically correct.
* This analysis can do certain things in terms of looking at the intended action if it is doable or not. In the context of what the language is telling you.
* For example C doesn’t care about boundaries of my iteration but some other languages can say I care about the boundaries and my semantic analysis says I will check the boundaries of your operation and if it cannot be done by the machine I will give you a semantic error.

Intermediate Code Generator:

* With parse tree we build the code very easily.

Code Optimizer:

* Generated code may not be the best code that you want to execute bc that analysis may force you to do with a parse tree inefficient code. It may have repetitions.

Code Generator:

* Once the code optimizer is done, you generate the code.

**SYNTAX**

Syntax of a programming language is a precise description of all grammatically correct programs

* Precise formal syntax was first used in ALGOL 60

Lexical Syntax: Basic symbols (names, values, operators, etc.). Get lexical analysis to generate symbols or tokens.

Concrete Syntax: Rules for writing expressions, statements, programs. Rules that tokens come together. Not any token come together in any combination.

Abstract syntax: Internal representation of expressions and statements, capturing their “meaning” (i.e., semantics)

I want to have a tool, a language (meta language) to describe the syntax of a programming language.

Grammars

A way to express the structure of a language formally.

A meta-language is a language used to define other languages.

A grammar is a meta-language used to define the syntax of a language. It consists of:

1. Finite set of terminal symbols
   1. Terminal symbols come from earlier analysis
      1. If this is a concrete syntax, terminal symbols will be the finite set of tokens that we get from lexical analysis.
      2. If this is a lexical analysis, terminal symbols will be the characters in ASCII table (or alphabetical and numberic).
2. Finite set of non-terminal symbols
   1. Symbols that you will generate for this grammar alone.
   2. Sort of placeholder for intermediate representations.
3. Finite set of production rules
   1. Production rule says I can take a few terminal symbols, put them together and build a non-terminal symbol. A few of these can be put together, build another one and so forth.
4. Start symbol
   * Says where we are gonna start or end our production rules.

Above 4 matters are Backus-Naur Form (BNF). Specific form of defining a grammar.

Terminal symbols: The base tokens of the language. For a programming language:

* keywords
* operators and other symbols
* the characters that can be used in identifiers, numbers, or other program elements

Nonterminal symbols: Used to represent pieces of the structure of the language. For a grammar of English, these would include noun, verb, sentence, etc. For a PL, statements, condition, subroutine, etc.

Elements of BNF:

* Terminals are simply written out: while
* Nonterminals are enclosed in angle brackets: <statement>
* Productions are in the form:
  + <nonterminal> ::= <sequence of terminals or nonterminals>
  + <sentence> ::= <noun phrase><verb phrase>
* We can use | to represent or
* <digit> ::= 0|1|2|3|4|5|6|7|8|9
* <integer> ::= <digit> | <digit><integer>
* <floating point> ::= <integer>.<integer>

Graphical user interface, application

Description automatically generated

*Language = (possibly infinite) set of all sequences of terminal symbols that can be derived by applying production rules starting from the start symbol.*

Example: Decimal Numbers

Define a grammar for decimal numbers. Not a pl, just for lexical analysis.

Decimal number is sequence of terminal symbols.

* Terminal symbols are digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
* + and – also could be my terminal symbols. We think unsigned decimal integers.
* Non-terminal symbols: Digit, Integer
* Production Rules:
  + Integer 🡪 Digit | Integer Digit
  + Digit 🡪 0|1|2|3|4|5|6|7|8|9
* Non-terminal symbols 🡪 combination of terminal and non-terminal symbols (PRODUCTION RULE)
* Start symbol: Integer

Let’s say I want to generate 313:

* Start with start symbol: Integer
* Integer can go to digit or integer digit.
  + If integer goes to digit, digit goes to 0, 1,…, 9. Not my number.
  + Integer goes to Integer Digit.
* I 🡪 I D ----------> I 3
* I 🡪 I D -----------> I 1
* I 🡪 D -----------> 3

We do here rightmost derivation. At each step, the rightmost non-terminal is replaced. I could also start with leftmost symbol and go that way. These are choices. Also this is not an algorithm.

If someones program comes to my compiler and I have the BNF, my algorithm has to test the program if it is following my BNF.

We cannot just have any BNF, we are gonna restrict that to be able to enforce what the BNF is intending.

**Diagram

Description automatically generatedCHOMSKY HIERARCHY**

Set of grammar hierarchy.

Regular is the easiest, recursively enumerable is the most difficult.

Difficulty comes with expressive power.

We are in a ideal scenario. We are saying let’s build a grammar to describe everything for a pl.

Regular Grammars:

* Regular expressions, finite-state automata
* Used to define lexical structure of the language

Context-free Grammars:

* Non-deterministic pushdown automata
* Used to define concrete syntax of the language
* BNF is sufficient up to here

Context-sensitive Grammars:

* Unrestricted grammars
* Recursively enumerable languages, Turing machines
* We are gonna use subset of this grammars to describe semantics

If I want to do type checking, in those type of cases I need to remember a lot of things from much earlier of the structures. That would render our context-free grammars not sufficient for those types of descriptions.

**REGULAR GRAMMARS**

Generate regular languages

Left regular grammar

* All production rules have the form
  + A 🡪 or A 🡪 B or A 🡪 (empty)
* Here A, B are non-terminal symbols; , are terminal symbols

Right regular grammar

* A 🡪 or A 🡪 B or A 🡪

Can be explained by regular expressions (compact notations for regular grammars)

Example: grammar of decimal integers, not the example we have done (It has non terminal to nonterminal-nonterminal : I 🡪 ID)

* It has to be sth like this:
  + I🡪D
  + I🡪D0
  + I🡪D1
  + …
* This is now left regular.

**Lexical Analysis:**

Source code = long string of ASCII characters

Lexical analyzer (lexical analysis can be easily done using regular grammars/language) splits source code into tokens

* Token = sequence of characters (symbolic name) representing a single terminal symbol for the next part of the analysis (the concrete syntax)

Identifiers: myVariable …  
Literals: 123 5.67 true …  
Keywords: char sizeof …  
Operators: + - \* / …  
Punctuation: ; , } { …  
Discard whitespace and comments



GRAMMAR: $Id 🡪 $Letter $Id / $Id 🡪 ------> $Letter 🡪 a|…|z|A|…|Z, This is not a regular grammar. We have nt🡪nt nt (nt: non terminal). Id is start symbol. We have 26+26+1 () terminal symbols and 2 non-terminal symbols.

Convention: Use $... (… is alphabetical) to describe non-terminals. Use alphabetical things to describe terminals. Underline the start symbol.

Table

Description automatically generated with low confidence

0-9 🡪 Anything between first and second character in ASCII table

**Examples of Tokens in C**

Lexical analyzer usually represents each token by a unique integer code, some of the tokens like operators. How do I tokenize them? Whenever I see a “+”, generate a token and return(PLUS). PLUS is some sort of a name for a token:

* “+” {return(PLUS);} //PLUS = 401
* “-“ {return(MINUS);} //MINUS = 402
* “\*” {return(MULT);} //MULT = 403
* “/” {return(DIV);} //DIV = 404

Some tokens require regular expressions

* [a-zA-Z\_][a-zA-Z0-9\_]\* {return(ID);} //identifier
* I need a letter that can be from a to z, A to Z or \_. I need one of them.
* Then followed by 0 or more of a to z, A to Z, 0 to 9 or \_
* [1-9][0-9]\* {return(DECIMALINT);}
  + There is no way this generates 0. So it means 0 is not a decimal integer or I have to deal with that.
* 0[0-7]\* {return(OCTALINT);}
* (0x|0X)[0-9a-fA-F]+ {return(HEXINT);}

**Reserved Keywords in C**

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, wchar\_t, while

C++ added a bunc: bool, catch, class, dynamic\_cast, inline, private, protected, public, static\_cast, template, this, virtual and others

Each keyword is mapped to its own token name

**Automatic Scanner Generation**

Lexer or scanner recognizes and separates lexical tokens

* Parser usually calls lexer when it’s ready to process the next symbol (lexer remembers where it left off)

Scanner code is usually generated automatically

* Input: lexical definition (e.g., regular expressions, regular grammar)
* Output: code implementing the scanner
* Typically, this is a deterministic finite automaton (DFA)
* Examples: Lex, Flex (C and C++), JLex (Java)

**Finite State Automata**

Set of states

* Usually represented as graph nodes, kind of non-terminals

Input alphabet + unique “end of program” symbol

Inputs are terminal symbols (ASCII characters in ID case)

State transition function

* Usually represented as directed graph edges (arcs)
* Automaton is deterministic (bc for a given input there is only 1 arc) if, for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol 🡪 uniqueness of computation

Unique start state

One or more final (accepting) states

**DFA for C Identifiers**

Diagram

Description automatically generated

by default any unmapped input (other than a-z or A-Z or \_)🡪 error

actually this has a fault, you can’t have \_ \_ \_ as an identifier in C.

Compiler or Interpreter will take your input as set of ASCII characters.

input:

#include <stdio.h>

void main(){

char c = ‘a’;

c++;

c = ‘b’;

}

Our task is to define some algorithm to check if this is indeed an instance of the , programming language that we have designed.

We don’t look at the whole thing and say this is a valid C program or not. This is not possible algorithmically. We divide-and-conquer the problems.

Preprocessing simply takes out comments and white spaces.

Compiler do this in several phases.

First step is lexical analysis. Compiler will work on my input and generate a set of tokens for me. Is combination of letters is right? We check this in lexical analysis. I don’t wanna look at the characters, I wanna look at the combination of characters that make sense. I have curly braces, semicolons, parantheses, smaller/greater signs, equality sign, ++, words like include, void, main, char. If program is valid lexical analysis says you have organized your letters in pieces. Look at the input, we have #, I know that # sign is a token for me. I know bc that is what I defined. This is token for me for my syntax, for my later purposes. then we have “include” which is a keyword - token. <, >, void, main are all tokens for me. Lexer says you got a void token etc. Some others are not easy to catch. For example a variable name varies. For those, I use my regular grammar/expressions. Lex format is another kind of a grammar for define the lexical analysis.

Beginnig of the file (LEX code), I have all my lexical tokens possible in this language that I want to generate. I am gonna define them and give them a unique id:

PLUS 401

Then my rules come.

‘’ + ‘’ {return (PLUS);}  
 ….

This rule says whenever you see a text with plus sign, return PLUS.

Output of this program is gonna be bunch of tokens. So for our program:

* SHARP 503
* INCLUDE 408
* LESSTHAN 502
* …

LEX will generate these matches.

Some tokens require regular expressions

* [a-zA-Z\_][a-zA-Z0-9\_]\* {return(ID);} -------> code i wrote for id token

There will be 4 places in our code that identifier happens:

1. stdio
2. h
3. main (function name, if it is not a keyword)
4. c (variable name)

Now Lexer is gonna generate me these and whatever the corresponding number is:

* SHARP
* INCLUDE
* LESSTHAN
* ID
* DOT
* ID
* GREATERTHAN
* VOID
* ID
* OPENPARANTHESES
* CLOSEPARANTHESES
* OPENCURLYBRACE
* …

Some of them are gonna be fixed keywords (‘’+’’ PLUS), some of them are gonna be defined by our regular expressions. For integers and characters you can define regular expressions.

You can treat ‘a’ as bunch of tokens (OPENQUOTE – SOMETHING – CLOSECODE) or you can have a particular grammar to capture these character literals. Regular expression (to accept and tokenizes ‘a’, ‘b’ … etc. as a character) for character definition (letters, numbers and all punctuations):

* [‘][a-zA-Z0-9$/\...][‘]

Regular epression that catches string literal without escape character:

* [“][a-zA-Z0-9$/\…]\*[“]

This is a simple regular expression to catch instances of string literals.

My final program for automatic scanner is gonna be like this. I am gonna have a portion of program (file that i write to describe my lexical syntax of my program).

* #Define All The Tokens
* PLUS = 5
* …
* STRING = 600
* INT = 601
* IDENTIFIER = 702

We do this one to be able to match them later for our purposes for syntax analysis. Then comes the rules:

* ‘’+‘’ {return (PLUS);} ----------> very simple DFA. Checks the input, if the input is +, it will execute it, it will return and go to the next line.
* “-“ {return(MINUS);} //MINUS = 402
* …
* [“][a-zA-Z0-9$/\…]\*[“] {return(STRING);}
* [-]\*[0-9]+ {return(INT);}

Order is important. I am asking my Lexer to check for first one (PLUS), if it is there do it, if not do next one, if not do next one…

You can associate a certain code for a rule and then associate the return part of it. Lexer is gonna say:

call (regex [-]\*[0-9]+ == success) return INT;

**Traversing a DFA**

Configuration = state + remaining input

New token for expressing an identifier for C where I don’t accept capital letters for the first one:

START SUCCESS

other

[a-z]



[A-Z]+

It is deterministic bc each arc consumes 1 unique entity. For a unique entity, I only have 1 arc from a state.

Configuration = state + remaining input

For example I have input (program) for this machine. I have done some of my input already. My input is:

… “azz = 5;}” ---> remaining of my program, remaining input

Currently I am in the start state.

This is called a configuration.

Move = traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it

Leftmost input symbol for our case: a

a will move me to final state. I could have stopped here. For this DFA, you can stop here and execute whatever happens after the stuff. Or you can say hey I can still accept new things. I can accept z, I can accept z. After this, I am done. If there is an other after this, I can say it is successful.

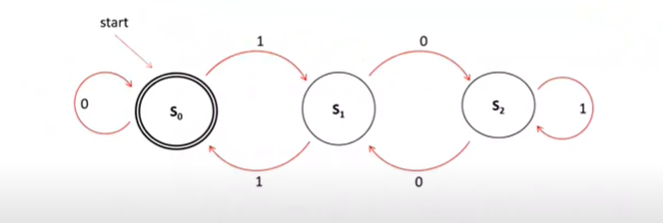
If no such arc, then:

* If no input and state is final, then accept
* Otherwise, error

Input is accepted if, starting with the start state, the automaton consumes all the input and halts in a final state.

Traversing a DFA – EXAMPLE

A DFA that accepts only binary numbers that are multiples of 3. The state S0 is both the start state and an accept (final) state.



1001 = 9 🡪 starts at start state and ends at final state so input is accepted. From right to left.

What if you get another digit other than 0 or 1? It is a error case. It is gonna say this is not a multiple of 3.

101 = 5 🡪 stops at S2 . No more input, i am not at the final state. DFA says fail.

101101 xyz … 🡪 Assume my lexical analysis asks sth like this. This is saying that I will only accept as tokens in this language it is, as far as this part of it is concerned, multiples of three in binary format. How do I know I reached the end? bc I have a whitespace, there is nothing to go any further than this one. I stopped there and say “did you accept it?” “no”, then this means that this particular rule is not gonna apply. So I go to the next rule, next rule… until I have no more rules available for me to apply (Your input is not a +, -, \*, /, ID, DECIMALINT…). If I don’t have any rules to apply program is gonna fail. It cannot generate a token that this language or lexer knows. For example if you have “12x = 12;” in your C code, none of your rules are gonna explain it. So it fails. It is gonna give lexical syntax error.

**CONTEXT-FREE GRAMMARS**

Assume lexer is done, I took my ASCII character mapped it to set of tokens. These tokens are tokens that I defined like a plus sign, minus sign, identifier, etc. This is part of the compiler.

Compilation has lexical analysis, concrete syntax, semantic, code generation, optimization, relocation, linking and so forth.

Context-free grammars are used to describe concrete syntax meaning however we are gonna interpret sequences of tokens as meaningful in terms of some sort of syntax, the way that they are organized.

Typically using BNF notation

Context-Free grammar has the same bnf we discussed. This grammars put some constraints on the structure of production rules. Production rules have the form A 🡪

A is a non-terminal symbol, is a string of terminal and non-terminal symbols

We don’t have any restrictions on the form of the string. You might have (T: terminal symbol, NT: non-terminal symbol) TTTT, NTNT, NTTNTT…

How am I gonna relate these tokens come out of the lexer? Are they in correct order? Are they defining things that I want to define?

For example, if i have an expression, if segment of this code “b + c” is gonna generate some tokens:

🡪 ID, PLUS, ID -----> tokens that this particular string generate

An expression ($Exp) is my non-terminal start symbol. Terminals are the tokens. These tokens come as numbers (401 for plus etc.). We are gonna represent them as names. How can I generate expressions? One way is this:

$Exp 🡪 ID $OP $Exp --------> OP: operator

$Exp 🡪 LIT $OP $Exp --------> LIT: literal

$Exp 🡪 ID | LIT

$OP 🡪 PLUS | MINUS | MULTIPLICATION | DIVISION

An expression in C is a literal - a number or an identifier – a variable or a function or a combination of these with some operators.

We are missing parantheses.

Adding two identifiers makes sense? Yes according to the rule we made. Later we ask can we add any type of these identifiers? That would be the semantic of the abstract syntax.

We are gonna also be worried about how they are gonna be meaningful in terms of semantic. b+c is syntactically good but semantically it may not be good if their types are different.

What type of an instance of concrete syntax can I generate? This is the question in context-free grammars. If someone is giving me some sentence, I am gonna check if this sentence can be generated by context-free grammar. To do that, we are gonna use concept of parse trees.

Parse tree = graphical representation of derivation using the production rule starting from the start symbol and I want to end up at the leaves.

* Each internal node = LHS of a production rule
  + Internal node must be a non-terminal symbol
* Children nodes = RHS of this production rule
* Each leaf node = terminal symbol (token) or “empty”

A 🡪 A x y B z ----------> small letters: terminals, capital letters: nonterminals

Context-Free Grammar

* set of terminal symbols
* set of non-terminal symbols
* set of production rules
* start state

Also you are given

* a series of tokens (order is important, so not a set, it is a list)
  + T1
  + T2
  + …
  + Tn

Given these, I am gonna have a parser that will generate a parse tree.

To generate a string of terminal symbols from a CFG:

* Begin with a string consisting of the start symbol;
* Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
* Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

Syntatic Correctness

Lexical analyzer produces a stream of tokens

Parser(syntactic analyzer) verifies that this token stream is syntactically correct by constructing a valid parse tree for the entire program

* Unique parse tree for each language construct
* Program = collection of parse trees rooted at the top by a special start symbol

Parser can be built automatically from the BNF description of the language’s CFG

* Example tools: yacc, Bison

Parser is gonna be verifying if the order of tokens that you received from the tokenization process is following your rules for that language.

You have to generate a unique tree for a specific code that is output of a tokenization or lexical analysis process. If you can generate more than one tree, there is ambiguity.

If I can generate unique tree, that means that I have one way of taking those tokens and mapping that to my concrete syntax in only one way.

Generating parse tree is automated like lexical analysis.

Unique parse tree is generated with an algorithm (that’s my goal, i need to have an algorithm). I can build that algorithm once and once you give me your concrete syntax in bnf form, I can ask my algorithm to generate the code for generating that parse tree for your language. This will be automated as we have done it for lexical analysis.

Let’s assume we have a single program file. Parsing assumes that you are able to get a series of tokens (parser doesn’t care what part of the program it is), and generate a single parse tree.

BNF is gonna be split into pieces. Your grammar is gonna be split into pieces. You will define a subgrammar that will actually check for expressions, statements, function definitions, if statements for example. If it is possible to generate a unique tree, each of these will give you subtree.

CFG For Floating Point Numbers

Problemin bir kısmını ele alacağız.

I am only interested in floating point numbers.

I am assuming that tokenization process gives me digits, most probably this will be part of the lexical analysis but I can handle this with CFG as well.

A picture containing chart

Description automatically generated

This is a simple definition of a concrete syntax of a floating point number. This is not that useful. You can do this with regular expressions and you can do this as part of your tokenization process. Nonetheless, we will try.

Sample parse tree for 3.14

Diagram

Description automatically generated

Root is start symbol. Leaves are terminals. Internal nodes are non terminals.

This is an instance of the language. This is not an algorithm.

Is this parse tree unique? Can I generate another parse tree? If I have a unique parse tree, I can interpret tree in a way. But if I have 2 ways of generating a parse tree, then I have to make sure that those two parse trees give me the same meaning.

Grammars and Derivations

Application of a sequence of rules

Text

Description automatically generated

This looks like a very simplified PASCAL language. For example this one is an instance (concrete syntax) of this language:

begin

A = A + B;

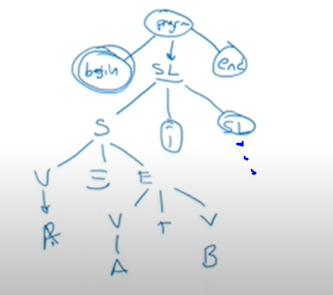
B = C - A;

end

<stmt\_list> may not go to anywhere. So we need:

* <stmt\_list> ::= <stmt> ; <stmt\_list> | <stmt>

program is my start symbol.



Each string in the derivation including <program> is called a sentential form

Or you can do it this way:

* <program> 🡪 begin <stmt\_list> end

🡪 begin <stmt> ; <stmt\_list> end

🡪 begin <var> = <expr> ; <stmt\_list> end

🡪 begin A = <expr> ; <stmt\_list> end

🡪 begin A = <var>+<var> ; <stmt\_list> end

🡪 begin A = B + <var>; <stmt\_list> end

🡪 begin A = B + C; <stmt\_list> end

🡪 begin A = B + C; <stmt> end

🡪 begin A = B + C; <var> = <expr> end

At some places I will have multiple choices. How am I gonna resolve those choices? Can I resolve them or can I leave them to the algorithm to explore and say no go back, explore no go back, … Problem is, if I don’t have an easy way to resolve which way to go. If you have average of 2 options to go from node to a child, and height is h:

* 2h different options you need to track

Code may have thousands of lines of code, generating thousands and thousands of tokens. Tree could be very very deep. You cannot hope to explore all these trees and your compiler responds with a unique parse tree.

One of the considerations for a programming language from the cost perspective is how fast can your compiler run.

Other question is “is this generate a unique tree?”.

Parse Trees

Grammars define hierarchical syntactic structure 🡪 parse trees

Text

Description automatically generated

If my input doesn’t have any equality sign, then I say this cannot be generated. This is not a nice algorithm bc I need to check the entire string for equality sign.

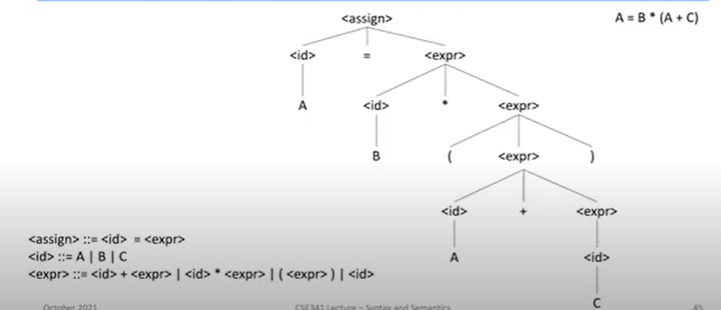
A picture containing shape

Description automatically generated

Computer cannot see directly ex is <id>\*<expr>.

<id>+<expr> is still a possibility for computer bc we have terminal +, also we have + in B\*(A+C). We parse the input and check for the + sign. If it is there, perhaps this (<id>+<expr>) is a good one.. We can only reject it in the next step. id goes to B, but there is no B+ thing is not in our input. So this will be a dead end.

Going further and coming back is costly, so we don’t want this.



Parsers

Parsers:

* Determine if the input program is syntactically correct in concrete syntax
* Produce a parse tree for the correct input program

Classify parsers by the order in which they build the parse tree:

* Top-down
* Bottom-up

Recursive Descent Parsing

Perform a depth-first search of the derivation tree for the input being parsed – top-down

Chart, radar chart

Description automatically generated

Example:

* <E> ::= x + <T>
* <T> ::= (<E>)
* <T> ::= x

LL Algorithms (Top-down)

Recursive descent – coded directly from BNF that is given to you

Parsing table – do not implement BNF rules

Both are versions of LL Algorithms

* Works on same subset of all CFGs
* 1st L: left to right scan of input
* 2nd L: leftmost derivation is generated (leftmost terminal is always first)

This algorithm is not gonna work for all BNFs and all CFGs that we can create. This is gonna work only on a subset of those.

Our input is sequence of tokens. Parse give me n tokens (T1 T2 T3 …Tn ). Left to right scanning of the input means that I am gonna be looking at my input sequence from left to right one at a time (maybe two at a time). We are gonna limit number of things that we can look ahead. In the simplest algorithm we are gonna be looking at only one entry, only given entry. In the more complex cases, multiple entries to do.

For example if we look at entire sequence at the same time and “there is a plus here so I can use this one” (like we did in first parse trees example), this is not gonna work in LL algorithm.

LR Parsing (Bottom-up)

LR Algorithms:

* L: left to right scan of input (order of which we parse the input)
* R: rightmost derivation is generated

Concrete vs. Abstract Syntax

Different languages have different concrete syntax for representing expressions, but expressions with common meaning have the same abstract syntax

* C: a+b\*c Forth: bc\*a+ (reverse Polish notation)

Graphical user interface, text, application

Description automatically generated

A + B \* C 🡪 With concrete syntax, this could be fine, it could be instance of your language

When it comes to meaning, the abstract syntax, this could be interpreted in 2 ways: + first or \* first. This is an ambiguity.

When we want to do the concrete syntax, we want to build a tree and we are gonna use that tree to define the abstract syntax as well. If I combine these 2 things, I cannot allow this type of ambiguities in my parse tree. This type of ambiguities come with abstract syntax.

Expression Notation

A picture containing graphical user interface

Description automatically generated

Mixed Expression Notation

A picture containing text

Description automatically generated

Expression Compilation Example

Timeline

Description automatically generated with medium confidence

Syntactic Ambiguity

Diagram

Description automatically generated with low confidence

Both trees have different meanings.

Parser may have more than 1 trees in ambiguity case. You have to handle that. Concrete syntax parser may pass it but abstract syntax may not be unique in some cases (for example this case).

Syntatic ambiguity happens when you can create 2 different trees using 2 different algorithms (leftmost and rightmost derivation algorithms).

Sentential Form

For a grammar G, with start symbol S, any derivation S 🡪 is called a sentential form:

* If contains only terminal symbols, is a sentence in L(G)
* If contains one or more non-terminals, it is just a sentential form (not a complete sentence but intermediate sentence in L(G))

A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence

A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence

E.g. (3 sentential forms):

* <P> 🡪 begin <stmt\_list> end
* <P> 🡪 begin A = <var> + <var>; <stmt\_list> end
* <P> 🡪 begin A = B + C; B = C end

Derivation Order

Leftmost derivation: the replaced non-terminal is always the left-most non terminal in the previous sentential form

A picture containing application

Description automatically generated

Rightmost derivation: other way around…

A close-up of a calculator

Description automatically generated with medium confidence

Derivation order has no effect on the language generated by grammar

Shift-Reduce Parsing

Stack implementation of shift-reduce parsing

Four possible action

* Shift – move the next input symbol to top of stack
* Reduce – replace the handle on the top of stack by non-terminal
* Accept – successful completion of parsing
* Error – syntax error discovered

Initial stack empty

End of input is empty

Note: LR parsers are more powerful than LL parsers

* they can see the entire RHS before choosing a production

BİR SONRAKİNE BAK, RHS OLACAK GİBİ DEĞİLSE REDUCE YAP.

The parser repeatedly matches the right-hand side (RHS) of a production against a substring in the current right-sentential form

At each match, it applies a reduction to build the parse tree:

* each reduction replaces the matched substring with the nonterminal on the left-hand side (LHS) of the production
* each reduction adds an internal node to the current parse tree
* the result is another right-sentential form

The final result is a rightmost derivation, in reverse

Handles

We are trying to find a substring of the current right-sentential form where:

* matches some production A :=
* reducing to A is one step in the reverse of a rightmost derivation

Call such a string a handle

A handle of a right-sentential form is a pair <𝐴 ⇒ 𝛽, k> where 𝐴 ⇒ 𝛽 is a production rule and k is the position in of 𝛽’s rightmost symbol.

If <𝐴 ⇒ 𝛽, k> is a handle, then replacing 𝛽 in at position k with A produces the previous right sentential form from which is derived in a rightmost derivation

S 🡪 A 🡪 𝛽 : change A with 𝛽

𝛽 include at least 1 terminal and non terminal symbols as a handle

Because is a right-sentential form, the substring to the right of handle contains only terminal symbols 🡪 the parser does not need to scan past the handle (only lookahead)

If the grammar is unambiguous, then every right sentential form of the grammar has exactly one handle

LR(1) Parsing

Diagram

Description automatically generated

REMEMBER

begin  
 X = Y + Z;  
 Y = X;  
end

LEXER  
(TOKENIZATION)

BEGIN ID EQ ID PLUS VALUE SC …

After the tokenization, I can do concrete syntax check.

$P 🡪 begin $LS end (LS: list of statements, $ means nonterminal)  
$LS 🡪 $S ; | $S ; $LS (S: statement)  
$S 🡪 ID = $E (E: expression)  
$E 🡪 ID | V | ID + $OP … (V: value, OP: operator)

Shift-Reduce Parsing (LR Algorithm)

Table

Description automatically generated with medium confidence

At step 2, we have looked to production rule and replace top of the stack with F.

At step 8, we can reduce E+T to E or we can shift \* to stack. Why we chose 2nd one? This is shift reduce conflict.

If there is start symbol at the end at our stack when there is no input left, it is successful.

Look ahead tells you that, for example in step 1, if you shift then you will have a id+ and id+ will not happen in any on the input at the grammar, therefore you have to move on. So algorithm reduce because of the look ahead.

Parse Tree for this example:

A picture containing chart

Description automatically generated

Conflicts

Generic shift-reduce strategy:

* If there is a handle on top of the stack, reduce
* Otherwise, shift

But what if there is a choice?

* If it is legal to shift or reduce, there is a
  + shift-reduce conflict
* If it is legal to reduce by two different production rule
  + there is a reduce-reduce conflict

Source of conflicts:

* Ambiguous grammars always cause conflicts
* So do many non-ambiguous grammars

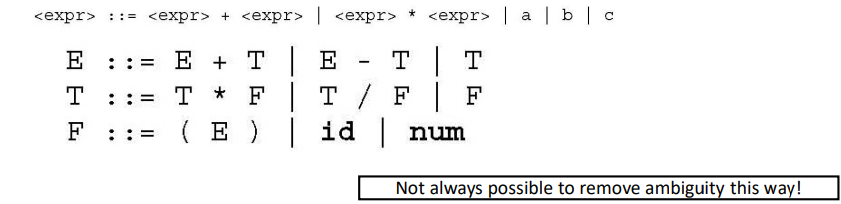
Conflict varsa bu durumda bunu yap vs. diyerek çözebilirsin veya CFG’ni değiştirirsin.

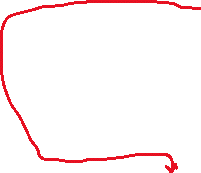
Removing Ambuguity

Define a distinct non-terminal symbol for each operator precedence level

Define RHS of production rule to enforce proper associativity

Extra non-terminal for smallest subexpressions





Our new grammar says this will take priority.

LR(1) 🡪 Here 1 is look ahead buffer. So you can look ahead 1 at a time.

So if you are currently processing id in “id + id \* 1”, you may be able to look ahead for the next input which is +.

This grammar is unambiguous:

Calendar

Description automatically generated

Left and Right Recursive Grammars

Diagram

Description automatically generated with medium confidence

Operator Associativity

Needed in the absence of parentheses for operators on both sides of an operand, e.g., … ^ a ^ …

Right associative: operators are grouped from right

5 ^ 4 ^ 3 ^ 2

⇒ 5^(4^(3^2))

Left-associative: operators are grouped from left

5 + 4 + 3 + 2 ⇒ ((5 + 4) + 3) + 2

left-associative: addition, subtraction, multiplication, and division

right-associative: exponentiation, assignment and conditional

Right Associative Exponentation:

A picture containing text

Description automatically generated

Dangling Else Ambiguity

Diagram

Description automatically generated

Solving the Dangling Else Ambiguity

Algol 60, C, C++: associate each else with closest if; use { … } or begin … end to override

Algol 68, Modula, Ada: use an explicit delimiter to end every conditional (e.g., if … endif)

Java: rewrite the grammar and restrict what can appear inside a nested if statement

* IfThenStmt → if ( Expr ) Stmt
* IfThenElseStmt → if ( Expr ) StmtNoShortIf else Stmt
  + The category StmtNoShortIf includes all except IfThenStmt

Removing Ambiguity – Factoring

Ambiguous grammar:

* S ::= if B then S
* S ::= if B then S else S

Factoring:

* S ::= if B then S Z (Z and S are statements)
* Z ::= ;
* Z ::= else S

Given

A ::= a b1 | a b2 | … | a bn

Invent a new non-terminal F and replace

A ::= a F   
F ::= b1 | b2 | … | bn

Removing Ambiguity – Substitution

Given all B productions on LHS …

* …
* A ::= B
* B ::= 𝛽1 | 𝛽2 | … | 𝛽n
* …

Replace A ::= B with

A ::= 𝛽1 | 𝛽2 | … | 𝛽n

EBNF

Extending BNF to improve on minor inconveniences

Use brackets for optional part on RHS

* <if\_stmt> := if (<expr>) <stmt> [else <stmt>]

Use braces on RHS for indefinite repeat or none

* <ident\_list> := <identifier>{ , <identifier>}

Multiple choice option

* <term> := <term> ( \* | / | % ) <factor>
* <term> := <term> \* <factor> | <term> / <factor> | <term> % <factor>

Meta-symbols: {}, [], ()

More Powerful Grammars

Context-sensitive: production rules have the form:

A 𝛽 → 𝛽

* A is a non-terminal symbol, , 𝛽, are strings of terminal and non-terminal symbols
* Deciding whether a string belongs to a language generated by a context-sensitive grammar is PSPACE-complete
* Emptiness of a language is undecidable
* LHS may have more than 1 terminal and non-terminal symbols

Unrestricted: equivalent to Turing machine

GRAMMARS

Diagram

Description automatically generated



* 256n many possible different ways of ASCII characters (n: number of characters in our program) to generate program.
  + For example we wrote a program with 100 characters in it. In principle, we might be able to organize them 256100 many different ways as a program. All of these different combinations of characters are not gonna be instance of my language. For example, for C, if I have 100 characters, 123xyz is not a meaningful thing.
* = x + y; -------> this passes the lexer. From the lexer perspective, it is all good (EQ ID PLUS ID SEMICOLON). From the concrete syntax perspective, this is not a statement. Assignment statement requires ID to be defined at LHS. This concrete syntax is gonna be defined by context-free grammars.
* x = x + y; -------> this passes the lexer, and concrete syntax (parser). But x and y may not be the right type. x may be integer and y may be string. This is defined by context-sensitive grammars.
* There is no way to define “correct” programs in a grammar. Our grammar approach helps us very well at lexer part, well for parser part, after this there is no way we are gonna be able to systematize this thing in a nice manner.

BNF (regular expression) 🡪 FLEX 🡪 C CODE

* FLEX algorithm is a program. It is executable. It creates your BNF and generates the C code, your tokens.

BNF (CFG) 🡪 YACC / BISON 🡪 C CODE

* YACC takes the BNF and generates the C code.
* Instances of CFGs that you can handle here is not gonna be infinitely many, it is gonna be limited. When C people designed this one (BNF), they actually knew this. Therefore they designed BNF such that it can use YACC to do that.
* Assume that you don’t like to use C bc you don’t have a C compiler, you have a Java compiler. So people go and take FLEX and YACC and get the Java version of these (JLEX and sth for YACC). Then you generate the Java code. Then you will update rewrite re-edit and all other things to generate rest of the actions.

Some of things after this point are handled by BISON. Rest of the grammars are gonna take the C code; edit update rewrite add code to implement remaining parts (context-sensitive and correct programs).

Semantic Analysis

This goes beyond the CFG

Beyond context free grammar

* Is x declared before it is used?
* Is x declared but never used?
* Is an expression type consistent?
* Is an array reference in bounds? (it has to be done dynamic)
* …

Choices

* Use context sensitive grammars
  + Hard to define and costly to use
    - You have to write all these exceptions, etc.
  + It may defeat the purpose that you would actually use a CFG at the first place
* Use attribute grammar
  + Can help to some extend
* Use ad hoc methods
  + Mostly required
  + Someone define sth, other person say “I can read his/her writing and I can write an algorithm to do that”. We are gonna handle this things by ad hoc.

Why am I using these grammars?

If you had a possibility that the tool about new language is given to you that can define everything about your language. You will define your rules. You have an algorithm, it will take your definition, and it will spit out a code (C code, assembly code, etc.) that can run on any platform. That would have been great. People stuied these things and they said there is no such possibility of a language like that you can define and unambiguously you can write an algorithm to take that and convert it to what you are intending. Context-sensitive grammars are sometimes unsolvable. You cannot decide if a given instance of a language is part of what context-free grammars define.

* define language by writing an novel (bc I have a lot of flexibility by that way) says that what my language do. I give it to someones. They read them and write many lines of code in some language (C, Java, etc.) that can parse the instance of the language that I wrote. Then semantic and compiler are implemented from it.
  + Everyone can understand different things from my novel.
  + I need to be careful about the usage of language defining my language. Natural languages are ambiguous.
  + We need to find an unambiguous thing. One such a thing is a grammar.
    - We have regular grammars but they are weak in terms of describing my needs.

Just having a grammar able to define or able to help me define my language is not sufficient. There is another aspect of this thing. If I am able to describe things with a meta language that can be understood by everybody, can I automatically write programs to get the meaning out of that grammar? If it is regular grammar, yes. If it is context-free sometimes you can. If it is context sensitive, no. Some part is handled, rest of it is gonna be handled by ad hoc methods. I rely on other people to read my descriptions properly and implement them in the right way.

I divided my problem into 3 sections. First 2 sections are handled automatically, the last section is the important part. This one part that cannot be automatically handled is defined or described in a natural language and then I can implement them properly. That part is ad hoc methods.

Concrete syntax can be solved with CFG.

Attribute Grammars

Simple thing that we have done or we do an augmentation on top of the context-free grammar rules that we have.

We will associate terminal and non-terminal symbols an attribute. These attributes (values) are gonna be evaluated by the semantic rules that are attached to those productions that they are involved.

Augment context free grammar with rules

* Each grammar symbol has an associated set of attributes
* The attributes are evaluated by the semantic rules attached to the productions

Syntax directed translation

* Use attribute grammar for semantic analysis
  + Type checking
  + Generate intermediate code or representation

Static Semantics

* Not for running of a program, but for legal forms of programs

ATTRIBUTE EXAMPLE:

Graphical user interface

Description automatically generated

Start symbol is L. Regular CFG that defines my concrete syntax Attributes

Attribute grammar has 2 parts: Regular CFG AND attributes and rules associated with each of production rules

We take each symbol and we associate an attribute with it. I want to get a value for input like “2\*3+4” so I am gonna associate every symbol with a attribute called value. This value is gonna be representing its actual value.

For each production rule, I will have rule that propagates values while using the production rules.

Diagram

Description automatically generated with medium confidence

Parse the input using CFG.

Parse tree says “I can generate a parse tree so this input is the right input in terms of concrete syntax”

Can I use attribute grammar to find the value at the end of this operation? This is our question.

If I have sth like this where A is assignment and E is expression:

* A 🡪 ID = E

For this one my attribute grammar says:

* A.val = E.val
* ID.val = E.val

So I may have more than one attribute rules.

We thought about C but some languages say assignment doesn’t have value, therefore you won’t have first rule in those languages.

If in the parse tree, if my semantic propagates the values from bottom up, this is good. Go bottom up in the parse tree to find the attributes that I have.

In for example typing, things comes from root can affect what I have at children. These types of things might not be easy to do with an attribute grammar.

Semantics and Attributed Grammar

A picture containing diagram

Description automatically generated

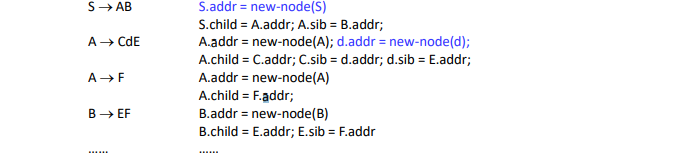
A picture containing graphical user interface

Description automatically generated

Build Parse Tree

Parse tree construction

* Can be done with only synthesized attributes:
  + child: first child pointer
  + sib: right sibling pointer
  + addr: parse tree node address



Parse tree construction example:

Graphical user interface, text

Description automatically generated with medium confidence

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

A picture containing chart

Description automatically generated

A picture containing diagram

Description automatically generated

Two types of attributes:

* Synthesized attributes
  + Shape, circle

    Description automatically generatedAttribute values are evaluated from bottom up
  + The value of the parent is defined on the values of the children
* Inherited attributes
  + Attribute values are evaluated from top down
  + The value of a node is defined on the values of its parent and/or siblings

Attribute rules

* Only reference local information
  + only refer to symbols in the corresponding production

S-Attribute Grammar

Table

Description automatically generated with low confidence

Inherited Attributes

Diagram

Description automatically generated



Synthesized Inherited

T’s type is gonna define L. Type of L has to be inherited by id.

root is defined or not defined, doesn’t matter. D may not need to have a type. I am interested in type of L.

Node either be calculated from its parent or siblings.

For:

$T id $id\_list

A picture containing shape

Description automatically generatedType of id is gonna be defined by the parent. Parent will have a type which is gonna be defined in the case of integer (assume T (type) is integer) then I know that this integer is already having its attribute coming from earlier definition, probably during the lexer, lexer says the keyword int will have an attribute and that attribute is a type attribute and type is specific type (type 0 for example).

That type is gonna propagated to the children.

You have CFG for concrete syntax, you are going into the abstract semantics. Some of the abstract semantics can be easily associated with this concrete syntax. So BNF has production rules, now you write your attribute rules alongside your BNF. That improves the usability of BNF or CFG.

For example you cant encode polymorphism type of things into a CFG. So you will have ad hoc methods to program that particular semantic into your code.

Complicated things will be done in ad hoc manner when you implement things.   
Some of the thing you will be incorporate into your attributed grammar alongside CFG that defines your concrete syntax.

Dependency Graph

Chart

Description automatically generated

Evaluation based on the dependencies

Circularity check

* Dependency graph should be acyclic

Build pass tree

* Fist build the parse tree, then evaluate
* Cannot be done with parsing, require a separate evaluation pass

Can we do better?

* L-attributed grammar
  + Parser stack based technique
  + Marker nonterminals (not covered)
* Rewrite grammar rules

Issues in Attributed Grammar

Graphical user interface, text, application

Description automatically generated



You can synthesize or you can inherit only based on the local production rule.

Addition information lets you have 3 copies. This will lead to excessive copying/inefficiencies.

Issue comes from because of the fact that we are using local rules to do this synthesis or inheritance. We can write code optimizer such as:

* a = b
* c = a
* d = c
  + Here I can simply do d = b instead of 3 assignments.

Graphical user interface, diagram, text, application

Description automatically generated

Graphical user interface

Description automatically generated