1 Compiler Toolchains

General Process

- Compiler :: Src $\rightarrow$ Object File
- Object Files contain machine code in a specific format (such as ELF)
- Linker :: Object File(s) $\rightarrow$ Machine Code
- Loader :: Machine Code $\rightarrow$ RAM $\rightarrow$ CPU (Running)

  Kinds of machine code

- Pure machine code - No OS, no libraries. Just straight code being executed
- Augmented Machine code - Uses an OS for I/O and system libraries
- Virtual Machine Code - System/library abstraction layer Boot strapping: Small initial/temporary compiler that generates a compiler that can compile that language ex: C compiler for Haskell will generate basic Haskell features. Then with those features, the main (reference) compiler can be created.

Compiler targets

- ASM: Create Assembly Code that will later be assembled, linked, etc
  C: The universal assembler. Go to a lower level language
- Relocatable Binary Format: External references, data addresses and function addresses are unbound. They will later be bound by the loader. Position independent code uses relative offsets for addresses. This promotes modular development and cross-language support (makes them easier to implement, that is)
- Absolute Binary Format: Executable format which is directly executable. Mostly in embedded space. Easy to understand, because it's all RIGHT there.

  Syntax - Structure of how it's said/expressed

- Sequence of symbols that are legal, independent of any notion of what they mean: CFG/CFL
  CFG promotes a rigid fundamental scan and tokenization process

- Stataic semantics: Types and type checking. Whether identifiers are declared (declared variables). Number of arguments for a procedure

Semantics - What is meant

- Can include syntactic rules that cannot be expressed by a CFG

Runtime Semantics

- Denotational Semantics Meaning of a construct in terms of its constituants: x=y L[x]m = L[y]m

Compiler process:

1. Source Code
2. → Scanner (pre-processor + tokenizer)
3. token stream
4. → Parser
5. AST (Abstract Syntax Tree) - Can be decorated for type-checking
6. → Some code options Target Code Right Now Optimization → Target code IR (Intermediate Representation) Code. Basically, go to an abstract idea of a machine. Then IR → Optimizer(s) chain. Finally to Target-Code Generator
7. Question: Some compiler writers use C as their target code. What some of the pros and cons of this?

AC to DC:

- f b i a a = 5 b = a + 3.2 p b
- f b ; i a ; a = 5 ; b = a + 3.2 ; p b
- We have a CFG for this.
- Syntactically Valid: Checking static semantics

Recursive-Descent parsing: 1 function per CFG Production. The function implements rules based on the CFG. It may call itself or other production functions

Implementing one of the Recursive-Descent Functions:

```
prodecure Expr()
    if ts.peek() = plus or ts.peek() = minus
    then
        if ts.peek() = plus
        then
            call Match(ts, plus)
        else
            if ts.peek() = minus
            then
                call Match(ts, minus)
            end
        end
        call Val()
        call Expr()
    else
        if ts.peek() = $
        // ignore lambda production
    end
end
```

Abstract Syntax Trees and Parse trees

- AST has terminal nodes that contain a node-type, and terminal value. The node-types represent the functions that have been called
- stmts wind up being multichild-trees, with a lot of 1-height trees. Like the parse tree, the AST MUST be interpreted left-to-right

Scanning and Regular Expressions

- Synonamous terms: Scanner == Lexer == Lexical Analysis
- Scanning tries to match the absolute longest possible match
- Regular Expressions are a good way to do lexing
- Alphabet: Σ
- We have Token Classes, which describe the class of tokens that are matched by a particular regular expression. An instance of a token class is a particular token that matches
- Regex Golf (without the golfing)

- Definitions:
  - Eol = End of Line
  - L = All Letters
  - D = All Digits
- Example: Phone Numbers: $D^3 - D^3 - D^4$
- An integer: (λ—-—+)D+
- A C identifier: (L—_)(D—L—_)*
- Java or C++ // comments: //Not(Eol)*Eol
- Floating Point number: (λ—-—+)(D*.D+—D+.D*)
- Scientific Notation: (λ—-—+)(λ—D*.)D+(e—E)(λ—-—+)D+
- A comment enclosed by '##' on both sides: ##Not(Eol)*##
- An identifier that does not permit two _'s together or a final _: Not(_)*(_—λ)
- A C style multiline comment: /(Not((?¡*))Not(/)—Not(*))*/
- A simple division expression with balanced parens:

Transitions in FAs can become Transducers by using an Action Table

- We have a table representing FA with transitions. With states vs input character
- The action table has the exact same structure, excpet that instead of moving to a state, it calls a function at the transition

Flex / Lex

- Structure: declarations - Data structures and types that are needed %% Optional Lex definitions. If this is empty, do not include an additional %% ex. Blank " " Digits [0-9]+ %% regular expression rules - Regular expressions used for tokenizaton into the types ex. Blank+ return SPACE; (Digits—Digits"."Digits) return NUM; %% subroutine definitions - functions used as outputs from regular expressions

RE → DFA → NFA Make deterministic...

1. Record start state (1) $\rightarrow$ 1,2
2. Find all the possible states that can be reached by all the characters, and create next set of states that can be reached

Parse f(v+v)

1. E
2. Prefix (E)
3. Prefix (v Tail)
4. Prefix (v + E)
5. Prefix (v + v Tail)
6. Prefix (v + v λ)
7. f(v + v)

for p in prods if lambda: true else if only nonterminals return
Follow: set of prod rules that produce the symbol (α)

Define LR Item - Has a production with an index into it, indicating how much has been parsed. λ productions Have an index only to 0, meaning that there's one parse step, and only one

Closure (LRItem) I is a sec of items from the grammer. It will yield another set of items K = dup(I) if $A \rightarrow \alpha \bullet B\pi$ is an item in K, AND there exists $B \rightarrow \gamma$, then add $B \rightarrow \bullet\gamma$ to K repeat for all in K return K

Goto(I, X) where I is a set of items and X is a grammer symbol (non terminal, terminal, or terminator). It will return a new set of items T = { t in I — X to the right of $\bullet$} T' = {t in T| $\bullet$ moved to the right, after X} return Closure(T')

- Closure ($\{S \rightarrow \bullet E\$\}$)
  =¿ $\{S \rightarrow \bullet E\$$,
  $E \rightarrow \bullet plusEE$,
  $E \rightarrow \bullet num\}$
  = \$1
- Goto (\$1, num)
  =¿ {Closure($E \rightarrow num\bullet$)}
  =¿ $\{E \rightarrow num\bullet\}$
  = \$2

- Goto (\$1, plus)
  =¿ {Closure($E \rightarrow plus \bullet EE$)}
  =¿ {$E \rightarrow plus \bullet EE$,
  $E \rightarrow \bullet num$,
  $E \rightarrow \bullet plusEE$}
  = \$3

- Goto (\$3, E)
  =¿ {Closure($E \rightarrow plusE \bullet E$)}
  =¿ {$E \rightarrow plusE \bullet E$,
  $E \rightarrow \bullet num$,
  $E \rightarrow \bullet plusEE$}
  = \$4

Now, bring it back in. Calculate the LR(0). Also called the LR Canonical States

1. $C = \{Closure(\{S- > \bullet RHS, ...\})\}$ #Set of all the possible start states. Closure that

2. repeat:
   m = —C—
   for each I in C:
   for each X (Grammer symbol):
   J = Goto(I, X)
   if J not empty and not in C:
   add J to C
   until —C— == m or until it doesnt change sizes

3. Create an FA from this, and/or an LR(0) parse table

The table:

	plus	num	$	S	E
I_0	2	3			1
I_1			acc		
I_2	2	3			4
I_3	E -¿ Num				
I_4					
I_5	E -¿ plus E E				
I_a	S -¿ E$				

Read through 6.4 p 224 #3 p 225 #6 a & c p 225 #9
p224 #3 Grammer:

(a) q$

0	Init	q$
0	rd $\lambda \rightarrow$ B	Bq$
0 B5	sh B	$
0 B5 q7	sh q	$
0 B5	rd q $\rightarrow$ Q	Q$
0 B5 Q6	sh Q	$
0	rd BQ $\rightarrow$ A	A$
0 A1	sh A	$
0 A1	rd $\lambda \rightarrow$ C	C$
0 A1 C14	sh C	$
0	rd AC $\rightarrow$ S	S$
0 S4	sh S	$
0 S4 $8	sh $	
0	rd S$ $\rightarrow$ Start	Start
0 Start	sh Start	

(b) c$

0	init	c$
0	rd $\lambda \rightarrow$ B	Bc$
0 B5	sh B	c$
0 B5	rd $\lambda \rightarrow$ Q	Qc$
0 B5 Q6	sh Q	c$
0	rd BQ $\rightarrow$ A	Ac$
0 A1	sh A	c$
0 A1 c11	sh c	$
0 A1	rd c $\rightarrow$ C	C$
0 A1 C14	sh C	$
0	rd AC $\rightarrow$ S	S$
0 S4	sh S	$
0 S4 $8	sh $	
0	rd S$ $\rightarrow$ Start	Start
0 Start	sh Start	

(c) adc$

0	init	adc$
0 a3	sh a	dc$
0 a3	rd $\lambda \rightarrow$ B	Bdc$
0 a3 B9	sh B	dc$
0 a3 B9	rd $\lambda \rightarrow$ C	Cdc$
0 a3 B9 C10	sh C	dc$
0 a3 B9 C10 d12	sh d	c$
0	rd aBCd $\rightarrow$ A	Ac$
0 A1	sh A	c$
0 A1 c11	sh c	$
0 A1	rd c $\rightarrow$ C	C$
0 A1 C14	sh C	$
0	rd AC $\rightarrow$ S	S$
0 S4	sh S	$
0 S4 $8	sh $	
0	rd S$ $\rightarrow$ Start	Start
0 Start	sh Start	

p 225 #9 The primary reason that this is un-ambiguous is the singular derivables in the right-hand side of all the rules. With each derivation, there's only one possible expansion choice

The grammer

- $S \rightarrow E\$$
- $E \rightarrow EplusE$
  — num

Creating the table

- $I0 = \{S \rightarrow \bullet E\$, E \rightarrow \bullet EplusE, E \rightarrow \bullet num\}$
- $Goto(I0, E) = I1\{S \rightarrow E \bullet \$, E \rightarrow E \bullet plusE\}$
- $Goto(I1, plus) = I2\{E \rightarrow Eplus \bullet E, E \rightarrow \bullet num\}$
- $Goto(I2, E) = I3\{E \rightarrow EplusE\bullet\}$
- $Goto()...$

Grammer which is C-like, but minimal

page 139 q7

Grammer:

1. Start → E $
2. E → T plus E
3. — T
4. T → T times F
5. — F
6. F → (E)
7. — num

(a) leftmost derivation of target: num plus num times num plus num $
Start →
E $ →
T plus E $ →
F plus E $ →
num plus E $ →
num plus T plus E $ →
num plus T times F plus E $ →
num plus F times F plus E $ →
num plus num times F plus E $ →
num plus num times num plus E $ →
num plus num times num plus T $ →
num plus num times num plus F $ →
num plus num times num plus num $

(b) Rightmost derivation of target: num times num plus num times num $
Start →
E $ →
T plus E $ →
T plus T $ →
T plus T times F $ →
T plus T times num $ →
T plus F times num $ →
T plus num times num $ →
T times F plus num times num $ →
T times num plus num times num $ →
F times num plus num times num $ →
num times num plus num times num $ →

(c) Pluses always come before times. They are the most important. Then leftmost causes Left associativity, and Right casuses Right assosiativity

page 225 q6 parts b,d

(a) Skip

(b) (a) S → StmtList \$

    (b) StmtList → Stmt semi StmtList

    (c) — Stmt

    (d) Stmt → s

    This grammer is NOT $LR(0)$ because the StmtList has two rules that start with the same symbol. When that symbol is on the left-side, we don't know if we should finish the rule or if we should read the next character

(c) Skip

(d) (a) S → StmtList \$

    (b) StmtList → s StTail

    (c) StTail → semi StTail

    (d) — λ

    Yes, this grammer is LR(0). The use of tail makes it very clear that the first 's' should be read and rule 2 used, THEN move to the next character/input, from which, the rule to use can be determined based on only the next input.