Intro

08 January 2018 14:04

LP's are crucial link between software and hardware

- networking protocols
- CAD (design automation)
- Systems integration
- Compilers

http://web.stanford.edu/class/archive/cs/cs143/cs143.1128

Course

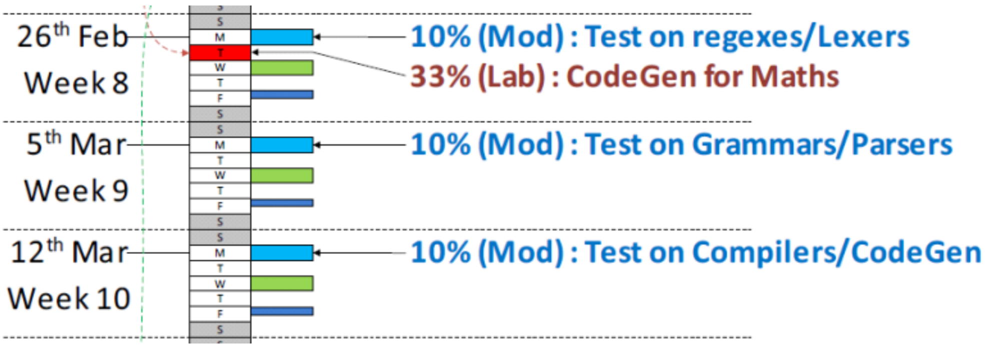
Deliverables due on Tuesdays at 22:00

Module:

- 24 Lecture/Tutorial Hours
- Application - 60%
  - one formative (Parsing 0%)
    - Tuesday **20th February**
  - one summative - C compiler suite for MIPS
    - Tuesday **27th March at 22:00**
    - 20% C Translator (to Python)
    - 30% C Compiler to MIPS assembly
    - 10% Test Suite - test cases for the above
- Knowledge - 40%
  - 3 Tests in class - 3 x 10%
    - 10 MCQ's per test
  - Compiler documentation 10%

Comp Lab: - Skills

- 6 Structured lab sessions
- Each Lab deliverable
  - practice lecture before first lab
  - 2 lab sessions
  - auto-assessment script for 50% of marks
- 3 submissions
  - Lexers - regular expressions and Flex
  - Parsers - grammars and Bison
  - CodeGen - emitting code from an AST



Getting Started

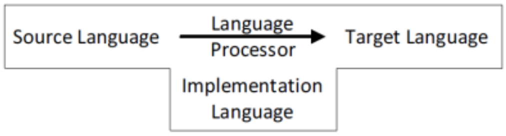
09 January 2018 23:02

Language Processor **transforms / maps** from source/input language to object/target language

LP - focus on written language
- formal - e.g. C++ - defined
- implicit - e.g. some data - not defined - dynamic

Formal Language:
- **Character** - atomic parts
- **Tokens** - group of characters
  - **keywords -** finite - spec
  - **identifiers -** user defined - infinite
  - **operators -** finite - spec
  - **literals -** infinite
- **Syntax** - ways to order tokens
- **Grammar** - rules defining restrictions on syntax
  - **Hierarchy - nested**
  - **declarations**
    - **declarations**
    - **statements**
    - **expressions**
  - **statements**
    - **statements**
    - **expressions**
  - **expressions**
    - **expressions**
- **Semantics** - valid meaning of what's been written
  - grammar can be correct - but no valid meaning present

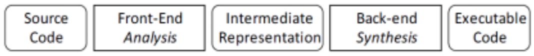
LP described using T-diagram



- **self-hosting** - Source and Implementation lang same
- **boot-strap** - Target and Implementation lang same

LP Evaluation:
- **correctness of output** - MUST be correct
- **follow input lang specification** - e.g. all C should be valid according to C spec
- Quality of output
  - code size
  - code execution speed
- LP speed
  - transformation speed
- User-friendliness
  - error message usefulness

Compiler Stages:



- Front-end - syntax, grammar and program correctness
- Back-end - analyse, optimise and generate output code
- FE and BE communicate via IR

General Principles of Compiler:
- **Input:** stream of bytes (symbols)
- **Lexing:** Group symbols into tokens
  - keywords, identifiers, operators, literals
- **Parsing:** Group tokens into structure using grammar
  - programs, functions, declarations, statements, expressions
- **Analysis:** assign meaning to structure
- **Synthesis:** transform the structure
  - optimise, verify
- **Output**

Grammar

14 January 2018 22:18

<u>Semi-formal definition of grammar:</u>

- Production Rules
  - left-hand side: syntactic construct being defined
  - right-hand side: one possible form of the construct
- Non-terminal symbols
  - groupings cluster/tree of terminal symbols
  - can be recursively defined
  - When talking in abstract - lower case: **a, b, c..**
- Terminal symbols
  - tokens / characters - leaf of tree
  - no more expansion
  - When talking in abstract - upper case: **A, B, C..**

- Sequences of symbols in production rules
  - represented as Greek letters in abstract
  - **empty** sequence - epsilon

<u>Regular Expressions:</u>

- like EBNF, but with **no non-terminals**
- only operate on characters
- still have repetition and alteration

Regexes also bring their own syntactic sugar:

Sets	[αγβ]	Match all characters in the brackets
Ranges	[α-γ]	Match characters between α and γ
Exclusion	[^α]	Match anything *except* α
Anything	.	Match any single character
Start of string/line	^	Matches just before the first character
End of string/line	$	Matches just after the last character

<u>Derivation:</u>

- strings from Grammar

Reduce input using grammar

R1: Expr → '(' Expr Oper Expr ')'
R2: Expr → 'x'
R3: Oper → '+'
R4: Oper → '*'

Rule	Derivation	Input
Start	Expr	(x + (x * x))
R1	'(' Expr Oper Expr ')'	x + (x * x))
R2	'(' 'x' Oper Expr ')'	+ (x * x))
R3	'(' 'x' '+' Expr ')'	(x * x))
R1	'(' 'x' '+' '(' Expr Oper Expr ')' ')'	x * x))
R2	'(' 'x' '+' '(' 'x' Oper Expr ')' ')'	* x))
R4	'(' 'x' '+' '(' 'x' '*' Expr ')' ')'	x))
R2	'(' 'x' '+' '(' 'x' '*' 'x' ')' ')'	))

<u>Computer does this via Trees:</u>

Many terminal symbols are irrelevant once tree is built

EBNF - Extended Backus-Naur Form

Name	Syntax	Meaning
Alternatives	A | B	Either A or B
Kleene Star	A*	Zero or more instances of A
Kleene Cross	A+	One or more instances of A
Optional	A?	Zero or one instances of A

- syntax for RHS of production rules
- can be expanded to pure production rule form

<u>Grammar Equivalencies</u>

- infinite number of equivalent grammars for same language
  - always add epsilon rules - trivial
- may also be infinite non-trivial grammars

<u>Left Recursion:</u>

- A non-terminal X
  - where X appears as the left-most rule in any production rule
  - either directly or via another non-terminal
- left-most derivation
  - always replace the left-most non-terminal next
  - doesn't always guarantee a unique derivation

<u>Ambiguity:</u>

- there exists a string with more than one left-most derivation
- algorithm to check for all ambiguity proven not to exist
  - so don't bother

<u>Fixing Ambiguity:</u>

- precedence
  - ordering of production rules
- switch to bottom-up parser
- restrictions
- semantic actions - user code to deal with certain issues
- Rewriting:
  - expand into greater number, but simpler production rules

Lexer

15 January 2018 19:38

Lexer - regex tools built for tokenising
part of LP front-end

.lex file into 'flex'
produces C file which compiles into an executable - the 'lexer'
Text Input -> Lexer -> Tokens

Lex Program:
%% separates the 3 parts - unique from C and regex
parts contain embedded C
whitespace - sensitive to whitespace sometimes - start rules in first column of line

- **Declarations**
  - define named patterns
    - separate definition of pattern from the corresponding action
  - %{...%} - put other C code in declarations
    - useful for global variables and **headers**
- **Pattern Matching**
  - map patterns to actions
  - Pattern - what to look for
    - multiple pattern match
      - Flex will pick *longest* matching token
      - if same length - pick first match
  - Action - C code to execute if pattern is found
  - **[.]** : is a catch all for any token that doesn't match a pattern
- **User Functions**
  - helper functions
    - separate description of action from use of action
  - for a standalone tokeniser - can put a main here

Lexer Usually a building block

- yylex called from the parser
- main not in lexer
- Parser generator can auto-generate lex input
  - for grammar descriptions that allow definition of tokens

yy

- **char *yytext** - text of the *current* token
- **int yyleng** - # of chars in current token

- **FILE *yyin** - source (default stdin)
- **FILE *yyout** - output (default stdout)

- **int yylex()** - initiates tokenisation of yyin
  - **PUSH** Operation
    - user calls yylex()
      - runs over all input tokens
    - actions cause side-effects - but do not return
  - **PULL** Operation (Preferred)
    - actions use **return** to pass back to caller
    - call to yylex() returns one token
    - lexer (program) remembers position in FILE between calls to yylex
- **int yyrestart(FILE *next)** - set yyin=next, restart tokenisation

- **int yywrap()** - what happens when input runs out
  - **%option noyywrap** - just exit

DFA and NFA TEST

22 January 2018 19:31

Regex's and Lexers

Finite state automata = fixed number of states, output only depends on input and current state

FSA:

- finite set of states
  - subset - set of final states
- finite set of input symbols
- single start state
- transition funciton

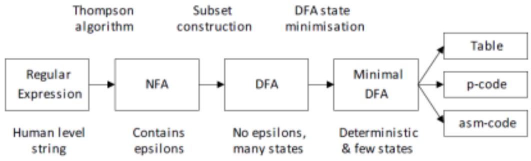
DFA - Deterministic FA

- no **epsilon** transition function

NFA - Non-deterministic FA

- **epsilon** transitions - state transition without consuming input symbol
- one or more transitions for same input
- number of active states can increase with time

Regex flow - producing a lexer from lex file

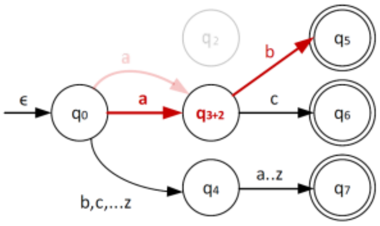


Summary:

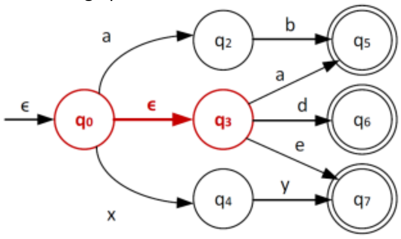
1. Thompsons algorithm: regex -> NFA
2. Subset construction NFA -> large DFA
3. DFA minimisation: large DFA -> small DFA
4. FSM interpretation/generator: small DFA -> FSM

NFA

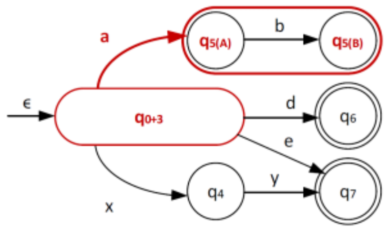
Eliminating non-deterministic arcs



Eliminating epsilon arcs



both final states



Thompson's Algorithm

A method to **convert regex into NFAs**

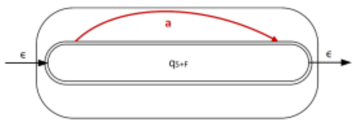
- automated
- uses eliminating techniques from the left
- guaranteed to complete in finite time
- doesn't worry about efficiency, lots of:
  - epsilons
  - redundant states
  - non-deterministic transitions

Composite Expression

- has ONE entry state - **qs**
- set of final states - **qf**

Composite Expression formed from Primitives:

- Empty Symbol
  - wire
- Single Symbol
  - consumes on input symbol
- Sequence
  - sequence of symbols from entry to final
- Alternation
  - two or more different paths from start to final
- Kleene Star
  - zero or more



Subset Construction

NFA to DFA

merge states connected by epsilon into *sets of states*

map sets of states in NFA into **one state** in DFA

- $\epsilon - closure(NFA, s)$ function
  - goes through power set of NFA states

DFA Minimisation

Worst case conversion - exponential growth

- meaning larger tables

Some states unreachable - i.e. not a valid input string for transition

Partitioning:

- split sets of state by equivalence
- two states are not equivalent if

map sets of states in NFA into **one state** in DFA

- $\epsilon - closure(NFA, s)$ function
  - goes through power set of NFA states
  - checks if transition is to equivalent state and is an epsilon transition
  - returns set of states reachable via epsilon
- *move* function
  - return set of states reachable from symbol c

Steps:

1. start = closure(NFA.start) - returns power set of starting states
2. stack - each element is an NFA state
3. stack.push(start) - must explore through all starting points
4. while stack length > 0
   a. current = stack.pop
   b. for each valid NFA input symbol
      i. dest_state_set = closure (move (current set of states, each NFA symbol))
      ii. add transition (current set of states, each NFA symbol, dest_state_set)
      iii. if dest not in set of NFA states
         1) new set of NFA states in set of sets
         2) push to stack
5. stop when stack is empty
6. Get DFA final set
   a. check every state, in set of sets of NFA states
   b. any NFA state set containing an NFA final is a DFA final
      i. add this to set of DFA states
7. return, start states, DFA states, transitions, and final states

Partitioning:

- split sets of state by equivalence
- two states are not equivalent if
  a. only one of them is final, or
  b. same input symbol maps each to a different state

So to split a set:

1. stack.push (same = DFA final (must be at least one))
2. while stack not empty
3. for all DFA symbols
   a. get power set from move(same, $symbol)
   b. get power set from move(altState, $symbol)
   c. check if sets match
      i. no match - state is diff
      ii. match - add to same
4. push diff to stack (if diff len > 0)
5. end when stack empty
6. return same - set of states that are the same

Parsing

29 January 2018 14:01

Grammar:

- refactor to have non-terminals defined at top - parser job - group terminals into non-terminals
- terminals defined in lower section - lexer job - split characters into non-overlapping terminals

1. **Lexical Analysis**
- raw character stream into tokens
    - **whitespace matters here**
- tokens described using regular expressions

2. **FSA**
- token type in data structure (reference)
- job:
    - match token with a token type - so internally know what each token does
    - do this once all tokens have been stored / detected
- **Synthesised Attributes**
    - a ***production rule*** (grammar item) can have attributes
        - made up from child tokens
    - all terminals have a ***raw*** attribute
        - e.g. 'yytext' - the raw characters00000000000000

3. **Parsing** - *recursive descent parser*
- non-terminal symbols rely on **recursion** in their definitions
- procedure
    - start from top of grammar - work inwards
    - use recursive functions
- function naming - pXXX - parse XXX
    a. map terminal token RAW into higher level attributes (parse data)
    b. non-terminals - follow productions rules closely
        i. RHS recursive - avoids infinite loop (LHS wouldn't know when to end in some cases)
- AIM: Produce the **Abstract Syntax Tree (AST)**
    - contains operator names, values, operations
        - but not whitespace or brackets
- TESTING
    - **'round-tripping'**
    - print out info from 'walking' the tree
        - should have same functionality / meaning
        - but will look different (brackets etc)

Bison - Yacc

Yacc file (.y) into Bison - gcc / link with lex file to produce executable

Structure: - separated by %%

1. **Declarations**
    a. **%code**
        code that will be placed in header file
    b. **%start**
        nonTerminal - start symbol for the grammar
    c. **%token**
        list types of tokens that lexer can produced
        separated by a space
    d. **%union**
        attributes a token can have
        C code inside {}
    e. **%type**
        match symbols with attributes
        each on a new line starting with *%type*
2. **Translation Rules** - grammar production rules
    - EBNF-like rules, : instead of ::=
    - A : B C {Action}
        - A - non-terminal being defined
        - B, C - *symbols* in the production
        - {Action} - C code
            - $ representation
            - $$ = A
            - $1.. = 1.. 'thing' in the rule
3. **User functions**
    - helper functions
    - main - run **yyparse** in push mode - parsers entire thing

Integration with Lexer

- relies in yylex to return next token
- gets value from yylval
- Need a .lex with a .y file
    - link both together into an executable
- Bison produces token types
    - include bison header in lex
- Use make files

Each Step Bison does:

- shift - consume terminal - put on stack
- reduce - collapse stack and simplify

Makes decision on which one by only looking at next token

Reduce-Reduce errors:

- Bison could reduce by more than one way

Shift-reduce errors:

- could shift a terminal or put on stack - both would be correct

Dangling Else Problem

- which if statement does the else belong to if no explicit scoping
- C - most recent if statement
    - indicate precedence to parser
    - or use user code intervention during pairsing

Intro to code gen

18 February 2018 19:43

AST Choices

18 February 2018 20:05

ABIs and Linking

18 February 2018 20:06