Compilers 2021/2022: Test 1 Cheat Sheet

Compiler Overview

The frontend

- Scanner
  - Maps character stream into tokens (name and attributes)
- Parser
  - Recognizes context-free syntax and reports error
  - Guides context-free syntax and reports errors
  - Guides context-sensitive ("semantic") analysis (type checking)
  - Builds IR for source program, ie. an AST

The backend

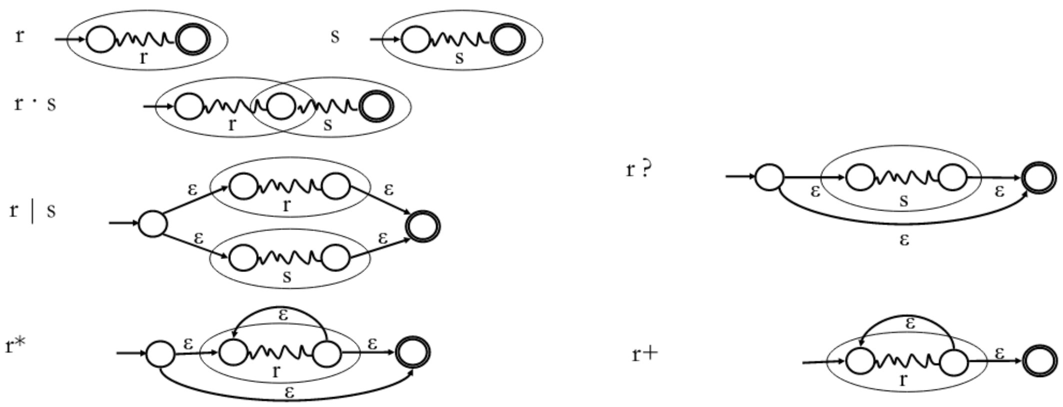
- Instruction Selection
  - Produce fast, compact code
  - A pattern matching problem
- Register Allocation
  - Have each value in a register when it is used
- Instruction Scheduling
  - Avoid stalls and interlocks

Lexical Analysis

Regular expressions

- Lexical patterns form a regular language
- Any finite language is regular
- Recognizable by DFAs

RE to NFA (Thompson Construction)



NFA to DFA (Subset Construction)

DFAedge

Given symbol c and a set of states S, what states can you reach?

$$DFAedge(S,c) = \varepsilon - closure\left(\bigcup_{s \in S} edge(s,c)\right)$$

DFA State	NFA States	ε-closure after transition on...		
		0...9	–	.
0	{1, 2}	{2, 3, 4, 8}	{2}	error

DFA State Minimization

- Normalization
  - Assure every state has a transition on every symbol
  - Add missing transitions to a trap state
- Algorithm
  - Start with accepting vs non-accepting partitions of states
  - Repartition based on transitions for each symbol: find same partitions for every symbol

DFA to RE (Kleene Construction)

- The sets that take the DFA from state qi to qj without going through any state numbered higher than k
- When k=0, consider direct transitions
- A dynamic programming approach

$$R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \mid R^{k-1}_{ij}$$

Syntatical Analysis

Context free grammars

A context free grammar $G = (\Sigma, N, S, P)$ is defined by:

Σ set of *terminal* symbols;

N set of *non-terminal* symbols;

$S \in N$ initial symbol;

P set of de *production rules* $X \to \alpha$ where:

- X is non-terminal;
- α is a sequence (maybe empty) of terminal or non-terminal symbols

Ambiguity

- A grammar producing same word with different syntax tree
- Eliminate forcing priority and/or associativity

Parsing

Top-down parsing

Recursive descent parsing

- Consume tokens left to right

- Map each none terminal to a function
- Map each production to a different case
- Decide which production to use using the next token

LL Parsing

- Recursive descent parsing technique
- LL(k) means: Left-to-right parse, Leftmost derivation, k-symbols lookahed
- Does not support left recursion

Left recursion removal

$$
\begin{aligned}
E &\to E\ +\ T \\
E &\to T
\end{aligned}
$$

E produces sums of terms, i.e. $E \Rightarrow^* T + T + \cdots + T$.

Let us define an equivalent grammar adding a new non-terminal symbol E':

$$
\begin{aligned}
E &\to T\ E' \\
E' &\to +\ T\ E' \\
E' &\to \varepsilon
\end{aligned}
$$

LL(1): Predictive parsing

- Sufficient for programming languages
- For each non-terminal symbol, each rule must correspond to a different FIRST set (must left factor rules)
- A parsing table maps non-terminals to input and corresponding rule to choose
- Build the table based on NULLABLE, FIRST and FOLLOW
- Rely on the parsing table and an auxiliary stack to parse input

We choose a production rule $N \to \alpha$ on input symbol c if:

1. $c \in FIRST(\alpha)$, or
2. $Nullable(\alpha)$ and $c \in FOLLOW(N)$.

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	input	action
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	$B \to bB$
$\underline{b}B\$$	$\underline{b}\$$	consume b
$\underline{B}\$$	$\underline{\$}$	$B \to \varepsilon$
$\underline{\$}$	$\underline{\$}$	consume $\$$
ε	ε	**accept**

Bottom-up parsing

LR Parsing

- LR(k) means: Left-to-right parse, Rightmost derivation (reversed), k symbols lookahed
- Deals easier with ambiguity and recursion
- Consult the parsing table to parse input using shift, reduce and goto actions
- Read back reductions to get the derivations

	a	b	c	$	T	R
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		

(0) $T' \to T\,\$$
(1) $T \to R$
(2) $T \to aTc$
(3) $R \to \varepsilon$
(4) $R \to bR$

state	stack	input	action
0	ε	aabbbcc$	shift 3
3	a	abbbcc$	shift 3
3	aa	bbbcc$	shift 4
4	aab	bbcc$	shift 4
4	aabb	bcc$	shift 4
4	aabbb	cc$	reduce $R \to \varepsilon$; go 6
6	aabbbR	cc$	reduce $R \to bR$; go 6
6	aabbR	cc$	reduce $R \to bR$; go 6
6	aabR	cc$	reduce $R \to bR$; go 2
2	aaR	cc$	reduce $T \to R$; go 5
5	aaT	cc$	shift 7
7	aaTc	c$	reduce $T \to aTc$; go 5
5	aT	c$	shift 7
7	aTc	$	reduce $T \to aTc$; go 1
1	T	$	**accept**

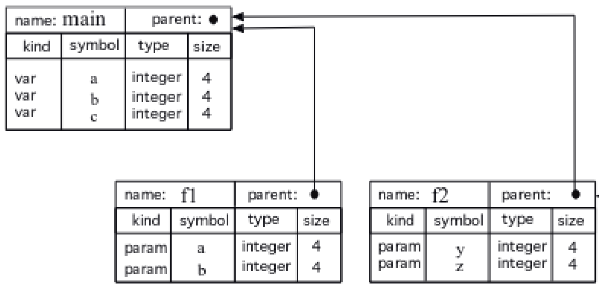
Semantic analysis

Lexical scope (static scope)

- Elements usage correpond to the closest declaration in the AST

Symbol table

- Relates identifiers with semantic information, such as registry location, types and variable values
- Typically implemented with a hash map
- Must represent scopes in some form
    - Open a new scope
    - Close a new scope restoring the previous stack



Type checking

- Assert correct function parameter types, variable attribution types
- Generate more efficient code and avoid errors at run-time

Attribute grammars

- Semantic rules for the grammar
- Often implemented with visitor pattern, recursively
- Attributes can be inherited (variable types) or synthesized (types of sub-expressions)

- Type checking may be made by traversing the AST (one or more times)
- As the AST is a recursive structure type checking uses recursive functions
- The compiler builds node attributes; examples:
    - Types;
    - Symbol Table (context)
- Synthesized attributes: bottom-up
- Inherited attributes: top-down

Grammar Rule	Semantic Rules
decl → *type var-list*	
type → **int**	*dtype* = *integer*
type → **float**	*dtype* = *real*
var-list$_1$ → **id** , *var-list*$_2$	*insert*(**id** .*name*, *dtype*)
var-list → **id**	*insert*(**id** .*name*, *dtype*)

(Attribute Grammar for Type Declarations)