Compiler Design

- A grammar is ambiguous if it has two different parse trees for any string by using either LMD only or by using either RMD only.
- If G is an unambiguous grammar then every right sentential form has a unique handle.
- Every DCFL has an unambiguous grammar
- The number of states of CLR is more than all LR parsers.
- Divide by integer verified at semantic analysis phase (NPTEL, GO test)
- The number of states of LR(0), SLR(1), and LALR(1) is the same
- Keywords of a language are recognized during Lexical Analysis phase
- The code-optimization on intermediate code generation will always enhance the portability of the compiler to target processors. The main reason behind this is, as the intermediate code is independent of the target processor on which the code will be executed, so the compiler is able to optimize the intermediate code more conveniently without bothering the underlying architecture of the target processor.
- The drawback for quadruple representation of 3-address code is it requires more space compared to triple notation.
- We can convert Regular Grammar into CNF
- S-attributed SDT uses Bottom up parser., If every L attributed is forced to only behave like S attrib, then it also can be parsed by Bottom up parser.
- Viable prefixes are always present at the stack., They can't contain symbols on the RHS of a handle.
- Theorem: Set of viable prefixes is regular language, recognised by a DFSA.
- Static single assignment is a valid intermediate code representation
- Live variable analysis
  Calculate Gen(X) and KILL(X) at each node
  Calculate OUT(X) = IN of all the outgoing edges
  Calculate IN(X) as GEN(X) union (OUT(x)-KILL(x)) at each node
  Move from bottom to upward transitions
  Live variable analysis is really very important for register allocation and dead code elimination
- An annotated parse tree is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annotating or decorating the parse tree. The drawback for quadruple representation of 3-address code is it requires more space compared to triple notation. The drawback of triples is that, in triples form statements cannot be moved, whereas it is indirect triples which has the drawback that it requires two memory accesses.
- Every LR(0) must be LALR(1) but every LALR(1) need not be LR(0).
- Register Spilling: When the number of live variables exceeds the number of registers, then some of the values in the registers are stored in the main memory. This concept is known as register spilling.
- Reduce entries decreases error entries increase when we change LR(0) to SLR(1)
- Symbol table entries are only created in the analysis phase i.e. LEX, SYNTAX ANALYSIS and SEMANTIC ANALYSIS
- Sentinel is a special character which can never be part of the source program, usually put at the end of the program eg. EOF
- CYK algorithm O(n^3) dynamic programming
- CNF : A->BC or A->a this form of the CFG is known as CNF
- If language is generating epsilon that is fine, but no CNF can generate epsilon on their own S->ϵ is allowed
- Bottom up parsing requires rightmost derivation in reverse (Gate 2019)
- Handle Pruning: When you are able to reduce something to a Terminal, that RHS is known as handle
- Shift Reduce Parsing :Four possible actions on SR parsers : - shift, reduce, error and accept
- Conflicts in SR parsers: For a conflict to happen, there must be at least a single reduce move.
- LR parsing: Simple LR parser
- LR(k) explained: L - Scan input from left to right, Rightmost derivation in reverse,(K) means lookahead
- LR parsers can detect syntactical errors as soon as it is possible to do on a left to right scan of the input.
- LR(k) is proper superset of LL(k) grammars
- SLR : put reduce moves only in the follow of LHS
- LALR : put reduce moves only in the lookahead after merging the states from CLR
- CLR is more powerful than LALR
- LALR and CLR put Reduce moves only in the lookahead.

- Viable Prefixes: These are the pigeons which always appear on the stack of SR parser. Equivalently, this means that the set of viable prefixes for a given SLR (1) grammar is a regular language! Viable prefix depends on what input string you want to parse Whatever is present on the stack, is a set of viable prefixes.
- Operator precedence parser, Ambiguity is allowed, but i. grammar must not contain ϵ, ii. No two Terminals should be concatenated without an operator in the middle.
- Syntax directed Translation Scheme : L attributed - inherit from only LHS or left sibling - top down parsing, S attributed - inherit only from RHS - bottom up, Synthesised Attributes Only S attributed, Inherited Attributes : Can be both S and L attributed
- Application of SDT: Construction of syntax tree, Infix to Postfix translation
- Intermediate Code generation: DAG,TAC,Quadruples, Indirect triples, Triples, Syntax Tree
- Static Single Assignment (no variable is repeated once used)
- Runtime environment: Stack , Static, Heap, Code
- Heap —> FREE MEMORY <—— Stack
- // They grow towards each other
- Stack: Activation Tree
- Procedure calls and returns are usually managed by a run time stack called the control stack. Each live activation has an activation record.(frame) having root at the bottom of the stack. Activation Record Global variables are allocated static storage
- Basic Blocks and flow graph

In addition to creating an intermediate representation, a compiler front end checks that the source program follows the syntactic and semantic rules of the source language. This checking is called *static checking*; in general "static" means "done by the compiler."[11] Static checking assures that certain kinds of programming errors, including type mismatches, are detected and reported during compilation.

Suppose we start with a regular expression r and convert it to an NFA. This NFA has at most $2|r|$ states and at most $4|r|$ transitions. Moreover, there are at most $|r|$ input symbols. Thus, for every DFA state constructed, we must construct at most $|r|$ new states, and each one takes at most $O(|r|)$ time. The time to construct a DFA of s states is thus $O(|r|^2 s)$.

Lexical errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.

Syntactic errors include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, in C or Java, the appearance of a `case` statement without an enclosing `switch` is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Semantic errors include type mismatches between operators and operands, e.g., the return of a value in a Java method with result type `void`.

Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator `=` instead of the comparison operator `==`. The program containing `=` may be well formed; however, it may not reflect the programmer's intent.

A ugmented
R eality
C A
S imple
L ocal
T rain

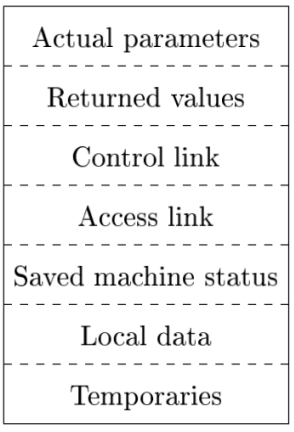

Figure 7.5: A general activation record

The lexical analyzer always tries to map to the maximal token. i.e.,++ is always preferred to +.

3.5.3 Conflict Resolution in Lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

by the NFA construction, which is $O(|r|)$, and therefore, we conclude that it is possible to take a regular expression r and string x, and tell whether x is in $L(r)$ in time $O(|r| \times |x|)$.

Automaton	Initial	Per String
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Figure 3.48: Initial cost and per-string-cost of various methods of recognizing the language of a regular expression

Control and Access Links

- The **control link** of a function is a pointer to the function that called it.
  - Used to determine where to resume execution after the function returns.
- The **access link** of a function is a pointer to the activation record in which the function was created.
  - Used by nested functions to determine the location of variables from the outer scope.

- The runtime stack is an optimization of the activation tree spaghetti stack.
- Most languages use a runtime stack, though certain language features prohibit this optimization.
- Activation records logically store a **control link** to the calling function and an **access link** to the function in which it was created.
- Decaf has the caller manage space for parameters and the callee manage space for its locals and temporaries.
- Call-by-value and call-by-name can be implemented using copying and pointers.
- More advanced parameter passing schemes exist!

Type casting is done in the semantic analysis phase.

For example

float x,y and int z

x=y/z ;

This statement is type casted as

x=y/ (float) z ;

So division by integer will also be detected in the semantic analysis phase.

$LR(0)$ grammars can generate the class of DCFLs having the prefix property (no string being a prefix of any other string). By adding an end string delimiter we can make any DCFL have the prefix property and then it can have an $SLR(1)$ grammar ($SLR(1)$ being a superset of $LR(0)$). So, option A is TRUE.

Option B is FALSE. Only for $LR(0)$, its language is a strict subset of $LR(1)$. For any $k > 1$, language of $LR(k)$ has an equivalent $LR(1)$ grammar.

Reference: https://gatecse.in/lr-parsing-part-2-language-of-ll-and-lr-grammars

Option C is TRUE. Consider the language $L = \{ab, aabb, aab\}$. Is it $LL(1)$? Is it $LL(2)$?

It is actually $LL(2)$ but not $LL(1)$.

- $S \rightarrow aX$
- $X \rightarrow aB \mid b$
- $B \rightarrow b \mid bb$

By seeing the next two symbols, if they are " bb " production $B \rightarrow bb$ will be used or else the production $B \rightarrow b$ will be used.

Option D is TRUE as for any DCFL we do have an $LR(1)$ grammar. (Even SLR(1) grammar provided every string is ended with a delimiter)

Option B.
$\{a^n b^m \mid m \geq n\}$

Here, the language is having strings with number of a's followed by equal or more number of b's. This can be represented by an $LL(1)$ grammar which chooses the production $A \rightarrow aAb$ whenever an a symbol is next and $B \rightarrow bB$ whenever b symbol is next.

- $S \rightarrow AB$
- $A \rightarrow aAb \mid \epsilon$
- $B \rightarrow bB \mid \epsilon$

Answer : A,B
Explanation :

Answer: Here is the first state of the LR(0) machine:

$$
\begin{aligned}
S' &\rightarrow .S \\
S &\rightarrow .Aa \\
S &\rightarrow .Bb \\
A &\rightarrow .Ac \\
A &\rightarrow .\epsilon \\
B &\rightarrow .Bc \\
B &\rightarrow .\epsilon
\end{aligned}
$$

We have that FOLLOW$(A) = \{a, c\}$ and FOLLOW$(B) = \{b, c\}$. We have a reduce-reduce conflict between production 5 ($A \rightarrow \epsilon$) and production 7 ($B \rightarrow \epsilon$), so the grammar is not SLR(1).

Misspelling of keywords mean it will be treated as an identifier by a compiler..As there is only change in spelling of keyword , so it is considered to be a valid token because this satisfies the pattern as needed by dfa of lexical analyser..

It is not a syntax error either as syntax error is something which is related to violation of grammar by any of the statements..

Now say int is mistakeably written as 'inta' ..Now it is identified as an identifier..But its type is not mentioned as there is no "int" in declaration part..Hence it will be a type checking error which is related to semantic phase..

As far as the statement "printf("%d)" is concerned , it will neither give compilation error nor give runtime error. It will give a valid output..For more clarification on this query , plz check the following :

inta , b;

The above is valid syntax as identifier, identifier. But can generate semantic error as type of inta and b may not be known.

inta a;

The above is invalid syntax.

C compiler has two components viz. Front end and back end. Back end is specific for the processor and operating system on which code is meant to run. Hence the code C can be run only on the computers which have the processor and operating system similar to computer Y.

- $FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}$
- $FIRST(E') = \{+, \epsilon\}$
- $FIRST(T') = \{*, \epsilon\}$
- $FOLLOW(E) = FOLLOW(E') = \{), \$\}$
- $FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$
- $FOLLOW(F) = \{+, *,), \$\}$

After FIRST and FOLLOW sets are done, Predictive parsing table is filled as:

For each production $A \rightarrow \alpha$, do the following :

For each terminal 'a' in FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$. (For everything in FIRST(A), $A \rightarrow \alpha$ is a valid parser move)

If ϵ is in $\text{FIRST}(\alpha)$ then for each terminal or end delimiter b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.

Now for the given grammar, for $E \rightarrow TE'$, FIRST(E) has $\{(, id\}$ and so we add $E \rightarrow TE'$ to both $M[E, (]$ and $M[E, id]$.

For $T' \rightarrow \epsilon$, we add $T' \rightarrow \epsilon$ to $M[T', \$]$, $M[T',)]$ and $M[T', +]$ as $FOLLOW(T') = \{+,), \$\}$.

$\therefore M[E, id] = E \rightarrow TE'$, and $M[T', \$] = T' \rightarrow \epsilon$.

Which of the following languages are $LL(1)$ (an $LL(1)$ grammar $DOES$ exist for it)? (Mark all the appropriate choices)

A. $\{a^n ob^n \mid n \geq 1\} \cup \{a^n b^n \mid n \geq 1\}$
B. $\{a^n b^m \mid m \geq n\}$
C. $\{a^n b^m \mid n \geq m\}$
D. $\{a^n b^m \mid n = m\}$

$$
\begin{aligned}
x &::= aT|z \\
Y &::= a|c \\
z &::= bY
\end{aligned}
$$

This grammar satisfies the LL(1) condition because for every non-terminal, all productions have disjoint FIRST sets (and no non-terminals are nullable):