Compiler Construction

Introduction

Overview of Compilation Process

• Lexical Analysis	Front-end of the compiler.
• Syntax Analysis	
• Semantic Analysis	
• Intermediate Code Generation	Back-end of the compiler
• Code Optimisation	
• Code Generation	

Lexical Analysis

- Convert a stream of characters into a stream of ***TOKENS***.
  - Lexeme
    - A token with a value
- Scanner
  - A piece of code that performs lexical analysis.

Regular Expressions

- Use Regex to specify what a token is.
  - Example:
    - NUMBER: [0 - 9]+
    - ID: [a-zA-Z][a-zA-Z0-9]*
    - WHILE: while

Regex Shorthands

[axby]	a or x or b or y
M?	Optional
M+	One or more occurences

“a+*”	Literally a+*
.	Everything except \n
[a-e]	abcde

Disambiguation Rules for Scanners

- 2 rules

1. Longest Match:

> *Longest Substring to match a regex is the choice that is taken.*

2. Rule Priority:

> *If Longest Substring matches many regex’s then the first regex to match the substring is chosen.*

Finite Automota

- Consists of nodes & edges
  - Edges link nodes and have a symbol.
  - Nodes represent states.
- 2 types of Finite Automota:

1. Deterministic Finite Automaton:

> *NO pair of edges leading away from a node share the same symbol.*

2. Nondeterministic Finite Automaton:

> *Two or more edges leading away from a node share the same symbol.*

- Finite Automaton can be encoded by:
  - **Transition matrix:**
    - 2D matrix

```
int edges[][] = {/* ws,..., 0, 1, 2, ...  d, e, f, ...  o, ...  */
     /* state 0 */ { 0,..., 0, 0, 0, ..., 0, 0, 0, ..., 0, ...  },
     /* state 1 */ { 0,..., 7, 7, 7, ..., 2, 4, 4, ..., 4, ...  },
     /* state 2 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 3, ...  },
     /* state 3 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
     /* state 4 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
     /* state 5 */ { 0,..., 6, 6, 6, ..., 0, 0, 0, ..., 0, ...  },
...
}
```

  - **Action array**:
    - Array indexed by final state number.

        - Contains a corressponding action.
- Generally we need to use a NFA to represent Regex expressions.

NFA to DFA

- Unable to execute a NFA
    - Since it "Guesses"
        - However in our case instead of seeing it as choosing one path at random, we view it as choosing all paths concurrently.
- **Subset Construction algorithm**
    - Converts NFA to DFA
    - 2 concepts:
        i. Є - closure(S):
            a. All states can be reached only using Є (epsilon- empty symbol).
        ii. DFAedge(d,c):
            a. All states in d can be reached by only consuming edges labelled with c & Є.
    - On algorithm termination:
        - Discard states array
        - Keep trans array to recognise tokens
        - State i in DFA is final iff states[i] is a final state.
            - Record type of TOKEN the final DFA state recognises
            - *Rule Priority*
        - Generated DFA is flawed:
            - Can merge equivalent states.

Reading Tokens

- JavaCC provide one method:

```
getNextToken()
```

    - This method isn't generally called directly but instead through the parser interface.
- 2 important Token attributes:

1. kind: kind of token
    a. (Example: REAL)
2. image: value of token
    a. (Example: 1234)

Introduction to Parsers

Adding Recursion

- Regex cannot count.
- Therefore we use mutual recursion

Examples:

```
expr = a b(c | d)e
// Is now written as
aux = c
aux = d
expr = a b aux e
```

```
expr = (a b c)*
// Is now written as
expr = (a b c) expr
expr = Є
```

- Simplified notation above is ***Context Free Grammar***

Context Free Grammars(CFGs)

- G is a 4-tuple (*Vt, Vn, S, P*)
  - Vt
    - Set of terminals (a, b, c,... Є Vt)
  - Vn
    - Set of nonterminals (A,B,C,... Є Vn)
  - V
    - (Vt Union Vn) aka vocabulary of G
  - S
    - Goal symbol
    - Distinguished nonterminal (S Є Vn)
  - P
    - Finite set of productions stating how terminals and non terminals can be combined
    - Rules
  - V*

        - α, β, γ,... Є V*
    - V*t
        - u, v, w,... Є V*t

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is a *single-step derivation* using $A \rightarrow \gamma$

Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps.

If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G.

$L(G) = \{w \in V_t^* | S \Rightarrow^+ w\}, w \in L(G)$ is called a *sentence* of G.

Note, $L(G) = \{\beta \in V^* | S \Rightarrow^* \beta\} \cap V_t^*$

Backus-Naur form (BNF)

- Grammars are written in this form of notation.

1. Non-terminals have angled brackets or capital letters.
2. Terminals normal font or underlined
3. Production (Each possible image of <expr>, line 2,3,4 are all separate productions)

1	<goal>	::=	<expr>
2	<expr>	::=	<expr><op><expr>
3		|	num
4		|	id
5	<op>	::=	+
6	<op>	::=	−
7	<op>	::=	*
8	<op>	::=	/

Derivations

- The resulting sequence of grammar rules after consuming all the characters.
    - Sequence of production applications is a ***derivation*** or a ***parse***.
    - **Parsing** is the process of discovering the *derivation*.

2 types of derivations:

> *Both examples use the previous CFG and the following input “x + 2*y”*

1. Left most derivation

```
<goal>  ⇒  <expr>
        ⇒  <expr><op><expr>
        ⇒  <expr><op><expr><op><expr>
        ⇒  <id,x><op><expr><op><expr>
        ⇒  <id,x> + <expr><op><expr>
        ⇒  <id,x> + <num,2><op><expr>
        ⇒  <id,x> + <num,2>*<expr>
        ⇒  <id,x> + <num,2>*<id,y>
```

2. Right most derivation

```
<goal>  ⇒  <expr>
        ⇒  <expr><op><expr>
        ⇒  <expr><op><id,y>
        ⇒  <expr>*<id,y>
        ⇒  <expr><op><expr>*<id,y>
        ⇒  <expr><op><num,2>*<id,y>
        ⇒  <expr> + <num,2>*<id,y>
        ⇒  <id,x> + <num,2>*<id,y>
```

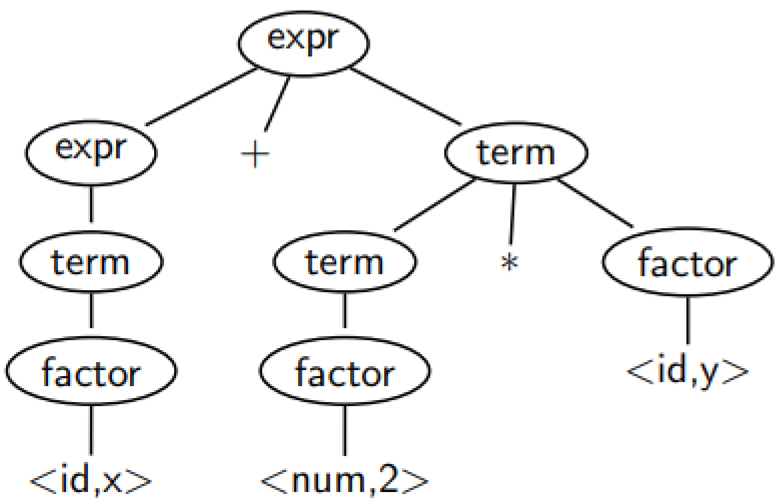
- Each derivation creates a different tree and therefore evalutates the input differently.
  - In the cases above:
    - left most derivation would result in x + (2*y)
    - right most derivation would result in (x+2) * y

Precedence

- Can add additional structure to create precedence.
- In the following example:
  - *terms* **must** be derived from *expr*
  - *factors* **must** be derived by *terms*

```
1 | <goal>    ::= <expr>
2 | <expr>    ::= <expr> + <term>
3 |             |  <expr> - <term>
4 |             |  <term>
5 | <term>    ::= <term> * <factor>
6 |             |  <term> / <factor>
7 |             |  <factor>
8 | <factor>  ::= num
9 |             |  id
```

- Applying this new structure corrects the tree thats generated.



Ambiguity

- A grammar is ambiguous if a sentence has 2 or more derivations
- To solve this issue just simply split the sentence into further sentences.

Top-down Parsing

- Starts with the root of the parse tree
  - Labelled with goal symbol for grammar

- Do the following steps until “fringe” of the parse tree matches input string.

1. At node A, select production A → α, construct the children for each member of α.
2. If a terminal is added to the fringe that doesn’t match input string, backtrack.
3. Find next node to be expanded.

Left Recursion

- Top-down parsers **CANNOT** handle left-recursion.
- To eliminate left-recursion follow the rules below.

$$\begin{array}{lll} A & ::= & A\alpha \\ & \mid & \beta \end{array}$$

Left Recursive Grammar

$$\begin{array}{lll} A & ::= & \beta A' \\ A' & ::= & \alpha A' \\ & \mid & \epsilon \end{array}$$

Introduction of A’ to remove left recursion.

Lookahead

- Further we look ahead in the input stream the easier it is to see what rules work.
  - Allows us to choose the right rules first time.
- CFG subclasses:
  - LL(1)
    - Left to right scan, Left-most derivation, look ahead of 1 token
  - LR(1)
    - Left to right scan, Right-most derivation, look ahead of 1 token

Predictive Parsing

- For any 2 productions of A → α|β
  - We’d like to choose the correct production.
  - FIRST(α)
    - The set of tokens that appear in some string derived from α.
- Note:

- If two productions A → α & A → β are in a grammar the following should happen.

$$\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) = \emptyset$$

  - This allows the parser to make the correct choice when lookahead == 1.

Left Factoring

- If a productions in a grammar all have the same left most non terminal.
  - Which is NOT Є

$$A \to \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_n$$

- We can left factor it to look like:

$$A \to \alpha A'$$
$$A' \to \beta_1|\beta_2|...|\beta_n$$

Indirect Left-recursion

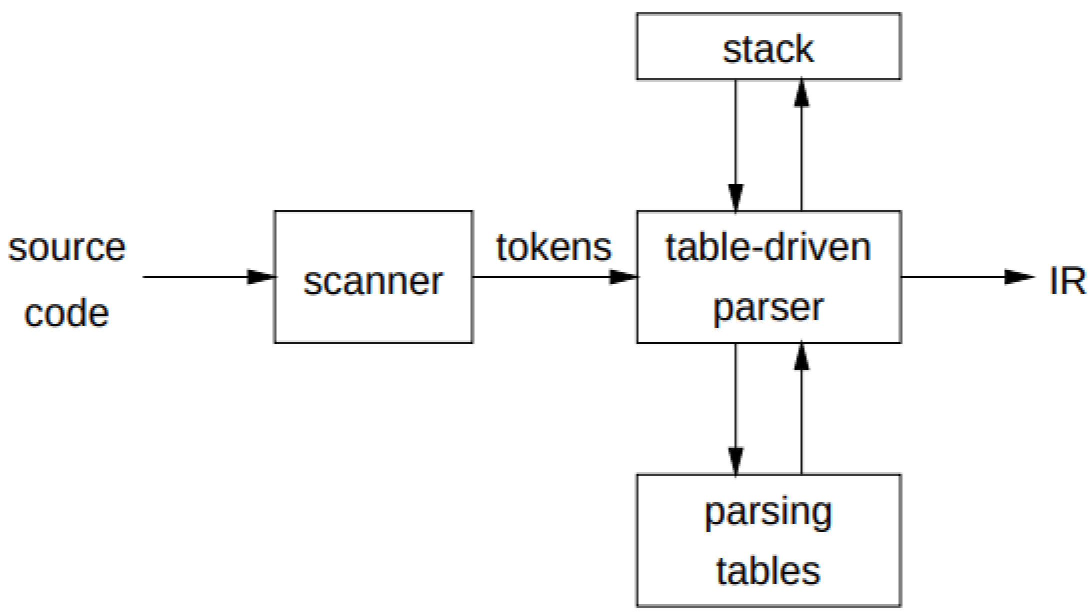
- Basically an infinite loop caused by a production outside of a rule.

$$A_0 \to A_1\alpha_1| \ldots$$
$$A_1 \to A_2\alpha_2| \ldots$$
$$\ldots$$
$$A_n \to A_0\alpha_{n+1}| \ldots$$

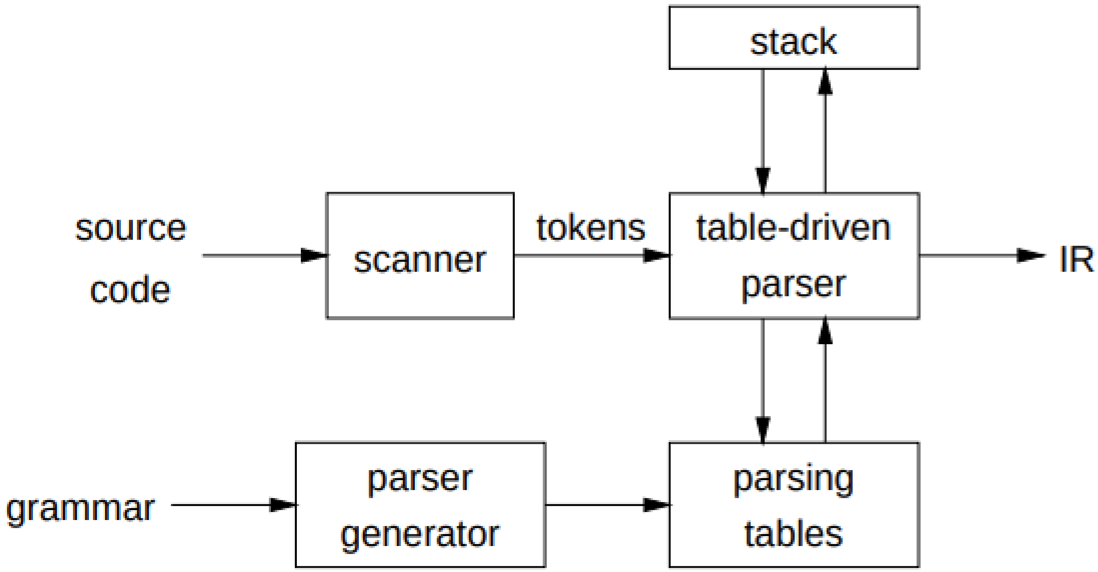
- Fix for this is to remove the additional step required and have one of the rules directly point to whats needed?

- For each non-terminal A_i in turn, do:
  - For each A_j such that $1 \leq j < i$ and there is a production rule of the form $A_i \to A_j\alpha$, where the A_j productions are $A_j \to \beta_1 \mid \ldots \mid \beta_n$:
    - Replace the production rule $A_i \to A_j\alpha$ with the rule $A_i \to \beta_1\alpha \mid \ldots \mid \beta_n\alpha$.

Table-Driven Parsing



- Generation of tables can be automated.



- An expression grammar and its parse table

1	<goal>	::=	<expr>
2	<expr>	::=	<term> <expr′>
3	<expr′>	::=	+ <expr>
4		|	- <expr>
5		|	ϵ
6	<term>	::=	<factor> <term′>
7	<term′>	::=	* <term>
8		|	/ <term>
9		|	ϵ
10	<factor>	::=	num
11		|	id

	id	num	+	-	*	/	$
<goal>	1	1	–	–	–	–	–
<expr>	2	2	–	–	–	–	–
<expr′>	–	–	3	4	–	–	5
<term>	6	6	–	–	–	–	–
<term′>	–	–	9	9	7	8	9
<factor>	11	10	–	–	–	–	–

- To generate this table we use FIRST FOLLOW & LOOKAHEAD

FIRST

- FIRST(α)
  - Set of terminal symbols

- {a Є Vt| a →* αβ}
- α →* Є then Є Є FIRST(α)
  - Read as alpha consists of 0 or more epsilons then epsilon is the set for FIRST(α).

- Compute FIRST(X) rules:

1. If X is a terminal, then FIRST(X) = {X}
2. If X is a nonterminal and X → Y1Y2...Yk is a production rule for some k >= 1, then add a to FIRST(X) if for some i, a Є FIRST(Yi) and Y1...Yi-1 are nullable(empty strings). If all Y1...Yk are nullable, add Є to FIRST(X).
3. If X → Є is a production rule, then add Є to FIRST(X).

FOLLOW

- FOLLOW(X) is set of terminals that can immediately follow X.
- Compute FOLLOW(A) rules:

1. Place $ in FOLLOW(S), S being start symbol and $ being end of input.
2. If a production rule of A → α B β, then everything in FIRST (β), except Є, is in FOLLOW(B).
3. If a production rule A → α B or A → α B β where FIRST(β) contains Є, then everything in FOLLOW(A) is in FOLLOW(B).

> *6 Examples are demonstrated in the video below.*

https://youtu.be/_uSIP91jmTM

LOOKAHEAD

- LOOKAHEAD(A → α)
  - Set of terminals which can appear in the next input.
- Build LOOKAHEAD(A → α)

1. Put FIRST(α) - {Є} in LOOKAHEAD(A →α)
2. If Є FIRST(α) then put FOLLOW(A) in LOOKAHEAD(A →α)

LL(1) parse table construction

- Input: Grammar G
- Output: Parsing table M
- Method:

1. For all productions A → α:
   For all a Є LOOKAHEAD(A → α), add A→α to M[A, a]
2. Set each undefined entry of M to ***error***

- If a cell has more than 1 entry in M
  - **NOT** LL(1) grammar

LL(1) Grammar facts that arent fun

1. No left recursion
2. No ambiguous grammar
3. Some languages have no LL(1) grammar
4. If each alternative expansion for A begins with a distinct terminal & is Є-free
   a. Then its a LL(1) grammar.

Example:

$$S \to aS|a$$

is not LL(1) because

$$\texttt{LOOKAHEAD}(S \to aS) = \texttt{LOOKAHEAD}(S \to a) = \{a\}$$

$$S \to aS'$$
$$S' \to aS'|\epsilon$$

accepts the same language and is LL(1).

Error Recovery

- For each non terminal
  - Construct a set of terminals that the parser can synch.
- If an error occurs looking for A
  - Scan until an element of SYNCH(A) is found.
- Building SYNCH:
  - a Є FOLLOW(A) → a Є SYNCH(A)
  - Place keywords that start statements in SYNCH(A)
  - Add symbols in FIRTS(A) to SYNCH(A)
- If a terminal can't be matched to top of stack:

- Pop it
- Print saying it was inserted
- Continue to parse