Chapter 2 (Scott) Programming Language Syntax

Programming Languages

Lexical and Syntactic Analysis
• Chomsky Grammar Hierarchy
• Lexical Analysis – Tokenizing
• Syntactic Analysis – Parsing

Lexical analysis is called "scanning" in your text.
Syntax

- The *syntax* of a programming language is a precise description of all its grammatically correct programs.
- Precise syntax was first used with Algol 60, and has been used ever since.
- Three levels:
  - Lexical syntax - all the basic symbols of the language (names, values, operators, etc.)
  - Concrete syntax - rules for writing expressions, statements and programs.
  - Abstract syntax – internal (tree) representation of the program in terms of categories like “statement”, “expression”

Dictionary Moment

- A *metalanguage* is a language used to make statements about statements in another language (the object language).

- **Meta-advertising** refers to a hybrid form of advertising, where the advertiser advertises for an advertisement
- **A Meta-critic** evaluates critics.
Grammars
Grammars: Metalanguages used to define the concrete syntax of a language.

Backus Normal Form – Backus Naur Form (BNF) [John Backus and Peter Naur]
• First used to define syntax of Algol 60
• Now used to define syntax of most major languages

Production:
\[ \alpha \rightarrow \beta \]
\[ \alpha \in \text{Nonterminal} \]
\[ \beta \in (\text{Nonterminal} \cup \text{Terminal})^* \]

ie, lefthand side is a single nonterminal, and \( \beta \) is a string of nonterminal and/or terminals (possibly empty).

• Example
  \[ \text{Integer} \rightarrow \text{Digit} \mid \text{Integer Digit} \]
  \[ \text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Chomsky Hierarchy – different powers of grammars
• Regular grammar – used for tokenizing
• Context-free grammar (BNF) – used for parsing
• Context-sensitive grammar – not really used for programming languages
Lexical Analysis: converting a sequence of characters into a sequence of tokens

- Why split lexical analysis from parsing?
  - Simplifies design
    - Parsers with whitespace and comments are more awkward
  - Efficiency
    - Only use the most powerful technique that works and nothing more
  - Portability
    - More modular code
    - More code re-use
Source Code Characteristics

• Code
  – Identifiers
    • Count, max, get_num
  – Language keywords: reserved (cannot be re-defined by user) or predefined (has prior meaning)
    • switch, if .. then.. else, printf, return, void
    • Mathematical operators
      – +, *, >> ....
      – <=, =, != ...
  – Literals
    • “Hello World”  2.6  -14e65
• Comments – ignored by compiler
• Whitespace

Reserved words versus predefined identifiers

• Reserved words cannot be used as an identifier.
• Predefined identifiers have special meanings, but can be redefined (although they probably shouldn’t be).
• Examples of predefined identifiers in Ruby:
  ARGV, STRERR, TRUE, FALSE, NIL.
• Examples of reserved words in Ruby: alias and BEGIN
  begin break case class def defined? do else elsif END end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield
• So TRUE has value true, but can be redefined. true cannot be redefined.
Terminology of Lexical Analysis

**Tokens**: category - Often implemented as an enumerated type, e.g. IDENTIFIER (value 20)

**Patterns**: regular expression which defines the token

**Lexemes**: actual string matched (captured)

Knowing Tokens are not enough...

- In the case of “switch” – the token is enough.
- In the case of “identifier” We need to know WHICH identifier.
- Such “other” data items are attributes of the tokens
Token delimiters

• When does a token/lexeme end?

e.g xtemp=-ytemp

Ambiguity in identifying tokens

• A programming language definition will state how to resolve uncertain token assignment

• <> Is it 1 or 2 tokens?

• If a token could be a reserved word or a identifier, Reserved keywords is assumed (e.g. “if”).

• Disambiguating rules state what to do if two choices.

• ‘Principle of longest substring’: greedy match (xtemp == 64.23)
Lexical Analysis

• How sophisticated is the process?
  – How do I describe the rules?
  – How do I find the tokens
  – What do I do with ambiguities: is ++ one symbol or two? Depends on my language
  – Does it take lots of lookahead/backtracking?
    Ex: no spaces
    • whileamerica>china
    • whileamerica==china
    When do I know how to interpret statement?

Wouldn’t it be nice?

Wouldn’t it be nice if we could specify how to do lexical analysis by providing:

• Here are the regular expressions which are associated with these tokens names. Please tell me the lexeme and token.

And by the way…

• Always match the longest string
• Among rules that match the same number of characters, use the one which I listed first.

This is what a program like lex/flex does.
In JavaCC (a commercial parser generator), you specify your tokens using regular expressions: Give a name and the regular expression.
System gives you a list of tokens and lexemes

<DIGIT> is a built-in class

Matching Regular Expressions
Try it – write a recognizer for (a|b)*bac

• Assume you have a string s which you want to see if the prefix matches (a|b)*bac.
• If so, return the part of s that matched
• How would you do it in C++?
• We want a “graph” of nodes and edges
   Nodes represent “states” such as “I have read ba”
   Arcs represent – input which causes you to go from
   one state to the next
   Some nodes are denoted as “final states” or “accept
   states” to mean the desired token is recognized
   Our states don’t really need to have names, but we
   might assign a name to them.

If we had a graph for (a|b)*bac
would that help you code?

accept state
Deterministic Finite Automata (DFA)

- A recognizer determines if an input string is a sentence in a language
- Uses regular expressions
- Turn the regular expression into a finite automaton
- Could be deterministic or non-deterministic

Transition diagram for identifiers
Notice the accept state means you have matched a token

- RE
  - Identifier -> letter (letter | digit)*

```
start 0 letter 1 letter accept
      digit
```
Visit with your neighbor

• Draw a graph to recognize $[hc]+a^*$

Visit with your neighbor

• Write a finite state machine which reads in zeroes and ones and accepts the string if there are an even number of zeroes.

• If you can find such a FSM, we say the FSM “recognizes” the language of strings of 0/1 such that there are an even number of zeroes.
Note it does the minimal work. It doesn’t count, just keeps track of odd and even

What does this match?
visit with your neighbor

- Write a finite state machine (FSM) aka finite state automata (FSA) which accepts a string with the same number of zeroes and ones.

- Is this easier if we know there will be at most 10 zeroes?

What does this match? How does this differ from the other kinds of FSA we have seen?
Thought question

• Does the non-deterministic FSA have more power than a FSA? In other words, are there languages (sets of strings) that a NFSA can “recognize” (say yes or no) than a FSA can’t make a decision about?

An NFA (nondeterministic finite automaton) is similar to a DFA but it also permits multiple transitions over the same character and transitions over $\epsilon$ (empty string). The $\epsilon$ transition doesn’t consume any input characters, so you may jump to another state “for free”.

• When we are at a state and we read a character for which we have more than one choice; the NFA succeeds if at least one of these choices succeeds.

• Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power.
Automating Scanner (lexical analyzer) Construction

To convert a specification into code:
1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators
- Lex and Flex work along these lines
- Algorithms are well-known and well-understood

From a Regular Expression to an NFA
Thompson’s Construction

\[(a \mid b)^* abb\]
**RE → NFA using Thompson’s Construction rules**

Key idea
- NFA pattern for each symbol & each operator
- Join them with $\varepsilon$ moves in precedence order

Example of Thompson’s Construction
Let’s try $a \ (b \ | \ c)^*$

1. $a$, $b$, & $c$

2. $b \ | \ c$

3. $(b \ | \ c)^*$
4. $a \ (b \mid c)^*$

5. Of course, a human would design something simpler ...

But, automatic production of first one is easier.

DFA for each token can easily be combined with epsilon transitions
Now need to convert to DFA (deterministic finite automata)

To convert to DFA: Ask, "How would you use a NFA?"

Non-deterministic finite state automata NFA

Equivalent deterministic finite state automata DFA
NFA → DFA (subset construction)

Define two functions:

1. The $\varepsilon$-closure function takes a state and returns the set of states reachable from it, based on (zero or more) $\varepsilon$-transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its $\varepsilon$-closure without consuming any input.

2. The function move takes a state and a character, and returns the set of states reachable by one transition on this character.

- We can generalize both these functions to apply to sets of states by taking the union of the application to individual states.

- Eg. If A, B and C are states, move({A,B,C},`a') = move(A,`a') $\cup$ move(B,`a') $\cup$ move(C,`a').

NFA -> DFA (cont)

- The Subset Construction Algorithm

The vision

1. States in DFA are sets of NFA states.
2. Create the start state of the DFA by taking the $\varepsilon$-closure of the start state of the NFA.
3. Perform the following for the new DFA state:
   - For each possible input symbol:
     - Apply move to the newly-created state and the input symbol; this will return a set of states.
     - Apply the $\varepsilon$-closure to this set of states, possibly resulting in a new set.
4. This set of NFA states will be a single state in the DFA.
5. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
6. The finish states of the DFA are those which contain any of the finish states of the NFA.
Use algorithm to transform

Non-deterministic finite state automata NFA

Equivalent deterministic finite state automata DFA

Transition Table (DFA)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>02</td>
<td>01</td>
<td>03</td>
</tr>
<tr>
<td>03</td>
<td>01</td>
<td>0</td>
</tr>
</tbody>
</table>
Writing a lexical analyzer

- The DFA helps us to write the scanner so that we recognize a symbol as early as possible without much lookahead.

Lexical Errors

- Only a small percentage of errors can be recognized during Lexical Analysis

Consider if (good == !“bad”)
Which of the following errors could be found during lexical analysis?

– Line ends inside literal string
– Illegal character in input file
– calling an undefined method
– missing operator
– missing paren
– unquoted string
– using an unopened file

Dealing with errors…

• What does a lexical error mean?
• Strategies for dealing with lexical error:
  – “Panic-mode”
    • Delete chars from input until something matches
  – Inserting characters
  – Re-ordering characters
  – Replacing characters
• For an error like “illegal character” then we should report it sensibly
Limitations of REs

• REs can describe many languages but not all

• In regular expression lingo, a language is a set of strings which are acceptable. Alphabet is set of tokens of the language.

• For example: Alphabet = \{a, b\}, My language is the set of strings consisting of a single a in the middle of an equal number of b’s

  \[S = \{a, bab, bbabb, bbbabbb, \ldots\}\]

  Can you represent the set of legal strings using RE?

With your neighbor try

• With RE can you represent the language of correctly nested grouping symbols \{ \}( )[ ] ?
  
  – \{\{[]\}\} is part of the language
  
  – \{(()))))\} is NOT part of the language
  
  – We say the RE accept, match, or recognize strings from the language.
Dictionary Moment

- **parse**: to analyze (a sentence) in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc

- example parse: *I have a picture of Joe at my house.*

Syntax Analysis

- also known as Parsing
- Grouping together tokens into larger structures

- **Input**:
  - Tokens (output of Lexical Analyzer)

- **Output**:
  - Structured representation of original program
A Context Free Grammar

- A grammar is a four tuple \((\Sigma, N, P, S)\) where
- \(\Sigma\) is the terminal alphabet
- \(N\) is the non terminal alphabet
- \(P\) is the set of productions
- \(S\) is a designated start symbol in \(N\)

Example CFG

- \(S \rightarrow \{S\}\)
- \(S \rightarrow [S]\)
- \(S \rightarrow (S)\)
- \(S \rightarrow SS\)
- \(S \rightarrow () | {} | []\)  // | means or

- Identify \((\Sigma, N, P, S)\)
- Sometimes tricky to distinguish between alphabet and grammar symbols
Using the grammar

derive
• {}({})
• ({{}{}})

• Can show derivation as a tree or as a string of sentential forms.

Slightly different version

• S → {S}
• S → [S]
• S → (S)
• S → SS
• S → ε // means empty string

• Identify (∑, N,P,S)
Parsing

- Example: define expression as series of added operands
- Basic need: a way to communicate what is allowed in a way that is easy to translate.

- Expression → number plus Expression | number
  
  - Similar to normal definitions:
    - Concatenation
    - Choice
    - No * – repetition by recursion
Now we’ll focus on the internal parse tree – which is the way we show structure.

**Parse Trees**

\[
\text{Integer} \rightarrow \text{Digit} \mid \text{Integer Digit} \\
\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

**Parse Tree for 352 as an Integer**

Explain the relationship between grammar and tree.

How did we know how to create the tree?
Arithmetic Expression Grammar

Expr → Expr + Term | Expr – Term | Term
Term → 0 | ... | 9 | ( Expr )

Parse of 5 - 4 + 3

Is there another possible parse tree?

Derivations

• Derivation:
  – Sequence of replacements of structure names by choices on the RHS of grammar rules
  – Begin: start symbol
  – End: string of token symbols
  – Each step - one replacement is made
Arithmetic Expression Grammar

Expr → Expr + Term | Expr – Term | Term
Term → 0 | ... | 9 | ( Expr )

Expr ⇒ Expr + Term ⇒ Expr – Term + Term
⇒ Term – Term + Term ⇒ 5 – Term + Term
⇒ 5–4+Term ⇒ 5–4+3

Terminals:

Derive: a–b*c+(b–c)

E → E + Term | E - Term | Term
Term → Term * Factor | Term/Factor | Factor
Factor → ( exp ) | number | id
id -> a|b|c|d

Note the different arrows:

⇒ Derivation applies grammar rules
→ Used to define grammar rules

Non-terminals: Exp, Op
Terminals: number, *

Terminals: because they terminate the derivation
• $E \rightarrow (E) \mid a$

• What **language** does this grammar generate?

An example derivation:
• $E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a))$
• Note that this is what we couldn’t achieve with regular expressions

---

**Recursive Grammars**

– Create a grammar to generate $a^n b^n$
– Create a grammar to generate $ab^n$
– Create a grammar to generate palindromes of $\Sigma = \{a, b\}$
  
  abbaaaabba  aba  aabaa
– Create a grammar to generate $a^n b^n c^n$
What does this generate?

1. $E \rightarrow E \alpha \mid \beta$

2. $S \rightarrow aB \mid bA$
   
   $A \rightarrow aA$  
   $B \rightarrow bB$

What does this generate?

- $E \rightarrow E \alpha \mid \beta$
  
  - derives $\beta, \beta \alpha, \beta \alpha \alpha, \beta \alpha \alpha \alpha, \beta \alpha \alpha \alpha \alpha, \ldots$
  
  - All strings beginning with $\beta$ followed by zero or more repetitions of $\alpha$
    
    - $\beta \alpha^*$
Write a context free grammar for

1. \( L = \{ a^m b^n c^k \}, \) where \( k = m + n \)

2. \( (a(ab)^*b)^* \)

- Do we expect that a CFG can match any regular expression? (True, if it is no less powerful)

---

Abstract Syntax Trees

Parse trees contain surplus information

Parse Tree – shows all productions

```
exp
  /\   /
exp op exp
```

Abstract Syntax Tree

```
+  3  4
   /\   /
   3  4
```

This is all the information we actually need.
An exercise

• Consider the grammar
  \[ S \rightarrow (L) | a \]
  \[ L \rightarrow L, S | S \]

(a) What are the terminals, non-terminals and start symbol?
(b) Find leftmost and rightmost derivations and parse trees for the following sentences
   i. \((a, a)\)
   ii. \((a, (a, a))\)
   iii. \((a, ((a, a), (a, a)))\)

Ambiguity

• If the same sentence has two distinct parse trees, the grammar is ambiguous.
• In English, the phrase “small dogs and cats” is ambiguous as we aren’t sure if the cats are small or not.
• ‘I have a picture of Joe at my house’ is also ambiguous.
• A language is said to be ambiguous if no unambiguous grammar exists for it.
  For example, English is ambiguous because for the sentence “I have a picture of Joe at my house” either interpretation is correct, grammatically.
Ambiguous Grammars

- Problem – no unique structure is expressed
- A grammar that generates a string with 2 distinct parse trees is called an ambiguous grammar

1) $5-4+2 = 5-(4+3)$
2) $5-4+2 = (5-4)+3$

- How does the grammar relate to meaning?
- Our experience of math says interpretation (2) is correct but the grammar below does not express this preference:
  $E \rightarrow E + E | E * E | (E) | -E | id$

Ambiguous Parse of $5-4+3$
Ambiguity is bad!!!!

- Goal – rewrite the grammar so you can never get two trees for the same input.
- If you can do that, PROBLEM SOLVED!!!
- The grammar becomes the precise definition of the language.
- We won’t spend time studying that, but realize that is usually the goal.

Dangling Else Ambiguous Grammars

IfStatement $\rightarrow$ if (Expression) Statement |
                    if (Expression) Statement else Statement
Statement $\rightarrow$ Assignment | IfStatement | Block
Block $\rightarrow$ { Statements }
Statements $\rightarrow$ Statements Statement | Statement

With which ‘if’ does the following ‘else’ associate

if (x < 0)
  if (y < 0) y = y - 1;
else y = 0;
Dangling Else Ambiguous Grammars

Show that this grammar is ambiguous.

- Consider the grammar
- \( S \rightarrow aS \mid aSbS \mid \varepsilon \)

How do you do that?
- Think of ONE SENTENCE that you think is problematic.
- Show TWO different parse trees for the sentence.
Is this grammar ambiguous?

- $S \rightarrow A1B$
- $A \rightarrow 0A|\varepsilon$
- $B \rightarrow 0B|1B|\varepsilon$

Precedence

$E \rightarrow E + Term | E - Term | Term$

$Term \rightarrow Term * Factor | Term/Factor | Factor$

$Factor \rightarrow (\ exp\ ) | number | id$

$id \rightarrow a|b|c|d$

- Operators of equal precedence are grouped together at the same ‘level’ of the grammar $\rightarrow$ 'precedence cascade’
- The lowest level operators have highest precedence
  (The first shall be last and the last shall be first.)
Precedence

- Using the previous grammar, show a parse tree for
  - $a + b \times c$ (notice how precedence is manifested)
  - $a \times b + c$ (notice how precedence is manifested)
  - $a - b - c$ (notice how associativity is manifested)

Associativity

- $45 - 10 - 5$? 30 or 40 Subtraction is left associative, left to right (=30)

- $E \rightarrow E + E | E - E | Term$
  
  Does not tell us how to split up $45 - 10 - 5$ (ambiguity)

- $E \rightarrow E + Term | E - Term | Term$
  
  Forces left associativity via left recursion

- $E \rightarrow Term + E | Term - E | Term$
  
  Forces right associativity via right recursion

- Precedence & associativity remove ambiguity of arithmetic expressions
  - Which is what our math teachers took years telling us!
Review: Precedence and Associativity

In the grammar below, where would you put unary negation and exponentiation? Precedence? Associativity?

\[ E \rightarrow E + \text{Term} \mid E - \text{Term} \mid \text{Term} \]
\[ \text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Term}/\text{Factor} \mid \text{Factor} \]
\[ \text{Factor} \rightarrow (\exp) \mid \text{number} \mid \text{id} \]
Try parsing $-2^2^3 \cdot -(a+b)$

Is this right?

$$E \rightarrow E + \text{Term} \mid E - \text{Term} \mid \text{Term}$$

$$\text{Term} \rightarrow \text{Term} \times \text{Factor} \mid \text{Term/Factor} \mid \text{Factor}$$

$$\text{Factor} \rightarrow -\text{Factor} \mid \text{Item}$$

$$\text{Item} \rightarrow \text{Part} \wedge \text{Item} \mid \text{Part}$$

$$\text{Part} \rightarrow (\exp) \mid \text{number} \mid \text{id}$$

Extended BNF (EBNF)

Additional metacharacters

- $(\ldots)^*$ a series of zero or more
- $(\ldots)^+$ a series of one or more
- $[\ldots]$ optional

EBNF is no more powerful than BNF, but its production rules are often simpler and clearer.
Syntax (railroad) Diagrams

- An alternative to EBNF.
- Rarely seen any more: EBNF is much more compact.
Formal Methods of Describing Syntax

1950: Noam Chomsky (American linguist, MIT) described generative devices which describe four classes of languages (in order of decreasing power)

1. recursively enumerable $x \rightarrow y$ where $x$ and $y$ can be any string of nonterminals and terminals.
2. context-sensitive $x \rightarrow y$ where $x$ and $y$ can be string of terminals and non-terminals but $y \geq x$ in length.
   - Can recognize $a^nb^nc^n$
3. context-free - nonterminals appear singly on left-side of productions. Any nonterminal can be replaced by its right hand side regardless of the context it appears in.
   - Ex: A Condition in a while statement is treated the same as a condition in an assignment or in an if, this is context free as context does not matter.
4. regular (only allow productions of the form $N \rightarrow aT$ or $N \rightarrow a$)
   - Can recognize $a^n b^m$

Chomsky was interested in the theoretic nature of natural languages.

Regular Grammar (like regular expression)

- A CFG is called regular if every
- production has one of the forms below
  $A \rightarrow aB$
  $A \rightarrow \epsilon$
Write the following regular expressions as regular grammars

Example 1: a* b

Example 2: a+b*

Context Sensitive

- Allows for left hand side to be more than just a single non-terminal. It allows for “context”
- **Context-sensitive**: context dependent
  Rules have the form xYz->xuz with Y being a non-terminal and x,u,z being terminals or non-terminals.
- Important for a natural language like English, but almost never used in programming languages.
Consider the following context sensitive grammar
G=({S,B,C}, {x,y,z}, P, S) where P are
1. S → xSBC
2. S → xyC
3. CB → BC
4. yB → yy
5. yC → yz
6. zC → zz

At seats, what strings does this generate?

1. S → xSBC
2. S → xyC
3. CB → BC No ambiguity as doesn’t go both ways
4. yB → yy
5. yC → yz
6. zC → zz Do have backtracking

S → xSBC → xxSBCBC → xxyCBCBC
   → xxyzBCBC OOPS, stuck

S → xSBC ⇒ x xSBCBC ⇒ xxyCBCBC
   ⇒ xxyyBCBC ⇒ xxyyBCBC ⇒ xxyyBBCC
   ⇒ xxyyBCCC ⇒ xxyyyyCC ⇒ xxyyyzCC
   ⇒ xxyyyzzC ⇒ xxyyyzzz
How is Parsing done?

1. Recursive descent (top down). I have symbol S and the first thing I want to generate is x. What production should I use?

2. Bottom up – I see something like looks like the right hand side of a rule. Let me replace it.
Sometimes called shift-reduce parsers.
(inside out)

Parsing is the subject of CS5300 (compilers).
Your text has a lot of information about parsing if you are interested.

Could we make the parsing so easy a simple machine can do it?

Table Driven Predictive Parsing

Parse: id + id * id

Leftmost derivation and parse tree using the grammar

E -> TP
P -> +TP | ε
T -> FU
U -> *FU | ε
F -> (E) | id
Parse: \( id + id \ast id \)

Leftmost derivation and parse tree using the grammar

\[
E \rightarrow TP \\
P \rightarrow +TP \mid \varepsilon \\
T \rightarrow FU \\
U \rightarrow *FU \mid \varepsilon \\
F \rightarrow (E) \mid id
\]

\[
E \Rightarrow TP \Rightarrow FUP \Rightarrow idUP \text{ (clear as don't want paren)} \\
\Rightarrow idP \text{ (clear as don't want *)} \\
\Rightarrow id +TP \text{ (clear as not ready to end string)} \Rightarrow id +FUP \\
\Rightarrow id +id UP \Rightarrow id +id UP \Rightarrow id +id *FUP \Rightarrow id +id *idUP \\
\Rightarrow id +id *idP \Rightarrow id +id *id
\]

Table Driven Predictive Parsing

Input \( a + b \) $ \\
\begin{array}{c}
X \\
Y \\
Z \\
$
\end{array}

Stack

Partial derivation Not yet matched

Output

Predicate Parsing Program

Parsing Table
Table Driven Predictive Parsing

<table>
<thead>
<tr>
<th>Non Terminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>E -&gt; TP</td>
</tr>
<tr>
<td>P</td>
<td>P -&gt; + TP</td>
</tr>
<tr>
<td>T</td>
<td>T -&gt; FU</td>
</tr>
<tr>
<td>U</td>
<td>U -&gt; ε</td>
</tr>
<tr>
<td>F</td>
<td>F -&gt; id</td>
</tr>
</tbody>
</table>

Use parse table to derive:  $a^*(a+a)^*a$  ($$ indicates end)
Notice we have only ONE choice for each cell in the table.

Rules for First Sets

- If $X$ is a terminal then $\text{First}(X)$ is just $X$
- If there is a Production $X \rightarrow \varepsilon$ then add $\varepsilon$ to $\text{first}(X)$
- If there is a Production $X \rightarrow Y_1Y_2..Y_k$ then add $\text{first}(Y_1Y_2..Y_k)$ to $\text{first}(X)$
- $\text{First}(Y_1Y_2..Y_k)$ is either
  - $\text{First}(Y_1)$ (if $\text{First}(Y_1)$ doesn't contain $\varepsilon$)
  - OR (if $\text{First}(Y_1)$ does contain $\varepsilon$) then $\text{First}(Y_1Y_2..Y_k)$ is everything in $\text{First}(Y_1)$ except for $\varepsilon$ as well as everything in $\text{First}(Y_2..Y_k)$
  - If $\text{First}(Y_1)$ $\text{First}(Y_2)..<\text{First}(Y_k)$ all contain $\varepsilon$ then add $\varepsilon$ to $\text{First}(Y_1Y_2..Y_k)$ as well.
Rules for Follow Sets

• First put $ (the end of input marker) in Follow(S) (S is the start symbol)
• If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).
• If there is a production A → aB, then put everything in FOLLOW(A) in FOLLOW(B)
• If there is a production A → aBb, where FIRST(b) contains ε, then put everything in FOLLOW(A) in FOLLOW(B)

Parse Table construction

• For each production A → α do
  1. For each terminal a in first(α) Add A → α to M[A,a]
  2. if ε in first(α) add Add A → α to M[A,b] for each terminal b in follow(A)
  3. if ε in first(α) and $ in follow(A), Add A → α to M[A, $]
  4. Set each undefined entry of M to error
Example

1. $E \rightarrow TE'$
2. $E' \rightarrow +E$
3. $E' \rightarrow -E$
4. $E' \rightarrow \varepsilon$
5. $T \rightarrow FT'$
6. $T' \rightarrow *T$
7. $T' \rightarrow /T$
8. $T' \rightarrow \varepsilon$
9. $F \rightarrow \text{num}$
10. $F \rightarrow \text{id}$

Example

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id num</td>
<td>$$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$+\varepsilon$</td>
<td>$$</td>
</tr>
<tr>
<td>$T$</td>
<td>id num</td>
<td>$+$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$ / *</td>
<td>$+$</td>
</tr>
<tr>
<td>$F$</td>
<td>id num</td>
<td>$/ + $</td>
</tr>
</tbody>
</table>

101

102
Predictive Parsing

- Which rule to use?
- I need to generate a symbol, which rule?
- Top down parsing
- LL(1) parsing (parses input from left to right and constructs a leftmost parse)
- Table driven predictive parsing (no recursion) versus recursive descent parsing where each nonterminal is associated with a procedure call
- No backtracking
Two grammar problems – for top down parsing

1. *Eliminating left recursion* (without changing associativity)

   \[ E \rightarrow E + a \mid a \]

   I can't tell which rule to use \( E \rightarrow E + a \) or \( E \rightarrow a \) as both generate same first symbol

2. removing two rules with same prefix

   \[ E \rightarrow aB \mid aC \]

   I can't tell which rule to use as both generate same symbol

---

Removing Left Recursion

Before
- \( A \rightarrow A \times \)
- \( A \rightarrow y \)

After
- \( A \rightarrow yA' \)
- \( A' \rightarrow x A' \)
- \( A' \rightarrow \epsilon \)
Removing common prefix
(left factoring)

Stmt -> if Exp then Stmt else Stmt
   | if Expr then Stmt

Change so you don’t have to pick rule until later
Before:
   A -> \alpha_1 \beta_1 | \alpha_2 \beta_2

After:
   A -> \alpha A'
   A' -> \beta_1 | \beta_2

Exercises

Eliminate left recursion from the following grammars.

a) S -> (L) | a
   L -> L, S | S

b) E -> E or T | T #or is part of boolean expression
   T -> T and G | G
   G -> not G | (E) | true | false
The idea behind Parsing

• You start out with your grammar symbol and “someone smart” tells you which production you should use.

• You have a non-terminal E and you want to match a “e”. If only one rule from E produces something that starts with an “e”, you know which one to use.