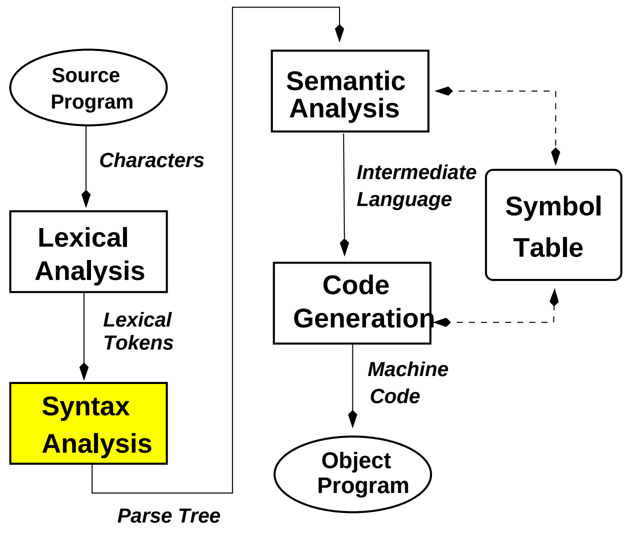
CSC 488S/CSC 2107S Lecture Notes



Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 4

Syntax Analysis

- The *syntax* of a language defines the sequences of language tokens that form legal sentences in the language.
- Examples:

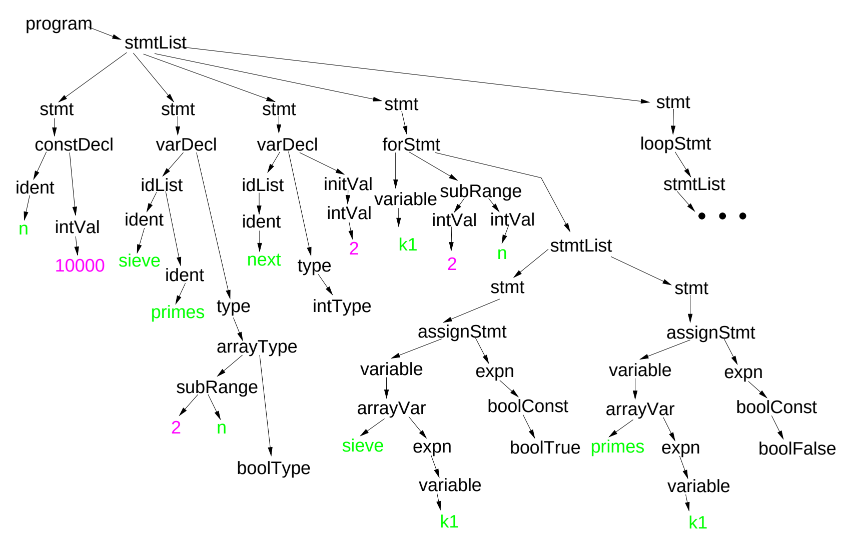
X := Y * Z + W / X	Legal
X =: * Y Z + / W X	Illegal
for(i = 0 ; i $<$ N ; i++)	Legal
for(i 0 ; i $=<$ N ; +i+)	Illegal

- Syntax analysis is the process of analyzing a sequence of lexical tokens to determine if they are a legal sentence in some given language.
- Syntax analysis is based on a precise formal definition of the source language.

Syntax Analysis Tasks

- Analyze a sequence of lexical tokens to determine its syntactic structure.
- Verify that this syntactic structure corresponds to a legal sentence in the language being processed.
- Transform the token stream into some intermediate form (a parse tree or an abstract syntax tree) that is suitable for subsequent processing.
- Handle syntax errors in the token stream.
- Maintain source program coordinates for use by subsequent passes.

Partial Parse Tree for Running Example (Slide 25)



Notations for Defining Languages

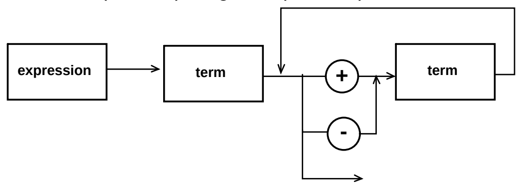
- Regular expressions

  expression = (identifier | integer | '(' expression ')')
  (($+$ | $-$ | $*$ | $\div$) (identifier | integer | '(' expression ')'))*

- Original Backus-Naur Form (BNF) (Algol 60)

< expression>	::=	< term>
	|	< expression> + < term>
	|	< expression> − < term>
< term>	::=	< factor>
	|	< term> ∗ < factor>
	|	< term> ÷ < factor>
< factor>	::=	< identifier>
	|	< integer>
	|	(< expression>)

- Transition (railroad) Diagrams (Pascal)



- Flex/JFlex Specification (Extended BNF)

expression	::=	term
	|	expression '+' term
	|	expression '−' term ;
term	::=	factor
	|	term '∗' factor
	|	term '÷' factor ;
factor	::=	identifier
	|	integer
	|	'(' expression ')' ;

How a Parser Operates

- Input: a stream of tokens from lexical analysis
  Output: A parse tree (or some equivalent form).
- Information available to the parser:
  1. The sequence of tokens that it has already received from lexical analysis.
  2. The sequence of steps that it has already taken in attempting to analyze the program.
  3. The next *k* lexical tokens in its input stream. (*k*-symbol lookahead).
- Most parsers use a stack to record the first two items.
- Unless the language being processed is particularly ugly, most parsers process the token stream left to right *without backtracking* and use $k = 1$.

Top Down vs. Bottom Up Parsing

- Top Down parsers start with the start symbol for the language (S) and attempt to find a sequence of production rules that transforms the start symbol into a given sequence of input tokens.
  Conceptually it builds the parse tree from the root to the leaves.
  A top down parse is called a **derivation**.
  Recursive Descent and LL(k) are top down parsing techniques.
- Bottom Up parsers start with a sequence of input tokens and attempt to find a sequence of production rules that will reduce the entire sequence of terminal symbols to the start symbol.
  Conceptually it builds the parse tree from the leaves to the root.
  A bottom up parse is called a **reduction** or a **reverse derivation**.
  LR(k), SLR(k) and LALR(k) are bottom up parsing techniques.
- For unambiguous, deterministic, context free languages, top down and bottom up parsers should produce the same parse tree for a given sequence of input tokens.

Basic Descriptive Terminology

Σ	A finite set of (terminal) symbols	$\{ a, b, c \}$
String	A finite sequence of symbols from Σ	$abbac$
λ	The empty string	
$\$$	End of file on input stream	
Concatenation	$X\ Y$	
	X^i means $XX\ldots X$ i times	
	$X^0 = \lambda$	
	X^* zero or more instances of X	
	X^+ one or more instances of X	
Language	Any set of strings formed from some vocabulary	

Notation	$A, B, S, \ldots$	Nonterminal symbols
	$a, b, c, \ldots$	Terminal symbols
	$\alpha, \beta, \gamma, \ldots$	Strings of terminal and nonterminal symbols.

Definitions for Parsing

- Let

N	Set of nonterminal symbols
Σ	Set of terminal symbols
S	The distinguished nonterminal start (goal) symbol
A	A *non-terminal* symbol in N
$\alpha, \beta, \gamma, \omega$	Strings in $(N \cup \Sigma)^*$

- The most general (*context sensitive*) form of a production rule is
  $$\gamma\ A\ \omega \ \rightarrow\ \gamma\ \alpha\ \omega$$
  In the left context γ and the right context ω , A derives α.
- For context free grammars, γ, ω are null and all rules are of the form:
  $$A \ \rightarrow\ \alpha$$

Programming Language Grammars

- A *context free grammar* G used to formally define programming languages is a 4-tuple $G = (N , \Sigma , S , P)$

  Σ A finite vocabulary of *terminal* symbols.
  This is the set of lexical tokens produced by lexical analysis.

  N A finite set of *nonterminal* symbols, the nonterminal vocabulary

  S The start symbol a distinguished nonterminal ($S \, \varepsilon \, N$)
  S is also called the *goal symbol*

  P A finite set of *productions* . The productions are also called *rewriting rules.*

  V The *vocabulary* of the grammar is $\Sigma \cup N$

- Each of the productions in P has the form:

  $$A \rightarrow X_1 \ldots X_m \qquad m \geq 0$$

  where

  $A \in N$

  $X_i \in V \quad , 1 \leq i \leq m , m \geq 0$

  Note $\quad A \rightarrow \quad$ is a valid production ($m = 0$)

Context Free Grammar Example

- For the language definition:

S	$\rightarrow$	A B
A	$\rightarrow$	a A
	|	a
B	$\rightarrow$	B b
	|	b

- Non terminals: $N = \{ \text{A , B , S} \}$
- Terminals: $\Sigma = \{ \text{a , b} \}$
- Goal symbol: S
- Productions:

  $$P = \{ \text{S} \rightarrow \text{A B} , \text{A} \rightarrow \text{a A} , \text{A} \rightarrow \text{a} , \text{B} \rightarrow \text{B b} , \text{B} \rightarrow \text{b} \}$$

Informal Definitions

- The productions in a grammar are a set of rules (in some notation) that define the sequences of terminal symbols that make up legal sentences in the language defined by the grammar.

- A grammar is unambiguous if there is exactly **one sequence of rules** for forming **each** legal sentence in the language. A **language** is ambiguous if **all** grammars for the language are ambiguous.
  Being unambiguous is good.

- A grammar is context free (CF) if every rule can be applied completely independent of the surrounding context.
  Being context free is good.

- A language is deterministic if it is always possible to determine which rule to apply next when parsing every sentence in the language.
  Being deterministic is good.

Formal Definitions for Parsing

- Define: *Sentential form:*

  The set of strings produced by the start symbol.

  $\{ w \mid S \Rightarrow^* w \}$

  w contains all strings that could occur during parse.

- Define: *Language:*

  The set of terminal strings produced by the start symbol

  $\{ w \mid S \Rightarrow^* w \} \cap \Sigma^*$

  This is all legal sentences (programs) in the language.

- Define: *Rewriting: The replacement of a nonterminal symbol A with the right side of one of the production rules for A e.g.* $A \rightarrow \alpha \; \beta \; \gamma$

- Define: *Derivation*

  Given $A \rightarrow \gamma$ *and sentential form* $\alpha A \beta$ *then*

  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ *is one derivation step*

  and

  $\alpha A \beta \Rightarrow^* \alpha \gamma \beta$ *derives in zero or more steps*

  $\alpha A \beta \Rightarrow^+ \alpha \gamma \beta$ *derives in one or more steps*

- A derivation can be represented as a *parse tree*.

  Applying a rule $A \rightarrow \alpha\ \beta\ \gamma$

  adds a node A with children α, β and γ to the parse tree

- If there is more than one nonterminal that can be rewritten in the sentential form then there are two conventions for choosing the nonterminal
  - *Leftmost derivation expand the nonterminals left to right*
  - *Rightmost derivation expand the nonterminals right to left*

  examples.

Nullability – Deriving the Empty String

- Define: *nullable*

  A string α *in* N^+ *is called nullable iff*

  $\alpha \Rightarrow^* \lambda$

- Nullability is a significant issue for the construction of parsers.

  When the string α actually produces λ then it provides **no** information that the parser can use to make a parsing decision.

- A grammar can be processed to determine which non terminal symbols can potentially produce λ, See *Fischer, Cytron, LeBlanc* Figure 4.7

- Example:

  A $\rightarrow$ B C D

  B $\rightarrow$ λ

  C $\rightarrow$ B | c

  D $\rightarrow$ B C | d

  A $\Rightarrow$ BCD $\Rightarrow$ CD $\Rightarrow$ BD $\Rightarrow$ D $\Rightarrow$ BC $\Rightarrow$ C $\Rightarrow$ B $\Rightarrow$ λ

Rewriting and Left/Right Recursion

- A production of the form $A \rightarrow \alpha\, \beta\, \gamma$ means

  $\alpha\, \beta\, \gamma$ can be derived from A in a top down parse

  $\alpha\, \beta\, \gamma$ can be reduced to A in a bottom up parse.

- Define: *recursive nonterminal*

  A nonterminal symbol A is recursive if $A \Rightarrow^* \alpha A \beta$

- Define: *left recursion*

  A nonterminal symbol A is left recursive if $A \Rightarrow^* A \beta$

- Define: *right recursive*

  A nonterminal symbol A is right recursive if $A \Rightarrow^* \alpha A$

- A grammar is ambiguous if any nonterminal is both left and right recursive.

  Example:

  $expression \Rightarrow expression + expression$

  In general the ambiguity of a context free grammar is not computable.

Grammar Design for Compiling

- A typical programming language will have two grammars.
  - A *reference grammar* that is made available to *users* of the language.
  - A *compiler grammar* that is used by the *implementation* of a language.
  - These two grammars **should** describe exactly the same language.
- The design of the compiler grammar
  - Affects the effort required to implement semantic analysis and code generation.
  - Determines the order in which constructs in the language will be processed (for a given parsing algorithm).
  - Determines the class of parsers that must be used to perform syntax analysis.
- Compiler implementors should use the degrees of freedom that they have in designing the compiler grammar wisely to make it easier to implement semantic analysis and code generation.

- For LALR(1) and SLR(1) parsers semantic and/or code generation actions can only be invoked when a reduction is applied.This means that actions can only occur at the right end of a rule.
- It is usually necessary to add extra productions to a grammar that provide a hook for attaching actions. YACC/Bison/Jcup can do this automatically.
- Example:

  Reference Grammar:

  ifStatement $\longrightarrow$ **if** expression **then** statement **else** statement **end**

  Compiler Grammar

  ifStatement $\longrightarrow$ ifHead statement elsePart **end** fixFwdBranch

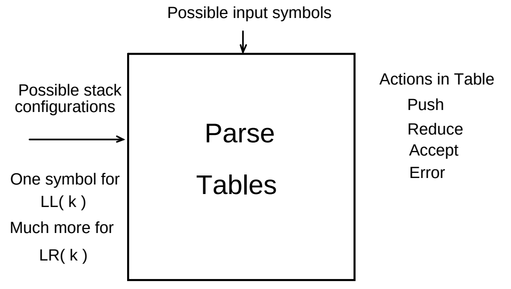
  ifHead $\longrightarrow$ **if** expression **then** checkBoolean emitBranchFalse

  elsePart $\longrightarrow$

  $\longrightarrow$ elseHead statement

  elseHead $\longrightarrow$ **else** emitFwdBranch fixFwdBranch

Table-Driven Parsing



- Conceptually a table for each parser state. Allow multiple actions per state so tables can be merged to one table for all parser states.
- Assume input stream ends with k end markers $\$^k$.
  Final state is accepting or rejecting.
- Top Down - stack represents what we expect to see.
  Bottom Up - stack represents what we've seen so far.

Practical Parsing Techniques

- **Recursive Descent** is a top down parsing technique with the potential for backtracking. Recursive descent parsers are usually written as a collection of interacting recursive functions. Recursive descent is useful for dealing with complicated or poorly designed languages (e.g. C (gcc), C++, Java, Fortran).
- **LL(1)** is a table–driven top down parsing technique.
  An LL(1) parser is a *deterministic push down automaton (DPDA)*
- **LR(1)** is a table–driven bottom up parsing technique.
  The parser uses a Shift/Reduce algorithm to record the state of its reduction.
  An LR(1) parser is a *deterministic push down automaton (DPDA)*
  SLR(1) and **LALR(1)** are practical versions of LR(1)

There are many other parsing techniques, but the ones listed here are used in most compilers.

First Sets

- First sets are a data structure used as a decision making tool in many parsers.
  For example if a top down parser is trying to find a *non-terminal* symbol B then it will look for *terminal symbols* in $first(B)$
- Define: $First(\,\alpha\,)$
  For a string α ($\alpha \in (\; N \;\cup\; \Sigma \;)^*$)
  $First(\,\alpha\,) = \{\, b \in \Sigma \;|\; \alpha \;\Rightarrow^*\; b\,\beta \,\}$
  The set $First(\,\alpha\,)$ is the set of **terminal symbols** that can occur at the beginning of strings derived from α.
- First Sets can also be defined recursively:

$First(\,c\,\beta\,)$	=	$\{c\}$	c is a terminal symbol
$First(\,\lambda\,)$	=	$\{\;\}$	
$First(\,B\,\gamma\,)$	=	$First(\,B\,)$	B is not nullable
	=	$First(\,B\,) \cup First(\,\gamma\,)$	B nullable

- For small grammars First sets can often by determined by inspection *as long as nullability is taken into account*. For larger grammars see the algorithm in *Fischer, Cytron, LeBlanc* Figure 4.8 .
- More generally $First_k(\,\alpha\,)$ uses k-symbol lookahead.

Follow Sets

- Follow sets are a data structure that are often used in making parsing decisions. For example if a nonterminal symbol A is *nullable* then the parser may look for *terminal symbols* in $Follow(A)$ to make a parsing decision.
- Define: $follow(\ A\ )$
  $Follow(\ A\ ) = \{\ b\ \in\ \Sigma\ \mid\ S\ \Rightarrow^{+}\ \alpha\, A\, b\, \beta\ \}$
  For a nonterminal symbol A, the set $follow(\ A\ )$ is the set of **terminal symbols** that can **immediately** follow A in **any** *sentential form* derived from the goal symbol S
- For *small* grammars it is possible to compute Follow sets by inspection, **but** this is much harder than computing First sets, since all possible instances of nullability must be accounted for.
  The calculation of Follow sets can be automated, See *Fischer, Cytron, LeBlanc* Figure 4.11
- More generally $Follow_k(\ \alpha\ )$ uses k-symbol lookahead.

Informal definition of Follow Sets

Follow Sets can also be defined intuitively:

If S is the goal symbol	Add $\{\ \$\ \}$ to $Follow(\ S\ )$	
If $S\ \Rightarrow^{+}\ \alpha\, A$	Add $\{\ \$\ \}$ to $Follow(A)$	
If $S\ \Rightarrow^{+}\ \alpha\, A\, c\, \beta$	Add $\{\ c\ \}$ to $Follow(A)$	
If $S\ \Rightarrow^{+}\ \alpha\, A\, B\, \beta$	Add $First(B)$ to $Follow(A)$	B not nullable
If $S\ \Rightarrow^{+}\ \alpha\, A\, B\, \beta$	Add $First(B) \cup First(\beta)$ to $Follow(A)$	B nullable
If $A\ \rightarrow\ \alpha\, B$	Add $Follow(A)$ to $Follow(B)$	

First and Follow Sets Example

Grammar:

A	$\rightarrow$	B C c
	$\rightarrow$	e D B
B	$\rightarrow$	λ
	$\rightarrow$	b C D E
C	$\rightarrow$	D a B
	$\rightarrow$	c a
D	$\rightarrow$	λ
	$\rightarrow$	d D
E	$\rightarrow$	e A f
	$\rightarrow$	c

B and D are *nullable*

First Sets (by inspection)

A $\rightarrow$ B C c			
A $\rightarrow$ e D B			
	$First(\ A\ )$	$= First(\ B\ ) \cup First(\ C\ c\ ) \cup \{\ e\ \}$	$= \{\ a, b, c, d, e\ \}$
B $\rightarrow$ λ			
B $\rightarrow$ b C D E			
	$First(\ B\ )$	$= First(\ \lambda\ ) \cup \{\ b\ \}$	$= \{\ b\ \}$
C $\rightarrow$ D a B			
C $\rightarrow$ c a			
	$First(\ C\ )$	$= First(\ D\ ) \cup First(\ a\ B\ ) \cup \{\ c\ \}$	$= \{\ a, c, d\}$
D $\rightarrow$ λ			
D $\rightarrow$ d D			
	$First(\ D\ )$	$= First(\ \lambda\ ) \cup \{\ d\ \}$	$= \{\ d\ \}$
E $\rightarrow$ e A f			
E $\rightarrow$ c			
	$First(\ E\ )$	$= \{\ e\ \} \cup \{\ c\ \}$	$= \{\ c, e\}$

Follow Sets (by inspection)

A → B C c

E → e A f

$Follow(A) = \{ \$ \} \cup \{ f \} \qquad = \{ f, \$ \}$

A → B C c

A → e D B

C → D a B

$Follow(B) = First(C) \cup Follow(A) \cup Follow(C) \qquad = \{ a, c, d, e, f, \$ \}$

A → B C c

B → b C D E

$Follow(C) = \{ c \} \cup First(D) \cup First(E) \qquad = \{ c, d, e \}$

A → e D B

B → b C D E

C → D a B

D → d D

$Follow(D) = First(B) \cup Follow(A) \cup First(E) \cup \{ a \} \qquad = \{ a, b, c, e, f, \$ \}$

B → b C D E

$Follow(E) = Follow(B) \qquad = \{ a, c, d, e, f, \$ \}$

Syntax Analysis – Good Language Design

- Use reserved words not keywords
- Design statements and declarations for ease of parsing
  e.g. statements start with distinct reserved words.
- Use sufficient bracketing so endings are clear.
  ambiguous **return** and **return** expression
  unambiguous **return** and **return** (expression)

  Semicolons as statement terminators are good.
- Unambiguous syntax
  ambiguous: **if** expression **then** statement **else** statement
  unambiguous: **if** expression **then** statement **else** statement **fi**
- Design for parsing with one token look ahead.

Syntacticallly Challenged Language - Python

The Python programming language uses *indentation* to mark the beginning and end of blocks. This includes delimiting the bodies of functions and the bodies of control statements.

```
Python                              Description
    def calc( x ) ;                 define function calc
        n = x * x + 7               assignment statement in calc
        return n * n + 5            return statement in calc
                                    end of calc
    def map ( n , m )               define function map
        if n < m :                  begin body of map
            i = n - m               body of if statement
            j = n + m               if statement continues
        if n > m :                  start new if statement
            i = n * m + 7           body of if statement
            j = i * 2 + 5           if statement continues
        return k - 17               end if statement
                                    end of map
    print map( 17, 23 )             start of main program
```

Since the scanner is the only part of the compiler that knows about indentation most of the heavy lifting should be done there. One possible solution.

- Define two lexical tokens `<INDENT>` and `<EXDENT>` that are emitted by the scanner whenever the level of indentation changes.
- Define the compilers parsing grammar in terms of these lexical tokens, e.g.
  body → `<INDENT>` statements `<EXDENT>`
  statement → `<INDENT>` statements `<EXDENT>`
- Suppress these invented tokens in any compiler error messages