Lecture 2:
A Simple One-Pass Compiler

COSC 4316

The Entire Compilation Process

- Grammars for Syntax Definition
- Syntax-Directed Translation
- Parsing - Top Down & Predictive
- Pulling Together the Pieces
- The Lexical Analysis Process
- Symbol Table Considerations
- A Brief Look at Code Generation
- Concluding Remarks/Looking Ahead

Overview

Programming Language can be defined by describing

1. The syntax of the language
    1. ***What its program looks like***
    2. ***We use CFG or BNF (Backus Naur Form)***
2. The semantics of the language
    1. ***What its program mean***
    2. ***Difficult to describe***
    3. ***Use informal descriptions and suggestive examples***

Grammars for Syntax Definition

- Context-free grammars are also useful to help guide the translation of programs using a technique known as syntax-directed translation.
- Informally, a context-free grammar is simply a set of rewriting rules, or productions.

  A production is of the form

  A --> B C D ....
- A is called the left-hand side (LHS) and B C D .... is the right-hand side. Every production in a CFG has exactly one symbol on its LHS.

Grammars for Syntax Definition

- A production represents the rule that any occurance of its LHS symbol can be replaced by the symbols on its RHS.
- Thus the production

  sentence --> *noun_phrase*   *verb_phrase*

  states that a *sentence* is required to be a noun phrase followed by a verb phrase. Other productions would be required to define what is mean by *noun_phrase* and *verb_phrase*

Grammars for Syntax Definition

- Two types of symbols in a CFG: **nonterminals** and **terminals**.
- Nonterminals

  -- often delimited by angle brackets (<...>).

  -- recognized by the fact that they appear on the LHS of productions.

  (i.e., they are placeholders)
- Terminals

  -- never changed or rewritten

  -- they represent the tokens of the language.

Grammars for Syntax Definition

- **Overall purpose of a CFG -- specify what sequences of terminals are legal**
- How does it do this?
  - select one of the nonterminals as the start symbol.
  - apply productions rewriting nonterminals until only terminals remain.

Grammars for Syntax Definition

- Def^{n}: A CFG is denoted by G=(T,N,P,S) where:
  1. T is a set of **tokens**, called **terminal** symbols
  2. N is a set of **nonterminals**
  3. P is a set of **productions** where each production consists of a nonterminal (LHS), an arrow, and a sequence of tokens and/or nonterminals (the RHS)
  4. S is a special nonterminal, called the **start symbol**. By convention, the LHS of the first production is the start symbol.

Grammars for Syntax Definition
Example CFG

$list \rightarrow list + digit$

$list \rightarrow list - digit$

$list \rightarrow digit$

$digit \rightarrow$ **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

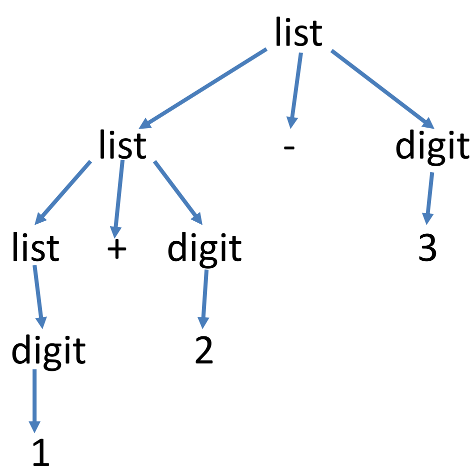
(the "|" means OR)

(So we could have written

$list \rightarrow list + digit \mid list - digit \mid digit$ **)**

N = { ***list, digit*** **}**
T = { +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
P – see box
S = ***list***

Grammars for Syntax Definition
Example CFG



Information

- ✓ A string of tokens is a sequence of zero or more tokens.
- ✓ The string containing with zero tokens, written as ε, is called ***empty*** string.
- ✓ A grammar derives ***strings*** by beginning with the start symbol and repeatedly replacing the non terminal by the right side of a production for that non terminal.
- ✓ The token strings that can be derived from the start symbol form the ***language*** defined by the grammar.

Grammars are Used to Derive Strings:

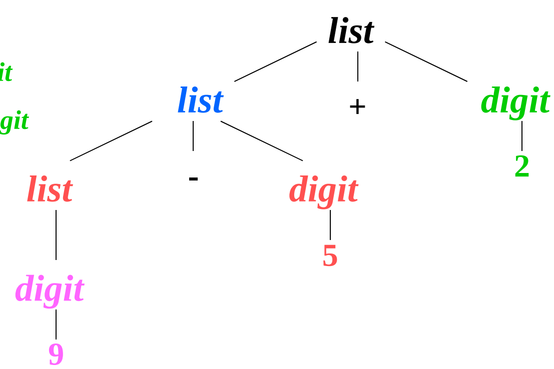
Using the CFG defined on the earlier slide, we can derive the string: 9 - 5 + 2 as follows:

list → *list* + *digit*	**P1 :** *list* → *list* + *digit*
→ *list* - *digit* + *digit*	**P2 :** *list* → *list* - *digit*
→ *digit* - *digit* + *digit*	**P3 :** *list* → *digit*
→ **9** - *digit* + *digit*	**P4 :** *digit* → **9**
→ **9** - **5** + *digit*	**P4 :** *digit* → **5**
→ **9** - **5** + **2**	**P4 :** *digit* → **2**

Grammars are Used to Derive Strings:

This derivation could also be represented via a Parse Tree
(parents on left, children on right)

list → *list* + *digit*
→ *list* - *digit* + *digit*
→ *digit* - *digit* + *digit*
→ 9 - *digit* + *digit*
→ 9 - 5 + *digit*
→ 9 - 5 + 2



A More Complex Grammar

block → **begin** *opt_stmts* **end**

opt_stmts → *stmt_list* | ε
stmt_list → *stmt_list* ; *stmt* | *stmt*

What is this grammar for ?
What does "ε" represent ?
What kind of production rule is this ?

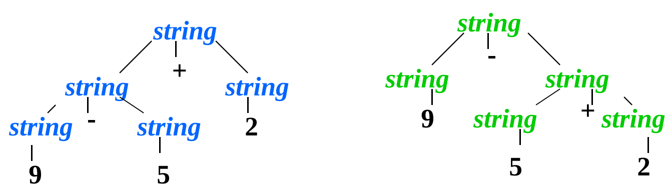
Defining a Parse Tree

- A parse tree pictorially shows **how** the start symbol of a grammar derives a string in the language.
- More Formally, a Parse Tree for a CFG Has the Following Properties:
  - Root Is Labeled With the Start Symbol
  - Leaf Node Is a Token or ε
  - Interior Node Is a Non-Terminal
  - If $A \rightarrow x_1 x_2 \ldots x_n$, Then A Is an Interior; $x_1 x_2 \ldots x_n$ are *children* of A and may be Non-Terminals or Tokens

Other Important Concepts

Ambiguity

Two derivations (Parse Trees) for the same token string.



Grammar:

string → *string* + *string* | *string* – *string* | **0** | **1** | **…**| **9**

Why is this a Problem ?

Other Important Concepts
Associativity of Operators
Left vs. Right

a+b+c is (a+b)+c = a+(b+c)
a-b-c is (a-b)-c =/ a-(b-c)

+, -, *, / are evaluated left to right (left associative)
exponentiation is right associative.
2**3**2 = 2**(3**2) = 2**9 = 512
≠ (2**3)**2 = 8**2 = 64

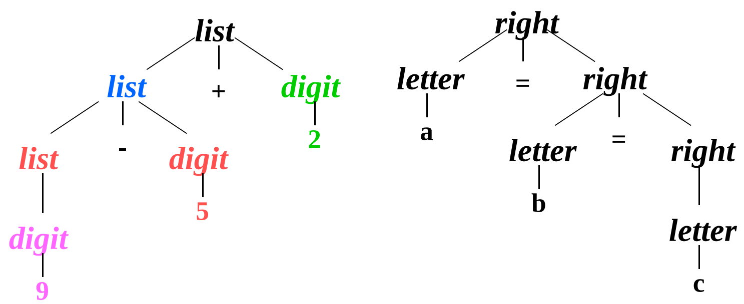
Other Important Concepts
Associativity of Operators
Left vs. Right

- The C assignment operator is also right associative
  a=b=c means a = (b=c)
- with left associative operators –
  the parse tree grows down to the left
  productions are left recursive
- With right associative operators –
  the parse tree grows down to the right
  productions are right recursive

Other Important Concepts

Associativity of Operators

Left vs. Right



list → *list* + *digit* |

| *list* - *digit* | *digit*

digit → **0 | 1 | 2 | …| 9**

right → *letter* = *right* | *letter*

letter → **a | b | c | …| z**

Embedding Associativity

- The language of arithmetic expressions with + -
  - (ambiguous) grammar that does not enforce associativity

    ***string* → *string + string* | *string – string* | 0 | 1 | …| 9**

  - non-ambiguous grammar enforcing left associativity (parse tree will grow to the left)

    string* → *string + digit* | *string - digit* | *digit

    ***digit* → 0 | 1 | 2 | …| 9**

  - non-ambiguous grammar enforcing right associativity (parse tree will grow to the right)

    string* → *digit + string* | *digit - string* | *digit

    ***digit* → 0 | 1 | 2 | …| 9**

Other Important Concepts
Operator Precedence

What does
9 + 5 * 2
mean?

Typically $\left\{\begin{array}{l} (\ ) \\ *\quad / \\ +\quad - \end{array}\right.$ is precedence order

This can be incorporated into a grammar via rules:

$$
\begin{aligned}
expr &\rightarrow expr + term \mid expr - term \mid term \\
term &\rightarrow term * factor \mid term \,/\, factor \mid factor \\
factor &\rightarrow digit \mid (\ expr\ ) \\
digit &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\end{aligned}
$$

Precedence Achieved by:
expr & *term* for each precedence level

Rules for each are left recursive **or** associate to the left

Other Important Concepts
Operator Precedence

What if we want to add exponentiation?
We redefine *factor* and add a new production for *primary*

$$
\begin{aligned}
expr &\rightarrow expr + term \mid expr - term \mid term \\
term &\rightarrow term * factor \mid term \,/\, factor \mid factor \\
factor &\rightarrow primary \wedge factor \mid primary \\
primary &\rightarrow digit \mid (\ expr\ ) \\
digit &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\end{aligned}
$$

Syntax for Statements

stmt → id := *expr*

| if *expr* then *stmt*

| if *expr* then *stmt* else *stmt*

| while *expr* do stmt

| begin *opt_stmts* end

Ambiguous Grammar?

Syntax-Directed Translation

- Associate Attributes With Grammar Rules and Translate as Parsing occurs

- The translation will follow the parse tree structure (and as a result the structure and form of the parse tree will affect the translation).

- First example: Inductive Translation.
- Infix to Postfix Notation Translation for Expressions
  - Translation defined inductively as: Postfix(E) where E is an Expression.

Rules

1. ***If*** **E** ***is a*** **variable** ***or*** **constant** ***then*** **Postfix(E) = E**
2. ***If*** **E** ***is*** **E1 op E2** ***then*** **Postfix(E)**

   = Postfix(E1 op E2) = Postfix(E1) Postfix(E2) op
3. ***If*** **E** ***is*** **(E1)** ***then*** **Postfix(E) = Postfix(E1)**

Examples

Postfix((9 – 5) + 2)
= Postfix((9 – 5)) Postfix(2) +
= Postfix(9 – 5) Postfix(2) +
= Postfix(9) Postfix(5) - Postfix(2) +
= 9 5 – 2 +

Postfix(9 – (5 + 2))
= Postfix(9) Postfix((5 + 2)) -
= Postfix(9) Postfix(5 + 2) –
= Postfix(9) Postfix(5) Postfix(2) + –
= 9 5 2 + –

Syntax-Directed Definitions

- syntax-directed definition
  - uses a CFG to specify the syntactic structure of the input.
  - with each grammar symbol, it associates a set of attributes (properties)
  - with each production, it associates a set of semantic rules (actions) for computing the values of the attributes associated with the symbols

Syntax-Directed Definition

- Each Production Has a Set of Semantic Rules
- Each Grammar Symbol Has a Set of Attributes
- For the Following Example, String Attribute "*t*" is Associated With Each Grammar Symbol

$$expr \rightarrow expr - term \mid expr + term \mid term$$
$$term \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \ldots \mid \mathbf{9}$$

- recall: What is a Derivation for 9 + 5 - 2?

$$list \rightarrow list - digit \rightarrow list + digit - digit \rightarrow digit + digit - digit$$
$$\rightarrow 9 + digit - digit \rightarrow 9 + 5 - digit \rightarrow 9 + 5 - 2$$

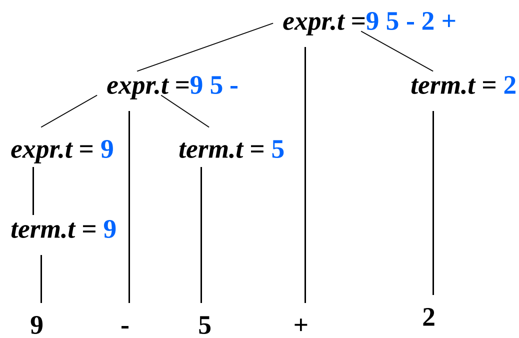
Syntax-Directed Definition (2)

- Each Production Rule of the CFG Has a Semantic Rule

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\, term.t \,\|$ '+'
$expr \rightarrow expr - term$	$expr.t := expr.t \,\|\, term.t \,\|$ '-'
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t :=$ '0'
$term \rightarrow 1$	$term.t :=$ '1'
….	….
$term \rightarrow 9$	$term.t :=$ '9'

- Note: Semantic Rules for *expr* define *t* as a "synthesized attribute" i.e., the various copies of *t* obtain their values from "children *t*'s"

Semantic Rules are Embedded in Parse Tree



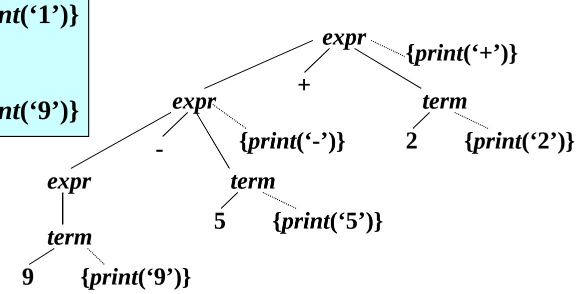
- It starts at the root and recursively visits the children of each node in left-to-right order
- The semantic rules at a given node are evaluated once all descendants of that node have been visited.
- A parse tree showing all the attribute values at each node is called annotated parse tree.

Translation Schemes

Embedded Semantic Actions into the right sides of the productions.

A translation scheme is like a syntax-directed definition except the order of evaluation of the semantic rules is explicitly shown.

expr → *expr* + *term*		{*print*('+')}
	→ *expr* - *term*	{*print*('-')}
	→ *term*	{*no action*}
term → *0*		{*print*('0')}
term → 1		{*print*('1')}
…		
term → 9		{*print*('9')}



Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar.

Parser must be capable of constructing the tree.

Two types of parser

`Top-down:`	*efficient parsers can be easily constructed*
• **`starts at root`**	*by hand using this method*
• **`proceeds towards leaves`**	
`Bottom-up:`	*handles a larger class of grammars*
• **`starts at leaves`**	*software tools generally use this method*
• **`proceeds towards root`**	

Parsing

- Most parsing methods process input in a "greedy" fashion -- construct as much of the parse tree as possible before proceeding to the next character.
- If the actions in the syntax-directed scheme proceed from left to right (most do) we can execute the actions while parsing and avoid explicitly building the parse tree.

Parsing – Top-Down & Predictive

- Top-Down Parsing ⇒ Parse tree / derivation of a token string occurs in a top down fashion.
- For Example, Consider the following simplified grammar for the types of Pascal:

Start symbol

type → *simple*
  | ↑ **id**
  | **array** [*simple*] **of** *type*
simple → **integer**
  | **char**
  | **num dotdot num**

Suppose input is :

array [**num dotdot num**] **of integer**

Parsing would begin with

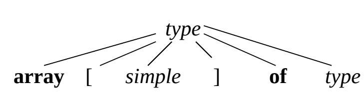
type → **???**

Top-Down Parse (type = start symbol)

Lookahead symbol

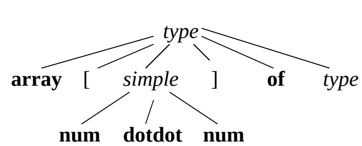
Input : **array** [**num dotdot num**] **of integer**

type
?



Lookahead symbol

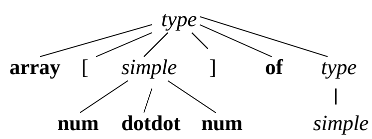
Input : **array** [**num dotdot num**] **of integer**



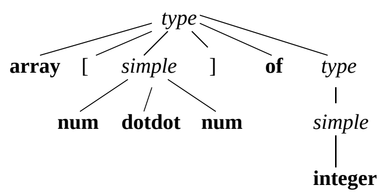
Top-Down Parse (type = start symbol)

Lookahead symbol

Input : **array** [**num dotdot num**] **of** **integer**



The selection of production for non terminal may involve trail and error



Top-Down Process
Recursive Descent or Predictive Parsing

- Parser Operates by Attempting to Match Tokens in the Input Stream
- The lookahead symbol unambiguously determines the next production to apply.
- The parse tree is not explicitly built
- Each invocation of a production calls a parsing procedure which can recognize any sequence of tokens generated by that nonterminal.
- To match a nonterminal **A**, we call the parsing procedure corresponding to **A**
- To match a terminal symbol, **t**, we call a procedure **match(t)**.
  - **match** compares t to the lookahead token
  - if the lookahead token is t, everything is correct and the scanner is called to get the next lookahead token.
  - Otherwise an error has occurred.

Top-Down Process

Recursive Descent or Predictive Parsing

- Parser Operates by Attempting to Match Tokens in the Input Stream
- Utilize both Grammar and Input Below to Motivate Code for Algorithm

array [**num dotdot num**] **of** **integer**

type → *simple*
  | ↑ **id**
  | **array** [*simple*] **of** *type*
simple → **integer**
  | **char**
  | **num dotdot num**

procedure *match* (*t* : token)
begin
  if *lookahead* = *t* **then**
    lookahead := *input.nextToken*()
  else
    error(*t* + "expected")
  endif
end ;

Top-Down Algorithm (Continued)

procedure *type*()
begin
  case *lookahead* **of**
    '↑': *match* ('↑') ;
      match(**id**);
    array: *match*(**array**);
      match('[');
      simple();
      match(']');
      match(**of**);
      type();
    otherwise:
      simple();
  endcase
end ;

Top-Down Algorithm (Continued)

```
procedure simple()
begin
      if lookahead = integer then
        match ( integer );
      elseif lookahead = char then
        match ( char );
      elseif lookahead = num then
        match (num);
        match (dotdot);
        match (num);
      else
        error("invalid type");
      endif
end ;
```

Tracing

Input: **array** [**num dotdot num**] **of integer**

To initialize the parser:

set global variable : **lookahead = array**

call procedure: **type**

Procedure call to **type** *with* **lookahead = array** *results in the actions:*
match(**array**); *match*('['); simple; *match*(']'); *match*(**of**); *type*

Procedure call to **simple** *with* **lookahead = num** *results in the actions:*
match (**num**); *match* (**dotdot**); *match* (**num**)

Procedure call to **type** *with* **lookahead = integer** *results in the actions:*
simple

Procedure call to **simple** *with* **lookahead = integer** *results in the actions:*
match (**integer**)

Limitations

- Can we apply the previous technique to every grammar?
- NO:

type $\rightarrow$ *simple*
| **array** [*simple*] **of** *type*
simple $\rightarrow$ **integer**
| **array** *digit*
digit $\rightarrow$ **0|1|2|3|4|5|6|7|8|9**

consider the string "**array** 6"
the predictive parser starts with *type* and lookahead= **array**
apply production *type* $\rightarrow$ *simple* OR *type* $\rightarrow$ **array** *digit* ??

Enhanced Grammar for Types

- consider the following grammar for the types of Pascal:

type --> *simple* | RECORD *fields* END
| ARRAY [*simple*] OF *type* | (*idlist*)
simple --> **integer** | **char** | **num dotdot num**
fields --> *field* ; *fields* | *field*
field --> *idlist* : *type*
idlist --> **identifier** , idlist | **identifier**

FIRST() sets

- Predictive parsing relies on information about what first symbols can be generated by the right side of a production.
- FIRST(s) -- set of tokens that appear as the first symbols of one or more strings generated from s.
  - e.g., FIRST(*simple*) = { intsym, num, charsym }
  - FIRST(*type*) = { arraysym, recsym, lparen, intsym, num, charsym }

When to Use ε-Productions

The recursive descent parser will use ***ε-productions*** as a default when no other production can be used.

stmt → **begin** ***opt_stmts*** **end**

opt_stmts → ***stmt_list*** | ε

While parsing ***opt_stmts*,** if the lookahead symbol is not in FIRST(***stmts_list*)**, then the ***ε-productions*** is used.

Designing a Predictive Parser

- Consider $A \rightarrow \alpha$
  - FIRST(α)=set of leftmost tokens that appear in α or in strings generated by α.
  - E.g. FIRST(*type*)={↑,**array,integer,char,num**}

- Consider productions of the form $A \rightarrow \alpha$, $A \rightarrow \beta$ the sets FIRST(α) and FIRST(β) should be disjoint; Otherwise, backtracking will be required

- Then we can implement predictive parsing
  - Starting with $A \rightarrow ?$ we find into which FIRST() set the *lookahead* symbol belongs to and we use this production.
  - Any non-terminal results in the corresponding procedure call
  - Terminals are matched.

Problems with Top Down Parsing

- Left Recursion in CFG May Cause Parser to Loop Forever.
- Indeed:
  - In the production $A \rightarrow A\ \alpha$ we write the program

```
procedure A
{
 if lookahead belongs to First(A α) then
                    call the procedure A
}
```

- Left recursion will result in an infinite recursive loop
- Solution: Remove Left Recursion...
  - without changing the Language defined by the Grammar.

Dealing with Left recursion

- Solution: Algorithm to Remove Left Recursion:

BASIC IDEA

$A \rightarrow A\ \alpha \mid \beta$ becomes

$A \rightarrow \beta\ R$

$R \rightarrow \alpha\ R \mid \varepsilon$

$$\begin{aligned} expr &\rightarrow expr + term \mid expr - term \mid term \\ term &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

$$\begin{aligned} expr &\rightarrow term\ \ rest \\ rest &\rightarrow +\ term\ \ rest \mid -\ term\ \ rest \mid \varepsilon \\ term &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

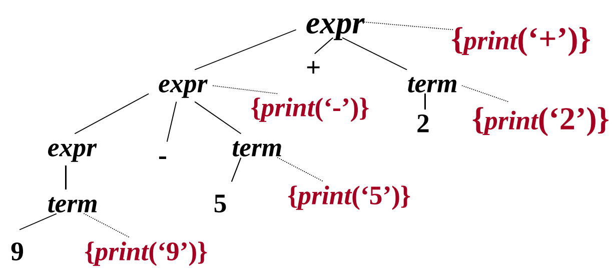
What happens to semantic actions?

expr → *expr* + *term*	{*print*('+')}
→ *expr* - *term*	{*print*('-')}
→ *term*	
term → *0*	{*print*('0')}
term → 1	{*print*('1')}
…	
term → 9	{*print*('9')}

expr → *term rest*	
rest → + *term* {*print*('+')} *rest*	
→ - *term* {*print*('-')} *rest*	
→ *ε*	
term → *0*	{*print*('0')}
term → 1	{*print*('1')}
…	
term → 9	{*print*('9')}

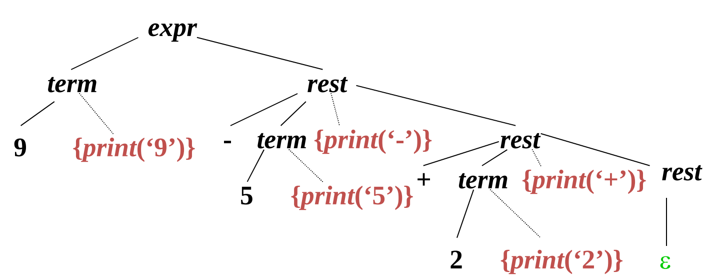
Comparing Grammars with Left Recursion

- Notice Location of Semantic Actions in Tree
- What is Order of Processing?



Comparing Grammars without Left Recursion

- Now, Notice Location of Semantic Actions in Tree for Revised Grammar
- What is Order of Processing in this Case?



Procedure for the Non terminals *expr*, *term*, and *rest*

```
expr()
{
        term(), rest();
}
rest()
{
        if ( lookahead == '+') {
                match('+'); term(); putchar('+'); rest();
        }
        else if ( lookahead = '-') {
                match('-'); term(); putchar('-'); rest();
        }
        else ;
}
```

Procedure for the Non terminals *expr*, *term*, and *rest (2)*

```
term()
{
        if (isdigit(lookahead)){
                putchar(lookahead); match();
        }
        else error();
}
```

Optimizing the translator

Tail recursion

When the last statement executed in a procedure body is a recursive call of the same procedure, the call is said to be *tail recursion*.

```
rest()
{
    done = false;
    repeat
        if ( lookahead == '+') {
            match('+'); term(); putchar('+');
        }
        else if ( lookahead = '-') {
            match('-'); term(); putchar('-');
        }
        else  done = true;
    until(done);
}
```

Optimizing the translator

- Now, because the only call to *rest* comes in *expr*, we can put the above code directly in *expr* and eliminate the procedure.
- This corresponds to the extended BNF
  `expr → term { (+|-) term }`
  ({ } means 0 or more times)

```
expr()
{
        term();
        while(1) {
            if ( lookahead == '+') {
                match('+'); term(); putchar('+');
            }
            else if ( lookahead = '-') {
                match('-'); term(); putchar('-');
            }
            else  break;
        }
}
```

Lexical Analysis

A lexical analyzer reads and converts the input into a stream of tokens to be analyzed by the parser.

A sequence of input characters that comprises a single token is called a **lexeme.**

Functional Responsibilities

1. White Space and Comments Are Filtered Out

- blanks, new lines, tabs are removed
- modifying the grammar to incorporate white space into the syntax is difficult to implement
- If the lexical analyzer removes comments and white space the parser will never need to consider it

Functional Responsibilities (2)

2. Recognition of Constants

- The job of collecting digits into integers is generally given to a lexical analyzer because numbers can be treated as single units during translation.
- If **num** is the token representing an integer, the lexical analyzer passes both the token and the attribute (value) to the parser
- **Example:**

`31 + 28 - 59` gives the sequence of tuples

<**num**, 31> <**plussym**, _ > <**num**, 28> < **minussym**, _ > <**num**, 59>

Note (important): 2nd Component of the tuples, the attributes, play no role during parsing, but needed during translation

Functional Responsibilities (3)

3. Recognizing Identifiers and Keywords

Compilers use identifiers as names of

- *Variables*
- *Arrays*
- *Functions*

A grammar for a language treats an **identifier** as **token**

Example:

alpha = alpha + beta;

Lexical analyzer would convert it like

id = id + id ;

and send the parser the sequence of tuples
<**id**, ptr_1> <**asgnsym**, _ > <**id**, ptr_1> <**plussym**, _ > <id, ptr_2> <**semi**, _ >

ptr is pointer to the appropriate symbol table entry.

Functional Responsibilities (3)

3. Recognizing Identifiers and Keywords (cont'd)

Languages use fixed character strings (**if**, **while**, **extern**) to identify certain constructs. We call them ***keywords.***

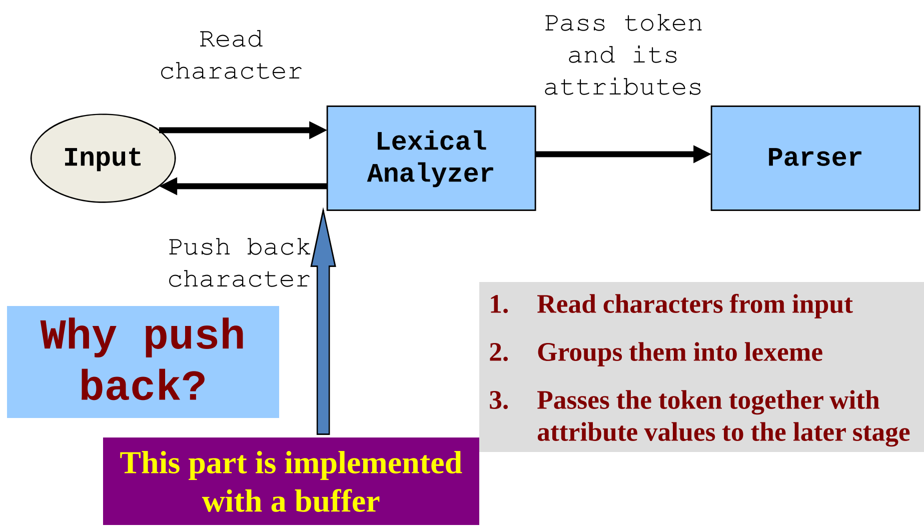
A mechanism is needed for deciding when a lexeme forms a keyword and when it forms an identifier.

Solution

1. **Keywords are reserved.**
   (Avoid problems like: `if if then then else else`)
2. **The character string forms an identifier only if it is not a keyword.**

Other issues: differentiate between <, <> <=, etc.

Interface to the Lexical Analyzer



Why Push Back?

- When do we push a character back?
  - e.g., when the lexical analyzer sees the char '<' it doesn't know if that is the entire token or not.
  - It could be '<' or '<>‘. The next character must be read.
  - If it is not '>', then the character must be pushed back into the input stream and the tuple passed to the parser.
  - Otherwise is passed to the parser.

The Lexical Analysis Process
A Graphical Depiction

uses getchar ( ) to read character

pushes back c using ungetc (c , stdin)

lexan ( ) lexical analyzer

returns token to caller

tokenval

Sets global variable to attribute value

How do we push back a char?
In C, use ungetc();
In Java, not really so difficult -- read a line as a string and use the string as a buffer.

Example of a Lexical Analyzer

function *lexan*: **integer** ; **[Returns an integer encoding of token]**
var *lexbuf* : **array**[0 .. 100] **of char** ;
c : **char** ;
begin
while(true) do
read a character into *c* ;
if ***c*** is a blank or a tab **then**
do nothing
elseif *c* is a newline **then**
lineno : = *lineno* + 1
elseif *c* is a digit **then**
set *tokenval* to the value of this and following digits ;
return NUM

Algorithm for Lexical Analyzer

```
elseif c is a letter then
        place c and successive letters and digits into lexbuf ;
        p := lookup ( lexbuf ) ;
        if p = 0 then
            p := insert ( lexbf, ID) ;
        tokenval := p
        return the token field of table entry p
else
        set tokenval to NONE ; /* there is no attribute */
        return integer encoding of character c
endif
    endwhile
end lexan
```

Note: Insert / Lookup operations occur against the Symbol Table !

Incorporating a Symbol Table

- The symbol table is:
  - used to store information about source language constructs
  - information gathered during the analysis phases
    - lexical analyzer
      creates a symbol table entry for an identifier associated with a given lexeme
    - syntax analyzer type of the identifier
  - information used during the synthesis phases
    - code generation phase
      uses the symbol table information to generate the proper code to access and store the variable.

Incorporating a Symbol Table

- The symbol table is a ADT data structure; therefore we need accessing functions:
  - procedure insert(s : string; t : token);
    - inserts lexeme in s along with token t in symbol table
  - and function lookup(s : string) : symbol_table_ptr;
    - returns pointer to the entry for s or ***null*** if s is not found

Handling Reserved Words

- Reserved words may be handled by initializing them in the symbol table; e.g.,
  insert("div",**divsy**)
  insert("mod",**modsy**)
  lookup("div") returns (a pointer to) the token **divsy**,
  so DIV cannot be used as an identifier.
- Alternatively, the reserved words may be placed in a separate table which is always scanned (using, say, binary search) whenever an identifier is recognized.

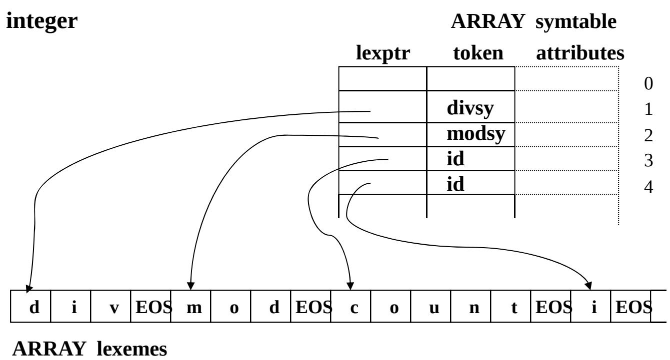
Symbol Table Considerations

OPERATIONS: Insert (string, token_ID)
Lookup (string)

NOTICE: Reserved words are placed into symbol table for easy lookup

Attributes may be associated with each entry, i.e.,
Semantic Actions
Typing Info: id → integer
etc.

Variations:
Store lexemes directly in table as a fixed-length char array



Abstract Stack Machines

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.

One popular form of intermediate representation is code for an ***abstract stack machine***.

I will show you how code will be generated for it.

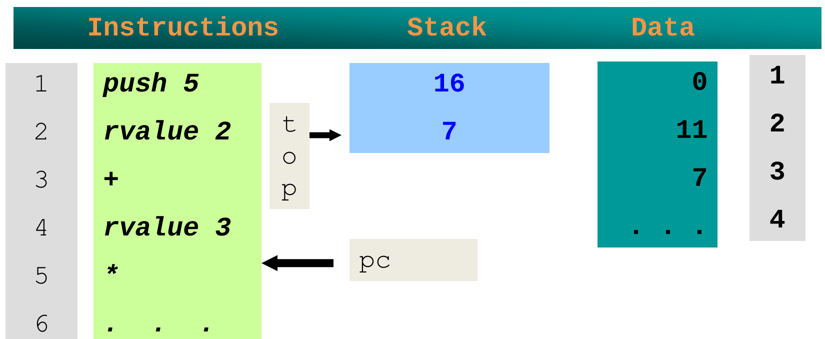
The properties of the machine

1. **Instruction memory**
2. **Data memory**
3. **All arithmetic operations are performed on values on a stack**

Instructions

Instructions fall into three classes.

1. Integer arithmetic
2. Stack manipulation
3. Control flow



L-value and R-value

What is the difference between left and right side identifier?

L-value Vs. R-value of an identifier

I : = 5 ; L - Location

I : = I + 1 ; R – Contents

The right side specifies an integer value, while left side specifies where the value is to be stored.

Usually,

r-values are what we think as values

l-values are locations.

Stack manipulation

push *v*	push v onto the stack
rvalue *l*	push contents on data location *l*
lvalue *l*	push address of data location *l*
pop	throw away value on top of the stack
:=	the r-value on top is placed in the l-value below it and both are popped
copy	push a copy of the top on the stack

Translation of Expressions

```
Day = (1461*y) mod 4 + (153*m +2 ) mod 5 + d
```

```
lvalue day
push 1461
rvalue y
*
push 4
mod
push 153
rvalue m
*
```

```
push  2
+
push 5
mod
+
rvalue d
+
:=
```

0	1
	2 day
1	3 y
2	4 m
-3	5 d
. . .	

Translation of Expressions (2)

```
lvalue day
push 1461
rvalue y
*
push 4
mod
push 153
rvalue m
*
 push  2
 +
 push 5
 mod
 +
 rvalue d
 +
 :=
```

2	2	2	2	2	2	2
	1461	1461	1461	1461	1	1
		1		4		153

0	1
	2 day
1	3 y
2	4 m
-3	5 d
. . .	

`Day = (1461*y) mod 4 + (153*m +2 ) mod 5 + d`

Translation of Expressions (3)

```
lvalue day
push 1461
rvalue y
*
push 4
mod
push 153
rvalue m
*
 push  2
 +
 push 5
 mod
 +
 rvalue d
 +
 :=
```

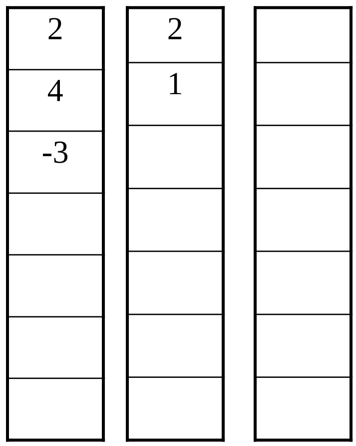
2	2	2	2	2	2	2
1	1	1	1	1	1	4
153	306	306	308	308	3	
2		2		5		

0	1
	2 day
1	3 y
2	4 m
-3	5 d
. . .	

`Day = (1461*y) mod 4 + (153*m +2 ) mod 5 + d`

```
lvalue day
push 1461
rvalue y
*
push 4
mod
push 153
rvalue m
*
 push  2
 +
 push 5
 mod
 +
 rvalue d
 +
 :=
```

Translation of Expressions (4)



0	1
1	2 day
2	3 y
-3	4 m
. . .	5 d

Control Flow

The control flow instructions for the stack machine are

label *l*	**target of jumps to *l*; has no other effect**
goto *l*	**next instruction is taken from statement with label *l***
gofalse *l*	**pop the top value; jump if it is zero**
gotrue *l*	**pop the top value; jump if it is nonzero**
halt	**stop execution**

Translation of statements

if
code for ***expr***
gofalse out
code for ***stmt1***
label out

```
stmt → if expr then stmt1
```

```
{ out := newlabel;
  stmt.t := expr.t ||
            'gofalse' out ||
            stmt1.t ||
            'label' out }
```

Translation of statements (2)

label test
code for expr
gofalse out
code for stmt1
goto test
label out

while

```
stmt → while expr do stmt1
```

```
{ test := newlabel;
  out  := newlabel;
  stmt.t := 'label' test ||
            expr.t ||
            'gofalse' out ||
            stmt1.t ||
            'goto' test ||
            'label' out }
```

Example 1

- IF a OR b THEN c := d + 1 ENDIF translates to:

```
rvalue  a
rvalue  b
or
gofalse labelnnn
lvalue  c
rvalue  d
push 1
add
:=
label   labelnnn
```

Emitting a Translation

- This code, for instance might emit the stack code for an if statement

```
procedure ifstmt()
begin
     match(ifsym)
     expr()
     match(thensym)
     target=newLabel()
     emit("gofalse\t" + target)
     stmt()
     emit("label\t" + target)
end ifstmt
```

Concluding Remarks / Looking Ahead

- We've Reviewed / Highlighted Entire Compilation Process
- Introduced Context-free Grammars (CFG) and Indicated /Illustrated Relationship to Compiler Theory
- Reviewed Many Different Versions of Parse Trees That Assist in Both Recognition and Translation
- Thus, we have ended our "View from 30,000 feet"
- We'll Return to Beginning - Lexical Analysis
- We'll Explore Close Relationship of Lexical Analysis to Regular Expressions, Grammars, and Finite Automatons