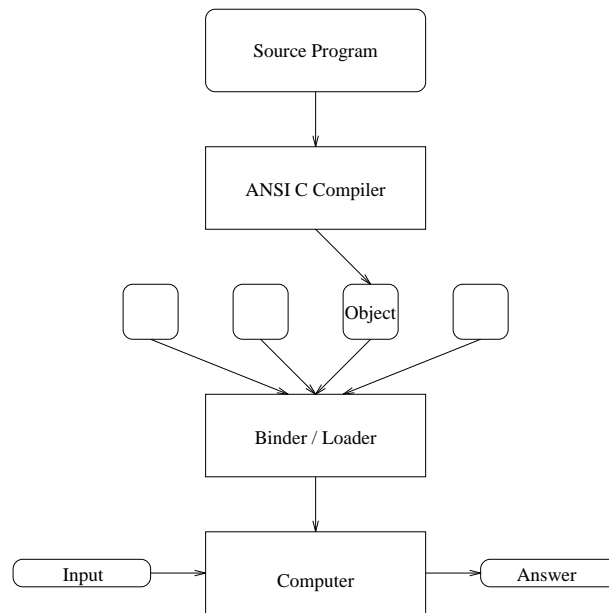


Tutorial outline

1. Introduction
2. Scanning and tokenizing
3. Grammars and ambiguity
4. Recursive descent parsing
5. Error repair
6. Table-driven LL parsing
7. Bottom-up table-driven parsing
8. Symbol tables
9. Semantic analysis via attributes
10. Abstract syntax trees
11. Type checking
12. Runtime storage management
13. Code generation
14. Optimization
15. Conclusion

What is a language translator?

You type: `cc foo.c . . .` What happens?



Language: Vehicle (architecture) for transmitting information between components of a system. For our purposes, a language is a *formal interface*. The goal of every compiler is correct and efficient language translation.

The process of language translation

1. A person has an idea of how to compute something:

$$fact(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ n \times fact(n-1) & \text{otherwise} \end{cases}$$

2. An algorithm captures the essence of the computation:

$$fact(n) = \text{if } n \leq 0 \text{ then } 1 \text{ else } n \times fact(n-1)$$

Typically, a *pseudocode* language is used, such as “pidgin ALGOL”.

3. The algorithm is expressed in some programming language:

```
int fact(int n) {
    if (n <= 0) return(1);
    else return(n*fact(n-1));
}
```

We would be done if we had a computer that “understood” the language directly.
So why don’t we build more C machines?

- | | |
|--|--|
| a) How does the machine know it’s seen a C program and not a Shakespeare sonnet? | c) It’s hard to build such machines. What happens when language extensions are introduced (C++)? |
| b) How does the machine know what is “meant” by the C program? | d) RISC philosophy says simple machines are better. |

Finally...

A compiler translates programs written in a *source* language into a *target* language. For our purposes, the source language is typically a *programming language*—convenient for humans to use and understand—while the target language is typically the (relatively low-level) instruction set of a computer.

Source Program

```
main() {
    int a;

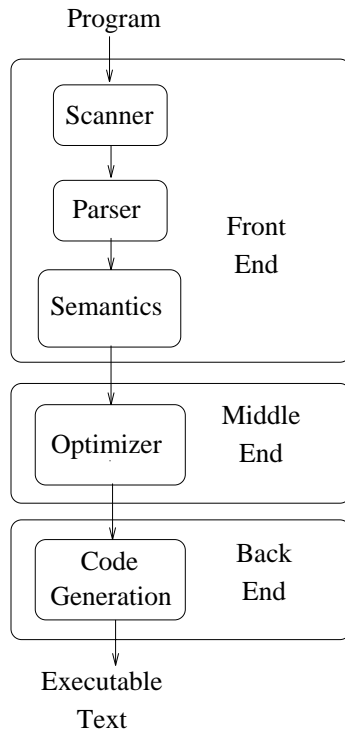
    a += 5.0;
}
```

Target Program (Assembly)

```
_main:
    !#PROLOGUE# 0
    sethi    %hi(LF12),%g1
    add     %g1,%lo(LF12),%g1
    save    %sp,%g1,%sp
    !#PROLOGUE# 1
    sethi    %hi(L2000000),%o0
    ldd     [%o0+%lo(L2000000)],%f0
    ld      [%fp+-0x4],%f2
    fitod   %f2,%f4
    faddd   %f4,%f0,%f6
    fdtoi   %f6,%f7
    st      %f7,[%fp+-0x4]
```

Running the Sun `cc` compiler on the above source program of 32 characters produces the assembly program shown to the right. The bound binary executable occupied in excess of 24 thousand bytes.

Structure of a compiler

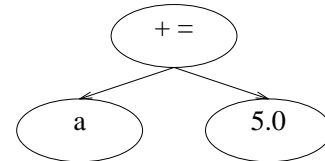


Front End

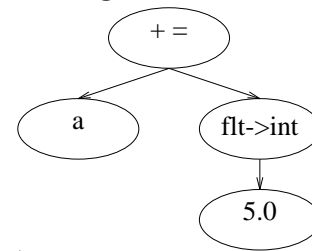
Scanner: decomposes the input stream into *tokens*. So the string "a += 5.0;" becomes

`a` `+` `=` `5.0` `;`

Parser: analyzes the tokens for correctness and structure:

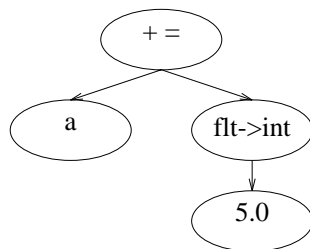


Semantic analysis: more analysis and type checking:

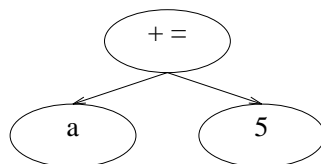


Structure of a compiler

Middle End



The middle end might eliminate the conversion, substituting the integer "5" for the float "5.0".



Code Generation

The code generator can significantly affect performance. There are many ways to compute "a+=5", some less efficient than others:

```

while (t ≠ a + 5) do
  t ← rand()
od
a ← t
  
```

While optimization can occur throughout the translation process, machine-independent transformations are typically relegated to the middle-end, while instruction selection and other machine-specific activities are pushed into code generation.

Bootstrapping a compiler

Often, a compiler is written in it "itself". That is, a compiler for PASCAL may be written in PASCAL. How does this work?

Initial Compiler for L on Machine M

1. The compiler can be written in a small subset of L , even though the compiler translates the full language.
2. A throw-away version of the subset language is implemented on M . Call this compiler α .
3. The L compiler can be compiled using the subset compiler, to generate a full compiler β .
4. The L compiler can also compile itself. The resulting object γ can be compared with β for verification.

Porting the Compiler

1. On machine M , the code generator for the full compiler is changed to target machine N .
2. Any program in L can now be cross-compiled from M to N .
3. The compiler can also be cross-compiled to produce an instance of γ that runs on machine N .

If the run-time library is mostly written in L , or in an intermediate language of β , then these can also be translated for N using the cross-compiler.

What else does a compiler do?

```
if (p)
    a = b + (c
else {d = f;
q = r;
```

Error detection. Strict language rules, consistently enforced by a compiler, increase the likelihood that a compiler-approved source program is bug-free.

Error diagnosis. Compilers can often assist the program author in addressing errors.

Error repair. Some ambitious compilers go so far as to insert or delete text to render the program executable.

```
for (i=1; i<=n; ++i)
{
    a[i] = b[i] + c[i]
}
```

Program optimization. The target produced by a compiler must be "observably equivalent" to the source interpretation. An optimizing compiler attempts to minimize the resource constraints (typically time and space) required by the target program.

Program instrumentation. The target program can be augmented with instructions and data to provide information for run-time debugging and performance analysis. Language features not checkable at compile-time are often checked at run-time by code inserted by the compiler.

Sophisticated error repair may include symbol insertion, deletion, and use of indentation structure.

Program optimization can significantly decrease the time spent on array index arithmetic. Since subscript ranges cannot in general be checked at compile-time, run-time tests may be inserted by the compiler.

Compiler design points – aquatic analogies

Powerboat Turbo-?. These compilers are fast, load-and-go. They perform little optimization, but typically offer good diagnostics and a good programming environment (sporting a good debugger). These compilers are well-suited for small development tasks, including small student projects.

Sailboat BCPL, Postscript. These compilers can do neat tricks but they require skill in their use. The compilers themselves are often small and simple, and therefore easily ported. They can assist in bootstrapping larger systems.

Tugboat C++ preprocessor, RATFOR. These compilers are actually front-ends for other (typically larger) back-ends. The early implementations of C++ were via a preprocessor.

Barge Industrial-strength. These compilers are developed and maintained with a company's reputation on the line. Commercial systems use these compilers because of their integrity and the commitment of their sponsoring companies to address problems. Increasingly these kinds of compilers are built by specialty houses such as Rational, KAI, etc.

Ferry Gnu compilers. These compilers are available via a General Public License from the Free Software Foundation. They are high-quality systems and can be built upon without restriction.

Another important design issue is the extent to which a compiler can respond *incrementally* to changes.

Compilers are taking over the world!

While compilers most prevalently participate in the translation of programming languages, some form of compiler technology appears in many systems:

Text processing Consider the "x-roff" text processing pipe:

PIC → TBL → EQN → TROFF

or the \LaTeX pipe:

$\text{\LaTeX} \rightarrow \text{\TeX}$

each of which may produce

DVI → POSTSCRIPT

Silicon compilers Such systems accept circuit specifications and compile these into VLSI layouts. The compilers can enforce the appropriate "rules" for valid circuit design, and circuit libraries can be referenced like modules in software library.

Compiler design vs. programming language design

Programming languages have	So compilers offer
Non-locals Recursion Dynamic Storage Call-by-name Modular structure Dynamic typing	Displays, static links Dynamic links Garbage collection Thunks Interprocedural analysis Static type analysis

It's expensive for a compiler to offer	So some languages avoid that feature
Non-locals Call-by-name Recursion Garbage collection	C C, PASCAL FORTRAN 66 C

In general, *simple* languages such as C, PASCAL, and SCHEME have been more successful than complicated languages like PL/1 and ADA.

Language design for humans

Procedure *foo*(*x*, *y*)

declare

x, *y* **integer**

a, *b* **integer**

p* **integer

p ← *rand*() ? &*a* : &*b*

**p* ← *x* + *y*

end

Syntactic simplicity. Syntactic signposts are kept to a minimum, except where aesthetics dictate otherwise: parentheses in C, semicolons in PASCAL.

Resemblance to mathematics. Infix notation, function names.

Flexible internal structures. Nobody would use a language in which one had to predeclare how many variables their program needed.

Freedom from specifying side-effects. What happens when *p* is dereferenced?

Programming language design is often a compromise between ease of use for humans, efficiency of translation, and efficiency of target code execution.

Language design for machines

```
(SymbolTable
  (NumSymbols 5)
  (Symbol
    (SymbolName x)
    (SymbolID 1)
  )
  (Symbol
    (SymbolName y)
    (SymbolID 2)
  )
  ...
)
(AliasRelations
  (NumAliasRelations 1)
  (AliasRelation
    (AliasID 1)
    (MayAliases 2 a b)
  )
)
)

(NodeSemantics
  (NodeID 2)
  (Def
    (DefID 2)
    (SymbID ?)
    (AliasWith 1)
    (DefValue
      (+
        (Use
          (UseID 1)
          (SymbID x)
        )
        (Use
          (UseID 2)
          (SymbID y)
        )
      )
    )
  )
)
)
```

We can require much more of our intermediate languages, in terms of details and syntactic form.

Compilers and target instruction sets

How should we translate $X = Y + Z$

In the course of its code generation, a simple compiler may use only 20% of a machine's potential instructions, because anomalies in an instruction set are difficult to "fit" into a code generator.

Consider two instructions

```
ADDREG  $R_1 R_2$   $R_1 \leftarrow R_1 + R_2$ 
ADDMEM  $R_1 Loc$   $R_1 \leftarrow R_1 + *Loc$ 
```

Each instruction is *destructive* in its first argument, so Y and Z would have to be refetched if needed.

```
LOAD    1 Y
ADDMEM  1 Z
STORE   1 X
```

A simpler model would be to do all arithmetic in registers, assuming a *nondestructive* instruction set, with a reserved register for results (say, R_0):

```
LOAD    1 Y
LOAD    2 Z
LOADREG 0 1
ADDREG  0 2
STORE   0 X
```

This code preserves the value of Y and Z in their respective registers.

A popular approach is to generate code assuming the nondestructive paradigm, and then use an instruction selector to optimize the code, perhaps using destructive operations.

Current wisdom on compilers and architecture

Architects should design “orthogonal” RISC instruction sets, and let the optimizer make the best possible use of these instructions. Consider the program

```
for  $i \leftarrow 1$  to 10 do  $X \leftarrow A[i]$ 
```

where A is declared as a 10-element array (1...10).

The VAX has an instruction essentially of the form

$$\text{Index}(A, i, \text{low}, \text{high})$$

with semantics

```
if ( $\text{low} \leq i \leq \text{high}$ ) then
  return ( $A + 4 \times i$ )
else
  return (error)
fi
```

Internally, this instruction requires two tests, one multiplication, and one addition.

However, notice that the loop does not violate the array bounds of A . Moreover, in moving from $A[i]$ to $A[i + 1]$, the new address can be calculated by adding 4 to the old address.

While the use of an *Index* instruction may seem attractive, better performance can be obtained by providing smaller, faster instructions to a compiler capable of optimizing their use.

A small example of language translation

$L(\text{Add}) =$
sums of two digits, expressed
expressed in the usual (infix)
notation

That's not very formal. What do we mean by this?

$$\{0 + 4, 3 + 7, \dots\}$$

```
input(s)
case(s)
  of("0+0") return(OK)
  ...
  of("9+9") return(OK)
  default return(BAD)
endcase
```

The program shown on the left *recognizes* the *Add* language. Suppose we want to *translate* strings in *Add* into their sum, expressed base-4.

```
input(s)
case(s)
  of("0+0") return("0")
  ...
  of("5+7") return("30")
  ...
  of("9+9") return("102")
  default oops(BAD)
endcase
```

A language is a set of strings. With 100 possibilities, we could easily list all strings in this (small) language. This approach seems like lots of work, especially for languages with infinite numbers of strings, like C. We need a finite specification for such languages.

Grammars

The grammar below *generates the Add* language:

$$\begin{array}{l} S \rightarrow D + D \\ D \rightarrow 0 \\ \quad | 1 \\ \quad | 2 \\ \quad \vdots \\ \quad | 9 \end{array}$$

A grammar is formally

$$G = (V, \Sigma, P, S)$$

where

V is the set of nonterminals. These appear on the left side of rules.

Σ is an alphabet of terminal symbols, that cannot be rewritten.

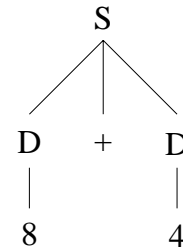
P is a set of rewrite rules.

S is the *start or goal* symbol.

The process by which a terminal string is created is called a *derivation*.

$$\begin{array}{l} S \Rightarrow D + D \\ \Rightarrow 8 + D \\ \Rightarrow 8 + 4 \end{array}$$

This is a *leftmost* derivation, since a string of nonterminals is rewritten from the left. A tree illustrates how the grammar and the derivation *structure* the string:

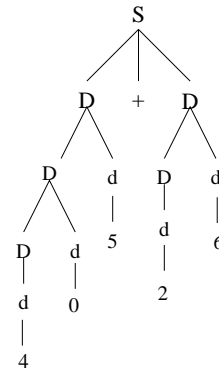


The above could be called a *derivation tree*, a (concrete) *syntax tree*, or a *parse tree*.

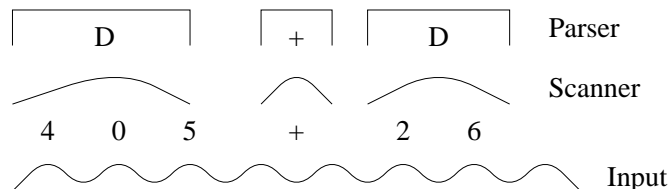
Strings in $L(G)$ are constructed by rewriting the symbol S according to the rules of P until a terminal string is derived.

Sums of two numbers

Consider the set of strings that represent the sum of two numbers, such as $405 + 26$. We could rewrite the grammar, as shown below:

$$\begin{array}{l} S \rightarrow D + D \\ D \rightarrow D d \\ \quad | d \\ d \rightarrow 0 \\ \quad | 1 \\ \quad \vdots \\ \quad | 9 \end{array}$$


Another solution would be to have a separate *tokenizing* process feed "D"s to the grammar, so that the grammar remains unchanged.



Scanners

Scanners are often the ugliest part of a compiler, but when cleverly designed, they can greatly simplify the design and implementation of a parser.

Typical tasks for a scanner:

- Recognize reserved keywords.
- Find integer and floating-point constants.
- Ignore comments.
- Treat blank space appropriately.
- Find string and character constants.
- Find identifiers (variables).

The C statement

```
if (++x==5) foo(3);
```

might be tokenized as

```
if ( ++ ID == 5 ) ID ( int )
```

Unusual tasks for a scanner:

- In (older) FORTRAN, blanks are optional. Thus, the phrases

```
DO10I=1,5 and DO10I=1.5
```

are distinguished only by the comma vs. the decimal. The first statement is the start of a DO loop, while the second statement assigns the variable DO10I.

- In C, variables can be declared by built-in or by user-defined types. Thus, in

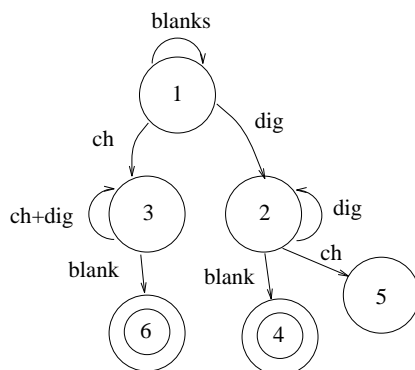
```
foo x,y;
```

the C grammar needs to know that `foo` is a type name, and not a variable name.

The balance of work between scanner and parser is typically dictated by restrictions of the parsing method and by a desire to make the grammar as simple as possible.

Scanners and Regular Languages

Most scanners are based on a simple computational model called the *finite-state automaton*.



These machines recognize *regular languages*.

To implement a finite-state transducer one begins with a GOTO table that defines transitions between states:

State	GOTO table		
	ch	dig	blank
1	3	2	1
2	5	2	4
3	3	3	6
4	3	2	4
5	5	5	5
6	3	2	6

which is processed by the driver

```

state ← 1
while (true) do
  c ← NextSym()
  /* Do action ACTION[state][c] */
  state ← GOTO[state][c]
od
  
```

Notice the similarity between states 1, 4, and 6.

Transduction

While the finite-state mechanism *recognizes* appropriate strings, action must now be taken to construct and supply tokens to the parser. Between states, actions are performed as prescribed by the ACTION table shown below.

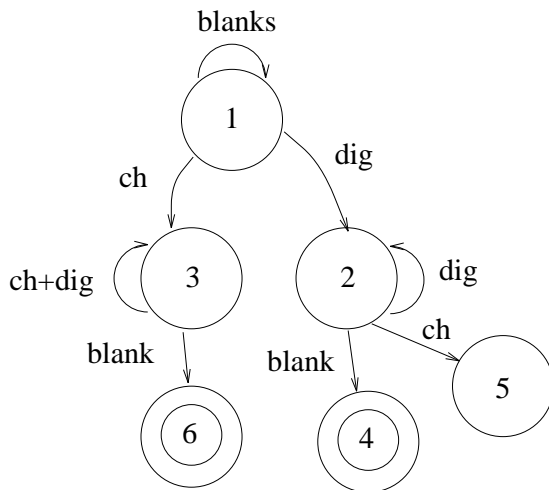
State	ACTION table		
	ch	dig	blank
1	1	2	3
2	4	5	6
3	7	7	8
4	1	2	3
5	4	4	4
6	1	2	3

Actions

1. $ID = ch$
2. $Num = dig$
3. Do nothing
4. Error
5. $Num = 10 \times Num + dig$
6. return NUM
7. $ID = ID || ch$
8. return ID

Technically, the ability to perform arbitrary actions makes our tokenizer more powerful than a finite-state automaton. Nonetheless, the underlying mechanism is quite simple, and can in fact be automatically generated. . . .

Regular grammars



- | | | | |
|---|---|-----------|---|
| 1 | → | blank | 1 |
| | | ch | 3 |
| | | dig | 2 |
| 2 | → | dig | 2 |
| | | ch | 5 |
| | | blank | 4 |
| 3 | → | ch | 3 |
| | | dig | 3 |
| | | blank | 6 |
| 4 | → | λ | |
| 6 | → | λ | |

In a regular grammar, each rule is of the form

$$A \rightarrow a A$$

$$A \rightarrow a$$

where $A \in V$ and $a \in (\Sigma \cup \{\lambda\})$.

LEX as a scanner

First, define character classes:

```
ucase  [A-Z]
lcase  [a-z]
letter ({ucase}|{lcase})
zero   0
nonzero [1-9]
sign   [+ -]
digit  ({zero}|{nonzero})
blanks [ \t\f]
newline \n
```

Next, specify patterns and actions:

<code>{L}({L} {D})*</code>	<code>{ String(yytext); return(ID); }</code>
<code>``++``</code>	<code>{ return(IncOP); }</code>

In selecting which pattern to apply, LEX uses the following rules:

1. LEX always tries for the longest match. If any pattern can “keep going” then LEX will keep consuming input until that pattern finishes or “gives up”. This property frequently results in buffer overflow for improperly specified patterns.
2. LEX will choose the pattern and action that succeeds by consuming the most input.
3. If there are ties, then the pattern specified earliest to LEX wins.

The notation used above is *regular expression* notation, which allows for choice, catenation, and repeats. One can show by construction that any language accepted by a finite-state automaton has an equivalent regular expression.

A comment

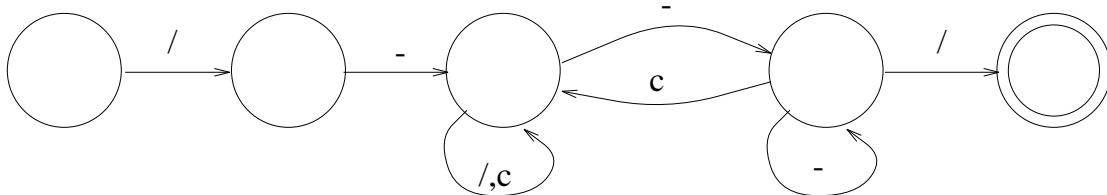
An interesting example is the C-like *comment* specification, which might be tempting to specify as:

```
"/- " . * "-/"
```

But in a longest match, this pattern will match the beginning of the first comment to the end of the last comment, and everything in between. If LEX's buffers don't overflow, most of the input program will be ignored by this faulty specification.

A better specification can be determined as follows:

1. Start with the wrong specification.
2. Construct the associated deterministic FSA.
3. Edit the FSA to cause acceptance at the end of the first comment (shown below).
4. Construct the regular expression associated with the resulting FSA.



with the corresponding regular expression

```
/- [ (/|c)* -(-)* c ]* (/|c)* -(-)* /
```

Teaching regular languages and scanners

Classroom

1. Motivate the study with examples from programming languages and puzzles (THINK-A-DOT, etc.).
2. Present deterministic FSA (DFA).
3. Present nondeterministic FSA (NFA).
4. Show how to construct NFAs from regular expressions.
5. Show good use of the empty string (λ or ϵ).
6. Eliminate the empty string.
7. Eliminate nondeterminism.
8. Minimize any DFA.
9. Construction of regular expressions from DFA.
10. Show the correspondence between regular grammars and FSAs.
11. The pumping lemma and nonregular languages.

Projects and Homework

1. Implement THINK-A-DOT.
2. Check if a YACC grammar is regular. If so, then emit the GOTO table for a finite-state driver.
3. Augment the above with ACTION clauses.
4. Process a YACC file for reserved keyword specifications:


```
%token <rk> then
and generate the appropriate pattern and action for recognizing these:
"then" { return(THEN); }
```
5. Show that regular expression notation is itself not regular.

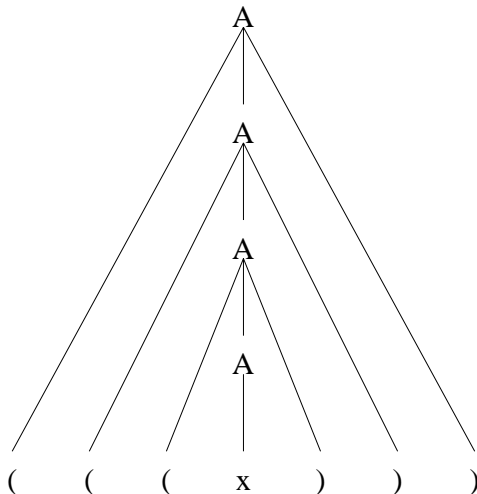
Some useful resources: [24, 28, 16, 2, 26].

Nonregular languages

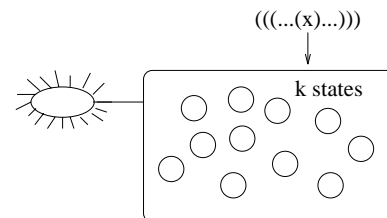
To grow beyond regular languages, we now allow a rule's right-hand side to contain any string of terminals or nonterminals.

$$A \rightarrow (A) \\ \quad | \quad x$$

describes the language $(^n x)^n$.



Suppose that some finite-state machine M of k states can recognize $\{ (^n x)^n \}$.



Consider the input string $z = (^k x)^k$. After processing the k^{th} '(', some state must have been visited twice. By repeating the portion of z causing this loop, we obtain a string

$$(^k (^j x)^k), k \geq 0, j > 0$$

which is not in the language, but is accepted by M .

Since the proof did not depend on any particular k , we have shown that no finite-state machine can accept exactly this language.

Some more sums

Grammar

$$E \rightarrow E + E$$

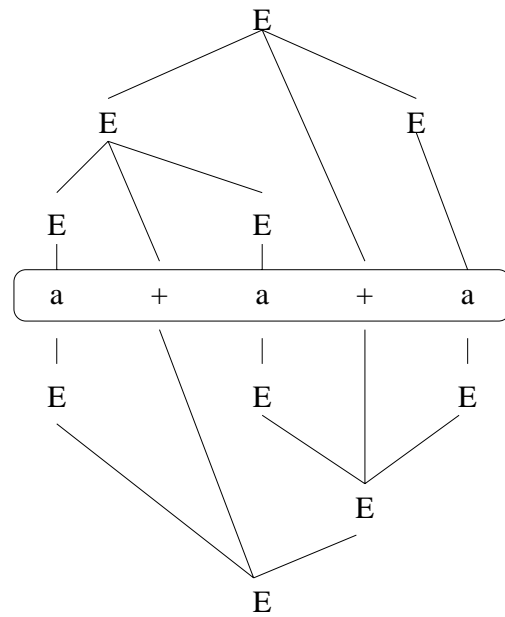
|
a

Leftmost derivation

$$\begin{aligned} E &\Rightarrow \boxed{E} + E \\ &\Rightarrow \boxed{E + E} + E \\ &\Rightarrow a + E + E \\ &\Rightarrow a + a + E \\ &\Rightarrow a + a + a \end{aligned}$$

Another leftmost derivation

$$\begin{aligned} E &\Rightarrow \boxed{E} + E \\ &\Rightarrow \boxed{a} + E \\ &\Rightarrow a + E + E \\ &\Rightarrow a + a + E \\ &\Rightarrow a + a + a \end{aligned}$$

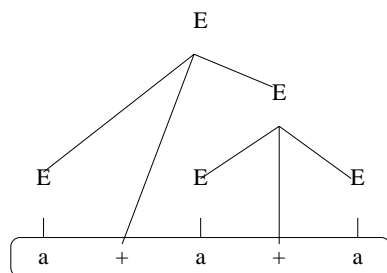


If the same string has two parse trees by a grammar G , then G is *ambiguous*. Equivalently, there are two distinct leftmost derivations of some string. Note that the language above is *regular*.

Ambiguity

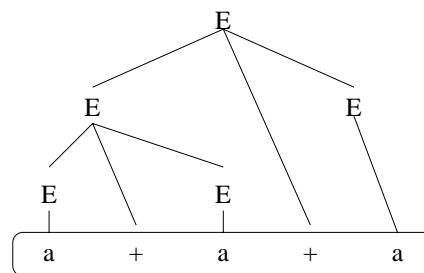
The parse tree below structures the input string as

$(a + (a + a))$



The parse tree below structures the input string as

$((a + a) + a)$



- With addition, the two expressions may be semantically the same. What if the a 's were the operands of subtraction?
- How could a compiler choose between multiple parse trees for a given string?
- Unfortunately, there is (provably) no mechanical procedure for determining if a grammar is ambiguous; this is a job for human intelligence. However, compiler construction tools such as YACC can greatly facilitate the location and resolution of grammar ambiguities.
- It's important to emphasize the difference between a *grammar* being ambiguous, and a *language* being (inherently) ambiguous. In the former case, a different grammar may resolve the ambiguity; in the latter case, there exists no unambiguous grammar for the language.

Syntactic ambiguity

A great source of humor in the English language arises from our ability to construct interesting syntactically ambiguous phrases:

1. I fed the elephant in my tennis shoes.

What does "in my tennis shoes" modify?

- (a) Was I wearing my tennis shoes while feeding the elephant?
- (b) Was the elephant wearing or inside my tennis shoes?

2. The purple people eater. What is purple?

- (a) Is the eater purple?
- (b) Are the people purple?

Suppose we modified the grammar for C, so that any {...} block could be treated as a primary value.

```
{ int i; i=3*5; } + 27;
```

would seem to have the value 42. But if we just rearrange the white space, we can get

```
{int i; i=3*5; }  
+27;
```

which represents two statements, the second of which begins with a unary plus.

A good assignment along these lines is to modify the C grammar to allow this simple language extension, and ask the students to determine what went wrong. The students should be fairly comfortable using YACC before trying this experiment.

Semantic ambiguity

In English, we can construct sentences that have only one parse, but still have two different meanings:

1. Milk drinkers turn to powder. Are more

milk drinkers using powdered milk, or are milk drinkers rapidly dehydrating?

2. I cannot recommend this student too highly. Do words of praise escape me, or am I

unable to offer my support.

In programming languages, the language standard must make the meaning of such phrases clear, often by applying elements of context.

For example, the expression

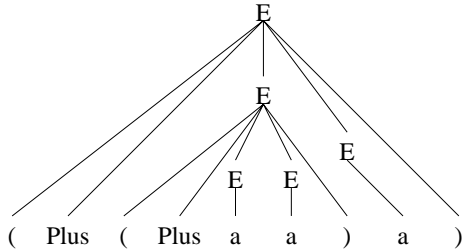
$$a + b$$

could connote an integer or floating-point sum, depending on the types of a and b .

A nonambiguous grammar

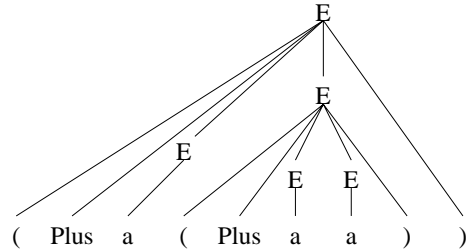
$$\begin{array}{l}
 E \rightarrow (\text{Plus } E E) \\
 \quad | (\text{Minus } E E) \\
 \quad | a
 \end{array}$$

It's interesting to note that the above grammar, intended to generate LISP-like expressions, is not ambiguous.



is the prefix equivalent of

$$((a + a) + a)$$



is the prefix equivalent of

$$(a + (a + a))$$

These are two *different strings* from this language, each associated explicitly with a particular grouping of the terms. Essentially, the parentheses are syntactic sentinels that simplify construction of an unambiguous grammar for this language.

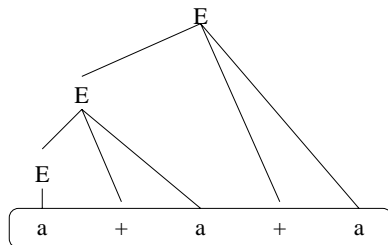
Addressing ambiguity

$$\begin{array}{l}
 E \rightarrow E + E \\
 \quad | a
 \end{array}$$

We'll try to rewrite the above grammar, so that in a (leftmost) derivation, there's only one rule choice that derives longer strings.

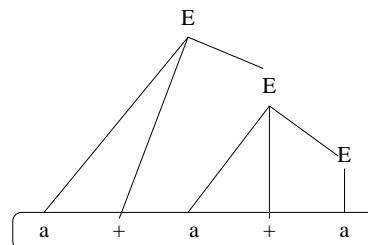
$$\begin{array}{l}
 E \rightarrow E + a \\
 \quad | E - a \\
 \quad | a
 \end{array}$$

These rules are *left recursive*, and the resulting derivations tend to associate operations from the left:



$$\begin{array}{l}
 E \rightarrow a + E \\
 \quad | a - E \\
 \quad | a
 \end{array}$$

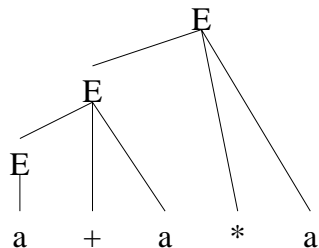
The grammar is still unambiguous, but strings are now associated from the right:



Addressing ambiguity (cont'd)

Our first try to expand our grammar might be:

$$\begin{array}{l}
 E \rightarrow E + a \\
 \quad | \quad E * a \\
 \quad | \quad a
 \end{array}$$



The above parse tree does not reflect the usual precedence of * over +.

To obtain *sums of products*, we revise our grammar:

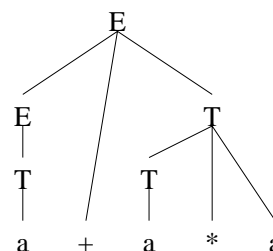
$$\begin{array}{l}
 E \rightarrow E + T \\
 \quad | \quad T
 \end{array}$$

This generates strings of the form

$$T + T + \dots + T$$

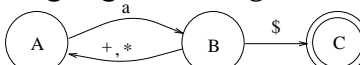
We now allow each *T* to generate strings of the form $a * a * \dots * a$

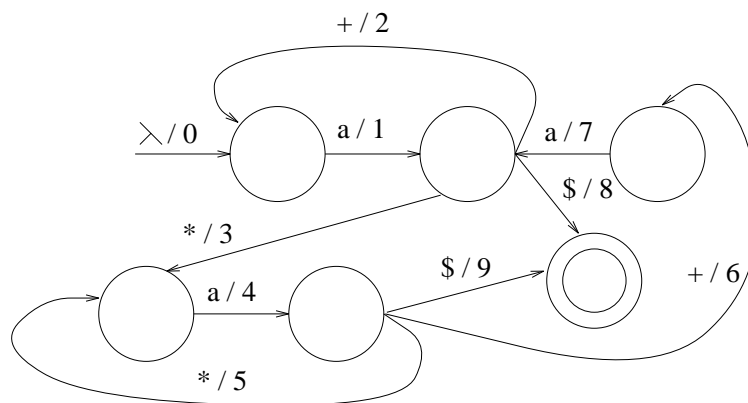
$$\begin{array}{l}
 E \rightarrow E + T \\
 \quad | \quad T \\
 T \rightarrow T * a \\
 \quad | \quad a
 \end{array}$$



Translating two-level expressions

Since our language is still *regular*, a finite-state machine could do the job. While the

machine  could do the job, there's not enough "structure" to this machine to accomplish the prioritization of * over +. However, the machine below can do the job.



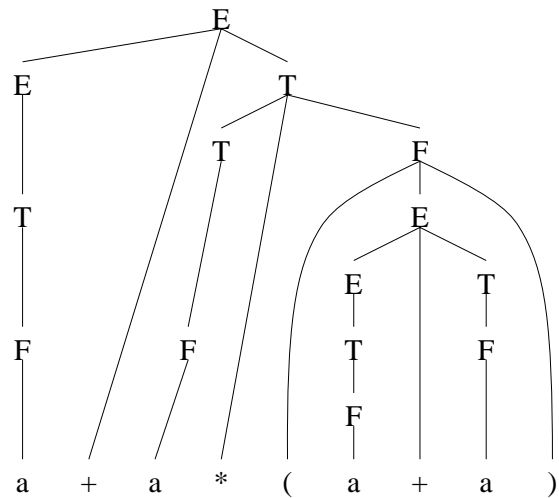
0 $Sum = 0$	5 $Prod = Prod \times Acc$
1 $Acc = a$	6 $Sum = Sum + (Prod \times Acc); Prod = 1$
2 $Sum = Sum + Acc$	7 $Acc = a$
3 $Prod = Prod \times Acc$	8 $Sum = Sum + Acc$
4 $Acc = a$	9 $Sum = Sum + (Prod \times Acc); Prod = 1$

Let's add parentheses

While our grammar currently structures inputs appropriately for operator priorities, parentheses are typically introduced to override default precedence. Since we want a parenthesized expression to be treated "atomically", we now generate sums of products of parenthesized expressions.

$$\begin{aligned} E &\rightarrow E + T \\ &\quad | \quad T \\ T &\rightarrow T * F \\ &\quad | \quad F \\ F &\rightarrow (E) \\ &\quad | \quad a \end{aligned}$$

This grammar generates a nonregular language. Therefore, we need a more sophisticated "machine" to parse and translate its generated strings.



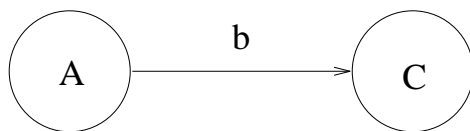
The grammar we have developed thus far is the textbook "expression grammar". Of course, we should make a into a nonterminal that can generate identifiers, constants, procedure calls, etc.

Beyond finite-state machines

For a rule of the form

$$A \rightarrow b C$$

we developed a finite-state mechanism of the form



After arrival at C , there is no need to remember how we got there.

Now, with a rule such as

$$F \rightarrow (E)$$

we cannot just arrive at an E and forget that we need exactly one closing parenthesis for each opening one that got us there.

Instead of "going to" a state E based on consuming an opening parenthesis, suppose we called a procedure E to consume all input ultimately derived from the nonterminal:

Procedure $F()$

 call $Expect(OpenParen)$

 call $E()$

 call $Expect(CloseParen)$

end

This style of parser construction is called *recursive descent*. The procedure associated with each nonterminal is responsible for directing the parse through the right-hand side of the appropriate production.

1. What about rules that are left-recursive?
2. What happens if there is more than one rule associated with a nonterminal?