Lexical Analysis — Part II: Constructing a Scanner from Regular Expressions

Lecture 6
CS434
Spring 2005

Department of Computer Science
University of Alabama
Joel Jones

What can be so hard? (Fortran 66/77)

```
      INTEGERFUNCTIONA
      PARAMETER(A=6,B=2)
      IMPLICIT CHARACTER*(A-B)(A-B)
      INTEGER FORMAT(10), IF(10), DO9E1
100   FORMAT(4H)=(3)
200   FORMAT(4   )=(3)
      DO9E1=1
      DO9E1=1,2
  9   IF(X)=1
      IF(X)H=1
      IF(X)300,200
300   CONTINUE
      END
C     THIS IS A "COMMENT CARD"
     $ FILE(1)
      END
```

How does a compiler scan this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scanable language
- Second pass is normal scanner

Example due to Dr. F.K. Zadeck

Building Faster Scanners from the DFA

Table-driven recognizers waste effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in action()
- Branch back to the top

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
   state ← δ(state,char);
   call action(state,char);
   char ← next character;

if T(state) = final then
   report acceptance;
else
   report failure;
```

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

Building Faster Scanners from the DFA

A direct-coded recognizer for r Digit Digit*

```
        goto s0;
s0:     word ← Ø;
        char ← next character;
        if (char = 'r')
          then goto s1;
          else goto se;
s1:     word ← word + char;
        char ← next character;
        if ('0' ≤ char ≤ '9')
          then goto s2;
          else goto se;
```

```
s2:     word ← word + char;
        char ← next character;
        if ('0' ≤ char ≤ '9')
          then goto s2;
          else if (char = eof)
            then report
success;
            else goto se;
se:     print error message;
        return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Building Faster Scanners

Hashing keywords versus encoding them directly

- Some (well-known) compilers recognize keywords as identifiers and check them in a hash table
- Encoding keywords in the DFA is a better idea
  - → O(1) cost per transition
  - → Avoids hash lookup on each identifier

It is hard to beat a well-implemented DFA scanner

Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting

Some Points of Disagreement with EAC

- Table-driven scanners are not fast
  - → EaC doesn't say they are slow; it says you can do better
- Faster code can be generated by embedding scanner in code
  - → This was shown for both LR-style parsers and for scanners in the 1980s
- Hashed lookup of keywords is slow
  - → EaC doesn't say it is slow. It says that the effort can be folded into the scanner so that it has no extra cost. Compilers like GCC use hash lookup. A word must fail in the lookup to be classified as an identifier. With collisions in the table, this can add up. At any rate, the cost is unneeded, since the DFA can do it for O(1) cost per character.

Building Faster Scanners from the DFA

Table-driven recognizers waste a lot of effort

index • Read (& classify) the next character

index • Find the next state

• Assign to the state variable

index • Trip through case logic in action()

• Branch back to the top

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
  state ← δ(state,char);
  call action(state,char);
  char ← next character;

if T(state) = final then
  report acceptance;
else
  report failure;
```

We can do better

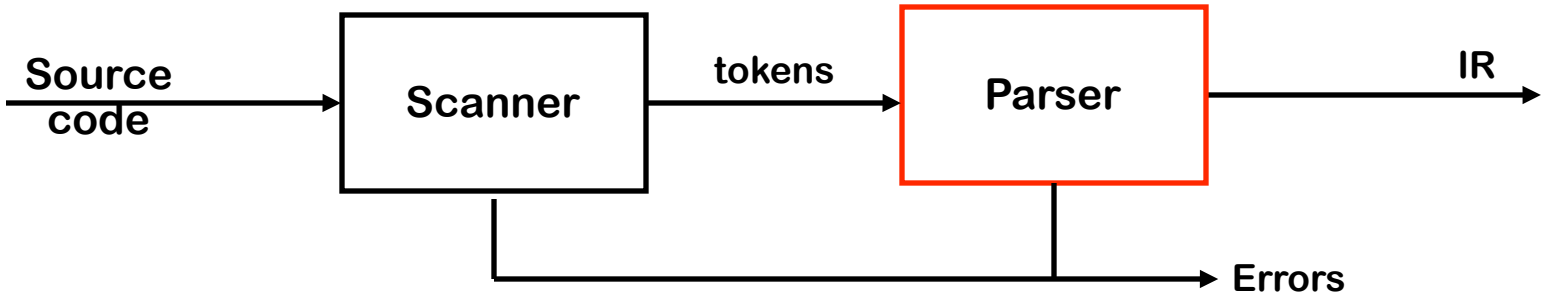
• Encode state & actions in the code

• Do transition tests locally

• Generate ugly, spaghetti-like code

• Takes (many) fewer operations per input character

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

Think of this as the mathematics of diagramming sentences

The Study of Parsing

The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar *G*
- Need an algorithm for testing membership in *L*(*G*)
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap

1. Context-free grammars and derivations
2. Top-down parsing
   - → Hand-coded recursive descent parsers
3. Bottom-up parsing
   - → Generated LR(1) parsers

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

SheepNoise → SheepNoise baa
| baa

This CFG defines the set of noises sheep normally make

It is written in a variant of Backus-Naur form

Formally, a grammar is a four tuple, G = (S,N,T,P)

- S is the start symbol (set of strings in L(G))
- N is a set of non-terminal symbols (syntactic variables)
- T is a set of terminal symbols (words)
- P is a set of productions or rewrite rules $(P : N \rightarrow (N \cup T)^{+})$

Example due to Dr. Scott K. Warren

Deriving Syntax

We can use the SheepNoise grammar to create sentences

→ use the productions as rewriting rules

Rule	Sentential Form
—	*SheepNoise*
2	baa

Rule	Sentential Form
—	*SheepNoise*
1	*SheepNoise* baa
2	baa baa

Rule	Sentential Form
—	*SheepNoise*
1	*SheepNoise* baa
1	*SheepNoise* baa baa
2	baa baa baa

And so on . . .

While it is cute, this example quickly runs out of intellectual steam ...

A More Useful Grammar

To explore the uses of CFGs,we need a more complex grammar

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
2	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

- Such a sequence of rewrites is called a derivation
- Process of discovering a derivation is called parsing

We denote this derivation: Expr ⇒* id – num * id

Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- Leftmost derivation — replace leftmost NT at each step
- Rightmost derivation — replace rightmost NT at each step

These are the two systematic derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a leftmost derivation

- Of course, there is also a rightmost derivation
- Interestingly, it turns out to be different

The Two Derivations for x – 2 * y

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	⟨id,x⟩ *Op Expr*
5	⟨id,x⟩ – *Expr*
1	⟨id,x⟩ – *Expr Op Expr*
2	⟨id,x⟩ – ⟨num,2⟩ *Op Expr*
6	⟨id,x⟩ – ⟨num,2⟩ * *Expr*
3	⟨id,x⟩ – ⟨num,2⟩ * ⟨id,y⟩

Leftmost derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* ⟨id,y⟩
6	*Expr* * ⟨id,y⟩
1	*Expr Op Expr* * ⟨id,y⟩
2	*Expr Op* ⟨num,2⟩ * ⟨id,y⟩
5	*Expr* – ⟨num,2⟩ * ⟨id,y⟩
3	⟨id,x⟩ – ⟨num,2⟩ * ⟨id,y⟩

Rightmost derivation

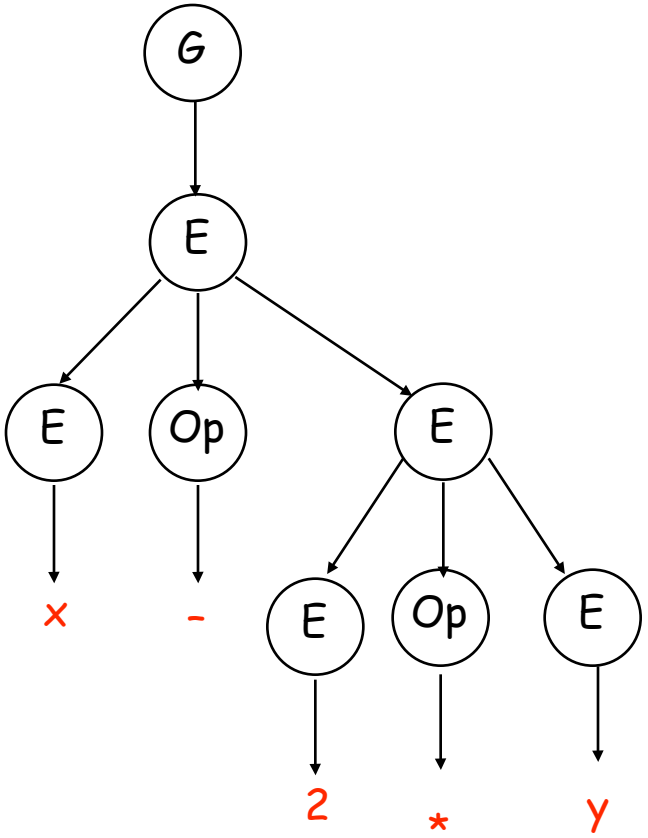
In both cases, Expr ⇒* id – num * id

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

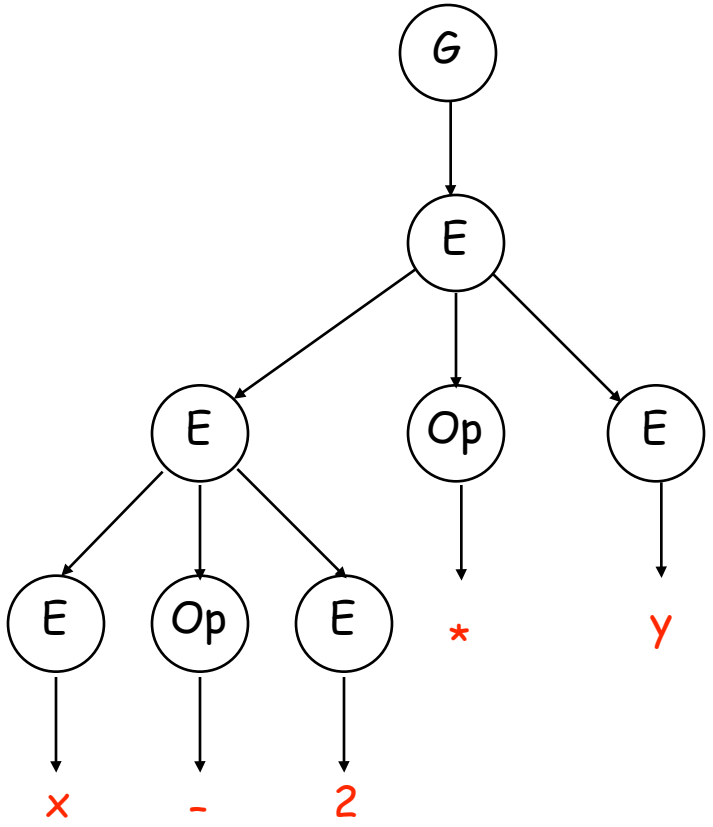


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* <id,y>
6	*Expr* * <id,y>
1	*Expr Op Expr* * <id,y>
2	*Expr Op* <num,2> * <id,y>
5	*Expr* - <num,2> * <id,y>
3	<id,x> - <num,2> * <id,y>



This evaluates as (x - 2) * y

Derivations and Precedence

These two derivations point out a problem with the grammar:

It has no notion of precedence, or implied order of evaluation

To add precedence

- Create a non-terminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first (level one)
- Subtraction and addition, next (level two)

Derivations and Precedence

Adding the standard algebraic precedence produces:

	1	*Goal*	→	*Expr*
level two	2	*Expr*	→	*Expr* + *Term*
	3		|	*Expr* - *Term*
	4		|	*Term*
level one	5	*Term*	→	*Term* * *Factor*
	6		|	*Term* / *Factor*
	7		|	*Factor*
	8	*Factor*	→	number
	9		|	id

This grammar is slightly larger

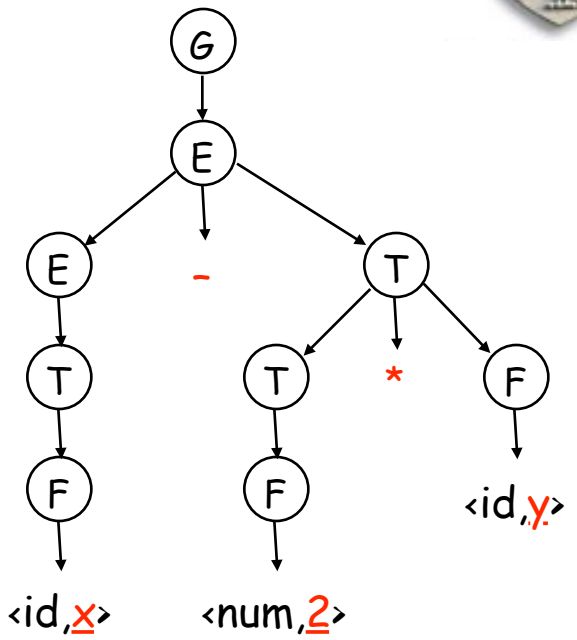
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses x - 2 * y

Derivations and Precedence

Rule	Sentential Form
—	*Goal*
1	*Expr*
3	*Expr* - *Term*
5	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
7	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
4	*Term* - <num,2> * <id,y>
7	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

The rightmost derivation



Its parse tree

This produces x - (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.

Ambiguous Grammars

Our original expression grammar had other problems

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
1	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

- This grammar allows multiple leftmost derivations for x - 2 * y
- Hard to automate derivation if > 1 choice
- The grammar is ambiguous

different choice than the first time

Two Leftmost Derivations for x – 2 * y

The Difference:

- Different productions chosen on the second step

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
(3)	<id,x> *Op Expr*
5	<id,x> – *Expr*
1	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

Original choice

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
(1)	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> – *Expr Op Expr*
2	<id,x> – <num,2> *Op Expr*
6	<id,x> – <num,2> * *Expr*
3	<id,x> – <num,2> * <id,y>

New choice

- Both derivations succeed in producing x - 2 * y

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the if-then-else problem

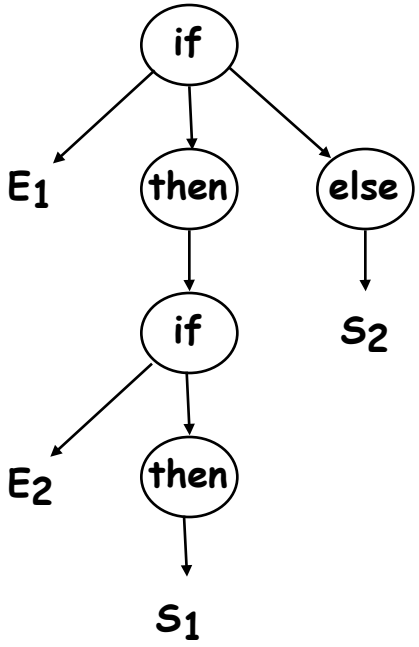
```
Stmt → if Expr then Stmt
     | if Expr then Stmt else Stmt
     | … other stmts …
```

This ambiguity is entirely grammatical in nature

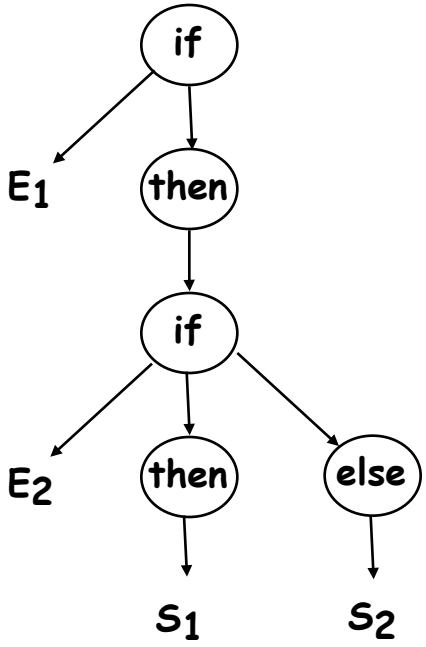
Ambiguity

This sentential form has two derivations

if Expr$_1$ then if Expr$_2$ then Stmt$_1$ else Stmt$_2$



production 2, then production 1



production 1, then production 2

Ambiguity

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if (common sense rule)

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

Intuition: a NoElse always has no else on its last cascaded else if statement

With this grammar, the example has only one derivation

Ambiguity

if Expr$_1$ then if Expr$_2$ then Stmt$_1$ else Stmt$_2$

Rule	*Sentential Form*
—	*Stmt*
2	*NoElse*
5	if *Expr* then *Stmt*
?	if E_1 then *Stmt*
1	if E_1 then *WithElse*
3	if E_1 then if *Expr* then *WithElse* else *WithElse*
?	if E_1 then if E_2 then *WithElse* else *WithElse*
4	if E_1 then if E_2 then S_1 else *WithElse*
4	if E_1 then if E_2 then S_1 else S_2

This binds the else controlling S_2 to the inner if

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

```
a = f(17)
```

In many Algol-like languages, f could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - → Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - → Knowledge of declarations, types, …
  - → Accept a superset of L(G) & check it by other means†
  - → This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- → Parsing techniques that "do the right thing"
- → i.e., always select the same derivation

†See Chapter 4