Outline

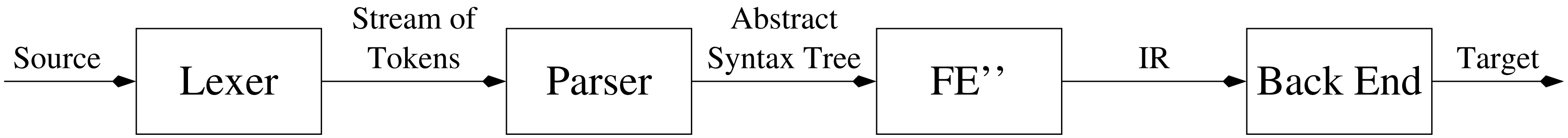
- Read Chapter 3, especially 3.2
- Recursive Descent Parser Generation

Syntax Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Recursive Descent Parsing

- Recall discussion on Context-Free Grammars: symbols (terminal, non-terminal), productions, derivations, etc.
- Can parse many grammars using algorithm called *recursive descent* parsing.
  - A.K.A.: *predictive parsing*
  - A.K.A.: *top-down parsing*
  - A.K.A.: *LL(1)* - Left-to-right parse, Leftmost-derivation, 1-symbol lookahead.
- One recursive function for each non-terminal.
- Each production becomes clause in function.

Example

Grammar:

non-terminals: `S, L, E`

terminals: `IF` (*if*), `THEN` (*then*), `ELSE` (*else*), `BEGIN` (*begin*),
`PRINT` (*print*), `END` (*end*), `SEMI` (*;*), `NUM`, `EQ` (=)

$S \rightarrow$ *if* E *then* S *else* S
$S \rightarrow$ *begin* S L
$S \rightarrow$ *print* E
$L \rightarrow$ *end*
$L \rightarrow$ *;* S L
$E \rightarrow$ *num* = *num*

```
datatype token = EOF | IF | THEN | ELSE | BEGIN |
                 PRINT | END | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok = t) then advance() else error()

fun S() = case !tok of
            IF    => (eat(IF); E(); eat(THEN); S();
                       eat(ELSE); S())
            BEGIN => (eat(BEGIN); S(); L())
            PRINT => (eat(PRINT); E())
            _     => error()
and L() = case !tok of
            END   => (eat(END))
            SEMI  => (eat(SEMI); S(); L())
            _     => error()
and E() =            (eat(NUM); eat(EQ); eat(NUM))
```

Another Example

Grammar:

$A \rightarrow S\ EOF$
$S \rightarrow id := E$
$S \rightarrow print\ (\ L\ )$

$E \rightarrow id$
$E \rightarrow num$
$L \rightarrow E$
$L \rightarrow L, E$

```
fun A() =                (S(); eat(EOF))
and S() = case !tok of
            ID      => (eat(ID); eat(ASSIGN); E())
            PRINT   => (eat(PRINT); eat(LPAREN);
                         L(); eat(RPAREN))
and E() = case !tok of
            ID      => (eat(ID))
            NUM     => (eat(NUM))
and L() = case !tok of
            ID      => (?????)
            NUM     => (?????)
```

Problem

- LL(1) Predictive parsing must guess which production to use based on seeing at most one symbol of the right hand side of the production.
- LR(1) parsing has read all of the symbols in the production right hand side and has placed them on the stack. Then, with one more symbol it makes its decision.
- If !tok = ID, parser cannot determine which of (1) or (2) to use:

$$L \rightarrow E \qquad (1)$$
$$L \rightarrow L, E \qquad (2)$$
$$E \rightarrow id$$

- E might derive *id* and therefore we should use rule 1 if we are parsing L
- L might derive E which might derive *id* and therefore we should use rule 2 if we are parsing L
- There is a conflict!

Problem

- Left recursive grammars:
  L $\rightarrow$ E
  L $\rightarrow$ L, E

  X $\rightarrow$ base1
  X $\rightarrow$ base2
  X $\rightarrow$ base3
  X $\rightarrow$ X repeat1
  X $\rightarrow$ X repeat2

- Grammars that are left-recursive can never be parsed by a predictive parser.

- Even if one grammar can't be parsed by a predictive parser, there may be another grammar for the same language that can.

Problem

- There is a procedure for eliminating left recursion:

$$\begin{array}{l} L \rightarrow E \\ L \rightarrow L, E \end{array} \quad \Longrightarrow \quad \begin{array}{l} L \rightarrow E\, M \\ M \rightarrow , E\, M \\ M \rightarrow \epsilon \end{array}$$

```
and L() = case !tok of
            ID     => (E(); M())
            NUM    => (E(); M())
and M() = case !tok of
            COMMA    => (eat(COMMA); E(); M())
            RPAREN   => ()
```

Left Recursion

In general, if our grammar looks like:

X $\rightarrow$ base
X $\rightarrow$ X repeat

It will generate strings that look like: “base repeat repeat repeat ...”

We can introduce a new non terminal X’ and rewrite it to:

X $\rightarrow$ base X’
X’ $\rightarrow$ repeat X’
X’ $\rightarrow$ ϵ

Algorithms for Constructing Predictive Parsers

To construct a predictive parser, what we want to know is which rule to apply when

- We have a particular non-terminal (say X) that needs to be expanded, and
- we see a particular terminal symbol (say a).

In general, we will apply the rule $X \to \gamma$ when

- a is the *first* symbol that can be generated by the string γ, or
- γ reduces to the empty string (is *nullable*) and a is the first symbol in a string that can *follow* X.

Algorithms for Constructing Predictive Parsers

Figuring out these First and Follow sets isn't quite as easy as it sounds at first. Consider this grammar:

$$
\begin{array}{ll}
1. & Y \rightarrow \\
2. & Y \rightarrow bb \\
3. & X \rightarrow c \\
4. & X \rightarrow Y Z \\
5. & Z \rightarrow d
\end{array}
$$

Non-terminal to expand	Terminal Seen	Rule to Apply
X	c	?
X	b	?
X	d	?

Algorithms for Constructing Predictive Parsers

To construct a predictive parser:

- Let γ be a string of terminal and non-terminal symbols
- Need to compute 3 values:

1. For each γ corresponding to RHS of production, must determine if γ can derive empty string (ϵ) $\rightarrow$ **nullable**.
2. For each γ corresponding to RHS of production, must determine set of all terminal symbols that can begin any string derived from γ $\rightarrow$ **first**(γ).
3. For each non-terminal X in grammar, must determine set of all terminal symbols that can immediately follow X in a derivation $\rightarrow$ **follow**(X).

Iterative Analysis Techniques

Iterative analysis techniques often come in handy when constructing compilers.

- Interative analysis applies when:
  - We want to compute the set of all objects that have some property P.
  - The property P is defined *inductively*. In other words, there are some base cases and some inductive cases that define the objects in the property:
    - Base cases: o_1, o_2, o_3 obviously have property P.
    - Inductive cases (inductive rules): if some objects (o_4, o_5 for example) have the property, then we can find other objects that also have the property.
  - We are guaranteed that the total number of objects in the set is *finite*. This guarantees that our interative analysis will eventually terminate.

Iterative Analysis Techniques

Iterative analysis algorithms have the form:

- Initialize the set S that we are looking for with all of the base cases.
- Apply the inductive rules over and over again (this is the iteration), adding all of the new objects until you reach a *fixed point*.

A fixed point is a set that does not change (does not gain new elements) when you apply any of the inductive rules.

- We will be able to construct Nullable, First, and Follow sets using iteration.
- Many program optimizations (dataflow analyses) often use iteration as well.
- Although the worst-case complexity of iterative techniques can be very bad, it often turns out that they are very efficient in practice and terminate in only a couple of iterations.

Computing Nullable Sets

Computation of Nullable:

- Base Case: If a non-terminal S can derive ϵ directly, it is nullable.
- Inductive Case: A non-empty string γ is nullable if every symbol $S \in \gamma$ is nullable.

The Nullable set can be computed first before the First and Follow sets.

Computing First Sets

- Base Case: If T is a terminal symbol, then first(T) = $\{T\}$.
- Inductive Case: If X is a non-terminal and $X \to Y_1 Y_2 Y_3 \ldots Y_n$, then

$$\begin{array}{l}
\text{first}(Y_1) \in \text{first}(X) \\
\text{first}(Y_2) \in \text{first}(X)\text{, if } Y_1 \text{ is nullable} \\
\text{first}(Y_3) \in \text{first}(X)\text{, if } Y_1, Y_2 \text{ is nullable} \\
\vdots \\
\text{first}(Y_n) \in \text{first}(X)\text{, if } Y_1, \ldots\ Y_{n-1} \text{ is nullable}
\end{array}$$

- Let $\gamma = S_1 S_2 \ldots S_k$. Then,

$$\text{first}(\gamma) = \begin{cases} \text{first}(S_1) \\ \text{first}(S_2)\text{, if } S_1 \text{ is nullable} \\ \text{first}(S_3)\text{, if } S_1, S_2 \text{ is nullable} \\ \vdots \\ \text{first}(S_k)\text{, if } S_1, S_2, \ldots, S_{k-1} \text{ is nullable} \end{cases}$$

(Notice computing First depends upon computing Nullable and Follow.)

Computing Follow Sets

Let X, Y be non-terminals; γ, γ_1, and γ_2 be strings of terminals and non-terminals

- Base Case: Follow(X) is the empty set, for all non-terminals X.
- Inductive Case: if grammar includes production: $X \rightarrow \gamma Y$
  $\rightarrow$ follow(X) $\in$ follow (Y).
- Inductive Case: if grammar includes production: $X \rightarrow \gamma_1 Y \gamma_2$
  $\rightarrow$ first (γ_2) $\in$ follow (Y)
  $\rightarrow$ follow (X) $\in$ follow (Y), if γ_2 is nullable.

Building Predictive Parser

$$Z \rightarrow X Y Z \qquad Y \rightarrow \text{c} \qquad X \rightarrow \text{a}$$
$$Z \rightarrow \text{d} \qquad Y \rightarrow \epsilon \qquad X \rightarrow \text{b}\, Y\, \text{e}$$

Initial:

	nullable	first	follow
Z	no		
Y	no		
X	no		

First step: Find nullable symbols.

Building Predictive Parser

$$Z \rightarrow XYZ \qquad Y \rightarrow \text{c} \qquad X \rightarrow \text{a}$$
$$Z \rightarrow \text{d} \qquad Y \rightarrow \epsilon \qquad X \rightarrow \text{b}\,Y\,\text{e}$$

Results of finding nullable symbols:

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Next: do first iteration of first sets.

Building Predictive Parser

$$Z \rightarrow XYZ \qquad Y \rightarrow \text{c} \qquad X \rightarrow \text{a}$$
$$Z \rightarrow \text{d} \qquad Y \rightarrow \epsilon \qquad X \rightarrow \text{b}\,Y\,\text{e}$$

Results of first iteration of first and follow:

	nullable	first	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

Do second iteration of first.

Building Predictive Parser

$$Z \to XYZ \qquad Y \to \text{c} \qquad X \to \text{a}$$
$$Z \to \text{d} \qquad Y \to \epsilon \qquad X \to \text{b}\,Y\,\text{e}$$

Results of second iteration of first:

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	
X	no	a,b	

Do first iteration of follow next.

Building Predictive Parser

$$Z \to X Y Z \qquad Y \to \mathrm{c} \qquad X \to \mathrm{a}$$
$$Z \to \mathrm{d} \qquad Y \to \epsilon \qquad X \to \mathrm{b}\, Y\, \mathrm{e}$$

Results:

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

Predictive Parsing Table

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

Build *predictive parsing table* from nullable, first, and follow sets.

	a	b	c	d	e
Z	$Z \to XYZ$	$Z \to XYZ$		$Z \to d$	
Y	$Y \to \epsilon$	$Y \to \epsilon$	$Y \to c$	$Y \to \epsilon$	$Y \to \epsilon$
X	$X \to a$	$X \to bYe$			

- Enter $S \to \gamma$ in row S, column T: for each $T \in$ first(γ).
- If γ is nullable, enter $S \to \gamma$ in row S, column T: for each $T \in$ follow(S).
- Entry in row S, column T tells parser which clause to execute if current function is S() and next token-type is T
- Blank entries are syntax errors.

Predictive Parsing Table

If the predictive parsing table contains *no* duplicate entries, can build predictive parser for grammar.

- Grammar is LL(1) (left-to-right parse, left-most derivation, 1 symbol lookahead).
- Grammar is LL(k) if its LL(k) predictive parsing table has no duplicate entries.
  - Rows correspond to non-terminals, columns correspond to every possible sequence of k terminals.
  - The first(γ) = set of all k-length terminal sequences that can begin any string derived from γ.
  - LL(k) paring tables can be too large.
  - Ambiguous grammars are not LL(k), for any k.

Example

$S' \to S\$$ | $S \to$ IF E THEN A ELSE A | $T \to$ NUM

$S \to E$ | $E \to E + T$ | $A \to$ ID = NUM

$S \to$ IF E THEN A | $E \to T$

Iteration 1:

	nullable	first	follow
S'	no		
S	no	IF	$
E	no		$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Iteration 2:

	nullable	first	follow
S'	no	IF	
S	no	IF	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Example

$S' \rightarrow S\$$ $\quad$ $S \rightarrow$ IF E THEN A ELSE A $\quad$ $T \rightarrow$ NUM

$S \rightarrow E$ $\quad$ $E \rightarrow E + T$ $\quad$ $A \rightarrow$ ID = NUM

$S \rightarrow$ IF E THEN A $\quad$ $E \rightarrow T$

Iteration 3:

	nullable	first	follow
S'	no	IF	
S	no	IF, NUM	\$
E	no	NUM	\$, THEN, +
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

Iteration 4:

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	\$
E	no	NUM	\$, THEN, +
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

No futher changes

Predictive Parsing Table

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	\$
E	no	NUM	\$, THEN, +
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

Build *predictive parsing table* from nullable, first, and follow sets.

	IF	THEN	ELSE	+	NUM	ID	=	\$
S'	$S' \to S$				$S' \to S$			
S	$\boxed{S \to \text{IF } E \text{ THEN } A}$ $\boxed{S \to \text{IF } E \text{ THEN } A \text{ ELSE } A}$				$S \to E$			
E					$\boxed{E \to E + T}$ $\boxed{E \to T}$			
T					$T \to$ NUM			
A						$A \to$ ID = NUM		

Table has duplicate entries $\Rightarrow$ grammar is not LL(1)!

Problems

1. $E \rightarrow E + T$
   $E \rightarrow T$

- first(E+T) = first(T)
- When in function `E()`, if next token is NUM, parser will get stuck.
- Grammar is *left-recursive* - we have seen this before....

Problems

2. $S \rightarrow$ IF E THEN A
$S \rightarrow$ IF E THEN A ELSE A

- Two productions begin with same symbol.
- first(IF E THEN A) = first(IF E THEN A ELSE A)
- Solution: use *left-factoring*
  $S \rightarrow$ IF E THEN A V
  $V \rightarrow \epsilon$
  $V \rightarrow$ ELSE A

Example

Show that modified grammar is LL(1).

$$S' \to S\$$$
$$S \to E$$
$$S \to \text{ IF } E \text{ THEN } A\ V$$
$$V \to \epsilon$$
$$V \to \text{ ELSE } A$$
$$E \to T E'$$
$$E' \to \epsilon$$
$$E' \to \text{ + } T E$$
$$T \to \text{ NUM}$$
$$A \to \text{ ID = NUM}$$

Example

Show that the grammar is LL(1).

$$S' \to S\$ \qquad V \to \text{ELSE } A \qquad T \to \text{NUM}$$
$$S \to E \qquad E \to T E' \qquad A \to \text{ID = NUM}$$
$$S \to \text{IF } E \text{ THEN } A\ V \qquad E' \to \epsilon$$
$$V \to \epsilon \qquad E' \to +TE$$

Iteration 4:

	nullable	first	follow
S'	no	IF,NUM	
S	no	IF,NUM	\$
V	yes	ELSE	\$
E	no	NUM	\$, THEN
E'	yes	+	\$, THEN
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

No futher changes

Example

Show that modified grammar is LL(1). Build predictive parsing table.

	nullable	first	follow
S'	no	IF,NUM	
S	no	IF,NUM	$
V	yes	ELSE	$
E	no	NUM	$, THEN
E'	yes	+	$, THEN
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

	IF	THEN	ELSE	+	NUM	ID	=	$
S'	$S' \to S$				$S' \to S$			
S	$S \to$ IF E THEN A V				$S \to E$			
V			$V \to$ ELSE A					$V \to \epsilon$
E					$E \to TE'$			
E'		$E' \to \epsilon$		$E' \to +TE'$				$E' \to \epsilon$
T					$T \to$ NUM			
A						$A \to$ ID = NUM		

Table does not have duplicate entries $\Rightarrow$ modified grammar is LL(1)!