Announcements

- Homework set 3 will be posted by Friday
- Homework set 2 solutions have been posted (sakai $\rightarrow$ Resources).
- The first programming project new deadline: Monday, October 29.
- Midterm exam, October 23, in class, 80 minutes. No lecture after the exam.

Project 1 – Polymorphic Type Reconstruction

The provided Scheme function *parse* maps a really-TINY program into an AST representation. Your type reconstructor should take as input an AST, a type environment, and a set of constraints:

```
(define TR
  (lambda (ast E C)
     ... )

(define TRec
  (lambda (m)
     ... ;; extract type expression from TR call
     (TR (parse m) init_E init_C) ... ))
```

Good luck, and start early!

Review – Formal Languages

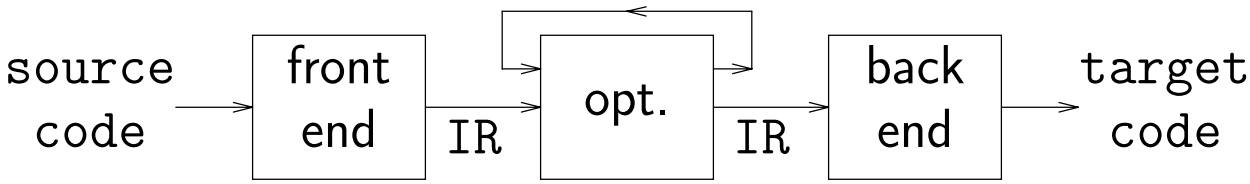
Alphabet: finite set of symbols – $\boxed{a, b, c \in \Sigma}$
String (or word): finite sequence of symbols from an alphabet – $\boxed{w, x, y \in \Sigma^*}$;
$\boxed{\epsilon \in \Sigma^*}$ denotes the empty word
Language: set of strings over an alphabet – $\boxed{L \subseteq \Sigma^*}$

Two problems:

- How to **specify** a formal language L?
  Specifications have to be finite.
- How to **recognize** that string $w \in L$?
  Procedure should be efficient (terminate?).

specification	**recognizing automaton**	**complexity**
Regular expressions	DFA	$\mathbf{O}(n)$
CFG	PDA	$\mathbf{O}(n)$ / $\mathbf{O}(n^3)$
CSG	LBA	P-SPACE COMPLETE
Arbitrary rewrite systems	TM	r.e.

Compilers



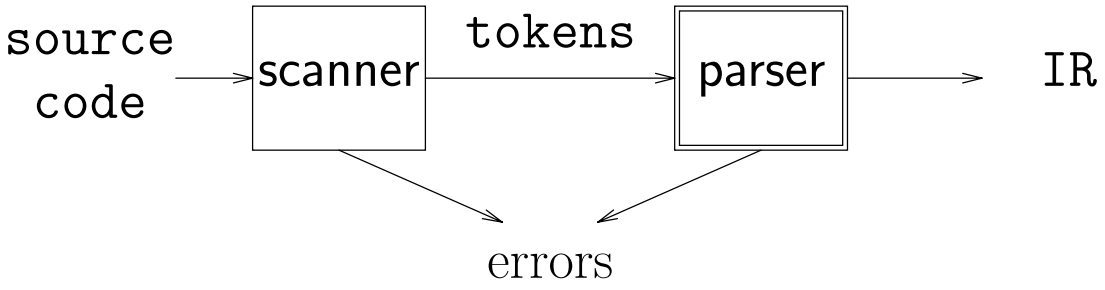
front end produce an intermediate representation (*IR*) for the program.

optimizer transforms the code in IR form into an equivalent program that may run more efficiently.

back end transforms the code in IR form into native code for the target machine

The IR encodes knowledge that the compiler has derived about the source program.

The role of the parser



Parser

- call scanner to get new token
- perform context-free syntax analysis
- guide the context-sensitive analysis
  (→ SDT / attribute grammars)
- construct an intermediate representation
  (→ SDT / attribute grammars)
- produce meaningful error messages
- attempt error correction

Syntax analysis

Context-free syntax is specified with a *grammar*.

Formally, a context-free grammar G is a four-tuple (T, NT, S, P)

T is the set of terminal symbols in the grammar. For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.

NT is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

S is a distinguished nonterminal ($S \in NT$) that denotes the entire set of strings in $L(G)$. This is sometimes called a *goal symbol* or *start symbol*. In an ECFG (extended/augmented CFG), S cannot appear on the right hand side of some production.

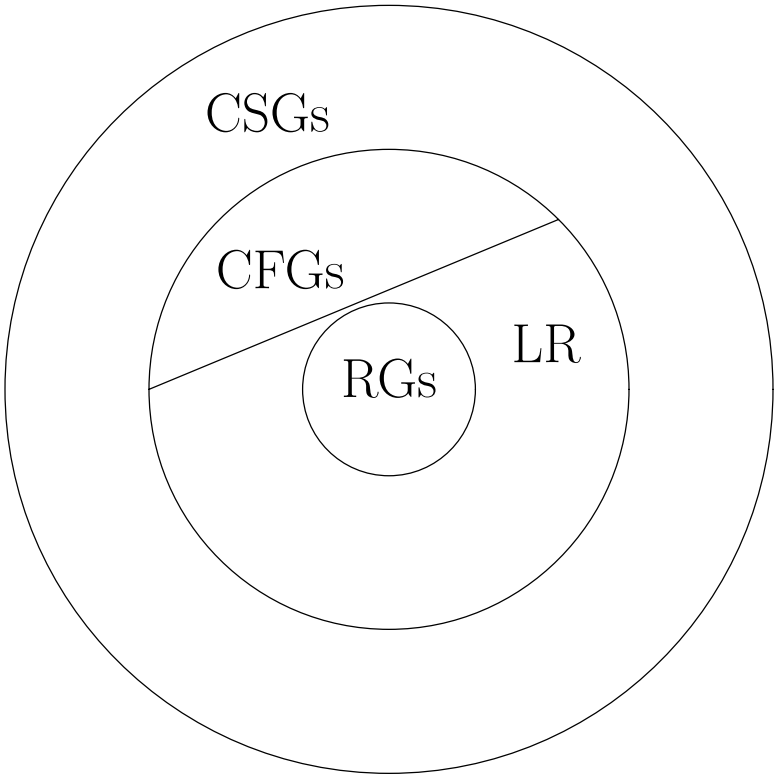
P is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

Why use context-free grammars?

Many advantages:

- precise syntactic specification of a programming language
- easy to understand, avoids ad hoc definition
- easier to maintain, add new language features
- can automatically construct efficient parser
- parser construction reveals ambiguity, other difficulties
- supports syntax-directed translation (SDT)

Complexity of parsing



Complexity:

Regular grammars	*dfa*s	$\mathbf{O}(n)$
LR grammars	Knuth's algorithm	$\mathbf{O}(n)$
Arbitrary CFGs	Early's algorithm	$\mathbf{O}(n^3)$
Arbitrary CSGs	*lba*s	*P-SPACE COMPLETE*

Review of some definitions

Parse tree (derivation tree):

"Graphical" representation for a derivation $S \Rightarrow^* w$; filters out the choice regarding non-terminal replacement order

Leftmost (rightmost) derivation

The leftmost (rightmost) non–terminal is replaced at each step

Sentential form:

Any $\alpha \in (T \cup NT)^*$ such that $S \Rightarrow^* \alpha$

Left (Right) sentential form:

Any $\alpha \in (T \cup NT)^*$ such that $S \Rightarrow^*_{lm} \alpha$ ($S \Rightarrow^*_{rm} \alpha$)

Ambiguity

If a grammar has multiple leftmost derivations for a single sentential form, the grammar is *ambiguous*.

Similarly, a grammar with multiple rightmost derivations for a single sentential form is *ambiguous*.

Example

```
<stmt> ::= if <expr> then <stmt>
         |  if <expr> then <stmt> else <stmt>
         |  .  .  .  /* other stmts */
```

Consider deriving the sentential form:

`if` E_1 `then if` E_2 `then` S_1 `else` S_2

It has two derivations.

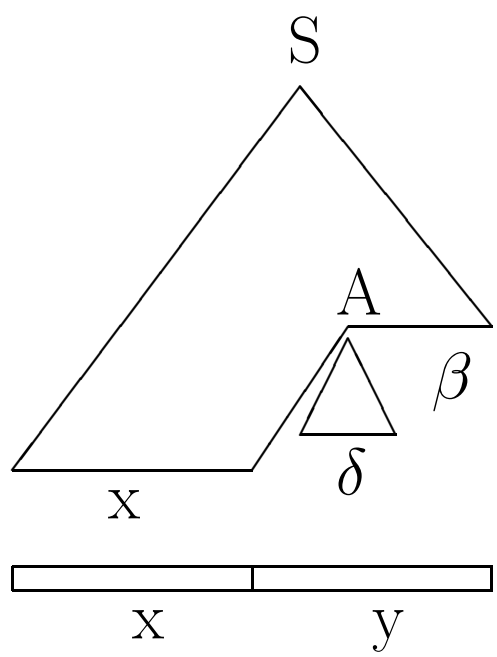
This ambiguity is purely grammatical. However, there are context-free *languages* for which every context-free *grammar* is ambiguous.
Example (see Hopcroft&Ullman, p.99):

$\{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- some grammars are backtrack-free (*predictive*)
- *LL Parsing:* reads input from *left to right* and constructs *leftmost* derivation (forwards);
  LL Parsing is predictive.

$$S \Rightarrow_{lm}^{*} xA\beta \Rightarrow_{lm} x\delta\beta \Rightarrow_{lm}^{*} xy$$



- $x, y \in T^*; S, A \in NT; \delta, \beta \in (T \cup NT)^*; A \rightarrow \delta \in P$

Predictive Parsing — LL(1)

Basic idea:

> For any two productions $A \ ::= \ \alpha \ | \ \beta$, we would like a distinct way of choosing the correct production to expand.

For some *rhs* $\alpha \ \in \ G$, define **FIRST**(α) as the set of **terminals** that appear as the first symbol in some string derived from α.

That is
$a \ \in \text{FIRST}(\alpha)$ *iff* $\alpha \ \Rightarrow^* \ a\gamma$ for some γ and $a \in T$,
$\epsilon \ \in \text{FIRST}(\alpha)$ *iff* $\alpha \ \Rightarrow^* \ \epsilon$.

For a non-terminal A, define **FOLLOW**(A) as the set of **terminals** that can appear immediately to the right of A in some sentential form.

$a \ \in \text{FOLLOW}(A)$ *iff* $S \ \Rightarrow^* \alpha A a \gamma$ for some
$\alpha, \gamma \ \in (T \cup NT)^*$, $a \in T$

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it. A terminal symbol has no FOLLOW set

Predictive Parsing — LL(1) (cont.)

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \epsilon$ then <u>in addition</u> to above condition: $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first condition, at most one of α or β can derive ϵ.

This would allow the parser to make a correct choice with a lookahead of only one symbol!

LL(1) grammars

Features

- input parsed from left to right
- leftmost derivation (forward)
- one token lookahead

Definition

A grammar G is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint, and
2. if $\alpha_i \Rightarrow^* \epsilon$, then in addition $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$, for all $1 \leq j \leq n, i \neq j$.

What rule to select for a given non-terminal and input token can be represented in a *parse table* M.

Algorithm for $LL(1)$ parse table construction must not result in multiple entries for any $M[A, a]$ or $M[A, eof]$ (Aho, Sethi, and Ullman, Algorithm 4.4).

$\Rightarrow$ Whether a grammar is $LL(1)$ or not is decidable.

Table-driven predictive parser — LL(1)

Input: a string w and a parsing table M for G

```
push eof
push Start Symbol
token ← next_token()

X ← top-of-stack
repeat
   if X is a terminal then
      if X = token then
         pop X
         token ← next_token()
      else error()

   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk, Yk−1, ···, Y1
      else error()

   X ← top-of-stack
until X = eof

if token ≠ eof then error()
```

Aho, Sethi, and Ullman, Algorithm 4.3

FIRST set construction

For a string of grammar symbols α, define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from α
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of α

STEP 1: Build $\text{FIRST}(X)$ for all grammar symbols X:

1. `if` X `is a terminal,` $\text{FIRST}(X)$ `is` $\{X\}$

2. `if` $X ::= \epsilon$`, then` $\epsilon \in \text{FIRST}(X)$

3. <u>`iterate`</u> <u>`until`</u> `no more terminals or` ϵ
`can be added to any` $\text{FIRST}(X)$`:`

`if` $X ::= Y_1 Y_2 \cdots Y_k$ `then`
$a \in \text{FIRST}(X)$ `if` $a \in \text{FIRST}(Y_i)$
`and` $\epsilon \in \text{FIRST}(Y_j)$ `for all` $1 \leq j < i$
$\epsilon \in \text{FIRST}(X)$ `if` $\epsilon \in \text{FIRST}(Y_i)$ `for all` $1 \leq i \leq k$
<u>`end`</u> <u>`iterate`</u>

(If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$)

STEP 2: Build FIRST(α) for $\alpha = X_1 X_2 \cdots X_n$:

- $a \in$ FIRST(α) `if` $a \in$ FIRST(X_i) `and` $\epsilon \in$ FIRST(X_j) `for all` $1 \leq j < i$
- $\epsilon \in$ FIRST(α) `if` $\epsilon \in$ FIRST(X_i) `for all` $1 \leq i \leq n$

FOLLOW set construction

For a non-terminal A, define FOLLOW(A) as

> the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it

A terminal symbol has no FOLLOW set

To build FOLLOW(X) for non-terminal X:

1. `place eof in` FOLLOW(⟨`goal`⟩)

<u>`iterate`</u> <u>`until`</u> `no more terminals or` ϵ
`can be added to any` FOLLOW(X)`:`

2. `if` $A ::= \alpha B \beta$ `then`
`put` $\{\text{FIRST}(\beta) - \epsilon\}$ `in` FOLLOW(B)

3. `if` $A ::= \alpha B$ `then`
`put` FOLLOW(A) `in` FOLLOW(B)

4. `if` $A ::= \alpha B \beta$ `and` $\epsilon \in$ FIRST(β) `then`
`put` FOLLOW(A) `in` FOLLOW(B)

<u>`end`</u> <u>`iterate`</u>

Predictive, Top-down Parsing - LL(1)

Example:

S ::= a S b | ϵ

$FIRST$(aSb) = {a}

$FIRST(\epsilon) = \{\epsilon\}$

$FOLLOW$(S) = {eof, b}

---

	a	b	eof	other
S	aSb	ϵ	ϵ	error

INPUT: $\boxed{\text{a a a b b b eof}}$

A larger example

expression grammar with precedence

$$
\begin{array}{lcl}
\langle \text{goal} \rangle & ::= & \langle \text{expr} \rangle \\
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \ \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & \texttt{+} \ \langle \text{expr} \rangle \\
 & | & \texttt{-} \ \langle \text{expr} \rangle \\
 & | & \epsilon \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \ \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & \texttt{*} \ \langle \text{term} \rangle \\
 & | & \texttt{/} \ \langle \text{term} \rangle \\
 & | & \epsilon \\
\langle \text{factor} \rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

LL(1) parse table

	id	num	+	-	*	/	eof
$\langle \text{goal} \rangle$	$g \rightarrow e$	$g \rightarrow e$	–	–	–	–	–
$\langle \text{expr} \rangle$	$e \rightarrow te'$	$e \rightarrow te'$	–	–	–	–	–
$\langle \text{expr}' \rangle$	–	–	$e' \rightarrow \texttt{+}e$	$e' \rightarrow \texttt{-}e$	–	–	$e' \rightarrow \epsilon$
$\langle \text{term} \rangle$	$t \rightarrow ft'$	$t \rightarrow ft'$	–	–	–	–	–
$\langle \text{term}' \rangle$	–	–	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow \texttt{*}t$	$t' \rightarrow \texttt{/}t$	$t' \rightarrow \epsilon$
$\langle \text{factor} \rangle$	$f \rightarrow \texttt{id}$	$f \rightarrow \texttt{num}$	–	–	–	–	–

Recursive descent parsing — LL(1)

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated **parsing procedure** that can recognize any sequence of tokens generated by that non–terminal.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful "computation" (syntax directed translation).

Recursive Descent Parsing (pseudo code)

	a	b	eof	other
S	aSb	ϵ	ϵ	error

```
main:  {
  token := next_token( );
  if (S( ) and token == eof) print ``accept'' else print ``error'';
}


bool S:
  switch token {
    case a:  token := next_token( );
             if (not S( )) return false; // recursive call to S;
             if token == b {
                  token := next_token( )
                  return true;
             }
             else
                  return false;
             break;
    case b,
    case eof:return true;
             break;
    default: return false;
  }
```

Next class

More on recursive descent parsing

Bottom-up parsing: LR(0) LR(1), SLR(1).

Syntax-directed translation schemes