CS3300 - Language Translators

Introduction

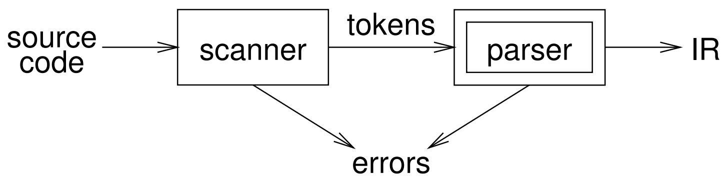
V. Krishna Nandivada

IIT Madras

Acknowledgement

These slides borrow liberal portions of text verbatim from Antony L. Hosking @ Purdue and Jens Palsberg @ UCLA.

The role of the parser



A parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

For the next several classes, we will look at parser construction

Syntax analysis by using a CFG

Context-free syntax is specified with a context-free grammar.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

- V_t is the set of terminal symbols in the grammar. For our purposes, V_t is the set of tokens returned by the scanner.
- V_n, the nonterminals, 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 $(S \in V_n)$ denoting the entire set of strings in $L(G)$. This is sometimes called a goal symbol.
- P is a finite set of productions specifying how 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.

The set $V = V_t \cup V_n$ is called the vocabulary of G

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V*$
- $u, v, w, \ldots \in V_t*$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a single-step derivation using $A \rightarrow \gamma$

Similarly, $\rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps

If $S \rightarrow^* \beta$ then β is said to be a sentential form of G

$L(G) = \{w \in V_t* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a sentence of G

Note, $L(G) = \{\beta \in V* \mid S \rightarrow^* \beta\} \cap V_t*$

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).
Example:

1	$\langle$goal$\rangle$	::=	$\langle$expr$\rangle$
2	$\langle$expr$\rangle$	::=	$\langle$expr$\rangle\langle$op$\rangle\langle$expr$\rangle$
3		|	`num`
4		|	`id`
5	$\langle$op$\rangle$	::=	$+$
6		|	$-$
7		|	$*$
8		|	$/$

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or underline
3. productions as in the example

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG:

$$\begin{aligned}
\langle\text{goal}\rangle &\Rightarrow \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{aligned}$$

We have derived the sentence `x` $+$ `2` $*$ `y`.
We denote this $\langle$goal$\rangle \rightarrow^*$ `id` $+$ `num` $*$ `id`.
pause
Such a sequence of rewrites is a derivation or a parse.
The process of discovering a derivation is called parsing.

Derivations

At each step, we chose a non-terminal to replace.
This choice can lead to different derivations.
Two are of particular interest:

leftmost derivation
the leftmost non-terminal is replaced at each step

rightmost derivation
the rightmost non-terminal is replaced at each step

The previous example was a leftmost derivation.

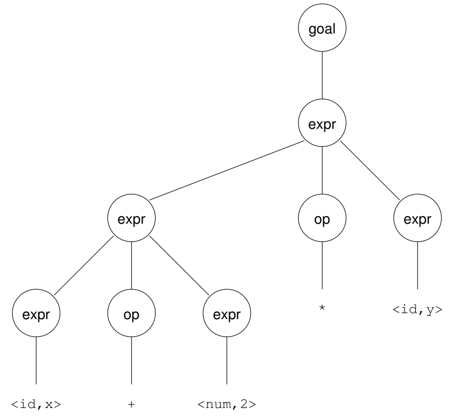
Rightmost derivation

For the string $\mathtt{x} + 2 * \mathtt{y}$:

$$\begin{array}{lcl} \langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \end{array}$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.

Precedence



Treewalk evaluation computes $(\mathtt{x} + 2) * \mathtt{y}$
— the "wrong" answer!
Should be $\mathtt{x} + (2 * \mathtt{y})$

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 takes additional machinery:

$$\begin{array}{r|lcl} 1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\ 2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\ 3 & & \mid & \langle\text{expr}\rangle - \langle\text{term}\rangle \\ 4 & & \mid & \langle\text{term}\rangle \\ 5 & \langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\ 6 & & \mid & \langle\text{term}\rangle / \langle\text{factor}\rangle \\ 7 & & \mid & \langle\text{factor}\rangle \\ 8 & \langle\text{factor}\rangle & ::= & \mathtt{num} \\ 9 & & \mid & \mathtt{id} \end{array}$$

This grammar enforces a precedence on the derivation:

- terms must be derived from expressions
- forces the "correct" tree

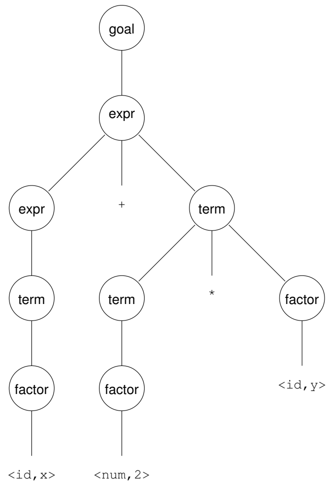
Precedence

Now, for the string $\mathtt{x} + 2 * \mathtt{y}$:

$$\begin{array}{lcl} \langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{factor}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{term}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{factor}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\ & \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \end{array}$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$, but this time, we build the desired tree.

Precedence



Treewalk evaluation computes $\mathtt{x} + (2 * \mathtt{y})$

Ambiguity

If a grammar has more than one derivation for a single sentential form, then it 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.
It is a context-free ambiguity.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

⟨stmt⟩	::=	⟨matched⟩
	|	⟨unmatched⟩
⟨matched⟩	::=	`if` ⟨expr⟩ `then` ⟨matched⟩ `else` ⟨matched⟩
	|	`other stmts`
⟨unmatched⟩	::=	`if` ⟨expr⟩ `then` ⟨stmt⟩
	|	`if` ⟨expr⟩ `then` ⟨matched⟩ `else` ⟨unmatched⟩

This generates the same language as the ambiguous grammar, but applies the common sense rule:

match each `else` with the closest unmatched `then`

This is most likely the language designer's intent.

Ambiguity

Ambiguity is often due to confusion in the context-free specification.
Context-sensitive confusions can arise from overloading.
Example:

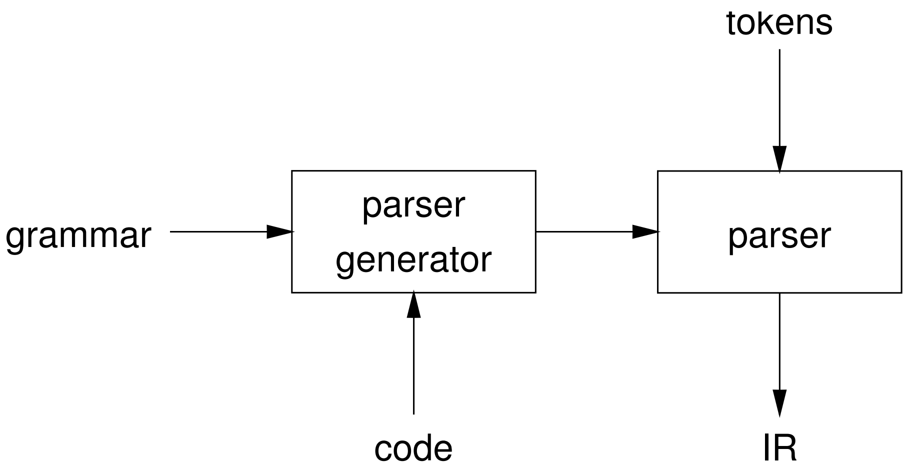
a = f(17)

In many Algol-like languages, `f` could be a function or subscripted variable. Disambiguating this statement requires context:

- need values of declarations
- not context-free
- really an issue of type

Rather than complicate parsing, we will handle this separately.

Parsing: the big picture



Our goal is a flexible parser generator system

Scanning vs. parsing

Where do we draw the line?

$$\begin{array}{lll} \textit{term} & ::= & [\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}]([\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}] \mid [0-9])^* \\ & \mid & 0 \mid [1-9][0-9]^* \\ \textit{op} & ::= & + \mid - \mid * \mid / \\ \textit{expr} & ::= & (\textit{term op})^* \textit{term} \end{array}$$

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

Context-free grammars are used to count:

- brackets: `()`, `begin...end`, `if...then...else`
- imparting structure: expressions

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes compiler more manageable.

Closing remarks

What did we do this week?

- Overview of the compilation process.
- Quick look at Lexical analysis.
- Introduction to Parsing.

Reading:

- Ch 1 and 3 from the Dragon book.
- Recap from previous year : regular expressions and context free grammars.

Announcement:

- Next class: Wednesday 11AM.
- Lab assignment – out! Due 17th Aug 2012.