CS6013 - Modern Compilers: Theory and Practise

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Written assignment = 5 marks.
- Programming assignments = 50 marks.
- Midterm = 20 marks, Final = 25 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms. Non compliance will lead to 'W' grade.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Fail grade guaranteed.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@cse.iitm.ac.in, Office: BSB 352.
TA : Aman Nougrahiya, Email: amannoug@cse

What, When and Why of Compilers

- **What**:
  - A compiler is a program that can read a program in one language and translates it into an equivalent program in another language.
- **When**
  - 1952, by Grace Hopper for A-0.
  - 1957, Fortran compiler by John Backus and team.
- **Why? Study?**
  - It is good to know how the food you eat, is cooked.
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - For a computer to execute programs written in these languages, these programs need to be translated to a form in which it can be executed by the computer.

Compilers – A "Sangam"

Compiler construction is a microcosm of computer science

- **Artificial Intelligence** greedy algorithms, learning algorithms, . . .
- **Algo** graph algorithms, union-find, dynamic programming, . . .
- **theory** DFAs for scanning, parser generators, lattice theory, . . .
- **systems** allocation, locality, layout, synchronization, . . .
- **architecture** pipeline management, hierarchy management, instruction set use, . . .
- **optimizations** Operational research, load balancing, scheduling, . . .

Inside a compiler, all these and many more come together. Has probably the healthiest mix of theory and practise.

Course outline

A rough outline (we may not strictly stick to this).

- Overview of Compilers
- Overview of lexical analysis and parsing.
- Semantic analysis (aka type checking)
- Intermediate code generation
- Data flow analysis
- Constant propagation
- Static Single Assignment and Optimizations.
- Loop optimizations
- Liveness analysis
- Register Allocation
- Bitwidth aware register allocation
- Code Generation
- Overview of advanced topics.

Your friends: Languages and Tools

Start exploring

- Java - familiarity a must - Use eclipse to save you valuable coding and debugging cycles.
- JavaCC, JTB – tools you will learn to use.
- Make Ant Scripts – recommended toolkit.
- Find the course webpage: http://www.cse.iitm.ac.in/∼krishna/cs6013/

Get set. Ready steady go!

Acknowledgement

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

Compilers – A closed area?

"Optimization for scalar machines was solved years ago"

Machines have changed drastically in the last 20 years

Changes in architecture ⇒ changes in compilers

- new features pose new problems
- changing costs lead to different concerns
- old solutions need re-engineering

Changes in compilers should prompt changes in architecture

- New languages and features

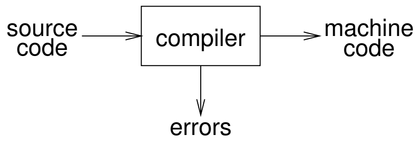
Expectations

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your expectations about this course

Abstract view

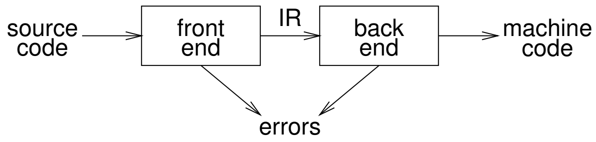


Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

Big step up from assembler — higher level notations

Traditional two pass compiler

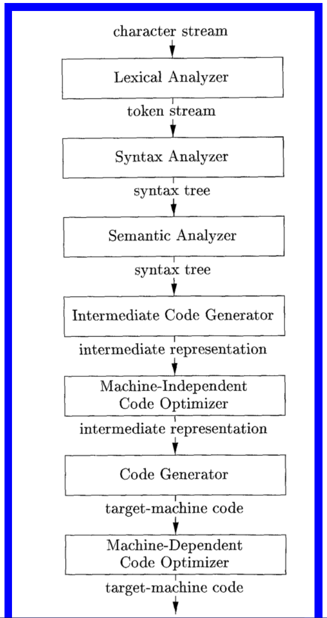


Implications:

- intermediate representation (IR).
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ better code

A rough statement: Most of the problems in the Front-end are simpler (polynomial time solution exists).
Most of the problems in the Back-end are harder (many problems are NP-complete in nature).
Our focus: Mainly back end (95%) and little bit of front end (5%).

Phases inside the compiler



Front end responsibilities:

- Recognize syntactically legal code; report errors.
- Recognize semantically legal code; report errors.
- Produce IR.

Back end responsibilities:

- Optimizations, code generation.

Our target

- five out of seven phases.
- glance over lexical and syntax analysis – read yourself or attend the undergraduate course, if interested.

Lexical analysis

- Also known as scanning.
- Reads a stream of characters and groups them into meaningful sequences, called lexems.

A scanner must recognize the units of syntax

Q: How to specify patterns for the scanner?

Examples:

- white space

  <ws> ::= <ws> ' '
  | <ws> '\t'
  | ' '
  | '\t'

- keywords and operators
  specified as literal patterns: `do`, `end`

More complex syntax

- identifiers
  alphabet followed by k alphanumerics (_, \$, &, ...)
- numbers
  - integers: 0 or digit from 1-9 followed by digits from 0-9
  - decimals: integer '.' digits from 0-9
  - reals: (integer or decimal) 'E' (+ or -) digits from 0-9
  - complex: '(' real ',' real ')'

We need a powerful notation to specify these patterns - regular expressions

Examples of Regular Expressions

- identifier
  letter $\rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$
  digit $\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$
  id $\rightarrow$ letter (letter | digit)*
- numbers
  integer $\rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9)$ digit$^*)$
  decimal $\rightarrow$ integer . (digit)*
  real $\rightarrow$ (integer | decimal) `E` $(+ \mid -)$ digit*
  complex $\rightarrow$ `'('` real `,` real `')'`

Most tokens can be described with REs
We can use REs to build scanners automatically

Generic examples of REs

Let $\Sigma = \{a, b\}$

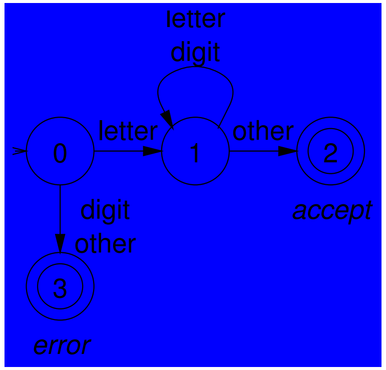
- $a|b$ denotes $\{a, b\}$
- $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
  i.e., $(a|b)(a|b) = aa|ab|ba|bb$
- $a*$ denotes $\{\varepsilon, a, aa, aaa, \ldots\}$
- $(a|b)*$ denotes the set of all strings of a's and b's (including ε)
  i.e., $(a|b)* = (a*b*)*$
- $a|a*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Recognizers

From a regular expression we can construct a

deterministic finite automaton (DFA)

Recognizer for identifier:



Grammars for regular languages

Can we place a restriction on the form of a grammar to ensure that it describes a regular language?
Provable fact:

For any RE r, $\exists$ a grammar g such that $L(r) = L(g)$

Grammars that generate regular sets are called regular grammars:

They have productions in one of 2 forms:

1. $A \to aA$
2. $A \to a$

where A is any non-terminal and a is any terminal symbol

These are also called type 3 grammars (Chomsky)

Finite Automata

A non-deterministic finite automaton (NFA) consists of:

1. a set of states $S = \{s_0, \ldots, s_n\}$
2. a set of input symbols Σ (the alphabet)
3. a transition function mapping state-symbol pairs to sets of states
4. a distinguished start state s_0
5. a set of distinguished accepting or final states F

A Deterministic Finite Automaton (DFA) is a special case:

1. no state has a ε-transition, and
2. for each state s and input symbol a, $\exists$ at most one edge labelled a leaving s

A DFA accepts x iff. $\exists$ a unique path through the transition graph from s_0 to a final state such that the edges spell x.

DFAs and NFAs are equivalent

1. DFAs are clearly a subset of NFAs
2. Any NFA can be converted into a DFA, by simulating sets of simultaneous states:
   - each DFA state corresponds to a set of NFA states
   - possible exponential blowup

Limits of regular languages

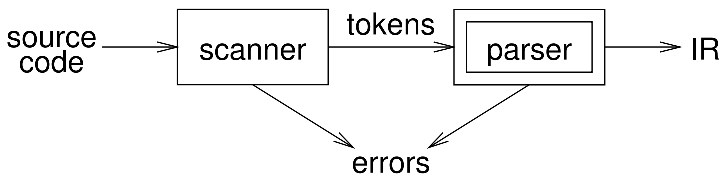
Not all languages are regular
One cannot construct DFAs to recognize these languages:

- $L = \{p(k)q(k)\}$
- $L = \{wcw(rev_{|}w \in \Sigma*\}$

Note: neither of these is a regular expression!
(DFAs cannot count!)
But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
  $(\varepsilon \mid 1)(01)*(\varepsilon \mid 0)$
- sets of pairs of 0's and 1's
  $(01 \mid 10)+$

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 couple of lecture hours, 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 $A \to \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a single-step derivation using $A \to \gamma$

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

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

Derivations

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

1	$\langle goal \rangle$	$::=$	$\langle expr \rangle$
2	$\langle expr \rangle$	$::=$	$\langle expr \rangle + \langle term \rangle$
3		$\mid$	$\langle expr \rangle - \langle term \rangle$
4		$\mid$	$\langle term \rangle$
5	$\langle term \rangle$	$::=$	$\langle term \rangle * \langle factor \rangle$
6		$\mid$	$\langle term \rangle / \langle factor \rangle$
7		$\mid$	$\langle factor \rangle$
8	$\langle factor \rangle$	$::=$	$\mathtt{num}$
9		$\mid$	$\mathtt{id}$

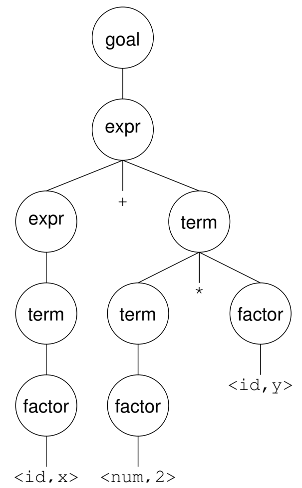
Deriving the derivation

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

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

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

Parse tree



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

Different ways of parsing: Top-down Vs Bottom-up

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (recognize valid prefixes)
- use a stack to store both state and sentential forms

Top-down parsing

A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar.
To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1.

If the parser makes a wrong step, the "derivation" process does not terminate.
Why is it bad?

Left-recursion

Top-down parsers cannot handle left-recursion in a grammar
Formally, a grammar is left-recursive if

$\exists A \in V_n$ *such that* $A \Rightarrow^+ A\alpha$ *for some string* α

Our simple expression grammar is left-recursive

Eliminating left-recursion

To remove left-recursion, we can transform the grammar
Consider the grammar fragment:

$$\begin{array}{lcl} \langle \text{foo} \rangle & ::= & \langle \text{foo} \rangle \alpha \\ & | & \beta \end{array}$$

where α and β do not start with $\langle \text{foo} \rangle$
We can rewrite this as:

$$\begin{array}{lcl} \langle \text{foo} \rangle & ::= & \beta \langle \text{bar} \rangle \\ \langle \text{bar} \rangle & ::= & \alpha \langle \text{bar} \rangle \\ & | & \varepsilon \end{array}$$

where $\langle \text{bar} \rangle$ is a new non-terminal

This fragment contains no left-recursion

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are:

LL(1): **l**eft to right scan, **l**eft-most derivation, **1**-token lookahead; and

LR(1): **l**eft to right scan, reversed **r**ight-most derivation, **1**-token lookahead

Predictive parsing

Basic idea:

- For any two productions $A \to \alpha \mid \beta$, 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 tokens that appear first in some string derived from α.
- That is, for some $w \in V_t^*$, $w \in$ FIRST(α) iff. $\alpha \Rightarrow^* w\gamma$.
- Key property:
  Whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like
  - FIRST$(\alpha) \cap$ FIRST$(\beta) = \phi$
- This would allow the parser to make a correct choice with a lookahead of only one symbol!

Left factoring

What if a grammar does not have this property?

Sometimes, we can transform a grammar to have this property.

For each non-terminal A find the longest prefix α common to two or more of its alternatives.

if $\alpha \neq \varepsilon$ then replace all of the A productions
$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$
with

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where A' is a new non-terminal.

Repeat until no two alternatives for a single non-terminal have a common prefix.

Example

There are two non-terminals to left factor:

$$\begin{array}{lcl} \langle\text{expr}\rangle & ::= & \langle\text{term}\rangle + \langle\text{expr}\rangle \\ & \mid & \langle\text{term}\rangle - \langle\text{expr}\rangle \\ & \mid & \langle\text{term}\rangle \\ \langle\text{term}\rangle & ::= & \langle\text{factor}\rangle * \langle\text{term}\rangle \\ & \mid & \langle\text{factor}\rangle / \langle\text{term}\rangle \\ & \mid & \langle\text{factor}\rangle \end{array}$$

Applying the transformation:

$$\begin{array}{lcl} \langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\langle\text{expr}'\rangle \\ \langle\text{expr}'\rangle & ::= & +\langle\text{expr}\rangle \\ & \mid & -\langle\text{expr}\rangle \\ & \mid & \varepsilon \\ \langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\langle\text{term}'\rangle \\ \langle\text{term}'\rangle & ::= & *\langle\text{term}\rangle \\ & \mid & /\langle\text{term}\rangle \\ & \mid & \varepsilon \end{array}$$

Indirect Left-recursion elimination

Given a left-factored CFG, to eliminate left-recursion:

if $\exists\, A \to A\alpha$ then replace all of the A productions

$A \to A\alpha \mid \beta \mid \ldots \mid \gamma$

with

$A \to NA'$

$N \to \beta \mid \ldots \mid \gamma$

$A' \to \alpha A' \mid \varepsilon$

where N and A' are new productions.

Repeat until there are no left-recursive productions.

Generality

Question:

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

Given a context-free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many context-free languages do not have such a grammar:

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

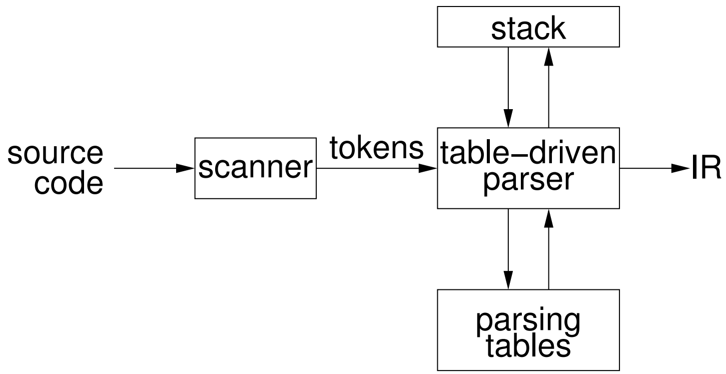
Must look past an arbitrary number of a's to discover the 0 or the 1 and so determine the derivation.

Self reading

Recursive decent parsing.

Non-recursive predictive parsing

Now, a predictive parser looks like:

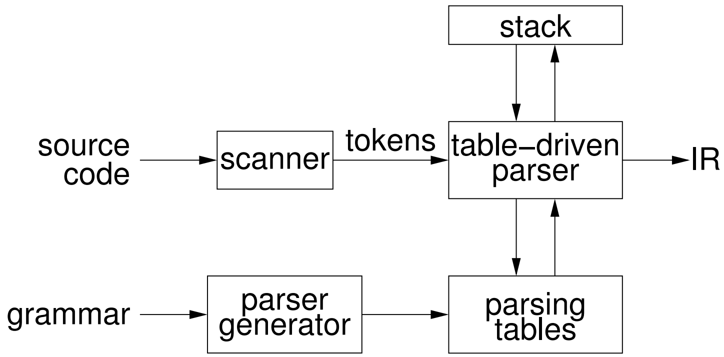


Rather than writing recursive code, we build tables.

Why?Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



- This is true for both top-down (LL) and bottom-up (LR) parsers
- This also uses a stack – but mainly to remember part of the input string; no recursion.

FIRST

For a string of grammar symbols α, define FIRST(α) as:

- the set of terminals that begin strings derived from α: $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in$ FIRST(α)

FIRST(α) contains the tokens valid in the initial position in α

To build FIRST(X):

1. If $X \in V_t$ then FIRST(X) is $\{X\}$
2. If $X \to \varepsilon$ then add ε to FIRST(X)
3. If $X \to Y_1 Y_2 \cdots Y_k$:
    1. Put FIRST$(Y_1) - \{\varepsilon\}$ in FIRST(X)
    2. $\forall i : 1 < i \leq k$, if $\varepsilon \in$ FIRST$(Y_1) \cap \cdots \cap$ FIRST(Y_{i-1}) (i.e., $Y_1 \cdots Y_{i-1} \Rightarrow^* \varepsilon$) then put FIRST$(Y_i) - \{\varepsilon\}$ in FIRST(X)
    3. If $\varepsilon \in$ FIRST$(Y_1) \cap \cdots \cap$ FIRST(Y_k) then put ε in FIRST(X)

   Repeat until no more additions can be made.

FOLLOW

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(A):

1. Put \$ in FOLLOW$(\langle \text{goal} \rangle)$
2. If $A \to \alpha B \beta$:
    1. Put FIRST$(\beta) - \{\varepsilon\}$ in FOLLOW(B)
    2. If $\beta = \varepsilon$ (i.e., $A \to \alpha B$) or $\varepsilon \in$ FIRST(β) (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW$(A)$ in FOLLOW(B)

   Repeat until no more additions can be made

LL(1) grammars

Previous definition

> *A grammar G is LL(1) iff. for all non-terminals A, each distinct pair of productions $A \to \beta$ and $A \to \gamma$ satisfy the condition* FIRST$(\beta) \bigcap$ FIRST$(\gamma) = \phi$.

What if $A \Rightarrow^* \varepsilon$?

Revised definition

> *A grammar G is LL(1) iff. for each set of productions $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$:*
>
> 1. FIRST(α_1), FIRST$(\alpha_2), \ldots,$ FIRST(α_n) *are all pairwise disjoint*
> 2. *If $\alpha_i \Rightarrow^* \varepsilon$ then* FIRST$(\alpha_j) \bigcap$ FOLLOW$(A) = \phi, \forall 1 \leq j \leq n, i \neq j$.

If G is ε-free, condition 1 is sufficient.

LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example

- $S \to aS \mid a$ is not LL(1) because FIRST$(aS) =$ FIRST$(a) = \{a\}$
- $S \to aS'$
  $S' \to aS' \mid \varepsilon$
  accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G
Output: Parsing table M
Method:

1. $\forall$ productions $A \to \alpha$:
   1. $\forall a \in$ FIRST(α), add $A \to \alpha$ to $M[A,a]$
   2. If $\varepsilon \in$ FIRST(α):
      1. $\forall b \in$ FOLLOW(A), add $A \to \alpha$ to $M[A,b]$
      2. If $\$ \in$ FOLLOW(A) then add $A \to \alpha$ to $M[A,\$]$
2. Set each undefined entry of M to `error`

If $\exists M[A,a]$ with multiple entries then grammar is not LL(1).

Note: recall $a,b \in V_t$, so $a,b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$$\begin{array}{l|l|l} S \to E_1 & E' \to +E_3 \mid -E_4 \mid \varepsilon_5 & T' \to *T_7 \mid /T_8 \mid \varepsilon_9 \\ E \to TE'_2 & T \to FT'_6 & F \to \texttt{num}_{10} \mid \texttt{id}_{11} \end{array}$$

	FIRST	FOLLOW	id	num	+	−	*	/	$
S	num, id	$	1	1	–	–	–	–	–
E	num, id	$	2	2	–	–	–	–	–
E'	ε, +, −	$	–	–	3	4	–	–	5
T	num, id	+, −, $	6	6	–	–	–	–	–
T'	ε, *, /	+, −, $	–	–	9	9	7	8	9
F	num, id	+, −, *, /, $	11	10	–	–	–	–	–
id	id	–							
num	num	–							
*	*	–							
/	/	–							
+	+	–							
−	−	–							

A grammar that is not LL(1)

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle \\ & \mid & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \texttt{else}\ \langle \text{stmt} \rangle \\ & \mid & \ldots \end{array}$$

Left-factored: $\langle \text{stmt} \rangle ::= \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \langle \text{stmt}' \rangle \mid \ldots$ Now,
$\langle \text{stmt}' \rangle ::= \texttt{else}\ \langle \text{stmt} \rangle \mid \varepsilon$

FIRST$(\langle \text{stmt}' \rangle) = \{\varepsilon, \texttt{else}\}$
Also, FOLLOW$(\langle \text{stmt}' \rangle) = \{\texttt{else}, \$\}$
But, FIRST$(\langle \text{stmt}' \rangle) \bigcap$ FOLLOW$(\langle \text{stmt}' \rangle) = \{\texttt{else}\} \neq \phi$
On seeing `else`, there is a conflict between choosing

$\langle stmt' \rangle ::= \texttt{else}\ \langle stmt \rangle$ *and* $\langle stmt' \rangle ::= \varepsilon$

$\Rightarrow$ grammar is not LL(1)!
The fix:

Put priority on $\langle stmt' \rangle ::= \texttt{else}\ \langle stmt \rangle$ *to associate* `else` *with closest previous* `then`.

Another example of painful left-factoring

- Here is a typical example where a programming language fails to be LL(1):
  `stmt → asginment | call | other`
  `assignment → id := exp`
  `call → id (exp-list)`
- This grammar is not in a form that can be left factored. We must first replace assignment and call by the right-hand sides of their defining productions:
  `stmt → id := exp | id ( exp-list ) | other`
- We left factor:
  `statement → id stmt' | other`
  `stmt' → := exp | (exp-list)`
- See how the grammar obscures the language semantics.

Revision 1/4

$$
\begin{aligned}
&S \to E_1\\
&E \to TE'_2\\
&E' \to +E_3 \mid -E_4 \mid \varepsilon_5\\
&T \to FT'_6\\
&T' \to *T_7 \mid /T_8 \mid \varepsilon_9\\
&F \to \mathtt{num}_{10} \mid \mathtt{id}_{11}
\end{aligned}
$$

- Compute the FIRST and the FOLLOW sets.

Revision 2/4: FIRST and FOLLOW sets

	FIRST	FOLLOW
S	$\{\mathtt{num},\mathtt{id}\}$	$\{\$\}$
E	$\{\mathtt{num},\mathtt{id}\}$	$\{\$\}$
E'	$\{\varepsilon,+,-\}$	$\{\$\}$
T	$\{\mathtt{num},\mathtt{id}\}$	$\{+,-,\$\}$
T'	$\{\varepsilon,*,/\}$	$\{+,-,\$\}$
F	$\{\mathtt{num},\mathtt{id}\}$	$\{+,-,*,/,\$\}$
id	$\{\mathtt{id}\}$	–
num	$\{\mathtt{num}\}$	–
$*$	$\{*\}$	–
$/$	$\{/\}$	–
$+$	$\{+\}$	–
$-$	$\{-\}$	–

- Build the parse table.

Revision 3/4: Parse Table

	id	num	$+$	$-$	$*$	$/$	$\$$
S	$S \to E$	$S \to E$	–	–	–	–	–
E	$E \to TE'$	$E \to TE'$	–	–	–	–	–
E'	–	–	$E' \to +E$	$E' \to -E$	–	–	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	–	–	–	–	–
T'	–	–	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \mathtt{id}$	$F \to \mathtt{num}$	–	–	–	–	–

Revision 4/4: Building the parse tree

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

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← root node
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
          pop and fill in node
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y_1 Y_2 ··· Y_k then
         pop X
         pop node for X
         build node for each child and
         make it a child of node for X
         push n_k, Y_k, n_{k-1}, Y_{k-1}, ··· , n_1, Y_1
      else error()
until X = EOF
```


Revision 3/4: Parse Table

	id	num	+	−	∗	/	\$
S	$S \to E$	$S \to E$	–	–	–	–	–
E	$E \to TE'$	$E \to TE'$	–	–	–	–	–
E'	–	–	$E' \to +E$	$E' \to -E$	–	–	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	–	–	–	–	–
T'	–	–	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \texttt{id}$	$F \to \texttt{num}$	–	–	–	–	–

Next: Bottum up Parsing.

Some definitions

Recall

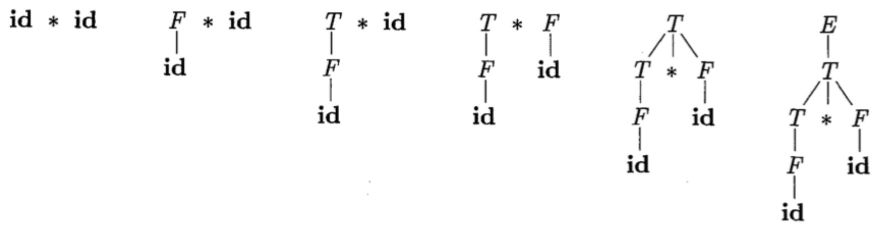
- For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a sentential form
- If $\alpha \in V_t^*$, then α is called a sentence in $L(G)$
- Otherwise it is just a sentential form (not a sentence in $L(G)$)

A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.
A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

Bottom-up parsing

Goal:

Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.



Reductions Vs Derivations

Reduction:

- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of the production.

Key decisions

- When to reduce?
- What production rule to apply?

Reduction Vs Derivations

- Recall: In derivation: a non-terminal in a sentential form is replaced by the body of one of its productions.
- A reduction is reverse of a step in derivation.

- Bottom-up parsing is the process of “reducing” a string w to the start symbol.
- Goal of bottum-up parsing: build derivation tree in reverse.

Example

Consider the grammar

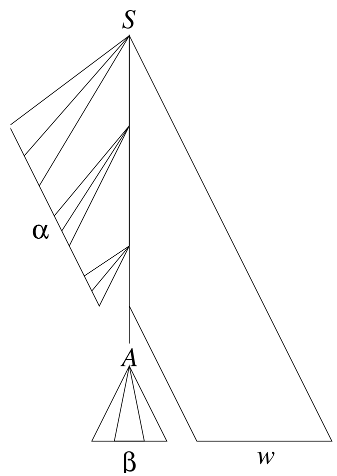
$$\begin{array}{c|lcl} 1 & S & \rightarrow & \mathtt{a}AB\mathtt{e} \\ 2 & A & \rightarrow & A\mathtt{bc} \\ 3 & & \mid & \mathtt{b} \\ 4 & B & \rightarrow & \mathtt{d} \end{array}$$

and the input string `abbcde`

Prod'n.	Sentential Form
3	$\mathtt{a}\boxed{\mathtt{b}}\mathtt{bcde}$
2	$\mathtt{a}\boxed{A\mathtt{bc}}\mathtt{de}$
4	$\mathtt{a}A\boxed{\mathtt{d}}\mathtt{e}$
1	$\boxed{\mathtt{a}AB\mathtt{e}}$
–	S

The trick appears to be scanning the input and finding valid sentential forms.

Handles



The handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

- Informally, a “handle” is a substring that matches the body of a production (not necessarily the first one).
- And reducing this handle, represents one step of reduction (or reverse rightmost derivation).

Handles

Theorem:

If G is unambiguous then every right-sentential form has a unique handle.

Proof: (by definition)

1. G is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \to \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position k at which $A \to \beta$ is applied
4. $\Rightarrow$ a unique handle $A \to \beta$

Example

The left-recursive expression grammar (original form)

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

Prod'n.	Sentential Form
–	$\langle$goal$\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle$expr$\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle$expr$\rangle - \langle$term$\rangle * \underline{\texttt{id}}$
7	$\langle$expr$\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle$expr$\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

Handle-pruning

The process to construct a bottom-up parse is called handle-pruning.
To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

we set i to n and apply the following simple algorithm

for i = n downto 1

1. find the handle $A_i \to \beta_i$ in γ_i
2. replace β_i with A_i to generate γ_{i-1}

This takes $2n$ steps, where n is the length of the derivation

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser.
Shift-reduce parsers use a stack and an input buffer

1. initialize stack with \$
2. Repeat until the top of the stack is the goal symbol and the input token is \$
   a) find the handle
   if we don't have a handle on top of the stack, shift an input symbol onto the stack
   b) prune the handle
   if we have a handle $A \to \beta$ on the stack, reduce
      i) pop $\mid \beta \mid$ symbols off the stack
      ii) push A onto the stack

Example: back to x − 2 * y

1	$S \rightarrow E$
2	$E \rightarrow E + T$
3	$\mid E - T$
4	$\mid T$
5	$T \rightarrow T * F$
6	$\mid T / F$
7	$\mid F$
8	$F \rightarrow$ num
9	$\mid$ id

Stack	Input	Action
$	id − num ∗ id	S
$id	− num ∗ id	R9
$⟨factor⟩	− num ∗ id	R7
$⟨term⟩	− num ∗ id	R4
$⟨expr⟩	− num ∗ id	S
$⟨expr⟩ −	num ∗ id	S
$⟨expr⟩ − num	∗ id	R8
$⟨expr⟩ − ⟨factor⟩	∗ id	R7
$⟨expr⟩ − ⟨term⟩	∗ id	S
$⟨expr⟩ − ⟨term⟩ ∗	id	S
$⟨expr⟩ − ⟨term⟩ ∗ id		R9
$⟨expr⟩ − ⟨term⟩ ∗ ⟨factor⟩		R5
$⟨expr⟩ − ⟨term⟩		R3
$⟨expr⟩		R1
$⟨goal⟩		A

Shift-reduce parsing

Shift-reduce parsers are simple to understand

A shift-reduce parser has just four canonical actions:

1. shift — next input symbol is shifted onto the top of the stack
2. reduce — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal LHS
3. accept — terminate parsing and signal success
4. error — call an error recovery routine

Key insight: recognize handles with a DFA:

- DFA transitions shift states instead of symbols
- accepting states trigger reductions

LR parsing

The skeleton parser:

```
push s0
token ← next_token()
repeat forever
  s ← top of stack
  if action[s,token] = "shift si" then
    push si
    token ← next_token()
  else if action[s,token] = "reduce A → β"
    then
    pop |β| states
    s′ ← top of stack
    push goto[s′,A]
  else if action[s, token] = "accept" then
    return
  else error()
```

"**How many** ops?":k shifts, l reduces, and 1 accept, where k is length of input string and l is length of reverse rightmost derivation

Example tables

state	ACTION				GOTO		
	id	+	∗	$	E	T	F
0	s4	–	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	s5	–	r3	–	–	–
3	–	r5	s6	r5	–	–	–
4	–	r6	r6	r6	–	–	–
5	s4	–	–	–	7	2	3
6	s4	–	–	–	–	8	3
7	–	–	–	r2	–	–	–
8	–	r4	–	r4	–	–	–

The Grammar

1	$S \rightarrow E$
2	$E \rightarrow T + E$
3	$\mid T$
4	$T \rightarrow F * T$
5	$\mid F$
6	$F \rightarrow$ id

Note: This is a simple little right-recursive grammar. It is not the same grammar as in previous lectures.

Example using the tables

Stack	Input	Action
$ 0	`id* id+ id$`	s4
$ 0 4	`* id+ id$`	r6
$ 0 3	`* id+ id$`	s6
$ 0 3 6	`id+ id$`	s4
$ 0 3 6 4	`+ id$`	r6
$ 0 3 6 3	`+ id$`	r5
$ 0 3 6 8	`+ id$`	r4
$ 0 2	`+ id$`	s5
$ 0 2 5	`id$`	s4
$ 0 2 5 4	$	r6
$ 0 2 5 3	$	r5
$ 0 2 5 2	$	r3
$ 0 2 5 7	$	r2
$ 0 1	$	acc

LR(k) grammars

Informally, we say that a grammar G is LR(k) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation:

1. isolate the handle of each right-sentential form, and
2. determine the production by which to reduce

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

Why study LR grammars?

LR(1) grammars are often used to construct parsers.
We call these parsers LR(1) parsers.

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

LL(k): recognize use of a production $A \rightarrow \beta$ seeing first k symbols derived from β

LR(k): recognize the handle β after seeing everything derived from β plus k lookahead symbols

LR parsing

Three common algorithms to build tables for an “LR” parser:

1. SLR
   - smallest class of grammars
   - smallest tables (number of states)
   - simple, fast construction
2. LR(1)
   - full set of LR(1) grammars
   - largest tables (number of states)
   - slow, large construction
3. LALR(1)
   - intermediate sized set of grammars
   - same number of states as SLR
   - canonical construction is slow and large
   - better construction techniques exist

SLR vs. LR/LALR

An LR(1) parser for either Algol or Pascal has several thousand states, while an SLR or LALR(1) parser for the same language may have several hundred states.

Left versus right recursion

Right Recursion:
- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:
- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:
- right recursion for top-down parsers
- left recursion for bottom-up parsers

Parsing review

- Recursive descent
  A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).
- LL(k)
  An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.
- LR(k)
  An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Closing remarks - parsing

- Overview of Parsing.
- Error checking.
- LR parsing.

Reading:
- Ch 1, 3, 4 from the Dragon book.

Announcement:
- Assignment 1 is out. Due on Friday.
- Next class: ?