Lecture 11

COMPILER DESIGN

Announcements

- **HW 3:** Compiling LLVMlite
  - Familiarize yourself with (a subset of) the LLVM IR
  - Implement a translation down to (inefficient) X86lite

RECAP

Parsing

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

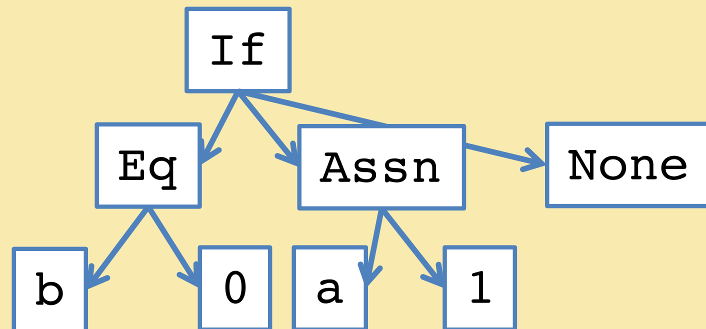
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	1	;	}

Parsing

Abstract Syntax Tree:



Analysis &
Transformation

Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b,
0
  br i1 %cnd, label %l2,
label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Backend

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

Context-free Grammars (CFGs)

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical token, but how about ε?)
  - A set of *nonterminals* (e.g., S and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of *productions*: LHS $\longmapsto$ RHS
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

- Example: The balanced parentheses language

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Derivations in CFGs

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

- Example: derive (1 + 2 + (3 + 4)) + 5
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$
  $\longmapsto (\underline{\mathbf{S}}) + S$
  $\longmapsto (\underline{\mathbf{E}} + S) + S$
  $\longmapsto (1 + \underline{\mathbf{S}}) + S$
  $\longmapsto (1 + \underline{\mathbf{E}} + S) + S$
  $\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$
  $\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$
  $\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$
  $\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$
  $\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$
  $\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$
  $\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$
  $\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$
  $\longmapsto (1 + 2 + (3 + 4)) + 5$

Left- and Rightmost Derivations

- Leftmost derivation
- **S** ↦ **E** + S
  ↦ (**S**) + S
  ↦ (**E** + S) + S
  ↦ (1 + **S**) + S
  ↦ (1 + **E** + S) + S
  ↦ (1 + 2 + **S**) + S
  ↦ (1 + 2 + **E**) + S
  ↦ (1 + 2 + (**S**)) + S
  ↦ (1 + 2 + (**E** + S)) + S
  ↦ (1 + 2 + (3 + **S**)) + S
  ↦ (1 + 2 + (3 + **E**)) + S
  ↦ (1 + 2 + (3 + 4)) + **S**
  ↦ (1 + 2 + (3 + 4)) + **E**
  ↦ (1 + 2 + (3 + 4)) + 5

Rightmost derivation

S ↦ E + **S**
↦ E + **E**
↦ **E** + 5
↦ (**S**) + 5
↦ (E + **S**) + 5
↦ (E + E + **S**) + 5
↦ (E + E + **E**) + 5
↦ (E + E + (**S**)) + 5
↦ (E + E + (E + **S**)) + 5
↦ (E + E + (E + **E**)) + 5
↦ (E + E + (**E** + 4)) + 5
↦ (E + **E** + (3 + 4)) + 5
↦ (**E** + 2 + (3 + 4)) + 5
↦ (1 + 2 + (3 + 4)) + 5

S ↦ E + S | E
E ↦ number | (S)

Searching for derivations

LL & LR PARSING

Top-down vs. Bottom up

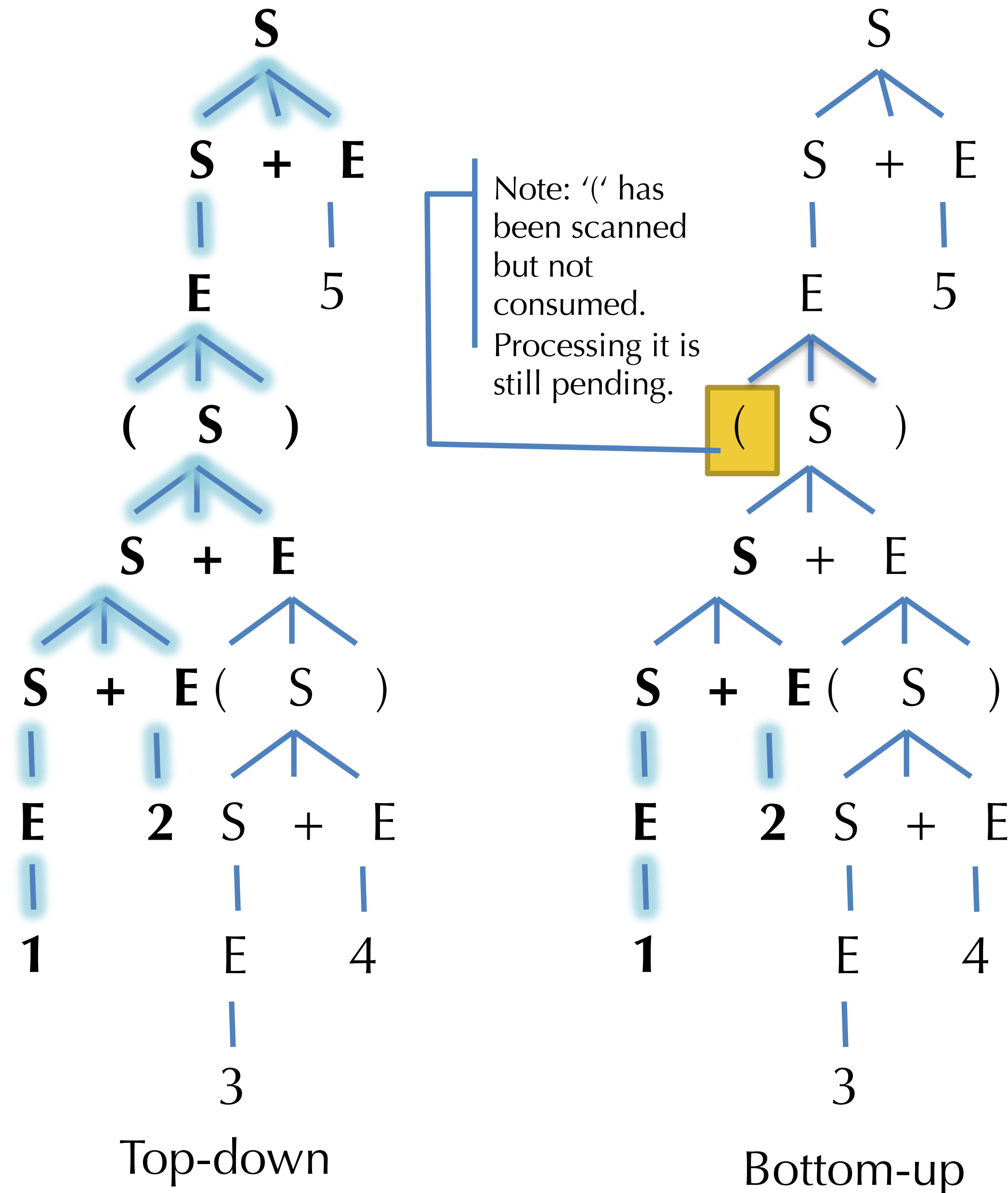
- Consider the left-recursive grammar

$S \mapsto S + E \mid E$
$E \mapsto \text{number} \mid (S)$

- (1 + 2 + (3 + 4)) + 5

- What part of the tree must we know after scanning just (1 + 2

- In top-down, must be able to guess which productions to use



Consider finding left-most derivations

Look at only **one** input symbol at a time

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
$\underline{\mathbf{S}}$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto \underline{\mathbf{E}} + S$	(	(1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{S}}) + S$	1	(/ 1 + 2 + (3 + 4)) + 5
$\longmapsto (\underline{\mathbf{E}} + S) + S$	1	(/ 1 + 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{S}}) + S$	2	(1 + / 2 + (3 + 4)) + 5
$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$	2	(1 + / 2 + (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$	(	(1 + 2 + / (3 + 4)) + 5
$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$	(	(1 + 2 + / (3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$	3	(1 + 2 + (/ 3 + 4)) + 5
$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$	3	(1 + 2 + (/ 3 + 4)) + 5
$\longmapsto \ldots$		

There is a problem

- Want to decide *which production to apply* based on the *look-ahead symbol*

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

- But, consider

(1) $S \longmapsto E \longmapsto (S) \longmapsto (E) \longmapsto (1)$

(1) + 2 $S \longmapsto E + S \longmapsto (S) + S \longmapsto (E) + S \longmapsto (1) + S \longmapsto (1) + E \longmapsto (1) + 2$

- Given the look-ahead symbol '(', it is unclear whether to pick first
  - $S \longmapsto E$ or
  - $S \longmapsto E + S$

LL(1) GRAMMARS

Grammar is the problem

- Not all grammars can be parsed "top-down" with a single lookahead
- *Top-down*
  - Start from the start symbol (root of the parse tree), and
  - Go down

- LL(1) means
  - **L**eft-to-right scanning
  - **L**eft-most derivation,
  - **1** lookahead symbol

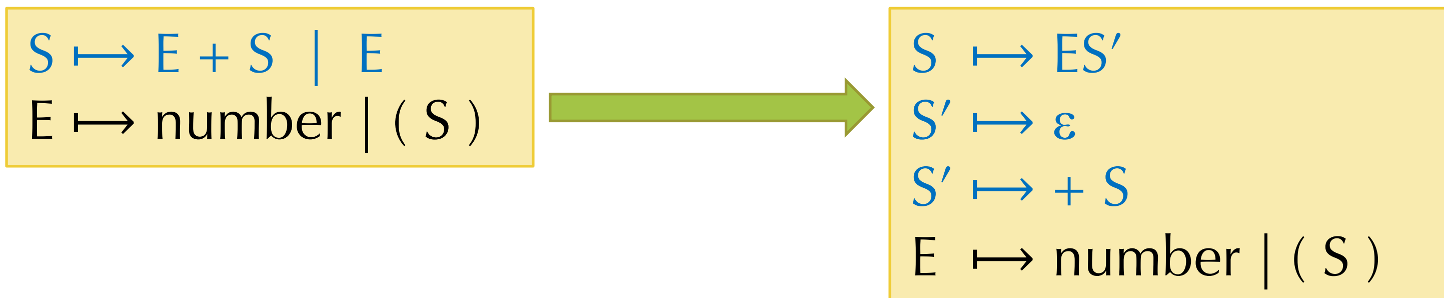
- This language isn't "LL(1)"
- Is it LL(k) for some k?

$$S \longmapsto E + S \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

- What can we do?

Making a grammar LL(1)

- ***Problem***: We can't decide which S production to apply until we see the symbol after the first expression
- ***Solution***: "**Left-factor**" the grammar. There is a common S prefix for each choice, so add a new non-terminal S′ at the decision point



Making a grammar LL(1)

- ***Problem***: We can't decide which S production to apply until we see the symbol after the first expression

- Also need to **eliminate left-recursion** somehow. Why?

- Consider

$$S \longmapsto S + E \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

$$S \longmapsto S + E \longmapsto S + E + E \longmapsto S + E + E + E \longmapsto \ldots$$

How to Remove Left Recursion?

- In general

$$S \to S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- Rewrite as

$$S \to \beta_1\,S' \mid \ldots \mid \beta_m\,S'$$
$$S' \to \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$$

$S \longmapsto S + E \mid E$
$E \longmapsto \text{number} \mid (\ S\ )$

$S \longmapsto E\ S'$
$S' \longmapsto +\ E\ S' \mid \varepsilon$
$E \longmapsto \text{number} \mid (\ S\ )$

LL(1) parse of the input string

Look at only **one** input symbol at a time

$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
S	(	(1 + 2 + (3 + 4)) + 5
⟼ **E** S′	(	(1 + 2 + (3 + 4)) + 5
⟼ (**S**) S′	1	(/ 1 + 2 + (3 + 4)) + 5
⟼ (**E** S′) S′	1	(/ 1 + 2 + (3 + 4)) + 5
⟼ (1 **S′**) S′	+	(1 / + 2 + (3 + 4)) + 5
⟼ (1 + **S**) S′	2	(1 + / 2 + (3 + 4)) + 5
⟼ (1 + **E** S′) S′	2	(1 + / 2 + (3 + 4)) + 5
⟼ (1 + 2 **S′**) S′	+	(1 + 2 / + (3 + 4)) + 5
⟼ (1 + 2 + **S**) S′	(	(1 + 2 + / (3 + 4)) + 5
⟼ (1 + 2 + **E** S′) S′	(	(1 + 2 + / (3 + 4)) + 5
⟼ (1 + 2 + (**S**)S′) S′	3	(1 + 2 + (/ 3 + 4)) + 5

Predictive Parsing

- Given an LL(1) grammar
  - For a given nonterminal, the lookahead symbol uniquely determines the production to apply
  - Top-down parsing = predictive parsing
  - Driven by a predictive parsing table
    nonterminal × input token → production

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ number		⟼ (S)		

- Note: It is convenient to add a special *end-of-file* token $ and a start symbol T (top-level) that requires $

LL(1) parse of the input string

	number	+	(	)	$ (EOF)
T	$\mapsto$ S$		$\mapsto$S$		
S	$\mapsto$ E S′		$\mapsto$E S′		
S′		$\mapsto$ + S		$\mapsto$ ε	$\mapsto$ ε
E	$\mapsto$ number		$\mapsto$ (S)		

Partly-derived String	Look-ahead	Parsed/Unparsed Input
T	(	(1 + 2 + (3 + 4)) + 5$
$\mapsto$ **S**$	(	(1 + 2 + (3 + 4)) + 5$
$\mapsto$ **E** S′$	(	(1 + 2 + (3 + 4)) + 5$
$\mapsto$ (**S**) S′$	1	(/ 1 + 2 + (3 + 4)) + 5$
$\mapsto$ (**E** S′) S′$	1	(/ 1 + 2 + (3 + 4)) + 5$
$\mapsto$ (1 **S′**) S′$	+	(1 / + 2 + (3 + 4)) + 5$
$\mapsto$ (1 + **S**) S′$	2	(1 + / 2 + (3 + 4)) + 5$
$\mapsto$ (1 + **E** S′) S′$	2	(1 + / 2 + (3 + 4)) + 5$
$\mapsto$ (1 + 2 **S′**) S′$	+	(1 + 2 / + (3 + 4)) + 5$
$\mapsto$ (1 + 2 + **S**) S′$	(	(1 + 2 + / (3 + 4)) + 5$
$\mapsto$ (1 + 2 + **E** S′) S′$	(	(1 + 2 + / (3 + 4)) + 5$
$\mapsto$ (1 + 2 + (**S**)S′) S′$	3	(1 + 2 + (/ 3 + 4)) + 5$

How do we construct the parse table?

- Consider a given production: A → γ
- **(Case 1)** Construct the set of all input tokens that may appear *first* in strings that can be derived from γ
  - Add the production → γ to the entry (A,token) for each such token
- **(Case 2)** If γ can derive ε (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal A in the grammar
  - Add the production → γ to the entry (A, token) for each such token
- **Important**: if there are two different productions for a given entry, the grammar is not LL(1)

Example

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { +, ε }
- First(E) = { number, ‘(‘ }

- Follow(S′) = Follow(S)
- Follow(S) = { $, ‘)′ } ∪ Follow(S′)

$T \longmapsto S\$$
$S \longmapsto ES'$
$S' \longmapsto \varepsilon$
$S' \longmapsto + S$
$E \longmapsto \text{number} \mid (S)$

Note: we want the *least* solution to this system of set equations… a *fixpoint* computation. More on these later in the course.

	number	**+**	**(**	**)**	**$ (EOF)**
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ number		⟼ (S)		

Converting the table to code

$$
\begin{aligned}
T &\longmapsto S\$ \\
S &\longmapsto ES' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto + S \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

- Define **N** mutually recursive functions
  - One for each nonterminal **A**: **parse_A**
  - **parse_A** of type `unit -> ast` if A is *not* an auxiliary nonterminal
  - Otherwise, **parse_A** takes extra ast's as inputs, one for each nonterminal in the "factored" prefix

Converting the table to code

	number	+	(	)	$ (EOF)
T	$\longmapsto$ S$		$\longmapsto$S$		
S	$\longmapsto$ E S′		$\longmapsto$E S′		
S′		$\longmapsto$ + S		$\longmapsto$ ε	$\longmapsto$ ε
E	$\longmapsto$ number		$\longmapsto$ (S)		

- Each function **parse_A**
  - "Peeks" at the lookahead token
  - follows the production rule in the corresponding entry
  - Consume terminal tokens from the input stream
  - Call **parse_X** to create sub-tree for nonterminal **X**
  - If the rule ends in an auxiliary nonterminal, call it with appropriate ast's (The auxiliary rule creates the ast after looking at more input)
  - Otherwise, this function builds the ast tree itself and returns it

LL(1) Summary

- Top-down parsing that finds the leftmost derivation

- Language grammar ⇒
  LL(1) grammar ⇒
  prediction table ⇒
  recursive-descent parser

- Parser generator based on LL: ANTLR

- Problems
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger)
  - Grammar cannot be left recursive (parser functions will loop!)

- Is there a better/alternative way?

	number	+	(	)	$ (EOF)
T	$\longmapsto$ S$		$\longmapsto$S$		
S	$\longmapsto$ E S′		$\longmapsto$E S′		
S′		$\longmapsto$ + S		$\longmapsto$ ε	$\longmapsto$ ε
E	$\longmapsto$ number		$\longmapsto$ (S)		

Hand-generated LL(1) code for the table above

DEMO: PARSER.ML