Warm-Up Problem

Consider the following grammar

$$
\begin{array}{llll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & aS & (1) \\
S & \rightarrow & B & (2) \\
B & \rightarrow & aBb & (3) \\
B & \rightarrow & \varepsilon & (4)
\end{array}
$$

Draw the bottom-up parsing DFA for this grammar as we did last time.

CS 241 Lecture 14

Bottom-Up Parsing and Type Checking
With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Notation and Procedure

Definition

An **item** is a production with a dot • somewhere on the right hand side of a rule.

- Items indicate a partially completed rule.
- We will begin in a state labelled by the rule $S' \to \bullet \vdash S \dashv$
- That dot is called the "**bookmark**"

LR(0) Construction

- From a state, for each rule in the state, move the dot forward by one character. The transition function is given by the symbol you jumped over.
  - For example, with $S' \to \bullet \vdash S \dashv$, we move the $\bullet$ over $\vdash$. Thus, the transition function will consume the symbol $\vdash$.
  - The state we end up in will contain the item $S' \to \vdash \bullet S \dashv$. It also contains more!

LR(0) Construction

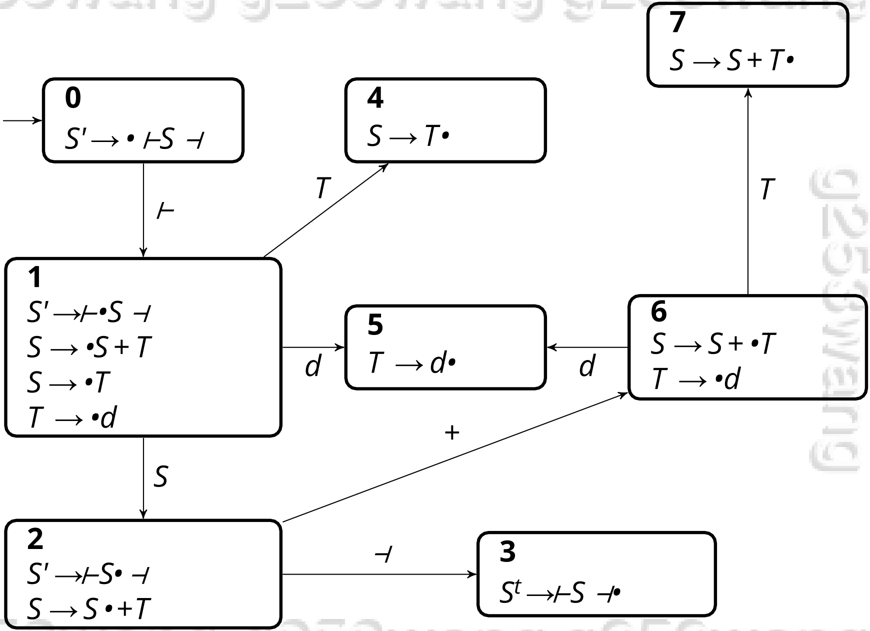
- In the new state, if in the set of items we have •A for some non-terminal A, we then add all rules with A in the left-hand side of a production with a dot preceding the right-hand side!
  - In this case, this state will include the rules $S \to \bullet S + T$ and $S \to \bullet T$.
  - Notice now we also have •T and so we also need to include the rules where T is the left-hand side, adding the rule $T \to \bullet d$.

LR(0) Construction

- If we find ourselves at a familiar state, reuse it instead of remaking it.
- We continue with these steps until there are no bookmarks left to move. Then we have the final DFA.
- We skipped the ε-NFA step by putting all these items in the same rule. You may see versions of this algorithm that involve building an ε-NFA and then converting, but the result will be the same.

Back to our Example

$$
\begin{array}{llll}
S' & \to & \vdash S \dashv & (0) \\
S & \to & S + T & (1) \\
S. & \to & T & (2) \\
T. & \to & d & (3)
\end{array}
$$


0
S' → • ⊢S ⊣
⊢
1
S' →⊢•S ⊣
S → •S + T
S → •T
T → •d
T
4
S → T•
d
5
T → d•
d
6
S → S + •T
T → •d
T
7
S → S + T•
S
+
2
S' →⊢S• ⊣
S → S •+T
⊣
3
St →⊢S ⊣•

Using the Automaton

- This automaton is our faerie! Run the *stack* through the automaton, and:
  - If you end up in a state with the bookmark at the right-hand side of an item, perform that reduction (you've read the right-hand side)
  - If you end up in a state with the bookmark elsewhere, shift
  - Else (error state), reject

Algorithm, Try 2

Algorithm LR(0) algorithm, inefficiently

for each symbol a in $\vdash x \dashv$ from left to right **do**
  S $\leftarrow$ final state of the LR(0) DFA run on the stack
  while S is a reduce state labeled with an item for some production $B \rightarrow \gamma$ **do**
    stack.pop symbols in γ
    stack.push B
    S $\leftarrow$ final state of the LR(0) DFA run on the stack
  end while
  if S is the error state **then**
    reject
  end if
  stack.push a
end for
accept

Observation

- The stack is a stack, so the bottom of the stack (beginning of our input) doesn't usually change
- We're rerunning the whole DFA even when the the prefix of our stack is the same
- Because of this, our algorithm is $O(n^2)$!

Fix

- Remember how we moved through the DFA in a *state stack*, and push and pop to the state stack at the same time as the symbol stack. That way, we don't repeat getting to a state with a prefix that hasn't changed.
- This brings us to O(n)

LR(0)

Algorithm 1 LR(0) algorithm, input LR(0) DFA(Σ,Q,q_0,δ,A)

stateStack.push q_0
for each symbol a in $\vdash x \dashv$ from left to right **do**
    while Reduce[stateStack.top] is some production $B \to \gamma$ **do**
        symStack.pop symbols in γ
        stateStack.pop $|\gamma|$ states
        symStack.push B
        stateStack.push $\delta[\text{stateStack.top}, B]$
    end while
    symStack.push a
    reject if $\delta[\text{stateStack.top}, a]$ is undefined
    stateStack.push $\delta[\text{stateStack.top}, a]$
end for
accept

Possible Issues

Issue one (Shift-Reduce): What if a state has two items of the form:

- $A \to \alpha \cdot a\beta$
- $B \to \gamma\cdot$

Should we shift or reduce?

Possible Issues

Issue two (Reduce-Reduce): What if a state has two items of the form:

- $A \rightarrow \alpha\cdot$
- $B \rightarrow \gamma\cdot$

Which reduction should we do?

Possible Issues

Note, having two items that shift, e.g.:

- $A \to \alpha \cdot a\beta$
- $B \to \gamma \cdot b\delta$

is *not* an issue! (Why?)

Definition

Definition

A grammar is LR(0) if and only if after creating the automaton, no state has a shift-reduce or reduce-reduce conflict.

Practice: The example of bottom-up parsing form last lecture was LR(0)!

Question

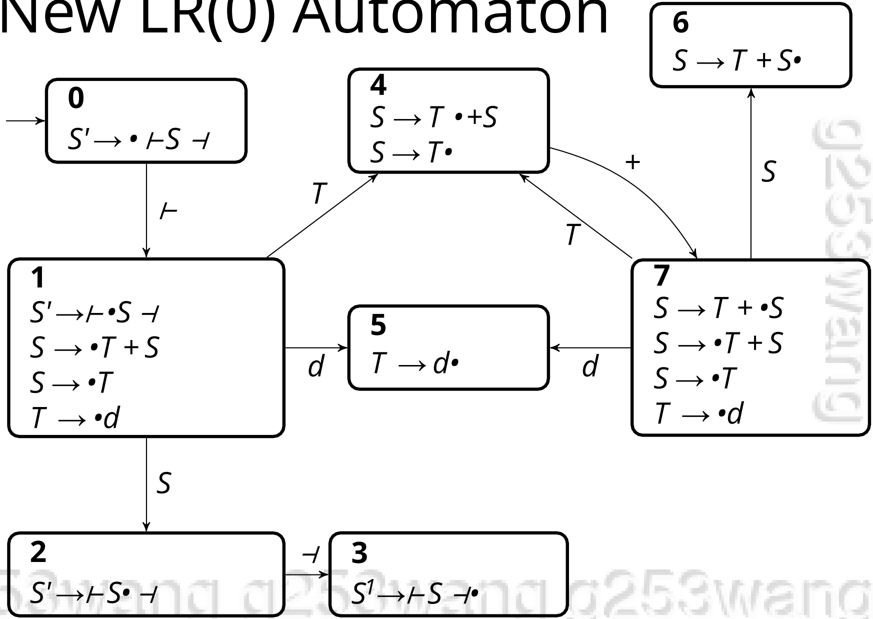
Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language?

Not usually! Bottom-up parsing can support left and right recursive grammars. However, not all grammars are LR(0) grammars. Consider the following grammar (changed rule 1):

$$
\begin{array}{llll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & T + S & (1) \\
S. & \rightarrow & T & (2) \\
T. & \rightarrow & d & (3)
\end{array}
$$

New LR(0) Automaton



Conflict

State 4 has a shift-reduce conflict.

- Suppose the input began with $\vdash$ d .
- This gives a stack of $\vdash$ d and then we reduce in state 5, so our stack changes to $\vdash$ T and we move to state 4 via state 1.
- Should we reduce $S \to T$?
- It depends! If the input is $\vdash$ d $\dashv$ then absolutely!
- If instead, the input was $\vdash$ d + ... then no!

How do we fix this?

Lookahead!

We'll add a lookahead to the automaton to fix the conflict!
For every $A \to \alpha \bullet$, attach Follow(A)! Recall:

$$
\begin{array}{llll}
S' & \to & \vdash S \dashv & (0) \\
S & \to & T + S & (1) \\
S. & \to & T & (2) \\
T. & \to & d & (3)
\end{array}
$$

What is Follow(S)? What about Follow(T)?

Follow Sets

Note that Follow(S) = $\{\dashv\}$ and Follow(T) = $\{+, \dashv\}$. So, state 4 becomes

$$S \to T \bullet +S \qquad \text{and} \qquad S \to T \bullet : \{\dashv\}$$

In other words, apply $S \to T \bullet +S$ if the next token is +, and apply $S \to T \bullet \{\dashv\}$ if the next token is $\dashv$.

With lookahead from Follow sets on reduce states, we call these parsers SLR(1) parsers! (Simplified LR with 1 character look ahead).

Like most names, this is a terrible name. SLR(1) isn't a simplified version of LR(1), it's just different from LR(1). Don't read too much into the name.

LR(1) Algorithm

Algorithm 2 LR(1) algorithm, input SLR(1) or LALR(1) or LR(1) DFA(Σ,Q,q_0,δ,A)

stateStack.push q_0
for each symbol a in $\vdash x \dashv$ from left to right **do**
  while Reduce[stateStack.top, a] is some production $B \to \gamma$ **do**
    symStack.pop symbols in γ
    stateStack.pop $|\gamma|$ states
    symStack.push B
    stateStack.push δ[stateStack.top, B]
  end while
  symStack.push a
  reject if δ[stateStack.top, a] is undefined
  stateStack.push δ[stateStack.top, a]
end for
accept

The only change!

SLR? What happened to LR?

- LR(1) parsing involves a more complicated procedure.
- Instead of adding all of Follow(S) to an item, you add only a subset of this set to each item.
- In this way, the number of states you get can blow up exponentially depending on your follow sets.

SLR? What happened to LR?

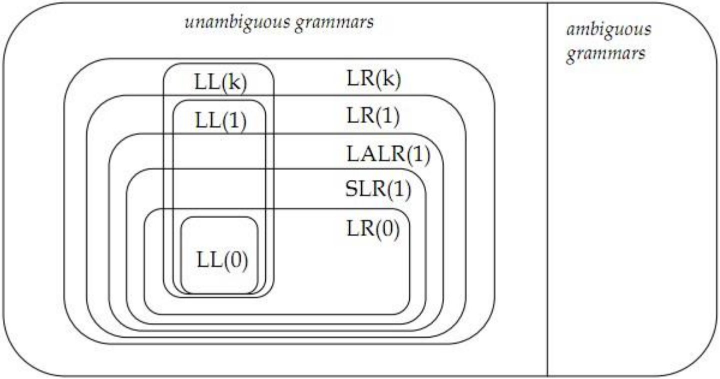
- However, the parsing mechanism is the same (just the automaton changes). Programming an SLR parser then swapping in an LR(1) automaton gives you an LR(1) parser.

SLR? What happened to LR?

- LR(1) parsers are extremely powerful; Knuth proved that if you have a language recognized by a LR(k) grammar for k > 1, then there is a LR(1) grammar recognizing the same language!
- We don't cover LR(1) in this course (or LALR) because SLR(1) is sufficient for nearly all practical languages, and as stated, the only difference is the automaton anyway

LL(1) versus LR(k)

A picture is worth a thousand words:



Source: https://i.stack.imgur.com/TqAkP.png

Recall: Every language accepted by a LR(k) grammar can be accepted by some LR(1) grammar!

LR(1) Algorithm

Algorithm 2 LR(1) algorithm, input SLR(1) or LALR(1) or LR(1) DFA(Σ,Q,q_0,δ,A)

stateStack.push q_0
for each symbol a in $\vdash x \dashv$ from left to right **do**
    while Reduce[stateStack.top, a] is some production $B \to \gamma$ **do**
        symStack.pop symbols in γ
        stateStack.pop $|\gamma|$ states
        symStack.push B
        stateStack.push δ[stateStack.top, B]
    end while
    symStack.push a
    reject if δ[stateStack.top, a] is undefined
    stateStack.push δ[stateStack.top, a]
end for
accept

The only change!

Building the Parse Tree

- With top-down parsing, when you, for example, pop S from the stack and push B, y and A: S is a node, make the new symbols the children.
- With bottom-up parsing, when you, e.g., reduce $A \rightarrow ab$ (from a stack with a and b). You then keep these two old symbols as children of the new node A.
  - Ideally, you have a stack of tree fragments!

Example

Recall our grammar:

$$
\begin{array}{lcll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & AcB & (1) \\
A & \rightarrow & ab & (2) \\
A & \rightarrow & ff & (3) \\
B & \rightarrow & def & (4) \\
B & \rightarrow & ef & (5)
\end{array}
$$

We processed $w = \ \vdash abcdef \dashv$ using this bottom-up technique

Now we'll build the parse tree on the board.

Recall Parsing Bottom-Up

Stack	Read	Processing	Action
	ε	$\vdash abcdef \dashv$	Shift $\vdash$
$\vdash$	$\vdash$	$abcdef \dashv$	Shift a
$\vdash a$	$\vdash a$	$bcdef \dashv$	Shift b
$\vdash ab$	$\vdash ab$	$cdef \dashv$	Reduce (2); pop b, a, push A
$\vdash A$	$\vdash ab$	$cdef \dashv$	Shift c
$\vdash Ac$	$\vdash abc$	$def \dashv$	Shift d
$\vdash Acd$	$\vdash abcd$	$ef \dashv$	Shift e
$\vdash Acde$	$\vdash abcde$	$f \dashv$	Shift f
$\vdash Acdef$	$\vdash abcdef$	$\dashv$	Reduce (4); pop f, d, e push B
$\vdash AcB$	$\vdash abcdef$	$\dashv$	Reduce (1); pop B, c, A push S
$\vdash S$	$\vdash abcdef$	$\dashv$	Shift $\dashv$
$\vdash S \dashv$	$\vdash abcdef \dashv$	ε	Reduce (0); pop $\dashv, S, \vdash$ push S'
S'	$\vdash abcdef \dashv$	ε	Accept

A Last Parser Problem

- Most famous problem in parsing: the dangling else!
- Let’s go over an if-then-else grammar on the board…

Context-Sensitive Analysis

Not everything can be enforced by a CFG!
Examples:

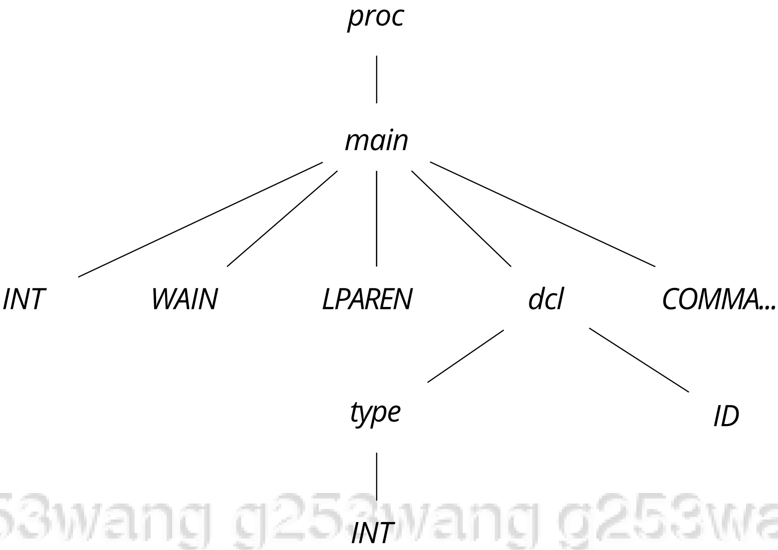
- Type checking
- Declaration before use
- Scoping (is a variable defined in the correct scope)
- Well-typed expressions (is a == b well-typed)

To solve these, we can move to context-sensitive languages

Context-Sensitive Languages?

- As it turns out, CSLs aren't a very useful formalism
- We already needed to give up many CFGs to make a parser handle CFLs; with CSLs, it would be even worse!
- As such, we treat context-sensitive analysis as *analysis* (looking over the parse tree generated by CFL parsing) instead of *parsing* (making its own data structure)

Simplified approach: We will traverse our parse tree to do our analysis



In Code

```
class Tree{ public:
    string rule; // e.g. expr
    vector<string> tokens; // e.g. expr + term
    vector<Tree> children;
};
```

Then could traverse a tree...

```
void doSomething(const Tree &t){
  for(const auto &i: t.children){
    doSomething(i);
  }
}
```

Errors

Errors we still need to check for:

- Variable declared more than once
- Variable used but not declared
- Type errors
- Scoping as it applies to the above

Declaration Errors

- How do we determine multiple/missing declaration errors?
- We've done this before!
- Construct a symbol table! To create:
  - Traverse the parse tree for any rules of the form dcl -> TYPE ID.
  - Add the ID to the symbol table
  - If the name is in the table, give an error.

Checking

- To verify that variables have been declared
- Check for rules of the form factor -> ID and lvalue -> ID.
- if ID is not in the symbol table, produce an error
- The previous two passes can be merged (and must be merged!)

Checking

- Thought experiment: With labels in MIPS in the assembler, we needed two passes. Why do we only need one in the compiler?
- We need to declare variables before using them! Not true for labels!

Types

- Note that in the symbol table, we should also keep track of the type of the variables.
- Why is this important?
- Just by looking at bits, we cannot figure out what it represents! Types for WLP4 allow us to interpret the contents of memory addresses.

Types

- Good systems prevent us from interpreting bits as something we shouldn't.
- For example
  ```
  int *a = NULL;
  a = 7;
  ```
  should be a type mismatch since we're trying to store an integer in a memory address.
- This is just a matter of interpretation! All the compiler is doing is making sure that we keep our own promises.

Types in WLP4

- In WLP4, there are two types: `int` and `int*` for integers and pointers to integers.
  - (This restriction is based on C's predecessor, B!)
- For type checking, we need to evaluate the types of expressions and then ensure that the operations we use between types corresponds correctly.

Types in WLP4

- If given a variable in the wild, how do we determine its type?
- Use its declaration! Need to add this to the symbol table.

Symbol Table Implementation

We can use a global variable to keep track of the symbol table:

```
map<string, string> symbolTable; // name -> type
```

but by now you know nothing is ever this easy! What can go wrong?

- This doesn’t take scoping into account!
- Also need something for functions/declarations!

Issues

- Consider the following code (specifically with x). Is there an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int wain(int x, int y) {
    return foo(y) + x;
}
```

- No! Duplicated variables in different procedures are okay!

Issues

- Is the following an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int wain(int a, int b) {
    return foo(b) + x;
}
```

- Yes! The variable *x* is not in scope in *wain*!

Issues

- Is the following an error?

```
int foo(int a) {
    int x = 0;
    return x + a;
}
int foo(int b) { return b; }
int wain(int a, int b) {
    return foo(b) + a;
}
```

- Yes! We have multiple declarations of *foo*.