Warm Up Problem

Consider the following grammar

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

Draw the SLR(1) automaton [and hence also an LR(0) automaton] for this grammar. Is it LR(0)? What about SLR(1)?

CS 241 Lecture 15

Bottom Up Parsing Continued

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

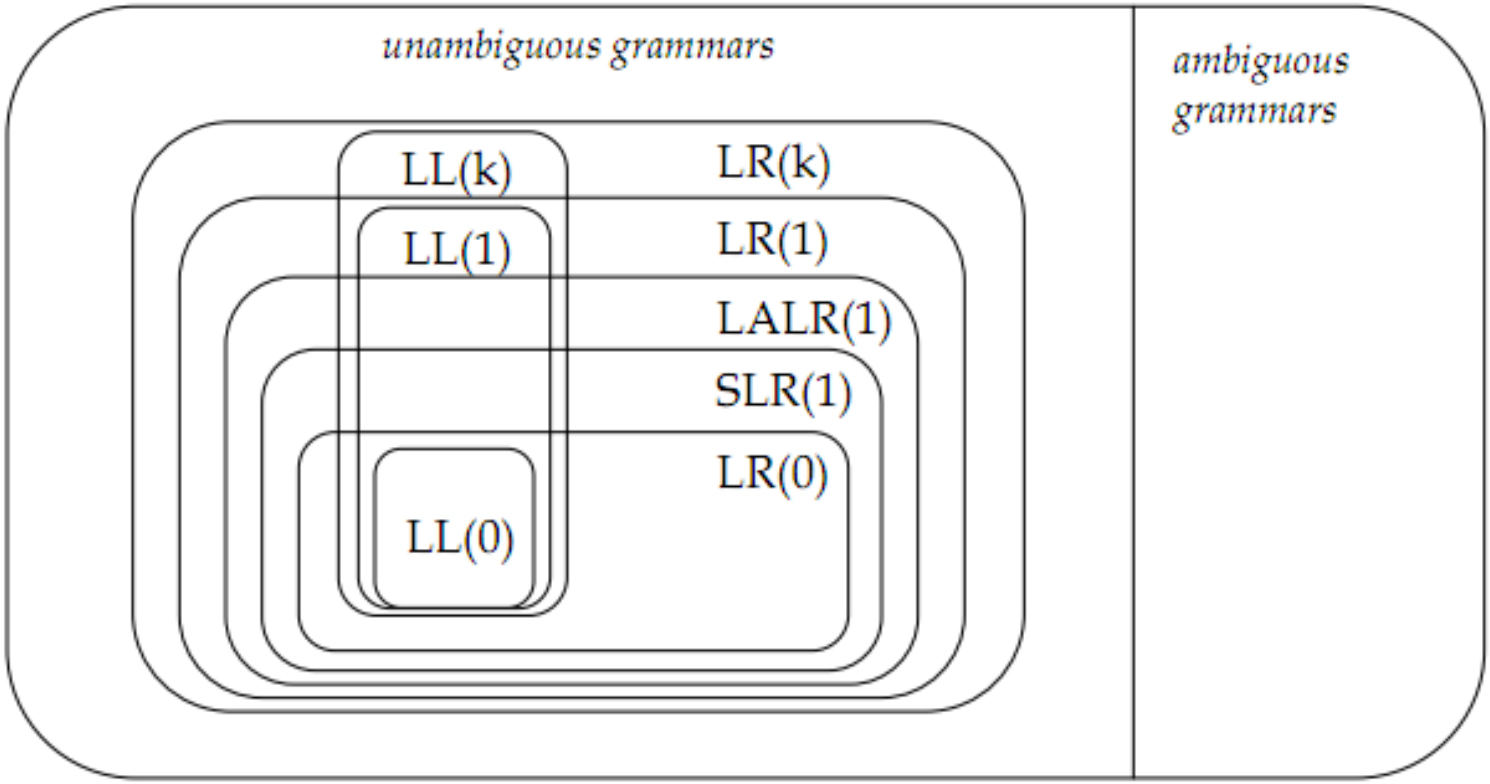
Wait... What is an LR(1) Parser?

- 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 blowup exponentially depending on your follow sets.
- 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.
- Yacc and Bison, for example, use LALR(1) (Lookahead LR) which lies somewhere between SLR(1) and LR(1).
- 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!

Grammar Picture

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!

Algorithm for LR(1) Automaton

Elements in an LR(1) automaton are items followed by a terminal symbol. Let S_M be the set of states of a automaton M

Algorithm 1 LR(1) Automaton Algorithm

1: Make the SLR(1) parser's automaton M
2: **for** each $A \to \alpha \bullet B\beta, \quad t$ line in a state s of S_M **do**
3: **for** each $B \to \gamma$ in P **do**
4: **for** each $b \in \text{First}(\beta t)$ **do**
5: Add $B \to \bullet\gamma, b$ to s
6: **end for**
7: **end for**
8: **end for**
9: Repeat the above until no more changes have occurred.

Sample LR(1) Grammar That is Not SLR(1)

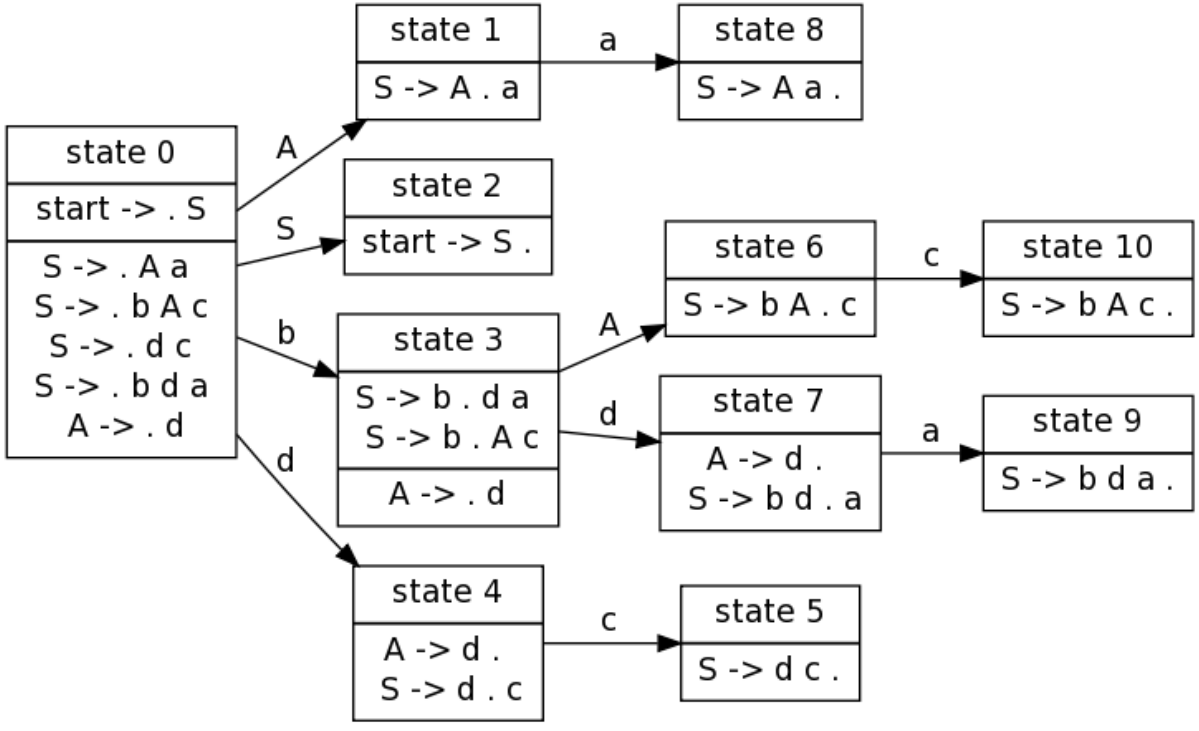
Omitting S' to be consistent with next pages notation.

$$
\begin{array}{lcll}
S & \rightarrow & Aa & (1) \\
S & \rightarrow & bAc & (2) \\
S & \rightarrow & dc & (3) \\
S & \rightarrow & bda & (4) \\
A & \rightarrow & d & (5)
\end{array}
$$

Source: `https://stackoverflow.com/questions/10505717/how-is-this-grammar-lr1-but-not-slr1`

SLR(1) Table

Source of next slides: `http://smlweb.cpsc.ucalgary.ca/`



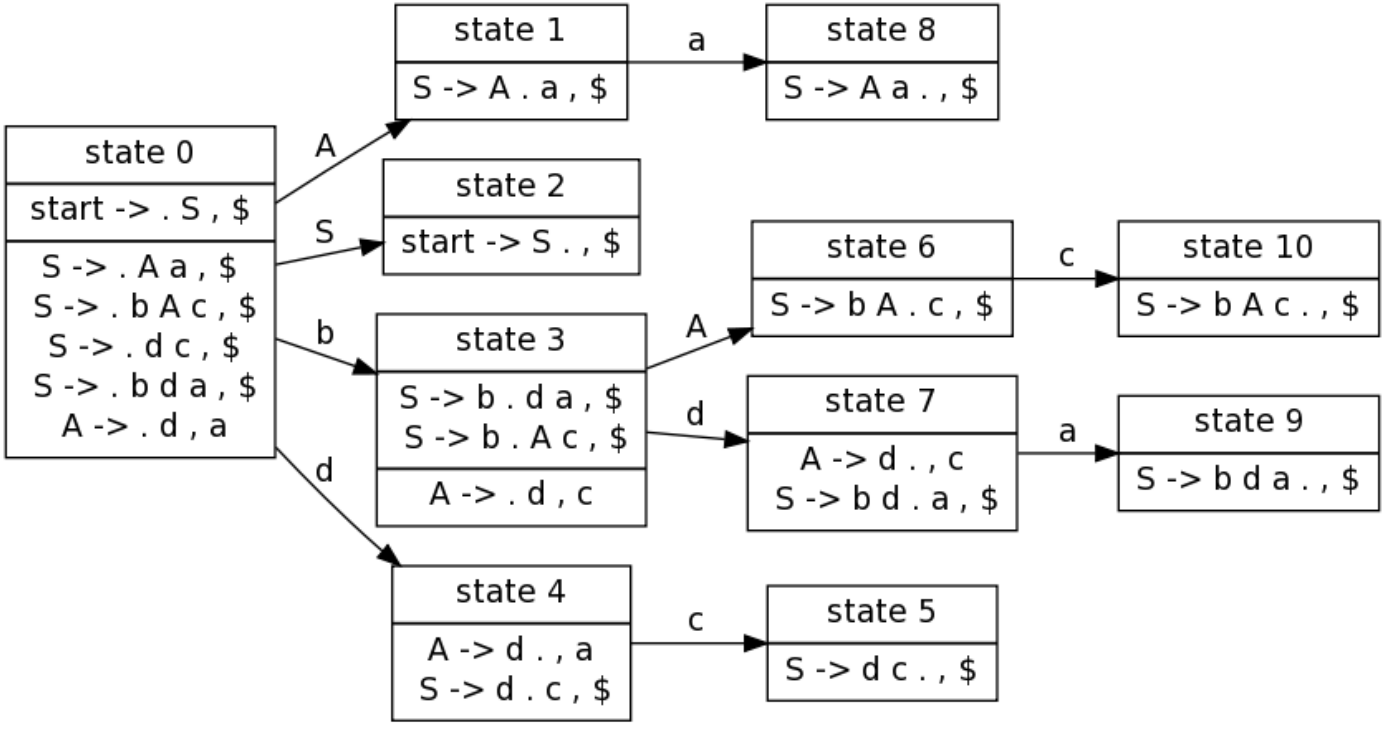
SLR(1) Table

SLR(1) Table

	$	d	a	b	c	S	A
0		s4		s3		s2	s1
1			s8				
2	acc						
3		s7					s6
4			r(A → d)		r(A → d)/s5		
5	r(S → d c)						
6					s10		
7			r(A → d)/s9		r(A → d)		
8	r(S → A a)						
9	r(S → b d a)						
10	r(S → b A c)						

LR(1) Table

Note $ means end of file.



LR(1) Table

LR(1) Table

	$	d	a	b	c	S	A
0		s4		s3		s2	s1
1			s8				
2	acc						
3		s7					s6
4			r(A → d)		s5		
5	r(S → d c)						
6					s10		
7			s9		r(A → d)		
8	r(S → A a)						
9	r(S → b d a)						
10	r(S → b A c)						

It is LR(1)

Building the Parse Tree

- With top-down parsing, say you pop a symbol S from the stack and push B, y and A. Keep S and make the new symbols the children .
- With bottom-up parsing, say you 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.

Example

Recall our grammar:

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

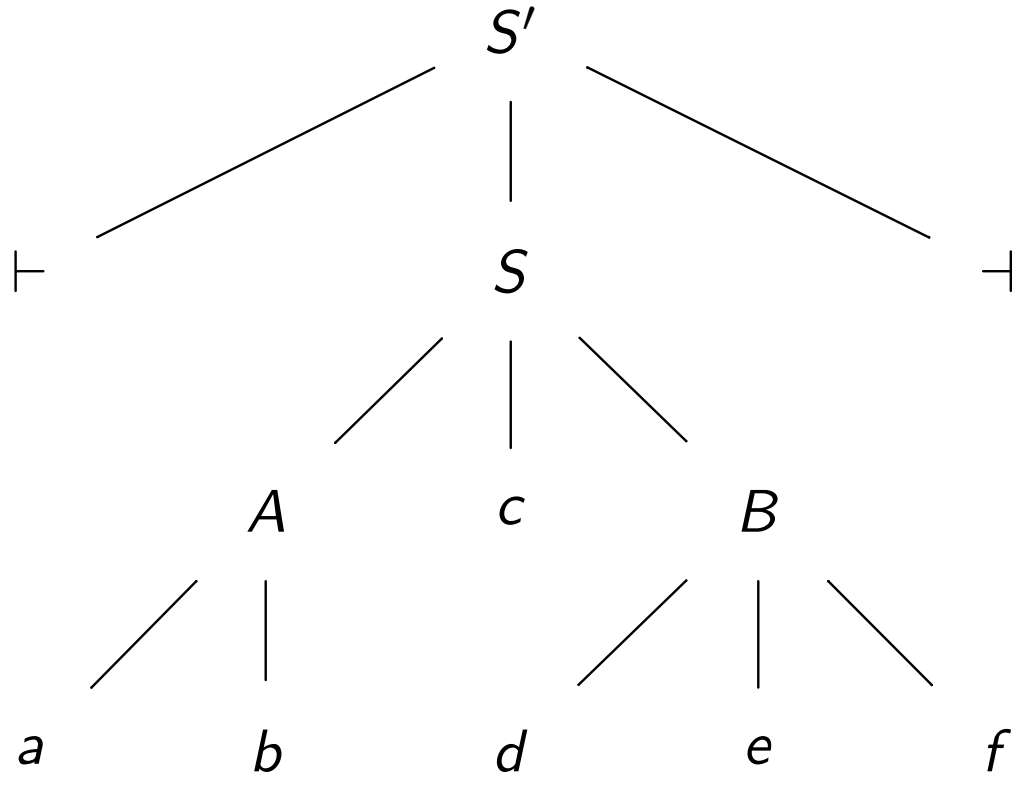
We processed $w = \ \vdash abcdef \dashv$ using this bottom up technique (recall next slide)

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

Parse Tree

- $\vdash$ (shift)
- $\vdash a$ (shift)
- $\vdash ab$ (shift)
- $\vdash {}_{a}A_{b}$ (Reduce)
- $\vdash {}_{a}A_{b}c$ (Shift)
- $\vdash {}_{a}A_{b}cd$ (Shift)
- $\vdash {}_{a}A_{b}cde$ (Shift)
- $\vdash {}_{a}A_{b}cdef$ (Shift)
- $\vdash {}_{a}A_{b}c\,{}^{B}_{d\,e\,f}$ (Reduce)
- $\vdash {}^{S}_{{}_{a}A_{b}c\,{}^{B}_{d\,e\,f}}$ (Reduce)
- $\vdash {}^{S}_{{}_{a}A_{b}c\,{}^{B}_{d\,e\,f}} \dashv$ (Shift)
- Reduce on right



Example Assignment

Your parser will output a .wlp4i file.
Example:

```
S → BOF e EOF
e → e + t
e → t
t → ID
```

with input `BOF a + b + c EOF` (with `a`, `b`, `c` as `ID`s) would produce the file...

Output

```
S BOF e EOF
BOF BOF
e e + t
e e + t
e t
t ID
ID a
+ +
t ID
ID b
+ +
t ID
ID c
EOF EOF
```

Assignment Overview:

- A6: WLP4 Text File to WLP4 Tokens and lexemes (Lexical Analysis)
- A7: WLP4 Tokens and lexemes to parse tree (Syntactic Analysis)
- A8 Parse Trees to Augmented Parse Tree and Symbol Tables (Context-Sensitive Analysis coming next!)
- A9 and 10: Augmented Parse Trees to MIPS Assembly Language (Code Generation)

Context Sensitive Analysis

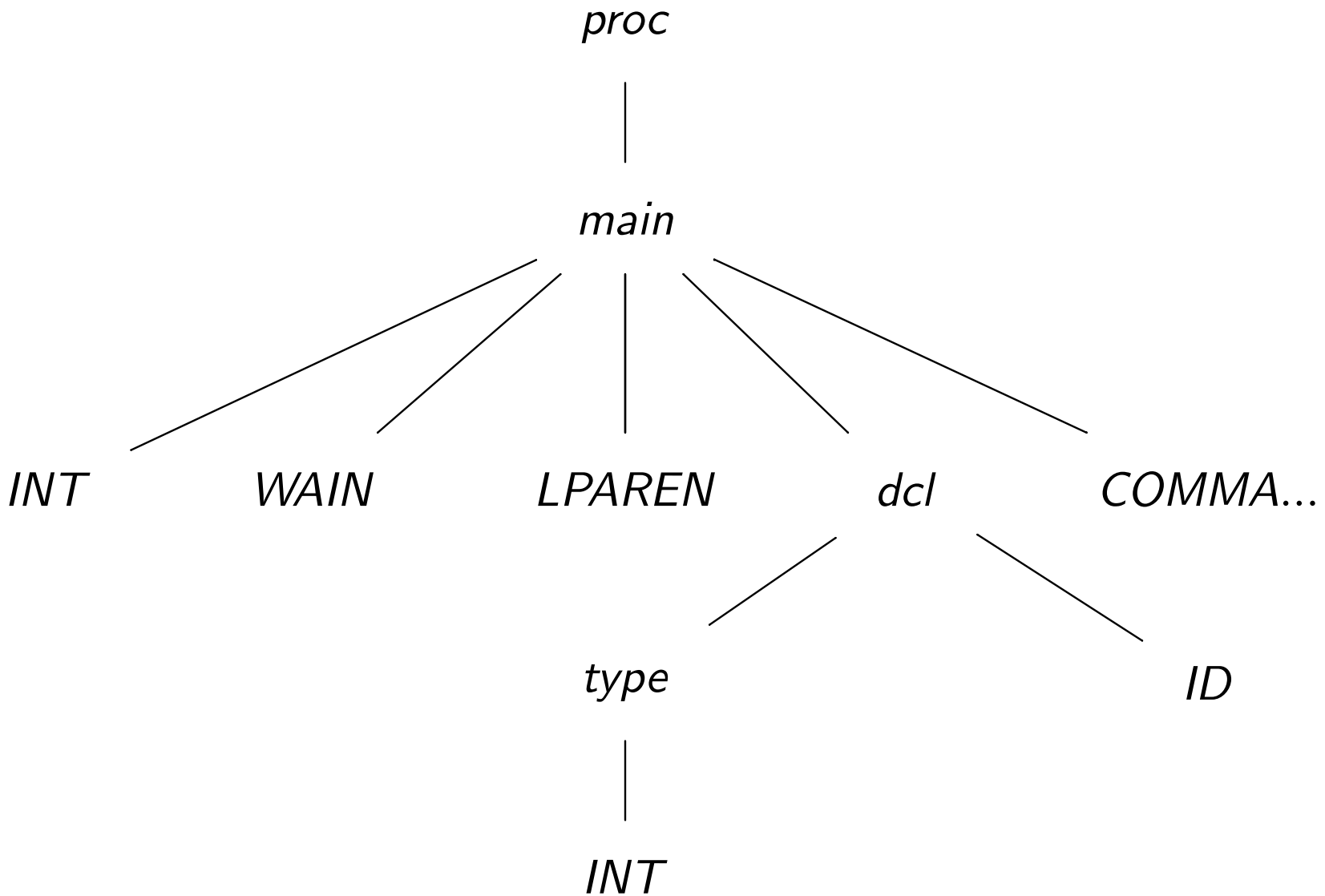
Not everything can be enforced by a CFG! Examples:

- 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 (ones where the context matters - an example was given in the slide deck that introduced CFGs)

Simplification

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



In Code

```
class Tree{
  public:
    string rule; //eg. expr expr PLUS term
    vector<string> tokens;
    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
- Issues with the above and scoping!

Declaration Errors

How do we determine multiple/missing declaration errors?

Declaration Errors

How do we determine multiple/missing declaration errors?

We've done this before!

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!)

Thought experiment: With labels in MIPS in the assembler, we needed two passes. Why do we only need one in the compiler?

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!)

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?

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.
- 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.

Types in WLP4

- In WLP4, there are two types: `int` and `int*` for integers and pointers to integers.
- For type checking, we need to evaluate the types of expressions and then ensure that the operations we use between types corresponds correctly.
- If given a variable in the wild, how do we determine its type?

Types in WLP4

- In WLP4, there are two types: `int` and `int*` for integers and pointers to integers.
- For type checking, we need to evaluate the types of expressions and then ensure that the operations we use between types corresponds correctly.
- 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?

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

Issues

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

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

No! Duplicated variables in different procedures are okay!

Issues

Is the following an error?

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

Issues

Is the following an error?

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

Yes! The variable x has scope belonging to f!

Issues

Is the following an error?

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

Issues

Is the following an error?

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

Yes! We have multiple declarations of f!

Resolution

Any ideas on how we can resolve this?

Resolution

Any ideas on how we can resolve this?

One idea is that we can have a top level symbol table that collects all the procedure names and symbol tables for each procedure:

```
map<string, map<string, string> > topSymbolTable;
```

So now, as we traverse the parse tree, when we encounter rules `proc -> INT ID LPAREN...` or `MAIN-> INT WAIN ...` we have a new procedure! Just check to make sure that the name is not already in the symbol table. If not, add it to the table and if so return an error.

Pro Tip: You may want a global variable `curProc` that keeps track of which procedure you are currently in. Update each time you see a new procedure or main production.

Not Quite Enough

- For variables, we also had to store the type for type checking.
- What, if anything, should we store for procedures?

Not Quite Enough

- For variables, we also had to store the type for type checking.
- What, if anything, should we store for procedures?
- Probably want to store the signature [to check for correct calls]! Note that in WLP4, everything (thankfully!) returns an `int` so we just need the parameter signature.
- Symbol table should really be procedure name, and pairs of signature and symbol tables:

```
map<string,
    pair<vector<string>,
         map<string, string> > >
             topSymbolTable;
```

Computing Signature

Simply need to traverse nodes in your parse tree of the form:

- `params ->`
- `params -> paramlist`
- `paramlist -> dcl`
- `paramlist -> dcl`
- `paramlist -> dcl COMMA paramlist`

Again, all of this can be done in a single pass.

An Example

Consider

```
int f() {
  int *a = NULL;
  return 9;
}
int wain(int a, int b){
  int x = 10;
  return x+a+b;
}
```

Then your symbol table contains two entries:

- `f -> <>, <a -> int*>`
- `wain -> <int, int>, <a -> int, b -> int, x -> int>`