Compilation 2013

Warm-up Project and Lexical Analysis

Erik Ernst

Aarhus University

A Straight-line Programming Language

- Consider a tiny language (no loops)
- Skip the lexing and parsing phases
- Allows focus on “meaning” – interpretation
- Syntax:

Stm	→	***Stm*** ; ***Stm***	(CompoundStm)	***ExpList***	→	***Exp*** , ***ExpList***	(ExpList)
Stm	→	id := ***Exp***	(AssignStm)	***ExpList***	→	***Exp***	(ExpList)
Stm	→	print (***ExpList***)	(PrintStm)	***Binop***	→	+	(Plus)
Exp	→	id	(IdExp)	***Binop***	→	–	(Minus)
Exp	→	num	(NumExp)	***Binop***	→	×	(Times)
Exp	→	***Exp Binop Exp***	(OpExp)	***Binop***	→	/	(Div)
Exp	→	(***Stm*** , ***Exp***)	(EseqExp)				

Q/A

- How do you know this cannot be parsed?

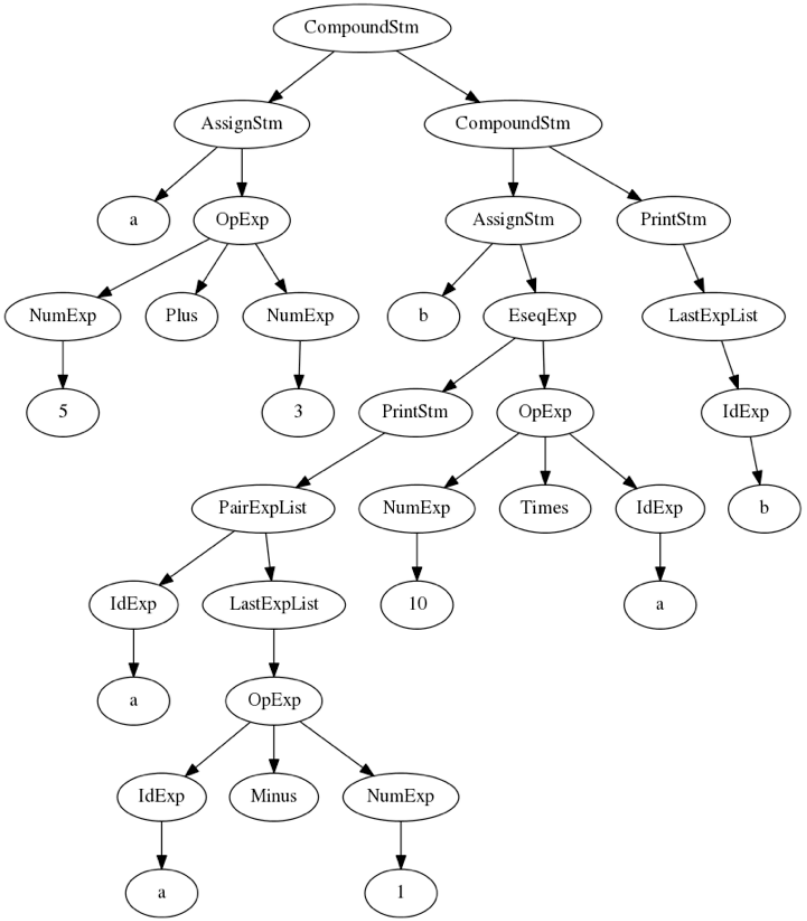
Stm	→	***Stm*** ; ***Stm***	(CompoundStm)	***ExpList***	→	***Exp*** , ***ExpList***	(ExpList)
Stm	→	id := ***Exp***	(AssignStm)	***ExpList***	→	***Exp***	(ExpList)
Stm	→	print(***ExpList***)	(PrintStm)	***Binop***	→	+	(Plus)
Exp	→	id	(IdExp)	***Binop***	→	–	(Minus)
Exp	→	num	(NumExp)	***Binop***	→	×	(Times)
Exp	→	***Exp Binop Exp***	(OpExp)	***Binop***	→	/	(Div)
Exp	→	(***Stm*** , ***Exp***)	(EseqExp)				

A Straight-line Program

- Source:

```
a := 5+3;
b := ( print(a,a-1)
     , 10*a);
print(b)
```

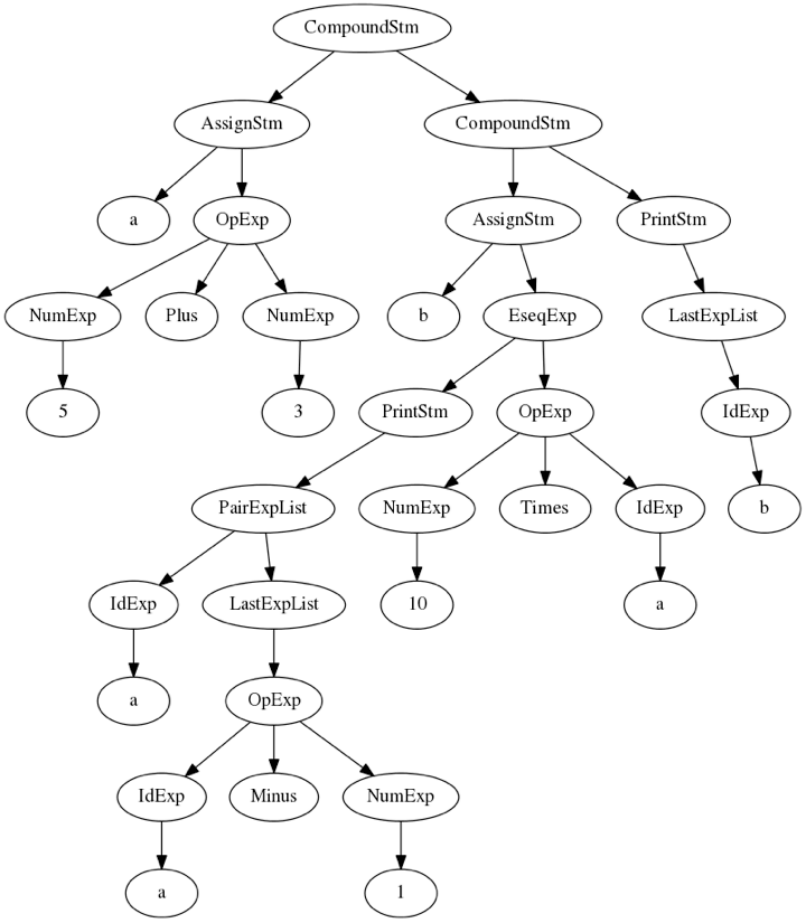
- Corresponding syntax tree:



An SLP syntax representation datatype

- SML declaration:

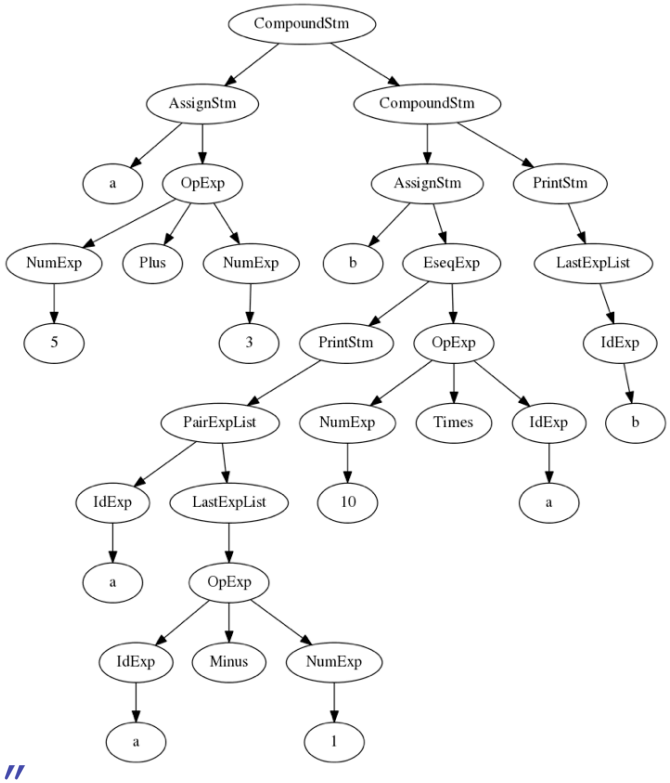
```
type id = string
datatype binop =
  Plus|Minus|Times|Div
datatype stm
  = CompoundStm of stm*stm
  | AssignStm of id*exp
  | PrintStm of exp list
and exp
  = IdExp of id
  | NumExp of int
  | OpExp of exp*binop*exp
  | EseqExp of stm*exp
```



An SLP syntax representation

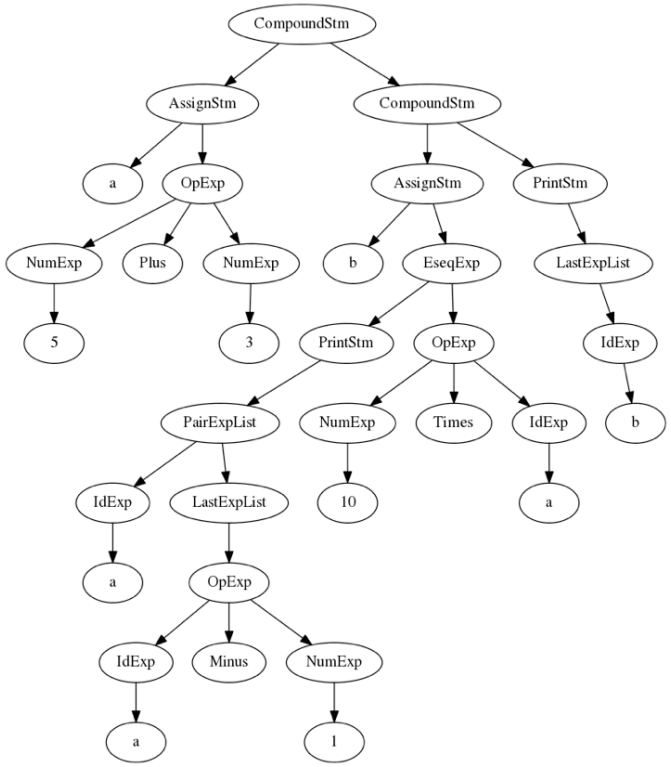
- SML value:

```
val prog =
  CompoundStm(
    AssignStm("a",
      OpExp( NumExp 5
           , Plus
           , Numexp 3)),
    CompoundStm(
      AssignStm("b",
        EseqExp( PrintStm [ IdExp "a"
                          , OpExp(..)]
               , OpExp(NumExp 10, ..))
      PrintStm [ IdExp "b" ]))
```



Project Assignment

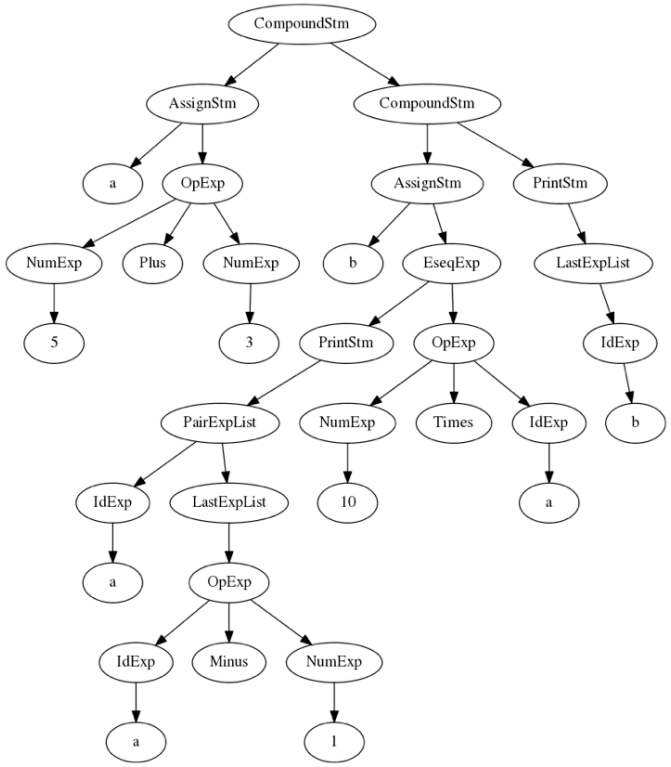
- Follow description p10-12
- “Modularity principles” p9-10: discussed on Friday, may be ignored at first
- General principle: inspect datatype, write function:




```
datatype bools = End
   | NotEnd of bool*bools
fun hasTrue End = false
   | hasTrue (NotEnd (b, bs)) =
     b orelse hasTrue bs
```

Project Assignment

- Follow description p10-12
- “Modularity principles” p9-10: discussed on Friday, may be ignored at first
- General principle: inspect datatype, write function:




```
datatype bools = End
   | NotEnd of bool*bools
fun hasTrue End = false
   | hasTrue (NotEnd (b, bs)) =
     b orelse hasTrue bs
```

Lexical Analysis

Lexical Analysis

- Grammars could describe every char (some do!)
- Messy: whitespace, details, uglifies grammar
- Use simpler tool for “words”: Regular expressions
- *Lexer (scanner)* reads text, delivers *tokens* to parser

- Note increasing power:
  - Lexical analysis ~ DFA, regular languages
  - Parsing ~ PDA, context-free languages
  - Type checking ~ Turing machines, general computation

Tokens

- *Tokens* are words and data, e.g.:

Type	Examples
ID	`foo n14 a' my-fun`
NUM	`73 0 070 0L 0x42`
REAL	`66.1 .5 10. 1.1e-67`
IF	`if If iF IF`
COMMA	`,`
NOTEQ	`!= ! =`
LPAREN	`(`
RPAREN	`)`

Non-tokens

- Finding tokens, the lexer skips certain things:

Type	Examples
comments	`/* my-fun, dead */`
	`// easy`
	`(* nest(*ing*) *)`
	`# script style`
preprocessor directives	`#include <stdio.h>`
	`#define MAX 5`
whitespace	

Token data structure

- Many tokens need no associated data, e.g.:

  IF, COMMA, NOTEQ, LPAREN, RPAREN

- Some tokens must carry a string, e.g.:

  ID("my-fun")

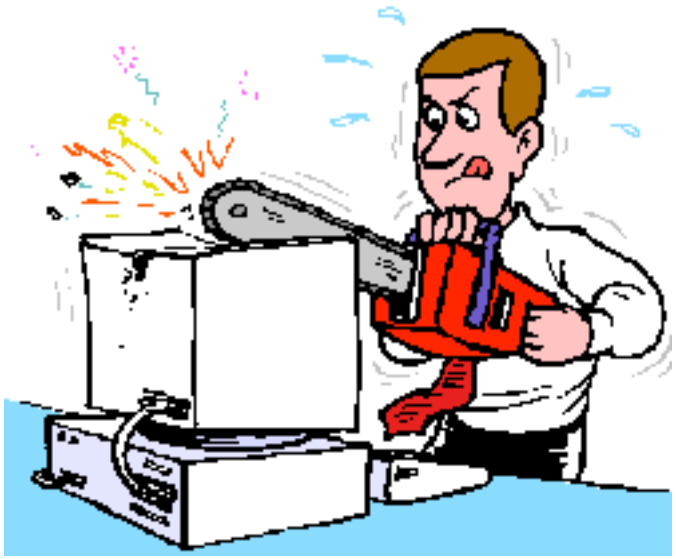
- Some tokens could carry some other type, e.g.:

  NUM(73), NUM(0), NUM(.7), NUM(IEEE754,01000111101011...)

- Useful additional information: Start/end position of input (line number + column, or charpos)

Q/A

- Consider "`For (int æ; æ ≤ 0; æ--) {}`"
- Language: case insensitive, ASCII
- How do you report the error of using "**æ**"?



Regular expressions

- Expected to be well-known
- Syntax:
  - Symbol `a`
  - choice `x|y`
  - concat `xy`
  - empty `ϵ`
  - repeat `x*`
- Each RegExp corresponds to an NFA, transformable to DFA
- Resulting table driven execution: Linear complexity

Regular expressions used for scanning

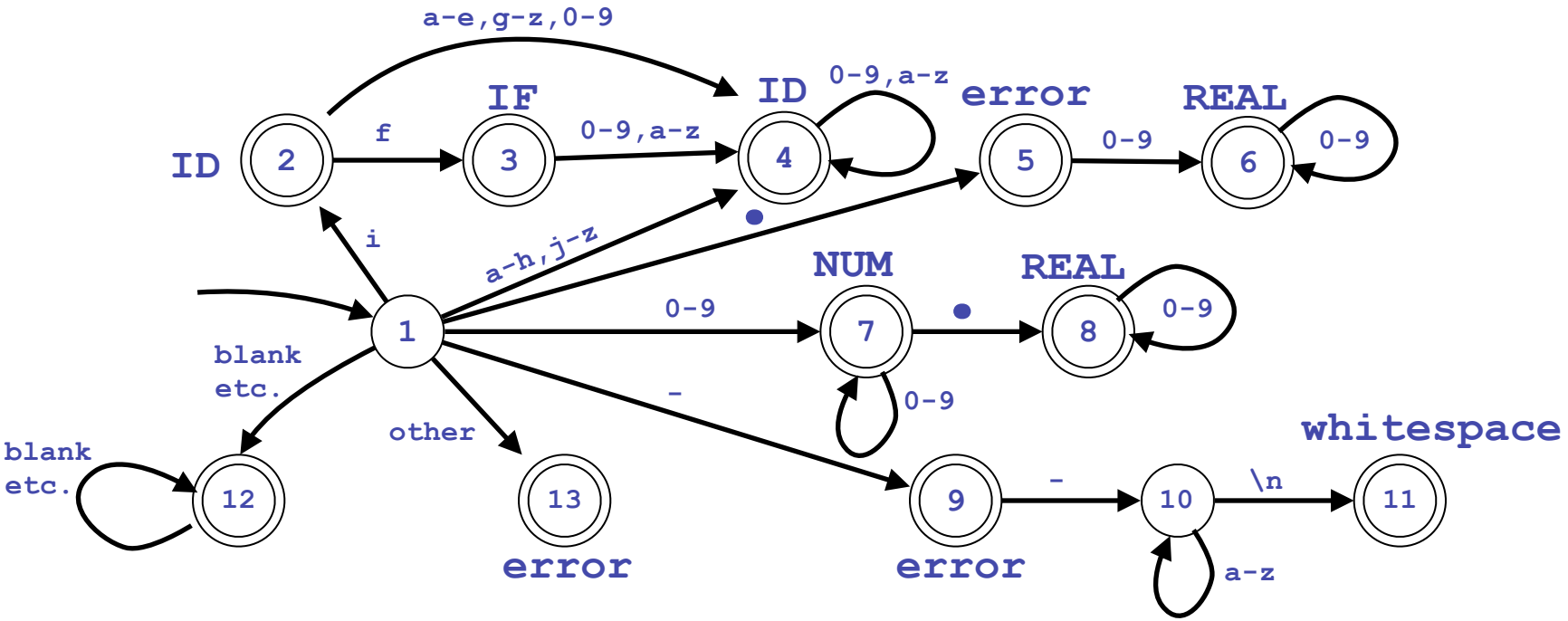
- Examples:
  - `if` `(IF);`
  - `[a-z][a-z0-9]*` `(ID);`
  - `[0-9]+` `(NUM);`
  - `([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)` `(REAL);`
  - `("--"[a-z]*"\n")|(" "|"\n"|"\t")` `(continue());`
  - `.` `(error(); continue());`

- Explain, please!
- Where are the comments? What's wrong?
- What does '`continue()`' do?

Finding the longest match

- The lexer is specified by an ordered sequence of regular expressions for the tokens: $r_1, r_2, \ldots, r_k$
- Let t_i be the longest prefix of the input string that is recognized by r_i for each i
- Let $k = max\{ |t_i| \}$
- Let $j = min\{ i \mid |t_i| = k \}$
- The next token is then t_j, matched by r_j

Total NFA for ID,IF,NUM,REAL



ML-Lex

- Lexer generator, “built-in” part of SML/NJ
- Accepts lexical specification, produces scanner
- Example specification:

```
(* SML declarations *)
type lexresult = Tokens.token
fun eof() = Tokens.EOF(0,0)
%%
(* Lex definitions *)
digits=[0-9]+
%%
(* Regular Expressions and Actions *)
if                    => (Tokens.IF(yypos,yypos+2));
[a-z][a-z0-9]*        => (Tokens.ID(yytext,yypos,yypos + size yytext));
{digits}              => (Tokens.NUM( Int.fromString yytext
                                    , yypos, yypos + size yytext);
({digits}"."[0-9]*)|([0-9]*"."{digits})
                      => (Tokens.REAL( Real.fromString yytext
                                     , yypos, yypos + size yytext));
("--"[a-z]*"\n")|(" "|"\n"|"\t")+
                      => (continue());
.                     => ( ErrorMsg.error yypos "Illegal character"
                         ; continue());
```

Lexer states

- Helpful when handling different “kinds” of tokens
- E.g., use state ..
  - INITIAL in general (automatic)
  - STRING when scanning the contents of a string
  - COMMENT when scanning a comment
- Point: Keep different concerns apart — simpler!
- Syntax:

```
...
(* Regular Expressions and Actions *)
<INITIAL>if              => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z][a-z0-9]* => (Tokens.ID(yytext,yypos,yypos + size yytext));
...
<INITIAL>"\""            => (YYBEGIN STRING; continue());
...
<STRING>.                => (continue());
...
```

Summary

- Warm-up project: Program in SML!
  - Straight-line programming language, no lexer/parser
  - Express programs: Use abstract syntax tree datatype
  - Project specified on website, essentially as in book
- Lexical analysis
  - Avoid complexity in grammar: Use lexer
  - Based on regular expressions, impl. via NFA/DFA
  - Theory assumed known
- Tools: ML-Lex
  - Scanner generator, outputs SML code from spec
  - Note lexer states