Glorious SPL Compiler

By Ischa Stork & Ward Theunisse

iCIS | Software Science
Radboud University

Presentation overview

1. Compiler choices
2. Lexer
3. Parser
4. Questions

Compiler choices

- Built in Python
- Top-down parser
- Leftmost derivation
- Non-ambiguous grammar
- Convergent left recursion
- LL(0) - 0 token lookahead

Lexer

- Lazy lexer: tokens are produced when they are needed
- 2 bit state (preceding whitespace or comment), which decides which tokens are matched.
- Greedy regex match on prefix;

```
REG_STR = re.compile("\"([^\0\a\b\f\n\r\t\v\\\\\'\"]|\\\\[0abfnrtv\\\\\"\'])+\"")
```

- We also support user defined operators.
- For example a--b, which is lexed as <a><--><b> by regex.

```
OpDecl =
'prefix' op '(' id ')' [':'  FunType] '{' VarDecl* Stmt+ '}'
| ('infixl' | 'infixr') int op '(' id ',' id ')' [':'  FunType] '{'
VarDecl* Stmt+ '}'
```

Lexer - Custom operators

- Syntax operators (::, ',', => and =) are lexed in the same way as normal operators;
- Operators have the following syntax:

```
Exp = ConvExp (op ConvExp)*
op = opChar+
opChar = '!'  | '#' | '$' | '%' | '&' | '*' | '+' | '/' | '<' | '='
| '>' | '?'  | '@' | '\', '^', '|', '-', ' ', ',', ':'
```

- The result is that we do not know the precedence and associativity of the operators, resulting in the following grammar rule for expressions:

```
Exp = ConvExp (op ConvExp)*
ConvExp = id Field | op Exp | int | char | string | 'False' | 'True'
| '(' Exp ')' | FunCall | []
```

Lexer - Type synonyms

Extension: Type synonyms (TypeSyn)

```
TypeSyn = 'type' type_id '=' Type
```

- Type_id - uppercase
- Normal_id - lowercase
- Char, Bool, Int also lexed this way

Parser

- The mapping between tokens and AST syntax constructs is non-injective.
- Implicit recursion through stack of active rules.
- Dual pass parse for expressions.
- Whenever there is a choice in rules, the next token is conclusive as a result of our language definition.

Questions