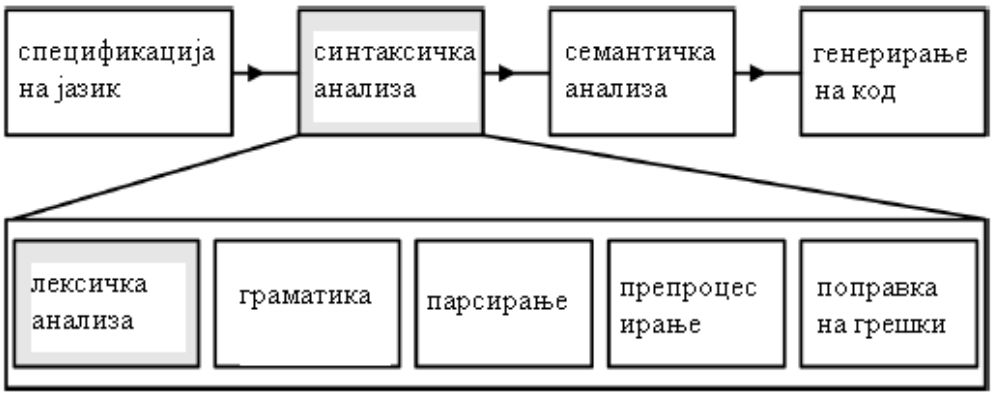
Синатаксичка анализа

Вовед

- Луѓето ги разбираат реченициte кои се искажани на говорен јазик, затоа што нивниот мозок е истрениран да ја разберат нивната смисла
- Ова е можно само ако реченициte следат некои граматички правила (синтакса)
- Граматичките правила и како се користат за да се парсира
- Визуелизација на реченициte преку синтаксно дрво
- Практичната имплементација
  - Делење налинија од текстот на индивидуални зборови
  - Распознавање на клучните зборови
  - Преминување по дрвото

Синтаксичка анализа



Лексер

- Првиот чекор во процесот на компајлирање е да се прочита изворниот код
  - Се проверува дали има грешки
  - Се бараат и детерминираат клучните (резервираните) зборови како на пример IF, WHILE, SWITCH и други – атомски зборови
  - Во некои случаи треба да се издвојат одредени индивидуални карактери како некој integer број (пр. 12345), стринг (пр. "hello, world") реален број пр. 12e-09) кои исто така би се гледале како зборови, но индивидуалните знаци од кои се изградени им даваат посебно значење
  - This distinction requires special processing of the input text, and this special
- Процесот на нивно одвојување бара посебно процесирање на внесениот текст, и обично се одвојува од парсерот и се става во посебен модул кој се нарекува ЛЕКСЕР

Лексер

- Лексерот треба да направи одделување на влезната низа во жетони (атомски зборови).
  - Овој процес се нарекува жетонизирање или скенирање
- Парсерот е модул кој се занимава со групирање на жетоните,
  - побарува жетони од лексерот, кој пак треба да прочита доволно знаци од влезниот текст за да направи еден жетон,
  - Таквите жетони се даваат на парсерот
  - Парсерот проверува дали тие се правилно наредени
- Пр. Брзата кафеава лисица го прерипна мрзливиот пес
  - Лексерот реченицата ја дели на: Брзата, кафеава, лисица, го, прерипна, мрзливиот, пес

Лексер

- Лексерот исто така ги дели жетоните по класи. Овој процес се вика прегледување.
- Пример: Сумата на 2 + 2 = 4.
- Лексерот ќе ја подели во класите:
  - збор: сумата
  - Збор: на
  - број: 2
  - плус: +
  - број: 2
  - еднакво: =
  - број: 4
  - точка: .

Лексер

- Лексерот треба да знае како да ги подели жетоните
  - Со бело место (таб, празно место нова линија)- меѓу зборови
  - Во формули, најцесто не се користи бело место
- Пр. Влез: sum=(2+2)*3;
- Се дели на: sum, =, (, 2, +, 2,), *, 3 и;

Лексер

- Лексерот исто така треба да ги исфрли деловите од програмата кои не се интересни за парсерот
  - Пример коментари
- Една класа жетони може да репрезентира голем спектар на вредности
  - OP_MULTIPLY го содржи само операторот *
  - LITERAL_INTEGER ги содржи сите броеви од тип integer
- Компајлерот не го интересира само дека жетонот е literal_integer, туку и неговата вредност.
  - Секогаш жетоните се придружени со нивната вредност
  - Вредноста треба да соодветствува со типот на жетонот Lexical
- Лексерите често ги зачувуваат вредностите на жетоните користејќи унија

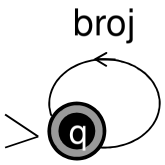
Поврзаност на лексер и парсер

- Лексерот претставува подмодул на парсерот
- Парсерот работи со контекстно слободни граматики, а лексерот со регуларни граматики
  - Едноставни жетони (IF или WHILE)
  - Комплексни жетони (INTEGER, кои покриваат голем број на случаи-кои се откриваат со помош на конечни автомати

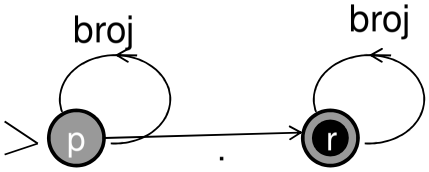
Поврзаност на лексер и парсер

- Прво почнува да работи парсерот
  - Парсерот од лексерот побарува да му идентифицира нареден жетон и да му прати во која класа спага истиот
- Лексерот чита од низата почнувајќи од претходното место каде што застанал
  - Користи конечен автомат кој е всушност е унија од сите конечни автомати за жетоните
  - Јаде знак по знак се додека може тоа да го направи, т.е. додека не стигне во состојба “црна дупка”
  - Ако претходно бил во некоја крајна состојба соодветна за некој валиден жетон, тоа го праќа на парсерот, ако не јавува грешка
  - Го памети местото на претпоследниот прочитан стринг, и од таму почнува да чита кога наредeн пат ќе биде повикан од парсерот

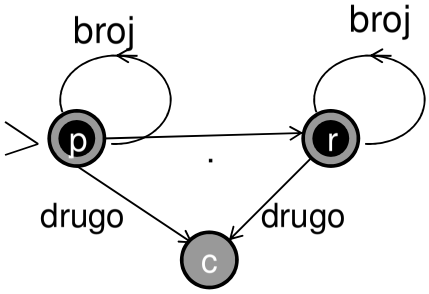
Пример



Integer



real



Пр.

(p,324+2...)-(p,24+2...) - (p, 4+2...) –(p,+2...)- (c,2...)

Дава на парсер integer

Памети позиција 4

Пр.

(p,3.2+2...)-(p,.3+2...) - (r, 3+2...) –(r,+2...)- (c,2...)

Дава на парсер real

Памети позиција 4

Поврзаност на лексер и парсер

- Парсерот го зема наредниот жетон и го јаде
- Имајќи ја во предвид граматиката врз која е изграден парсерот се обидува да го намали изразот во стекот се додека тоа е можно
- Кога веќе парсерот не може да направи некоја друга акција, побарува нов жетон од лексерот

Видови на лексери

- Индиректен
- Директен

Индиректен лексер

- Во индиректниот лексер се очекува да се определи стринг кој е подстринг на друг стринг
  - Резервираниот збор for и идентификатор forma
- При ова може да се направи грешка при компајлирањето
  - Решение е ако направи грешка, некаде понатаму нема да може да се испарсира програмата, па треба да се врати назад

Индиректен лексер

- Може да се направи и добар автомат, со користење на разлика на регуларни јазици
  - Потешка постапка
  - Пр. (<идентификатор>=<буква>{<буква>}-{for, if, start,….})
- Автоматот има повеќе завршни состојби, соодветни за секој тип на жетон

Директен лексер

- Директниот лексер пребарува по повеќе автомати, автомат за секој жетон
- Убрзување може да се направи ако се пребарува паралелно
- Бара во кој автомат ќе го препознае стрингот
- Повеќето автомати може да се комбинираат во еден автомат

Директен лексер

- Проблем со идентификатори и клучни зборови
  - Да се даде предност на празно место
- Ние ја креираме граматиката
  - Може да се ограничи секогаш да се ставаат празни места по секој жетон
  - На тој начин секогаш празното место ќе значи крај на жетонот, и веднаш можеме да видиме во која состојба се наоѓаме

Отстранување на непотребни знаци

- Лексерот треба да отстрани непотребни знаци (коментари) или повеќе празни места
- Лексерот на парсерот жетоните ги дава без празно место
- Коментарите се изоставуваат и не се даваат на парсерот

Отстранување на непотребни знаци

- Вишок празни места:
  - Почнува од еден изеден знак наназад (празните места се лоцираат пред жетон)
  - Останува во почетна состојба се додека има празен збор
- Коментар низ повеќе линии
  - Се лоцира пред жетон
  - Кога се отвара коментарот се преминува во состојба “otvoren komentar” со секој нареден знак се останува во таа состојба, кога се затвара се враќа во почетна состојба
- Коментар во една линија
  - Се лоцира пред жетон
  - Кога се отвара коментарот се преминува во состојба “red komentar” со секој нареден знак се останува во таа состојба, кога се доаѓа до ENTER се враќа во почетна состојба

Работа на лексерот

- Пример: како (34+12) се дели на жетони.
- Нека јазикот се состои само од реченици од тип (број + број).
- Регуларни изрази за жетони

Token	Regular expression
(	(
)	)
+	+
number	[0-9]+

Работа на лексерот

- Лексерот ги користи состојбите за да определи кои знаци да ги очекува а кои не
- За едноставните жетони (,) и + е лесно, или се чита еден знак или не
- За жетонот <број> се потребни повеќе состојби
  - Кога ќе се прочита една цифра, лексерот влегува во состојба во која очекува една или повеќе цифри, и ништо друго.
  - Ако прочита цифра, останува во состојбата и ја додава цифратана бројот
  - Ако се прочита нешто друго, се напушта состојбата и го дава жетонот на парсерот

Работа на лексерот (34+12)

- Лексерот почнува во почетната состојба

Проч. жетон	состојба	дејство
(	почетна	Враќа (на парсерот
3	почетна	зачувува 3, влегува во состојба број
4	број	зачувува 4
+	број	+ неочекувано. Ја напушта состојбата број и враќа 34 на парсерот
+	почетна	Враќа + на парсерот
1	почетна	зачувува 1, влегува во состојба број
2	број	зачувува 2
)	број	) неочекувано. Ја напушта состојбата број и враќа 12 на парсерот
)	почетна	Враќа) на парсерот

Слични регуларни изрази

- Некои регуларни изрази за жетоните се слични или се подзбор еден од друг (гледани како регуларен израз)
- Самите жетони се подзбор еден на друг
  - Пример = и ==
  - Се решаваат со стратегија да се зема знак се додека не се стигне до неочекуван знак

Integer броеви

- Се состојат само од цифри
- Завршуваат кога ќе се прочита знак кој не е цифра
  - За некои компајлери предолгите броеви генерираат грешка *прекорачување на големина*. (може да се испечати ваква грешка и да се продолжи со парсирање)
- Регуларниот израз за integer е

[0-9]+

Реални броеви

- Покомплексна структура од integer броевите.
  - 1.0, .001, 1e-09, .2e+5
- Регуларниот израз за нив е:

  $[0\text{-}9]^* \; . \; [0\text{-}9]^+ \; (\; e \; [+|-] \; [0\text{-}9]^+ \;)$
- Треба да се провери секој дел за да не има пречекорување во должината на броевите.

Практична забелешка-долг регуларен израз

- Ако регуларниот израз е предолг, може да се подели на повеќе машини
- Лексерот ќе ги анализира внатрешните конечни автомати и ќе работи без проблем

Стрингови

- Бараат посебна анализа од лексерот
  - Пример. "3+4“
- Решение: се додава нова состојба наречена *ексклузивна состојба.* Кога лексерот е во оваа состојба ќе процесира само регуларни изрази кои се обележани со оваа состојба

Регуларен израз	Дејство
“	Влегува во состојба *string*.
string .	Го зачувува знакот. (.) значи било што. Овој регуларен израз се разгледува само кога лексерот е во состојба *string*.
string "	се враќа претходната состојба. Се дава стрингот на парсерот. Овој регуларен израз се разгледува само кога лексерот е во состојба *string*.

Стрингови

- Вака направена граматика за стринг не проверува дали еден стринг се протега во повеќе редови
- Не се води сметка дали ќе се преполни баферот-ова не се проверува од страна на компајлерот
- Може да дојде до преполнување на меморија што би јавило грешка
- Може да се дели стрингот и потоа да се спојуваат
- За да се избегне преполнување на баферот може да се лимитира големината (во меморија) и ако се пречекори таа да се јави таков тип на грешка

Коментари

- Во повеќето компајлери филтрирањето на коментарите се прави во лексерот
- Почетокот и крајот на коментарите јасно се маркира

Јазик	Стил на коментарComment style
C	/* коментар */
C++	// коментар (во една линија)
Pascal	{коментар }
BASIC	REM коментар :

Коментари

- Во повеќе редови

Регуларен израз	ДејствоRegular expression Action
/*	Влегување во состојба за коментар.
comment .	Се игнорира знакот. (.) значи било што Овој регуларен израз се разгледува само кога лексерот е во состојба *comment* .
comment */	се враќа претходната состојба. Не се дава ништо на парсерот. Овој регуларен израз се разгледува само кога лексерот е во состојба *comment* .

- Со мали модификации може да направиме да може да се вметне коментар. Пример со регуларен израз /* (.)* */
- За линиски коментари: //(.)*\n

Генератори на лексери

- Лексерот може да се напише на рака, т.е. Во било кој јазик
  - Ова е тешка задача ако јазикот станува побогат
- Користење на генератор за лексер
  - Кодот е побрз и пократок од било кој код кој може да се напише на рака

Генератори на лексери

- Кандидати за генератори:
- AT&T lex
  - Не е слободен, постар, има имплементација за UNIX и Linux
- GNU flex
  - Слободен, модерен, имплементација за Linux
- Bumblebee lex,
  - слободен, модерен, имплементација за Windows
- Компајлерот за Inger е конструиран со користење на GNU flex;

Синтакса на flex

- Распоредот на влезовите во flex (екестензија .1) во псевдокод е:

```
%{
        Било кој пред код во C (вклучувања,  дефинирања) кои би
        се залепиле на крајниот .C фајл
%}
   Било кои дефиниции во flex
   %%
   Регуларни изрази
   %%
   Било кој код во C кој ќе се додаде на крајниот .C фајл
```

Синтакса на flex

- Кога некој регуларен израз ќе се поклопи се некој влезен текст, лексерот ќе изврши дејство.
  - Го информира парсерот (повикувачот) дека е најдена класа жетони
- Регуларен израз со вклучена акција ја има следната форма:

```
[0-9]+ {
                intValue_g = atoi( yytext );
                return( INTEGER );
                }
```

  - Со return(INTEGER), лексерот го информира парсерот дека е најден integer. Тој можеже да врати само еден случај (класата жетони) така да вистинската вредност на integer-от поминува низ парсерот преку глбалната променлива intValue_g.
  - Flex автоматски го зачувува знакот за тековниот жетон во глобалниот стринг yytext.

Пример за влезен фајл во flex

- (broj+broj) и дозволува празно место (свен во жетоните)

```
%{
    #define NUMBER 1000
    int intValue_g;
%}
%%
"(" { return( `(` ); }
")" { return( `)' ); }
"+" { return( `+' ); }
[0-9]+ {
            intValue_g = atoi( yytext );
            return( NUMBER );
            }
%%
int main()
{
        int result;
        while( ( result = yylex() ) != 0 )
        {
        printf( "Token class found:
%d\n", result );
}
return( 0 );
}
```

Спецификации за лексерот на Inger

- Inger разликува неколку категории:
  - Резервирани зборови (IF, WHILE ...),
  - оператори (+, % и др.),
  - Комплексни жетони (integer numbers, floating point numbers, и strings),
  - разделувачи (средни и мали загради) и празни места.

Клучни или резервирани зборови

- Inger очекува сите клучни зборови да бидат напишани со мали букви

Token	Regular Expression	Token identifier
break	break	KW_BREAK
case	case	KW_CASE
continue	continue	KW_CONTINUE
default	default	KW_DEFAULT
do	do	KW_DO
else	else	KW_ELSE
false	false	KW_FALSE
goto_considered _harmful	goto_considered _harmful	KW_GOTO
if	if	KW_IF
label	label	KW_LABEL
module	module	KW_MODULE
return	return	KW_RETURN
start	start	KW_START
switch	switch	KW_SWITCH
true	true	KW_TRUE
while	while	KW_WHILE

Типови

- Имињата на типовите се непромеливи жетони и се спаруваат со нивното полно име

Token	Regular Expression	Token identifier
bool	bool	KW_BOOL
char	char	KW_CHAR
float	float	KW_FLOAT
int	int	KW_INT
untyped	untyped	KW_UNTYPED

Комплексни жетони

- Комплексни жетони во Inger променливите идентификатори, целобројните литералите, реалните литерали и знаците литерали

Token	Regular Expression	Token identifier
integer literal	`[0-9]+`	INT
identifier	`[_A-Za-z][_A-Za-z0-9]*`	IDENTIFIER
float	`[0-9]*\.[0-9]+([eE][\+-][0-9]+)?`	FLOAT
char	`\'.\'`	CHAR

Стрингови

- Стринговите во Inger не се протегаат на повеќе линии.

```
\"                      { BEGIN STATE_STRING; }
<STATE_STRING>\"        { BEGIN 0; return( STRING ); }
<STATE_STRING>\n        { ERROR( "unterminated string" ); }
<STATE_STRING>.         { (store a character) }
<STATE_STRING>\\\"      { (add " to string) }
```

- Ако се појави карактерот за нов ред, лексерот вади грешка.
- Секој знак кој се чита додека лексерот е во состојба за читање стринг, се додава на стрингот, освен ", со кој стрингот завршува и му јавува на лексерот да излезе од ексклузивната состојба за читање на стринг.
- Со користење на контролниот код \“, програмерот може да додаде " во стрингот.

Коментари

- Inger подржува два типа на коментари
  - Линиски коментари-кои завршуваат со карактер за нов ред
  - Блок коментари кои експлицитно треба да бидат завршени
- Линиските коментари може да се претстават со следниов регуларен израз:

  "//"[^\n]*

Коментари

- Блок коментарите бараат во лексерот да се воведе нова ексклузивна состојба.
- Пр.

```
 /*                          { BEGIN STATE_COMMENTS; ++commentlevel; }
<STATE_COMMENTS>"/*“         { ++commentlevel; }
<STATE_COMMENTS>.            { }
<STATE_COMMENTS>\n           { }
<STATE_COMMENTS>"*/"         { if( --commentlevel == 0 ) BEGIN 0; }
```

- Кога еднаш коментарот ќе пошне со /*, лексерот го подесува нивото на коментар на 1 и влегува во состојба на коментар.
- Нивото накоментар се зголемува секогаш кога се наидува на /* а намалува кога се чита */
- Додека е во состојба на коментар сите знаци се игнорираат
- Состојбата се напушта кога ќе заврши последниот блок коментари.

Оператори

- Во Inger се појавуваат голем број на оператори со различен приоритет.
- Само атомските (не функции или оператори со низи)
- оператори кои се состојат од повеќе знаци.
  - Се решава со гледање еден знак наназад

Оператори

Token	Regular Expression	Token identifier
addition	+	OP_ADD
assignment	=	OP_ASSIGN
bitwise and	&	OP_BITWISE_AND
bitwise complement	~	OP_BITWISE_COMPLEMENT
bitwise left shift	<<	OP_BITWISE_LSHIFT
bitwise or	|	OP_BITWISE_OR
bitwise right shift	>>	OP_BITWISE_RSHIFT
bitwise xor	^	OP_BITWISE_XOR
division	/	OP_DIVIDE
equality	==	OP_EQUAL
greater than	>	OP_GREATER
greater or equal	>=	OP_GREATEREQUAL
less than	<	OP_LESS
less or equal	<=	OP_LESSEQUAL
logical and	&&	OP_LOGICAL_AND
logical or	||	OP_LOGICAL_OR
modulus	%	OP_MODULUS
multiplication	*	OP_MULTIPLY
logical negation	!	OP_NOT
inequality	!=	OP_NOTEQUAL
subtract	-	OP_SUBTRACT
ternary if	?	OP_TERNARY_IF

Одделувачи

Листа на одделувачи

Token	Regexp	Token identifier
precedes function return type	->	ARROW
start code block	{	LBRACE
end code block	}	RBRACE
begin array index	[	LBRACKET
end array index	]	RBRACKET
start function parameter list	:	COLON
function argument separation	,	COMMA
expression priority, function application	(	LPAREN
expression priority, function application	)	RPAREN
statement terminator	;	SEMICOLON

The full source to the Inger lexical analyzer is included in appendix F.