پروژه درس کامپایلر

توضیحات الزامی پروژه درس کامپایلر

در این پروژه، زبان برنامه‌نویسی توسط استاد درس مشخص و ابلاغ شده است. دانشجویان موظف‌اند برای این زبان، کلیه فازهای اصلی یک کامپایلر را طراحی و پیاده‌سازی نمایند. هدف پروژه، نمایش صحیح و مستند فرآیند کامپایل است و صرفاً اجرای برنامه نهایی کافی نیست

۱. تحلیلگر لغوی (Lexical Analyzer)

تحلیلگر لغوی باید کد منبع را دریافت کرده و آن را به دنباله‌ای از توکن‌ها تبدیل نماید.

ابزار پیشنهادی:

- Flex / Lex
- یا پیاده‌سازی دستی با زبان‌هایی مانند Python، Java یا C/C++

خروجی الزامی:

فایل یا خروجی مستقل شامل لیست توکن‌ها به شکل <TokenType, Lexeme>.

در صورت درخواست استاد، دانشجو باید صرفاً خروجی تحلیلگر لغوی را ارائه دهد.

۲. تحلیلگر نحوی (Parser)

تحلیلگر نحوی، توکن‌های تولیدشده توسط تحلیلگر لغوی را بر اساس گرامر زبان ابلاغی تحلیل می‌کند و صحت ساختاری برنامه را بررسی می‌نماید.

ابزار پیشنهادی:

- Yacc / Bison
- یا پیاده‌سازی دستی(Recursive Descent) ، LL، LRو(...

خروجی الزامی:

- Parse Treeیا
- Abstract Syntax Tree (AST)

در صورت درخواست استاد، ارائه خروجی مستقل تحلیلگر نحوی الزامی است.

۳. تحلیلگر معنایی (Semantic Analyzer)

در این مرحله، درستی معنایی برنامه بررسی می‌شود.

موارد مورد بررسی:

- جلوگیری از استفاده از متغیر تعریف‌نشده

- جلوگیری از تعریف تکراری
- بررسی تطابق نوع‌ها
- صحت انتساب‌ها و اعمال

خروجی الزامی:

- جدول نمادها ((Symbol Table
- گزارش خطاهای معنایی (در صورت وجود)

ارائه خروجی این بخش باید مستقل از سایر فازها امکان‌پذیر باشد.

۴. تولید کد میانی (Intermediate Code Generation)

دانشجو موظف است خروجی تحلیل معنایی را به کد میانی مستقل از ماشین تبدیل کند.

نوع کد میانی:

- Three Address Code (TAC)

خروجی الزامی:

فایل یا لیست دستورات کد میانی، شامل متغیرهای موقت و برچسب‌ها.

۵. بهینه‌سازی کد میانی (Optimization)

حداقل یک روش ساده بهینه‌سازی باید روی کد میانی اعمال شود.

نمونه بهینه‌سازی‌ها:

- Constant Folding
- Dead Code Elimination
- Common Subexpression Elimination

خروجی الزامی:

نمایش کد میانی قبل و بعد از بهینه‌سازی.

۶. خروجی‌های قابل درخواست توسط استاد

دانشجو موظف است در هر زمان، بنا به درخواست استاد، هر یک از خروجی‌های زیر را به‌صورت مجزا ارائه دهد:

- خروجی تحلیلگر لغوی

- خروجی تحلیلگر نحوی (Parse Tree یا AST)
- جدول نمادها
- گزارش خطاهای معنایی
- کد میانی
- کد میانی بهینه‌شده

نکته پایانی

هدف این پروژه، درک و پیاده‌سازی فرایند کامل کامپایلر است؛
نه صرفاً تولید خروجی نهایی یا اجرای برنامه.

پروژه:

اعداد صحیح، اعداد حقیقی، بولین و تعریف متغیرها

اعداد صحیح: شامل رشته‌هایی از یک یا بیشتر از بین کاراکترهای 0-9 می‌باشند. این اعداد شامل اعداد ۳۲ بیتی در قالب مکمل ۲ و یا ۶۴ بیتی، بازهم در سیستم مکمل ۲ می‌باشد. (درک اعداد بزرگ‌تر لازم نیست.)

همچنین اعداد صحیح می‌توانند به صورت Hex نیز بکار روند، مثال: 0x230 = 35

اعداد حقیقی به چهار صورت قابل پیاده‌سازی هستند:

۱- رشته‌ای از ارقام به صورتی که سمت چپ کاراکتر "." شامل یک یا بیشتر رقم باشد مثال: 1.

۲- رشته‌ای از ارقام به صورتی که سمت راست کاراکتر "." شامل یک یا بیشتر رقم باشد مثال: 1.

۳- رشته‌ای از ارقام به صورتی که سمت چپ کاراکتر "." شامل یک یا بیشتر رقم باشد مثال:1.1

توجه کنید که سمت چپ هر یک از ثوابت عددی می‌تواند علامت منفی باشد.

همچنین دو ثابت Boolean از نوع bool، true و false نیز در زبان گنجانده شده‌اند.

متغیرهای رشته‌ای (Strings) می‌توانند شامل همه کاراکترهای ASCII بجز کاراکتر EOF باشند. حداکثر اندازه یک رشته ۶۰۰۰ کاراکتر است. البته باید این میزان به صورت پویا تخصیص داده شود (منظور حافظه پویا تخصیص به اندازه مورد نیاز است، یعنی نباید به هر رشته اندازه ثابت ۶۰۰۰ تخصیص داده شود.)

متغیرها شامل رشته‌هایی از کاراکترهای حروف الفبای زبان انگلیسی، اعداد و کاراکتر "_" (Underscore) می‌باشد، همچنین هیچ متغیری با عدد شروع نمی‌شود.

ثوابت رشته‌ای، کامنت‌ها، و دیگر ملاحظات

ثوابت رشته‌ای با کاراکتر {"}آغاز شده و به همین کاراکتر نیز ختم می‌شوند.

همه قواعد زبان **C++** در مورد ثوابت رشته ایی (کاراکترهای درونی آنها برقرار می‌باشند.) این مورد در رابطه با کاراکتری نیز برقرار است به جز اینکه کاراکتر آغازین/پایانی آن {"}می‌باشد. مثال از رشته:

"this is a\t valid String"... \n"

کامنت‌های تک‌خطی با علامت ## شروع می‌شوند، همچنین کامنت‌های چندخطی با #/ شروع شده و با /# خاتمه می‌یابند. مثال:

```
##single line comment

 /#multi line

Comment#/
```

نمادها و عملوندها

==	Equal
!=	Not Equal
<=	Less or Equal
<	Less than
>	Bigger than
>=	Bigger or equal
=	Assignment
!	Not
~	Bitwise Negation
&	Arithmetic And
&&	Logical and
|	Arithmetic Or
||	Logical Or
^	Logical/Arithmetic Xor
*	Production
+	Add

++	Increment
--	Decrement
-	Sub and unary Minus
/	Div
%	Mod
{ } 	Opening and Closing Curly Brace
( ) 	Opening and Closing Parenthesis
.	Dot
,	Comma
:	Colon
;	Semi-Colon
[ ] 	Opening and Closing Brace

گرامر زبان

این گرامر، (تقریباً) به فرم BNF نوشته شده است. این فرم برای توصیف زبان‌های برنامه‌نویسی به کار می‌رود. در حقیقت، زبان برنامه‌سازی به کمک یک زبان مستقل از متن توصیف می‌شود و در این فرم با قراردادهایی این زبان را توصیف می‌کنیم.

- تمام علائمی که به فرم (a)هستند، واژه‌های نحوی یا Variable یا همان non-terminalهای گرامر زبان هستند.

تمامی علائمی که درشت نوشته شده‌اند مثل for , terminalها یا پایانه‌های زبان هستند.

اگر علامتی به صورت ([X]) ظاهر شد، به معنای آن است که حضور x در آن عبارت اختیاری است.

عبارات *x یا +x به ترتیب به معنای صفر یا بیشتر و یک یا بیشتر رخداد x هستند. (از قواعد عبارات منظم هم استفاده شده است)

همچنین به کمک | سمت راست قواعد تولید با سمت چپ یکسان از یکدیگر تفکیک شده‌اند.

از نماد {} صرفاً جهت گروه‌بندی عبارات استفاده شده است.

هر جا که از نمادهای کمکی بالا در زبان استفاده شده، از Single Quotation استفاده شده است.

<program> → [<ft_dcl>] <ft_def>*

<ft_dcl> → declare '{' <func_dcl> + <type_dcl> + <globl_var> '}'

<func_dcl> → '{' <func_prot> ':' '}'

<func_prot> → '(' [<args>] ')' id '(' [<params>] ')'

<globl_var> → '{' [<type> id;]* '}'

<args> → <type>, <args> | <type>

<params> → <type> id, <params> | <type> id

<type_dcl> → '{' id ',' '}'

<ft_def> → (<func_def> + <type_def>)*

<type_def> → type id '{' (<field> ',')+ '}'

<field> → <type> id;

<func_def> → function id ':' <inout> '{' <block> '}'

<inout> → [<input_list>] [<output_list>]

<input_list> → input <params>

<output_list> → output <params>

<block> → '{' { <var_dcl> | <statement> }* '}'

<type> → int | bool | float | long | char | double | id | string | <type> '['] '

<var_dcl> → [const] <type> <var_dcl_cnt> [, <var_dcl_cnt>]*;

<var_dcl_cnt> → <variable> [= {<expr> | allocate}]

<statement> → <assignment>;

| <func_call>;

| <cond_stmt>

| <loop_stmt>

| return;

| <goto>

| <label>

| <expr>;

| break;

| continue;

| destruct [{'["']'}*] id;

| sizeof(<type>);

<assignment> → <variable> = <expr>

| <variable> = new

| '(' <variable> [.<variable>]* ')' = <func_call>

<variable> → id [{'[' <expr> ']'}+] [.<variable>]

| --<variable>

| ++<variable>

| --<variable>

| ++<variable>

<func_call> → id '(' [<parameters>] ')'

<parameters> → <variable>

| <variable>, <parameters>

<cond_stmt> → if '(' <expr> ')' <block> [else <block>]

| switch (id) of: '{' [{<case> int_const: <block>}*] default: <block> '}'

<loop_stmt> → for '(' [<var_dcl>] ; <expr> ; [<assignment> | <expr>] ')' <block>

<goto> → goto id

<label> → id:

<expr> → <expr> <binary_op> <expr>

| '(' <expr> ')'

| <func_call>

| <variable>

| <const_val>

| -<expr>

| !<expr>

<binary_op> → <arithmatic> | <conditional>

<arithmatic> → + | - | * | / | % | & | '|' | ^ | '||' | &&

<conditional> → == | != | >= | <= | < | >

<const_val> → int_const | real_const | char_const | bool_const | string_const | long_const

موفق باشید